Founding Tenets of Software Design

Get ahead of the curve!
Know what MIT, Stanford Engineers know.

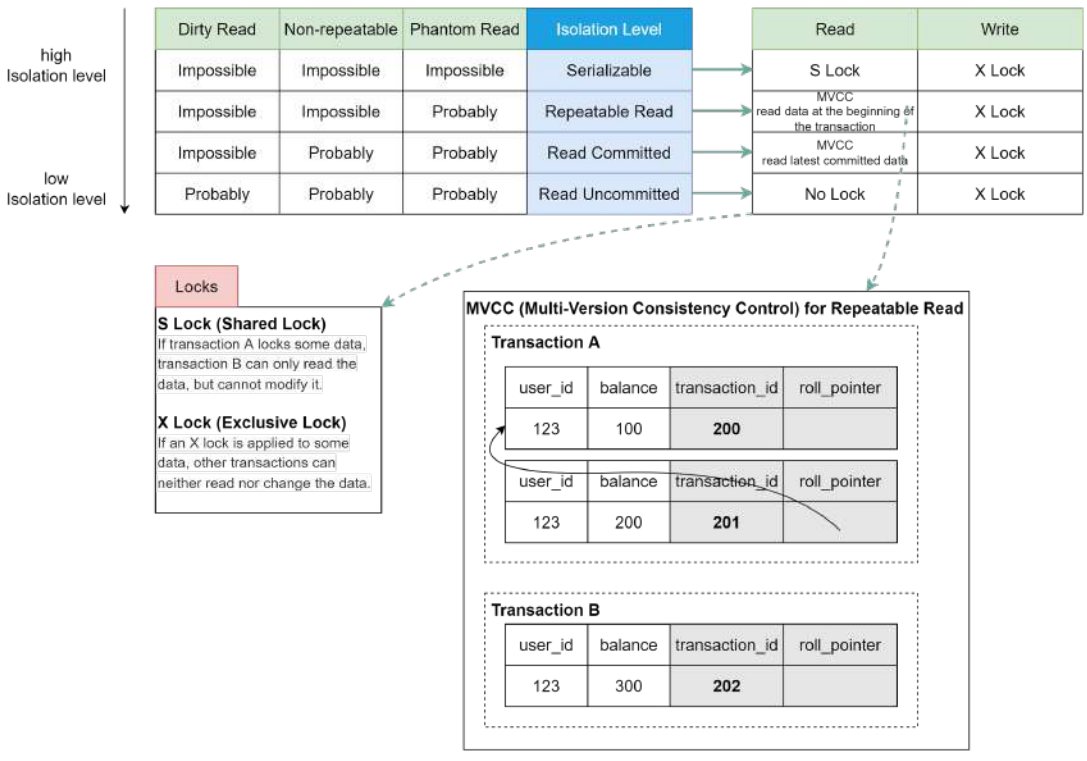
System Design

What are database isolation levels? What are they used for?

Database isolation allows a transaction to execute as if there are no other concurrently running transactions.

The diagram below illustrates four isolation levels.

> Database Isolation Level Illustrated



🔹 Serializalble: This is the highest isolation level. Concurrent transactions are guaranteed to be executed in sequence.

🔹 Repeatable Read: Data read during the transaction stays the same as the transaction starts.

🔹 Read Committed: Data modification can only be read after the transaction is committed.

🔹 Read Uncommitted: The data modification can be read by other transactions before a transaction is committed.

The isolation is guaranteed by MVCC (Multi-Version Consistency Control) and locks.

The diagram below takes Repeatable Read as an example to demonstrate how MVCC works:

There are two hidden columns for each row: transaction_id and roll_pointer. When transaction A starts, a new Read View with transaction_id=201 is created. Shortly afterward, transaction B starts, and a new Read View with transaction_id=202 is created.

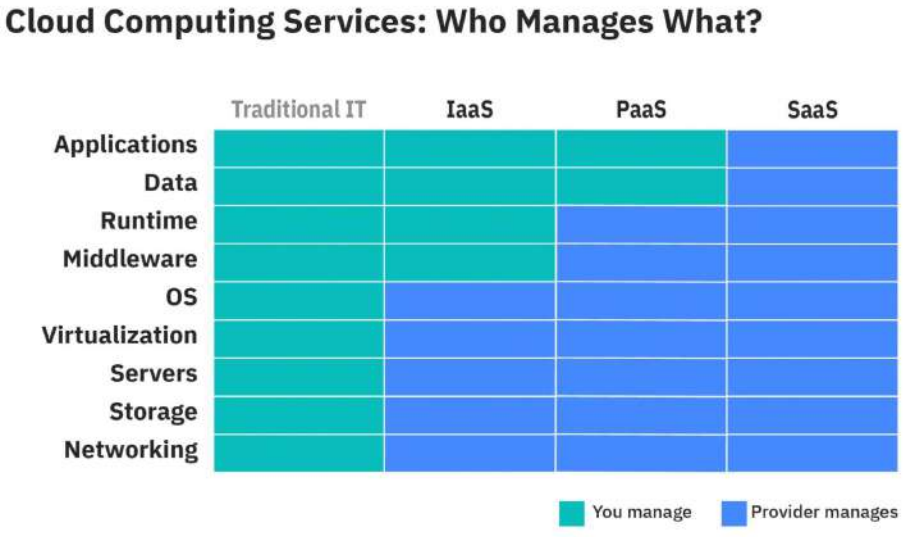
Now transaction A modifies the balance to 200, a new row of the log is created, and the roll_pointer points to the old row. Before transaction A commits, transaction B reads the balance data. Transaction B finds that transaction_id 201 is not committed, it reads the next committed record(transaction_id=200).

Even when transaction A commits, transaction B still reads data based on the Read View created when transaction B starts. So transaction B always reads the data with balance=100.

Over to you: have you seen isolation levels used in the wrong way? Did it cause serious outages?

What is IaaS/PaaS/SaaS?

The diagram below illustrates the differences between IaaS (Infrastructure-as-a-Service), PaaS (Platform-as-a-Service), and SaaS (Software-as-a-Service).



For a non-cloud application, we own and manage all the hardware and software. We say the application is on-premises.

With cloud computing, cloud service vendors provide three kinds of models for us to use: IaaS, PaaS, and SaaS.

IaaS provides us access to cloud vendors' infrastructure, like servers, storage, and networking. We pay for the infrastructure service and install and manage supporting software on it for our application.

PaaS goes further. It provides a platform with a variety of middleware, frameworks, and tools to build our application. We only focus on application development and data.

SaaS enables the application to run in the cloud. We pay a monthly or annual fee to use the SaaS product.

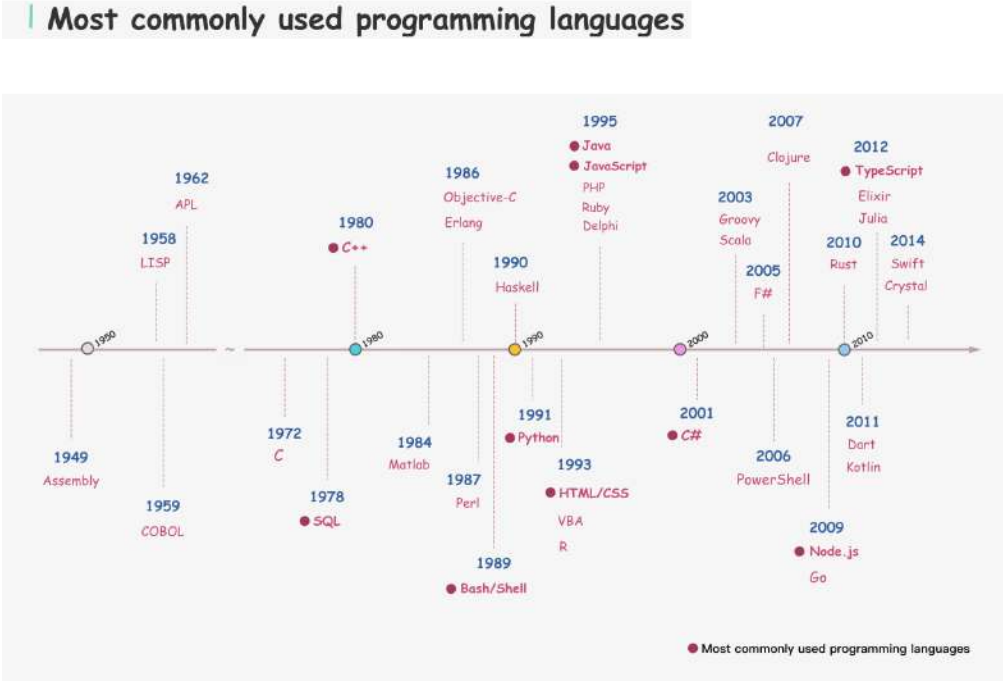
Over to you: which IaaS/PaaS/SaaS products have you used? How do you decide which architecture to use?

Image Source: https://www.ibm.com/cloud/learn/iaas-paas-saas

Most popular programming languages

Programming languages come and go. Some stand the test of time. Some already are shooting stars and some are rising rapidly on the horizon.

I draw a diagram by putting the top 38 most commonly used programming languages in one place, sorted by year. Data source: StackOverflow survey.



1 JavaScript
2 HTML/CSS
3 Python
4 SQL
5 Java
6 Node
7 TypeScript
8 C
9 Bash/Shell
10 C
11 PHP

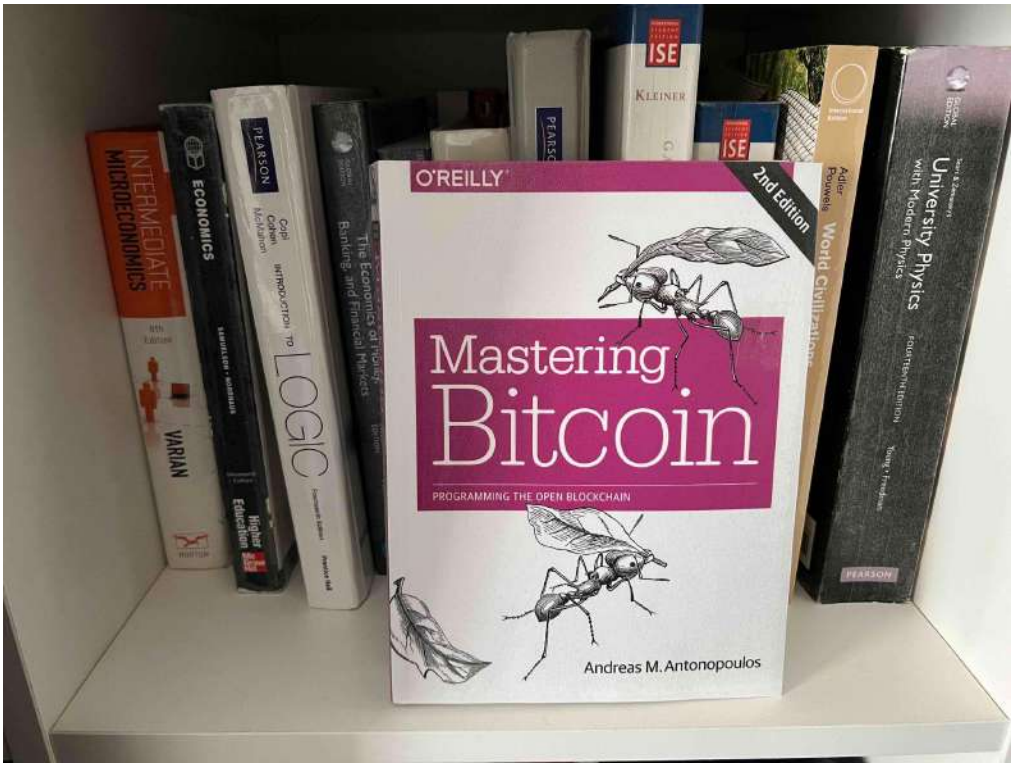
12 C
13 PowerShell
14 Go
15 Kotlin
16 Rust
17 Ruby
18 Dart
19 Assembly
20 Swift
21 R
22 VBA
23 Matlab
24 Groovy
25 Objective-C
26 Scala
27 Perl
28 Haskell
29 Delphi
30 Clojure
31 Elixir
32 LISP
33 Julia
34 F
35 Erlang
36 APL
37 Crystal
38 COBOL

Over to you: what's the first programming language you learned? And what are the other languages you learned over the years?

What is the future of online payments?

I don’t know the answer, but I do know one of the candidates is the blockchain.

As a fan of technology, I always seek new solutions to old challenges. A book that explains a lot about an emerging payment system is ‘Mastering Bitcoin’ by Andreas M. Antonopoulos. I want to share my discovery of this book with you because it explains very clearly bitcoin and its underlying blockchain. This book makes me rethink how to renovate payment systems.



Here are the takeaways:

1. The bitcoin wallet balance is calculated on the fly, while the traditional wallet balance is stored in the database.

2. The golden source of truth for bitcoin is the blockchain, which is also the journal. It's the same if we use Event Sourcing architecture to build a traditional wallet, although there are other options.

3. There is a small virtual machine for bitcoin - and also Ethereum. The virtual machine defines a set of bytecodes to do basic tasks such as validation.

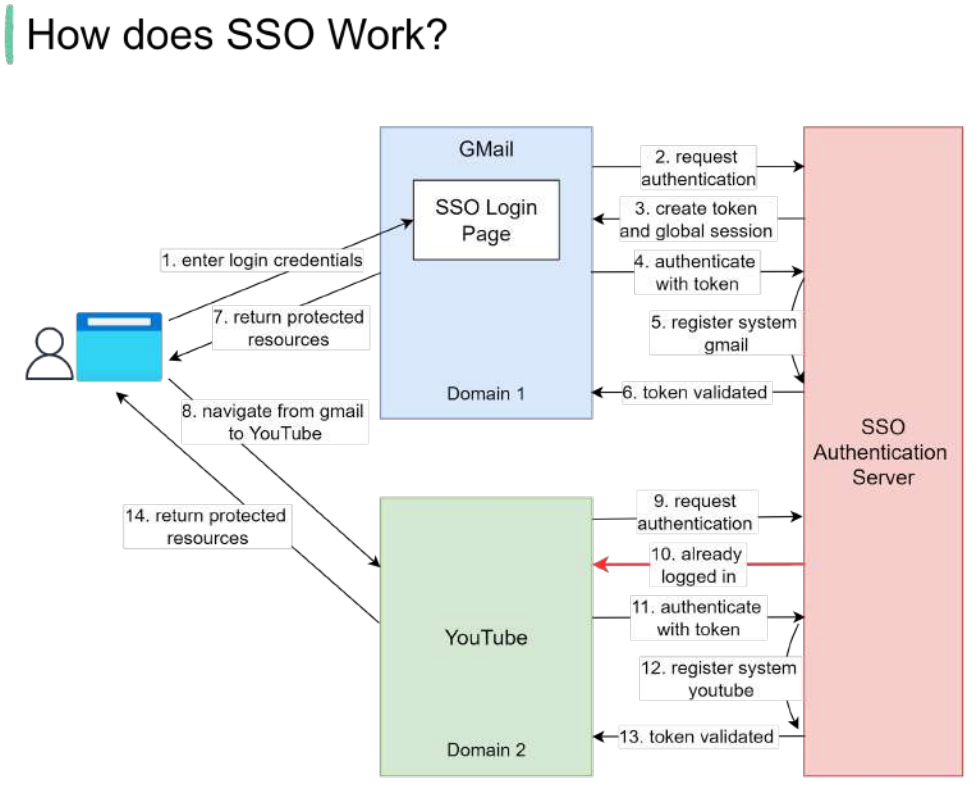
Over to you: if Elon Musk set up a base on planet Mars, what payment solution will you recommend?

What is SSO (Single Sign-On)?

A friend recently went through the irksome experience of being signed out from a number of websites they use daily. This event will be familiar to millions of web users, and it is a tedious process to fix. It can involve trying to remember multiple long-forgotten passwords, or typing in the names of pets from childhood to answer security questions. SSO removes this inconvenience and makes life online better. But how does it work?

Basically, Single Sign-On (SSO) is an authentication scheme. It allows a user to log in to different systems using a single ID.

The diagram below illustrates how SSO works.



Step 1: A user visits Gmail, or any email service. Gmail finds the user is not logged in and so redirects them to the SSO authentication server, which also finds the user is not logged in. As a result, the user

is redirected to the SSO login page, where they enter their login credentials.

Steps 2-3: The SSO authentication server validates the credentials, creates the global session for the user, and creates a token.

Steps 4-7: Gmail validates the token in the SSO authentication server. The authentication server registers the Gmail system, and returns “valid.” Gmail returns the protected resource to the user.

Step 8: From Gmail, the user navigates to another Google-owned website, for example, YouTube.

Steps 9-10: YouTube finds the user is not logged in, and then requests authentication. The SSO authentication server finds the user is already logged in and returns the token.

Step 11-14: YouTube validates the token in the SSO authentication server. The authentication server registers the YouTube system, and returns “valid.” YouTube returns the protected resource to the user.

The process is complete and the user gets back access to their account.

Over to you:

Question 1: have you implemented SSO in your projects? What is the most difficult part?

Question 2: what’s your favorite sign-in method and why?

How to store passwords safely in the database?

Let’s take a look.

Things NOT to do

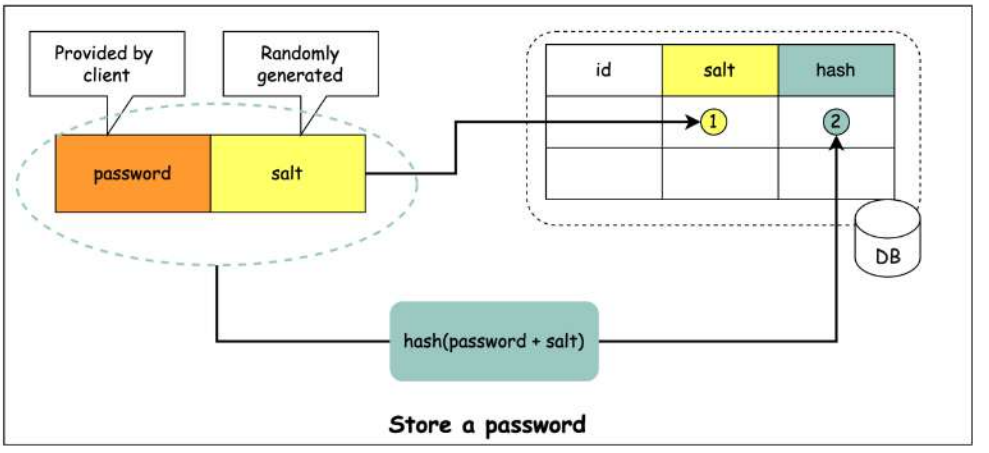
- Storing passwords in plain text is not a good idea because anyone with internal access can see them.

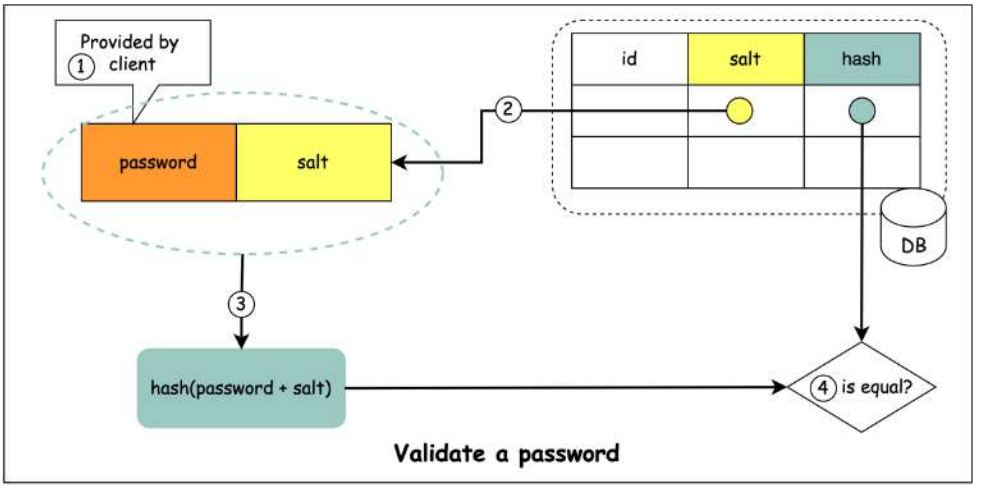
- Storing password hashes directly is not sufficient because it is pruned to precomputation attacks, such as rainbow tables.

- 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 passwords in DB?





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.

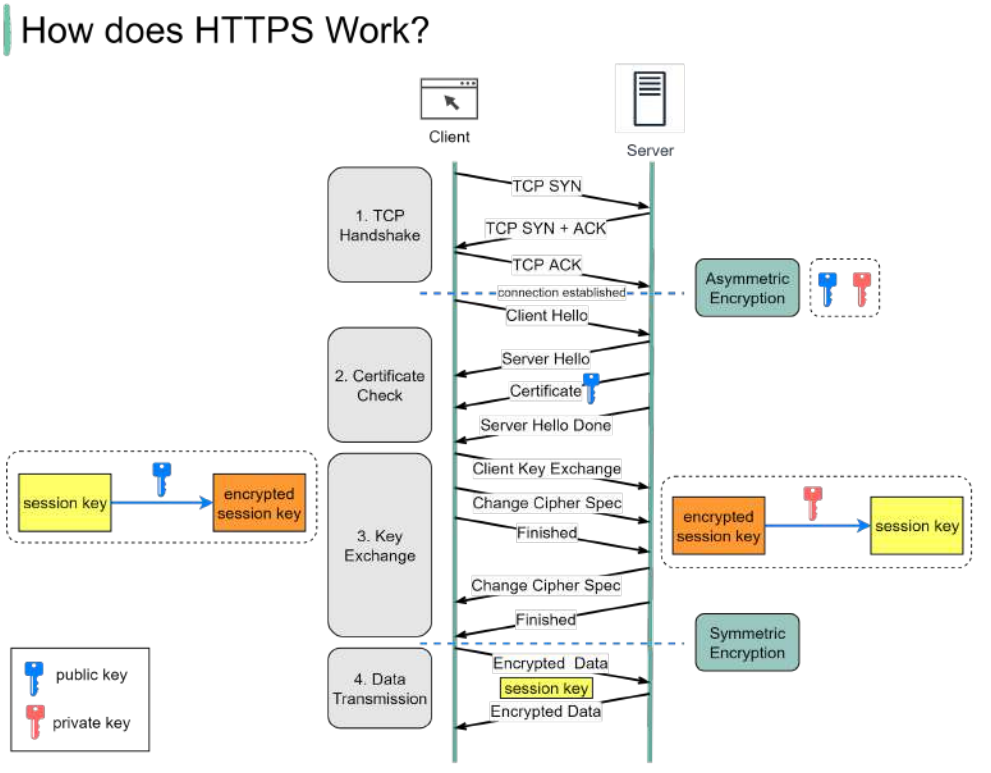
2 The system fetches the corresponding salt from the database.

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?

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.

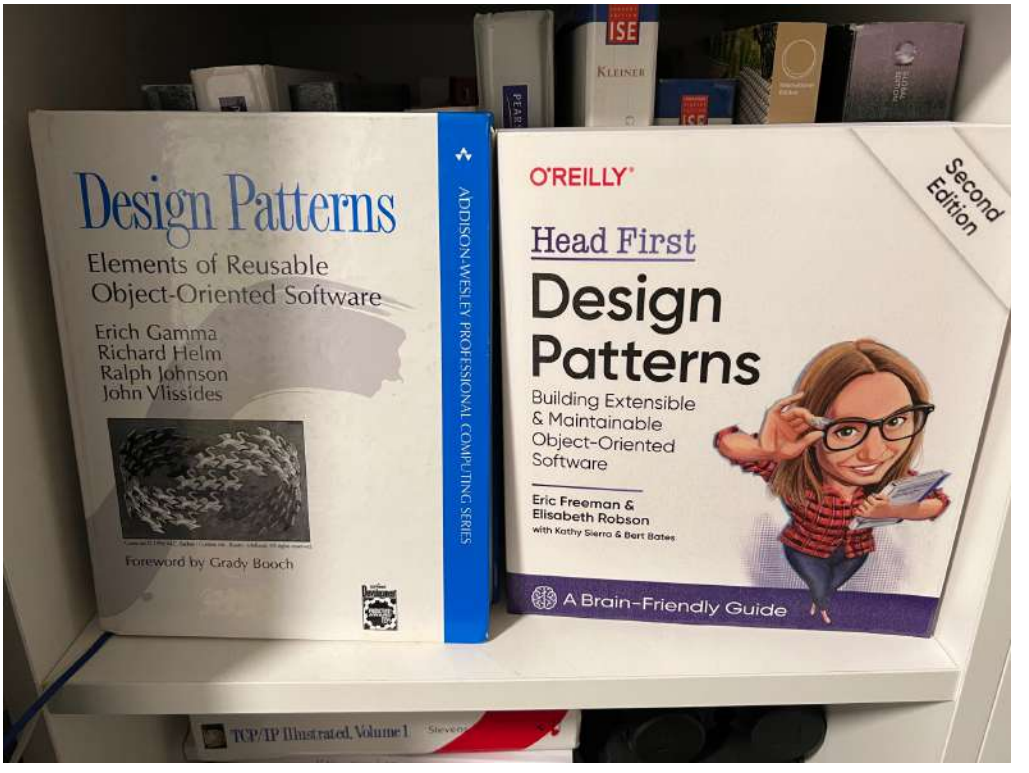
2. Server resources: The asymmetric encryption adds quite a lot of mathematical overhead. It is not suitable for data transmissions in long sessions.

Over to you: how much performance overhead does HTTPS add, compared to HTTP?

How to learn design patterns?

Besides reading a lot of well-written code, a good book guides us like a good teacher.

Head First Design Patterns, second edition, is the one I would recommend.



When I began my journey in software engineering, I found it hard to understand the classic textbook, **Design Patterns,** by the Gang of Four. Luckily, I discovered Head First Design Patterns in the school library. This book solved a lot of puzzles for me. When I went back to the Design Patterns book, everything looked familiar and more understandable.

Last year, I bought the second edition of Head First Design Patterns and read through it. Here are a few things I like about the book:

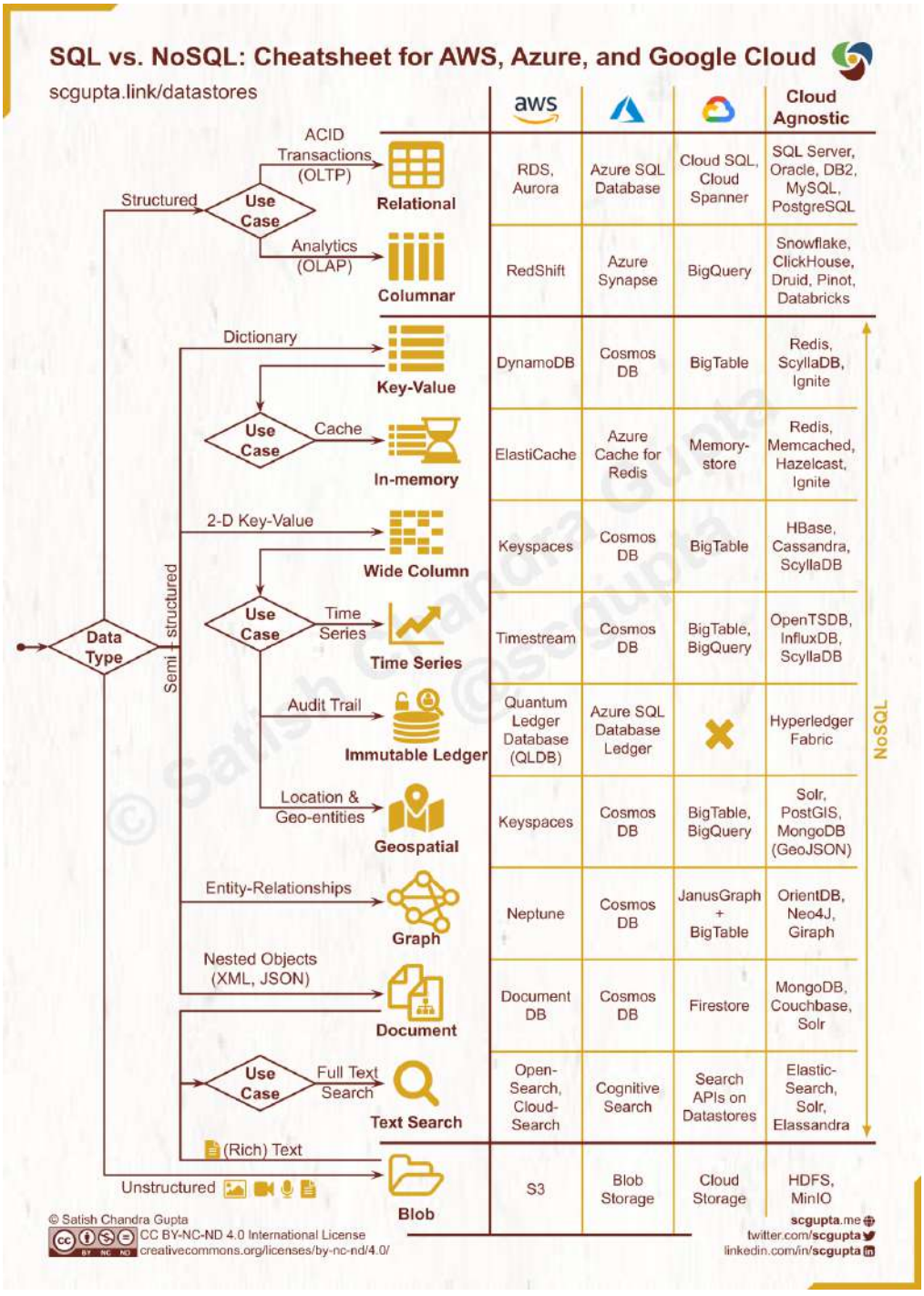
🔹 This book solves the challenge of software's abstract, "invisible" nature. Software is difficult to build because we cannot see its architecture; its details are embedded in the code and binary files. It is even harder to understand software design patterns because these are higher-level abstractions of the software. The book fixes this by using visualization. There are lots of diagrams, arrows, and comments on almost every page. If I do not understand the text, it's no problem. The diagrams explain things very well.

🔹 We all have questions we are afraid to ask when we first learn a new skill. Maybe we think it's an easy one. This book is good at tackling design patterns from the student's point of view. It guides us by asking our questions and clearly answering them. There is a Guru in the book and there's also a Student.

Over to you: which book helped you understand a challenging topic? Why do you like it?

A visual guide on how to choose the right Database

Picking a database is a long-term commitment so the decision shouldn't be made lightly. The important thing to keep in mind is to choose the right database for the right job.



Data can be structured (SQL table schema), semi-structured (JSON, XML, etc.), and unstructured (Blob).

Common database categories include:

- Relational
- Columnar
- Key-value
- In-memory
- Wide column
- Time Series
- Immutable ledger
- Geospatial
- Graph
- Document
- Text search
- Blob

Thanks, Satish Chandra Gupta

Over to you - Which database have you used for which workload?

Do you know how to generate globally unique IDs?

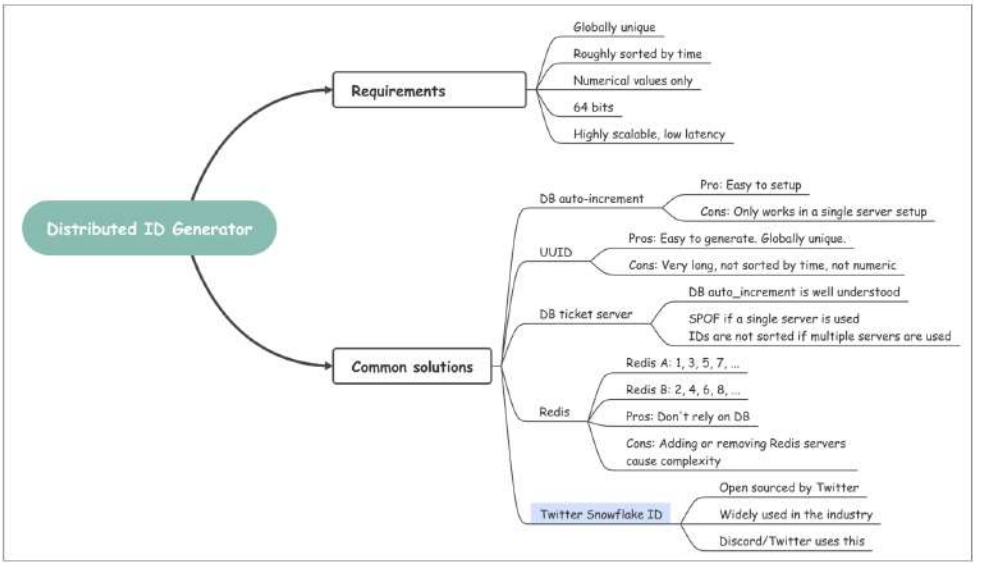
In this post, we will explore common requirements for IDs that are used in social media such as Facebook, Twitter, and LinkedIn.

Requirements:

- Globally unique
- Roughly sorted by time
- Numerical values only
- 64 bits
- Highly scalable, low latency

Unique ID Generator

	Reasons
Globally unique	If IDs are not globally unique, there could be collisions.
Roughly sorted by time	So user IDs, post IDs can be sorted by time without fetching additional info
Numerical values only	Naturally sortable by time
64 bits	2^32 = ~4 billion -> not enough IDs. 2^64 is big enough 2^128 wastes space and is too long
Highly scalable, low latency	Ability to generate a lot of IDs per second in low latency fashion is critical.

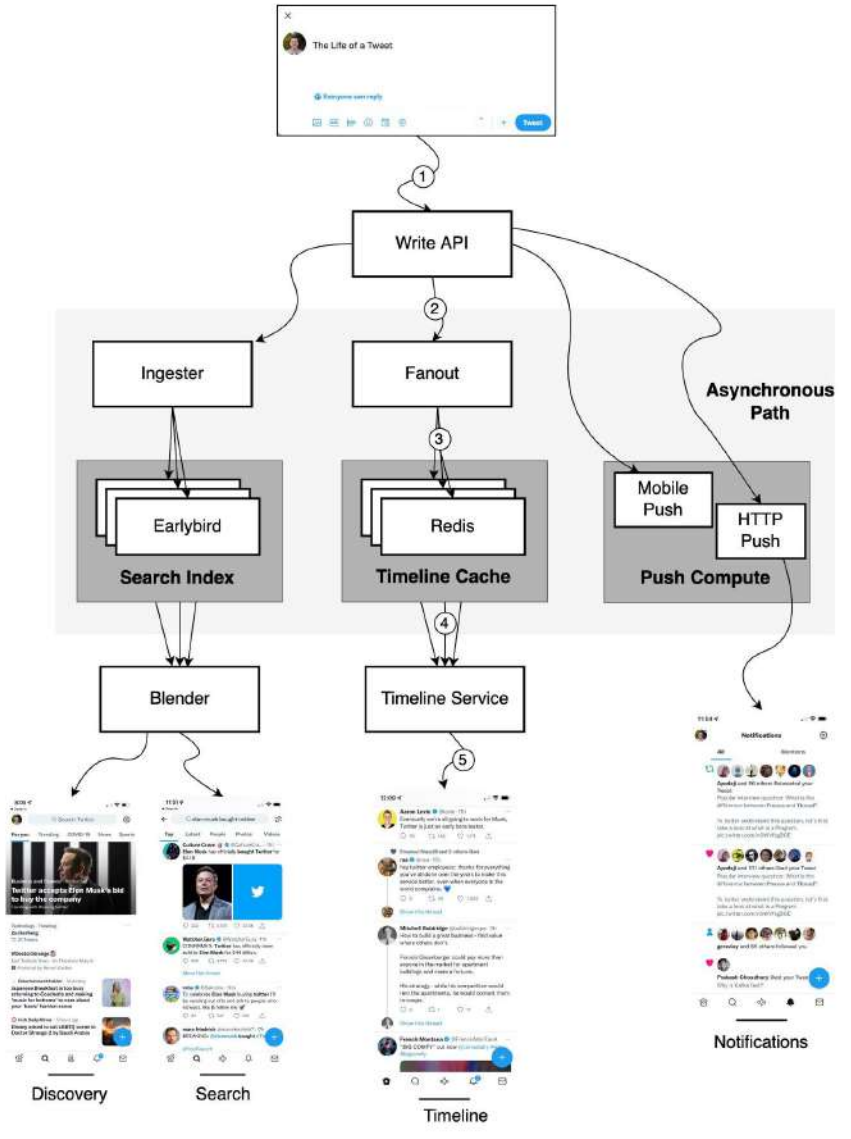


The implementation details of the algorithms can be found online so we will not go into detail here.

Over to you: What kind of ID generators have you used?

How does Twitter work?

This post is a summary of a tech talk given by Twitter in 2013. Let's take a look.



The Life of a Tweet:

1 A tweet comes in through the Write API.

2 The Write API routes the request to the Fanout service.

3 The Fanout service does a lot of processing and stores them in the Redis cache.

4️⃣ The Timeline service is used to find the Redis server that has the home timeline on it.
5️⃣ A user pulls their home timeline through the Timeline service.

Search & Discovery

- Ingester: annotates and tokenizes Tweets so the data can be indexed.
- Earlybird: stores search index.
- Blender: creates the search and discovery timelines.

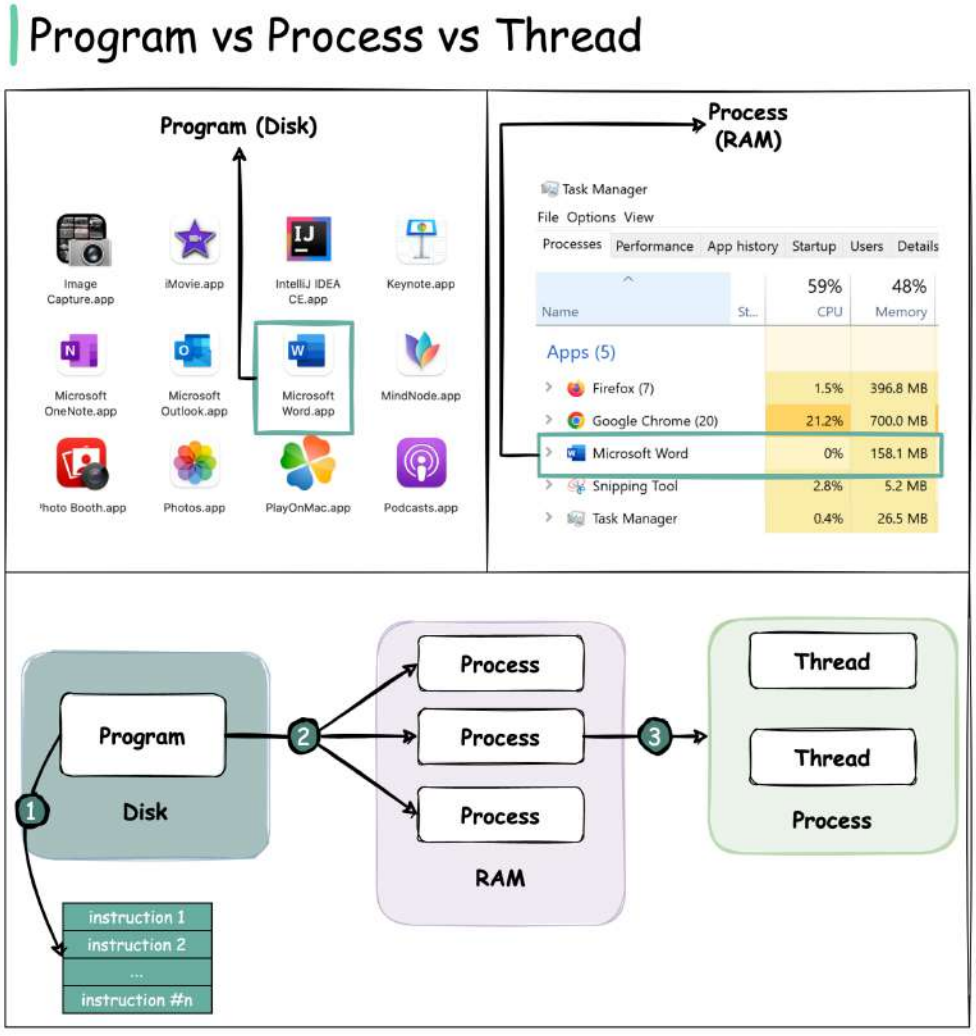
Push Compute

- HTTP push
- Mobile push

Disclaimer: This article is based on the tech talk given by Twitter in 2013 (https://bit.ly/3vNfjRp). Even though many years have passed, it's still quite relevant. I redraw the diagram as the original diagram is difficult to read.

Over to you:
Do you use Twitter? What are some of the biggest differences between LinkedIn and Twitter that might shape their system architectures?

What is the difference between Process and Thread?



To better understand this question, let's first take a look at what is a Program. 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.

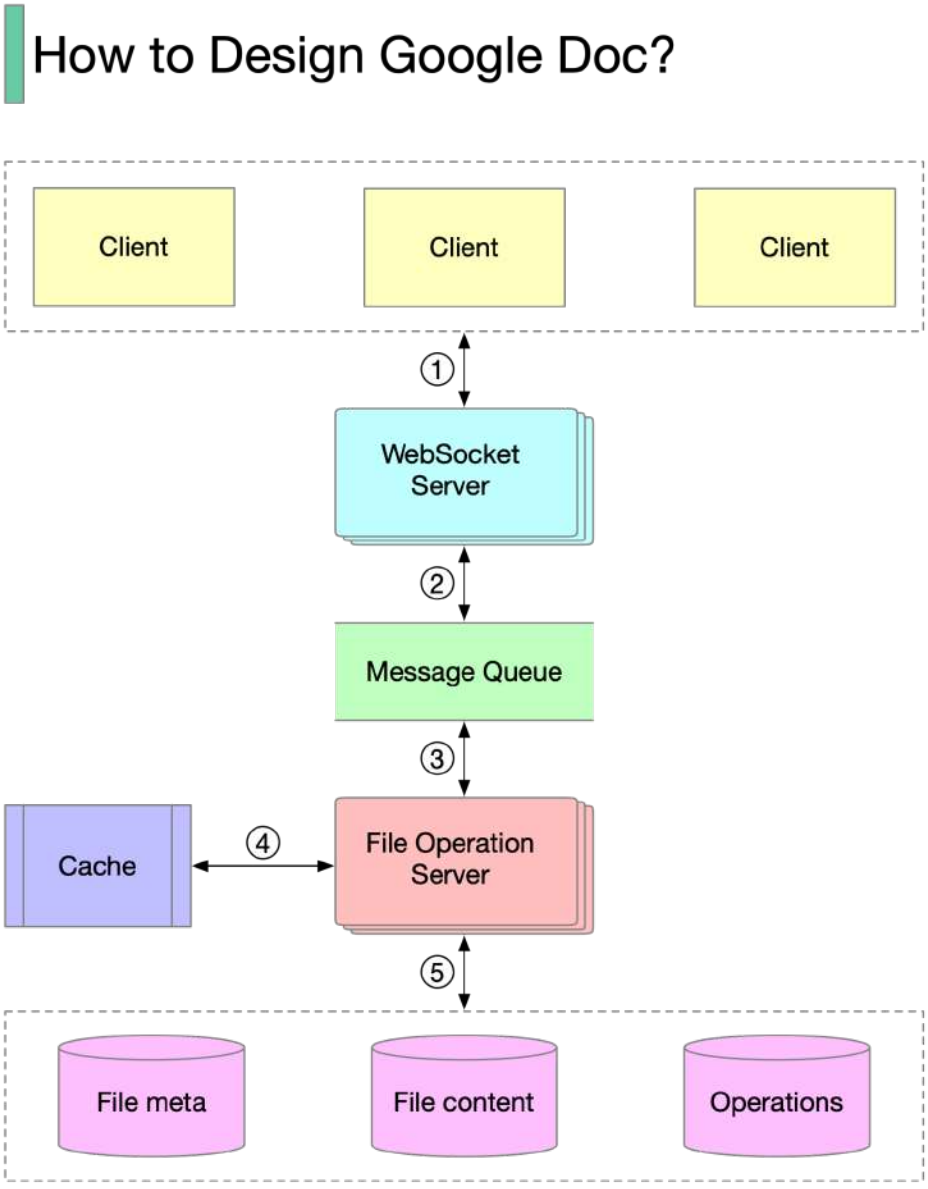
Main differences between process and thread:

- 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?

Interview Question: design Google Docs



1 Clients send document editing operations to the WebSocket Server.
2 The real-time communication is handled by the WebSocket Server.
3 Documents operations are persisted in the Message Queue.

4️⃣ The File Operation Server consumes operations produced by clients and generates transformed operations using collaboration algorithms.
5️⃣ Three types of data are stored: file metadata, file content, and operations.

One of the biggest challenges is real-time conflict resolution. Common algorithms include:

- 🔹 Operational transformation (OT)
- 🔹 Differential Synchronization (DS)
- 🔹 Conflict-free replicated data type (CRDT)

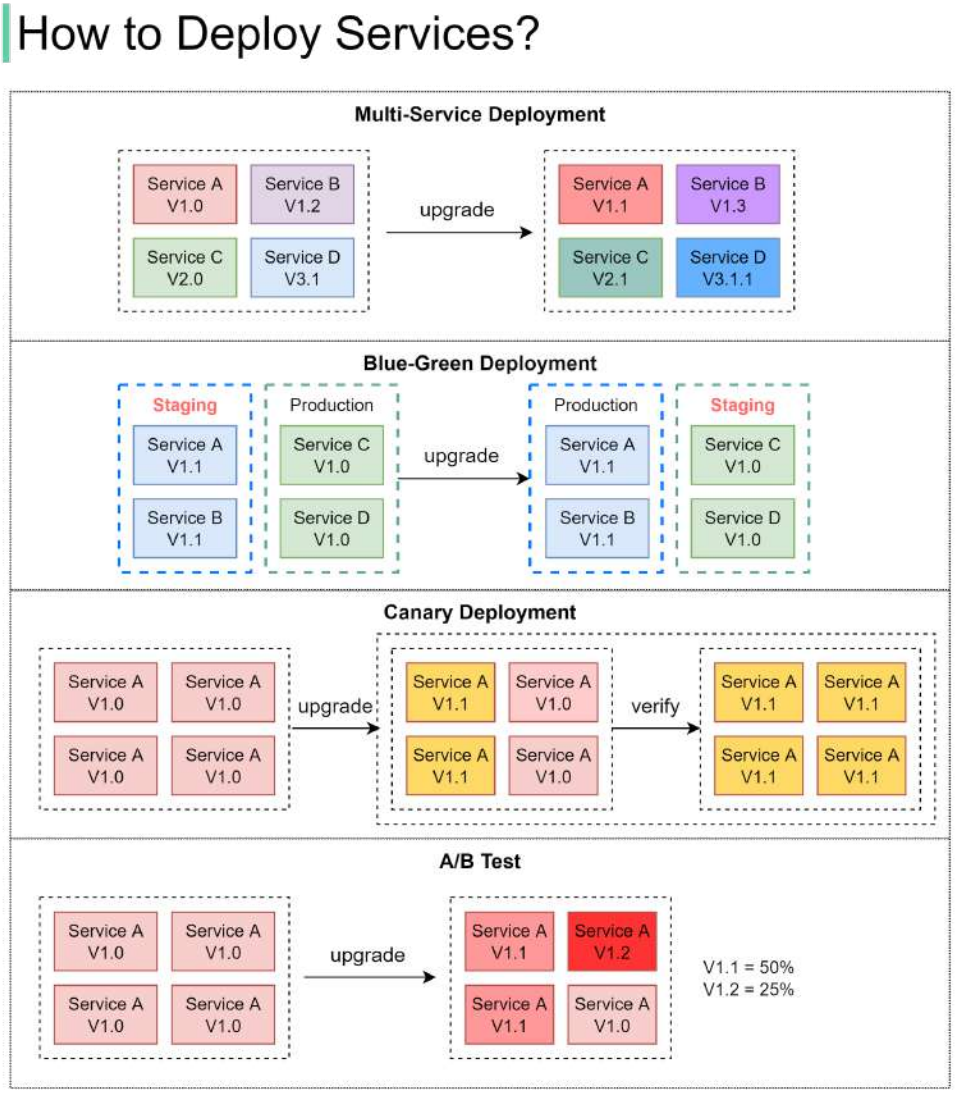
Google Doc uses OT according to its Wikipedia page and CRDT is an active area of research for real-time concurrent editing.

Over to you - Have you encountered any issues while using Google Docs? If so, what do you think might have caused the issue?

Deployment strategies

Deploying or upgrading services is risky. In this post, we explore risk mitigation strategies.

The diagram below illustrates the common ones.



Multi-Service Deployment

In this model, we deploy new changes to multiple services simultaneously. This approach is easy to implement. But since all the services are upgraded at the same time, it is hard to manage and test dependencies. It's also hard to rollback safely.

Blue-Green Deployment
With blue-green deployment, we have two identical environments: one is staging (blue) and the other is production (green). The staging environment is one version ahead of production. Once testing is done in the staging environment, user traffic is switched to the staging environment, and the staging becomes the production. This deployment strategy is simple to perform rollback, but having two identical production quality environments could be expensive.

Canary Deployment
A canary deployment upgrades services gradually, each time to a subset of users. It is cheaper than blue-green deployment and easy to perform rollback. However, since there is no staging environment, we have to test on production. This process is more complicated because we need to monitor the canary while gradually migrating more and more users away from the old version.

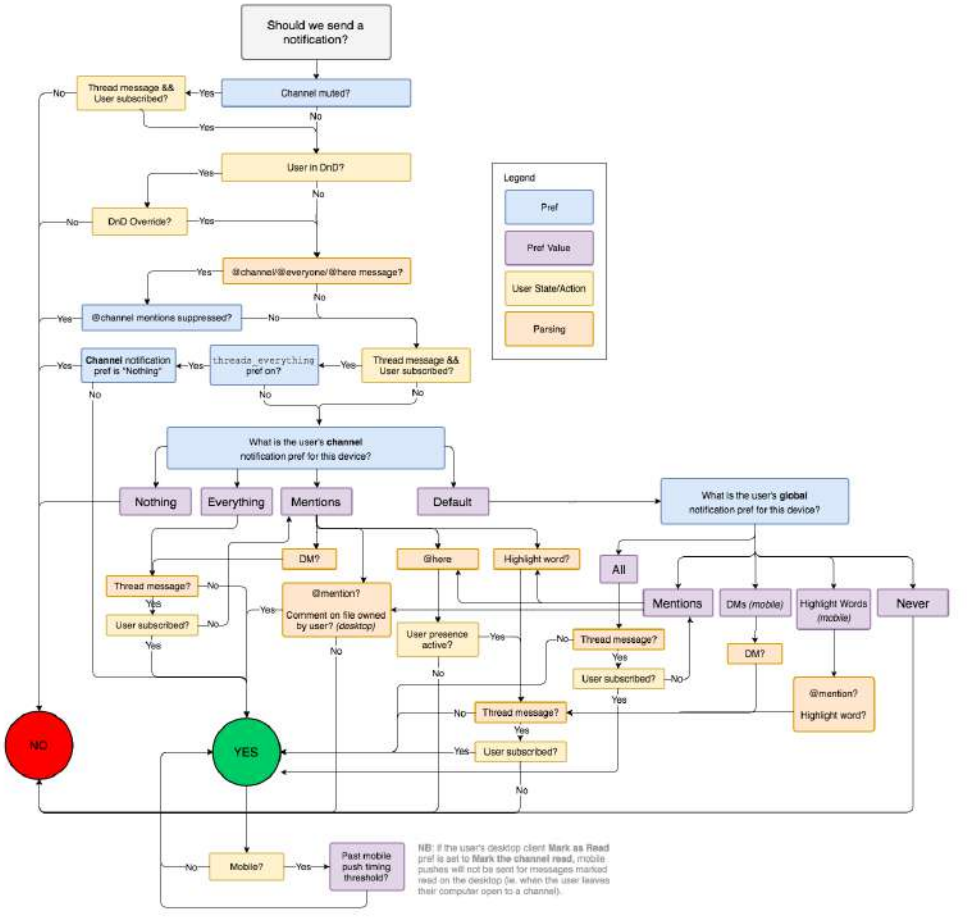
A/B Test
In the A/B test, different versions of services run in production simultaneously. Each version runs an “experiment” for a subset of users. A/B test is a cheap method to test new features in production. We need to control the deployment process in case some features are pushed to users by accident.

Over to you - Which deployment strategy have you used? Did you witness any deployment-related outages in production and why did they happen?

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.

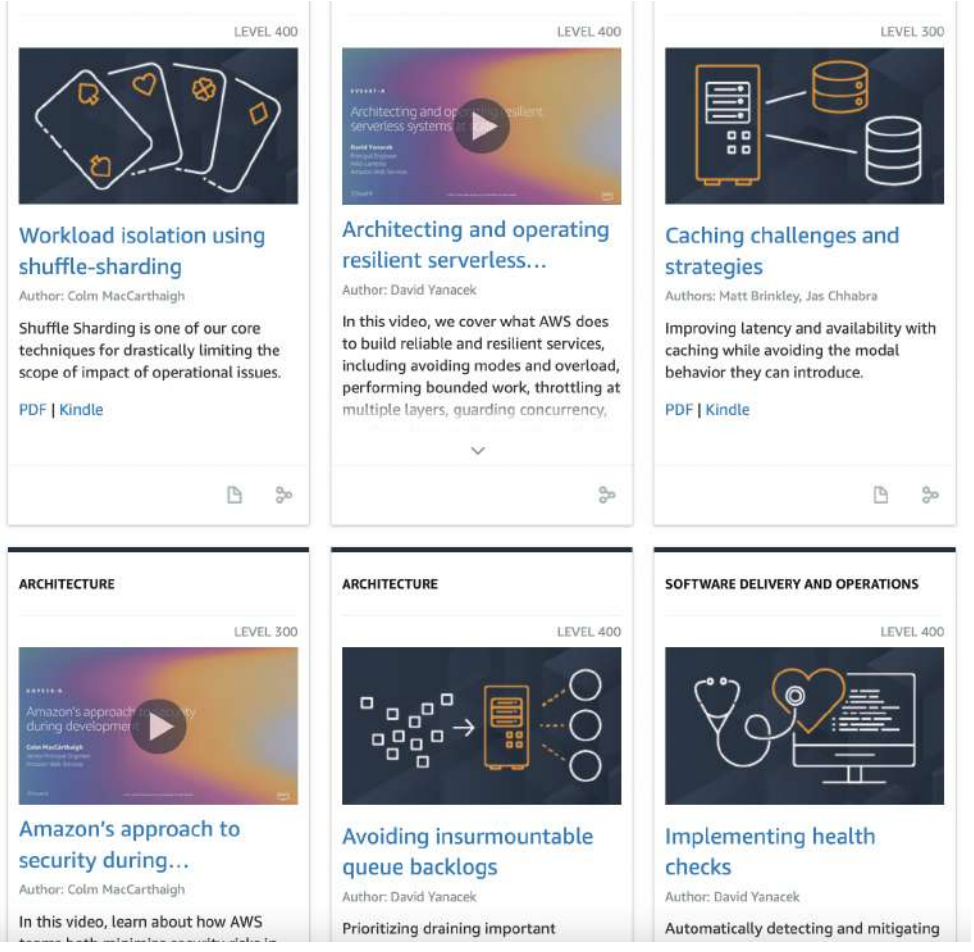


What's your takeaway from this diagram?

Image source:
https://slack.engineering/reducing-slacks-memory-footprint/

How does Amazon build and operate the software?

In 2019, Amazon released The Amazon Builders' Library. It contains architecture-based articles that describe how Amazon architects, releases, and operates technology.



As of today, it published 26 articles. It took me two weekends to go through all the articles. I've had great fun and learned a lot. Here are some of my favorites:

- Making retries safe with idempotent APIs
- Timeouts, retries, and backoff with jitter
- Beyond five 9s: Lessons from our highest available data planes
- Caching challenges and strategies
- Ensuring rollback safety during deployments
- Going faster with continuous delivery

- Challenges with distributed systems
- Amazon's approach to high-availability deployment

Over to you: what’s your favorite place to learn system design and design principles?

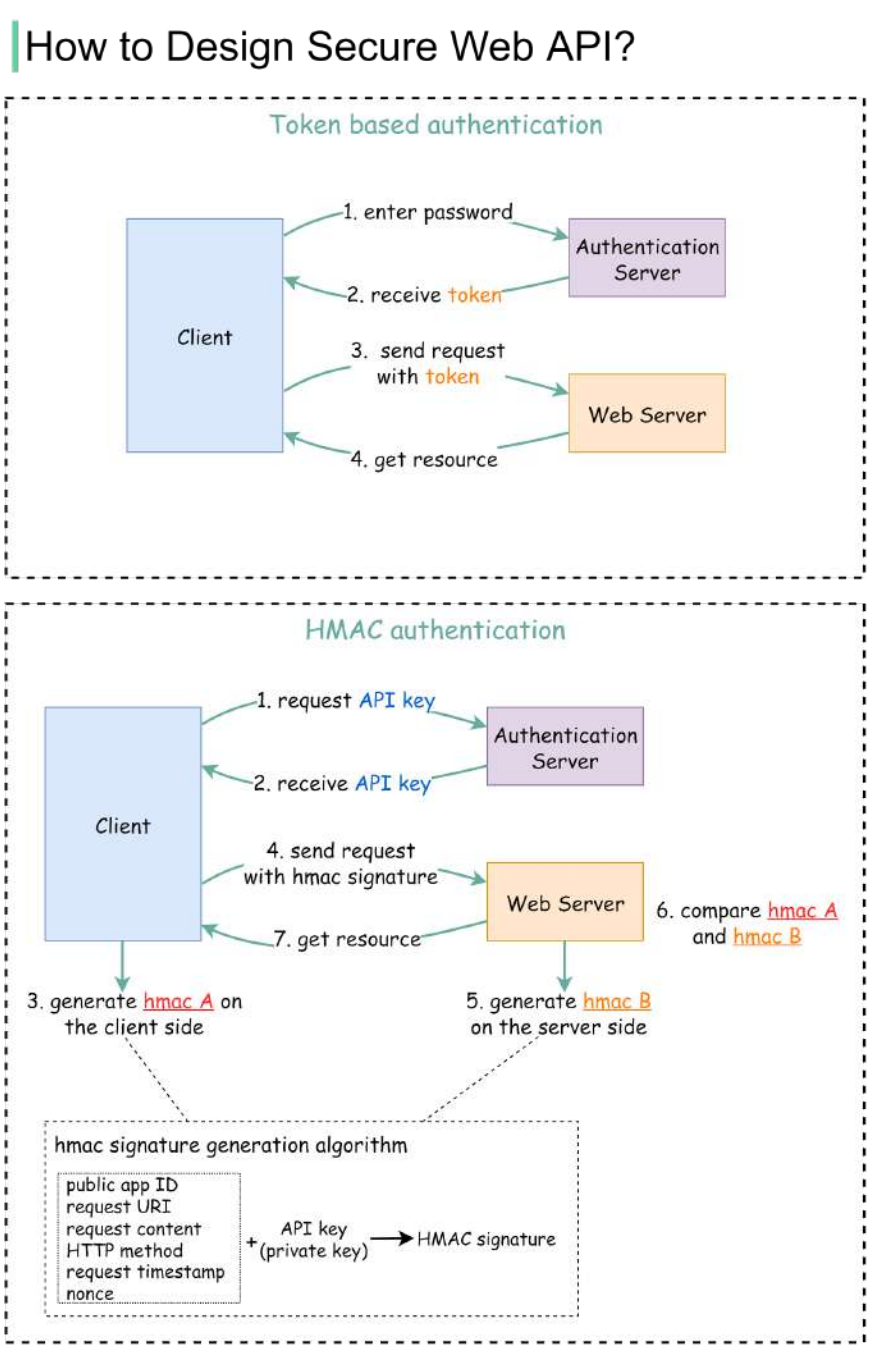
Link to The Amazon Builders' Library: aws.amazon.com/builders-library

How to design a secure web API access for your website?

When we open web API access to users, we need to make sure each API call is authenticated. This means the user must be who they claim to be.

In this post, we explore two common ways:
1. Token based authentication
2. HMAC (Hash-based Message Authentication Code) authentication

The diagram below illustrates how they work.



Token based

Step 1 - the user enters their password into the client, and the client sends the password to the Authentication Server.

Step 2 - the Authentication Server authenticates the credentials and generates a token with an expiry time.

Steps 3 and 4 - now the client can send requests to access server resources with the token in the HTTP header. This access is valid until the token expires.

HMAC based

This mechanism generates a Message Authentication Code (signature) by using a hash function (SHA256 or MD5).

Steps 1 and 2 - the server generates two keys, one is Public APP ID (public key) and the other one is API Key (private key).
Step 3 - we now generate a HMAC signature on the client side (hmac A). This signature is generated with a set of attributes listed in the diagram.

Step 4 - the client sends requests to access server resources with hmac A in the HTTP header.

Step 5 - the server receives the request which contains the request data and the authentication header. It extracts the necessary attributes from the request and uses the API key that's stored on the server side to generate a signature (hmac B.)

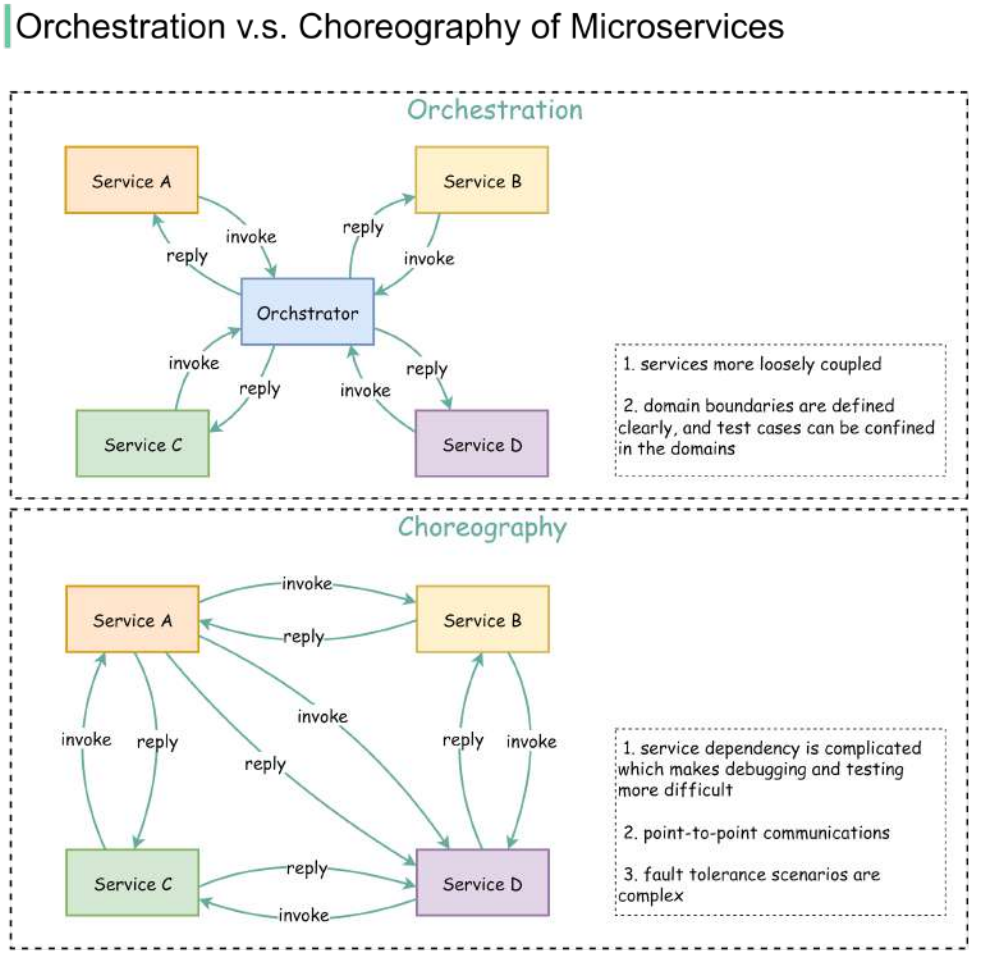
Steps 6 and 7 - the server compares hmac A (generated on the client side) and hmac B (generated on the server side). If they are matched, the requested resource will be returned to the client.

Question - How does HMAC authentication ensure data integrity? Why do we include "request timestamp" in HMAC signature generation?

How do microservices collaborate and interact with each other?

There are two ways: **orchestration** and **choreography**.

The diagram below illustrates the collaboration of microservices.



Choreography is like having a choreographer set all the rules. Then the dancers on stage (the microservices) interact according to them. Service choreography describes this exchange of messages and the rules by which the microservices interact.

Orchestration is different. The orchestrator acts as a center of authority. It is responsible for invoking and combining the services. It

describes the interactions between all the participating services. It is just like a conductor leading the musicians in a musical symphony. The orchestration pattern also includes the transaction management among different services.

The benefits of orchestration:
1. Reliability - orchestration has built-in transaction management and error handling, while choreography is point-to-point communications and the fault tolerance scenarios are much more complicated.
2. Scalability - when adding a new service into orchestration, only the orchestrator needs to modify the interaction rules, while in choreography all the interacting services need to be modified.

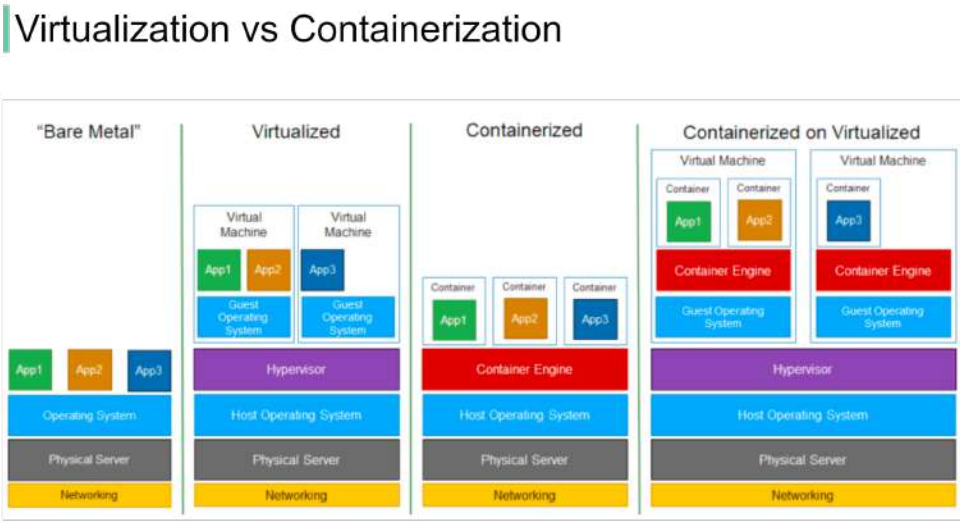
Some limitations of orchestration:
1. Performance - all the services talk via a centralized orchestrator, so latency is higher than it is with choreography. Also, the throughput is bound to the capacity of the orchestrator.
2. Single point of failure - if the orchestrator goes down, no services can talk to each other. To mitigate this, the orchestrator must be highly available.

Real-world use case: Netflix Conductor is a microservice orchestrator and you can read more details on the orchestrator design.

Question - Have you used orchestrator products in production? What are their pros & cons?

What are the differences between Virtualization (VMware) and Containerization (Docker)?

The diagram below illustrates the layered architecture of virtualization and containerization.



“Virtualization is a technology that allows you to create multiple simulated environments or dedicated resources from a single, physical hardware system” [1].

“Containerization is the packaging together of software code with all its necessary components like libraries, frameworks, and other dependencies so that they are isolated in their own "container" [2].

The major differences are:

- In virtualization, the hypervisor creates an abstraction layer over hardware, so that multiple operating systems can run alongside each other. This technique is considered to be the first generation of cloud computing.

- Containerization is considered to be a lightweight version of virtualization, which virtualizes the operating system instead of hardware. Without the hypervisor, the containers enjoy faster resource provisioning. All the resources (including code, dependencies) that are

needed to run the application or microservice are packaged together, so that the applications can run anywhere.

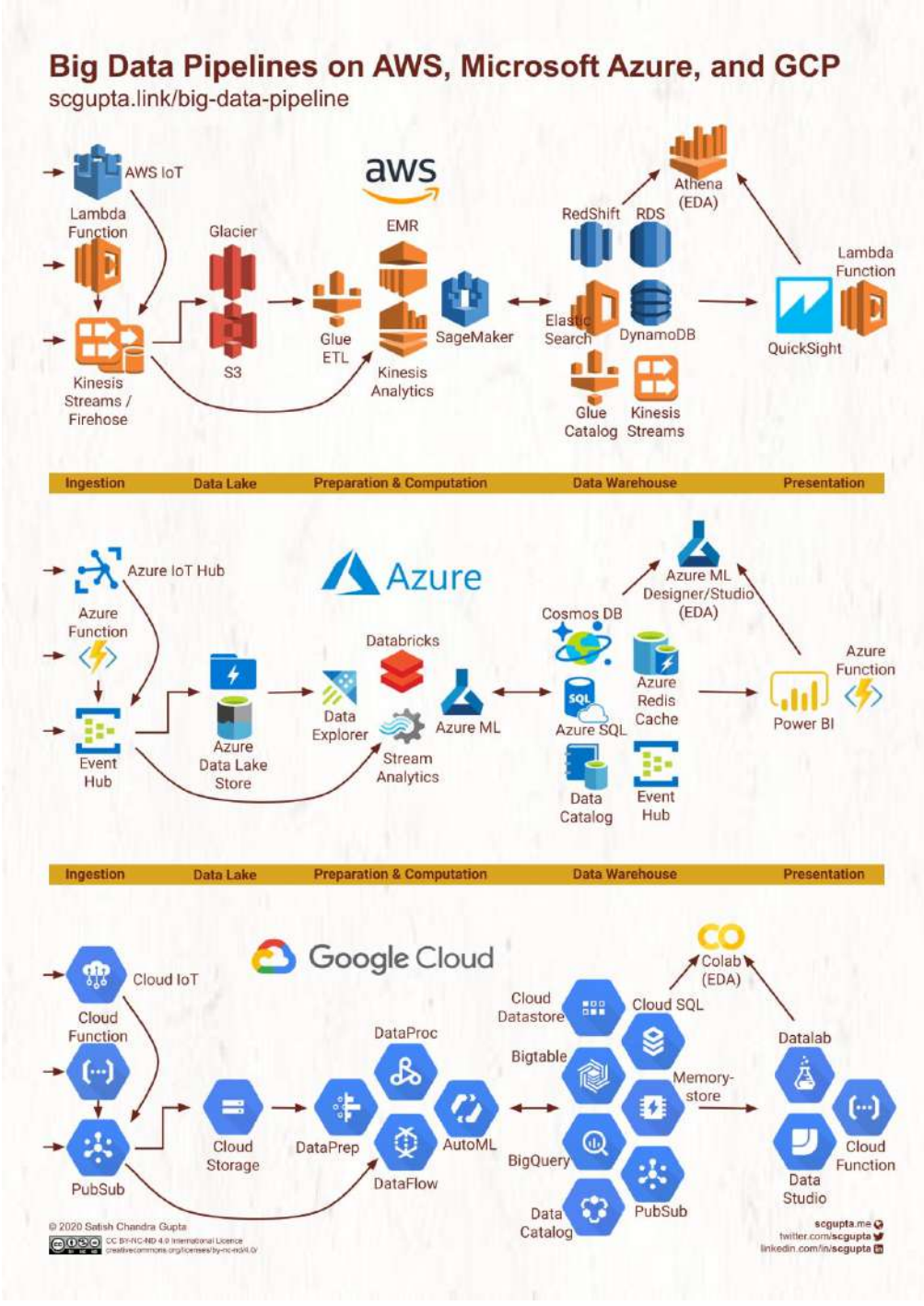
Question: how much performance differences have you observed in production between virtualization, containerization, and bare-metal?

Image Source: https://lnkd.in/gaPYcGTz

Sources:
[1] Understanding virtualization: https://lnkd.in/gtQY9gkx
[2] What is containerization?: https://lnkd.in/gm4Qv_x2

Which cloud provider should be used when building a big data solution?

The diagram below illustrates the detailed comparison of AWS, Google Cloud, and Microsoft Azure.



The common parts of the solutions:

1. Data ingestion of structured or unstructured data.
2. Raw data storage.
3. Data processing, including filtering, transformation, normalization, etc.
4. Data warehouse, including key-value storage, relational database, OLAP database, etc.
5. Presentation layer with dashboards and real-time notifications.

It is interesting to see different cloud vendors have different names for the same type of products.

For example, the first step and the last step both use the serverless product. The product is called “lambda” in AWS, and “function” in Azure and Google Cloud.

Question - which products have you used in production? What kind of application did you use it for?

Source: S.C. Gupta’s post

How to avoid crawling duplicate URLs at Google scale?

Option 1: Use a Set data structure to check if a URL already exists or not. Set is fast, but it is not space-efficient.

Option 2: Store URLs in a database and check if a new URL is in the database. This can work but the load to the database will be very high.

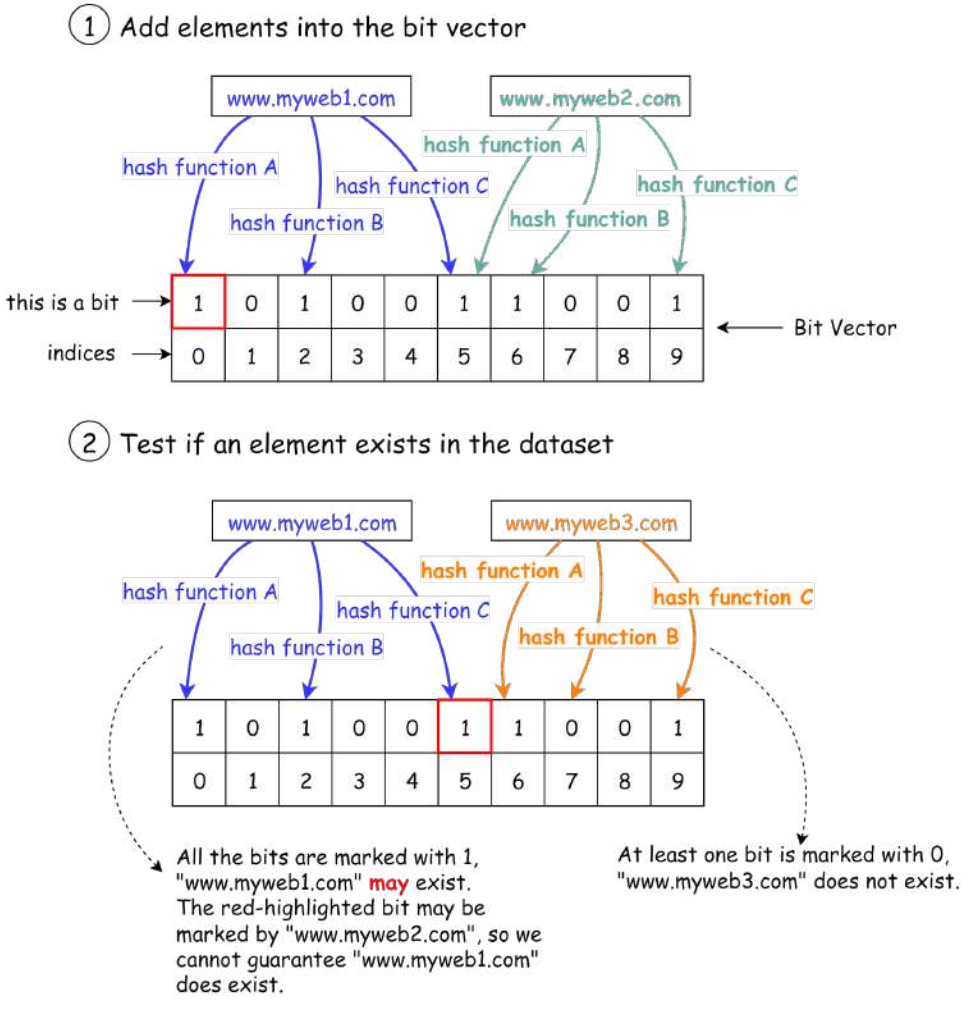
Option 3: **Bloom filter**. This option is preferred. Bloom filter was proposed by Burton Howard Bloom in 1970. It is a probabilistic data structure that is used to test whether an element is a member of a set.

- false: the element is definitely not in the set.
- true: the element is probably in the set.

False-positive matches are possible, but false negatives are not.

The diagram below illustrates how the Bloom filter works. The basic data structure for the Bloom filter is Bit Vector. Each bit represents a hashed value.

How to Dedupe Massive URLs



Step 1: To add an element to the bloom filter, we feed it to 3 different hash functions (A, B, and C) and set the bits at the resulting positions. Note that both “www.myweb1.com” and “www.myweb2.com” mark the same bit with 1 at index 5. False positives are possible because a bit might be set by another element.

Step 2: When testing the existence of a URL string, the same hash functions A, B, and C are applied to the URL string. If all three bits are

1, then the URL may exist in the dataset; if any of the bits is 0, then the URL definitely does not exist in the dataset.

Hash function choices are important. They must be uniformly distributed and fast. For example, RedisBloom and Apache Spark use murmur, and InfluxDB uses xxhash.

Question - In our example, we used three hash functions. How many hash functions should we use in reality? What are the trade-offs?

Why is a solid-state drive (SSD) fast?

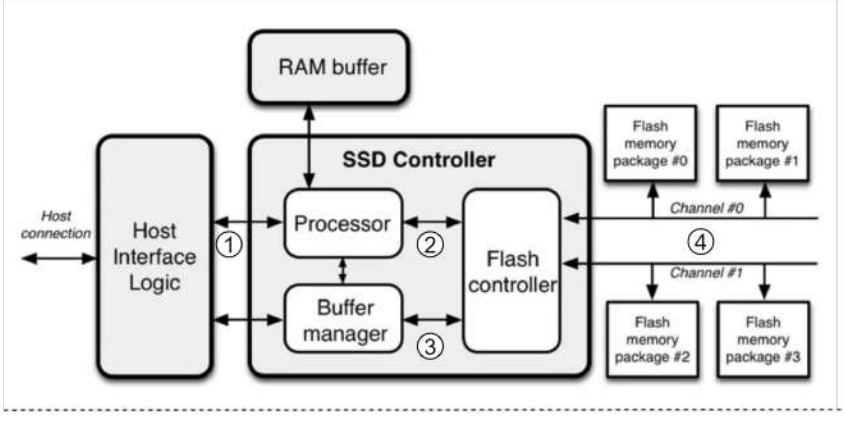
“A solid state drive reads up to 10 times faster and writes up to 20 times faster than a hard disk drive.” [1].

“An SSD is a flash-memory based data storage device. Bits are stored into cells, which are made of floating-gate transistors. SSDs are made entirely of electronic components, there are no moving or mechanical parts like in hard drives (HDD)” [2].

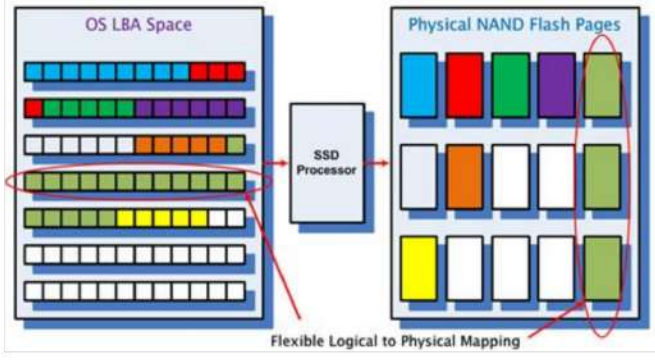
The diagram below illustrates the SSD architecture.

Why is SSD(Solid State Drive) Fast?

SSD Architecture



Mapping of Logical and Physical Pages



Step 1: “Commands come from the user through the host interface” [2]. The interface can be Serial ATA (SATA) or PCI Express (PCIe).
Step 2: “The processor in the SSD controller takes the commands and passes them to the flash controller” [2].
Step 3: “SSDs also have embedded RAM memory, generally for caching purposes and to store mapping information” [2].
Step 4: “The packages of NAND flash memory are organized in gangs, over multiple channels” [2].

The second diagram illustrates how the logical and physical pages are mapped, and why this architecture is fast.

SSD controller operates multiple FLASH particles in parallel, greatly improving the underlying bandwidth. When we need to write more than one page, the SSD controller can write them in parallel [3], whereas the HDD has a single head and it can only read from one head at a time.

Every time a HOST Page is written, the SSD controller finds a Physical Page to write the data and this mapping is recorded. With this mapping, the next time HOST reads a HOST Page, the SSD knows where to read the data from FLASH [3].

Question - What are the main differences between SSD and HDD?

If you are interested in the architecture, I recommend reading Coding for SSDs by Emmanuel Goossaert in reference [2].

Sources:
[1] SSD or HDD: Which Is Right for You?:
https://www.avg.com/en/signal/ssd-hdd-which-is-best
[2] Coding for SSDs:
https://codecapsule.com/2014/02/12/coding-for-ssds-part-1-introduction-and-table-of-contents/
[3] Overview of SSD Structure and Basic Working Principle:
https://www.elinfor.com/knowledge/overview-of-ssd-structure-and-basic-working-principle1-p-11203

Handling a large-scale outage

This is a true story about handling a large-scale outage written by Staff Engineers at Discord Sahn Lam.

About 10 years ago, I witnessed the most impactful UI bugs in my career.

It was 9PM on a Friday. I was on the team responsible for one of the largest social games at the time. It had about 30 million DAU. I just so happened to glance at the operational dashboard before shutting down for the night.

Every line on the dashboard was at zero.

At that very moment, I got a phone call from my boss. He said the entire game was down. Firefighting mode. Full on.

Everything had shut down. Every single instance on AWS was terminated. HA proxy instances, PHP web servers, MySQL databases, Memcache nodes, everything.

It took 50 people 10 hours to bring everything back up. It was quite a feat. That in itself is a story for another day.

We used a cloud management software vendor to manage our AWS deployment. This was before Infrastructure as Code was a thing. There was no Terraform. It was so early in cloud computing and we were so big that AWS required an advanced warning before we scaled up.

What had gone wrong? The software vendor had introduced a bug that week in their confirmation dialog flow. When terminating a subset of nodes in the UI, it would correctly show in the confirmation dialog box the list of nodes to be terminated, but under the hood, it terminated everything.

Shortly before 9PM that fateful evening, one of our poor SREs fulfilled our routine request and terminated an unused Memcache pool. I could only imagine the horror and the phone conversation that ensured.

What kind of code structure could allow this disastrous bug to slip through? We could only guess. We never received a full explanation.

What are some of the most impactful software bugs you encountered in your career?

AWS Lambda behind the scenes

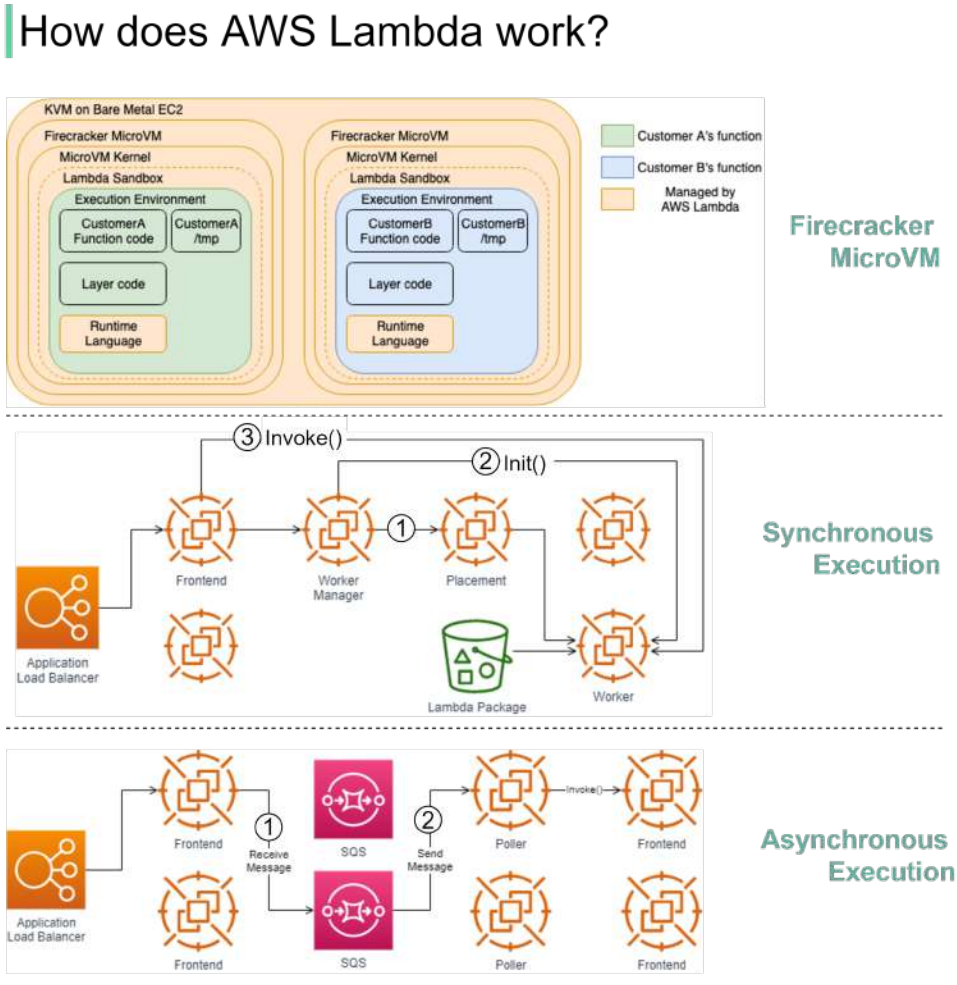
Serverless is one of the hottest topics in cloud services. How does AWS Lambda work behind the scenes?

Lambda is a **serverless** computing service provided by Amazon Web Services (AWS), which runs functions in response to events.

Firecracker MicroVM

Firecracker is the engine powering all of the Lambda functions [1]. It is a virtualization technology developed at Amazon and written in Rust.

The diagram below illustrates the isolation model for AWS Lambda Workers.



Lambda functions run within a sandbox, which provides a minimal Linux userland, some common libraries and utilities. It creates the Execution environment (worker) on EC2 instances.

How are lambdas initiated and invoked? There are two ways.

Synchronous execution
Step1: "The Worker Manager communicates with a Placement Service which is responsible to place a workload on a location for the given host (it's provisioning the sandbox) and returns that to the Worker Manager" [2].

Step 2: "The Worker Manager can then call *Init* to initialize the function for execution by downloading the Lambda package from S3 and setting up the Lambda runtime" [2]

Step 3: The Frontend Worker is now able to call *Invoke* [2].

Asynchronous execution
Step 1: The Application Load Balancer forwards the invocation to an available Frontend which places the event onto an internal queue(SQS).
Step 2: There is "a set of pollers assigned to this internal queue which are responsible for polling it and moving the event onto a Frontend synchronously. After it's been placed onto the Frontend it follows the synchronous invocation call pattern which we covered earlier" [2].

Question: Can you think of any use cases for AWS Lambda?

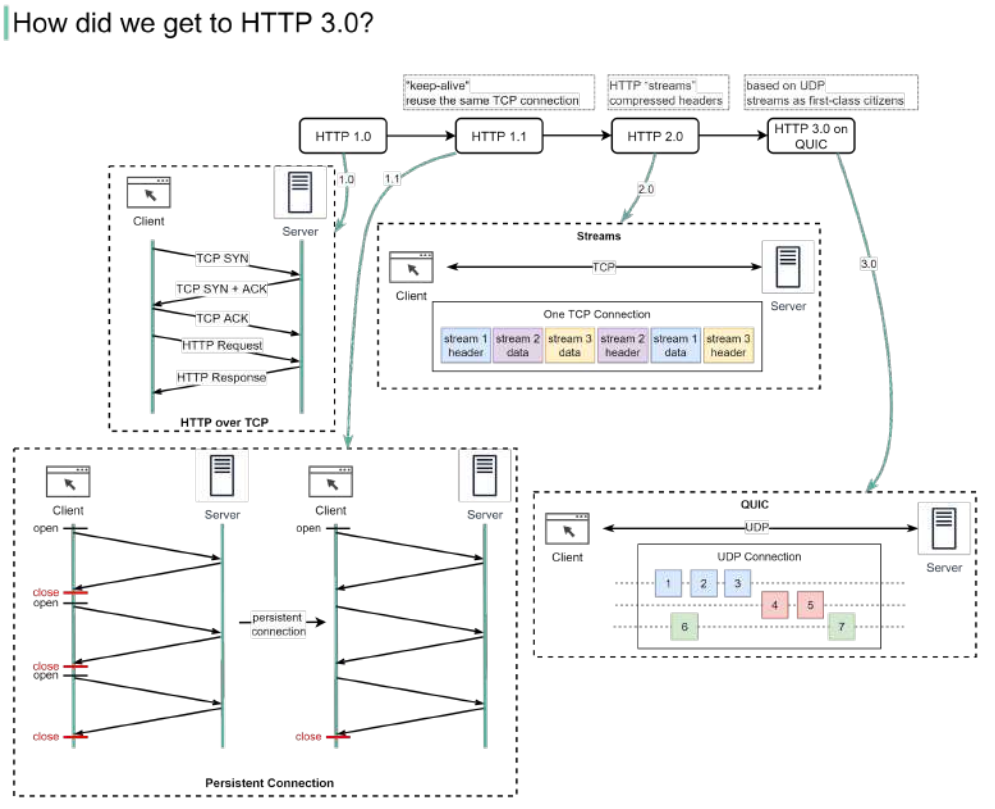
Sources:
[1] AWS Lambda whitepaper:
https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-executions.html
[2] Behind the scenes, Lambda:
https://www.bschaatsbergen.com/behind-the-scenes-lambda/

Image source: [1] [2]

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.

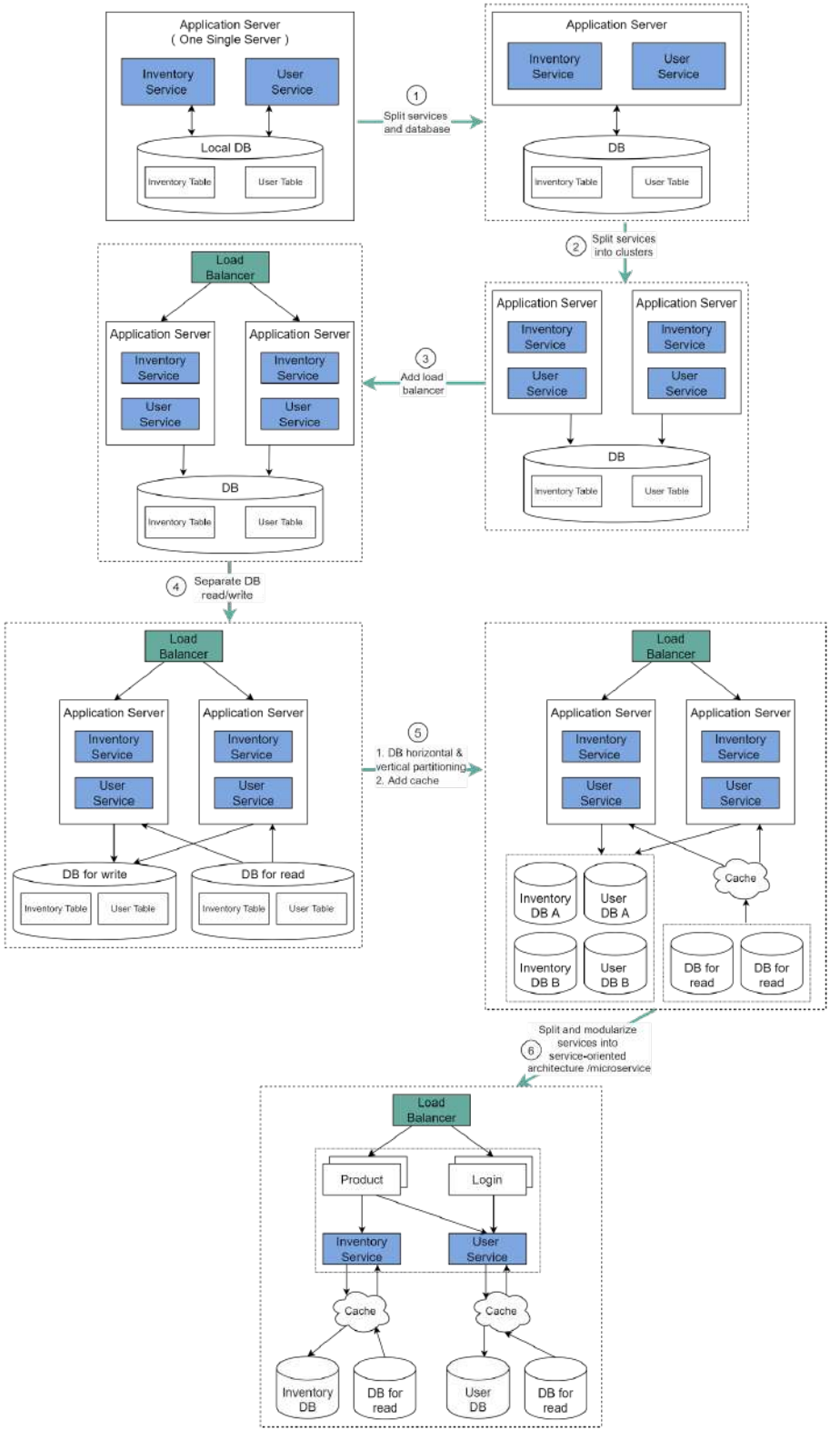
Question: When shall we upgrade to HTTP 3.0? Any pros & cons you can think of?

How to scale a website to support millions of users?

We will explain this step-by-step.

The diagram below illustrates the evolution of a simplified eCommerce website. It goes from a monolithic design on one single server, to a service-oriented/microservice architecture.

How to Scale a Website Step-by-Step?



Suppose we have two services: inventory service (handles product descriptions and inventory management) and user service (handles user information, registration, login, etc.).

Step 1 - With the growth of the user base, one single application server cannot handle the traffic anymore. We put the application server and the database server into two separate servers.

Step 2 - The business continues to grow, and a single application server is no longer enough. So we deploy a cluster of application servers.

Step 3 - Now the incoming requests have to be routed to multiple application servers, how can we ensure each application server gets an even load? The load balancer handles this nicely.

Step 4 - With the business continuing to grow, the database might become the bottleneck. To mitigate this, we separate reads and writes in a way that frequent read queries go to read replicas. With this setup, the throughput for the database writes can be greatly increased.

Step 5 - Suppose the business continues to grow. One single database cannot handle the load on both the inventory table and user table. We have a few options:

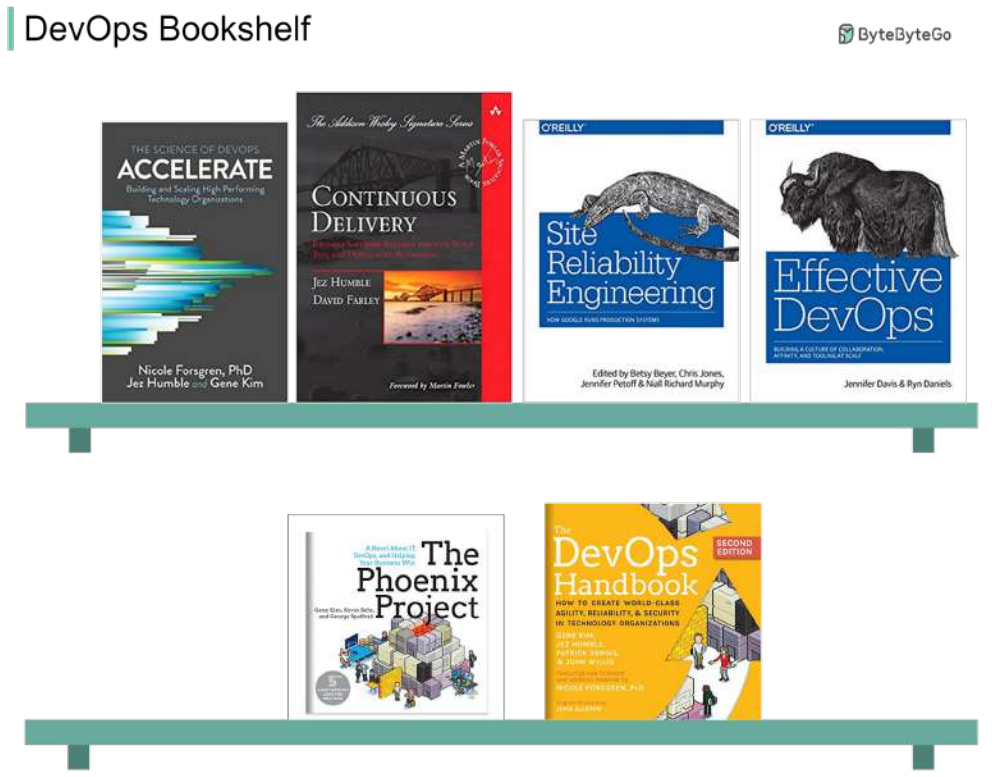
1. Vertical partition. Adding more power (CPU, RAM, etc.) to the database server. It has a hard limit.
2. Horizontal partition by adding more database servers.
3. Adding a caching layer to offload read requests.

Step 6 - Now we can modularize the functions into different services. The architecture becomes service-oriented / microservice.

Question: what else do we need to support an e-commerce website at Amazon’s scale?

DevOps Books

Some **DevOps** books I find enlightening:



🔹 Accelerate - presents both the findings and the science behind measuring software delivery performance.

🔹 Continuous Delivery - introduces automated architecture management and data migration. It also pointed out key problems and optimal solutions in each area.

🔹 Site Reliability Engineering - famous Google SRE book. It explains the whole life cycle of Google's development, deployment, and monitoring, and how to manage the world's biggest software systems.

🔹 Effective DevOps - provides effective ways to improve team coordination.

🔹 The Phoenix Project - a classic novel about effectiveness and communications. IT work is like manufacturing plant work, and a system must be established to streamline the workflow. Very interesting read!

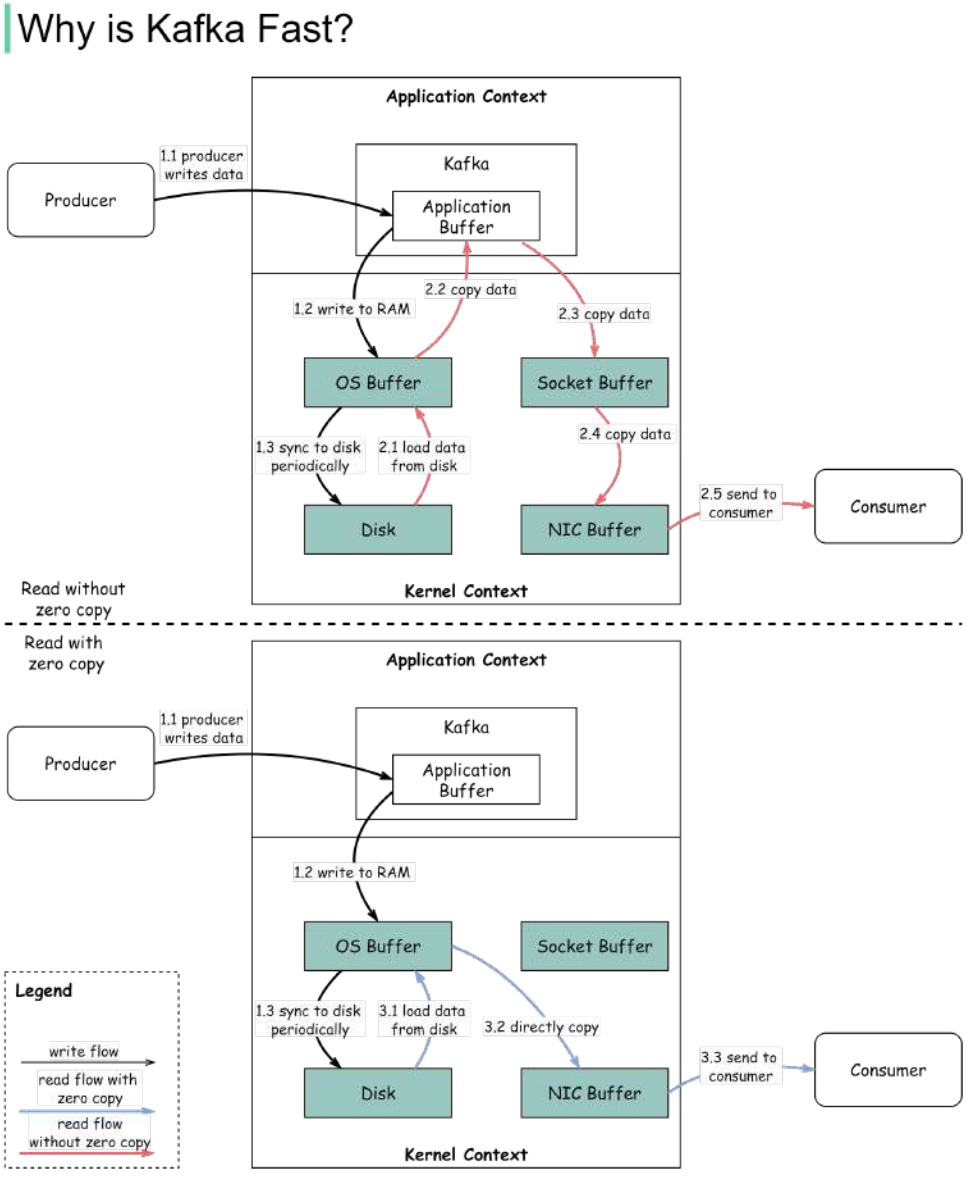
🔹 The DevOps Handbook - introduces product development, quality assurance, IT operations, and information security.

What’s your favorite dev-ops book?

Why is Kafka fast?

Kafka achieves low latency message delivery through Sequential I/O and Zero Copy Principle. The same techniques are commonly used in many other messaging/streaming platforms.

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

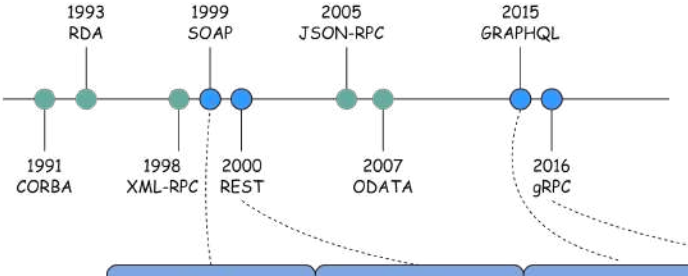
🔹 Step 3: Consumer reads data with zero-copy
3.1: The data is loaded from disk to OS cache
3.2 OS cache directly copies the data to the network card via sendfile() command
3.3 The network card sends data out to the consumer

Zero copy is a shortcut to save the multiple data copies between application context and kernel context. This approach brings down the time by approximately 65%.

SOAP vs REST vs GraphQL vs RPC.

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

API Architectural Styles Comparison



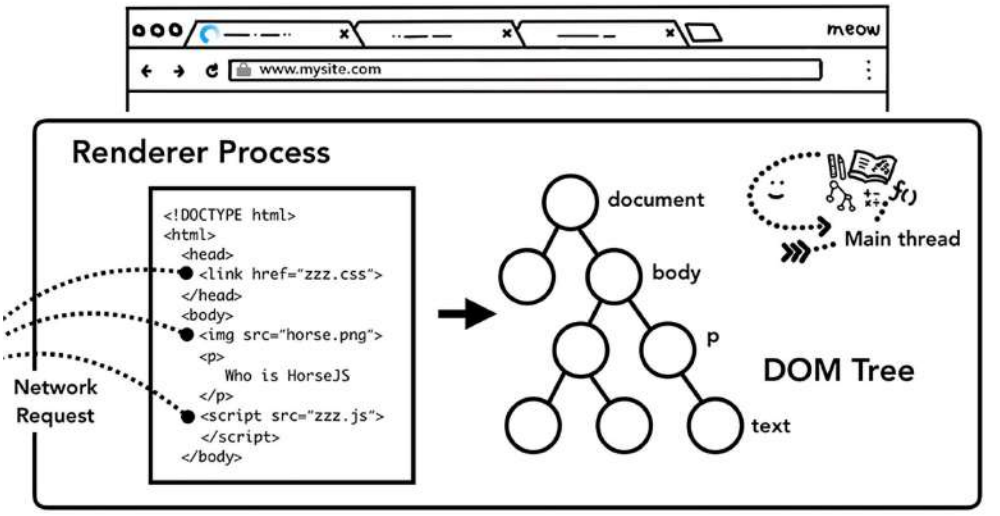
	SOAP (Simple Object Access Protocol)	REST (REpresentational State Transfer)	GraphQL	RPC (Remote Procedure Call)
Organized in terms of	enveloped message structure	compliance with six architectural constraints	schema & type system	local procedure call
Format	XML only	XML, JSON, HTML, plain text	JSON	JSON, XML, Protobuf, Thrift, FlatBuffers
Learning curve	Difficult	Easy	Medium	Easy
Community	Small	Large	Growing	Large
Use cases	- payment gateways - identity management - CRM solutions - financial and telecommunication services - legacy system support	- public APIs - simple resource-driven apps	- mobile APIs - complex systems - micro-services	- command and action-oriented APIs - high performance communication in massive micro-services systems

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.

Source: https://lnkd.in/gFgi33RY I combined a few diagrams together. The credit all goes to AltexSoft.

How do modern browsers work?

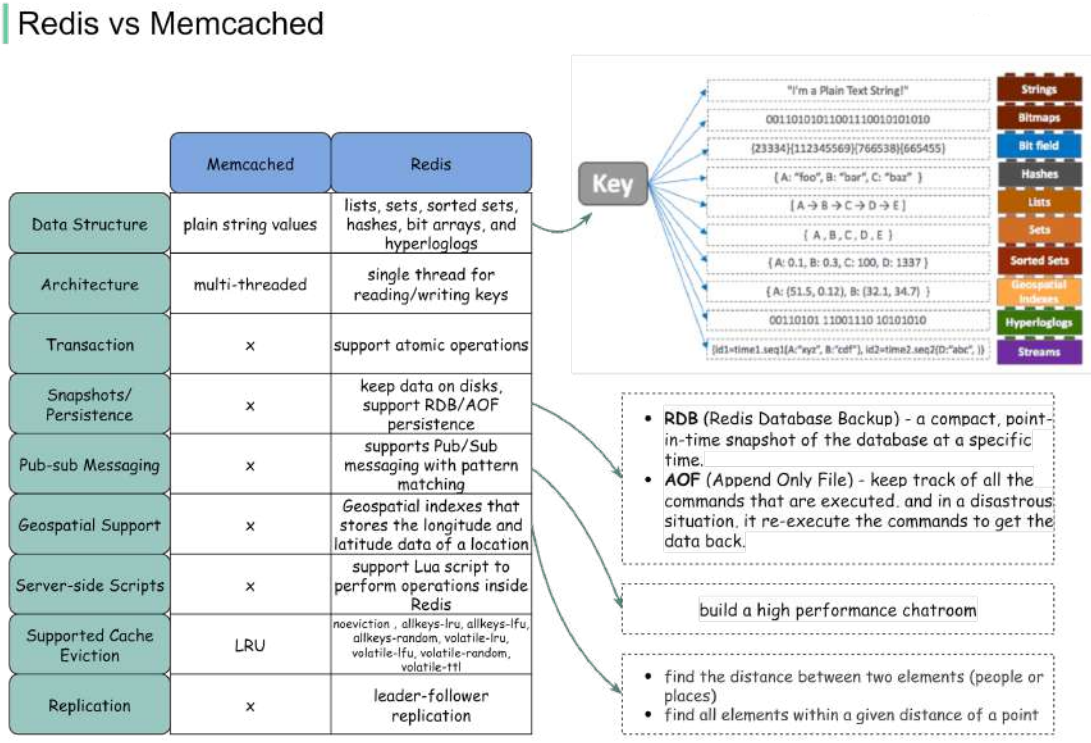


Google published a series of articles about "Inside look at modern web browser". It's a great read.

Links:
https://developer.chrome.com/blog/inside-browser-part1/
https://developer.chrome.com/blog/inside-browser-part2/
https://developer.chrome.com/blog/inside-browser-part3/
https://developer.chrome.com/blog/inside-browser-part4/

Redis vs Memcached

The diagram below illustrates the key differences.



The advantages on data structures make Redis a good choice for:

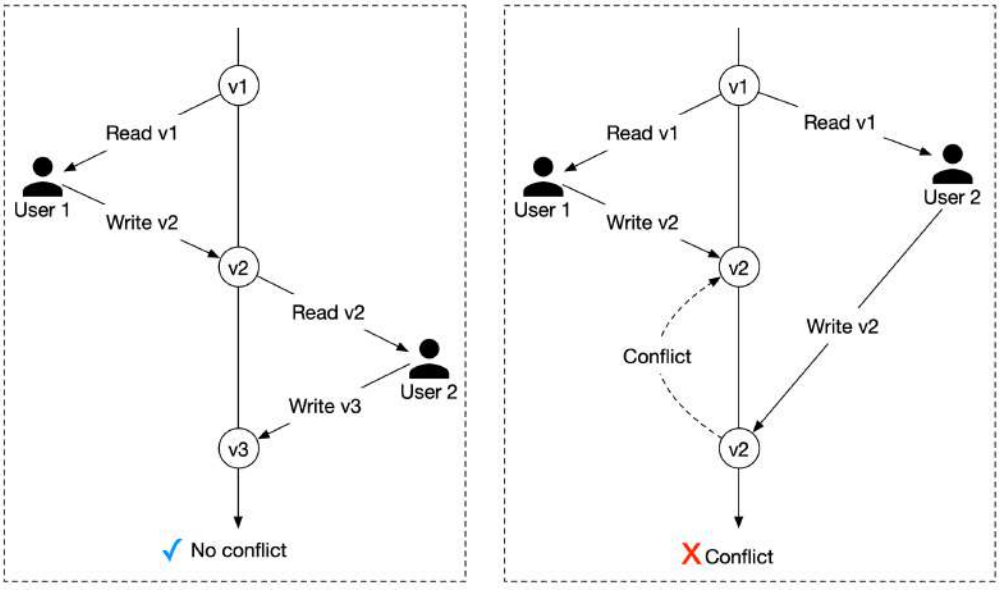
- Recording the number of clicks and comments for each post (hash)
- Sorting the commented user list and deduping the users (zset)
- Caching user behavior history and filtering malicious behaviors (zset, hash)
- Storing boolean information of extremely large data into small space. For example, login status, membership status. (bitmap)

Optimistic locking

Optimistic locking, also referred to as optimistic concurrency control, allows multiple concurrent users to attempt to update the same resource.

There are two common ways to implement optimistic locking: version number and timestamp. Version number is generally considered to be a better option because the server clock can be inaccurate over time. We explain how optimistic locking works with version number.

The diagram below shows a successful case and a failure case.



1. A new column called “version” is added to the database table.

2. Before a user modifies a database row, the application reads the version number of the row.

3. When the user updates the row, the application increases the version number by 1 and writes it back to the database.

4. A database validation check is put in place; the next version number should exceed the current version number by 1. The transaction aborts if the validation fails and the user tries again from step 2.

Optimistic locking is usually faster than pessimistic locking because we do not lock the database. However, the performance of optimistic locking drops dramatically when concurrency is high.

To understand why, consider the case when many clients try to reserve a hotel room at the same time. Because there is no limit on how many clients can read the available room count, all of them read back the same available room count and the current version number. When different clients make reservations and write back the results to the database, only one of them will succeed, and the rest of the clients receive a version check failure message. These clients have to retry. In the subsequent round of retries, there is only one successful client again, and the rest have to retry. Although the end result is correct, repeated retries cause a very unpleasant user experience.

Question: what are the possible ways of solving race conditions?

Tradeoff between latency and consistency

Understanding the **tradeoffs** is very important not only in system design interviews but also designing real-world systems. When we talk about data replication, there is a fundamental tradeoff between **latency** and **consistency**. It is illustrated by the diagram below.

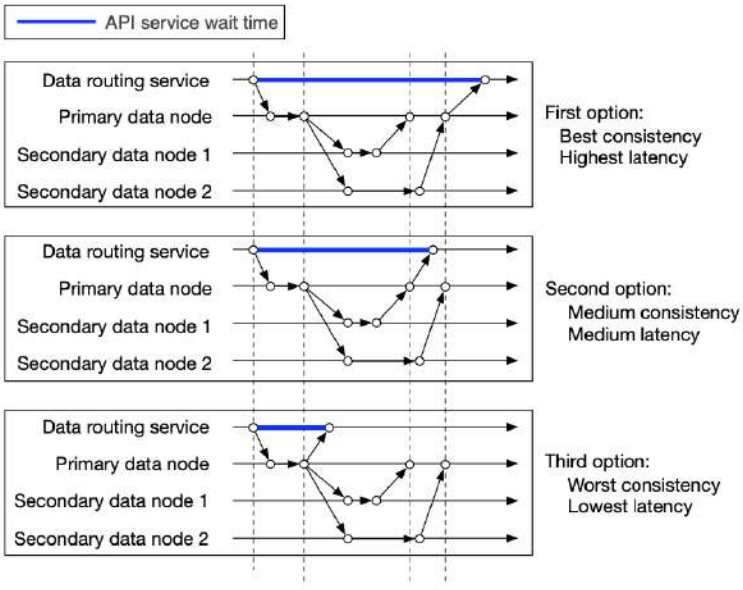


Figure 9.11: Trade-off between consistency and latency

1. Data is considered as successfully saved after all three nodes store the data. This approach has the best consistency but the highest latency.
2. Data is considered as successfully saved after the primary and one of the secondaries store the data. This approach has a medium consistency and medium latency.
3. Data is considered as successfully saved after the primary persists the data. This approach has the worst consistency but the lowest latency.

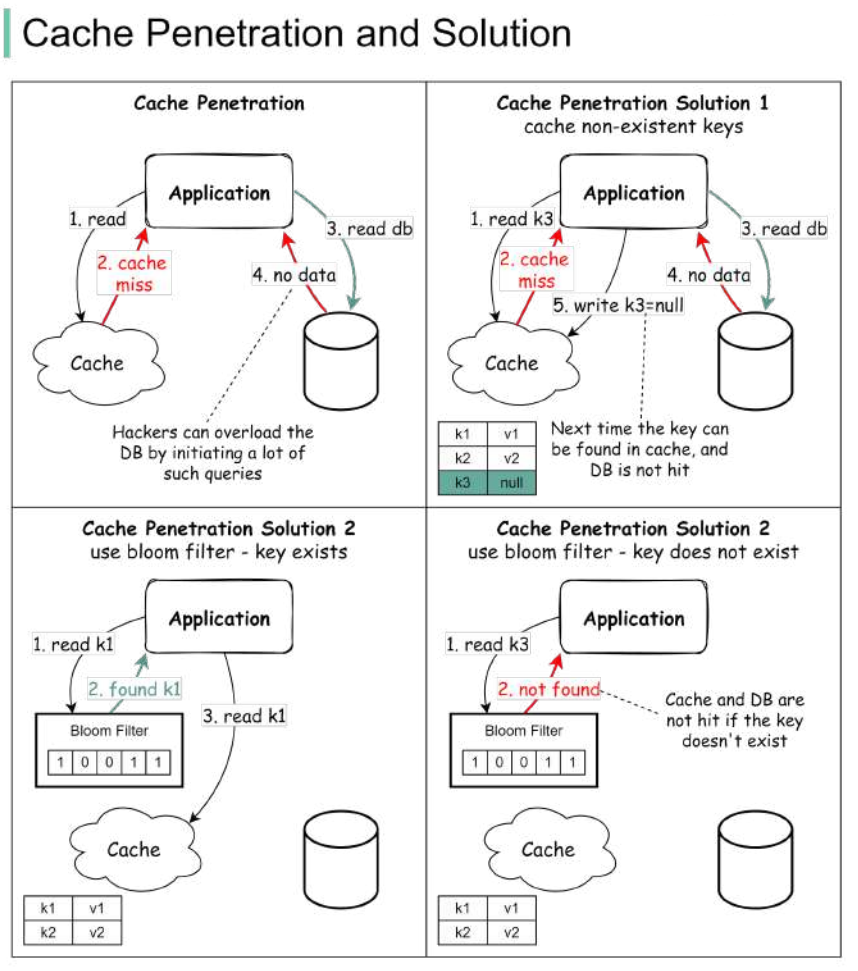
Both 2 and 3 are forms of eventual consistency.

Cache miss attack

Caching is awesome but it doesn't come without a cost, just like many things in life.

One of the issues is **Cache Miss Attack**. Correct me if this is not the right term. It refers to the scenario where data to fetch doesn't exist in the database and the data isn't cached either. So every request hits the database eventually, defeating the purpose of using a cache. If a malicious user initiates lots of queries with such keys, the database can easily be overloaded.

The diagram below illustrates the process.



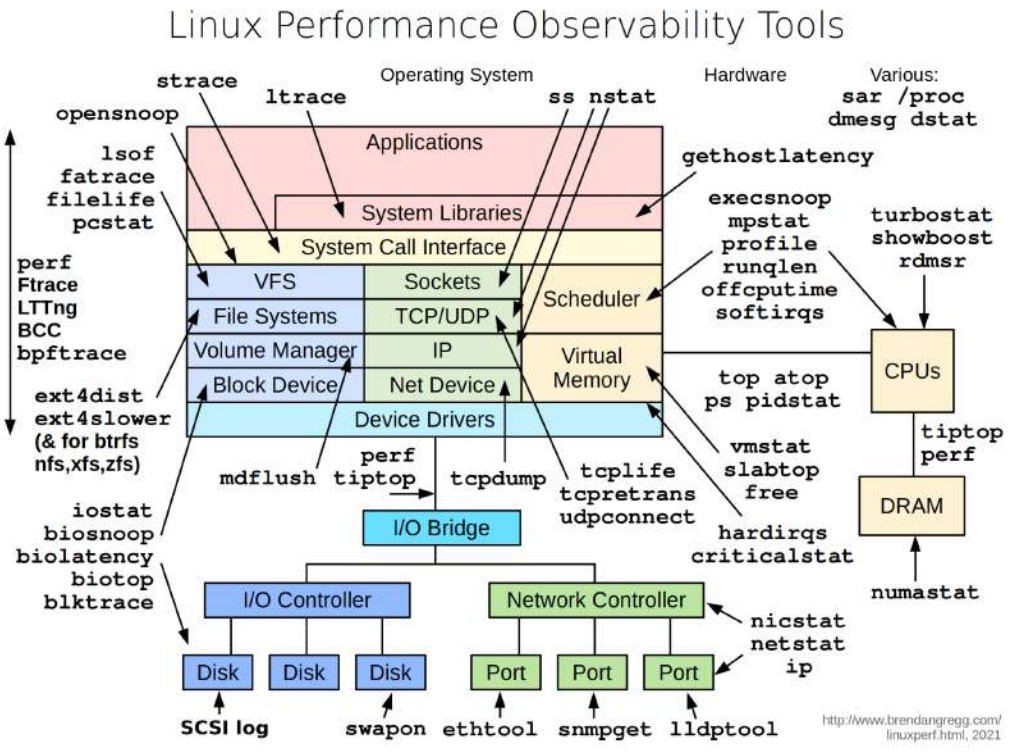
Two approaches are commonly used to solve this problem:

🔹 Cache keys with null value. Set a short TTL (Time to Live) for keys with null value.

🔹 Using Bloom filter. A Bloom filter is a data structure that can rapidly tell us whether an element is present in a set or not. If the key exists, the request first goes to the cache and then queries the database if needed. If the key doesn't exist in the data set, it means the key doesn’t exist in the cache/database. In this case, the query will not hit the cache or database layer.

How to diagnose a mysterious process that's taking too much CPU, memory, IO, etc?

The diagram below illustrates helpful tools in a Linux system.



- 'vmstat' - reports information about processes, memory, paging, block IO, traps, and CPU activity.

- 'iostat' - reports CPU and input/output statistics of the system.

- 'netstat' - displays statistical data related to IP, TCP, UDP, and ICMP protocols.

- 'lsof' - lists open files of the current system.

- 'pidstat' - monitors the utilization of system resources by all or specified processes, including CPU, memory, device IO, task switching, threads, etc.

What are the top cache strategies?

Read data from the system:

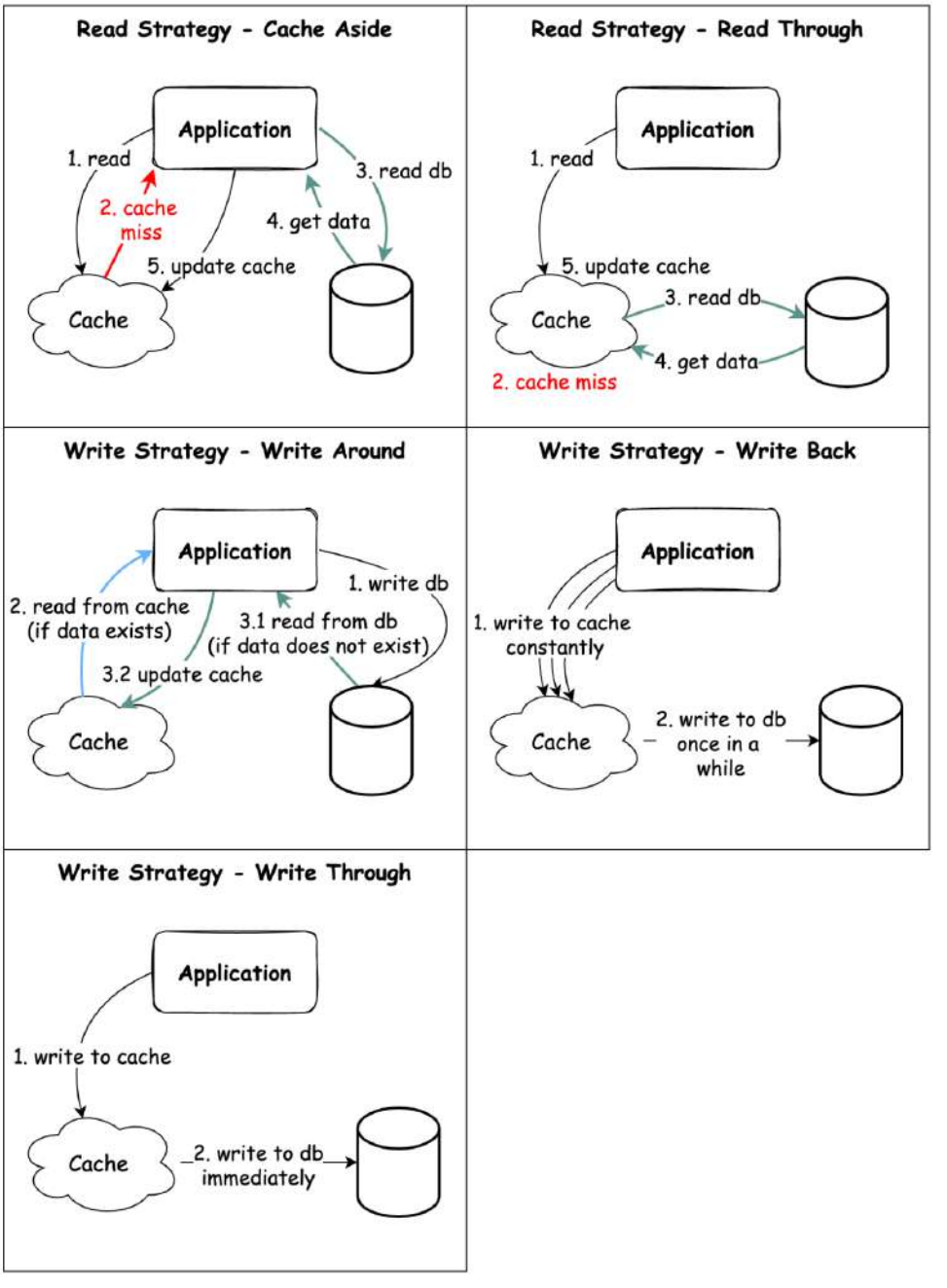
- Cache aside
- Read through

Write data to the system:

- Write around
- Write back
- Write through

The diagram below illustrates how those 5 strategies work. Some of the caching strategies can be used together.

Top caching strategies



I left out a lot of details as that will make the post very long. Feel free to leave a comment so we can learn from each other.

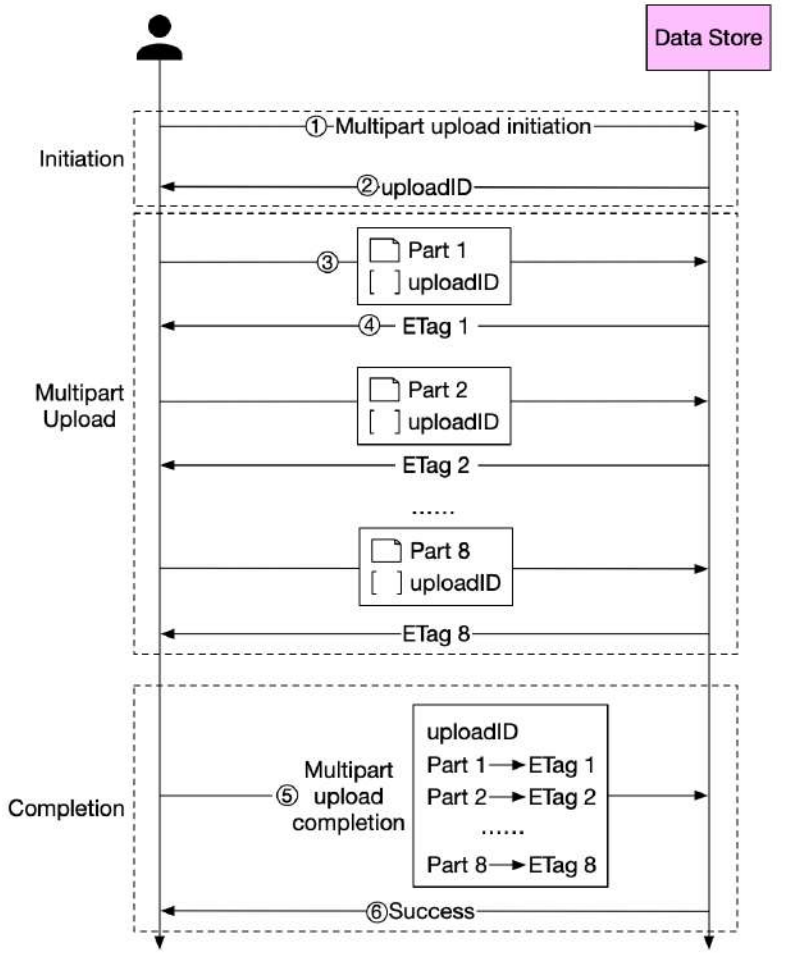
Question: What are the pros and cons of each caching strategy? How to choose the right one to use?

Upload large files

How can we optimize performance when we **upload large files** to object storage service such as S3?

Before we answer this question, let's take a look at why we need to optimize this process. Some files might be larger than a few GBs. It is possible to upload such a large object file directly, but it could take a long time. If the network connection fails in the middle of the upload, we have to start over. A better solution is to slice a large object into smaller parts and upload them independently. After all the parts are uploaded, the object store re-assembles the object from the parts. This process is called **multipart upload**.

The diagram below illustrates how multipart upload works:



1. The client calls the object storage to initiate a multipart upload.

2. The data store returns an uploadID, which uniquely identifies the upload.

3. The client splits the large file into small objects and starts uploading. Let’s assume the size of the file is 1.6GB and the client splits it into 8 parts, so each part is 200 MB in size. The client uploads the first part to the data store together with the uploadID it received in step 2.

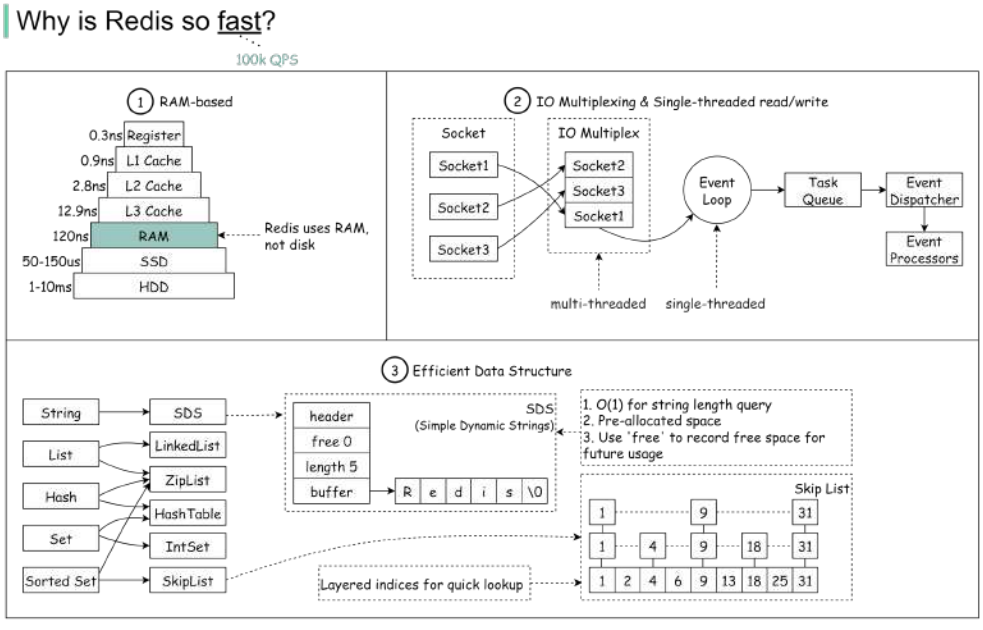
4. When a part is uploaded, the data store returns an ETag, which is essentially the md5 checksum of that part. It is used to verify multipart uploads.

5. After all parts are uploaded, the client sends a complete multipart upload request, which includes the uploadID, part numbers, and ETags.

6. The data store reassembles the object from its parts based on the part number. Since the object is really large, this process may take a few minutes. After reassembly is complete, it returns a success message to the client.

Why is Redis so Fast?

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



1. Redis is a RAM-based database. 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.

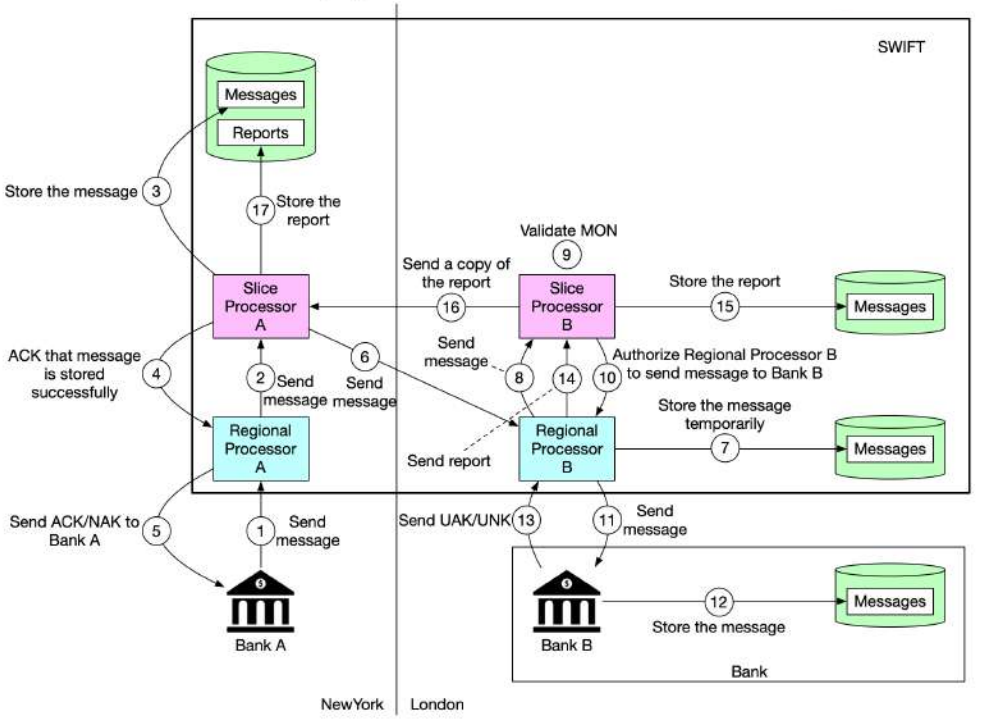
Question: Another popular in-memory store is Memcached. Do you know the differences between Redis and Memcached?

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

SWIFT payment network

You probably heard about **SWIFT**. What is SWIFT? What role does it play in cross-border payments? You can find answers to those questions in this post.

SWIFT for cross-border payments



The Society for Worldwide Interbank Financial Telecommunication (SWIFT) is the main secure **messaging system** that links the world's banks.

The Belgium-based system is run by its member banks and handles millions of payment messages per day. The diagram below illustrates how payment messages are transmitted from Bank A (in New York) to Bank B (in London).

Step 1: Bank A sends a message with transfer details to Regional Processor A in New York. The destination is Bank B.

Step 2: Regional processor validates the format and sends it to Slice Processor A. The Regional Processor is responsible for input message validation and output message queuing. The Slice Processor is responsible for storing and routing messages safely.

Step 3: Slice Processor A stores the message.

Step 4: Slice Processor A informs Regional Processor A the message is stored.

Step 5: Regional Processor A sends ACK/NAK to Bank A. ACK means a message will be sent to Bank B. NAK means the message will NOT be sent to Bank B.

Step 6: Slice Processor A sends the message to Regional Processor B in London.

Step 7: Regional Processor B stores the message temporarily.

Step 8: Regional Processor B assigns a unique ID MON (Message Output Number) to the message and sends it to Slice Processor B

Step 9: Slice Processor B validates MON.

Step 10: Slice Processor B authorizes Regional Processor B to send the message to Bank B.

Step 11: Regional Processor B sends the message to Bank B.

Step 12: Bank B receives the message and stores it.

Step 13: Bank B sends UAK/UNK to Regional Processor B. UAK (user positive acknowledgment) means Bank B received the message without error; UNK (user negative acknowledgment) means Bank B received checksum failure.

Step 14: Regional Processor B creates a report based on Bank B’s response, and sends it to Slice Processor B.

Step 15: Slice Processor B stores the report.

Step 16 - 17: Slice Processor B sends a copy of the report to Slice Processor A. Slice Processor A stores the report.

At-most once, at-least once, and exactly once

In modern architecture, systems are broken up into small and independent building blocks with well-defined interfaces between them. Message queues provide communication and coordination for those building blocks. Today, let's discuss different delivery semantics: at-most once, at-least once, and exactly once.

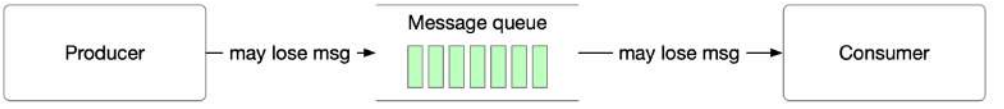


Figure 1 At-most once

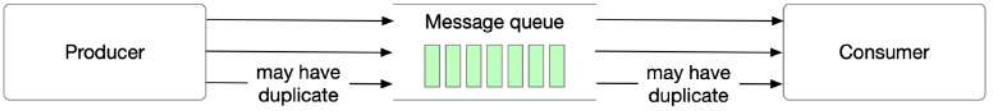


Figure 2 At-least once

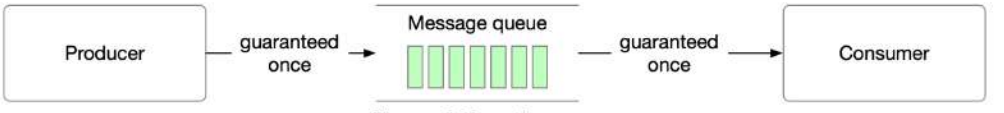


Figure 3 Exactly-once

At-most once

As the name suggests, at-most once means a message will be delivered not more than once. Messages may be lost but are not redelivered. This is how at-most once delivery works at the high level.

Use cases: It is suitable for use cases like monitoring metrics, where a small amount of data loss is acceptable.

At-least once

With this data delivery semantic, it's acceptable to deliver a message more than once, but no message should be lost.

Use cases: With at-least once, messages won't be lost but the same message might be delivered multiple times. While not ideal from a user perspective, at-least once delivery semantics are usually good enough for use cases where data duplication is not a big issue or deduplication

is possible on the consumer side. For example, with a unique key in each message, a message can be rejected when writing duplicate data to the database.

Exactly once

Exactly once is the most difficult delivery semantic to implement. It is friendly to users, but it has a high cost for the system's performance and complexity.

Use cases: Financial-related use cases (payment, trading, accounting, etc.). Exactly once is especially important when duplication is not acceptable and the downstream service or third party doesn't support idempotency.

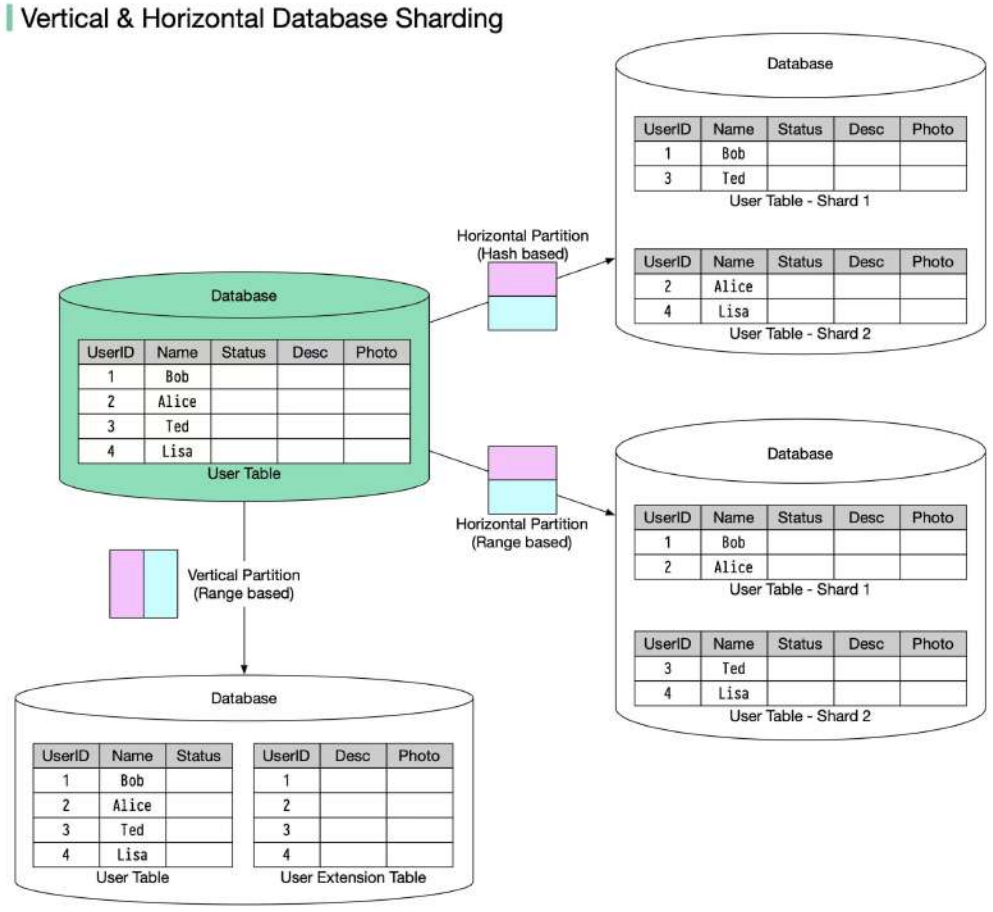
Question: what is the difference between message queues vs event streaming platforms such as Kafka, Apache Pulsar, etc?

Vertical partitioning and Horizontal partitioning

In many large-scale applications, data is divided into partitions that can be accessed separately. There are two typical strategies for partitioning data.

- Vertical partitioning: it means some columns are moved to new tables. Each table contains the same number of rows but fewer columns (see diagram below).

- Horizontal partitioning (often called sharding): it divides a table into multiple smaller tables. Each table is a separate data store, and it contains the same number of columns, but fewer rows (see diagram below).



Horizontal partitioning is widely used so let’s take a closer look.

Routing algorithm
Routing algorithm decides which partition (shard) stores the data.

🔹 Range-based sharding. This algorithm uses ordered columns, such as integers, longs, timestamps, to separate the rows. For example, the diagram below uses the User ID column for range partition: User IDs 1 and 2 are in shard 1, User IDs 3 and 4 are in shard 2.

🔹 Hash-based sharding. This algorithm applies a hash function to one column or several columns to decide which row goes to which table. For example, the diagram below uses **User ID mod** 2 as a hash function. User IDs 1 and 3 are in shard 1, User IDs 2 and 4 are in shard 2.

Benefits
🔹 Facilitate horizontal scaling. Sharding facilitates the possibility of adding more machines to spread out the load.

🔹 Shorten response time. By sharding one table into multiple tables, queries go over fewer rows, and results are returned much more quickly.

Drawbacks
🔹 The order by the operation is more complicated. Usually, we need to fetch data from different shards and sort the data in the application's code.

🔹 Uneven distribution. Some shards may contain more data than others (this is also called the hotspot).

This topic is very big and I’m sure I missed a lot of important details. What else do you think is important for data partitioning?

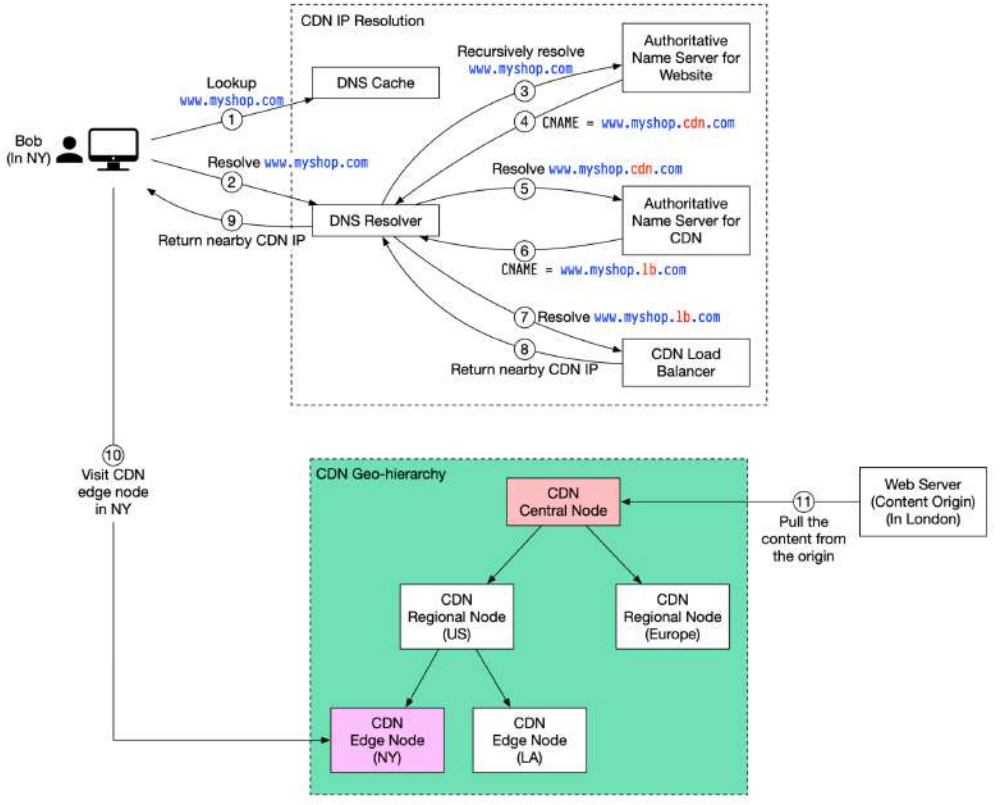
CDN

A content delivery network (CDN) refers to a geographically distributed servers (also called edge servers) which provide fast delivery of static and dynamic content. Let's take a look at how it works.

Suppose Bob who lives in New York wants to visit an eCommerce website that is deployed in London. If the request goes to servers located in London, the response will be quite slow. So we deploy CDN servers close to where Bob lives, and the content will be loaded from the nearby CDN server.

The diagram below illustrates the process:

How does CDN work



1. Bob types in www.myshop.com in the browser. The browser looks up the domain name in the local DNS cache.

2. If the domain name does not exist in the local DNS cache, the browser goes to the DNS resolver to resolve the name. The DNS resolver usually sits in the Internet Service Provider (ISP).

3. The DNS resolver recursively resolves the domain name (see my previous post for details). Finally, it asks the authoritative name server to resolve the domain name.

4. If we don’t use CDN, the authoritative name server returns the IP address for www.myshop.com. But with CDN, the authoritative name server has an alias pointing to www.myshop.cdn.com (the domain name of the CDN server).

5. The DNS resolver asks the authoritative name server to resolve www.myshop.cdn.com.

6. The authoritative name server returns the domain name for the load balancer of CDN www.myshop.lb.com.

7. The DNS resolver asks the CDN load balancer to resolve www.myshop.lb.com. The load balancer chooses an optimal CDN edge server based on the user’s IP address, user’s ISP, the content requested, and the server load.

8. The CDN load balancer returns the CDN edge server’s IP address for www.myshop.lb.com.

9. Now we finally get the actual IP address to visit. The DNS resolver returns the IP address to the browser.

10. The browser visits the CDN edge server to load the content. There are two types of contents cached on the CDN servers: static contents and dynamic contents. The former contains static pages, pictures, and videos; the latter one includes results of edge computing.

11. If the edge CDN server cache doesn't contain the content, it goes upward to the regional CDN server. If the content is still not found, it will go upward to the central CDN server, or even go to the origin - the

London web server. This is called the CDN distribution network, where the servers are deployed geographically.

Over to you: How do you prevent videos cached on CDN from being pirated?

Erasure coding

A really cool technique that's commonly used in object storage such as S3 to improve durability is called **Erasure Coding**. Let's take a look at how it works.

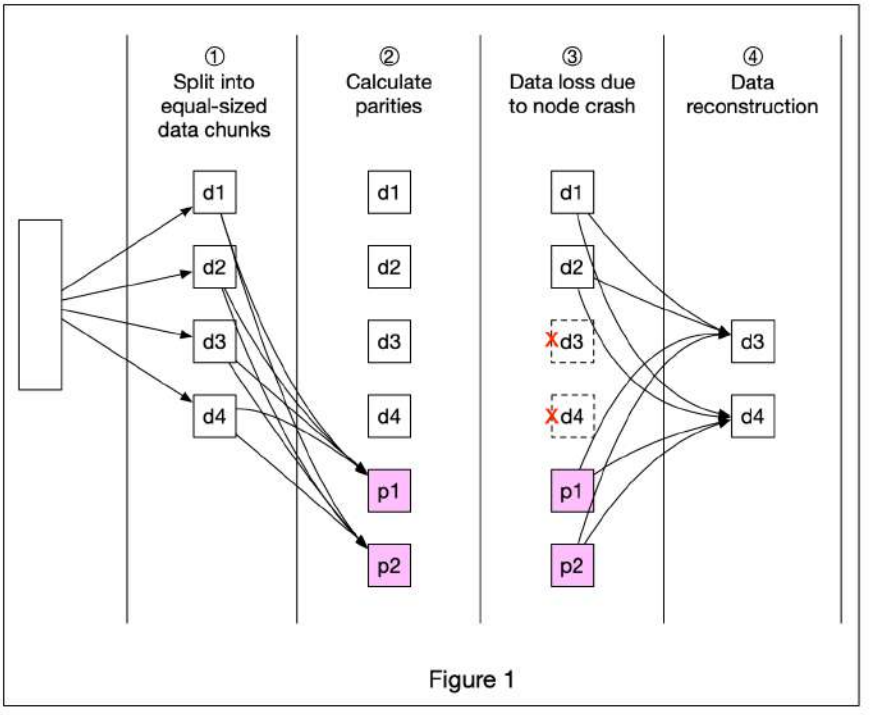


Figure 1

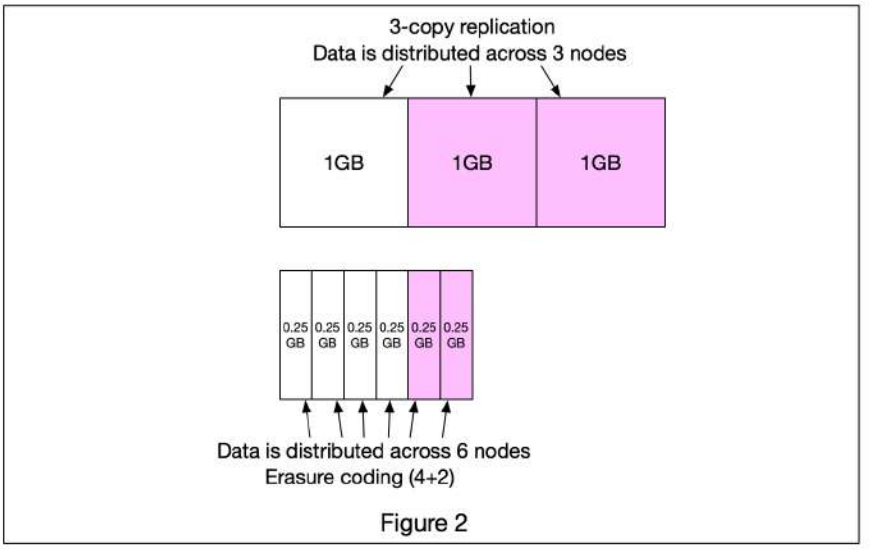


Figure 2

Erasure coding deals with data durability differently from replication. It chunks data into smaller pieces (placed on different servers) and creates parities for redundancy. In the event of failures, we can use chunk data and parities to reconstruct the data. Let's take a look at a concrete example (4 + 2 erasure coding) as shown in Figure 1.

1 Data is broken up into four even-sized data chunks d1, d2, d3, and d4.

2 The mathematical formula is used to calculate the parities p1 and p2. To give a much simplified example, p1 = d1 + 2*d2 - d3 + 4*d4 and p2 = -d1 + 5*d2 + d3 - 3*d4.

3 Data d3 and d4 are lost due to node crashes.

4 The mathematical formula is used to reconstruct lost data d3 and d4, using the known values of d1, d2, p1, and p2.

How much extra space does erasure coding need? For every two chunks of data, we need one parity block, so the storage overhead is 50% (Figure 2). While in 3-copy replication, the storage overhead is 200% (Figure 2).

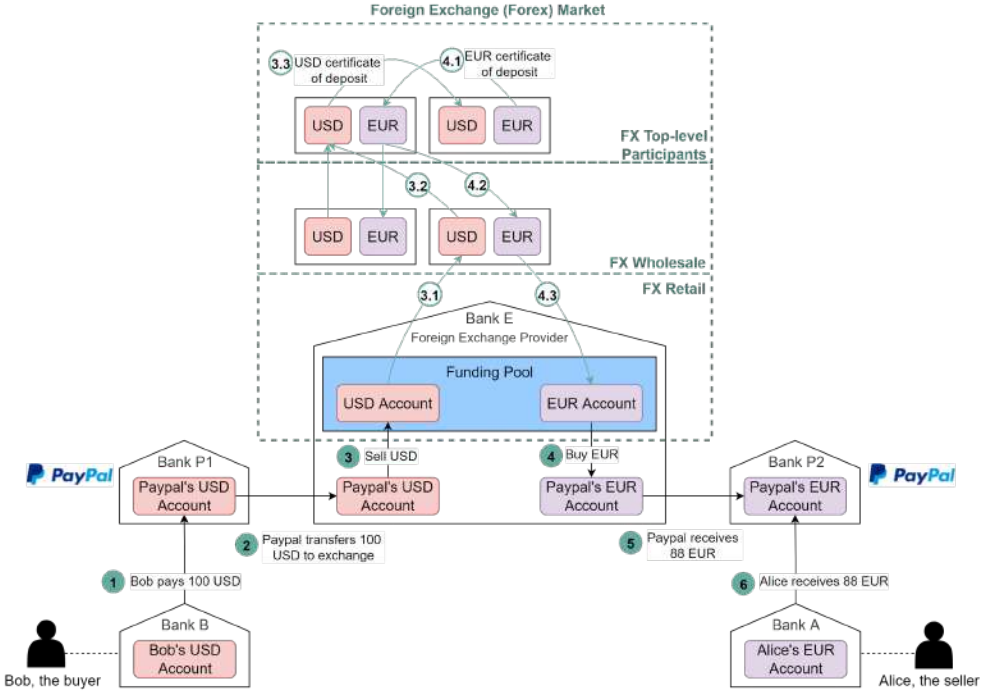
Does erasure coding increase data durability? Let's assume a node has a 0.81% annual failure rate. According to the calculation done by Backblaze, erasure coding can achieve 11 nines durability vs 3-copy replication can achieve 6 nines durability.

What other techniques do you think are important to improve the scalability and durability of an object store such as S3?

Foreign exchange in payment

Have you wondered what happens under the hood when you pay with USD online and the seller from Europe receives EUR (euro)? This process is called foreign exchange.

Foreign Exchange in Payments



Suppose Bob (the buyer) needs to pay 100 USD to Alice (the seller), and Alice can only receive EUR. The diagram below illustrates the process.

1. Bob sends 100 USD via a third-party payment provider. In our example, it is Paypal. The money is transferred from Bob's bank account (Bank B) to Paypal's account in Bank P1.

2. Paypal needs to convert USD to EUR. It leverages the foreign exchange provider (Bank E). Paypal sends 100 USD to its USD account in Bank E.

3. 100 USD is sold to Bank E's funding pool.

4. Bank E's funding pool provides 88 EUR in exchange for 100 USD. The money is put into Paypal's EUR account in Bank E.

5. Paypal's EUR account in Bank P2 receives 88 EUR.

6. 88 EUR is paid to Alice's EUR account in Bank A.

Now let's take a close look at the foreign exchange (forex) market. It has 3 layers:

- Retail market. Funding pools are parts of the retail market. To improve efficiency, Paypal usually buys a certain amount of foreign currencies in advance.
- Wholesale market. The wholesale business is composed of investment banks, commercial banks, and foreign exchange providers. It usually handles accumulated orders from the retail market.
- Top-level participants. They are multinational commercial banks that hold a large number of certificates of deposit from different countries. They exchange these certificates for foreign exchange trading.

When Bank E's funding pool needs more EUR, it goes upward to the wholesale market to sell USD and buy EUR. When the wholesale market accumulates enough orders, it goes upward to top-level participants. Steps 3.1-3.3 and 4.1-4.3 explain how it works.

If you have any questions, please leave a comment.

What foreign currency did you find difficult to exchange? And what company have you used for foreign currency exchange?

Interview Question: Design S3

What happens when you upload a file to Amazon S3? Let's design an S3 like object storage system.

Upload a File to S3

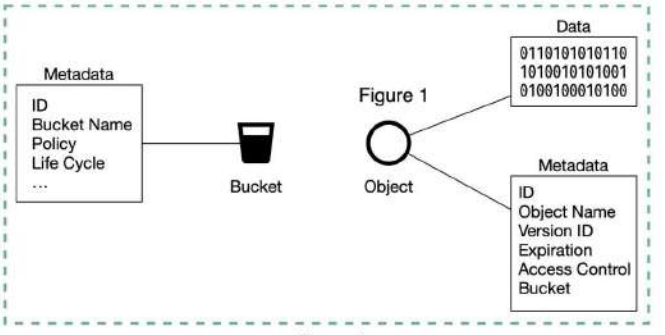


Figure 1

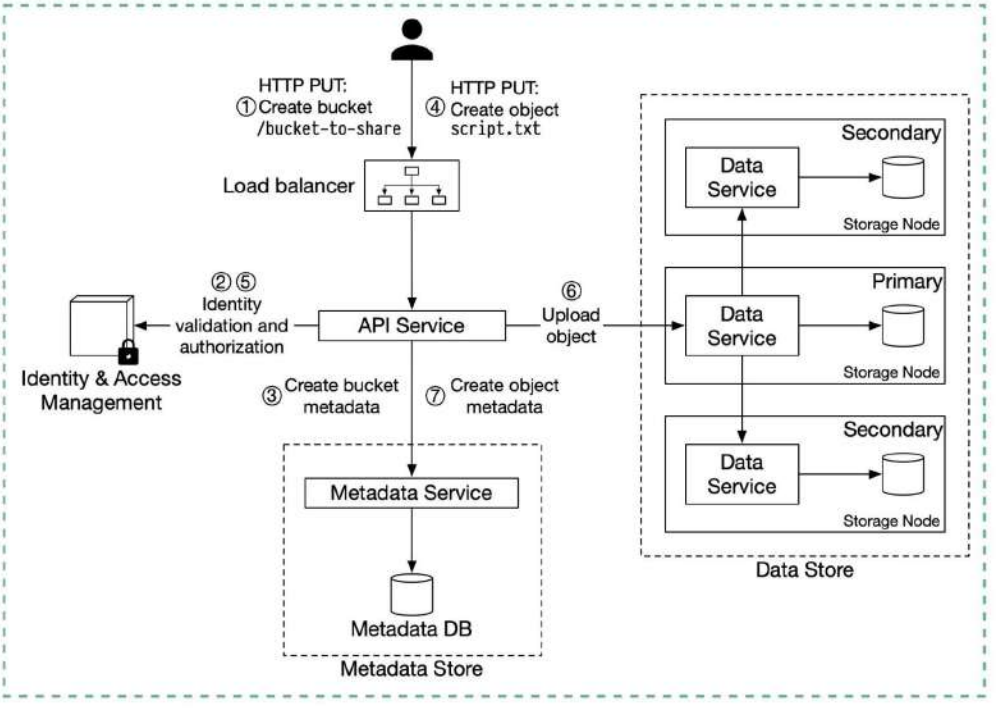


Figure 2

Before we dive into the design, let's define some terms.

Bucket. A logical container for objects. The bucket name is globally unique. To upload data to S3, we must first create a bucket.

Object. An object is an individual piece of data we store in a bucket. It contains object data (also called payload) and metadata. Object data can be any sequence of bytes we want to store. The metadata is a set of name-value pairs that describe the object.

An S3 object consists of (Figure 1):

- Metadata. It is mutable and contains attributes such as ID, bucket name, object name, etc.
- Object data. It is immutable and contains the actual data.

In S3, an object resides in a bucket. The path looks like this: /bucket-to-share/script.txt. The bucket only has metadata. The object has metadata and the actual data.

The diagram below (Figure 2) illustrates how file uploading works. In this example, we first create a bucket named “bucket-to-share” and then upload a file named “script.txt” to the bucket.

1. The client sends an HTTP PUT request to create a bucket named “bucket-to-share.” The request is forwarded to the API service.

2. The API service calls the Identity and Access Management (IAM) to ensure the user is authorized and has WRITE permission.

3. The API service calls the metadata store to create an entry with the bucket info in the metadata database. Once the entry is created, a success message is returned to the client.

4. After the bucket is created, the client sends an HTTP PUT request to create an object named “script.txt”.

5. The API service verifies the user’s identity and ensures the user has WRITE permission on the bucket.

6. Once validation succeeds, the API service sends the object data in the HTTP PUT payload to the data store. The data store persists the payload as an object and returns the UUID of the object.

7. The API service calls the metadata store to create a new entry in the metadata database. It contains important metadata such as the object_id (UUID), bucket_id (which bucket the object belongs to), object_name, etc.

Block storage, file storage and object storage

Yesterday, I posted the definitions of block storage, file storage, and object storage. Let's continue the discussion and compare those 3 options.

	Block storage	File storage	Object storage
Mutable Content	Y	Y	N (object versioning is supported, in-place update is not)
Cost	High	Medium to high	Low
Performance	Medium to high, very high	Medium to high	Low to medium
Consistency	Strong consistency	Strong consistency	Strong consistency
Data access	SAS/iSCSI/FC	Standard file access, CIFS/SMB, and NFS	RESTful API
Scalability	Medium scalability	High scalability	Vast scalability
Good for	Virtual machines (VM), high-performance applications like database	General-purpose file system access	Binary data, unstructured data

Table 1 Storage options

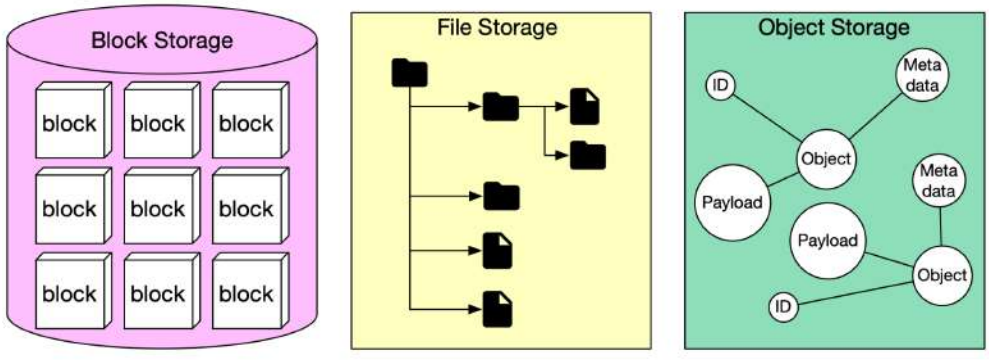
Block storage, file storage and object storage

In this post, let's review the storage systems in general.

Storage systems fall into three broad categories:

- Block storage
- File storage
- Object storage

The diagram below illustrates the comparison of different storage systems.



Block storage

Block storage came first, in the 1960s. Common storage devices like hard disk drives (HDD) and solid-state drives (SSD) that are physically attached to servers are all considered as block storage.

Block storage presents the raw blocks to the server as a volume. This is the most flexible and versatile form of storage. The server can format the raw blocks and use them as a file system, or it can hand control of those blocks to an application. Some applications like a database or a virtual machine engine manage these blocks directly in order to squeeze every drop of performance out of them.

Block storage is not limited to physically attached storage. Block storage could be connected to a server over a high-speed network or over industry-standard connectivity protocols like Fibre Channel (FC)

and iSCSI. Conceptually, the network-attached block storage still presents raw blocks. To the servers, it works the same as physically attached block storage. Whether to a network or physically attached, block storage is fully owned by a single server. It is not a shared resource.

File storage

File storage is built on top of block storage. It provides a higher-level abstraction to make it easier to handle files and directories. Data is stored as files under a hierarchical directory structure. File storage is the most common general-purpose storage solution. File storage could be made accessible by a large number of servers using common file-level network protocols like SMB/CIFS and NFS. The servers accessing file storage do not need to deal with the complexity of managing the blocks, formatting volume, etc. The simplicity of file storage makes it a great solution for sharing a large number of files and folders within an organization.

Object storage

Object storage is new. It makes a very deliberate tradeoff to sacrifice performance for high durability, vast scale, and low cost. It targets relatively “cold” data and is mainly used for archival and backup. Object storage stores all data as objects in a flat structure. There is no hierarchical directory structure. Data access is normally provided via a RESTful API. It is relatively slow compared to other storage types. Most public cloud service providers have an object storage offering, such as AWS S3, Google block storage, and Azure blob storage.

Domain Name System (DNS) lookup

DNS acts as an address book. It translates human-readable domain names (google.com) to machine-readable IP addresses (142.251.46.238).

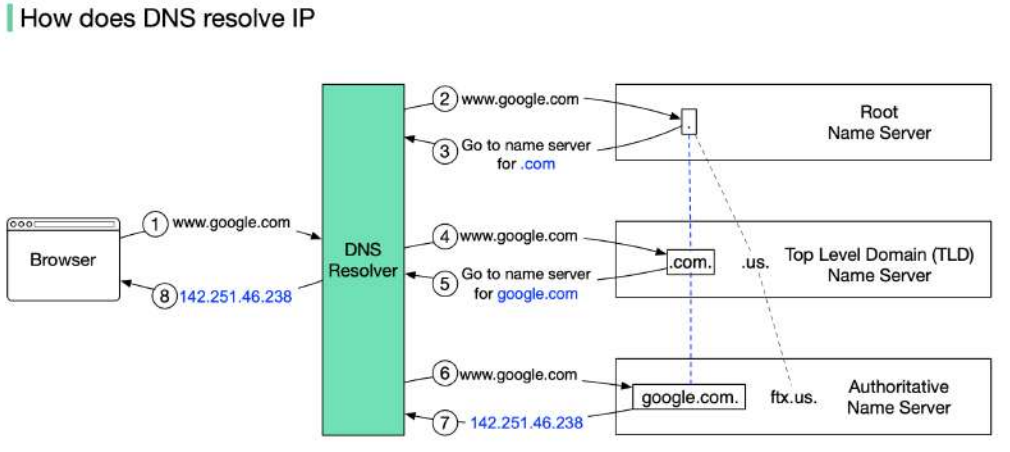
To achieve better scalability, the DNS servers are organized in a hierarchical tree structure.

There are 3 basic levels of DNS servers:

1. Root name server (.). It stores the IP addresses of Top Level Domain (TLD) name servers. There are 13 logical root name servers globally.
2. TLD name server. It stores the IP addresses of authoritative name servers. There are several types of TLD names. For example, generic TLD (.com, .org), country code TLD (.us), test TLD (.test).

3. Authoritative name server. It provides actual answers to the DNS query. You can register authoritative name servers with domain name registrar such as GoDaddy, Namecheap, etc.

The diagram below illustrates how DNS lookup works under the hood:



1. google.com is typed into the browser, and the browser sends the domain name to the DNS resolver.

2. The resolver queries a DNS root name server.

3. The root server responds to the resolver with the address of a TLD DNS server. In this case, it is .com.

4. The resolver then makes a request to the .com TLD.

5. The TLD server responds with the IP address of the domain's name server, google.com (authoritative name server).

6. The DNS resolver sends a query to the domain's nameserver.

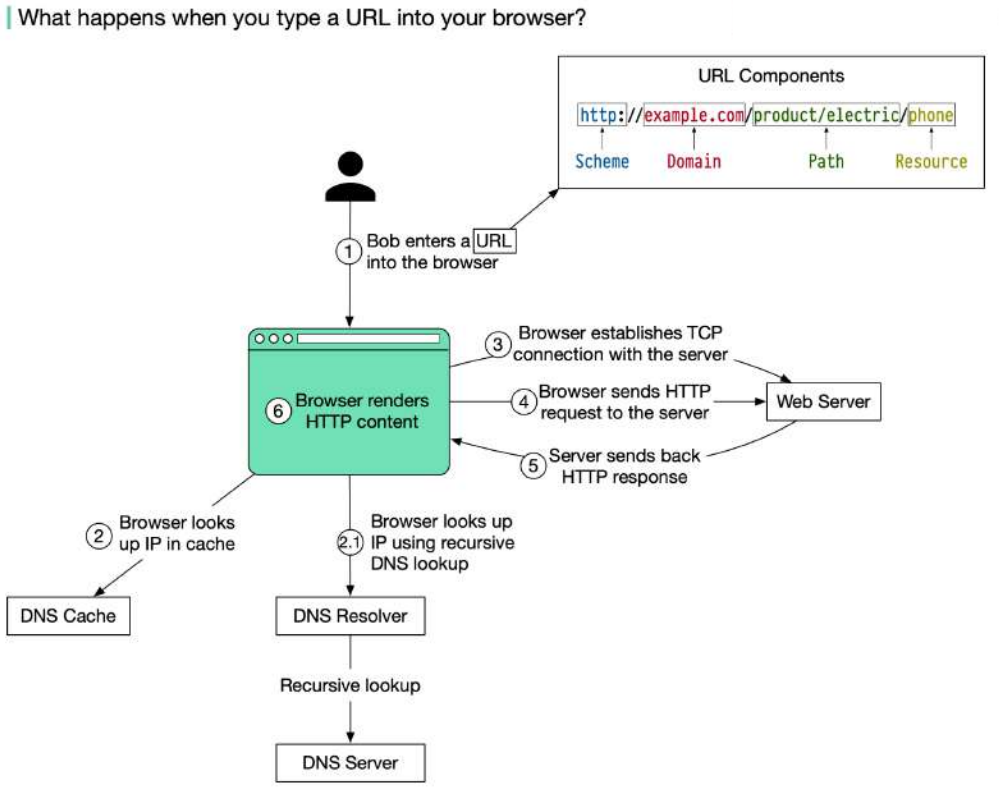
7. The IP address for google.com is then returned to the resolver from the nameserver.

8. The DNS resolver responds to the web browser with the IP address (142.251.46.238) of the domain requested initially.

DNS lookups on average take between 20-120 milliseconds to complete (according to YSlow).

What happens when you type a URL into your browser?

The diagram below illustrates the steps.



1. Bob enters a URL into the browser and hits Enter. In this example, the URL is composed of 4 parts:

- scheme - ***https***://. This tells the browser to send a connection to the server using HTTPS.
- domain - ***example.com***. This is the domain name of the site.
- path - ***product/electric***. It is the path on the server to the requested resource: phone.
- resource - ***phone***. It is the name of the resource Bob wants to visit.

2. The browser looks up the IP address for the domain with a domain name system (DNS) lookup. To make the lookup process fast, data is cached at different layers: browser cache, OS cache, local network cache and ISP cache.

2.1 If the IP address cannot be found at any of the caches, the browser goes to DNS servers to do a recursive DNS lookup until the IP address is found (this will be covered in another post).

3. Now that we have the IP address of the server, the browser establishes a TCP connection with the server.

4. The browser sends a HTTP request to the server. The request looks like this:

```
GET /phone HTTP/1.1
Host: example.com
```

5. The server processes the request and sends back the response. For a successful response (the status code is 200). The HTML response might look like this:

```
HTTP/1.1 200 OK
Date: Sun, 30 Jan 2022 00:01:01 GMT
Server: Apache
Content-Type: text/html; charset=utf-8

<!DOCTYPE html>
<html lang="en">
Hello world
</html>
```

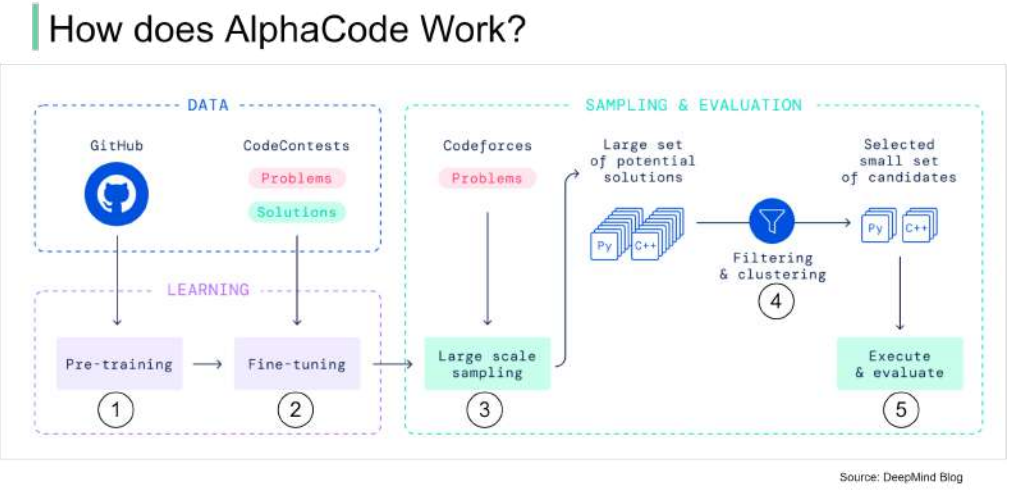
6. The browser renders the HTML content.

AI Coding engine

DeepMind says its new AI coding engine (AlphaCode) is as good as an average programmer.

The AI bot participated in the 10 Codeforces coding competitions and was ranked 54.3%. It means its score exceeded half of the human contestants. If we look at its score for the last 6 months, AlphaCode ranks at 28%.

The diagram below explains how the AI bot works:



1. Pre-train the transformer models on GitHub code.

2. Fine-tune the models on the relatively small competitive programming dataset.

3. At evaluation time, create a massive amount of solutions for each problem.

4. Filter, cluster and rerank the solutions to a small set of candidate programs (at most 10), and then submit for further assessments.

5. Run the candidate programs against the test cases, evaluate the performance, and choose the best one.

Do you think AI bot will be better at Leetcode or competitive programming than software engineers five years from now?

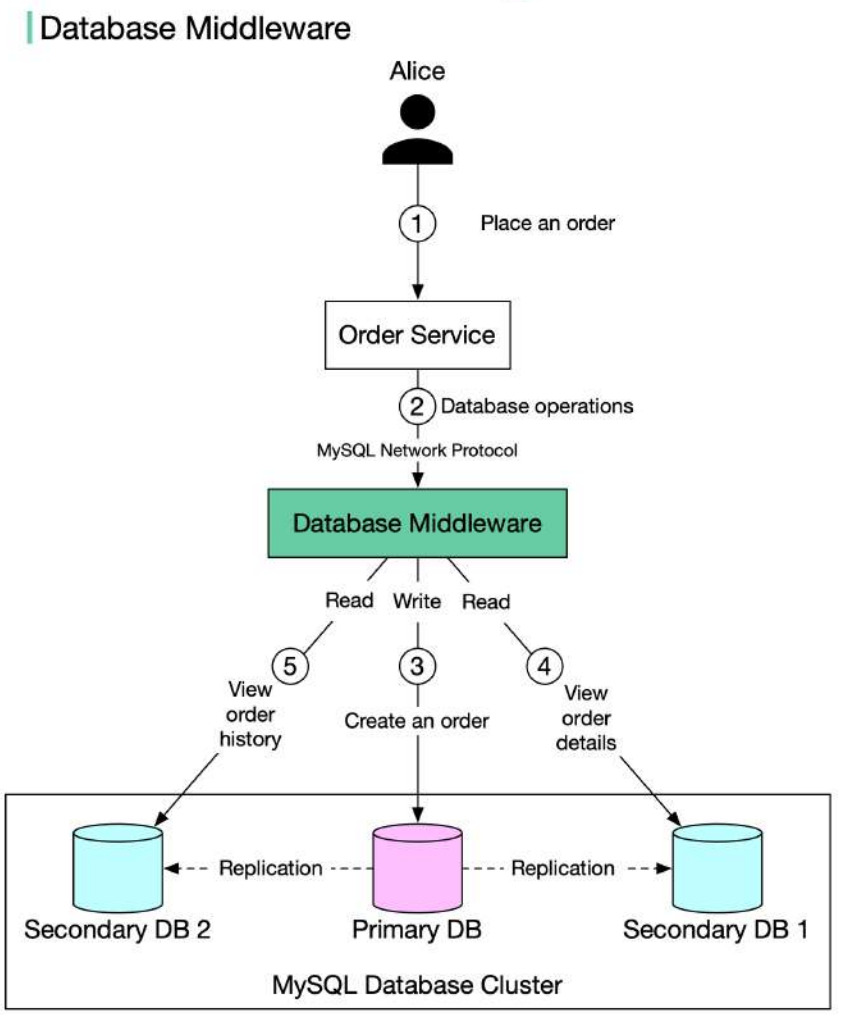
Read replica pattern

There are two common ways to implement the read replica pattern:

1. Embed the routing logic in the application code (explained in the last post).
2. Use database middleware.

We focus on option 2 here. The middleware provides transparent routing between the application and database servers. We can customize the routing logic based on difficult rules such as user, schema, statement, etc.

The diagram below illustrates the setup:



1. When Alice places an order on amazon, the request is sent to Order Service.

2. Order Service does not directly interact with the database. Instead, it sends database queries to the database middleware.

3. The database middleware routes writes to the primary database. Data is replicated to two replicas.

4. Alice views the order details (read). The request is sent through the middleware.

5. Alice views the recent order history (read). The request is sent through the middleware.

The database middleware acts as a proxy between the application and databases. It uses standard MySQL network protocol for communication.

Pros:
- Simplified application code. The application doesn't need to be aware of the database topology and manage access to the database directly.

- Better compatibility. The middleware uses the MySQL network protocol. Any MySQL compatible client can connect to the middleware easily. This makes database migration easier.

Cons:
- Increased system complexity. A database middleware is a complex system. Since all database queries go through the middleware, it usually requires a high availability setup to avoid a single point of failure.

- Additional middleware layer means additional network latency. Therefore, this layer requires excellent performance.

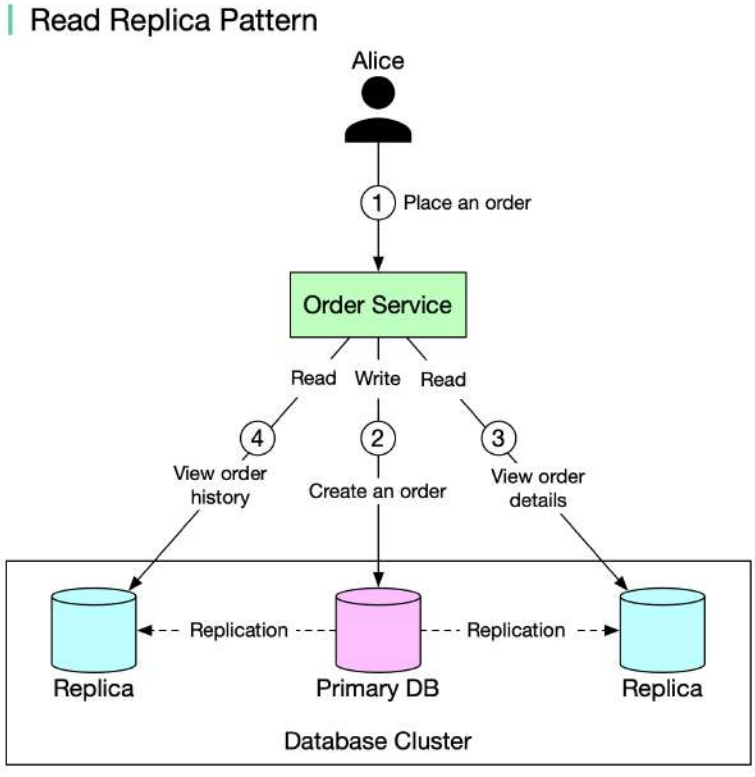
Read replica pattern

In this post, we talk about a simple yet commonly used database design pattern (setup): **Read replica pattern**.

In this setup, all data-modifying commands like insert, delete, or update are sent to the primary DB, and reads are sent to read replicas.

The diagram below illustrates the setup:

1. When Alice places an order on amazon.com, the request is sent to Order Service.
2. Order Service creates a record about the order in the primary DB (write). Data is replicated to two replicas.
3. Alice views the order details. Data is served from a replica (read).
4. Alice views the recent order history. Data is served from a replica (read).



There is one major problem in this setup: **replication lag**.

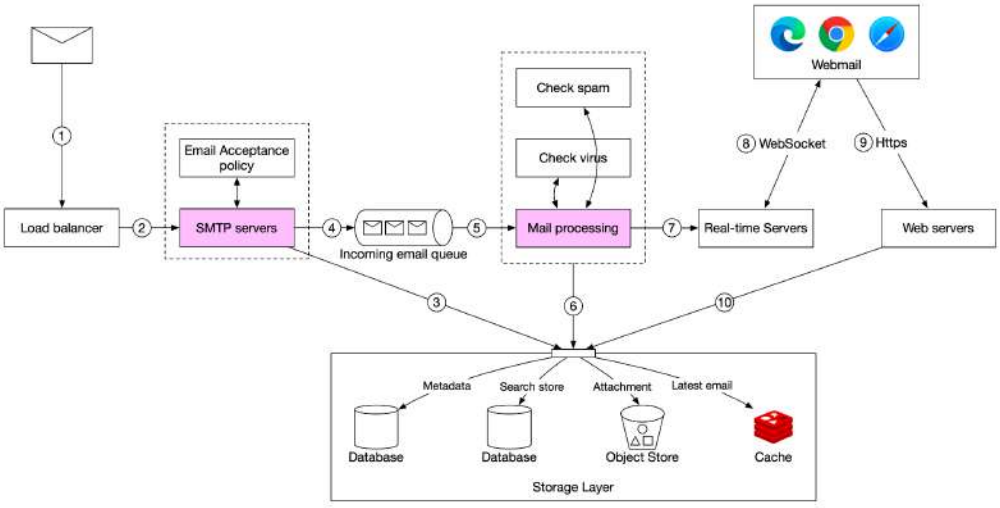
Under certain circumstances (network delay, server overload, etc.), data in replicas might be seconds or even minutes behind. In this case, if Alice immediately checks the order status (query is served by the replica) after the order is placed, she might not see the order at all. This leaves Alice confused. In this case, we need “read-after-write” consistency.

Possible solutions to mitigate this problem:

1. Latency sensitive reads are sent to the primary database.
2. Reads that immediately follow writes are routed to the primary database.
3. A relational DB generally provides a way to check if a replica is caught up with the primary. If data is up to date, query the replica. Otherwise, fail the read request or read from the primary.

Email receiving flow

The following diagram demonstrates the email receiving flow.



1. Incoming emails arrive at the SMTP load balancer.

2. The load balancer distributes traffic among SMTP servers. Email acceptance policy can be configured and applied at the SMTP-connection level. For example, invalid emails are bounced to avoid unnecessary email processing.

3. If the attachment of an email is too large to put into the queue, we can put it into the attachment store (s3).

4. Emails are put in the incoming email queue. The queue decouples mail processing workers from SMTP servers so they can be scaled independently. Moreover, the queue serves as a buffer in case the email volume surges.

5. Mail processing workers are responsible for a lot of tasks, including filtering out spam mails, stopping viruses, etc. The following steps assume an email passed the validation.

6. The email is stored in the mail storage, cache, and object data store.

7. If the receiver is currently online, the email is pushed to real-time servers.

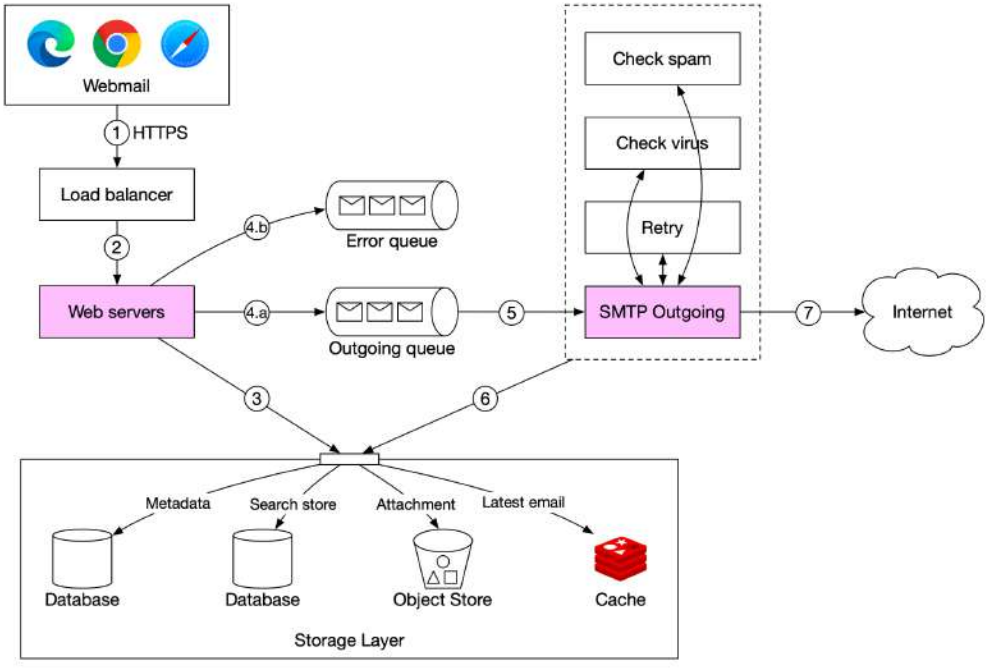
8. Real-time servers are WebSocket servers that allow clients to receive new emails in real-time.

9. For offline users, emails are stored in the storage layer. When a user comes back online, the webmail client connects to web servers via RESTful API.

10. Web servers pull new emails from the storage layer and return them to the client.

Email sending flow

In this post, we will take a closer look at the email sending flow.



1. A user writes an email on webmail and presses the “send” button. The request is sent to the load balancer.

2. The load balancer makes sure it doesn’t exceed the rate limit and routes traffic to web servers.

3. Web servers are responsible for:

- Basic email validation. Each incoming email is checked against pre-defined rules such as email size limit.

- Checking if the domain of the recipient’s email address is the same as the sender. If it is the same, email data is inserted to storage, cache, and object store directly. The recipient can fetch the email directly via the RESTful API. There is no need to go to step 4.

4. Message queues.

4.a. If basic email validation succeeds, the email data is passed to the outgoing queue.

4.b. If basic email validation fails, the email is put in the error queue.

5. SMTP outgoing workers pull events from the outgoing queue and make sure emails are spam and virus free.

6. The outgoing email is stored in the “Sent Folder” of the storage layer.

7. SMTP outgoing workers send the email to the recipient mail server.

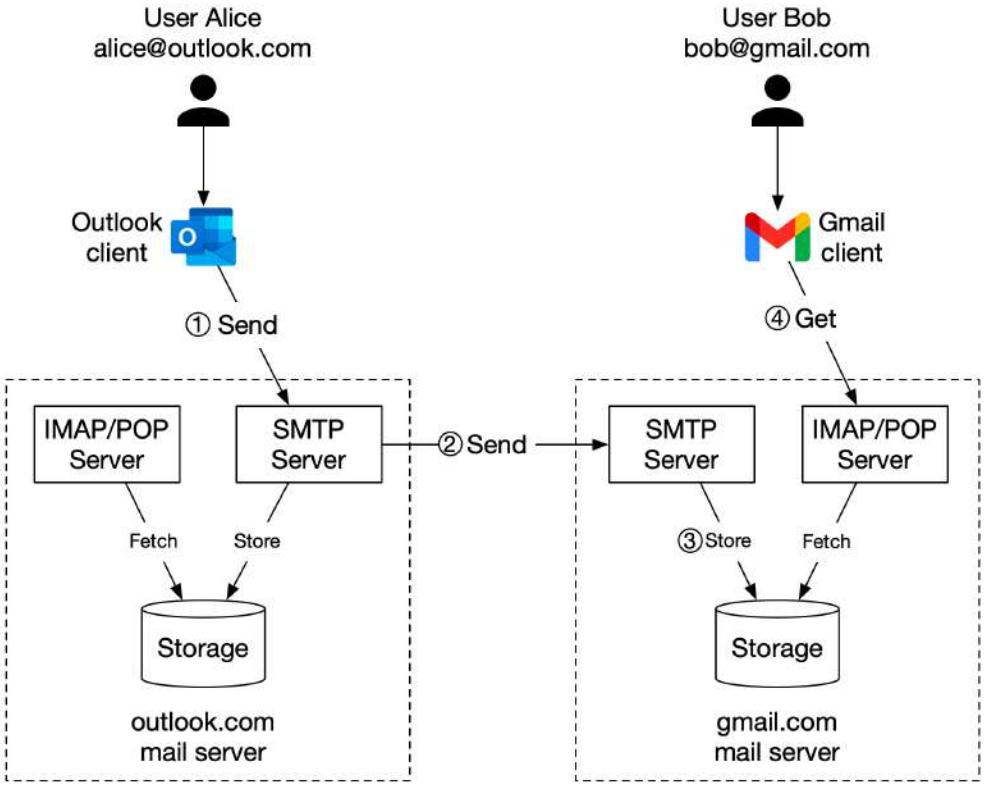
Each message in the outgoing queue contains all the metadata required to create an email. A distributed message queue is a critical component that allows asynchronous mail processing. By decoupling SMTP outgoing workers from the web servers, we can scale SMTP outgoing workers independently.

We monitor the size of the outgoing queue very closely. If there are many emails stuck in the queue, we need to analyze the cause of the issue. Here are some possibilities:
- The recipient’s mail server is unavailable. In this case, we need to retry sending the email at a later time. Exponential backoff might be a good retry strategy.

- Not enough consumers to send emails. In this case, we may need more consumers to reduce the processing time.

Interview Question: Design Gmail

One picture is worth more than a thousand words. In this post, we will take a look at what happens when Alice sends an email to Bob.



1. Alice logs in to her Outlook client, composes an email, and presses “send”. The email is sent to the Outlook mail server. The communication protocol between the Outlook client and mail server is SMTP.

2. Outlook mail server queries the DNS (not shown in the diagram) to find the address of the recipient’s SMTP server. In this case, it is Gmail’s SMTP server. Next, it transfers the email to the Gmail mail server. The communication protocol between the mail servers is SMTP.

3. The Gmail server stores the email and makes it available to Bob, the recipient.

4. Gmail client fetches new emails through the IMAP/POP server when Bob logs in to Gmail.

Please keep in mind this is a highly simplified design. Hope it sparks your interest and curiosity:) I'll explain each component in more depth in the future.

Map rendering

Google Maps Continued. Let's take a look at **Map Rendering** in this post.

Pre-Computed Tiles

One foundational concept in map rendering is tiling. Instead of rendering the entire map as one large custom image, the world is broken up into smaller tiles. The client only downloads the relevant tiles for the area the user is in and stitches them together like a mosaic for display. The tiles are pre-computed at different zoom levels. Google Maps uses 21 zoom levels.

For example, at zoom level 0, The entire map is represented by a single tile of size 256 * 256 pixels. Then at zoom level 1, the number of map tiles doubles in both north-south and east-west directions, while each tile stays at 256 * 256 pixels. So we have 4 tiles at zoom level 1, and the whole image of zoom level 1 is 512 * 512 pixels. With each increment, the entire set of tiles has 4x as many pixels as the previous level. The increased pixel count provides an increasing level of detail to the user.

This allows the client to render the map at the best granularities depending on the client's zoom level without consuming excessive bandwidth to download tiles with too much detail. This is especially important when we are loading the images from mobile clients.

Road Segments

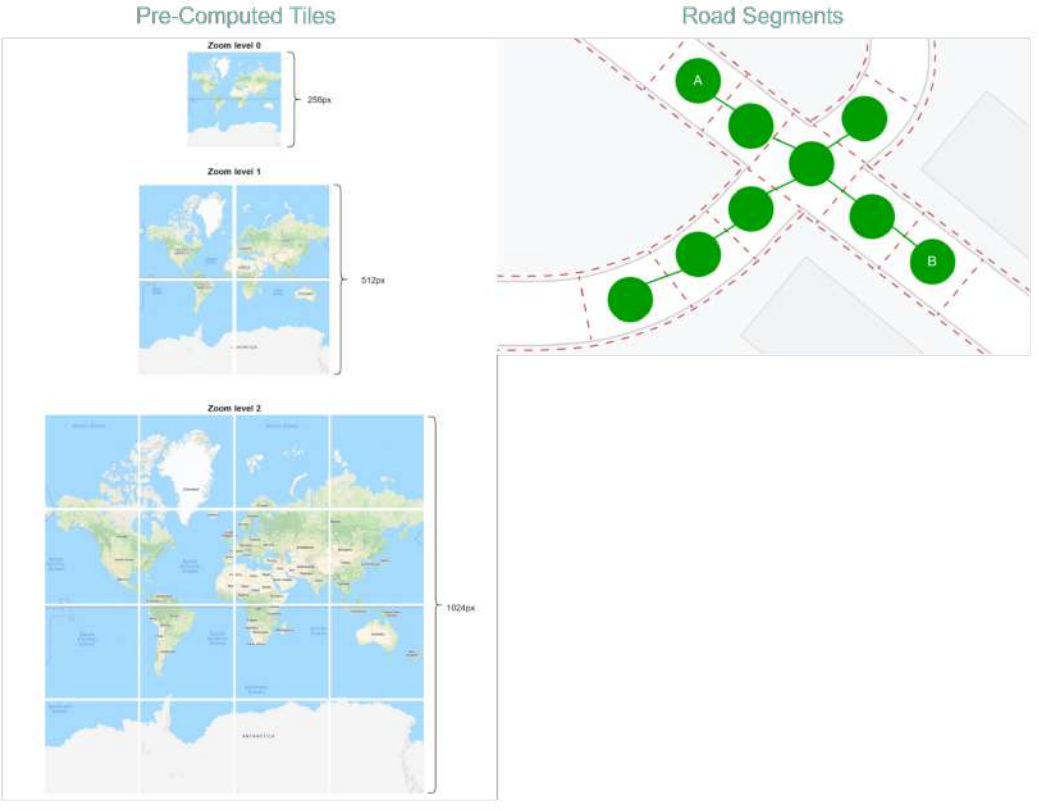
Now that we have transformed massive maps into tiles, we also need to define a data structure for the roads. We divide the world of roads into small blocks. We call these blocks road segments. Each road segment contains multiple roads, junctions, and other metadata.

We group nearby segments into super segments. This process can be applied repeatedly to meet the level of coverage required.

We then transform the road segments into a data structure that the navigation algorithms can use. The typical approach is to convert the map into a ***graph***, where the nodes are road segments, and two nodes are connected if the corresponding road segments are reachable

neighbors. In this way, finding a path between two locations becomes a shortest-path problem, where we can leverage Dijkstra or A* algorithms.

Google Maps - Map Rendering

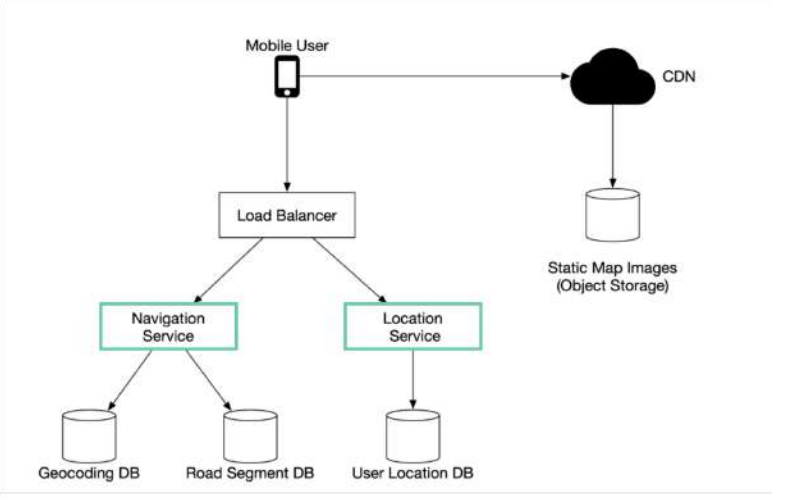


Interview Question: Design Google Maps

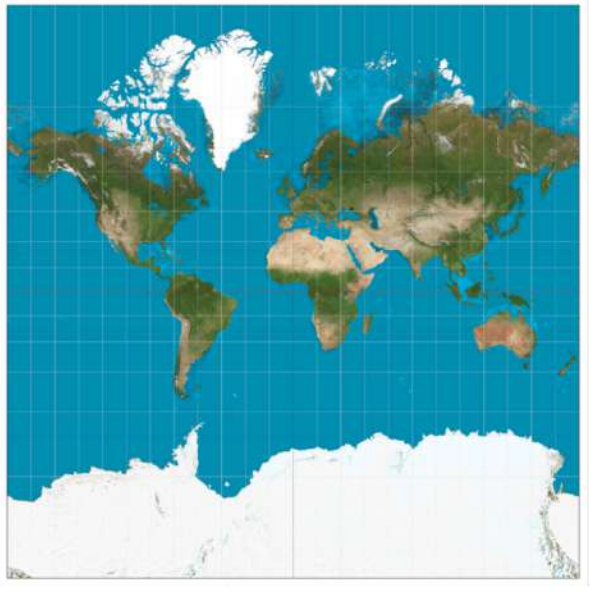
Google started project Google Maps in 2005. As of March 2021, Google Maps had one billion daily active users, 99% coverage of the world in 200 countries.

Although Google Maps is a very complex system, we can break it down into 3 high-level components. In this post, let's take a look at how to design a simplified Google Maps.

Google Maps



2D Map Projection



Location Service

The location service is responsible for recording a user's location update. The Google Map clients send location updates every few seconds. The user location data is used in many cases:

- detect new and recently closed roads
- improve the accuracy of the map over time
- used as an input for live traffic data.

Map Rendering

The world's map is projected into a huge 2D map image. It is broken down into small image blocks called "tiles" (see below). The tiles are static. They don't change very often. An efficient way to serve static tile files is with a CDN backed by cloud storage like S3. The users can load the necessary tiles to compose a map from nearby CDN.

What if a user is zooming and panning the map viewpoint on the client to explore their surroundings?

An efficient way is to pre-calculate the map blocks with different zoom levels and load the images when needed.

Navigation Service

This component is responsible for finding a reasonably fast route from point A to point B. It calls two services to help with the path calculation:

1. Geocoding Service: resolve the given address to a latitude/longitude pair

2. Route Planner Service: this service does three things in sequence:

    - Calculate the top-K shortest paths between A and B
    - Calculate the estimation of time for each path based on current traffic and historical data
    - Rank the paths by time predictions and user filtering. For example, the user doesn't want to avoid tolls.

Pull vs push models

There are two ways metrics data can be collected, pull or push. It is a routine debate as to which one is better and there is no clear answer. In this post, we will take a look at the pull model.

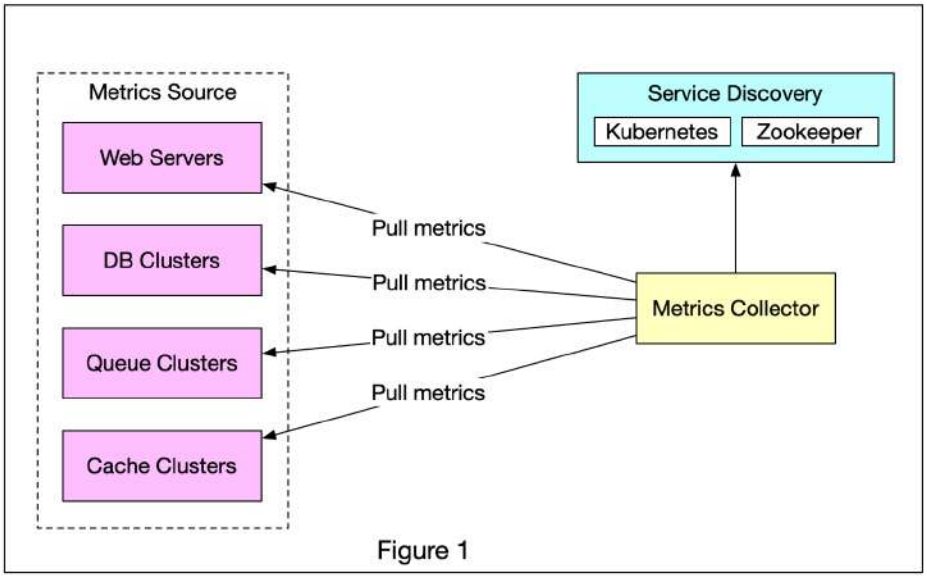


Figure 1

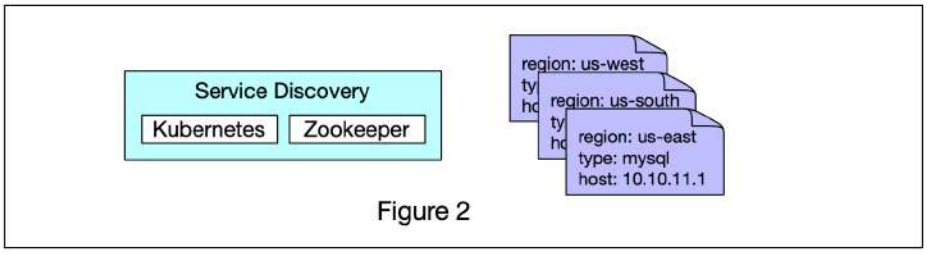


Figure 2

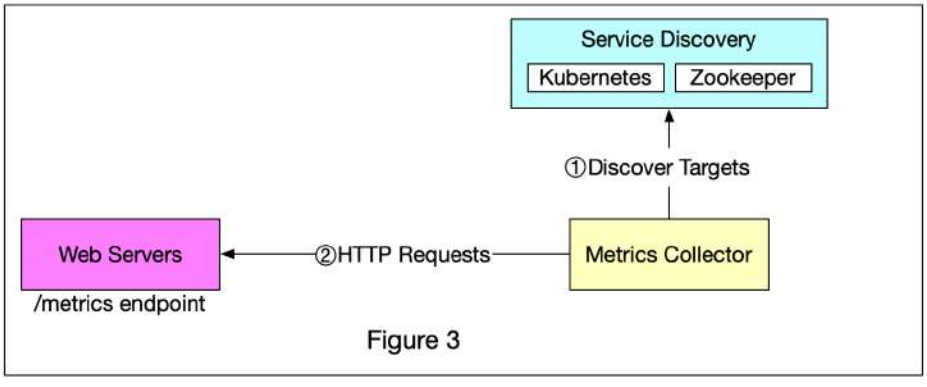


Figure 3

Figure 1 shows data collection with a pull model over HTTP. We have dedicated metric collectors which pull metrics values from the running applications periodically.

In this approach, the metrics collector needs to know the complete list of service endpoints to pull data from. One naive approach is to use a file to hold DNS/IP information for every service endpoint on the “metric collector” servers. While the idea is simple, this approach is hard to maintain in a large-scale environment where servers are added or removed frequently, and we want to ensure that metric collectors don’t miss out on collecting metrics from any new servers.

The good news is that we have a reliable, scalable, and maintainable solution available through Service Discovery, provided by Kubernetes, Zookeeper, etc., wherein services register their availability and the metrics collector can be notified by the Service Discovery component whenever the list of service endpoints changes. Service discovery contains configuration rules about when and where to collect metrics as shown in Figure 2.

Figure 3 explains the pull model in detail.

1 The metrics collector fetches configuration metadata of service endpoints from Service Discovery. Metadata include pulling interval, IP addresses, timeout and retries parameters, etc.

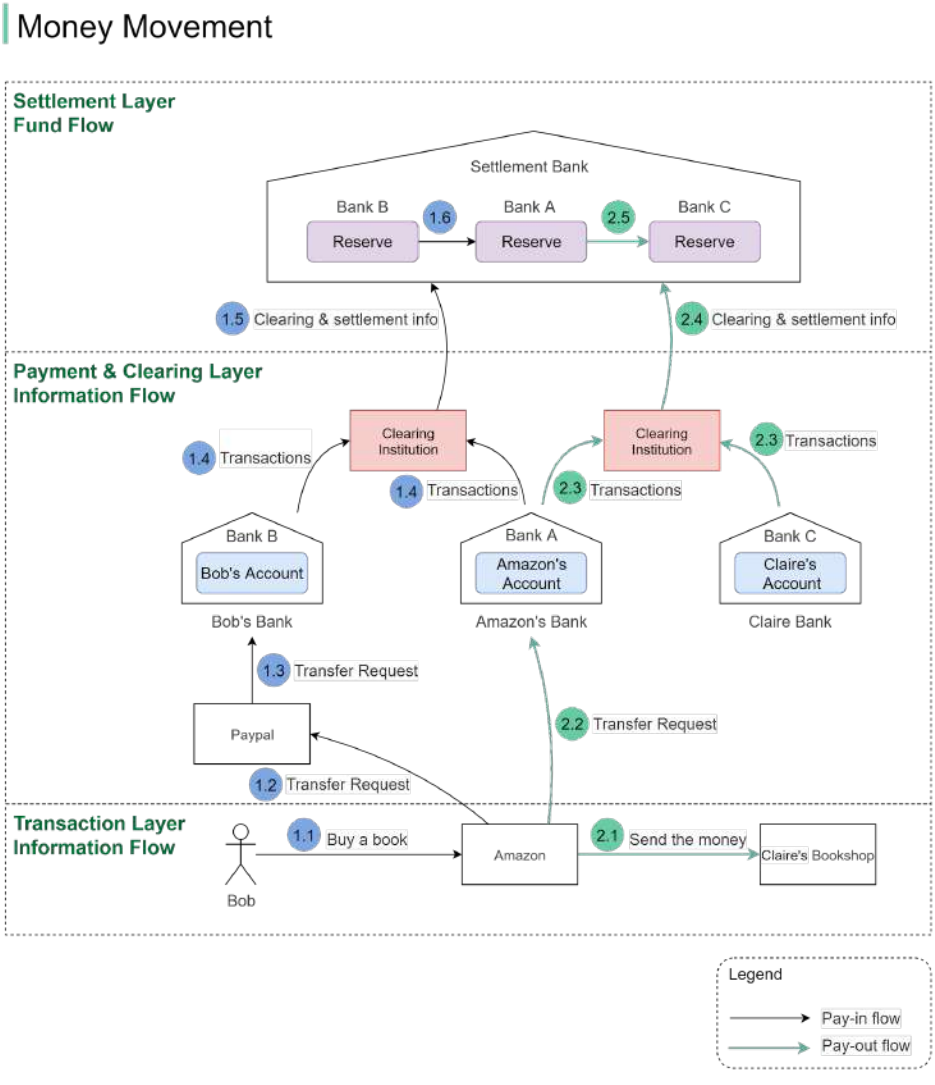
2 The metrics collector pulls metrics data via a pre-defined HTTP endpoint (for example, /metrics). To expose the endpoint, a client library usually needs to be added to the service. In Figure 3, the service is Web Servers.

3 Optionally, the metrics collector registers a change event notification with Service Discovery to receive an update whenever the service endpoints change. Alternatively, the metrics collector can poll for endpoint changes periodically.

Money movement

One picture is worth more than a thousand words. This is what happens when you buy a product using Paypal/bank card under the hood.

To understand this, we need to digest two concepts: **clearing** & **settlement**. Clearing is a process that calculates who should pay whom with how much money; while settlement is a process where real money moves between reserves in the settlement bank.



Let’s say Bob wants to buy an SDI book from Claire’s shop on Amazon.

- Pay-in flow (Bob pays Amazon money):
1.1 Bob buys a book on Amazon using Paypal.
1.2 Amazon issues a money transfer request to Paypal.
1.3 Since the payment token of Bob’s debit card is stored in Paypal, Paypal can transfer money, on Bob’s behalf, to Amazon’s bank account in Bank A.
1.4 Both Bank A and Bank B send transaction statements to the clearing institution. It reduces the transactions that need to be settled. Let’s assume Bank A owns Bank B $100 and Bank B owns bank A $500 at the end of the day. When they settle, the net position is that Bank B pays Bank A $400.
1.5 & 1.6 The clearing institution sends clearing and settlement information to the settlement bank. Both Bank A and Bank B have pre-deposited funds in the settlement bank as money reserves, so actual money movement happens between two reserve accounts in the settlement bank.

- Pay-out flow (Amazon pays the money to the seller: Claire):
2.1 Amazon informs the seller (Claire) that she will get paid soon.
2.2 Amazon issues a money transfer request from its own bank (Bank A) to the seller bank (bank C). Here both banks record the transactions, but no real money is moved.
2.3 Both Bank A and Bank C send transaction statements to the clearing institution.
2.4 & 2.5 The clearing institution sends clearing and settlement information to the settlement bank. Money is transferred from Bank A’s reserve to Bank C’s reserve.

Notice that we have three layers:
- Transaction layer: where the online purchases happen
- Payment and clearing layer: where the payment instructions and transaction netting happen
- Settlement layer: where the actual money movement happen

The first two layers are called information flow, and the settlement layer is called fund flow.

You can see the **information flow and fund flow are separated**. In the info flow, the money seems to be deducted from one bank account and added to another bank account, but the actual money movement happens in the settlement bank at the end of the day.

Because of the asynchronous nature of the info flow and the fund flow, reconciliation is very important for data consistency in the systems along with the flow.

It makes things even more interesting when Bob wants to buy a book in the Indian market, where Bob pays USD but the seller can only receive INR.

Reconciliation

My previous post about painful payment reconciliation problems sparked lots of interesting discussions. One of the readers shared more problems we may face when working with intermediary payment processors in the trenches and a potential solution:

1. Foreign Currency Problem: When you operate a store globally, you will come across this problem quite frequently. To go back to the example from Paypal - if the transaction happens in a currency different from the standard currency of Paypal, this will create another layer, where the transaction is first received in that currency and exchanged to whatever currency your Paypal is using. There needs to be a reliable way to reconcile that currency exchange transaction. It certainly does not help that every payment provider handles this differently.

2. Payment providers are only that - intermediaries. Each purchase does not trigger two events for a company, but actually at least 4. The purchase via Paypal (where both the time and the currency dimension can come into play) trigger the debit/credit pair for the transaction and then, usually a few days later, another pair when the money is transferred from Paypal to a bank account (where there might be yet another FX discrepancy to reconcile if, for example, the initial purchase was in JPY, Paypal is set up in USD and your bank account is in EUR). There needs to be a way to reconcile all of these.

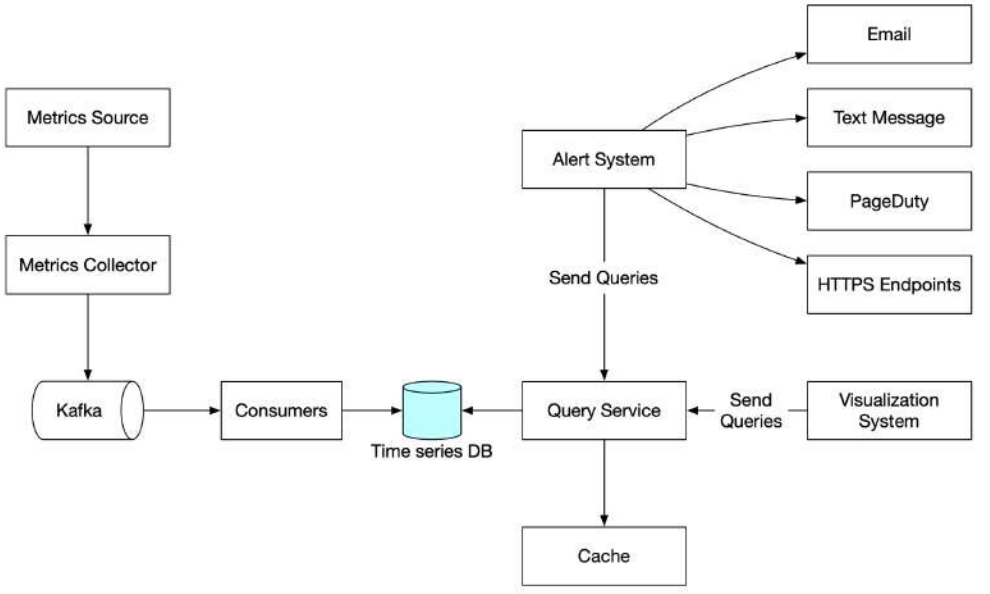
3. Some problems also pop up on the buyer side that is very platform-specific. One example is shadow transaction from Paypal: if you buy two items on Paypal with 1 week of time between the two transactions, Paypal will first debit money from your bank account for transaction A. If at the time of transaction B, transaction A has not gone through completely or is canceled, there might be a world where Paypal will use the money from transaction A to partially pay for transaction B, which leads to only a partial amount of transaction B being withdrawn from the bank account.

In practice, this usually looks something like this:
1) Your shop assigns an order number to the purchase

2) The order number is carried over to the payment provider
3) The payment provider creates another internal ID, which is carried over across transactions within the system
4) The payment ID is used when you get the payout on your bank account (or the payment provider bundles individual payments, which can be reconciled within the payment provider system)
5) Ideally, your payment provider and your shop have an integration/API with the tool you use to (hopefully automatically) create invoices. This usually carries over the order id from the shop (closing the loop) and sometimes even the payment id to match it with the invoice id, which you then can use to reconcile it with your accounts receivable/payable. :)

Credit: A knowledgeable reader who prefers to stay private. Thank you!

Continued: how to choose the right database for metrics collecting service?



There are many storage systems available that are optimized for time-series data. The optimization lets us use far fewer servers to handle the same volume of data. Many of these databases also have custom query interfaces specially designed for the analysis of time-series data that are much easier to use than SQL. Some even provide features to manage data retention and data aggregation. Here are a few examples of time-series databases.

OpenTSDB is a distributed time-series database, but since it is based on Hadoop and HBase, running a Hadoop/HBase cluster adds complexity. Twitter uses MetricsDB, and Amazon offers Timestream as a time-series database. According to DB-engines, the two most popular time-series databases are InfluxDB and Prometheus, which are designed to store large volumes of time-series data and quickly perform real-time analysis on that data. Both of them primarily rely on an in-memory cache and on-disk storage. And they both handle durability and performance quite well. According to the benchmark, an InfluxDB with 8 cores and 32GB RAM can handle over 250,000 writes per second.

Since a time-series database is a specialized database, you are not expected to understand the internals in an interview unless you explicitly mentioned it in your resume. For the purpose of an interview, it's important to understand the metrics data are time-series in nature and we can select time-series databases such as InfluxDB for storage to store them.

Another feature of a strong time-series database is efficient aggregation and analysis of a large amount of time-series data by labels, also known as tags in some databases. For example, InfluxDB builds indexes on labels to facilitate the fast lookup of time-series by labels. It provides clear best-practice guidelines on how to use labels, without overloading the database. The key is to make sure each label is of low cardinality (having a small set of possible values). This feature is critical for visualization, and it would take a lot of effort to build this with a general-purpose database.

Which database shall I use for the metrics collecting system?

This is one of the most important questions we need to address in an interview.

Data access pattern
As shown in the diagram, each label on the y-axis represents a time series (uniquely identified by the names and labels) while the x-axis represents time.

The write load is heavy. As you can see, there can be many time-series data points written at any moment. There are millions of operational metrics written per day, and many metrics are collected at high frequency, so the traffic is undoubtedly write-heavy.

At the same time, the read load is spiky. Both visualization and alert services send queries to the database and depending on the access patterns of the graphs and alerts, the read volume could be bursty.

Choose the right database
The data storage system is the heart of the design. It's not recommended to build your own storage system or use a general-purpose storage system (MySQL) for this job.

A general-purpose database, in theory, could support time-series data, but it would require expert-level tuning to make it work at our scale. Specifically, a relational database is not optimized for operations you would commonly perform against time-series data. For example, computing the moving average in a rolling time window requires complicated SQL that is difficult to read (there is an example of this in the deep dive section). Besides, to support tagging/labeling data, we need to add an index for each tag. Moreover, a general-purpose relational database does not perform well under constant heavy write load. At our scale, we would need to expend significant effort in tuning the database, and even then, it might not perform well.

How about NoSQL? In theory, a few NoSQL databases on the market could handle time-series data effectively. For example, Cassandra and Bigtable can both be used for time series data. However, this would require deep knowledge of the internal workings of each NoSQL to devise a scalable schema for effectively storing and querying time-series data. With industrial-scale time-series databases readily available, using a general purpose NoSQL database is not appealing.

There are many storage systems available that are optimized for time-series data. The optimization lets us use far fewer servers to handle the same volume of data. Many of these databases also have custom query interfaces specially designed for the analysis of time-series data that are much easier to use than SQL. Some even provide features to manage data retention and data aggregation. Here are a few examples of time-series databases.

OpenTSDB is a distributed time-series database, but since it is based on Hadoop and HBase, running a Hadoop/HBase cluster adds complexity. Twitter uses MetricsDB, and Amazon offers Timestream as a time-series database. According to DB-engines, the two most popular time-series databases are InfluxDB and Prometheus, which are designed to store large volumes of time-series data and quickly perform real-time analysis on that data. Both of them primarily rely on an in-memory cache and on-disk storage. And they both handle durability and performance quite well. According to the benchmark listed on InfluxDB website, a DB server with 8 cores and 32GB RAM can handle over 250,000 writes per second.

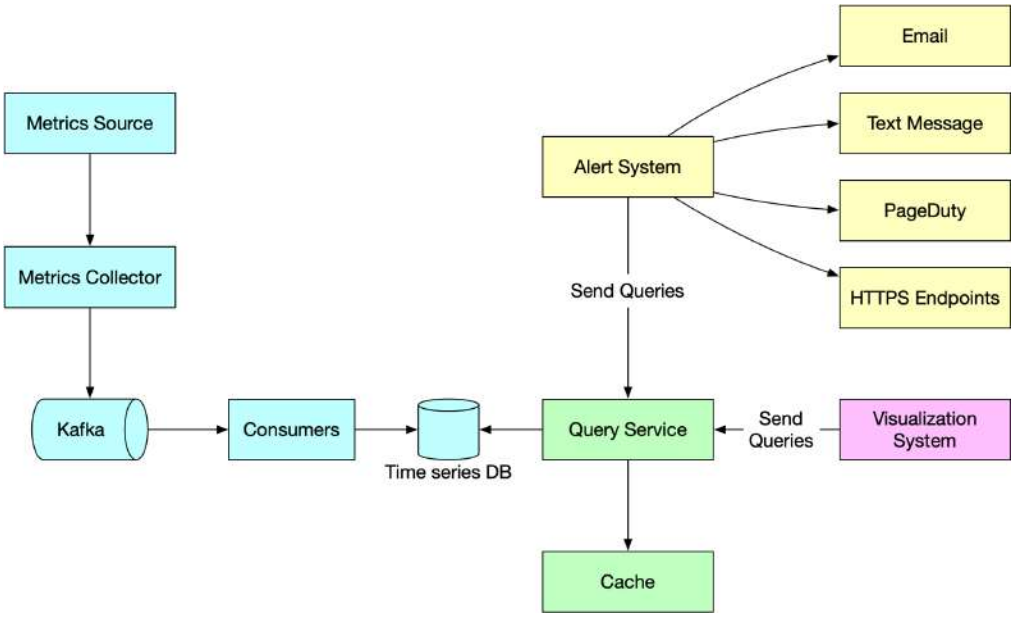
Since a time-series database is a specialized database, you are not expected to understand the internals in an interview unless you explicitly mentioned it in your resume. For the purpose of an interview, it’s important to understand the metrics data are time-series in nature and we can select time-series databases such as InfluxDB for storage to store them.

Another feature of a strong time-series database is efficient aggregation and analysis of a large amount of time-series data by labels, also known as tags in some databases. For example, InfluxDB builds indexes on labels to facilitate the fast lookup of time-series by

labels. It provides clear best-practice guidelines on how to use labels, without overloading the database. The key is to make sure each label is of low cardinality (having a small set of possible values). This feature is critical for visualization, and it would take a lot of effort to build this with a general-purpose database.

Metrics monitoring and altering system

A well-designed **metrics monitoring** and alerting system plays a key role in providing clear visibility into the health of the infrastructure to ensure high availability and reliability. The diagram below explains how it works at a high level.



Metrics source: This can be application servers, SQL databases, message queues, etc.

Metrics collector: It gathers metrics data and writes data into the time-series database.

Time-series database: This stores metrics data as time series. It usually provides a custom query interface for analyzing and summarizing a large amount of time-series data. It maintains indexes on labels to facilitate the fast lookup of time-series data by labels.

Kafka: Kafka is used as a highly reliable and scalable distributed messaging platform. It decouples the data collection and data processing services from each other.

Consumers: Consumers or streaming processing services such as Apache Storm, Flink and Spark, process and push data to the time-series database.

Query service: The query service makes it easy to query and retrieve data from the time-series database. This should be a very thin wrapper if we choose a good time-series database. It could also be entirely replaced by the time-series database's own query interface.

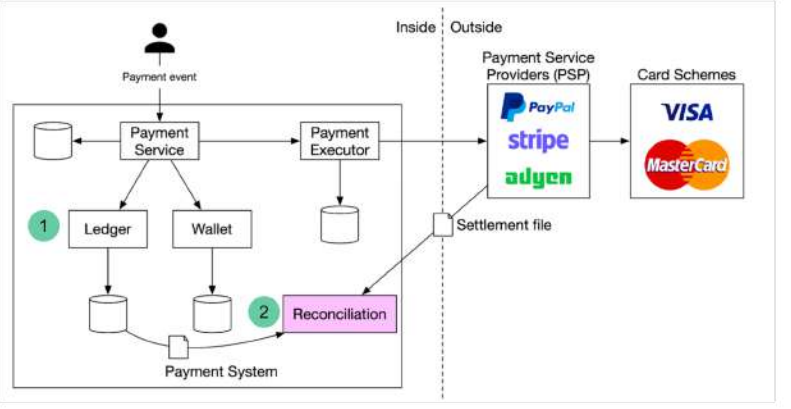
Alerting system: This sends alert notifications to various alerting destinations.

Visualization system: This shows metrics in the form of various graphs/charts.

Reconciliation

Reconciliation might be the most painful process in a payment system. It is the process of comparing records in different systems to make sure the amounts match each other.

Reconciliation in Payment System



Double-entry Bookkeeping in Ledger

Account	Debit	Credit
buyer	$200	
seller		$200

For example, if you pay $200 to buy a watch with Paypal:
- The eCommerce website should have a record about the purchase order of $200.
- There should be a transaction record of $200 in Paypal (marked with 2 in the diagram).
- The Ledger should record a debit of $200 dollars for the buyer, and a credit of $200 for the seller. This is called double-entry bookkeeping (see the table below).

Let's take a look at some pain points and how we can address them:

Problem 1: Data normalization. When comparing records in different systems, they come in different formats. For example, the timestamp can be “2022/01/01” in one system and “Jan 1, 2022” in another.
Possible solution: we can add a layer to transform different formats into the same format.

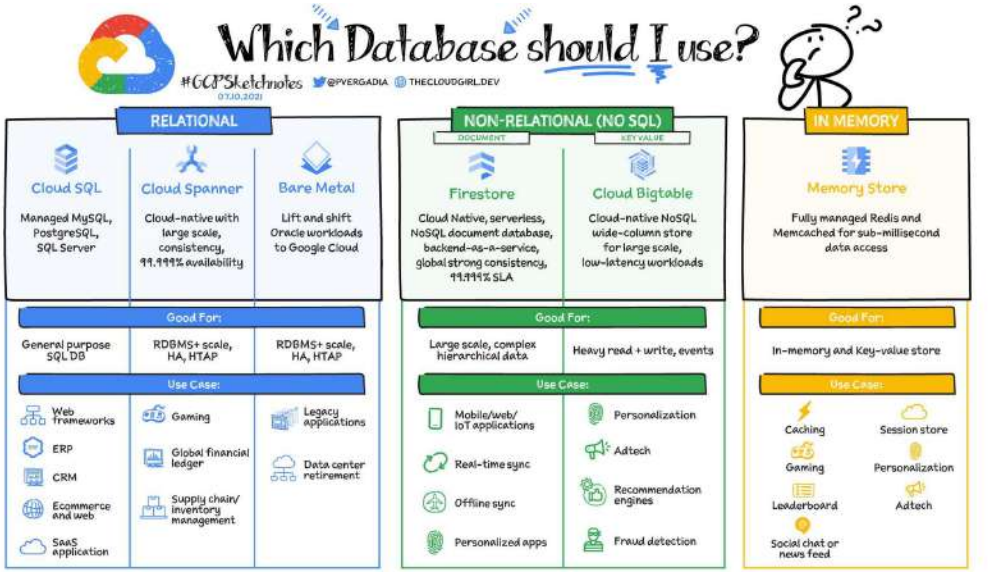
Problem 2: Massive data volume
Possible solution: we can use big data processing techniques to speed up data comparisons. If we need near real-time reconciliation, a streaming platform such as Flink is used; otherwise, end-of-day batch processing such as Hadoop is enough.

Problem 3: Cut-off time issue. For example, if we choose 00:00:00 as the daily cut-off time, one record is stamped with 23:59:55 in the internal system, but might be stamped 00:00:30 in the external system (Paypal), which is the next day. In this case, we couldn’t find this record in today’s Paypal records. It causes a discrepancy.
Possible solution: we need to categorize this break as a “temporary break” and run it later against the next day’s Paypal records. If we find a match in the next day’s Paypal records, the break is cleared, and no more action is needed.

You may argue that if we have exactly-once semantics in the system, there shouldn’t be any discrepancies. But the truth is, there are so many places that can go wrong. Having a reconciliation system is always necessary. It is like having a safety net to keep you sleeping well at night.

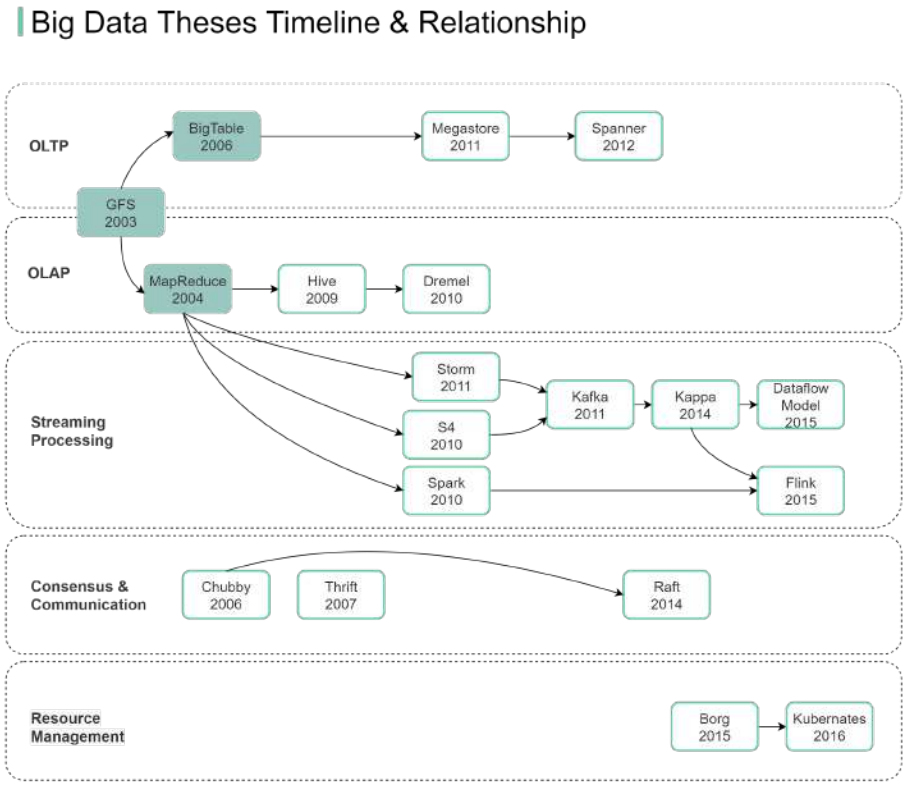
Which database shall I use? This is one of the most important questions we usually need to address in an interview.

Choosing the right database is hard. Google Cloud recently posted a great article that summarized different database options available in Google Cloud and explained which use cases are best suited for each database option.



Big data papers

Below is a timeline of important big data papers and how the techniques evolved over time.



The green highlighted boxes are the famous 3 Google papers, which established the foundation of the big data framework. At the high-level:

Big Data Techniques = *Massive data* + *Massive calculation*

Let's look at the **OLTP** evolution. BigTable provided a distributed storage system for structured data but dropped some characteristics of relational DB. Then Megastore brought back schema and simple transactions; Spanner brought back data consistency.

Now let's look at the **OLAP** evolution. MapReduce was not easy to program, so Hive solved this by introducing a SQL-like query

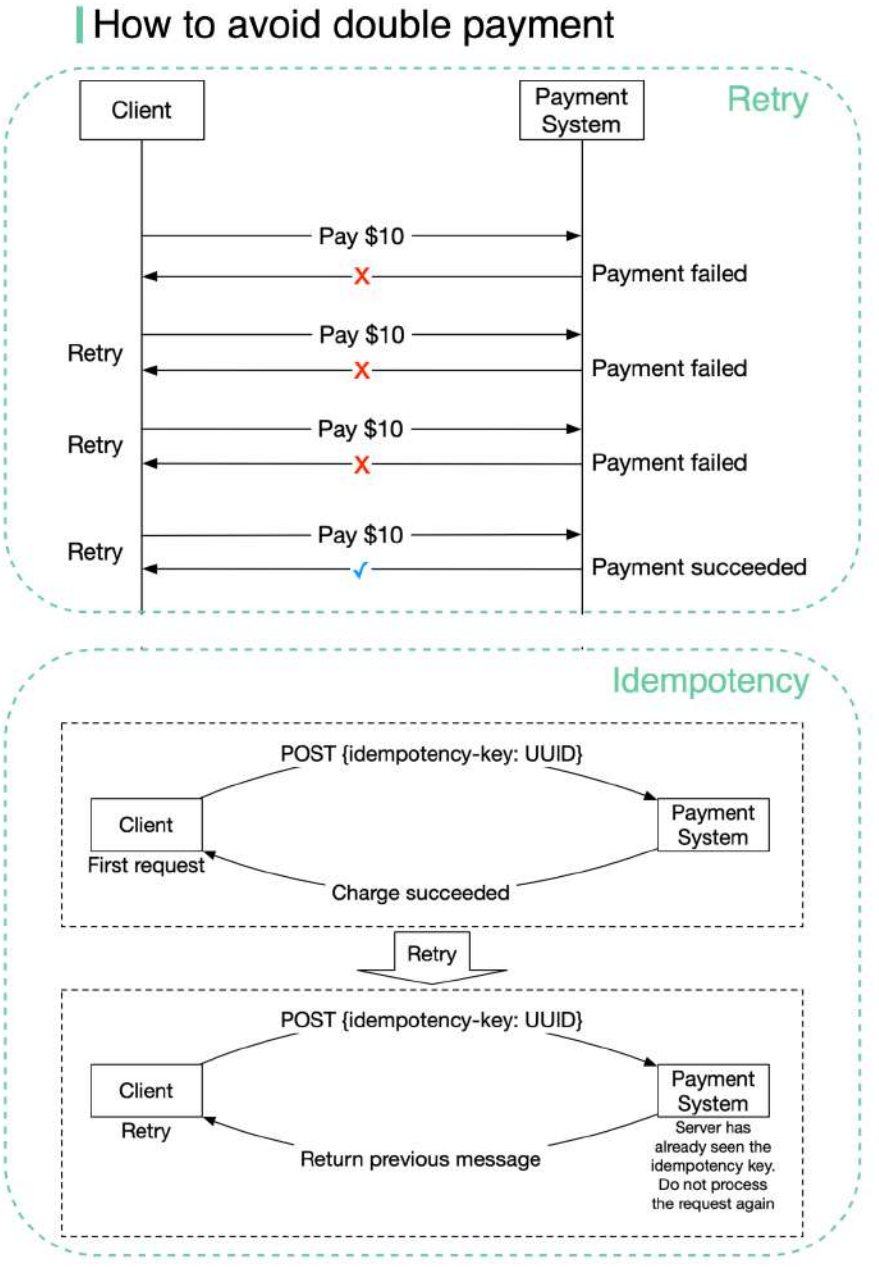
language. But Hive still used MapReduce under the hood, so it's not very responsive. In 2010, Dremel provided an interactive query engine.

Streaming processing was born to further solve the latency issue in OLAP. The famous ***lambda*** architecture was based on Storm and MapReduce, where streaming processing and batch processing have different processing flows. Then people started to build streaming processing with apache Kafka. ***Kappa*** architecture was proposed in 2014, where streaming and batching processings were merged into one flow. Google published The Dataflow Model in 2015, which was an abstraction standard for streaming processing, and Flink implemented this model.

To manage a big crowd of commodity server resources, we need resource management Kubernetes.

Avoid double charge

One of the most serious problems a payment system can have is to **double charge a customer**. When we design the payment system, it is important to guarantee that the payment system executes a payment order exactly-once.



At the first glance, exactly-once delivery seems very hard to tackle, but if we divide the problem into two parts, it is much easier to solve. Mathematically, an operation is executed exactly-once if:

1. It is executed at least once.

2. At the same time, it is executed at most once.

We now explain how to implement at least once using retry and at most once using idempotency check.

Retry

Occasionally, we need to retry a payment transaction due to network errors or timeout. Retry provides the at-least-once guarantee. For example, as shown in Figure 10, the client tries to make a $10 payment, but the payment keeps failing due to a poor network connection. Considering the network condition might get better, the client retries the request and this payment finally succeeds at the fourth attempt.

Idempotency

From an API standpoint, idempotency means clients can make the same call repeatedly and produce the same result.

For communication between clients (web and mobile applications) and servers, an idempotency key is usually a unique value that is generated by clients and expires after a certain period of time. A UUID is commonly used as an idempotency key and it is recommended by many tech companies such as Stripe and PayPal. To perform an idempotent payment request, an idempotency key is added to the HTTP header: *<idempotency-key: key_value>*.

Payment security

A few weeks ago, I posted the high-level design for the payment system. Today, I'll continue the discussion and focus on payment security.

The table below summarizes techniques that are commonly used in payment security. If you have any questions or I missed anything, please leave a comment.

Problem	Solution
Request/response eavesdropping	Use HTTPS
Data tampering	Enforce encryption and integrity monitoring
Man-in-the-middle attack	Use SSL and authentication certificates
Data loss	Database replication across multiple regions and take snapshot of data
Distributed denial-of-service attack (DDoS)	Rate limiting and firewall
Card theft	Tokenization. Instead of using real card numbers, tokens are stored and used for payment
PCI compliance	PCI DSS is an information security standard for organizations that handle branded credit cards
Fraud	Address verification, card verification value (CVV), user behavior analysis, etc.

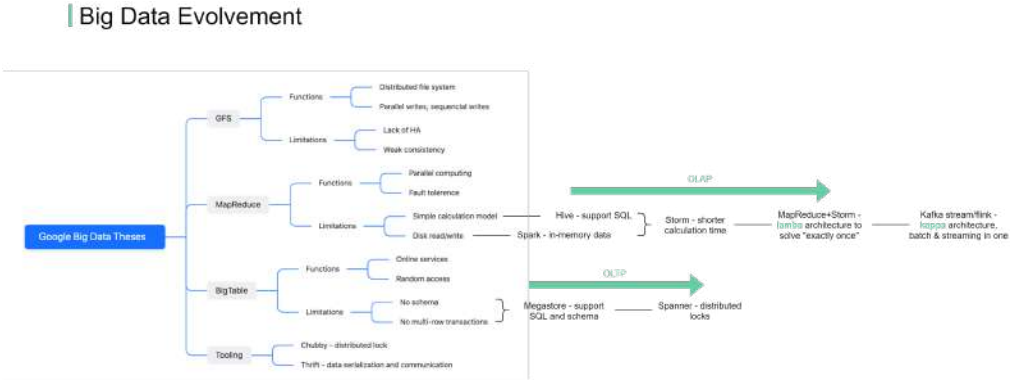
System Design Interview Tip

One pro tip for acing a system design interview is to read the engineering blog of the company you are interviewing with. You can get a good sense of what technology they use, why the technology was chosen over others, and learn what issues are important to engineers.

For example, here are 4 blog posts Twitter Engineering recommends:
1. The Infrastructure Behind Twitter: Scale
2. Discovery and Consumption of Analytics Data at Twitter
3. The what and why of product experimentation at Twitter
4. Twitter experimentation: technical overview

Big data evolvement

I hope everyone has a great time with friends and family during the holidays. If you are looking for some readings, classic engineering papers are a good start.



A lot of times when we are busy with work, we only focus on scattered information, telling us “**how**” and “what” to get our immediate needs to get things done.

However, reading the classics helps us know “**why**” behind the scenes, and teaches us how to solve problems, make better decisions, or even contribute to open source projects.

Let’s take **big data** as an example.

Big data area has progressed a lot over the past 20 years. It started from 3 Google papers (see the links in the comment), which tackled real engineering challenges at Google scale:

- GFS (2003) - big data storage
- MapReduce (2004) - calculation model
- BigTable (2006) - online services

The diagram below shows the functionalities and limitations of the 3 techniques, and how they evolve over time into two streams: OLTP and OLAP. Each evolved product was trying to solve the limitations of the

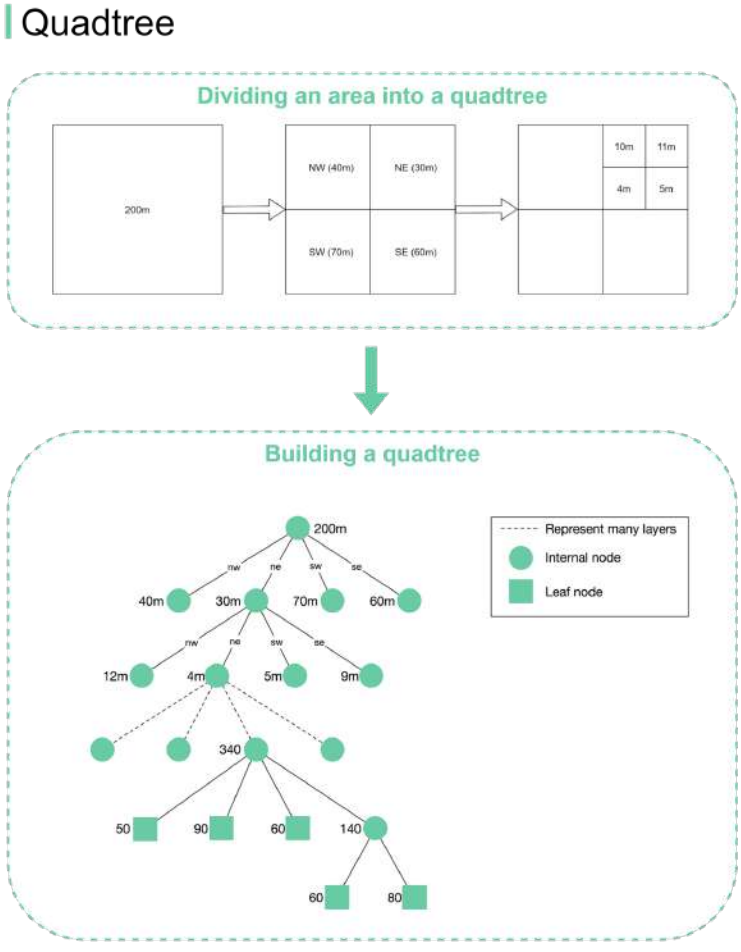
last generation. For example, “Hive - support SQL” means Hive was trying to solve the lack of SQL in MapReduce.

If you want to learn more, you can refer to the papers for details. What other classics would you recommend?

Quadtree

In this post, let's explore another data structure to find nearby restaurants on Yelp or Google Maps.

A quadtree is a data structure that is commonly used to partition a two-dimensional space by recursively subdividing it into four quadrants (grids) until the contents of the grids meet certain criteria (see the first diagram).



Quadtree is an **in-memory data structure** and it is not a database solution. It runs on each LBS (Location-Based Service, see last week's post) server, and the data structure is built at server start-up time.

The second diagram explains the quadtree building process in more detail. The root node represents the whole world map. The root node is **recursively** broken down into 4 quadrants until no nodes are left with more than 100 businesses.

How to get nearby businesses with quadtree?
- Build the quadtree in memory.

- After the quadtree is built, start searching from the root and traverse the tree, until we find the leaf node where the search origin is.

- If that leaf node has 100 businesses, return the node. Otherwise, add businesses from its neighbors until enough businesses are returned.

Update LBS server and rebuild quadtree
- It may take a few minutes to build a quadtree in memory with 200 million businesses at the server start-up time.

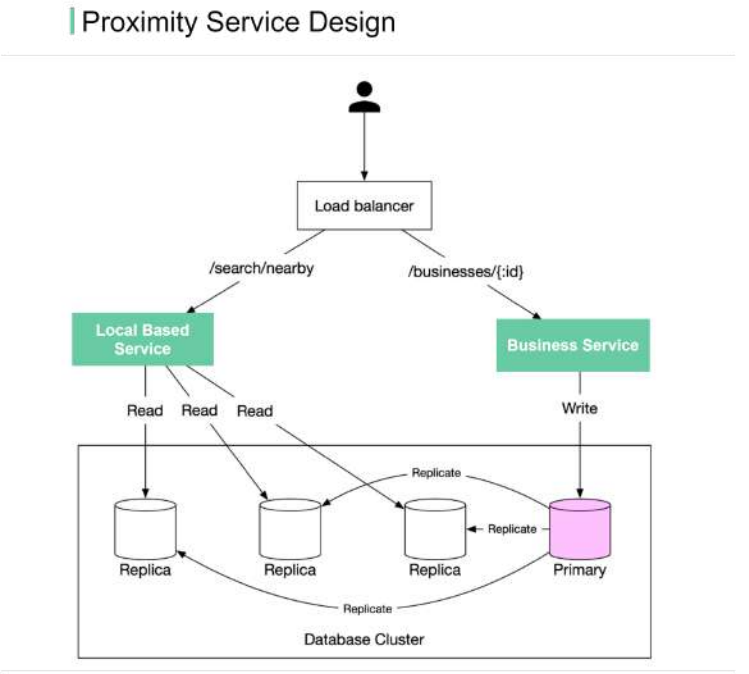
- While the quadtree is being built, the server cannot serve traffic.

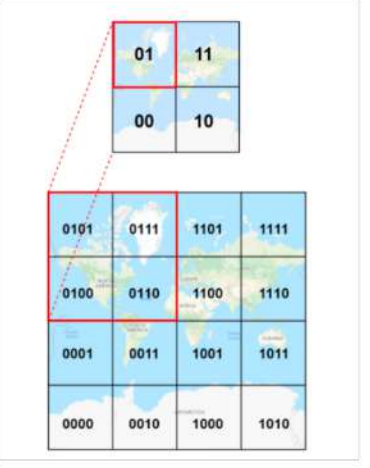
- Therefore, we should roll out a new release of the server incrementally to **a small subset** of servers at a time. This avoids taking a large swathe of the server cluster offline and causes service brownout.

How do we find nearby restaurants on Yelp?

Here are some design details behind the scenes.

There are two key services (see the diagram below):





- Business Service

- Add/delete/update restaurant information
- Customers view restaurant details

- Local-based Service (LBS)

- Given a radius and location, return a list of nearby restaurants

How are the restaurant locations stored in the database so that LBS can return nearby restaurants efficiently?

Store the latitude and longitude of restaurants in the database? The query will be very inefficient when you need to calculate the distance between you and every restaurant.

One way to speed up the search is using the **geohash algorithm**.

First, divide the planet into four quadrants along with the prime meridian and equator：

- Latitude range [-90, 0] is represented by 0

- Latitude range [0, 90] is represented by 1

- Longitude range [-180, 0] is represented by 0

- Longitude range [0, 180] is represented by 1

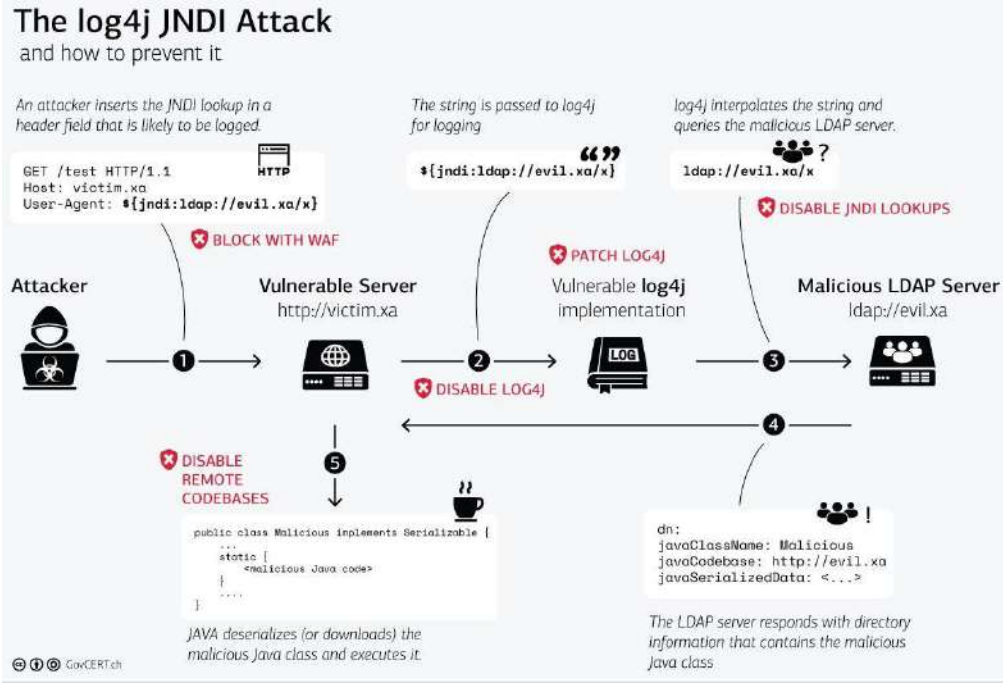
Second, divide each grid into four smaller grids. Each grid can be represented by alternating between longitude bit and latitude bit.

So when you want to search for the nearby restaurants in the red-highlighted block, you can write SQL like:

```
SELECT * FROM geohash_index WHERE geohash LIKE `01%`
```

Geohash has some limitations. There can be a lot of restaurants in one block (downtown New York), but none in another block (ocean). So there are other more complicated algorithms to optimize the process. Let me know if you are interested in the details.

One picture is worth more than a thousand words. Log4j from attack to prevention in one illustration.



Credit GovCERT

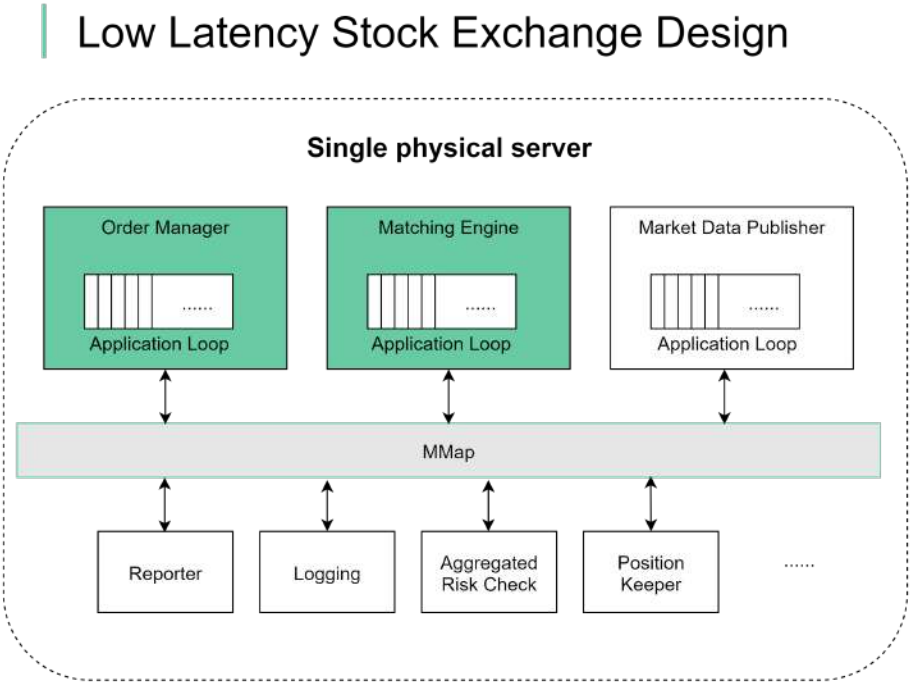
Link:
https://www.govcert.ch/blog/zero-day-exploit-targeting-popular-java-library-log4j/

How does a modern stock exchange achieve microsecond latency?

The principal is:

Do less on the critical path !

- Fewer tasks on the critical path
- Less time on each task
- Fewer network hops
- Less disk usage



For the stock exchange, the critical path is:
- **start**: an order comes into the order manager
- mandatory risk checks
- the order gets matched and the execution is sent back
- **end**: the execution comes out of the order manager

Other non-critical tasks should be removed from the critical path.

We put together a design as shown in the diagram:

- deploy all the components in a single giant server (no containers)

- use shared memory as an event bus to communicate among the components, no hard disk

- key components like Order Manager and Matching Engine are single-threaded on the critical path, and each pinned to a CPU so that there is **no context switch** and **no locks**

- the single-threaded application loop executes tasks one by one in sequence

- other components listen on the event bus and react accordingly

Match buy and sell orders

Stocks go up and down. Do you know what data structure is used to efficiently match buy and sell orders?

Buy 2700 shares: 2700 - 200 - 400 - 1100 - 100 - 900 = 0

Stock exchanges use the data structure called **order books**. An order book is an electronic list of buy and sell orders, organized by price levels. It has a buy book and a sell book, where each side of the book contains a bunch of price levels, and each price level contains a list of orders (first in first out).

The image is an example of price levels and the queued quantity in each price level.

So what happens when you place a market order to buy 2700 shares in the diagram?

- The buy order is matched with all the sell onrders at price 100.10, and the first order at price 100.11 (illustrated in light red).

- Now because of the big buy order which “eats up” the first price level on the sell book, the best ask price goes up from 100.10 to 100.11.

- So when the market is bullish, people tend to buy stocks, and the price goes up and up.

An efficient data structure for an order book must satisfy:

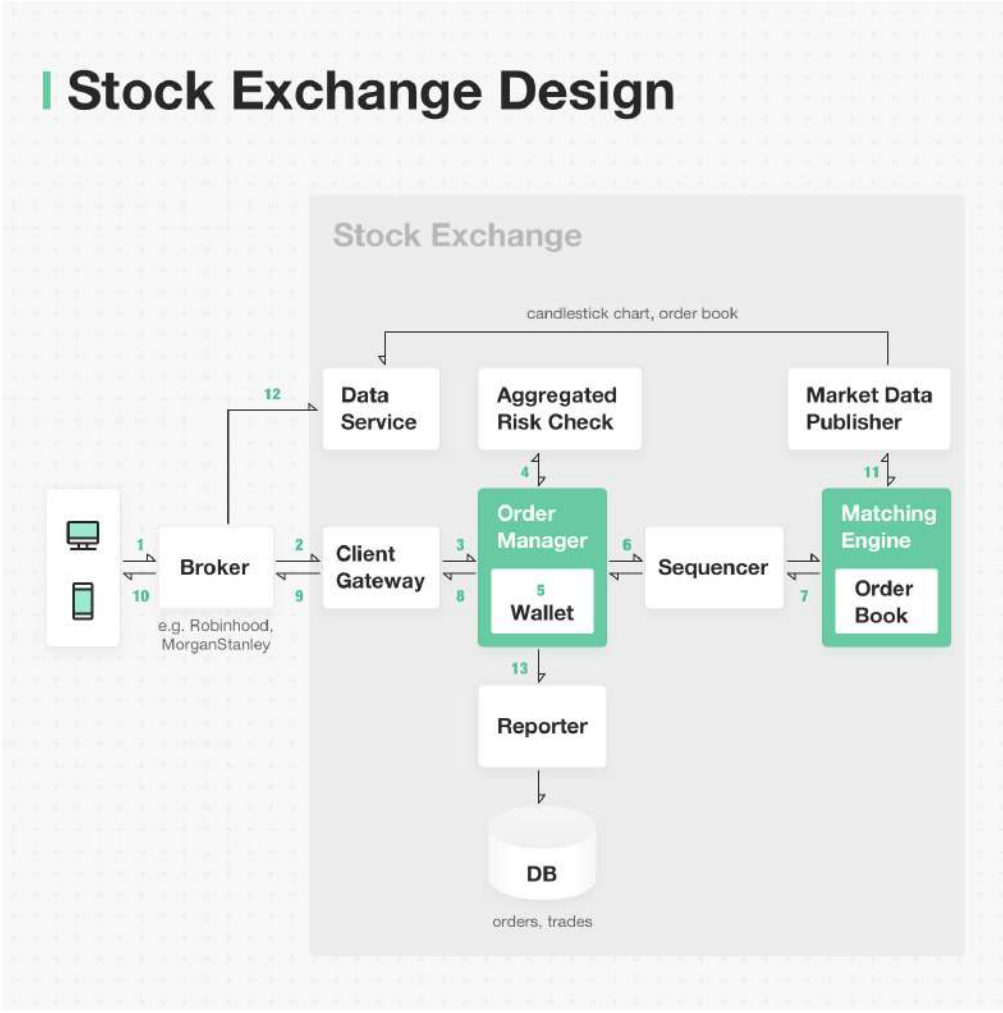
- Constant lookup time. Operations include: get volume at a price level or between price levels, query best bid/ask.

- Fast add/cancel/execute/update operations, preferably O(1) time complexity. Operations include: place a new order, cancel an order, and match an order.

Stock exchange design

The stock market has been volatile recently.

Coincidentally, we just finished a new chapter “Design a stock exchange”. I’ll use plain English to explain what happens when you place a stock buying order. The focus is on the exchange side.



Step 1: client places an order via the broker’s web or mobile app.

Step 2: broker sends the order to the exchange.

Step 3: the exchange client gateway performs operations such as validation, rate limiting, authentication, normalization, etc, and sends the order to the order manager.

Step 4: the order manager performs risk checks based on rules set by the risk manager.

Step 5: once risk checks pass, the order manager checks if there is enough balance in the wallet.

Step 6-7: the order is sent to the matching engine. The matching engine sends back the execution result if a match is found. Both order and execution results need to be sequenced first in the sequencer so that matching determinism is guaranteed.

Step 8 - 10: execution result is passed all the way back to the client.

Step 11-12: market data (including the candlestick chart and order book) are sent to the data service for consolidation. Brokers query the data service to get the market data.

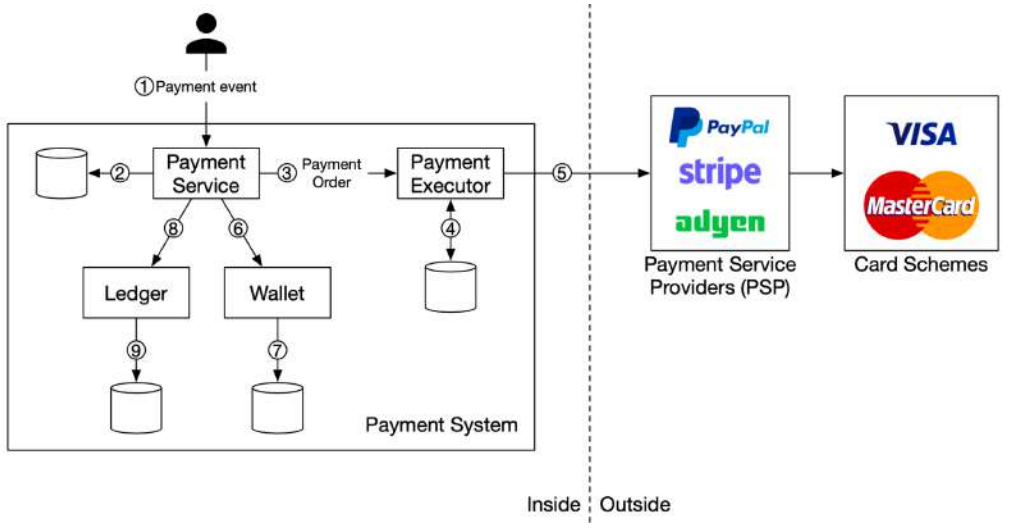
Step 13: the reporter composes all the necessary reporting fields (e.g. client_id, price, quantity, order_type, filled_quantity, remaining_quantity) and writes the data to the database for persistence

A stock exchange requires **extremely low latency**. While most web applications are ok with hundreds of milliseconds latency, a stock exchange requires **micro-second level latency**. I'll leave the latency discussion for a separate post since the post is already long.

Design a payment system

Today is Cyber Monday. Here is how money moves when you click the Buy button on Amazon or any of your favorite shopping websites.

I posted the same diagram last week for an overview and a few people asked me about the detailed steps, so here you go:



1. When a user clicks the “Buy” button, a payment event is generated and sent to the payment service.

2. The payment service stores the payment event in the database.

3. Sometimes a single payment event may contain several payment orders. For example, you may select products from multiple sellers in a single checkout process. The payment service will call the payment executor for each payment order.

4. The payment executor stores the payment order in the database.

5. The payment executor calls an external PSP to finish the credit card payment.

6. After the payment executor has successfully executed the payment, the payment service will update the wallet to record how much money a given seller has.

7. The wallet server stores the updated balance information in the database.

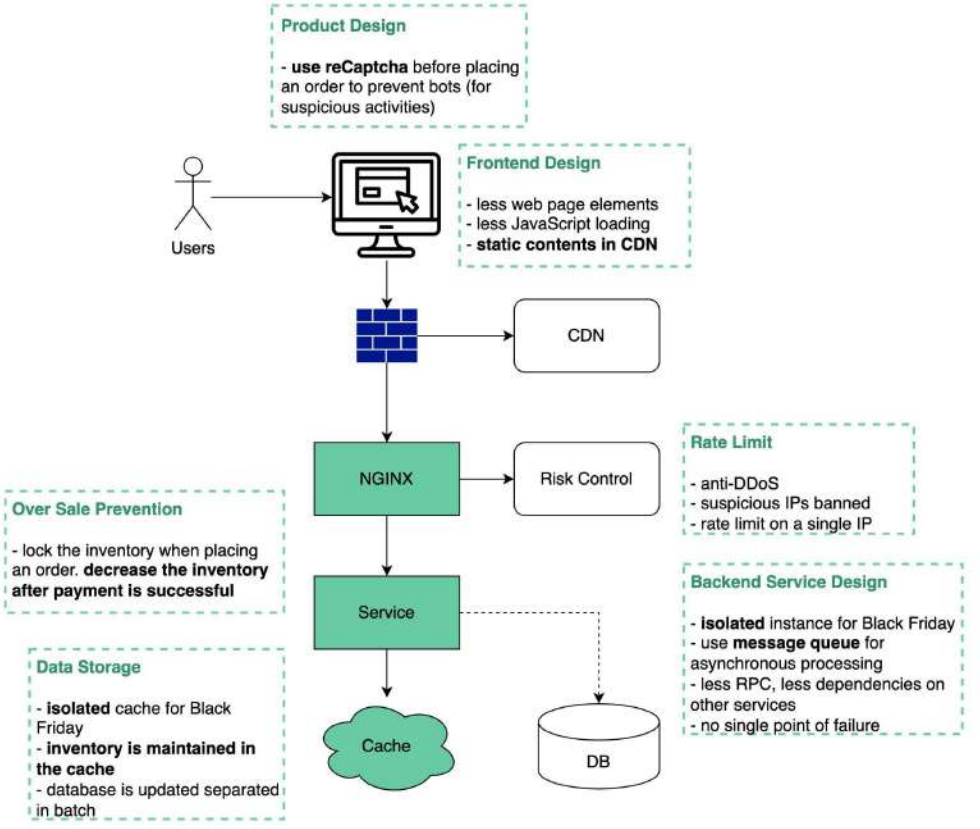
8. After the wallet service has successfully updated the seller's balance information, the payment service will call the ledger to update it.

9. The ledger service appends the new ledger information to the database.

10. Every night the PSP or banks send settlement files to their clients. The settlement file contains the balance of the bank account, together with all the transactions that took place on this bank account during the day.

Design a flash sale system

Black Friday is coming. Designing a system with extremely high concurrency, high availability and quick responsiveness needs to consider many aspects **all the way from frontend to backend.** See the below picture for details:



Design principles:

1. Less is more - less element on the web page, fewer data queries to the database, fewer web requests, fewer system dependencies
2. Short critical path - fewer hops among services or merge into one service
3. Async - use message queues to handle high TPS
4. Isolation - isolate static and dynamic contents, isolate processes and databases for rare items
5. Overselling is bad. When Decreasing the inventory is important

6. User experience is important. We definitely don’t want to inform users that they have successfully placed orders but later tell them no items are actually available

Back-of-the-envelope estimation

Recently, a few engineers asked me whether we really need back-of-the-envelope estimation in a system design interview. I think it would be helpful to clarify.

Estimations are important because we need them to understand the scale of the system and justify the design. It helps answer questions like:

- Do we really need a distributed solution?
- Is a cache layer necessary?
- Shall we choose data replication or sharding?

Here is an example of how the estimations shape the design decision.

One interview question is to design proximity service and how to scale geospatial index is a key part of it. Here are a few paragraphs we wrote to show why jumping to a sharding design without estimations is a bad idea:

“One common mistake about scaling the geospatial index is to quickly jump to a sharding scheme without considering the actual data size of the table. In our case, the full dataset for the geospatial index table is not large (quadtree index only takes 1.71G memory and storage requirement for geohash index is similar). The whole geospatial index can easily fit in the working set of a modern database server. However, depending on the read volume, a single database server might not have enough CPU or network bandwidth to service all read requests. If that is the case, it will be necessary to spread the read load among multiple database servers.

There are two general approaches to spread the load of a relational database server. We can add read replicas or shard the database.

Many engineers like to talk about sharding during interviews. However, it might not be a good fit for the geohash table. Sharding is complicated. The sharding logic has to be added to the application layer. Sometimes, sharding is the only option. In this case though, since everything can fit in the working set of a database server, there is no strong technical reason to shard the data among multiple servers.

A better approach, in this case, is to have a series of read replicas to help with the read load. This method is much simpler to develop and maintain. Thus, we recommend scaling the geospatial index table through replicas.”