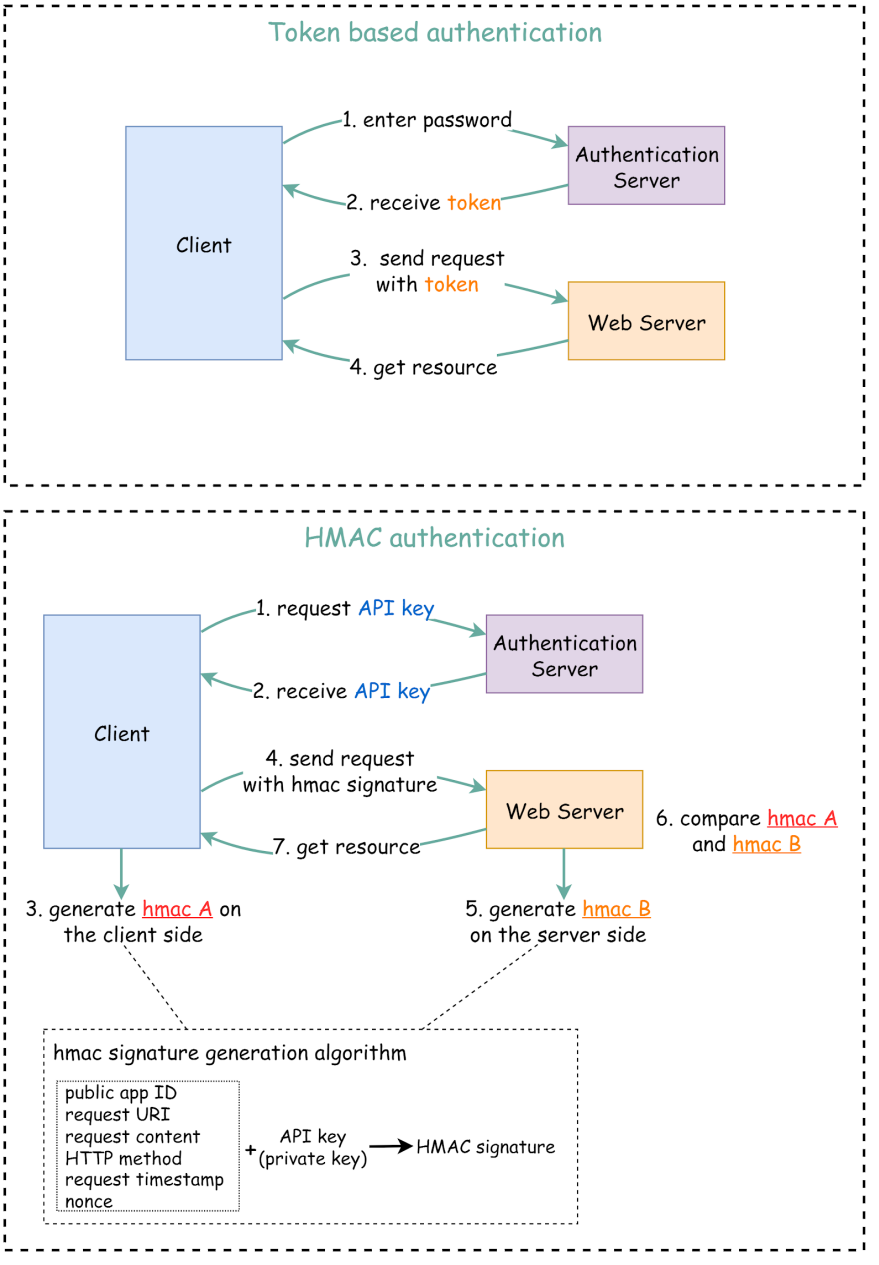
How to design a secure web API access for your website?

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.

How to Design Secure Web API?



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?

—

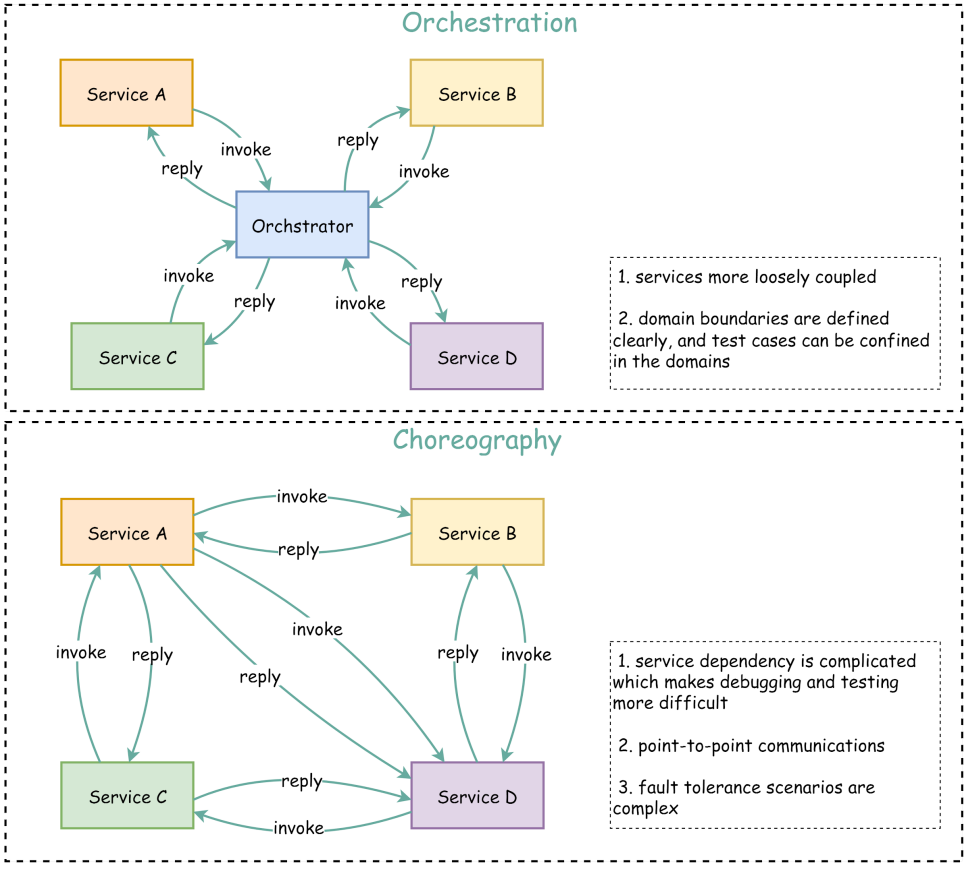
Check out our bestselling system design books.
Paperback: Amazon Digital: ByteByteGo.

How do microservices collaborate and interact with each other?

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

The diagram below illustrates the collaboration of microservices.

> Orchestration v.s. Choreography 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?

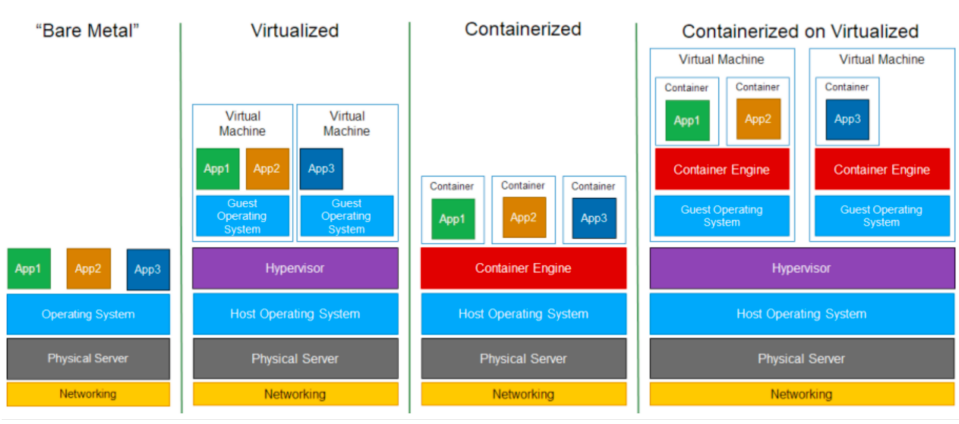
—

Check out our bestselling system design books.
Paperback: Amazon Digital: ByteByteGo.

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

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

> Virtualization vs 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.

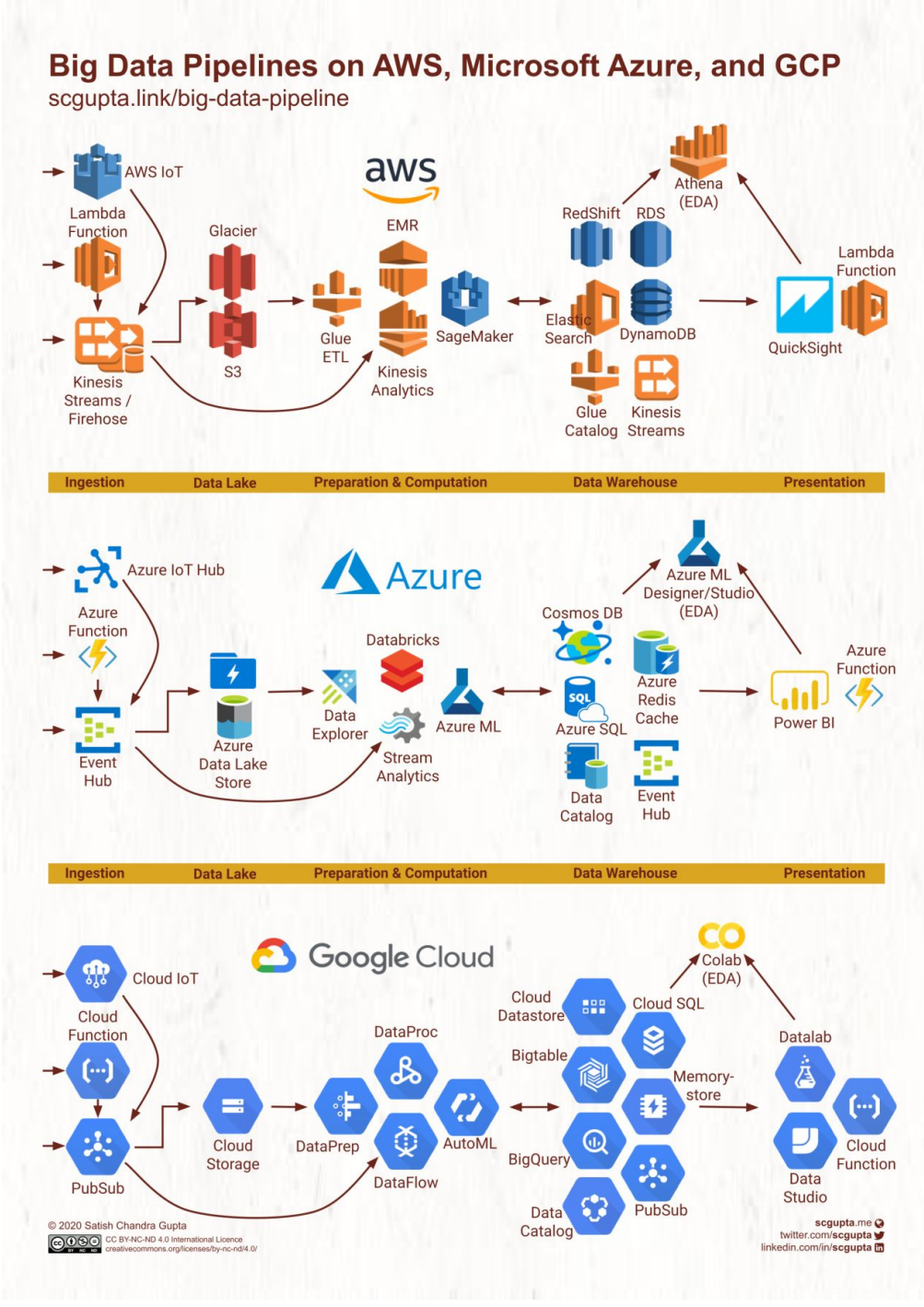
Question: how much performance differences have you observed in production between virtualization, containerization, and bare-metal?

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.

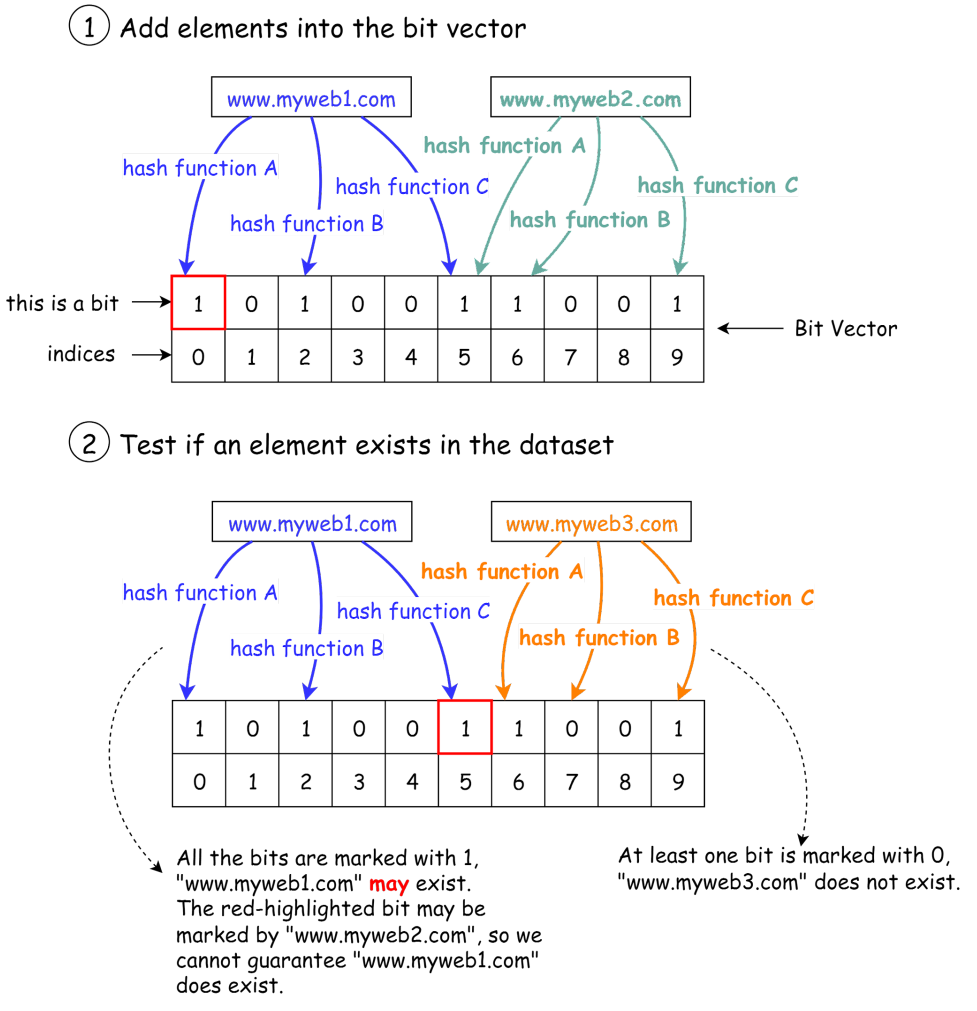
Option 3: **Bloom filter**. This option is preferred. Bloom filter was proposed by Burton Howard Bloom in 1970. It is a probabilistic data structure that is used to test whether an element is a member of a set.

- 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?

—

Check out our bestselling system design books.
Paperback: Amazon Digital: ByteByteGo.

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

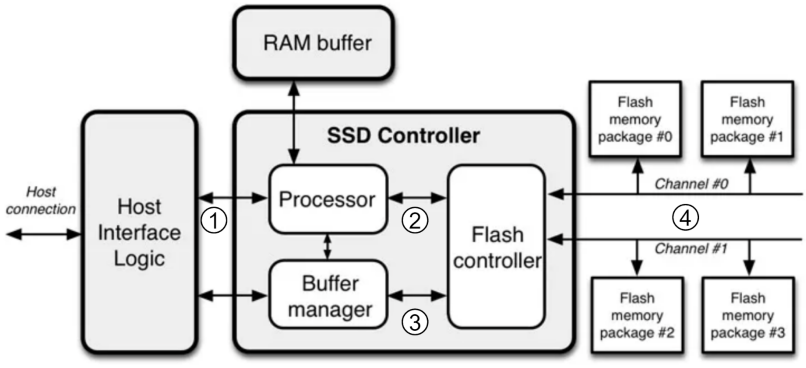
“A solid state drive reads up to 10 times faster and writes up to 20 times faster than a hard disk drive.” [1].

“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].

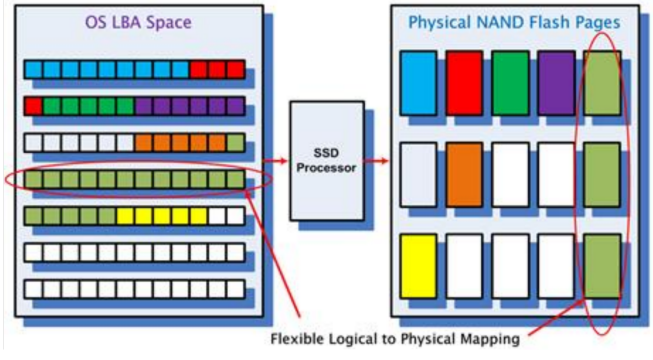
The diagram below illustrates the SSD architecture.

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?

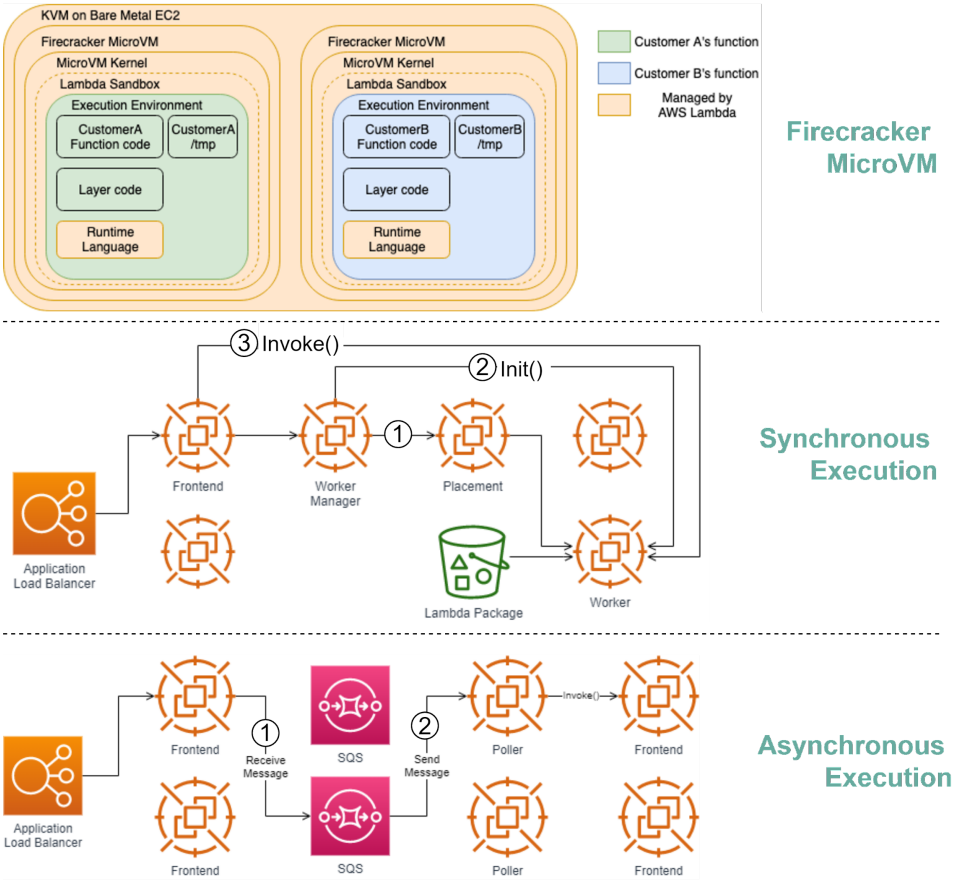
Lambda is a **serverless** computing service provided by Amazon Web Services (AWS), which runs functions in response to events.

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.

How does AWS Lambda work?



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?

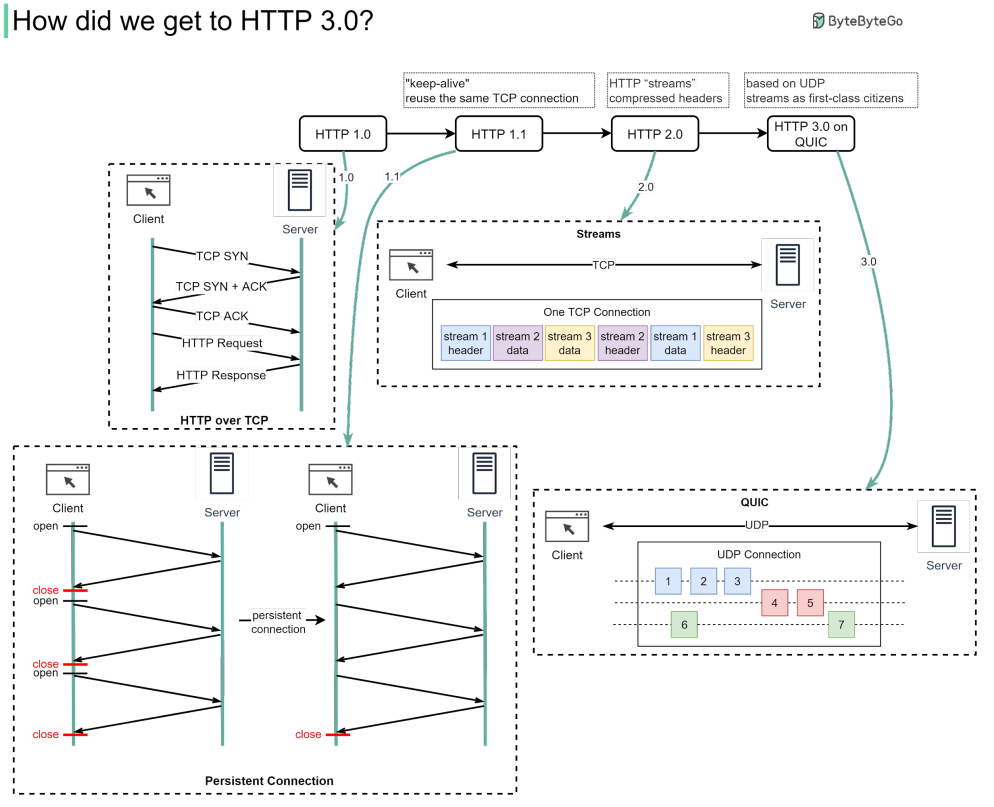
Sources:
[1] AWS Lambda whitepaper:
https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-executions.html
[2] Behind the scenes, Lambda:
https://www.bschaatsbergen.com/behind-the-scenes-lambda/

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?

—
Check out our bestselling system design books.
Paperback: Amazon Digital: ByteByteGo.

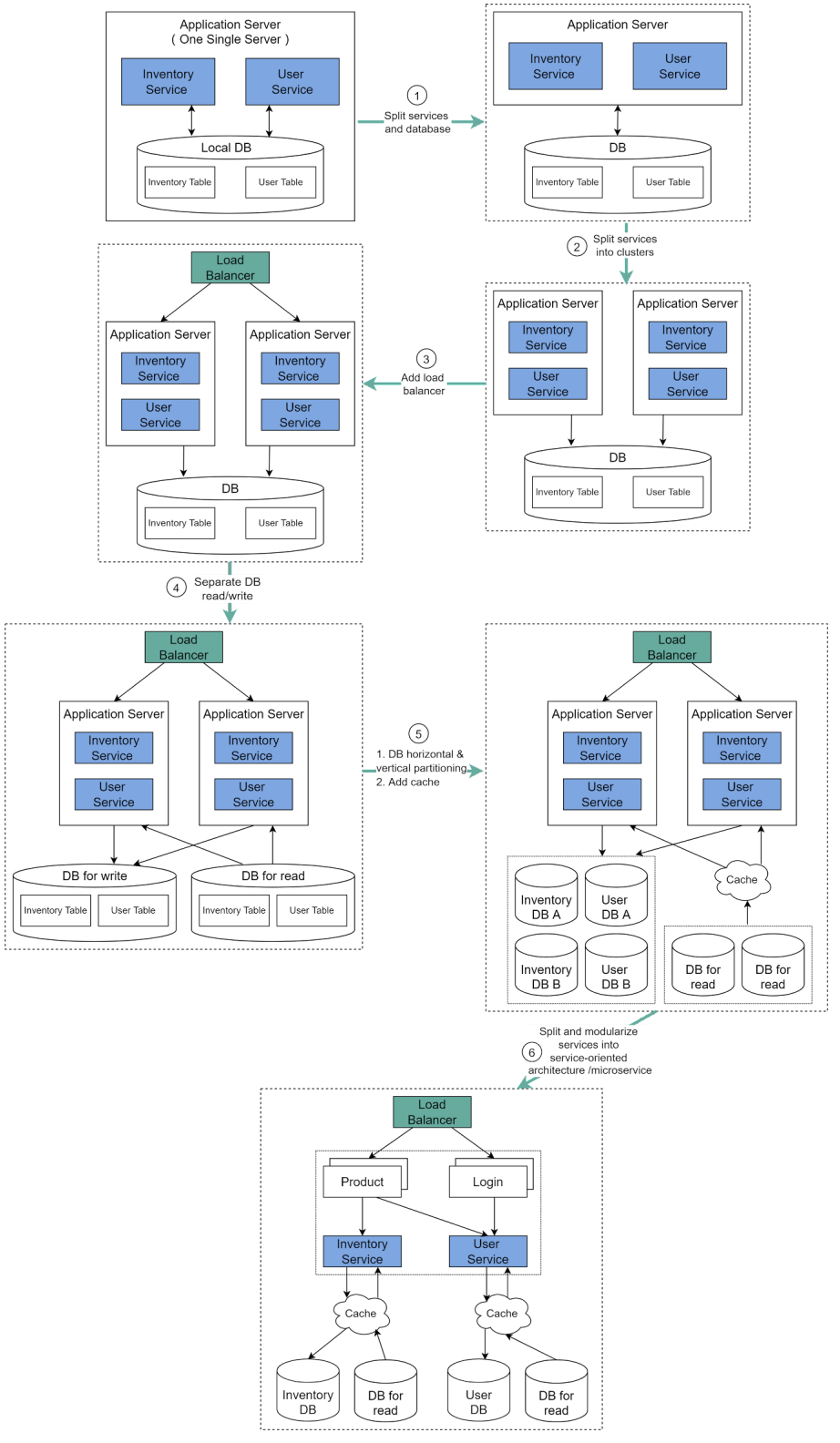
How to scale a website to support millions of users?

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?

ByteByteGo



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!

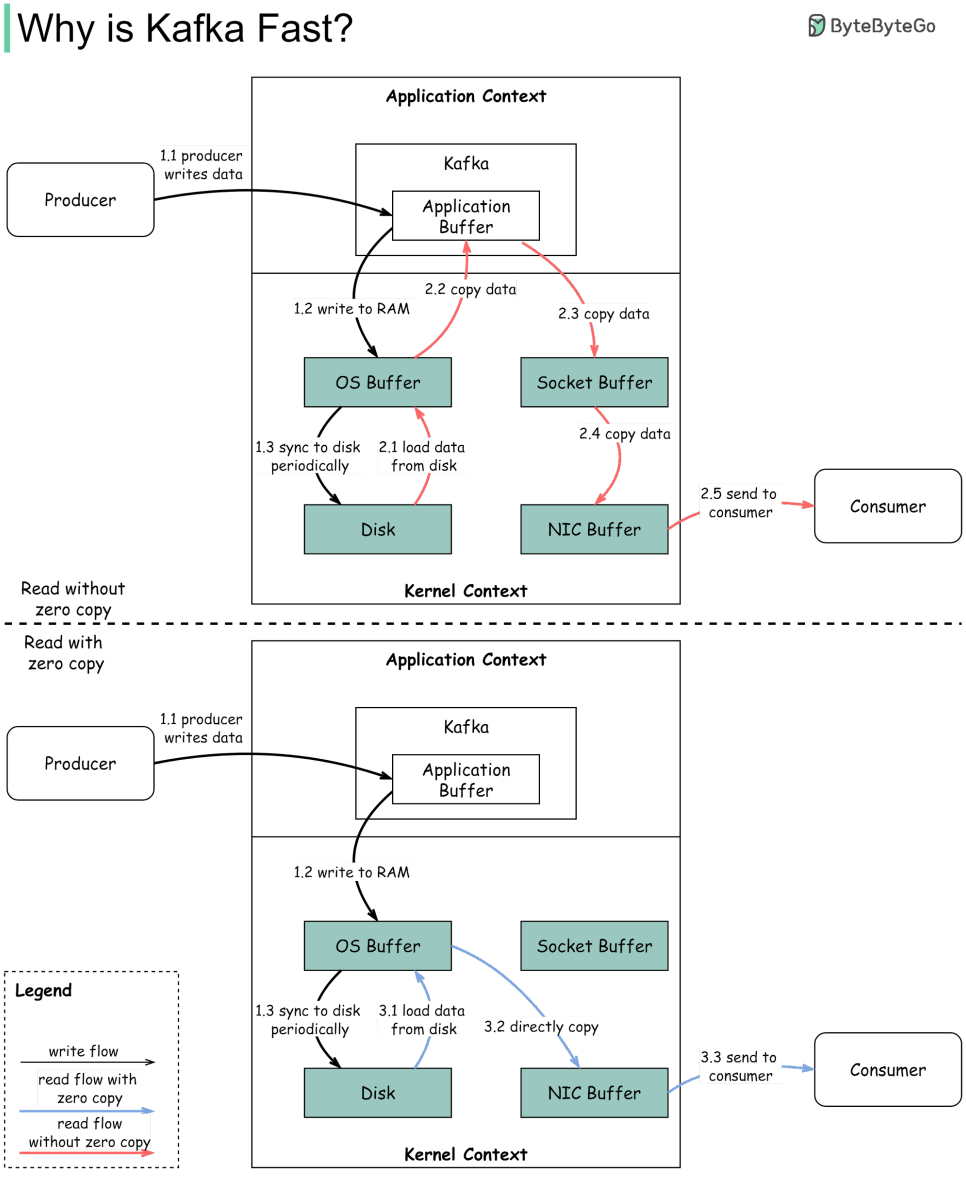
🔹 The DevOps Handbook - introduces product development, quality assurance, IT operations, and information security.

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%.

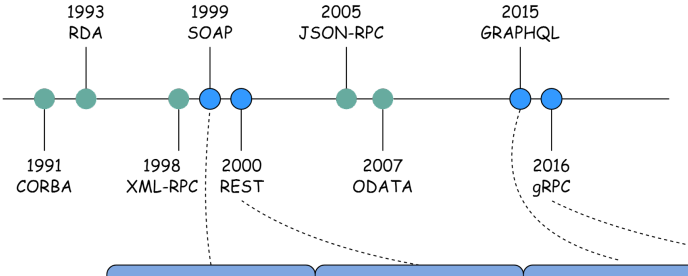
—

Check out our bestselling system design books.
Paperback: Amazon Digital: ByteByteGo.

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.

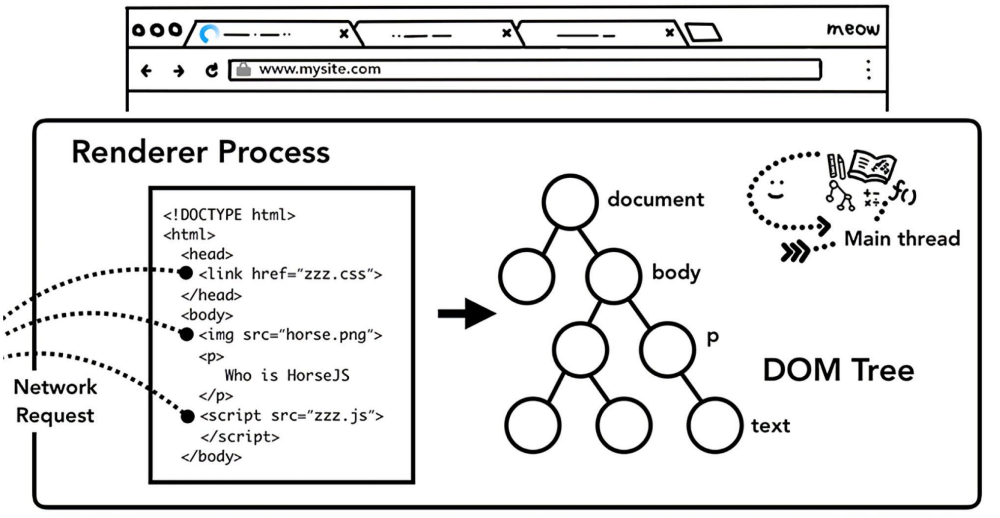
You can check out the use cases of each style in the diagram.

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

—

Check out our bestselling system design books.
Paperback: Amazon Digital: ByteByteGo.

How do modern browsers work?



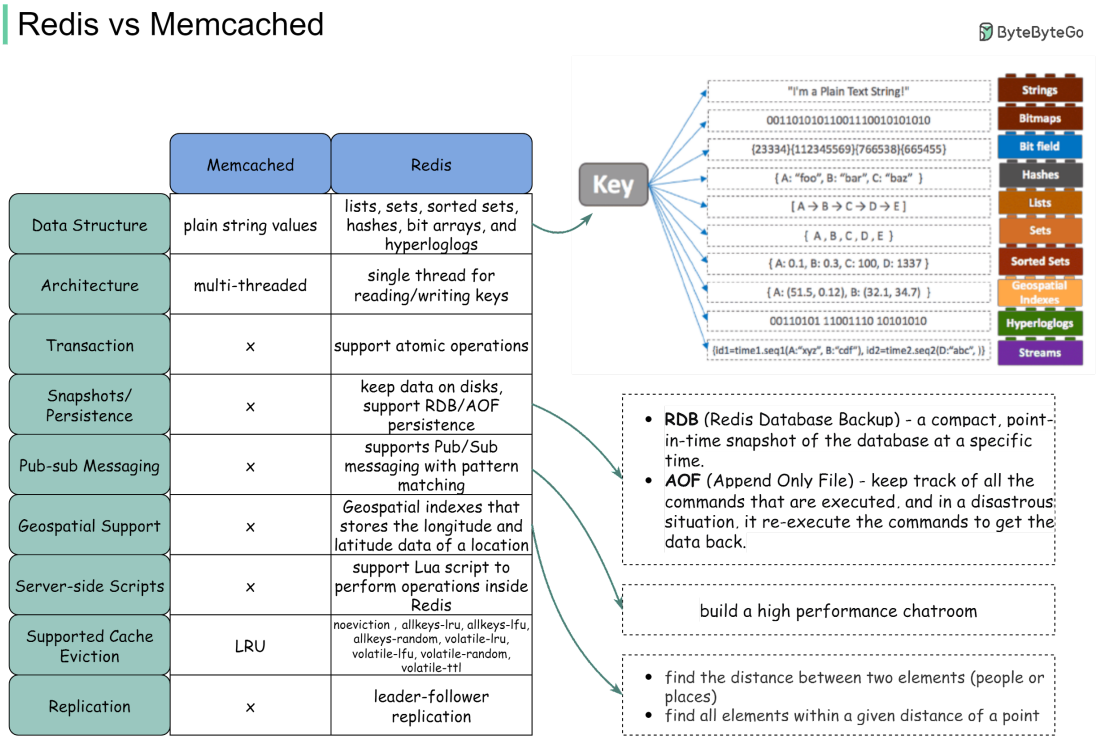
Google published a series of articles about "Inside look at modern web browser". It's a great read.

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.

Redis vs Memcached



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

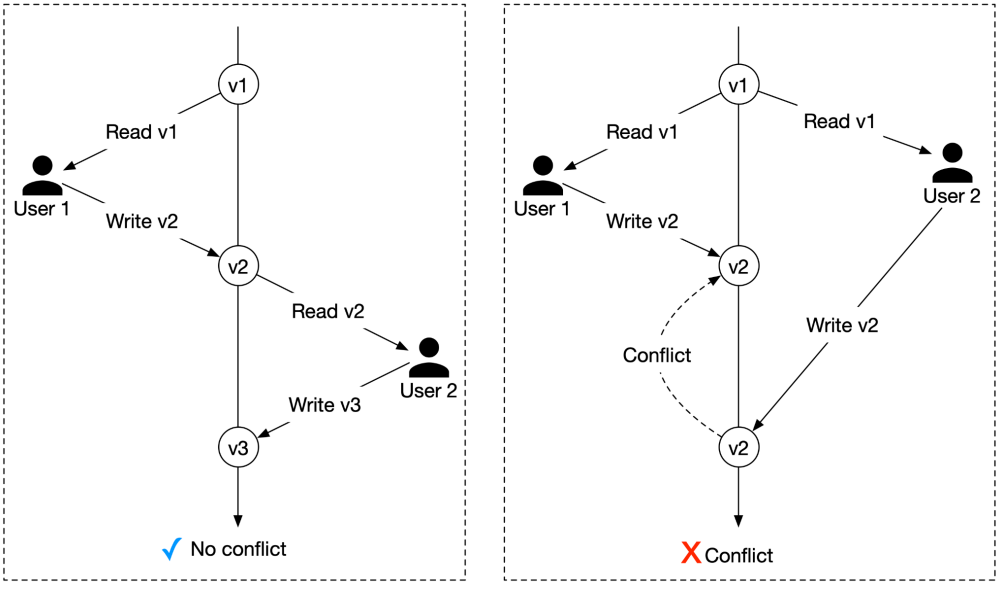
- Recording the number of clicks and comments for each post (hash)
- Sorting the commented user list and deduping the users (zset)
- Caching user behavior history and filtering malicious behaviors (zset, hash)
- Storing boolean information of extremely large data into small space. For example, login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

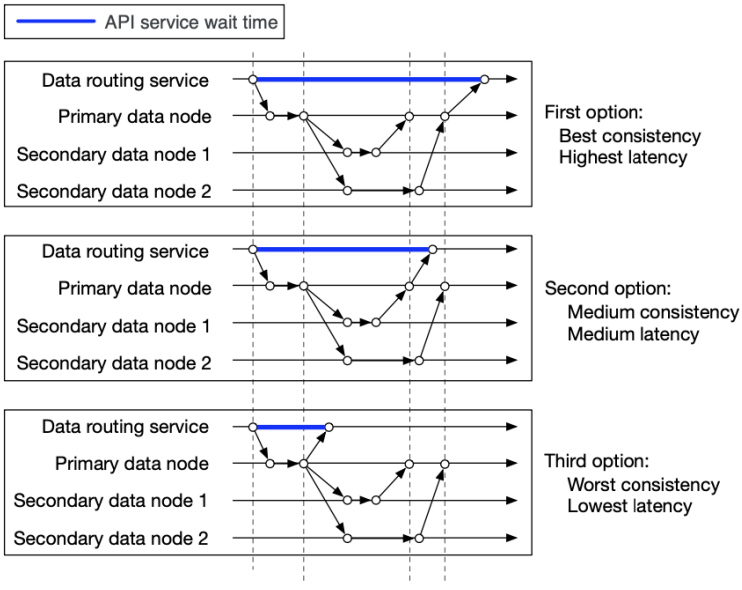


Figure 9.11: Trade-off between consistency and latency

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.

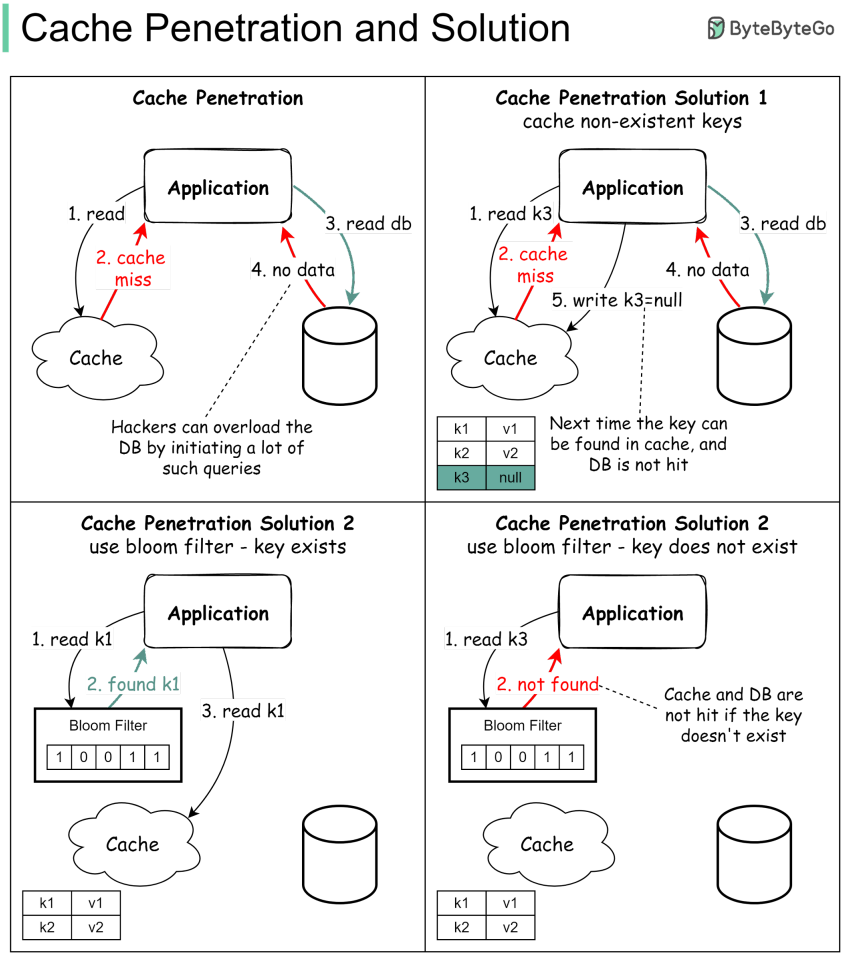
Both 2 and 3 are forms of eventual consistency.

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.

—

Check out our bestselling system design books.
Paperback: Amazon Digital: ByteByteGo.