Acing the System Design Interview

Zhiyong Tan

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Acing the System Design Interview
Version 4

For more information on this and other Manning titles go to
manning.com

welcome

Thanks for purchasing the MEAP for *Acing the System Design Interview*.

System design interviews are common during the interview process for software engineers and engineering managers. However, an engineer or student who searches for online learning materials on system design interviews will find a vast quantity of content which vary in quality and diversity of the topics covered. This is confusing and hinders learning. There are few dedicated books on system design interviews.

This book offers a structured and organized approach to start preparing for system design interviews, or to fill gaps in knowledge and understanding from studying the large amount of fragmented material. Equally valuably, it teaches how to demonstrate one's engineering maturity and communication skills during a system design interview, such as clearly and concisely articulating one's ideas, knowledge, and questions to the interviewer within the brief ~50 minutes.

This book is divided into two main parts. Part 1 is presented like a typical textbook, with chapters that cover the various topics discussed in a system design interview. Part 2 consists of discussions of sample interview questions that reference the concepts covered in part 1, and discusses antipatterns and common misconceptions and mistakes.

The system design interview open-ended and rapidly evolving nature guarantees that there will be much to discuss and debate. Please post questions and comments in the liveBook Discussion forum, as I look forward to learning with you.

--Zhiyong Tan

brief contents

APPENDIXES

1 The system design interview

This chapter covers:

- The importance of the system design interview.
- Evaluating a candidate.
- Scaling a backend service.
- Cloud hosting vs. bare metal.

A system design interview is a discussion between the candidate and the interviewer about designing a software system that is typically provided over a network. The interviewer begins the interview with a short and vague request to the candidate to design a particular software system. Depending on the particular system, the userbase may be non-technical or technical.

System design interviews are conducted for most software engineering, software architecture, and engineering manager job interviews. (In this book, we collectively refer to software engineers, architects, and managers as simply "engineers".) Other components of the interview process include coding and behavioral/cultural interviews.

1.1 It is a discussion about tradeoffs

The following factors attest to the importance of system design interviews and preparing well for them for as a candidate and an interviewer.

Your performance as a candidate in the system design interviews is used to estimate your breadth and depth of system design expertise and your ability to communicate and discuss system designs with other engineers, and this is a critical factor in determining they level of seniority that you will be hired into the company. The ability to design and review large-scale systems is regarded as more important with increasing engineering seniority. Correspondingly, system design interviews are given more weight in interviews for senior

positions. Preparing for them, both as an interviewer and candidate, is a good investment of time for a career in tech.

The tech industry is unique in that it is common for engineers tend to change companies every few years, unlike other industries where an employee may stay in her company for many years or her whole career. This means that a typical engineer will go through system design interviews many times in her career as a candidate. If the engineer is employed at a highly-desirable company, she will go through even more system design interviews as an interviewer. As an interview candidate, you have less than 1 hour to make the best possible impression, and the other candidates who are your competition are among the smartest and most motivated people in the world.

System Design is an art, not a science. It is not about perfection. We make tradeoffs and compromises to design the system we can achieve with the given resources and time that most closely suits current and possible future requirements. All the discussions of various systems in this book involve estimates and assumptions, and are not academically-rigorous, exhaustive, or scientific. We may refer to software design patterns and architectural patterns, but will not formally describe these principles. Readers should refer to other resources for more details.

A System Design interview is not about the right answer. It is about one's ability to discuss multiple possible approaches and weigh their tradeoffs in satisfying the requirements. Knowledge of the various types of requirements and common systems discussed in part 1 will help us design our system, evaluate various possible approaches, and discuss tradeoffs.

1.2 Should you read this book?

The open-ended nature of system design interviews makes it a challenge to prepare for, and know how or what to discuss during an interview. An engineer or student who searches for online learning materials on system design interviews will find a vast quantity of content which vary in quality and diversity of the topics covered. This is confusing and hinders learning. Moreover, until recently, there were few dedicated books on this topic, though a trickle of such books is beginning to be published. I believe that this is because a high-quality book dedicated to the topic of system design interviews is, quoting the celebrated 19th century French poet and novelist Victor Hugo, "an idea whose time has come". Multiple people will get this same idea at around the same time, and this affirms its relevance.

This is not an introductory software engineering book. This book is best used after one has acquired a minimal level of industry experience; perhaps if you are a student in your first internship, you can read the documentation websites and other introductory materials of unfamiliar tools, and discuss them together with other unfamiliar concepts in this book with engineers at your workplace. This book discusses how to approach System Design interviews, and will minimize duplication of introductory material that we can easily find online or in other books. At least intermediate proficiency in coding and SQL are assumed.

This book offers a structured and organized approach to start preparing for system design interviews, or to fill gaps in knowledge and understanding from studying the large amount of fragmented material. Equally valuably, it teaches how to demonstrate one's engineering maturity and communication skills during a system design interview, such as clearly and

concisely articulating one's ideas, knowledge, and questions to the interviewer within the brief ~50 minutes.

A system design interview, like any other interview, is also about communication skills, quick thinking, asking good questions, and performance anxiety. One may forget to mention points that the interviewer is expecting. Whether this interview format is flawed can be endlessly debated. From personal experience, with seniority one spends an increasing amount of time in meetings, and essential abilities include quick thinking, being able to ask good questions, steer the discussion to the most critical and relevant topics, and communicate one's thoughts succinctly. This book emphasizes that one must effectively and concisely express one's system design expertise within the < 1 hour interview, and drive the interview in the desired direction by asking the interviewer the right questions. Reading this book, along with practicing system design discussions with other engineers, will allow you to develop the knowledge and fluency required to pass System Design interviews and participate well in designing systems in the company you join. It can also be a resource to interviewers to conduct system design interviews.

One may excel in written over verbal communication, and forget to mention important points during the ~50-minute interview. System design interviews are biased in favor of engineers with good verbal communication, and against engineers less proficient in verbal communication, even though the latter may have considerable system design expertise and have made valuable system design contributions in the organizations where they worked. This book prepares engineers for these and other challenges of system design interviews, shows how to approach them in an organized way, and not be intimidated.

If you are a software engineer looking to broaden your knowledge of system design concepts, improve your ability to discuss a system, or simply looking for a collection of system design concepts and sample system design discussions, read on.

1.3 Overview of this book

This book is divided into 2 parts. Part 1 is presented like a typical textbook, with chapters that cover the various topics discussed in a system design interview. Part 2 consists of discussions of sample interview questions that reference the concepts covered in part 1, and also discusses antipatterns and common misconceptions and mistakes. In those discussions, we also state the obvious that one is not expected to possess all knowledge of all domains. Rather, one should be able to reason that certain approaches will help satisfy requirements better, with certain tradeoffs. For example, we don't need to calculate file size reduction or CPU and memory resources required for Gzip compression on a file, but we should be able to state that compressing a file before sending it will reduce network traffic but consume more CPU and memory resources on both the sender and recipient.

An aim of this book is to bring together a bunch of relevant materials and organize them into a single book, so you may build a knowledge foundation or identify gaps in your knowledge, from which you can study other materials.

The rest of this chapter is a prelude to a sample system design that mentions some of the concepts that will be covered in part I. Based on this context, we will discuss many of the concepts in dedicated chapters.

1.4 Prelude – a brief discussion of scaling the various services of a system

We begin this book with a brief description of a typical initial setup of an app, and a general approach to adding scalability into our app's services as needed. Along the way, we introduce numerous terms and concepts and many types of services required by a tech company, which we discuss in greater detail in the rest of the book.

The scalability of a service is the ability to easily and cost-effectively vary resources allocated to it so as to serve changes in load. This applies to both increasing or decreasing user numbers and/or requests to the system. This is discussed more in chapter 2.

1.4.1 The beginning - a small initial deployment of our app

Riding the rising wave of interest in artisan bagels, we have just built an awesome consumer-facing app named "Beigel" that allows users to read and create posts about nearby bagel cafes.

Initially, Beigel consists primarily of the following components.

- Our consumer apps. They are the essentially the same app, one for each of the three common platforms:
    - A browser app. This is a ReactJS browser consumer app that makes requests to a JavaScript runtime service. To reduce the size of the JavaScript bundle that users need to download, we compress it with Brotli. gzip is an older and more popular choice, but Brotli produces smaller compressed files.
    - An iOS app, which is downloaded on a consumer's iOS device.
    - An Android app, likewise downloaded on a consumer's Android device.
- A stateless backend service that serves the consumer apps. It can be a Go or Java service.
- A SQL database contained in a single cloud host.

We have two main services, the Web Server service and the Backend service. Figure 1.1 illustrates these components. As illustrated, the consumer apps are client-side components, while services and database are server-side components.

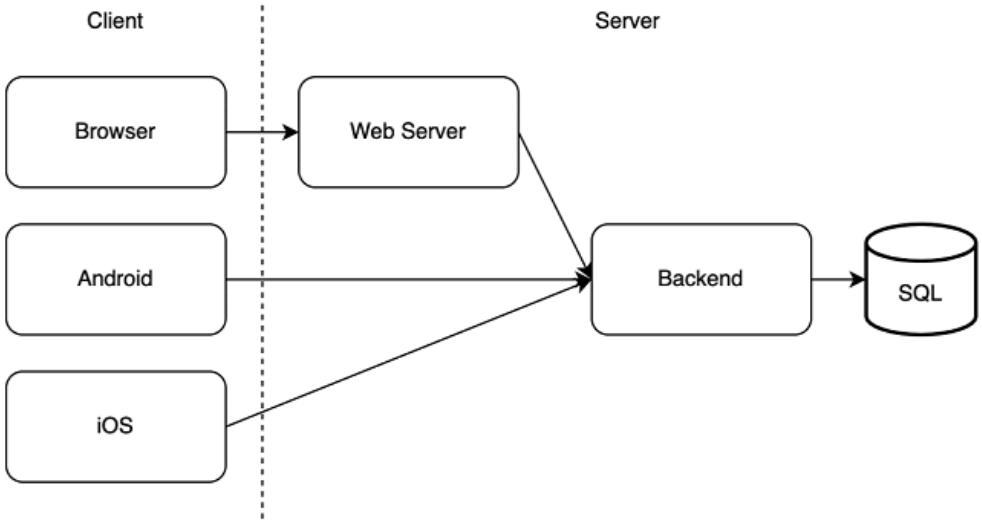

Figure 1.1 Initial system design of our app. For a more thorough discussion on the rationale for having 3 client applications and 2 server applications (excluding the SQL application/database), refer to the chapter in the appendix "Functional partitioning and various frameworks".

When we first launch a service, it may only have a small number of users and thus a low request rate. A single host may be sufficient to handle the low request rate. We will setup our DNS to direct all requests at this host.

Initially, we can host the two services within the same datacenter, each on a single cloud host. (We compare cloud vs bare metal in the next section.) We configure our DNS to direct all requests from our browser app to our node.js host, and from our node.js host and two mobile apps to our backend host.

1.4.2 Scaling with GeoDNS

Months later, Beigel has gained hundreds of thousands of daily active users in Asia, Europe, and North America. During periods of peak traffic, our backend service receives thousands of requests per second, and our monitoring system is starting to report status code 504 responses due to timeouts. We must scale up our system.

We have been observing the rise in traffic and preparing for this situation. The first approach to perform functional partitioning on the backend, to use separate hosts and/or clusters to serve different functions. Our service is stateless as per standard best practices, so we can provision multiple identical backend hosts, and place each host in a different data center in a different part of the world. Referring to figure 1.2, when a client makes a request to our backend via its domain "beigel.com", we use GeoDNS to direct the client to the data center closest to it.

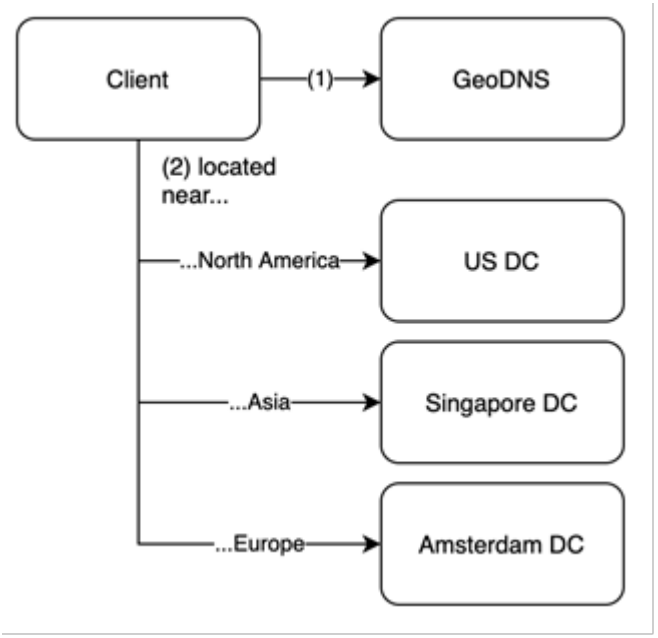

Figure 1.2 We may provision our service in multiple geographically-distributed data centers. Depending on the client's location (inferred from its IP address), a client obtains the IP address of a host of the closest data center, to which it sends its requests. The client may cache this host IP address for as long as its own IP address does not change. As discussed later in this section, when we horizontally scale our service, the host IP address may actually belong to a load balancer, which balances traffic across our service's hosts.

If our service serves users from a specific country or geographical region in general, we will typically host our service in a nearby data center to minimize latency. If your service serves a large geographically-distributed userbase, we can host it on multiple data centers, and use GeoDNS to return a user the IP address of our service hosted in the closest data center. This is done by assigning multiple A records to our domain for various locations, and a default IP address for other locations. (An A record is a DNS configuration that maps a domain to an IP address.)

When a client makes a request to the server, the GeoDNS obtains the client's location from her IP address, and assigns the client the corresponding host IP address. In the unlikely but possible event that the data center is inaccessible, GeoDNS can return an IP address of the service on another data center. This IP address can be cached at various levels, including the user's Internet Service Provider (ISP), OS, and browser.

1.4.3 Adding a caching service

Referring to figure 1.3, we next set up a Redis cache service to serve cached requests from our consumer apps. We select certain backend endpoints with heavy traffic to serve from the

cache. That bought us some time as our user base and request load continued to grow. In the meanwhile, further steps are needed to scale up.

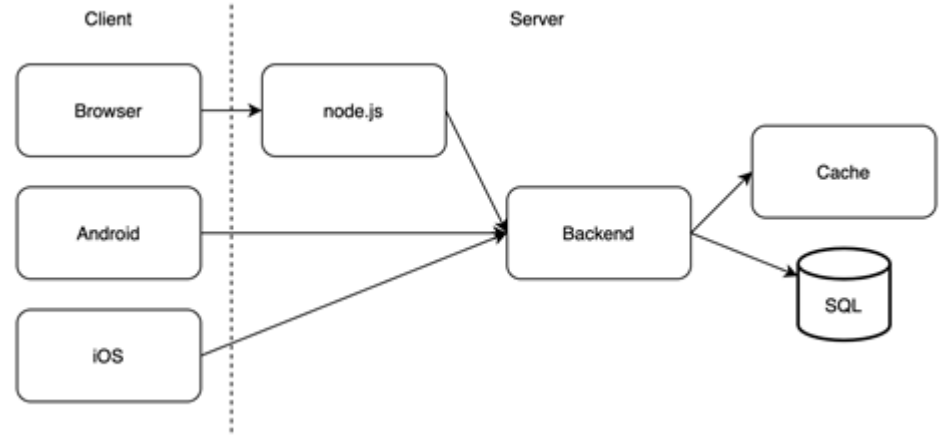

Figure 1.3 Adding a cache to our service. Certain backend endpoints with heavy traffic can be cached. The backend will request data from the database on cache miss or for SQL databases/tables that were not cached.

1.4.4 Content Distribution Network (CDN)

Our browser apps had been hosting static content/files that are displayed the same to any user, and unaffected by user input, such JavaScript and CSS libraries, and some images and videos. We had placed these files within our app's source code repository, and our users had been downloading them from our node.js service together the rest of the app. Referring to figure 1.4, we decide to use a third party CDN to host the static content. We select and provisioned sufficient capacity from a CDN to host our files, uploaded our files onto our CDN instance, rewrote our code to fetch the files from the URLs of the CDN, and removed the files from our source code repository.

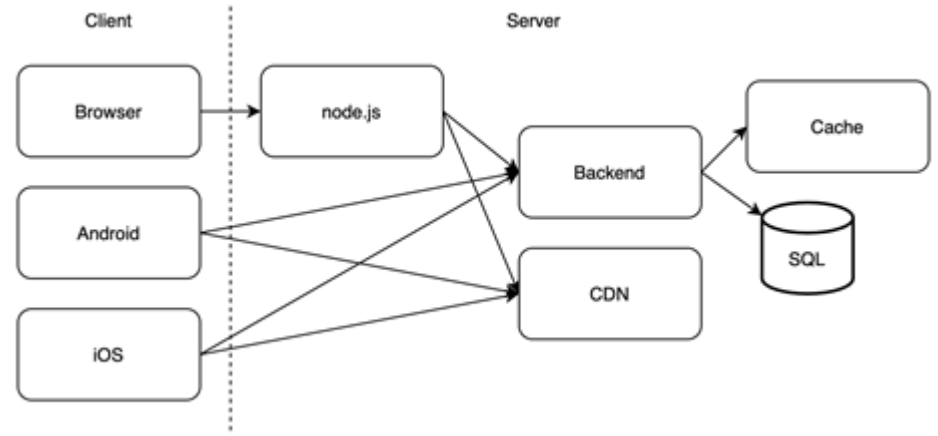

Figure 1.4 Adding a CDN to our service. Clients can obtain CDN addresses from the backend, or certain CDN addresses can be hardcoded in the clients or node.js service.

Referring to figure 1.5, a CDN stores copies of the static files in various datacenters across the world, so a user can download these files from the datacenter which can provide her the lowest latency, which is usually but not always the geographically-closest one.

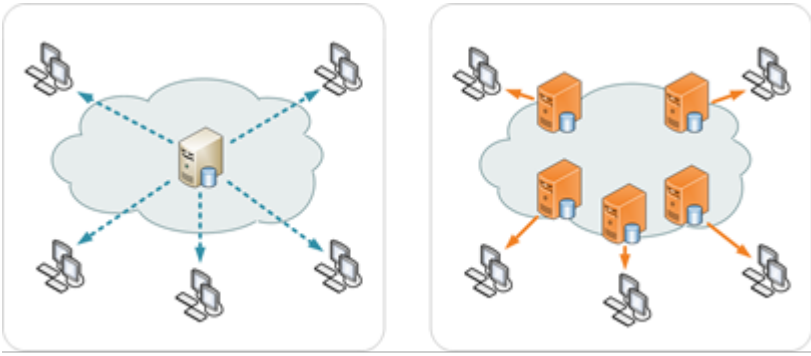

Figure 1.5 The left illustration shows all clients downloading from the same host. The right illustration shows clients downloading from various hosts of a CDN. Copyright cc-by-sa https://creativecommons.org/licenses/by-sa/3.0/. Image by Kanoha from https://upload.wikimedia.org/wikipedia/commons/f/f9/NCDN_-_CDN.png.

Using a CDN improved latency, throughput, reliability, and cost. (We discuss all these buzzwords/concepts in chapter 2.) Using a CDN, unit costs decrease with demand, as maintenance, integration overhead, and customer support are spread over a larger load.

Popular CDNs include CloudFlare, Rackspace, and AWS CloudFront.

1.4.5 A brief discussion of horizontal scalability and cluster management, Continuous Integration (CI) and Continuous Deployment (CD)

Our frontend and backend services are idempotent (We discuss idempotency and its benefits in chapter 2.) thus horizontally scalable, so we can provision more hosts to support our larger request load without rewriting any source code, and deploy the frontend or backend service to those hosts as needed.

Each of our services has multiple engineers working on its source code. Our engineers submit new commits every day. We change software development and release practices to support this larger team and faster development, hiring 2 DevOps engineers in the process to develop the infrastructure to manage a large cluster. As scaling requirements of a service can change quickly, we want to be able easily resize its cluster. We need to be able to easily deploy our services and required configurations to new hosts. We also want to easily build and deploy code changes to all the hosts in our service's cluster. We can take advantage of our large userbase for experimentation by deploying different code or configurations to various hosts. This section is a brief discussion of cluster management for horizontal scalability and experimentation.

CI/CD AND INFRASTRUCTURE AS CODE (IAC)

To allow new features to be released quickly while minimizing the risk of releasing bugs, we adopt Continuous Integration (CI) and Continuous Deployment (CD) with Jenkins and unit testing and integration testing tools. (A detailed discussion of CI/CD is outside the scope of this book.) We use Docker to containerize our services, Kubernetes (or Docker Swarm) to manage our host cluster including scaling and providing load balancing, and Ansible or Terraform for configuration management of our various services running on our various clusters.

Mesos is widely considered obsolete. Kubernetes is the clear winner. A couple of relevant articles are https://thenewstack.io/apache-mesos-narrowly-avoids-a-move-to-the-attic-for-now/ and https://www.datacenterknowledge.com/business/after-kubernetes-victory-its-former-rivals-change-tack.

Terraform allows an infrastructure engineer to create single configuration compatible with multiple cloud providers. A configuration is authored in Terraform's domain-specific language (DSL), and communicates with cloud APIs to provision infrastructure. In practice, a Terraform configuration may contain some vendor-specific code, which we should minimize. The overall consequence is less vendor lock-in.

This approach is also known as Infrastructure as Code (IaC). Infrastructure as Code (IaC) is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools[1].

[1] Wittig, Andreas; Wittig, Michael (2016). Amazon Web Services in Action. Manning Press. p. 93. ISBN 978-1-61729-288-0

GRADUAL ROLLOUTS, AND ROLLBACKS

In this section, we briefly discuss gradual rollouts and rollbacks, so we can contrast it with experimentation in the next section.

When we deploy a build to production, we may do so gradually. We may deploy the build to a certain percentage of hosts, monitor it, then increase the percentage, repeating this process until 100% of production hosts are running this build. For example, we may deploy to 1%, 5%, 10%, 25%, 50%, 75%, then finally 100%. We may manually or automatically roll back deployments if we detect any issues, such as:

- Bugs that slipped through testing.
- Crashes.
- Increased latency or timeouts.
- Memory leaks.
- Increased resource consumption like CPU, memory, or storage utilization.
- Increased user churn. We may also need to consider user churn in gradual rollouts i.e., that new users are signing on and using the app, and certain users may stop using the app. We can gradually expose an increasing percentage of users to a new build, and study its effect on churn. User churn may occur due to the other factors stated above e.g., crashes or timeouts, or to unexpected issues.

For example, a new build may increase latency beyond an acceptable level. We can use a combination of caching and dynamic routing to handle this. Our service may specify a 1-second latency. When a client makes a request that is routed to a new build, and a timeout occurs, our client may read from its cache, or it may repeat its request and be routed to a host with an older build. We should log the requests and responses so we can troubleshoot the timeouts.

We can configure our CD pipeline to divide our production cluster into several groups, and our CD tool will determine the appropriate numbers of hosts in each group and assign hosts to groups. Reassignments and redeployments may occur if we resize our cluster.

EXPERIMENTATION

As we make UX changes in developing new features (or removing features) and aesthetic designs in our application, we may wish to gradually roll them out to an increasing percentage of users, rather than to all users at once. The purpose of experimentation is to determine the effect of UX changes on user behavior, in contrast to gradual rollouts which are about the effect of new deployments on application performance and user churn. Common experimentation approaches are A/B testing (An introduction to A/B testing is outside the scope of this book. Refer to other sources like https://www.optimizely.com/optimization-glossary/ab-testing/.) and multivariate testing such as multi-armed bandit (Refer to other sources like https://www.optimizely.com/optimization-glossary/multi-armed-bandit/ for an introduction to multi-armed bandit.).

Experimentation is also done for personalization, to deliver personalized user experiences.

Another difference between experimentation vs gradual rollouts and rollbacks is that in experimentation, the percentage of hosts running various builds is often tuned by an experimentation or feature toggle tool that is designed for that purpose, while in gradual rollouts and rollbacks, the CD tool is used to manually or automatically roll back hosts to previous builds if issues are detected.

CD and experimentation allow short feedback cycles to new deployments and features.

In web and backend applications, each UX experience is usually packaged in a different build. A certain percentage of hosts will contain a different build. Mobile apps are usually different. Many UX experiences are coded into the same build, but each individual user will only be exposed to a subset of these UX experiences. The main reasons for this are:

- Mobile application deployments must be made through the app store. It may take many hours to deploy a new version to user devices. There is no way to quickly roll back a deployment.
- Compared to Wi-Fi, mobile data is slower, less reliable, and more expensive. Slow speed and unreliability mean we need to have much content served offline, already in the app. Mobile data plans in many countries are still expensive, and may come with data caps and overage charges. We should avoid exposing users to these charges, or they may use the app less or uninstall it altogether. To conduct experimentation while minimizing data usage from downloading components and media, we simply include all these components and media in the app, and expose the desired subset to each individual user.
- A mobile app may also include many features that any particular user will never use because it is not applicable to her. For example, chapter 19.1 discusses various methods of payments in an app. There are possibly thousands of payment solutions in the world. The app needs to contain all the code and SDKs for every payment solution, so it can present each user with the small subset of payment solutions that she may have.

A consequence of all this is that a mobile app can be over 100MB in size. The techniques to address this are outside the scope of this book. We need to achieve a balance and consider tradeoffs. For example, YouTube mobile app's installation obviously cannot include many YouTube videos.

1.4.6 Functional partitioning and centralization of cross-cutting concerns

Functional partitioning is about separating various functions into different services or hosts.

Many services have common concerns that can be extracted into shared services.

SHARED SERVICES

Our company is expanding rapidly. Our daily active user count has grown to millions. We expand our engineering team to 5 iOS engineers, 5 Android engineers, 10 frontend engineers, 100 backend engineers, and create a data science team.

Our expanded engineering team can work on many services beyond the apps directly used by consumers, such as services for our expanding customer support and operations

departments. We add features within the consumer apps for consumers to contact customer support, and for operations to create and launch variations of our products.

Many of our apps contain search bars. We create a shared search service with Elasticsearch.

In addition to horizontal scaling, we use functional partitioning to spread out data processing and requests across a large number of geographically-distributed hosts, by partitioning based on functionality and geography. We already did functional partitioning of our cache, node.js service, backend service, and database service into separate hosts, and we do functional partitioning for other services as well, placing each service on its own cluster of geographically-distributed hosts. Figure 1.6 shows the shared services that we add to Beigel.

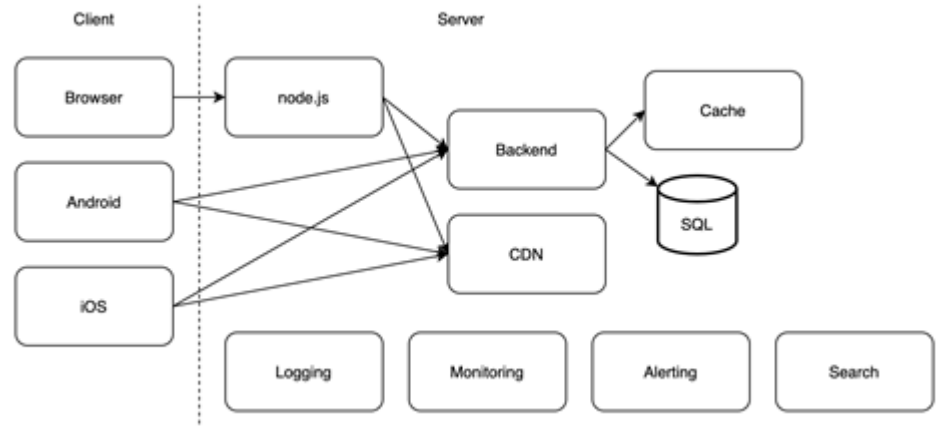

Figure 1.6 Functional partitioning. Adding shared services.

We had added a logging service, consisting of a log-based message broker. We can use ELK (Elasticsearch, Logstash, Kibana). We also use a distributed tracing system such as Zipkin or Jaeger or distributed logging, to trace a request as it traverses through our numerous services; our services attach span IDs to each request so they can be assembled as traces and analyzed. Chapter 9 discusses logging, monitoring, and alerting.

We also added monitoring and alerting services. We build internal browser apps for our customer support employees to better assist customers. These apps process the consumer app logs generated by the customer and present them with good UI so our customer support employees can more easily understand the customer's issue.

API gateway and Service Mesh are two ways to centralized cross-cutting concerns. Other ways are the decorator pattern and aspect-oriented programming, which are outside the scope of this book.

API GATEWAY

By the time, app users make less than half of our API requests. Most requests originate from other companies, which offer services such as recommending useful products and services to our users based on their in-app activities. We develop an API Gateway layer to expose some of our APIs to external develops.

An API gateway is a reverse proxy that routes client requests to the appropriate backend services. It provides the common functionality to many services, so individual services do not duplicate them:

- Authorization and authentication, and other access control and security policies.
- Logging, monitoring, and alerting at the request level.
- Rate limiting
- Billing
- Analytics
- Pods and Sidecars

Our initial architecture involving an API gateway and its services is illustrated in figure 1.7. A request to a service goes through a centralized API gateway. The API gateway carries out all the functionality described in the previous section, does a DNS lookup, then forwards the request to a host of the relevant service. The API gateway makes requests to services such as DNS, identity and access control and management, rate limiting configuration service etc. We also log all configuration changes done through the API gateway.

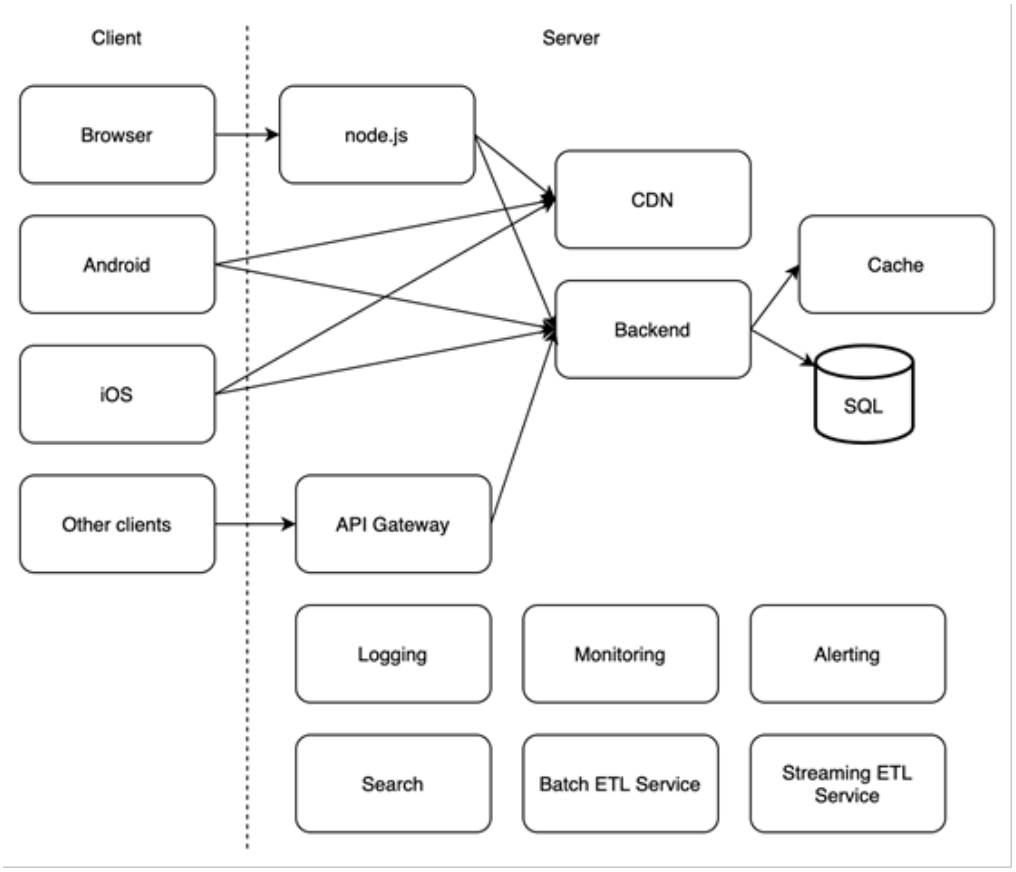

Figure 1.7 Initial architecture with our API gateway and services. Requests to services go through the API gateway.

However, this architecture has the following drawbacks. The API gateway adds latency, and requires a large cluster of hosts. The API gateway host and a service's host that serve a particular request may be in different data centers. A system design that tries to route requests through API gateway hosts and service hosts will be an awkward and complex design.

A solution is to use a Service Mesh, also called the Sidecar pattern. We discuss service mesh further in chapter 5. Figure 1.8 illustrates our service mesh. We can use a service mesh framework such as Istio. Each host of each service can run a sidecar along the main service. We use Kubernetes pods to accomplish this. Each pod can contain its service (in one container) as well as its sidecar (in another container). We provide an admin interface to configure policies, and these configurations can be distributed to all sidecars.

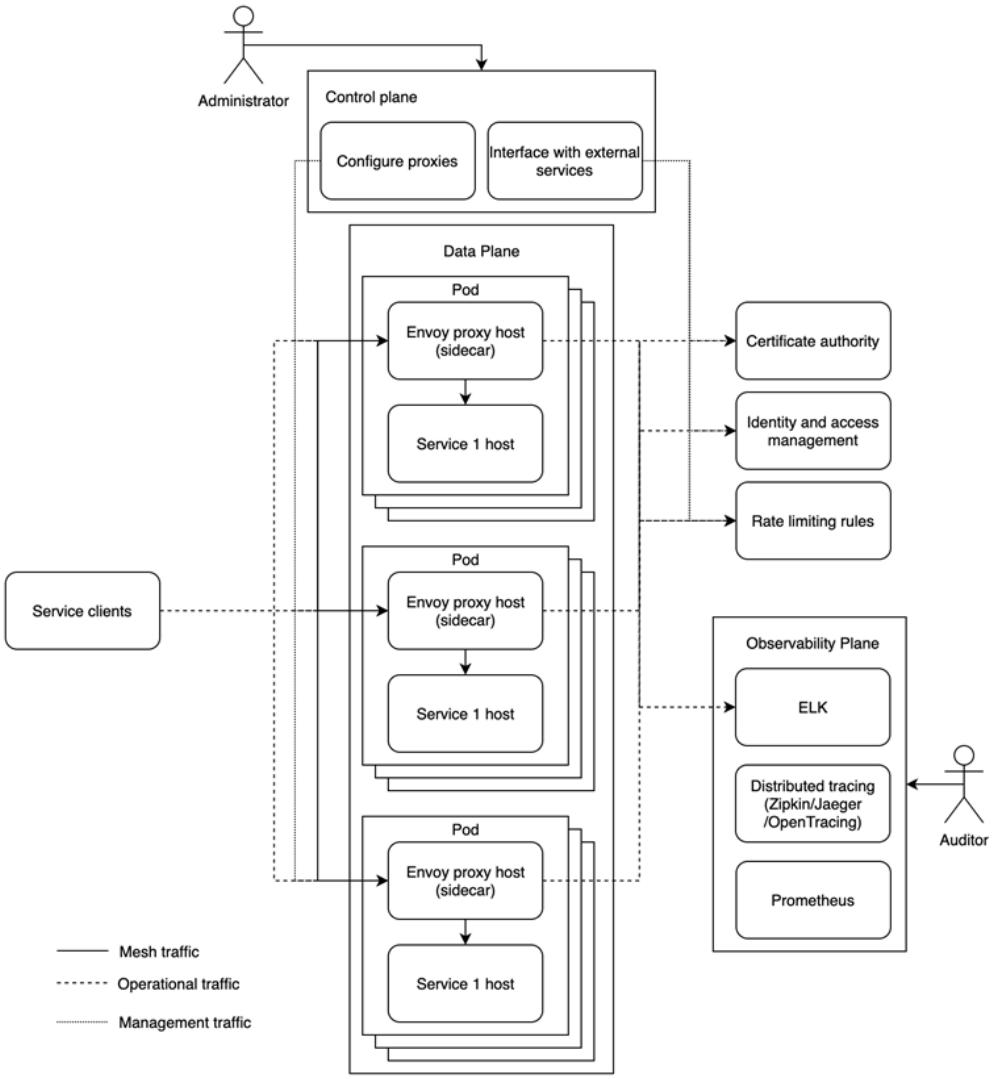

Figure 1.8 Illustration of a service mesh. Prometheus makes requests to each proxy host to pull/scrape metrics, but this is not illustrated in the diagram as the many arrows will make it too cluttered and confusing. Figure adapted from https://livebook.manning.com/book/cloud-native/chapter-10/146.

With this architecture, all service requests and responses are routed through the sidecar. The service and sidecar are on the same host i.e., same machine, so they can address each other over localhost, and there is no network latency. However, the sidecar does consume system resources.

SIDECARLESS SERVICE MESH - THE CUTTING EDGE

The service mesh required our system to nearly double the number of containers. For systems that involve communication between internal services (a.k.a. ingress or east-west), we can reduce this complexity by placing the sidecar proxy logic into client hosts that make requests to service hosts. In the design of sidecarless service mesh, client hosts receive configurations from the control plane. Client hosts must support the control plane API, so they must also include the appropriate network communication libraries.

A limitation of sidecarless service mesh is that there must be a client that is in the same language as the service.

The development of sidecarless service mesh platforms is in their ea›rly stages. Google Cloud Platform (GCP) Traffic Director is an implementation that was released in April 2019 (https://cloud.google.com/blog/products/networking/traffic-director-global-traffic-management-for-open-service-mesh).

COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS)

Command Query Responsibility Segregation (CQRS) is a microservices pattern where command/write operations and query/read operations are functionally partitioned onto separate services. Message brokers and ETL jobs are examples of CQRS. Any design where data is written to one table then this data is transformed and inserted into another table is an example of CQRS. CQRS introduces complexity, but has lower latency, better scalability, and easier to maintain and use. The write and read services can be scaled separately.

You will see many examples of CQRS in this book, though they will not be called out. Chapter 19.3 has one such example, where an Airbnb host writes to the Listing Service, but guests read from the Booking Service. (Though the Booking Service also provides write endpoints for guests to request bookings, which is unrelated to a host updating her listings.)

You can easily find more detailed definition of CQRS in other sources.

1.4.7 Batch and streaming extract, transform and load (ETL)

Some of our systems have unpredictable traffic spikes, and certain data processing requests do not have to be synchronous (i.e., process immediately and return response), such as:

- Some requests that involve large queries to our databases (such as queries that process gigabytes of data).
- It may make more sense to periodically preprocess certain data ahead of requests rather than process it only when a request is made. For example, our app's home page may display the top 10 most frequently learned words across all users in the last hour or in the 7 days. This information should be processed ahead of time once an hour or once a day. Moreover, the result of this processing can be reused for all users, rather than repeating the processing for each user.
- Another possible example is that it may be acceptable for users to be shown data that is outdated by some hours or days e.g., users do not need to see the most updated statistics of the numbers of users who have viewed their shared content. It is acceptable to show them statistics that are out-of-date by a few hours.
- Writes (e.g., INSERT, UPDATE, DELETE database requests) that do not have to be executed immediately. For example, writes to the logging service do not have to be

immediately written to the hard disk drives of logging service hosts. These write requests can be placed in a queue and executed later.

In the case of certain systems like logging which receive large request volumes from many other systems, if we do not use an asynchronous approach like ETL, the logging system cluster will have to have thousands of hosts to process all these requests synchronously.

We can use a combination of event streaming systems like Kafka (or Kinesis if we use AWS) and batch ETL tools such as Airflow for such batch jobs.

If we wish to continuously process data, rather than periodically running batch jobs, we can use streaming tools such as Flink. For example, if a user had input some data into our app and we want to use it to send certain recommendations or notifications to her within seconds or minutes, we can create a Flink pipeline that processes recent user inputs. A logging system is usually streaming, because it expects a non-stop stream of requests. If the requests are less frequent, a batch pipeline will be sufficient.

Figure 1.9 shows our system so far, after adding the API Gateway and the ETL Services. We can also suggest using a service mesh.

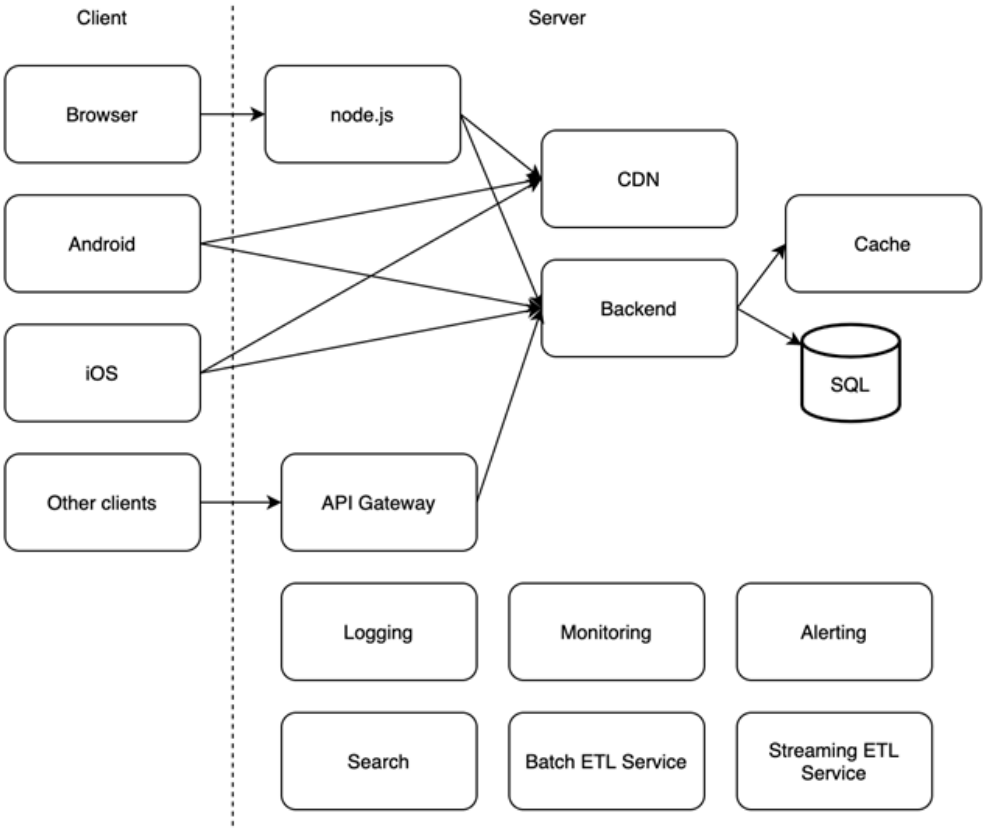

Figure 1.9 Some of our Beigel system components, including the API Gateway for developer clients and our batch and streaming ETL services.

1.4.8 Other common services

As our company grows and our userbase expands, we develop more products, and our products should become increasingly customizable and personalized to serve this large, growing, and diverse userbase. We will require numerous other services to satisfy the new requirements that comes with this growth and to take advantage of it. They include the following:

- Customer/external user credentials management, for external user authentication and authorization.
- Various storage services, including database services. The specific requirements of each system means that there are certain optimal ways that the data it uses should be persisted, processed, and served. We will need to develop and maintain various shared storage services that use different technologies and techniques.

- Asynchronous processing. Our large userbase requires more hosts and may create unpredictable traffic spikes to our services. To handle traffic spikes, we need asynchronous processing to efficiently utilize our hardware and reduce unnecessary hardware expenditure.
- Notebooks service for analytics and machine learning, including experimentation, model creation, and deployment. We can leverage our large customer base for experimentation to discover user preferences, to personalize user experiences, attract more users, and discover other ways to increase our revenue.
- Internal search and subproblems e.g., autocomplete/typeahead service. Many of our web or mobile applications can have search bars for users to search for their desired data.
- Privacy compliance services and teams. Our expanding user numbers and large amount of customer data will attract malicious external and internal actors, who will attempt to steal data. A privacy breach on our large userbase will affect numerous people and organizations. We must invest in safeguarding user privacy.
- Fraud detection. The increasing revenue of our company will make it a tempting target for criminals and fraudsters, so effective fraud detection systems are a must.

1.4.9 Cloud vs bare metal

We can manage our own hosts and data centers, or outsource this to cloud vendors. This section is a comparative analysis of both approaches.

GENERAL CONSIDERATIONS

At the beginning of this section, we decided to use cloud services (renting hosts from providers such as Amazon's AWS, DigitalOcean, or Microsoft Azure) instead of bare metal (owning and managing our own physical machines).

Cloud providers provide many services we will require, including CI/CD, logging, monitoring, alerting, and simplified setup and management of various database types including caches, SQL, and NoSQL.

If we had chosen bare metal from the beginning, we would have had to set up and maintain any of these services that we require. This may take away attention and time from feature development, which may prove costly to our company.

We must also consider the cost of engineering labor vs cloud tools. Engineers are very expensive resources, and besides being monetarily costly, good engineers tend to prefer challenging work. Bore them with menial tasks such as small-scale setups of common services, and they may move to another company and be difficult to replace in a competitive hiring market.

Cloud tools are often cheaper than hiring engineers to set up and maintain your bare metal infrastructure. We most likely do not possess the economies of scale and their accompanying unit cost efficiencies or the specialized expertise of dedicated cloud providers. If our company is successful, it may reach a growth stage where we have the economies of scale to consider bare metal.

Using cloud services instead of bare metal has other benefits including the following:

SIMPLICITY OF SETUP

On a cloud provider's browser app, we can easily choose a package most suited for our purposes. On bare metal, we would need steps such as installing server software like Apache, or set up network connections and port forwarding.

COST ADVANTAGES

There is no initial upfront cost of purchasing physical machines/servers. A cloud vendor allows us to pay for incremental use, and may offer bulk discounts. Scaling up or down in response to unpredictably changing requirements is easy and fast. If we chose bare metal, we may end up in a situation where we have too few or too many physical machines. Also, some cloud providers offer "auto-scaling" services which automatically resize our cluster to suit the present load.

That being said, cloud is not always cheaper than bare metal. Dropbox (https://www.geekwire.com/2018/dropbox-saved-almost-75-million-two-years-building-tech-infrastructure/) and Uber (https://www.datacenterknowledge.com/uber/want-build-data-centers-uber-follow-simple-recipe) are two examples of companies that host on their own data centers because their requirements meant it was the more cost-efficient choice.

CLOUD SERVICES MAY PROVIDE BETTER SUPPORT AND QUALITY

Anecdotal evidence suggests that cloud services are generally provide superior performance, user experience, support, and have fewer and less serious outages. A possible reason is that cloud services must be competitive in the market to attract and retain customers, compared to bare metal which an organization's users have little choice but to use. Many organizations tend to value and pay more attention to customers than internal users or employees, possibly because customer revenue is directly measurable, while the benefit of providing high quality services and support to internal users may be more difficult to quantify. The corollary is that the losses to revenue and morale from poor quality internal services are also difficult to quantify. Cloud services may also have economies of scale that bare metal lacks, as the efforts of the cloud service's team are spread across a larger user base.

External-facing documentation may be better than internal-facing documentation. It may be better written, updated more often, and be placed on a well-organized website that is easy to search. There may be more resources allocated, so videos and step-by-step tutorials may be provided.

External services may provide higher-quality input validation than internal services. Considering a simple example, if a certain UI field or API endpoint field requires the user to input an email address, the service should validate that the user's input is actually a valid email address. A company may pay more attention to external users who complain about the poor quality of input validation, as they may stop using and paying for the company's product. Similar feedback from internal users who have little choice may be ignored.

When an error occurs, a high-quality service should return instructive error messages that guide the user on how to remedy the error, preferably without the time-consuming process of having to contact support personnel or the service's developers. External services may provide better error messages as well as allocate more resources and incentives to providing high-quality support.

If a customer sends a message, she may receive a reply within minutes or hours, while it may take hours or days to respond to an employee's questions. Sometimes a question to an internal helpdesk channel is not responded to at all. The response to an employee may be to direct her to poorly-written documentation.

An organization's internal services can only be as good as external services if the organization provides adequate resources and incentives. As better user experience and support improves users' morale and productivity, an organization may consider setting up metrics to measure how well internal users are served. One way to avoid these complications is to use cloud services. These considerations can be generalized to external vs internal services.

Lastly, it is the responsibility of individual developers to hold oneself to high standards, but do not make assumptions on the quality of others' work. However, the persistent poor quality of internal dependencies can hurt organizational productivity and morale.

UPGRADES

Both the hardware and software technologies used in an organization's bare metal infrastructure will age, and be difficult to upgrade. This is obvious for finance companies that use mainframes. It is extremely costly, difficult and risky to switch from mainframes to commodity servers, so such companies continue to buy new mainframes, which are far more expensive than their equivalent processing power in commodity servers. Organizations that use commodity servers also need the expertise and effort to constantly upgrade their hardware and software. For example, even upgrading the version of MySQL used in a large organization takes considerable time and effort. Many organizations prefer to outsource such maintenance to cloud providers.

SOME DISADVANTAGES

One disadvantage of cloud providers is vendor lock-in. Should we decide to transfer some or all components of our app to another cloud vendor, this process may not be straightforward. We may need considerable engineering effort to transfer data and services from one cloud provider to another, and pay for duplicate services during this transition.

There are many possible reasons we will want to migrate out of a vendor. Today, the vendor may be a well-managed company that fulfills a demanding SLA at a competitive price, but there is no guarantee this will always be true. The quality of a company's service may degrade in the future, and it may fail to fulfill its SLA. The price may become uncompetitive, as bare metal or other cloud vendors become cheaper in the future. Or the vendor may be found to be lacking in security or other desirable characteristics.

Another disadvantage is lack of ownership over the privacy and security of our data and services. We may not trust the cloud provider to safeguard our data or ensure the security of our services. With bare metal, we can personally verify privacy and security.

For these reasons, many companies adopt a multi-cloud strategy, using multiple cloud vendors instead of a single one, so these companies can migrate away from any particular vendor at short notice should the need suddenly arise.

1.4.10 Serverless - Function as a Service (FaaS)

If a certain endpoint or function is infrequently used, or does not have strict latency requirements, it may be cheaper to implement it as a function on a FaaS platform, such as AWS Lambda or Azure Functions. Running a function only when needed means that there does not need to be hosts continuously waiting for requests to this function.

OpenFaaS and Knative are open-source FaaS solutions that we can use to support FaaS on our own cluster, or as a layer on AWS Lambda to improve the portability of our functions between cloud platforms. As of this book's writing, there is no integration between open-source FaaS solutions and other vendor-managed FaaS such as Azure Functions.

Lambda functions have a timeout of 15 minutes. FaaS is intended to process requests that can complete within this time. In a typical configuration, an API gateway service receives incoming requests and triggers the corresponding FaaS functions. The API gateway is needed because there needs to be a continuously-running service that waits for requests.

Another benefit of FaaS is that service developers need not manage deployments and scaling, and can concentrate on coding their business logic.

Note that a single run of a FaaS function requires steps such as starting a Docker container, starting the appropriate language runtime (Java, Python, node.js, etc.) and running the function, and terminating the runtime and Docker container. This is commonly referred to as "cold start". Frameworks that take minutes to start, such as certain Java frameworks, may be unsuitable for FaaS.

Why is this overhead required? Why can't all functions be packaged into a single package and run across all host instances, similar to a monolith? The reasons are the disadvantages of monoliths (refer to appendix A).

Why not have a frequently-used function deployed to certain hosts for a certain amount of time i.e., with an expiry? Such a system is similar to auto-scaling microservices, and can be considered when using frameworks that take a long time to start.

FaaS has poor portability. An organization which has done much work in a proprietary FaaS like AWS Lambda can become locked-in; migrating to another solution becomes difficult, time-consuming and expensive. This problem becomes especially significant at scale, when FaaS may become much more expensive than bare metal. Open-source FaaS platforms are not a complete solution, because one must provision and maintain one's own hosts, which defeats the scalability purpose of FaaS.

1.4.11 Conclusion - Scaling backend services

In this rest of Part 1, we discuss concepts and techniques to scale a backend service. A frontend/UI service is usually a node.js service, and all it does is serve the same browser app written in a JavaScript framework like ReactJS or Vue.js to any user, so it can be scaled simply by adjusting the cluster size and using GeoDNS. A backend service is dynamic, and can return a different response to each request. Its scalability techniques are more varied and complex. We discussed Functional Partitioning in the above example, and will occasionally touch on it as needed.

1.5 Summary

- System design interview preparation is critical to your career, and also benefits your company.
- The system design interview is a discussion between engineers about designing a software system that is typically provided over a network.
- GeoDNS, caching, and CDN are some basic techniques to scaling our service.
- CI/CD tools and practices allow feature releases to be faster with fewer bugs. They also allow us to divide our users into groups and expose each group to a different version of our app, for experimentation purposes.
- Infrastructure as Code (IaC) tools like Terraform are useful automation tools for cluster management, scaling, and feature experimentation.
- Functional partitioning and centralization of cross-cutting concerns are key elements of system design.
- ETL jobs can be used to spread out the processing of traffic spikes over a longer time period, which reduces our required cluster size.
- Cloud hosting has many advantages. Cost is often but not always an advantage. There are also possible disadvantages such as vendor lock-in and potential privacy and security risks.
- Serverless is an alternative approach to services. In exchange to the cost advantage of not having to keep hosts constantly running, it imposes limited functionality.

2

Non-functional requirements

This chapter covers:

- **Discussing the non-functional requirements of a system at the beginning of the interview.**
- **Techniques and technologies to fulfill various non-functional requirements.**
- **Optimizing for the non-functional requirements.**

A system has functional and non-functional requirements. Functional requirements describe the inputs and outputs of the system. You can represent them as a rough API specification and endpoints.

Non-functional requirements refer to requirements other than the system inputs and outputs. Typical non-functional requirements include the following, to be discussed in detail later in this chapter.

- Scalability: The ability of a system to adjust its hardware resource usage easily and with little fuss to cost-efficiently support its load.
- Availability: The percentage of time a system can accept requests and return the desired response.
- Performance/Latency/P99 and Throughput: Performance or latency is the time taken for a user's request to the system to return a response. Throughput is the maximum request rate that a system can process. It is the inverse of latency.
- Fault-tolerance: The ability of a system to continue operating if some of its components fail, and the prevention of permanent harm (such as data loss) should downtime occur.
- Security: Prevention of unauthorized access to systems.
- Privacy: Access control to personally identifiable information (PII), information which can be used to uniquely identify a person.
- Accuracy: A system's data may not need to be perfectly accurate, and accuracy tradeoffs to improve costs or complexity is often a relevant discussion.

- Consistency: Whether data in all nodes/machines match.
- Cost: We can lower costs by making tradeoffs against other non-functional properties of the system.
- Complexity, Maintainability, Debuggability and Testability: These are related concepts that determine how difficult it is to build a system, and it after it is built.

A customer, whether technical or non-technical, may not explicitly request non-functional requirements and assume that the system will satisfy them. This means that the customer's stated requirements will almost always be incomplete, incorrect, and sometimes excessive. Without clarification, there will be misunderstandings on the requirements. We may not obtain certain requirements, thus inadequately satisfy them, or we may assume certain requirements which are actually not required, and provide an excessive solution.

A beginner is more likely to fail to clarify non-functional requirements, but lack of clarification can occur for both functional and non-functional requirements. We must begin any Systems Design discussion with discussion and clarification of both the functional and non-functional requirements.

Non-functional requirements are commonly traded off against each other. In any System Design interview, we must discuss how various design decisions can be made for various tradeoffs.

It is tricky to separately discuss non-functional requirements and techniques to address them, as certain techniques have trade off gains on multiple non-functional requirements for losses on others. In the rest of this chapter, we briefly discuss each non-functional requirement and some techniques to fulfill it, followed by a detailed discussion of each technique.

2.1 Scalability

Scalability is the ability of a system to adjust its hardware resource usage easily and with little fuss to cost-efficiently support its load.

The process of expanding to support a larger load or number of users is called "scaling". Scaling requires increases in CPU processing power, RAM, storage capacity, and network bandwidth. Scaling can refer to vertical scaling or horizontal scaling.

Vertical scaling is conceptually straightforward and can be easily achieved just by spending more money. It means upgrading to a more powerful and expensive host, one with a faster processor, more RAM, a bigger hard disk drive, a solid-state drive instead of a spinning hard disk for lower latency, or a network card with higher bandwidth. There are 3 main disadvantages of vertical scaling.

Firstly, we will reach a point where monetary cost increases faster than the upgraded hardware's performance. For example, a custom mainframe that has multiple processors will cost more than the same number of separate commodity machines that have one processor each.

Secondly, vertical scaling has technological limits. Regardless of budget, current technological limitations will impose a maximum amount of processing power, RAM, or storage capacity that is technologically-possible on a single host.

Thirdly, vertical scaling may require downtime. We have to stop our host, change its hardware, then start it again. To avoid downtime, we need to provision another host, start our service on it, then direct requests to the new host. Moreover, this is only possible if the service's state is stored on a different machine from the old or new host. As we discuss later in this book, directing requests to specific hosts or storing a service's state in a different host are techniques to achieve many non-functional requirements like scalability, availability, and fault-tolerance.

As vertical scaling is conceptually trivial, in this book, unless otherwise stated, our use of terms like "scalable" and "scaling" refer to horizontally-scalable and horizontal scaling.

Horizontal scaling refers to spreading out the processing and storage requirements across multiple hosts. "True" scalability can only be achieved by horizontal scaling. Horizontal scaling is almost always discussed in a Systems Design interview.

Based on these questions, we determine the customer's scalability requirements.

- How much data coming to system and retrieved from system?
- How many read queries per second?
- How much data per request?
- How many video views per second?
- How big are sudden traffic spikes?

2.1.1 Stateless and stateful services

HTTP is a stateless protocol, so a backend service that uses it is easy to scale horizontally. Chapter 3 describes horizontal scaling of database reads. A stateless HTTP backend combined with horizontally-scalable database read operations is a good starting point to discuss a scalable system design.

Refer to chapter 7 for a discussion of various common communication architectures, including the tradeoffs between stateful and stateless.

2.1.2 Scaling writes to a shared storage is difficult

Writes to shared storage are the most difficult to scale. We discuss techniques including replication, compression, aggregation, denormalization, and Metadata Service later in this book.

2.1.3 Basic load balancer concepts

Every horizontally-scaled service uses a load balancer, which may be one of the following:

1. A hardware load balancer, a specialized physical device with that distributes traffic across multiple hosts. Hardware load balancers are known for being expensive, and can cost anywhere from a few thousand to few hundred thousand dollars.
2. A shared load balancer service, also referred to as LBaaS (load balancing as a service).
3. A server with load balancing software installed. HAProxy and Nginx are the most common.

This section discusses some basic concepts of load balancers that we may use in an interview.

In the system diagrams in this book, we draw rectangles to represent various services or other components, and arrows between them to represent requests. It is usually understood that requests to a service go through a load balancer and are router to a service's hosts. We usually do not illustrate the load balancers themselves.

We can tell the interviewer that we need not include a load balancer component in our system diagrams, as it is implied, and drawing it and discussing it on our system diagrams is a distraction from the other components and services that compose our service.

LEVEL 4 VS LEVEL 7

We should be able to distinguish between level 4 and level 7 load balancers, and discuss which one is more suitable for any particular service. A level 4 load balancer operates at the transport layer (TCP), so it cannot inspect packet contents; it can only forward the packets. A level 7 load balancer operates at the application layer (HTTP), so it has these capabilities:

1. Make load balancing/routing decisions based on a packet's contents.
2. Authentication. It can return 401 if a specified authentication header is absent.
3. TLS termination. Security requirements for traffic within a data center may be lower than traffic over the Internet, so performing TLS termination means there is no encryption/decryption overhead between data center hosts. If our application requires traffic within our data center to be encrypted (i.e., encryption in transit), we will not do TLS termination.

STICKY SESSIONS

Sticky session refers to a load balancer sending requests from a particular client to a particular host for a duration set by the load balancer or the application. Sticky sessions are used for stateful services. The main advantage of a sticky session is that a host can use its RAM for caching, so the number of network hops is reduced, and overall latency is improved.

A sticky session can be implemented using duration-based or application-controlled cookies. In a duration-based session, the load balancer issues a cookie to a client that defines a duration. Each time the load balancer receives a request, it checks the cookie. In an application-controlled session, the application generates the cookie. The load balancer still issues its own cookie on top of this application-issued cookie, but the load balancer's cookie follows the application cookie's lifetime. This approach ensures clients are not routed to another host after the load balancer's cookie expires, but is more complex to implement because it requires additional integration between the application and the load balancer.

SESSION REPLICATION

In session replication, writes to a host are copied to several other hosts in the cluster that are assigned to the same session. These hosts may form a backup ring. E.g., if there are 3 hosts in a session, when host A receives a write, it writes to host B, which in turn writes to host C. Another way is for the load balancer to make write requests to all the hosts assigned to a session. Reads can be routed to any host with that session.

OTHER NOTES

You may come across the term "reverse proxy" in other system design interview preparation materials. If you're unfamiliar with the term "reverse proxy", https://www.nginx.com/resources/glossary/reverse-proxy-vs-load-balancer/ is a good article.

FURTHER READING

https://www.cloudflare.com/learning/performance/types-of-load-balancing-algorithms/ is a good brief description of various load balancing algorithms.

https://rancher.com/load-balancing-in-kubernetes is a good introduction to load balancing in Kubernetes.

https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancerhttps://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/ describes how to attach an external cloud service load balancer to a Kubernetes service.

2.2 Availability

Availability is the percentage of time a system can accept requests and return the desired response. Common benchmarks for availability are shown on table 2.1.

Table 2.1 Common benchmarks for availability.

Availability %	Downtime per year	Downtime per month	Downtime per week	Downtime per day
99.9 (three nines)	8.77 hours	43.8 minutes	10.1 minutes	1.44 minutes
99.99 (four nines)	52.6 minutes	4.38 minutes	1.01 minutes	8.64 seconds
99.999 (five nines)	5.26 minutes	26.3 seconds	6.05 seconds	864 milliseconds

Refer to https://netflixtechblog.com/active-active-for-multi-regional-resiliency-c47719f6685b for a detailed discussion on Netflix's multi-region active-active deployment for high availability. In this book, we discuss similar techniques for high availability, such as replication within and across data centers in different continents. We also discuss monitoring and alerting.

High availability is required in most services, and other non-functional requirements may be traded off to allow high availability without unnecessary complexity.

When discussing the non-functional requirements of a system, first establish whether high availability is required. Do not assume that strong consistency and low latency are required. Discuss if we can trade them off for higher availability. As far as possible, suggest using asynchronous communication techniques that accomplish this, such as event sourcing and saga, discussed in chapters 5 and 6.

Services where requests do not need to be immediately processed and responses immediately returned are unlikely to require strong consistency and low latency, such as

programmatic requests or requests made between services. Examples include logging to long-term storage or sending a request in Airbnb to book a room for some days from now.

Use synchronous communication protocols such as REST and RPC only when an immediate response is absolutely necessary, typically for requests made directly by people using your app.

Nonetheless, do not assume that requests made by people need immediate responses with the requested data. Consider if the immediate response can be an acknowledgement, and the requested data can be returned minutes or hours later. For example, if a user requests to transfer money between two accounts, this transfer need not happen immediately. The service can queue the request internally and immediately respond to the user that the request will be processed in minutes or hours. The transfer can be later processed by a streaming job or a periodic batch job, then the user can be notified of the result (such as whether the transfer succeeded or failed) through channels such as email, text, or app notifications.

An example of a situation where high availability may not be required is in-memory caching. As in-memory caching may be used to reduce the latency and network traffic of a request, and not actually strictly needed to fulfill the request, we may decide to trade off availability for lower latency.

Another example is rate limiting, discussed in chapter 12.

Availability can also be measured with incident metrics. https://www.atlassian.com/incident-management/kpis/common-metrics describes various incident metrics like MTTR (Mean Time to Recovery) and MTBF (Mean Time Between Failures).

2.3 Fault-tolerance

Fault-tolerance is the ability of a system to continue operating if some of its components fail, and the prevention of permanent harm (such as data loss) should downtime occur. This allows graceful degradation, so our system can maintain some functionality when parts of it fail, rather than a complete catastrophic failure. This buys engineers time to fix the failed sections and restore the system to working order.

Availability and fault-tolerance are often discussed together. While availability is a measure of uptime/downtime, fault-tolerance is not a measure, but rather the system's characteristics.

A closely related concept is Failure Design, which is about smooth error handling. Consider how we will handle errors in 3rd party APIs which are outside our control as well as silent/undetected errors.

Techniques for fault-tolerance include the following.

2.3.1 Replication and Redundancy

Replication is discussed in chapter 3.

One replication technique is to have multiple (such as 3) redundant instances/copies of a component, so up to 2 can be simultaneously down without affecting uptime. As discussed in the Replication chapter, update operations are usually assigned a particular host (which in

designs other than single-leader depends on the value being updated), so update performance is affected only if the other hosts are on different datacenters geographically further away from the requester, but reads are often done on all replicas, so read performance decreases when components are down.

1 instance is designated as the source of truth (often called the leader) while the other 2 components are designated as replicas (or followers). There are various possible arrangements of the replicas. 1 replica on a different server rack within the same datacenter and another replica in a different datacenter. Another arrangement is to have all 3 instances on different datacenters, which maximizes fault-tolerance with trade off of lower performance.

An example is the Hadoop Distributed File System (HDFS), which has a configurable property called "replication factor" to set the number of copies of any block. The default value is 3.

Replication also helps to increase availability.

2.3.2 Forward Error Correction (FEC) and Error Correction Code (ECC)

FEC is a technique to prevent errors in data transmission over noise or unreliable communication channels by encoding the message in a redundant way such as by using an ECC.

FEC is a protocol-level rather than system-level concept. We can mention FEC and ECC during System Design interviews, but it is unlikely that we will need to explain it in detail.

2.3.3 Circuit Breaker

Circuit Breaker is a mechanism that stops a client from repeatedly attempting an operation that is likely to fail. With respect to downstream services, a Circuit Breaker calculates the number of requests that failed within a recent interval. If an error threshold is exceeded, the client stops calling downstream services. Sometime later, the client attempts a limited number of requests. If they are successful, the client assumes that the failure is resolved and resumes sending requests without restrictions.

If a service B depends on a service A, A is the upstream service and B is the downstream service.

Circuit Breaker saves resources from being spent to make requests that are likely to fail. It also prevents clients from adding additional burden to an already overburdened system.

However, Circuit Breaker makes the system more difficult to test. For example, a load test that may previously overwhelm downstream services and fail will pass with a Circuit Breaker. It is also difficult to estimate the appropriate error threshold and timers.

A circuit breaker can be implemented on the server side. An example is Resilience4j (https://github.com/resilience4j/resilience4j). It was inspired by Hystrix (https://github.com/Netflix/Hystrix), which was developed at Netflix and transitioned to

maintenance mode in 2017 (https://github.com/Netflix/Hystrix/issues/1876#issuecomment-440065505). Netflix's focus has shifted towards more adaptive implementations that react to an application's real time performance rather than pre-configured settings, such as adaptive concurrency limits (https://netflixtechblog.medium.com/performance-under-load-3e6fa9a60581).

2.3.4 Exponential backoff and retry

Exponential backoff and retry is similar to Circuit Breaker. When a client receives an error response, it will wait before reattempting the request, and exponentially increase the wait duration between retries. The client also adjusts the wait period by a small random negative or positive amount, a technique called "jitter". This prevents multiple clients from submitting retries at exactly the same time, causing a "retry storm" that may overwhelm the downstream service. Similar to Circuit Breaker, when a client receives a success response, it assumes that the failure is resolved and resumes sending requests without restrictions.

2.3.5 Caching responses of other services

Our service may depend on external services for certain data. How should we handle the case where an external service is unavailable? It is generally preferable to have graceful degradation instead of crashing or returning an error. We can use a default or empty response in place of the return value. If using stale data is better than no data, we can cache the external service's responses whenever we make successful requests, and use these responses when the external service is unavailable.

2.3.6 Checkpointing

A machine may perform certain data aggregation operations on many data points by systematically fetching a subset of them, performing the aggregation on them, then writing the result to a specified location, repeating this process until all data points are processed or infinitely such as in the case of a streaming pipeline. Should this machine fail during data aggregation, the replacement machine should know which data points to resume the aggregation. This can be done by writing a checkpoint after each subset of data points are processed and the result is successfully written. The replacement machine can resume processing at the checkpoint.

Checkpointing is commonly applied to ETL pipelines that use message brokers such as Kafka. A machine can fetch several events from a Kafka topic, process the events then write the result followed by writing a checkpoint. Should this machine fail, its replacement can resume at the most recent checkpoint.

Kafka offers offset storages at the partition level in Kafka (https://kafka.apache.org/22/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html). Flink consumes data from Kafka topics and periodically checkpoints using Flink's distributed checkpointing mechanism (https://ci.apache.org/projects/flink/flink-docs-master/docs/dev/datastream/fault-tolerance/checkpointing/).

2.3.7 Dead Letter queue

If a write request to a 3rd party API fails, we can queue the request in a dead letter queue and try the requests again later.

Are dead letter queues stored locally or on a separate service? We can trade off complexity and reliability.

1. Simplest: If it is acceptable to miss requests, just drop failed requests.
2. Implement the dead letter queue locally with a try-catch block. Requests will be lost if host fails.
3. A more complex and reliable option is to use an event streaming platform like Kafka.

In an interview, you should discuss multiple approaches and their tradeoffs. Don't just state one approach.

2.3.8 Logging and periodic auditing

One method to handle silent errors is to log our write requests and perform periodic auditing. An auditing job can process the logs and verify that the data on service we write to matches the expected values. This is discussed further in chapter 10 and chapter 14 (Periodic Auditing Service).

2.3.9 Bulkhead

The bulkhead pattern is a fault-tolerance mechanism where a system is divided into isolated pools, so a fault in one pool will not affect the entire system.

For example, the various endpoints of a service can each have their own thread pool, and not share a thread pool, so if an endpoint's thread pool is exhausted, this will not affect the ability of other endpoints to serve requests. (Indrasiri and Siriwardena)

Another example of bulkhead is discussed by Nygard. A certain request may cause a host to crash due to a bug. Each time this request is repeated, it will crash another host. Dividing the service into bulkheads i.e. dividing the hosts into pools prevents this request from crashing all the hosts and causing a total outage. This request should be investigated, so the service must have logging and monitoring. Monitoring will detect the offending request, and engineers can use the logs to troubleshoot the crash and determine its cause.

Or a requestor may have a high request rate to a service and prevent the latter from serving other requestors. The bulkhead pattern allocates certain hosts to a particular requestor, preventing the latter from consuming all the service's capacity. (Rate limiting discussed in chapter 12 is another way to prevent this situation.)

A service's hosts can be divided into pools, and each pool is allocated requestors. This is also a technique to prioritize certain requestors by allocating more resources to them.

In figure 2.1, a service serves two other services. Unavailability of the service's hosts will prevent it from serving any requestor.

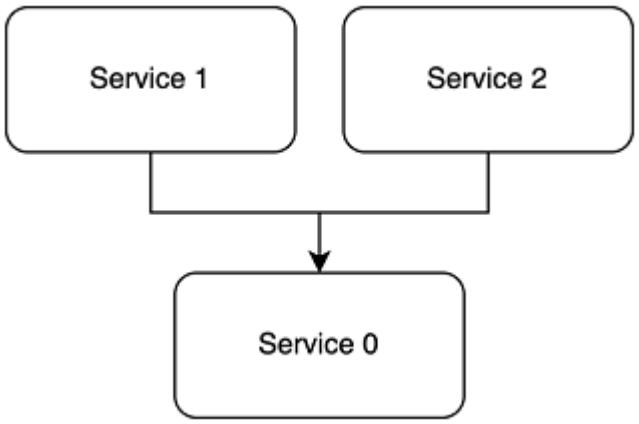

Figure 2.1 All requests to service 0 are load-balanced across its hosts. Unavailability of service 0's hosts will prevent it from serving any requestor. Figure from Nygard (refer to text).

In figure 2.2, a service's hosts are divided into pools which are allocated to requestors. Unavailability of the hosts of one pool will not affect other requestors.

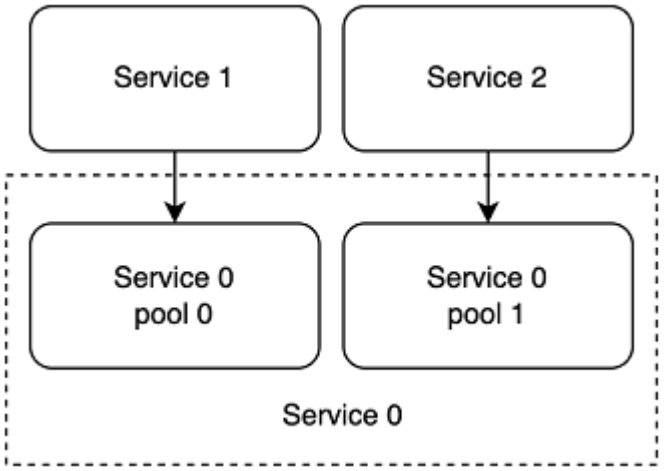

Figure 2.2 Service 0 is divided into two pools, each allocated to a requestor. Unavailability of one pool will not affect the other. Figure from Nygard (refer to text).

Refer to Michael Nygard's book "Release It!: Design and Deploy Production-Ready Software (2018), The Pragmatic Bookshelf" for other examples of the bulkhead pattern.

We will not mention bulkhead in the system design discussions of part 2, but it is generally applicable to most systems and you can discuss it during an interview.

2.4 Performance/latency and throughput

Performance or latency is the time taken for a user's request to the system to return a response. A typical request on a consumer-facing app (e.g. view a restaurant's menu on a food delivery app or submit a payment on an ecommerce app) has a desired latency of tens of milliseconds to several seconds. Enterprise apps may have lower latency requirements. High-frequency trading applications may demand latency of several milliseconds.

Strictly speaking, latency refers to the travel time of a packet to from its source to its destination. However, the term "latency" has become commonly-used have the same meaning as "performance", and both terms are often used interchangeably. We still use the term "latency" if we need to discuss packet travel time.

The term "latency" can also be used to describe the request-response time between components within the system, rather than the user's request-response time. For example, if a backend host makes a request to a logging or storage system to store data, the system's latency is the time required to log/store the data and return a response to the backend host.

The system's functional requirements may mean that a response may not actually need to contain the information requested by the user, but may simply be an acknowledgement along with a promise that after a specified duration, the requested information will be sent to the user or will be available for the user to obtain by making another request. Such a tradeoff may simplify the system's design, so we must always clarify requirements and discuss how soon information is required after a user's request.

Throughput is the maximum request rate that a system can process. It is the inverse of latency. A system with low latency has high throughput.

Typical design decisions to achieve low latency include the following.

We can deploy the service in a datacenter geographically close to its users, so packets between users and our service do not need to travel far. If our users are geographically-dispersed users, we may deploy our service in multiple datacenters which are chosen to minimize geographical distance to clusters of users. If hosts across datacenters need to share data, our service must be horizontally scalable. We describe scalability in a later section.

Occasionally, there may be other factors that contribute more to latency than the physical distance between users and datacenters, such as traffic or network bandwidth. We can use test requests between users and various datacenters to determine the datacenter with the lowest latency for users in a particular location.

Other techniques include using a CDN, caching, decreasing the data size with RPC instead of REST, and designing your own protocol with a framework like Netty to use TCP and UDP instead of HTTP, and using batch and streaming techniques.

In examining latency and throughput, we discuss the characteristics of the data and how it gets in and out of the system, then we can suggest strategies. Can we count views several hours after they happened? This will allow batch or streaming approaches. What is the

response time? If small, data must already be aggregated, and aggregation should be done during writes, with minimal or no aggregation during reads.

2.5 Consistency

Consistency has different meanings in ACID and CAP (from the CAP theorem). ACID consistency focuses on data relationships like foreign keys and uniqueness. As stated in Designing Data-intensive Applications, CAP consistency is actually linearizability, defined as all nodes containing the same data at a moment in time, and changes in data must be linear i.e., all nodes must start serving the changes at the same time.

Table 2.2 Databases that favor availability vs linearizability.

Favor availability	Favor linearizability
HBase MongoDB Redis	Cassandra CouchDB Dynamo Hadoop Riak

Eventually-consistent databases trade off consistency for improvements in availability, scalability, and latency. An ACID database including RDBMS databases cannot accept writes when it experiences a network partition, because it cannot maintain ACID consistency if writes occur during a network partition. Summarized on table 2.2, MongoDB, HBase, and Redis trade off availability for linearizability, while CouchDB, Cassandra, Dynamo, Hadoop, and Riak trade off linearizability for availability.

During the discussion, we should emphasize the distinction between ACID and CAP consistency, and the tradeoffs between linearizability vs eventual consistency. In this book, we will discuss various techniques for linearizability and eventual consistency, including the following.

Techniques for linearizability:

- Full mesh
- Quorum.

Techniques for eventual consistency that involve writing to a single location, which propagates this write to the other relevant locations.

- Event sourcing (chapter 4.1), a technique to handle traffic spikes.
- Coordination service.
- Distributed cache.

Techniques for eventual consistency that trade off consistency and accuracy for lower cost

- Gossip protocol
- Random leader selection

Disadvantages of linearizability include the following:

- Lower availability, since most or all nodes must be sure of consensus before they can serve requests. This becomes more difficult with a larger number of nodes.
- More complex and expensive.

2.5.1 Full Mesh

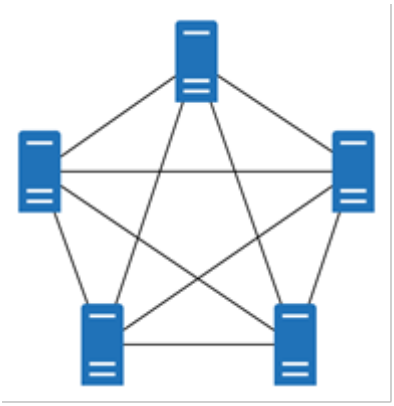

Figure 2.3 Illustration of full mesh. Every host is connected to every other host.

Figure 2.3 illustrates an example of full mesh. Every host in the cluster has the address of every other host, and broadcasts messages to all of them.

How do hosts discover each other? When a new host is added, how is its address sent to other hosts? Solutions for host discovery include:

- Maintain the list of addresses in a configuration file. Each time the list changes, we will need to deploy this file across all hosts/nodes.
- Use a 3rd party service that listens for heartbeats from every host. A host is kept registered as long as the service receives heartbeats. All hosts use this service to obtain the full list of addresses.

Full mesh is easier to implement than other techniques, but is not scalable. The number of messages grows quadratically with the number of hosts. Full mesh works well for small clusters but cannot support big clusters.

In quorum, every host does not need to have the same data for the system to be considered consistent. A client

During an interview, we can briefly mention full mesh, and compare it with scalable approaches.

2.5.2 Coordination Service

Figure 2.4 illustrates a coordination service, a 3rd party component that chooses a leader node or set of leader nodes. Having a leader decreases the number of messages. All other nodes send their messages to the leader, and the leader may do some necessary processing

and send back the final result. Each node only needs to communicate with its leader or set of leaders, and each leader manages a number of nodes.

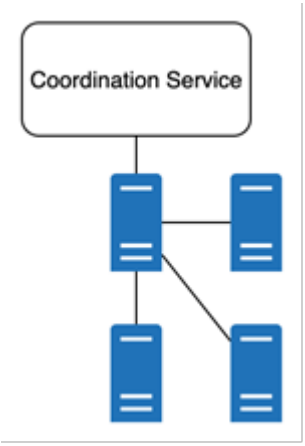

Figure 2.4 Illustration of a coordination service.

Example algorithms are Paxos, Raft, and Zab. Another example is single leader multiple follower in SQL (section 3.2.2), a technique to allow scalable reads. ZooKeeper (https://zookeeper.apache.org/) is a distributed coordination service. ZooKeeper has the following advantages over a config file stored on a single host. (Most of these advantages are discussed at https://stackoverflow.com/q/36312640/1045085.) We can implement these features on distributed filesystem or distributed database, but ZooKeeper already provides them.

- Access control (https://zookeeper.apache.org/doc/r3.1.2/zookeeperProgrammers.html#sc_ZooKeeperAccessControl).
- ZooKeeper stores data in memory for high performance.
- Scalability, with horizontal scaling by adding hosts to the ZooKeeper Ensemble (https://zookeeper.apache.org/doc/r3.1.2/zookeeperAdmin.html#sc_zkMulitServerSetup).
- Guaranteed eventual consistency within a specified time bound, or strong consistency with higher cost (https://zookeeper.apache.org/doc/current/zookeeperInternals.html#sc_consistency). ZooKeeper trades off availability for consistency; it is a CP system in the CAP theorem.
- Ordering. Clients can read data in the order it is written.

Complexity is the main disadvantage of a coordination service. A coordination service is a sophisticated component that has to be highly reliable, and ensure 1 and only 1 leader is elected. (The situation where 2 nodes both believe they are the leader is called "split brain".

Refer to Kleppmann, Martin, Designing Data-Intensive Applications, O'Reilly, 2017, page 158.)

2.5.3 Distributed Cache

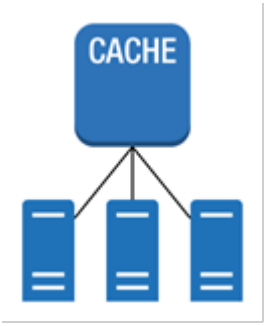

Figure 2.5 Illustration of using a distributed cache to broadcast messages. The nodes can make requests to an in-memory store like Redis to update data, or periodic requests to fetch new data.

We can use a distributed cache like Redis. Referring to figure 2.5, our service's nodes can make periodic requests to the origin to fetch new data, then make requests to the distributed cache (e.g. an in-memory store like Redis) to update its data. This solution is simple, has low latency, and the distributed cache cluster can be scaled independently of our service. However, this solution has more requests than every other solution here except the full mesh.

Redis is an in-memory cache, not a typically distributed one by definition. It is used as a distributed cache for practical intents and purposes. Refer to https://redis.io/docs/about/ and https://stackoverflow.com/questions/18376665/redis-distributed-or-not.

Both a sender and receiver host can validate that a message contains its required fields. This is often done by both sides because the additional cost is trivial while reducing the possibility of errors in either side resulting in an invalid message. When a sender host sends an invalid message to a receiver host via a HTTP request, and the receiver host can detect that this message is invalid, it can immediately return a 400 or 422. We can set up high-urgency alerts to trigger on 4xx errors, so we will immediately be alerted of this error and can immediately investigate. However, if we use Redis, invalid data written by a node may stay undetected until it is fetched by another node, so there will be a delay in alerts.

Requests sent directly from one host to another goes through schema validation. However, Redis is just a database, so it does not validate schema, and hosts can write arbitrary data to it. This may create security issues. (Refer to

https://www.trendmicro.com/en_us/research/20/d/exposed-redis-instances-abused-for-remote-code-execution-cryptocurrency-mining.html and https://www.imperva.com/blog/new-research-shows-75-of-open-redis-servers-infected.) Redis is designed to be accessed by trusted clients inside trusted environments (https://redis.io/topics/security). Redis does not support encryption, which may be a privacy concern. Implementing encryption at rest increases complexity, costs and reduces performance (https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/at-rest-encryption.html).

A coordination service addresses these disadvantages, but has higher complexity and cost.

2.5.4 Gossip Protocol

Gossip protocol is modeled after how epidemics spread. Referring to figure 2.6, each node randomly selects another node periodically or with a random interval, then shares data. This approach trades off consistency for lower cost and complexity.

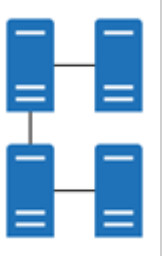

Figure 2.6 Illustration of gossip communication.

2.5.5 Random Leader Selection

Referring to figure 2.7, random leader selection uses a simple algorithm to elect a leader. This simple algorithm does not guarantee 1 and only 1 leader, so there may be multiple leaders. This is a minor issue, as each leader can share data with all other hosts, so all hosts including all leaders will have the correct data. The disadvantage is possible duplicate requests and unnecessary network traffic.

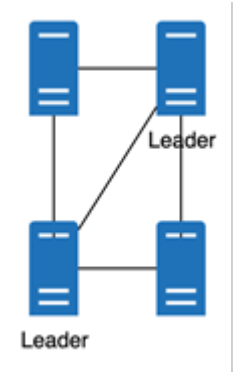

Figure 2.7 Illustration of multiple leaders, which can result from random leader selection.

2.6 Accuracy

Accuracy is a relevant non-functional requirement in systems with complex data processing or high rate of writes.

Estimation algorithms trade off accuracy for lower complexity. Examples of estimation algorithms include HyperLogLog for cardinality (COUNT DISTINCT) estimate in the Presto distributed SQL query engine, and count-min sketch for estimating frequencies of events in a stream of data.

A cache is stale if the data in its underlying database has been modified. A cache may have a refresh policy where it will fetch the latest data at a fixed periodic interval. A short refresh policy is more costly. An alternative is for the system to update or delete the associated cache key when data is modified, which increases complexity.

Accuracy is somewhat related to consistency. Eventually-consistent systems trade off accuracy for improvements in availability, complexity, and cost. When a write is made to an eventually-consistent system, results from reads made after this write may not include the effects of the write, which makes them inaccurate. The eventually-consistent system is inaccurate until the replicas are updated with the effects of the write operation. However, we use the term "consistency" to discuss such a situation, not "accuracy".

2.7 Complexity and Maintainability

The first step to minimize complexity is to clarify both functional and non-functional requirements, so we do not design for unnecessary requirements.

As we sketch design diagrams, note which components may be separated into independent systems. Use common services to reduce complexity and improve maintainability. Common services which are generalizable across virtually all services include:

- Load balancer service.
- Rate limiting. Refer to chapter 12.
- Authentication and authorization. Refer to appendix C.

- Usage data collection. Refer to chapter 9.
- Logging, monitoring, alerting. Refer to chapter 9.
- TLS termination. Refer to other sources for more information.
- Caching. Refer to chapter 8.
- DevOps and CI/CD if applicable. This is outside the scope of this book.

Services are which generalizable for certain organizations, such as those that collect user data for Data Science, include analytics and machine learning.

Complex systems may require yet more complexity for high availability and high fault-tolerance. If a system that has an unavoidable degree of complexity, consider tradeoffs of complexity for lower availability and fault-tolerance.

Discuss possible trade offs in other requirements to improve complexity, such as ETL pipelines to delay data processing operations that need not occur in real-time.

A common technique to trade off complexity for better latency and performance is to use techniques that minimize the size of messages in network communication. Such techniques include RPC serialization frameworks and Metadata Services. (Refer to section 5.5 for a discussion on Metadata Service.)

RPC serialization frameworks such as Avro, Thrift, and protobuf can reduce message size at the expense of maintaining schema files. (Refer to chapter 7 for a discussion of REST vs RPC.) We should always suggest using such serialization frameworks in any interview, and will not mention this point again in the book.

We should also discuss how outages can occur, evaluate the impact of various outages to the users and the business, and how to prevent and mitigate outages. Common concepts include replication, failover, and authoring runbooks. Runbooks are discussed in chapter 12.

We will discuss complexity in all chapters of part 2.

2.7.1 Continuous Deployment (CD)

Continuous Deployment (CD) was first mentioned in this book in section 1.4.5. As mentioned in that section, CD allows easy deployments and rollbacks. We have a fast feedback cycle, that improves our system's maintainability. If we accidentally deploy a buggy build to production, we can easily roll it back. Fast and easy deployments of incremental upgrades and new features lead to a fast software development lifecycle. This is a major advantage of services over monoliths, as discussed in appendix A.

Other CD techniques include blue/green deployments, also referred to as zero downtime deployments. Refer to sources such as https://spring.io/blog/2016/05/31/zero-downtime-deployment-with-a-database, https://dzone.com/articles/zero-downtime-deployment, and https://craftquest.io/articles/what-are-zero-downtime-atomic-deployments for more information.

Static code analysis tools like SonarQube (https://www.sonarqube.org/) also improves our system's maintainability.

2.8 Cost

In system design discussions, we can suggest trading off other non-functional requirements for lower cost. Examples:

- Higher cost for lower complexity by vertical scaling instead of horizontal scaling.
- Lower availability for improved costs by decreasing the redundancy of a system (such as the number of hosts, or the replication factor in a database).
- Higher latency for improved costs by using a data center in a cheaper location that is further away from users.

Discuss the cost of implementation, cost of monitoring, and cost of each non-functional requirement such as high availability.

Production issues vary in seriousness and in how quickly they must be addressed and resolved, so do not implement more monitoring and alerting than required. Cost is higher if engineers need to be alerted to an issue as soon as it occurs, compared to when it is permissible for alerts to be created hours after an issue.

Besides the cost of maintenance in the form of addressing possible production issues, there will also be costs due to the natural atrophy of software over time as libraries and services are deprecated. Identify components which may need future updates. Which dependencies (such as libraries) will prevent other components from being easily updated should these dependencies become unsupported in the future? How may we design our system to more easily replace these dependencies should updates be required?

How likely do we need to change dependencies in the future, particularly third-party dependencies where we have less control? Third-party dependencies may be decommissioned or prove unsatisfactory for our requirements, such as reliability or security issues.

Consider the costs to decommission the system if necessary. We may decide to decommission the system for multiple reasons, such as the team deciding to change its focus, or if the system has too few users to justify its development and maintenance costs. We may decide to provide the existing users with their data, so we will need to extract the data into various text and/or CSV files for our users.

2.9 Security

During an interview, we may need to discuss possible security vulnerabilities in our system, and how we will prevent and mitigate security breaches.

includes both external and internal access.

The following topics are commonly discussed with regard to your system:

- TLS termination vs keeping data encrypted between services or hosts in a data center (called encryption in transit). TLS termination is usually done to save processing, as encryption between hosts in a data center is usually not required. There may be exceptions for sensitive data, on which we use encryption in transit.
- Which data can be stored unencrypted and which should be stored encrypted (called encryption at rest). Encryption at rest is conceptually different from storing hashed data.

We should have some understanding of OAuth 2.0 and OpenID Connect, which are described in appendix C.

We may also discuss rate limiting to prevent DDoS attacks. A rate limiting system can be asked as its own interview question, discussed in chapter 12. It should be mentioned during design of almost any external facing system.

2.10 Privacy

Personally Identifiable Information (PII) is data that can be used to uniquely identify a customer, such as full name, government identifiers, addresses, email addresses, and bank account identifiers. PII must be safeguarded to comply with regulations such as the General Data Protection Regulation (GDPR) and California Consumer Privacy Act (CCPA). This includes both external and internal access.

Within our system, access control mechanisms should be applied to PII stored in databases and files. We can use mechanisms such as the Lightweight Directory Access Protocol (LDAP). We can encrypt data both in transit (using SSL) and at rest.

Consider using hashing algorithms such as SHA-2 and SHA-3 to mask PII and maintain individual customer privacy in computing aggregate statistics e.g., mean number of transactions per customer.

If PII is stored on an append-only database or file system like HDFS, a common privacy technique is to assign each customer an encryption key. The encryption keys can be stored in a mutable storage system like SQL. Data associated with a particular customer should be encrypted with her encryption key before it is stored. If a customer's data needs to be deleted, all that has to be done is to delete the customer's encryption key, then all of the customer's data on the append-only storage becomes inaccessible and hence effectively deleted.

We can discuss the complexity, cost, and effects of privacy on many aspects such as customer service or personalization including machine learning.

We should also discuss prevention and mitigation strategies for data breaches.

2.10.1 External vs. Internal services

If we design an external service, we definitely should design security and privacy mechanisms. What about internal services that only serves other internal services? We may decide to rely on the security mechanisms of our user services against malicious external attackers, and assume that internal users will not attempt malicious actions, so security measures are not required for our rate limiter service. We may also decide that we trust our user services not to request data about rate limiter requestors from other user services, so privacy measures are not required.

However, it is likely that we will decide that our company should not trust internal users to properly implement security mechanisms, should not trust that internal users are not malicious, and should not trust internal users to not inadvertently or maliciously violate our customer's privacy. We should adopt an engineering culture of implementing security and privacy mechanisms by default. This is consistent with the internal access controls and privacy policies of all kinds of services and data adopted by most organizations. For example, most organizations have role-based access control for each service's Git repository and CI/CD. Most organizations also have procedures to grant access to employee and customer

data only to persons they deem necessary to have access to this data. These access controls and data access are typically limited in scope and duration as much as possible. There is no logical reason to adopt such policies for certain systems and not adopt them for others. We should ensure that our internal service does not expose any sensitive features or data before we decide that it can exclude security and privacy mechanisms. Moreover, every service, external or internal, should log access to sensitive databases.

Another privacy mechanism is to have a well-defined policy for storing user information. Databases that store user information should be behind services that are well documented, and have tight security and strict access control policies. Other services and databases should only store user IDs, and no other user data. The user IDs can be changed either periodically or in the event of a security or privacy breach.

Figure 1.8 illustrates a service mesh, including security and privacy mechanisms, illustrated as an external request to an identity and access management service.

Refer to chapter 10 for a discussion on system requirements in an interview.

2.11 Cloud Native

Cloud Native is an approach to address non-functional requirements including scalability, fault-tolerance, and maintainability. The definition of Cloud Native by the Cloud Native Computing Foundation is as follows (https://github.com/cncf/toc/blob/main/DEFINITION.m). We bold certain words for emphasis.

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. **Containers, service meshes, microservices, immutable infrastructure**, and **declarative APIs** exemplify this approach.

These techniques enable **loosely-coupled** systems that are **resilient, manageable**, and **observable**. Combined with **robust automation**, they allow engineers to make high-impact **changes frequently** and **predictably** with minimal toil.

The Cloud Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone.

This is not a book on Cloud Native, but we utilize cloud native techniques (containers, service meshes, microservices, serverless functions, immutable infrastructure or Infrastructure as Code (IaC), declarative APIs, automation) throughout this book to achieve the benefits (resilient, manageable, observable, allow frequent and predictable changes), and include references to materials on the relevant concepts.

2.12 Further reading

Interested readers can look up the PACELC theorem, which we do not discuss in this book.

A useful resource that has content similar to this chapter is Microservices for the Enterprise: Designing, Developing, and Deploying (2018, Apress) by Kasun Indrasiri and Prabath Siriwardena.

2.13 Summary

- We must discuss both the functional and non-functional requirements of a system. Do not make assumptions on the non-functional requirements. Non-functional characteristics can be traded off against each other to optimize for the non-functional requirements.
- Scalability is the ability to easily adjust the system's hardware resource usage for cost efficiency. This is almost always discussed, as it is difficult or impossible to predict the amount of traffic to our system.
- Availability is the percentage of time a system can accept requests and return the desired response. Most but not all systems require high availability, so we should clarify if it is a requirement in our system.
- Fault-tolerance is the ability of a system to continue operating if some components fail, and the prevention of permanent harm should downtime occur. This allows our users to continue using some features, and buys time for engineers to fix the failed components.
- Performance or latency is the time taken for a user's request to the system to return a response. Users expect interactive applications to load fast and respond quickly to their input.
- Consistency is defined as all nodes containing the same data at a moment in time, and when changes in data occur, all nodes must start serving the changed data at the same time. In certain systems such as financial systems, multiple users viewing the same data must see the same values, while in other systems such as social media, it may be permissible for different users to view slightly different data at any point in time, as long as the data is eventually the same.
- Eventually consistent systems trade-off accuracy for lower complexity and cost.
- Complexity must be minimized so the system is cheaper, and easier to build and maintain. Use common techniques such as common services wherever applicable.
- Cost discussions include minimizing complexity, cost of outages, cost of maintenance, cost of switching to other technologies, and cost of decommissioning.
- Security discussions include which data must be secured and which can be unsecured, followed by using concepts such as encryption in transit and encryption at rest.
- Privacy considerations include access control mechanisms and procedures, deletion or obfuscation of user data, and prevention and mitigation of data breaches.
- Cloud native is an approach to system design that employs a collection of techniques to achieve common non-functional requirements.

3

Scaling databases

This chapter covers:

- **Various types of storage services.**
- **Replicating databases.**
- **Aggregating events to reduce database writes.**
- **Normalization vs denormalization.**
- **Caching frequent queries in memory.**

In this chapter, we discuss concepts in scaling databases, their tradeoffs, and common databases that utilize these concepts in their implementation. We consider these concepts when choosing databases for various services in our system.

3.1 Brief prelude on storage services

Storage services are stateful services. Compared to stateless services, *stateful services* have mechanisms to ensure consistency, and require redundancy to avoid data loss. A stateful service may choose mechanisms like Paxos for strong consistency or eventual-consistency mechanisms. These complex decisions and tradeoffs have to be made, and they depend on the various requirements like consistency, complexity, security, latency, performance. This is one reason we keep all services stateless as much as possible, and keep state only in stateful services.

In strong consistency, all accesses are seen by all parallel processes (or nodes, processors, etc.) in the same order (sequentially). Therefore, only one consistent state can be observed, as opposed to weak consistency, where different parallel processes (or nodes, etc.) can perceive variables in different states.

Another reason is that if we keep state in individual hosts of a web or backend service, we will need to implement sticky sessions, consistently routing the same user to the same host. We will also need to replicate the data in case a host fails, and handle failover (such as routing the users to the appropriate new host when their host fails). By pushing all state to a stateful storage service, we can choose the appropriate storage/database technology for our requirements, and take advantage of not having to design, implement, and make mistakes with managing state.

Storage can be broadly classified into the following. We should know how to distinguish between these categories. The following are brief notes required to follow the discussions in this book. A complete introduction to various storage types is outside the scope of this book. Refer to other materials if required.

- Database
    - SQL - Has relational characteristics such as tables and relationships between tables, including primary keys and foreign keys. SQL must have ACID properties.
    - NoSQL - A database that does not have all SQL properties.
        - Column-oriented - Organizes data into columns instead of rows for efficient filtering. Examples are Cassandra and HBase.
        - Key-value - Data is stored as a collection of key-value pairs. Each key corresponds to a disk location via a hashing algorithm. Read performance is good. Keys must be hashable, so they are primitive types, and cannot be pointers to objects. Values don't have this limitation; they can be primitives or pointers. Key-value databases are usually used for caching, employing various techniques like Least Recently Used (LRU). Cache has high performance, but does not require high availability (because if the cache is unavailable, the requestr can query the original data source). Examples are Memcached and Redis.
- Document - Can be interpreted as a key-value databases where values have no size limits or much larger limits than key-value databases. Values can be in various formats. Text, JSON or YAML are common. An example is MongoDB.
- Graph - Designed to efficiently store relationships between entities. Examples are
- File storage - Data stored in files, which can be organized into directories/folders. We can see it as a form of key-value, with path as the key.
- Block storage - Stores data in evenly-sized chunks with unique identifiers. We are unlikely to use block storage in web applications. Block storage is relevant for designing low-level components of other storage systems (such as databases).
- Object storage - Flatter hierarchy than file storage. Objects are usually accessed with simple HTTP APIs. Writing objects is slow, and objects cannot be modified, so object storage is suited for static data. AWS S3 is a cloud example.

3.2 When to use vs avoid databases

When deciding on how to store a service's data, you may discuss using a database vs other possibility such as file, block and object storage. During the interview, remember that even

though you may prefer certain approaches and you can state a preference during an interview, you must be able to discuss all relevant factors, and consider others' opinions. In this section, we discuss various factors that you may bring up. As always, discuss various approaches and tradeoffs.

A commonly-cited conclusion from a 2006 Microsoft paper (https://www.microsoft.com/en-us/research/publication/to-blob-or-not-to-blob-large-object-storage-in-a-database-or-a-filesystem) states "objects smaller than 256K are best stored in a database while objects larger than 1M are best stored in the filesystem. Between 256K and 1M, the read:write ratio and rate of object overwrite or replacement are important factors."

A few other points:

- SQL Server requires special configuration settings to store files larger than 2 GB.
- Database objects are loaded entirely into memory, so it is inefficient to stream a file from a database.
- Replication will be slow if database table rows are large objects, as these large blob objects will need to be replicated from the leader node to follower nodes.

3.3 Replication

We scale a database i.e. implement a distributed database onto multiple hosts (commonly called nodes in database terminology) via replication, partitioning and sharding. Replication is making copies of data, called replicas, and storing them on different nodes. Partitioning and sharing are both about dividing a data set into subsets. Sharding implies the subsets are distributed across multiple nodes while partitioning does not.

A single host has limitations, so it cannot fulfill our requirements:

- Fault-tolerance. Each node can back up its data onto other nodes within and across data centers in case of node or network failure. We can define a failover process for other nodes to take over the roles and partitions/shards of failed nodes.
- Higher storage capacity. A single node can be vertically scaled to contain multiple hard drives of the largest available capacity, but this is monetarily expensive, and along the way the node's throughput may become an issue.
- Higher throughput. The database needs to process reads and writes for multiple simultaneous processes and users. Vertical scaling approaches its limits with the fastest network card and better CPU and more memory.
- Lower latency. We can geographically distribute replicas to be closer to dispersed users. We can increase the number of particular replicas on a data center if there are more reads on that data from that locality.

To scale reads (SELECT operation), we simply increase the number of replicas of that data. Scaling writes is more difficult, and much of this chapter is about handling the difficulties of scaling write operations.

3.3.1 Distributing replicas

A typical design is to have 1 backup onto a host on the same rack and 1 backup on a host on a different rack or data center or both.

The data may also be sharded, which provides the following benefits. The main tradeoff of sharding is increased complexity from needing to track the shards' locations.

- Scale storage. If a database/table is too big to fit into a single node, sharding across nodes allows the database/table to remain a single logical unit.
- Scale memory. If a database is stored in memory, it may need to be sharded, since vertical scaling of memory on a single node quickly becomes monetarily expensive.
- Scale processing. A sharded database may take advantage of parallel processing.
- Locality. A database may be sharded such that the data a particular cluster node needs is likely to be stored locally rather than on another shard on another node.

For linearizability, certain partitioned databases like HDFS implement deletion as an append operation (called a logical soft delete). In HDFS, this is called appending a tombstone. This prevents disruptions and inconsistency to read operations that are still running while deletion occurs.

3.3.2 Single-leader replication

In single-leader replication, all write operations occur on a single node, called the leader. Single-leader replication is about scaling reads, not writes. Some SQL distributions such as MySQL and Postgres have configurations for single-leader replication. The SQL service loses its ACID consistency. This is a relevant consideration if we choose to horizontally scale a SQL database to serve a service with high traffic.

Figure 3.1 illustrates single-leader replication with primary-secondary leader failover. All writes occur on the primary leader node, and are replicated to its followers, including the secondary leader. If the primary leader fails, the failover process promotes the secondary leader to primary. When the failed leader is restored, it becomes the secondary leader.

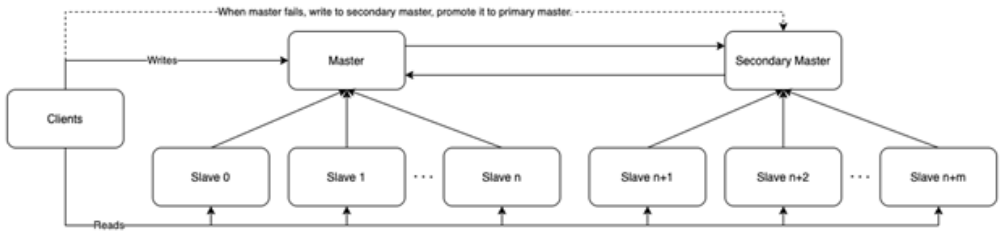

Figure 3.1 Single-leader replication with primary-secondary leader failover. Figure adapted from Ejsmont, Artur. (2015) Web Scalability for Startup Engineers, figure 5.4, McGraw Hill.

A single node has a maximum throughput which must be shared by its followers, imposing a maximum number of followers which in turn limits read scalability. To scale reads further, we can use multi-level replication, shown on figure 3.2.

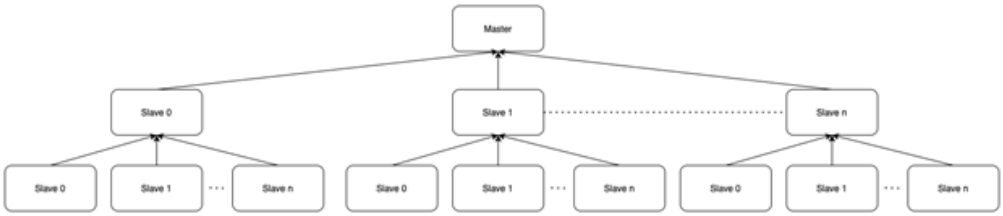

Figure 3.2 Multi-level replication. Each node replicates to its followers, which in turn replicates to their followers. This architecture ensures a node replicates to a number of followers that it is capable of handling, with the trade-off that consistency is further delayed. Figure adapted from Ejsmont, Artur. (2015) Web Scalability for Startup Engineers, figure 5.8, McGraw Hill.

Single-leader replication is the simplest to implement. The main limitation of single-leader replication is that the entire database must fit into a single host. Another limitation is eventual consistency, as write replication to followers takes time.

MySQL binlog-based replication is an example of single-leader replication. Refer to chapter 5 of the book Web Scalability for Startup Engineers (2015) by Artur Ejsmont for a good discussion. A Google search for "binlog replication" will also return many relevant results. Here are some relevant online documents:

- https://dev.to/tutelaris/introduction-to-mysql-replication-97c
- https://dev.mysql.com/doc/refman/8.0/en/binlog-replication-configuration-overview.html
- https://www.digitalocean.com/community/tutorials/how-to-set-up-replication-in-mysql
- https://docs.microsoft.com/en-us/azure/mysql/single-server/how-to-data-in-replication
- https://www.percona.com/blog/2013/01/09/how-does-mysql-replication-really-work/
- https://hevodata.com/learn/mysql-binlog-based-replication/

A HACK TO SCALING SINGLE-LEADER REPLICATION - QUERY LOGIC IN APPLICATION LAYER

Manually-entered strings increase database size slowly, which you can verify with simple estimates and calculations. If data was programmatically generated or has accumulated for a long period of time, storage size may grow beyond a single node.

If we cannot reduce the database size, but we wish to continue using SQL, a possible way is to divide the data between multiple SQL databases. This means that our service has to be configured to connect to more than 1 SQL database, and need to rewrite our SQL queries in the application to query from the appropriate database.

If we had to split a single table into 2 or more databases, then our application will need to query multiple databases and combine the results. Querying logic is no longer encapsulated in the database, and has spilled into the application. The application has to store metadata to track which databases contain particular data. This is essentially multi-leader replication with metadata management in the application. The services and databases are more difficult to maintain, particularly if there are multiple services using these databases.

We may suggest this during a discussion as a possibility, but it is highly unlikely that we will use this design. We should use databases with multi-leader or leaderless replication.

3.3.3 Multi-leader replication

Multi-leader and leaderless replication are techniques to scale writes and database storage size. They require handling of race conditions, which are not present in single-leader replication.

In multi-leader replication, as the name suggests, there are multiple nodes designated as leaders, and writes can be made on any leader. Each leader must replicate its writes to all other nodes.

CONSISTENCY ISSUES AND APPROACHES

This replication introduces consistency race conditions for operations where sequence is important. For example, if a row is updated in 1 leader while it is being deleted in another, what should be the final outcome? Using timestamps to order operations does not work because the clocks on different nodes cannot be perfectly synchronized. Attempting to use the same clock on different nodes doesn't work because each node will receive the clock's signals at different times, a well-known phenomenon called clock skew. So even server clocks that are periodically synchronized with the same source will differ by a few milliseconds or greater. If queries made to different servers within time intervals smaller than this difference, it is impossible to determine the order they were made.

Here we discuss replication issues and scenarios related to consistency that we commonly encounter in a system design interview. These situations may occur with any storage format, including databases and file systems. Designing Data-Intensive Applications and its references have more thorough treatments of replication pitfalls.

What is the definition of database consistency? Consistency ensures a database transaction brings the database from one valid state to another, maintaining database invariants; any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, or any combination thereof.

As discussed elsewhere in this book, consistency has a complex definition. A common informal understanding of consistency is that the data must be the same to every user.

1. The same query on multiple replicas should return the same results, even though the replicas are on different physical servers.
2. Data Manipulation Language (DML) queries i.e. INSERT, UPDATE, DELETE on different physical servers that affect the same rows should be executed in the sequence that they were sent.

We may accept eventual consistency, but any particular user may need to receive data that is a valid state to her. For example, if user A queries for a counter's value, increments a counter by 1, then queries again for that counter's value, it will make sense to user A to receive a value incremented by 1. Meanwhile, other users who query for the counter may be provided the value before it was incremented. This is called read-after-write consistency.

In general, look for ways to relax the consistency requirements. Find approaches that minimize the amount of data that must be kept consistent to all users.

DML queries on different physical servers that affect the same rows may cause race conditions. Some possible situations:

- DELETE and INSERT the same row on a table with a primary key.
    - If the DELETE executed first, the row should exist. If the INSERT was first, the primary key prevents execution, and DELETE should delete the row.
- 2 UPDATE operations on the same cell with different values. Only 1 should be the eventual state.

What about DML queries sent at the same millisecond to different servers? This is an exceedingly unlikely situation, and there seems to be no common convention for resolving race conditions from such situations. We can suggest various approaches. 1 approach is to prioritize DELETE over INSERT/UPDATE, and randomly break the ties for other INSERT/UPDATE queries. Anyway, a competent interviewer will not waste seconds of the 50-min interview on discussions that yield no signals like this one.

3.3.4 Leaderless replication

In leaderless replication, all nodes are equal. Reads and writes can occur on any node. How are race conditions handled? One method is to introduce the concept of quorum. A quorum is the minimum number of nodes that must be in agreement for consensus. It is easy to reason that if our database has n nodes, and reads and writes both have quorums of n/2 + 1 nodes, consistency is guaranteed. If we desire consistency, we choose between fast writes and fast reads. If fast writes are required, set a low write quorum and high read quorum, and vice versa for fast reads. Otherwise, only eventual consistency is possible, and UPDATE and DELETE operations cannot be consistent.

Cassandra, Dynamo, Riak, and Voldemort are examples of databases which use leaderless replication. In Cassandra, UPDATE operations suffer from race conditions, while DELETE operations are implemented using tombstones instead of the rows actually being deleted. In HDFS, reads and replication are based on rack locality, and all replicas are equal. https://stackoverflow.com/questions/53300112/which-replication-algorithm-is-used-in-hadoop-hdfs?rq=1

WORKAROUND FOR CONSISTENT UPDATE

Let's illustrate with an example a schema design trick that can be used when consistent UPDATE is required on a distributed database. Consider a table that has 2 columns, an integer ID column and an integer counter column. If we had consistent updates, we can increment, decrement, and query the counter as follows:

```
UPDATE sample_table SET counter = counter + 1 WHERE sample_id = %(id);
UPDATE sample_table SET counter = counter - 1 WHERE sample_id = %(id);
SELECT counter FROM sample_table WHERE sample_id = %(id);
```

The workaround is as follows. We can create a table that has a string counter column instead of integer, and append/INSERT rows with values "UP"/ "DOWN".

```
INSERT INTO sample_table (%(id), 'UP');
INSERT INTO sample_table (%(id), 'DOWN');
```

```
SELECT (
  SELECT Count(*) FROM sample_table WHERE counter = 'UP'
) - (
  SELECT Count(*) FROM sample_table WHERE counter = 'DOWN'
);
```

3.3.5 HDFS replication

This is a brief refresher section on HDFS, Hadoop, and Hive. Detailed discussions are outside the scope of this book.

HDFS replication does not fit cleanly into any of these 3 approaches. A HDFS cluster has an active NameNode, a passive (backup) NameNode, and multiple DataNode nodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode. User data never flows through the NameNode. HDFS stores a table as 1 or more files in a directory. Each file is divided into blocks which are sharded across DataNode nodes. The default block size is 64MB; this value can be set by admins.

Hadoop is a framework that stores and processes distributed data using the MapReduce programming model. Hive is a data warehouse solution built on top of Hadoop. Hive has the concept of partitioning tables by 1 or more columns for efficient filter queries.

For example, we can create a partitioned Hive table as follows.

```
CREATE TABLE sample_table (user_id STRING, created_date DATE, country STRING) PARTITIONED
       BY (created_date, country);
```

Figure 3.3 illustrates the directory tree of this table. The table's directory has subdirectories for the date values, which in turn have subdirectories for the column values. Queries filtered by created_date and/or country will process only the relevant files, avoiding the waste of a full table scan.

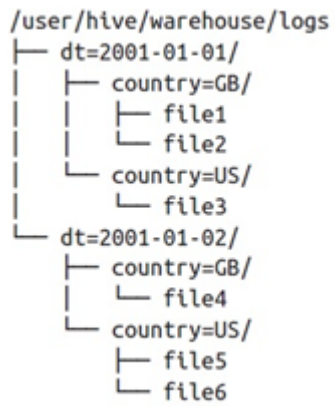

Figure 3.3 An example HDFS directory tree of a table "sample_table" whose columns include date and country, and the table is partitioned by these 2 columns. The sample_table directory has subdirectories for the

date values, which in turn have subdirectories for the column values. Image from https://stackoverflow.com/questions/44782173/hive-does-hive-support-partitioning-and-bucketing-while-usiing-external-tables.

HDFS is append-only, and does not support UPDATE or DELETE operations, possibly because of possible replication race conditions from UPDATE and DELETE. INSERT does not have race conditions.

HDFS has name quotas, space quotas, and storage type quotas. Regarding a directory tree:

- A name quota is a hard limit on the number of file and directory names.
- A space quota is a hard limit on the number of bytes in all files.
- A storage type quota is a hard limit on the usage of specific storage types. Discussion of HDFS storage types is outside the scope of this book.

Note: Novices to Hadoop and HDFS often use the Hadoop INSERT command, which should be avoided. An INSERT query creates a new file with a single row, which will occupy an entire 64MB block and is wasteful. It also contributes to the number of names, and programmatic INSERT queries will soon exceed the name quota. Refer to https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsQuotaAdminGuide.html for more information.

We should use `saveAsTable` or `saveAsTextFile` instead, such as the following example code snippet. An introduction to Spark is outside the scope of this book. Refer to the Spark documentation such as https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html.

```
val spark = SparkSession.builder().appName("Our app").config("some.config",
      "value").getOrCreate()
val df = spark.sparkContext.textFile({hdfs_file})
df.createOrReplaceTempView({table_name})
spark.sql({spark_sql_query_with_table_name}).saveAsTextFile({hdfs_directory})
```

3.3.6 Further reading

Refer to Designing Data-Intensive Applications by Martin Kleppmann for more discussion on topics such as:

- Consistency techniques like read repair, anti-entropy, and tuples.
- Multi-leader replication consensus algorithm and implementations in CouchDB, MySQL Group replication, and Postgres.
- Failover issues, like split brain.
- Various consensus algorithms to resolve these race conditions. A consensus algorithm is for achieving agreement on a data value.

3.4 Scaling storage capacity with sharded databases

If the database size grows to exceed the capacity of a single host, we will need to delete old rows. If we need to retain this old data, we should store it in sharded storage such as HDFS or Cassandra. Sharded storage is horizontally scalable and in theory should support an

infinite storage capacity simply by adding more hosts. There are production HDFS clusters with over 100 PB (https://eng.uber.com/uber-big-data-platform/). Cluster capacity of YB is theoretically possible, but the monetary cost of the hardware required to store and perform analytics on such amounts of data will be prohibitively expensive.

We can use a database with low latency such as Redis or SQL to store data used to directly serve consumers.

Another approach is to store the data in the consumer's devices or browser cookies and localStorage. However, this means that any processing of this data must also be done on frontend and not backend.

3.5 Aggregating events

Database writes are difficult and expensive to scale, so we should try to reduce the rate of database writes wherever possible in our system's design. Sampling and aggregation are common techniques to reduce database write rate. An added bonus is slower database size growth.

Besides reducing database writes, we can also reduce database reads by techniques such as caching and approximation. Chapter 21 discusses count-min sketch, an algorithm for creating an approximate frequency table for events in a continuous data stream.

Sampling is conceptually trivial and is something we can mention during an interview.

Aggregating events is about aggregating/combining multiple events into a single event, so instead of multiple database writes, only a single database write has to occur. We can consider aggregation if the exact timestamps of individual events are unimportant.

Aggregation can be implemented using a streaming pipeline. The first stage of the streaming pipeline may receive a high rate of events and require a large cluster with thousands of hosts. Without aggregation, every succeeding stage will also require a large cluster. Aggregation allows each succeeding stage to have fewer hosts. We also use replication and checkpointing in case hosts fail. Referring to chapter 4, we can use a distributed transaction algorithm such as Saga, or quorum writes, to ensure that each event is replicated to a minimum number of replicas.

3.5.1 Single-tier aggregation

Aggregation can be single or multi-tier. Figure 3.4 illustrates an example of single-tier aggregation, for counting numbers of values. In this example, an event can have value A, B, C, etc. These events can be evenly distributed across the hosts by a load balancer. Each host can contain a hash table in memory, and aggregate these counts in its hash table. Each host can flush the counts to the database periodically (such as every 5 minutes) or when it is running out of memory, whichever is sooner.

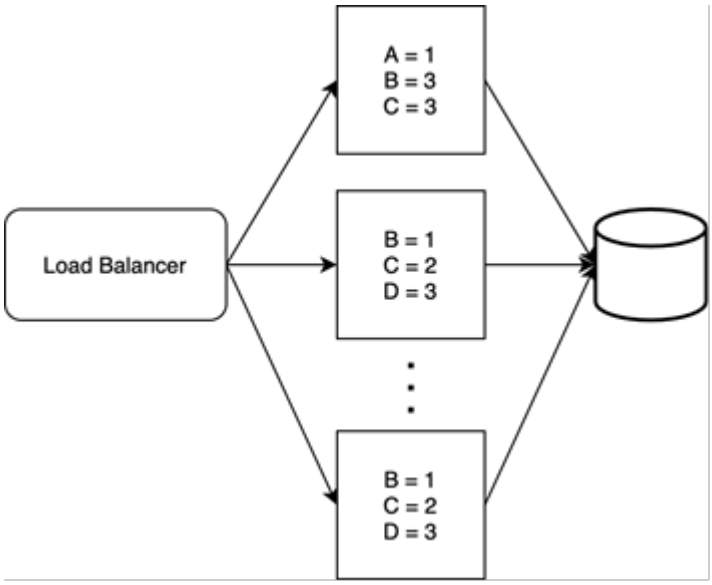

Figure 3.4 An example illustration of single-tier aggregation. A load balancer distributes events across a single layer/tier of hosts, which aggregate them, then write these aggregated counts to the database. If the individual events were written directly to the database, the write rate will be much higher, and the database will have to be scaled up. Not illustrated here are the host replicas, which are required if high availability and accuracy are necessary.

3.5.2 Multi-tier aggregation

Figure 3.5 illustrates multi-tier aggregation. Each layer of hosts can aggregate events from its ancestors in the previous tier. We can progressively reduce the number of hosts in each layer until there are a desired number of hosts (this number is up to our requirements and available resources) in the final layer, which writes to the database.

The main trade-offs of aggregation are eventual consistency and increased complexity. Each layer adds some latency to our pipeline and thus our database writes, so database reads may be stale. Implementing replication and logging, monitoring and alerting also add complexity to this system.

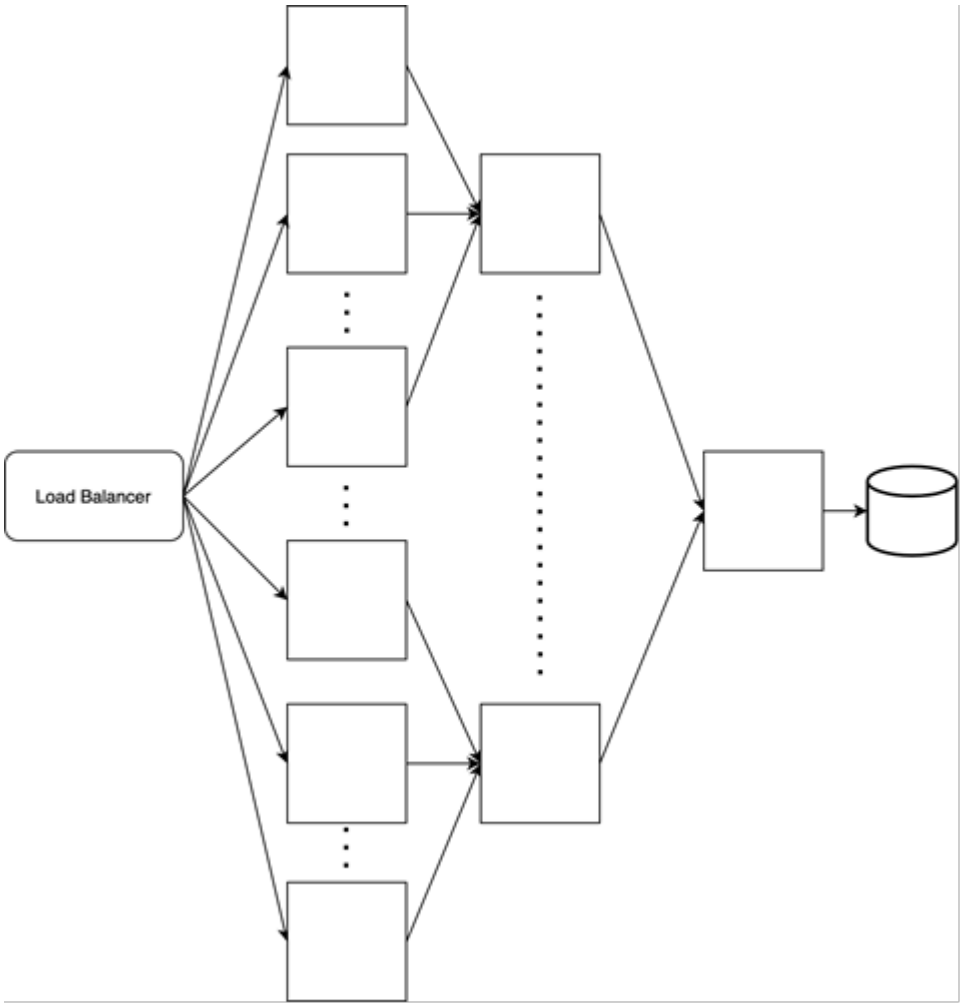

Figure 3.5 An example illustration of multi-tier aggregation.

3.5.3 Partitioning

This requires a level 7 load balancer. (Refer to section 2.1.3 for a brief description of a level 7 load balancer.) The load balancer can be configured to process incoming events and forward them to certain hosts depending on the events' contents.

Referring to the example on figure 3.6, if the events are simply values from A-Z, the load balancer can be configured to forward events with values of A-I to certain hosts, events with values J-R to certain hosts, and events with values S-Z to certain hosts. The hash tables from the first layer of hosts are aggregated into a second layer of hosts, then into a final

hash table host. Finally, this hash table is sent to a max-heap host which constructs the final max-heap.

We can expect event traffic to follow normal distribution, which means certain partitions will receive disproportionately high traffic. To address this, referring to figure 3.6, we observe that we can allocate a different number of hosts to each partition. Partition A-I has 3 hosts, J-R has 1 host, and S-Z has 2 hosts. We make these partitioning decisions as traffic is uneven across partitions, and certain hosts may receive disproportionately high traffic i.e. they become "hot", more than what they are able to process.

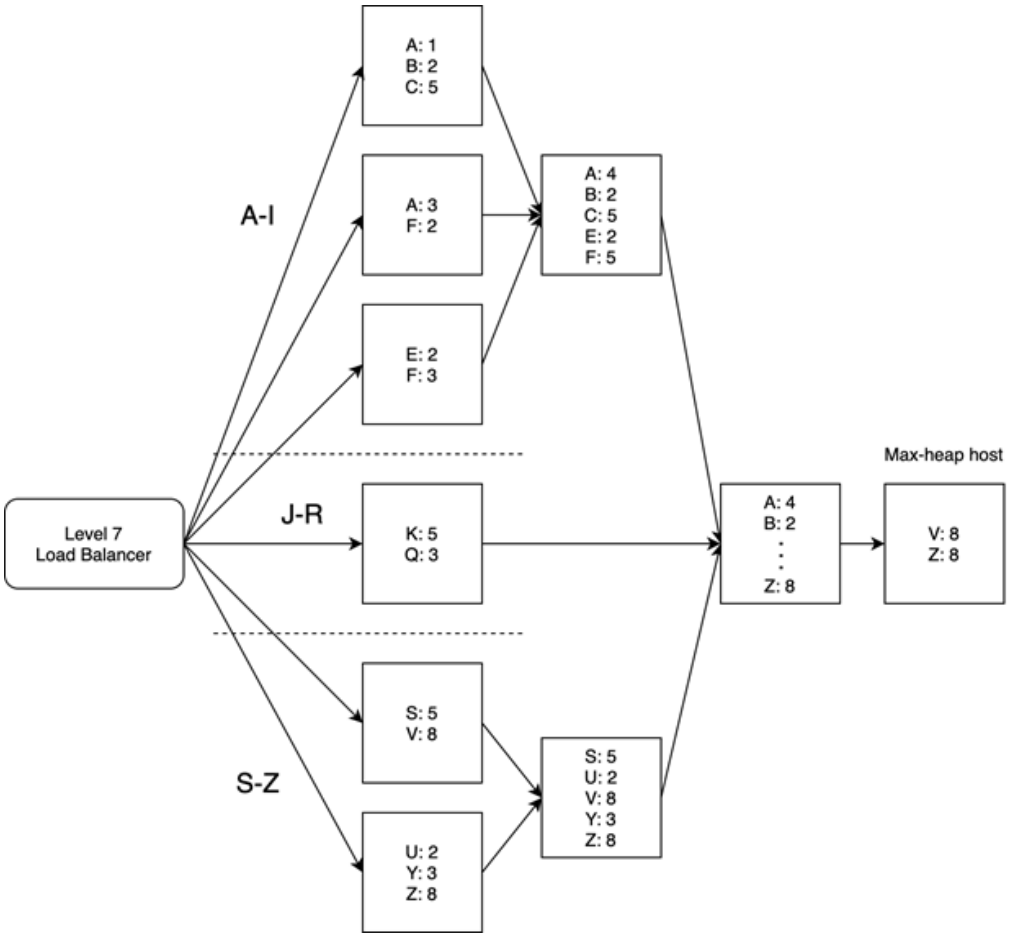

Figure 3.6 An example illustration of multi-tier aggregation with partitioning.

We also observe that partition J-R has only 1 host, so it does not have a 2nd layer. As designers, we can make such decisions based on our situation.

Besides allocating a different number of hosts to each partition, another way to evenly distribute traffic is to adjust the number and width of the partitions. For example, instead of {A-I, J-R, S-Z}, we can create partitions {{A-B, D-F}, {C, G-J}, {K-S}, {T-Z}} i.e. we changed from 3 to 4 partitions, and put C in the 2nd partition. We can be creative and dynamic in addressing our system's scalability requirements.

3.5.4 Handling a large key space

Figure 3.6 in the previous section illustrates a tiny key space of 26 keys from A-Z. In a practical implementation, the key space will be much larger. We must ensure that the combined key spaces of a particular level do not cause memory overflow in the next level. The hosts in the earlier aggregation levels should limit their key space to less than what their memory can accommodate, so that the hosts in the later aggregation levels have sufficient memory to accommodate all the keys. This may mean that the hosts in earlier aggregation levels will need to flush more frequently.

For example, figure 3.7 illustrates a simple aggregation service with only 2 levels. There are 2 hosts in the 1st level and 1 host in the 2nd level. The 2 hosts in the 1st level should limit their key space to ½ of what they can actually accommodate, so the host in the 2nd level can accommodate all keys.

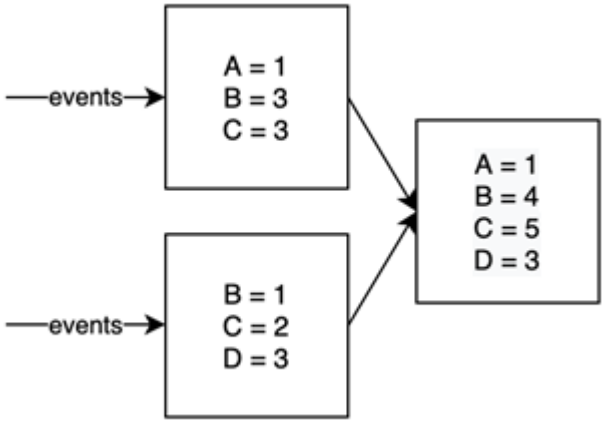

Figure 3.7 A simple aggregation service with only 2 levels, 2 hosts in the 1st level and 1 host in the 2nd level.

We can also provision hosts with less memory for earlier aggregation levels, and hosts with more memory in later levels.

3.5.5 Replication and fault-tolerance

So far, we have not discussed replication and fault tolerance. If a host goes down, it loses all of its aggregated events. Moreover, this is a cascading failure, as all its earlier hosts may overflow and these aggregated events will likewise be lost.

We can use checkpointing and dead letter queues as discussed in sections 2.3.5 and 2.3.6. However, since a large number of hosts may be affected by the outage of a host that is many levels deep, a large amount of processing has to be repeated, which is a waste of resources. This outage may also add considerable latency to the aggregation.

A possible solution is to convert each node into an independent service with a cluster of multiple stateless nodes that make requests to a shared in-memory database like Redis. Figure 3.8 illustrates such a service. The service can have multiple e.g., 3 stateless hosts. A shared load balancing service can spread requests across these hosts. Scalability is not a concern here, so each service can have just a few e.g., 3 hosts for fault-tolerance.

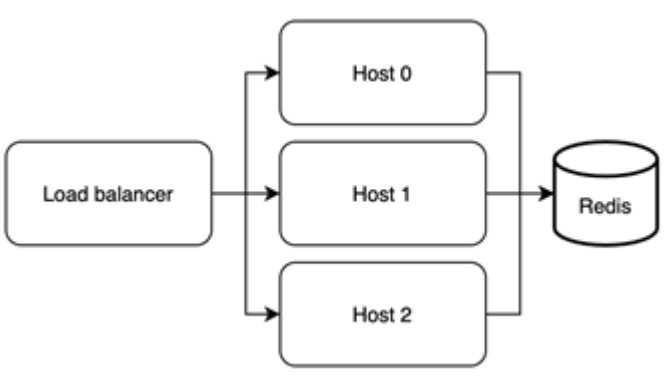

Figure 3.8 We can replace a node with a service, which we refer to as an aggregation unit. This unit has 3 stateless hosts for fault-tolerance, but we can use more hosts if desired.

At the beginning of this chapter, we discussed that we wanted to avoid database writes, which we seem to go back on here. However, each service has a separate Redis cluster, so there is no competition for writing to the same key. Moreover, these aggregated events are deleted each successful flush, so the database size will not grow uncontrollably.

We can use Terraform to define this entire aggregation service. Each aggregation unit can be a Kubernetes cluster with 3 pods, and 1 host per pod (2 hosts if we are using a sidecar service pattern).

3.6 Batch and streaming ETL

ETL (Extract, Transform, Load) is a general procedure of copying data from one or more sources into a destination system which represents the data differently from the source(s) or in a different context than the source(s). Batch refers to processing the data in batches, usually periodically but can also be manually-triggered. Streaming refers to a continuous flow of data to be processed in real-time.

We can think of batch vs streaming as analogous to polling vs interrupt. Similar to polling, a batch job always runs at a defined frequency regardless of whether there are new events to process, while a streaming job runs whenever a trigger condition is met, which is usually the publishing of a new event.

Airflow and Luigi are common batch tools. Kafka and Flink are common streaming tools. Flume and Scribe are specialized streaming tools for logging; they aggregate log data streamed in real-time from many servers. Here we briefly introduce some ETL concepts.

An ETL pipeline consists of a Directed Acyclic Graph (DAG) of tasks. In the DAG, a node corresponds to a task, and its ancestors are its dependencies. A job is a single run of an ETL pipeline.

3.6.1 A simple batch ETL pipeline

A simple batch ETL pipeline can be implemented using a crontab, 2 SQL tables, and a script (i.e. a program written in a scripting language) for each job. cron is suitable for small noncritical jobs with no parallelism where a single machine is adequate.

The following are the 2 example SQL tables.

```
CREATE TABLE cron_dag (
  id INT,          -- ID of a job.
  parent_id INT,  -- Parent job. A job can have 0, 1, or multiple parents.
  PRIMARY KEY (id),
  FOREIGN KEY (parent_id) REFERENCES cron_dag (id)
);
CREATE TABLE cron_jobs (
  id INT,
  name VARCHAR(255),
  updated_at INT,
  PRIMARY KEY (id)
);
```

The crontab's instructions can be a list of the scripts. In this example, we used Python scripts, though we can use any scripting language. We can place all the scripts in a common directory /cron_dag/dag/, and other Python files/modules in other directories. There are no rules on how to organize the files; this is up to what we believe is the best arrangement.

```
0 * * * * ~/cron_dag/dag/first_node.py
0 * * * * ~/cron_dag/dag/second_node.py
```

Each script can follow the following algorithm. Steps 1 and 2 can be abstracted into reusable modules.

1. Check that the updated_at value of the relevant job is less than its dependent jobs.
2. Trigger monitoring if necessary.
3. Execute the specific job.

The main disadvantages of this setup are:

- Not scalable. All jobs run on a single host, which carries all the usual disadvantages of a single host:
    - Single point of failure.
    - There may be insufficient computational resources to run all the jobs scheduled at a particular time.
    - The host's storage capacity may be exceeded.

- A job may consist of numerous smaller tasks, like sending a notification to millions of devices. If such a job fails and needs to be retried, we need to avoid repeating the smaller tasks that succeeded, i.e. the individual tasks should be idempotent. This simple design does not provide such idempotency.
- No validation tools to ensure the job IDs are consistent in the Python scripts and SQL tables, so this setup is vulnerable to programming errors.
- No GUI (unless we make one ourselves).
- We have not yet implemented logging, monitoring, or alerting. This is very important, and should be our next step.

Dedicated job scheduling systems include Airflow and Luigi. These tools come with web UIs for DAG visualization and GUI user-friendliness. They are also vertically scalable, and can be run on clusters to manage large numbers of jobs. In this book, whenever we need a batch ETL service, we use an organizational-level shared Airflow service.

3.6.2 Messaging terminology

The section clarifies common terminology for various types of messaging and streaming setups that one tends to encounter in technical discussions or literature.

MESSAGING SYSTEM

A Messaging System is a general term for a system that transfers data from one application to another to reduce the complexity of data transmission and sharing in applications, so application developers can focus on data processing. https://www.sciencedirect.com/topics/computer-science/messaging-system

MESSAGE QUEUE

A message contains a work object of instructions sent from one service to another, waiting in the queue to be processed. Each message is processed only once by a single consumer.

PRODUCER/CONSUMER

Producer/consumer aka publisher/subscriber or pub/sub is an asynchronous messaging system that decouples services that produce events from services that process events. A producer/consumer system contains one or more message queues.

MESSAGE BROKER

A Message Broker is a program that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. A Message Broker is a translation layer. Kafka and RabbitMQ are both message brokers. RabbitMQ claims to be "the most widely deployed open source message broker". AMQP is one of the messaging protocols implemented by RabbitMQ. A description of AMQP is outside the scope of this book. Kafka implements its own custom messaging protocol.

EVENT STREAMING

Event streaming is a general term that refers to a continuous flow of events, that are processed in real-time. An event contains information about a change of state. Kafka is the most common Event Streaming platform.

PULL VS PUSH

Inter-service communication can be done by pull or push. In general, pull is better than push, and this is the general concept behind producer-consumer architectures. In pull, the consumer controls the rate of message consumption, and will not be overloaded.

Load testing and stress testing may be done on the consumer during its development, and monitoring its throughput and performance with production traffic and comparing the measurements with the tests allows the team to accurately determine if more engineering resources are needed to improve the tests. The consumer can monitor its throughput and producer queue sizes over time, and the team can scale it as required.

If our production system has continuous high load, it is unlikely that the queue will be empty for any significant period, and our consumer can keep polling for messages. If we have a situation where we must maintain a large streaming cluster to process unpredictable traffic spikes within a few minutes, we should use this cluster to for other lower-priority messages too i.e. a common Flink, Kafka, or Spark service for the organization.

Another situation where polling or pull from a user is better than push is if the user is firewalled, or if the dependency has frequent changes and will make too many push requests. Pull also will have 1 less setup step than push. The user is already making requests to the dependency. However, the dependency usually does not make requests to the user.

The flip side (https://engineering.linkedin.com/blog/2019/data-hub) is if our system collects data from many sources using crawlers, development and maintenance of all these crawlers may be too complex and tedious. It may be more scalable for individual data providers to push information to our central repository. Push also allows more timely updates.

One more exception where push is better than pull is in lossy applications like audio and video live streaming. These applications do not resend data which failed to deliver the first time, and generally use UDP to push data to their recipients.

3.6.3 Kafka vs RabbitMQ

In practice, most companies have a shared Kafka service, that is used by other services. In the rest of this book, we will use Kafka when we need a messaging or event streaming service. In an interview, rather than risk the ire of an opinionated interviewer, it is safer to display our knowledge of the details of and differences between Kafka and RabbitMQ and discuss their tradeoffs.

Both can be used to smooth out uneven traffic, preventing our service from being overloaded by traffic spikes, and keep our service cost-efficient as we do not need to provision a large number of hosts just to handle periods of high traffic.

Kafka is more complex than RabbitMQ, and provides a superset of capabilities over RabbitMQ i.e. Kafka can always be used in place of RabbitMQ but not vice versa.

If RabbitMQ is sufficient for our system, we can suggest to use RabbitMQ, and also state that our organization likely has a Kafka service that we can use so as to avoid the trouble of setup and maintenance (including logging, monitoring and alerting) of another component i.e. RabbitMQ.

Table 3.1 has some differences between Kafka and RabbitMQ.

Table 3.1 Some differences between Kafka and RabbitMQ.

Kafka	RabbitMQ
Designed for scalability, reliability, and availability. More complex setup required than RabbitMQ. Require ZooKeeper to manage the Kafka cluster. This includes configuring IP addresses of every Kafka host in Zookeeper.	Simple to set up, but not scalable by default. We can implement scalability on our own at the application level, by attaching our application to a load balancer, and producing to and consuming from the load balancer. But this will take more work to set up than than Kafka and being far less mature will almost certainly be inferior in many ways.
A durable Message Broker, because it has replication. We can adjust the replication factor on Zookeeper, and arrange replication be to done on different server racks and data centers.	Not scalable, so not durable by default. Messages are lost if downtime occurs. Has a "lazy queue" feature to persist messages to disk for better durability, but this does not protect against disk failure on the host.
Events on the queue are not removed after consumption, so the same event can be consumed repeatedly. This is for failure tolerance, in case the consumer fails before it finished processing the event, and needs to reprocess the event. In this regard, it is conceptually inaccurate to use the term "queue" in Kafka. It is actually a list. But this term "Kafka queue" is commonly used. We can configure a retention period in Kafka, which is 7 days by default, so an event is deleted after 7 days regardless of whether it has been consumed. We can choose to set the retention period to infinite and use Kafka as a database.	Messages on the queue are removed upon dequeuing, as per the definition of "queue". We may create several queues to allow several consumers per message, one queue per consumer. But this this is not the intended use of having multiple queues.
No concept of priority.	Has the concept of AMQP standard per-message queue priority. We can create multiple queues with varying priority. Messages on a queue are not dequeued until higher-priority queues are empty. No concept of fairness or consideration of starvation.

3.6.4 Lambda architecture

Lambda architecture is a data-processing architecture for processing big data running batch and streaming pipelines in parallel. In informal terms, it refers to having parallel fast and

slow pipelines that update the same destination. The fast pipeline trades off consistency and accuracy for lower latency i.e. fast updates, and vice versa for the slow pipeline.

The fast pipeline employs techniques such as:

- Approximation algorithms (discussed in appendix A).
- In-memory databases like Redis.
- May avoid replication for faster processing, so there may be some data loss and lower accuracy from node outages.

The slow pipeline usually uses MapReduce databases, such as Hive and Spark with HDFS.

We can suggest lambda architecture for systems that involve big data and require consistency and accuracy.

Note on various database solutions

There are numerous database solutions. Common ones include various SQL distributions, Hadoop and HDFS, Kafka, Redis, and Elasticsearch. There are numerous less common ones, including MongoDB, neo4j, AWS DynamoDB and Google's Firebase Realtime Database. In general, knowledge of less common databases, especially proprietary databases, is not expected in a Systems Design interview. Proprietary databases are seldom adopted. If a startup does adopt a proprietary database, it should consider migrating to an open-source database sooner rather than later. The bigger the database, the worse the vendor lock-in as the migration process will be more difficult, error-prone, and expensive.

3.7 Denormalization

If our service's data can fit into a single host, a typical approach is to choose SQL and normalize our schema. The benefits of normalization include the following:

- Consistency. No duplicate data, so there will not be tables with inconsistent data.
- Inserts and updates are faster since only 1 table has to be queried. In a denormalized schema, an insert or update may need to query multiple tables.
- Smaller database size as there is no duplicate data. Smaller tables will have faster read operations.
- Normalized tables tend to have fewer columns, so they will have fewer indexes. Index rebuilds will be faster.
- Queries can JOIN only the tables that are needed.

The disadvantages of normalization include the following:

- JOIN queries are much slower than queries on individual tables. In practice, denormalization is frequently done because of this.
- The fact tables contain codes rather than data, so most queries both for our service and ad hoc analytics will contain JOIN operations. JOIN queries tend to be more verbose than queries on single tables, so they are more difficult to write and maintain.

An approach to faster read operations that is frequently mentioned in interviews is to trade off storage for speed by denormalizing our schema to avoid JOIN queries.

3.8 Caching

For databases that store data on disk, we can cache frequent or recent queries in memory. In an organization, various database technologies can be provided to users as shared database services, such as a SQL service or Spark with HDFS. Such services can also utilize caching, such as with a Redis cache.

3.9 Further reading

This chapter uses material from Ejsmont, Artur. (2015) Web Scalability for Startup Engineers, McGraw Hill.

3.10 Summary

- Designing a stateful service is much more complex and error-prone than a stateless service, so system designs try to keep services stateless, and use shared stateful services.
- Each storage technology falls into a particular category. We should know how to distinguish these categories, which are as follows.
    - Database, which can be SQL or NoSQL. NoSQL can be categorized into column-oriented or key-value.
    - Document.
    - Graph.
    - File storage.
    - Block storage.
    - Object storage.
- Deciding on how to store a service's data involves deciding to use a database vs another storage category.
- There are various replication techniques to scale databases, including single-leader replication, multi-leader replication, leaderless replication, and other techniques such as HDFS replication that do not fit cleanly into these 3 approaches.
- Sharding is needed if a database exceeds the storage capacity of a single host.
- Database writes are expensive and difficult to scale, so we should minimize database writes wherever possible. Aggregating events helps to reduce the rate of database writes.
- Lambda architecture involves using parallel batch and streaming pipelines to process the same data, so as to realize the benefits of both approaches while allowing them to compensate for each other's disadvantages.
- Denormalizing is frequently used to optimize read latency and simpler SELECT queries, with tradeoffs like consistency, slower writes, more storage required, and slower index rebuilds.
- Caching frequent queries in memory reduces average query latency.

4

Distributed transactions

This chapter covers:

- **Using distributed transactions for data consistency across multiple services.**
- **Using event sourcing for scalability, availability, and lower cost and consistency.**
- **Using Change Data Capture (CDC) to write a change to multiple services.**
- **Carrying out a business transaction that involves multiple services, using choreography vs orchestration saga.**

A transaction is a way to group several reads and writes into a logical unit. They execute atomically, as a single operation, and either the entire transaction succeeds (commit) or fails (abort, rollback). A transaction has ACID properties, though the understanding of ACID concepts differs between databases, so the implementations also differ.

In a system design, we may need to write the same data to multiple services. Each write to each service is a separate request/event. If we can use an event streaming platform like Kafka to distribute these writes, allowing downstream services to pull instead of pushing these writes, we should do so. (Refer to page XXX for a discussion of pull vs push.) For other situations, we introduce the concept of a distributed transaction, which combines these separate write requests as a single distributed (atomic) transaction. We introduce the concept of consensus i.e. all the services agree that the write event has occurred (or not occurred). For consistency across the services, consensus should occur despite possible faults during write events.

This section describes algorithms for maintaining consistency in distributed transactions.

- Event sourcing.
- Change data capture, including the transaction log tailing pattern.
- Checkpointing and dead-letter queue.
- Saga.
- Two-phase commit. (Outside the scope of this book.)

Two-phase commit and saga achieve consensus (all commit or all abort), while the other techniques are designed to designate a particular database as a source of truth should inconsistency result from failed writes.

4.1 Event sourcing

Event sourcing is a pattern for storing data or changes to data as events in an append-only log. Referring to figure 4.1, event sourcing consists of publishing and persisting state-changing events of an entity as a sequence of events. These events are stored in a log, and subscribers process the log's event to determine the entity's current state. So, the publisher service asynchronously communicates with the subscriber service via the event log.

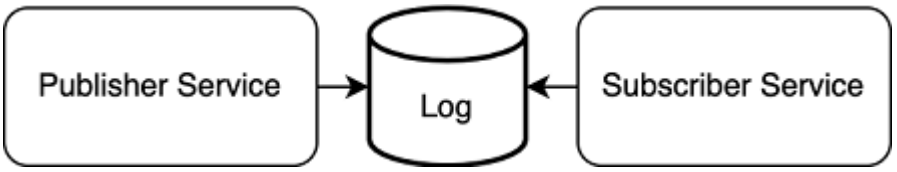

Figure 4.1 In event sourcing, a publisher service publishes a sequence of events to a log that indicate changes to the state of an entity. A subscriber service processes the log events in sequence to determine the entity's current state.

This can be implemented in various ways. The publisher can append to a file while a subscriber tails the file. Or the publisher can write to a message broker queue such as a Kafka topic, or a database table, and the subscriber consumes from the queue or queries the database table.

Event sourcing trades offs low latency and strong consistency for scalability, high availability, and lower cost and complexity. The event log services are often implemented as shared Kafka or database services. A company will devote considerable resources to ensure high availability and reliability of these shared services, including more engineers and support staff. These shared services also change much less often than other services. The latter may have code changes and deployments every day. Less frequent changes means that Kafka and database services have lower risk of outages from bugs.

There may be less resources and attention paid to individual services, because outages may have little or no visible impact to the ability of a company to provide its services. There may be insufficient engineering and support resources to ensure high availability and reliability.

As long as the event log is highly available, the entire system is highly available. The system continues to be available even if services that consume from the event log are down. The overall system is eventually-consistent.

What if a subscriber host crashes while processing an event? How will the subscriber service know that it must process that event again? We can use checkpointing, as discussed in section Checkpointing on page XXX.

The alternative to event sourcing is that a service makes a request directly to another service. Regardless if such a request was blocking or non-blocking, unavailability or slow

performance of either service mean that the overall system is unavailable. This request also consumes a thread in each service, so there is one less thread available during the time the request takes to process, and this effect is especially noticeable if the request takes a long time to process or during traffic spikes. A traffic spike can overwhelm the service and cause 504 timeouts. To prevent this, we will need to use complex auto-scaling solutions, or maintain a large cluster of hosts which incurs more expense. However, this approach ensures strong consistency and lower latency. Each request is processed immediately, unlike in event sourcing where a request can stay in the log for some time before a subscriber processes it.

A less resource-intensive approach is to publish an event onto an event log. The publisher service does not need to continuously consume a thread to wait for the subscriber service to finish processing an event.

These alternatives are more expensive, complex, error-prone, and less scalable. The strong consistency and low latency that they provide may not actually be needed by users.

Event sourcing is a common concept of the other techniques discussed in this section, namely Change Data Capture (CDC) and Saga. Its purpose, advantages, and disadvantages carry over to those techniques.

4.2 Transaction supervisor

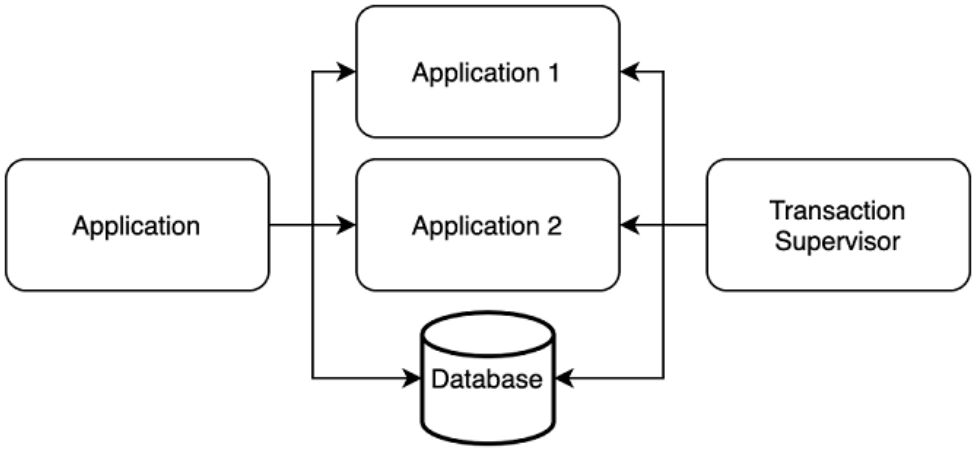

Figure 4.2 Example illustration of a transaction supervisor. An application may write to multiple downstream applications and databases. A transaction supervisor periodically syncs the various destinations in case any writes fail.

A transaction supervisor is a process that ensures a transaction is successfully completed or is compensated. It can be implemented as a periodic batch job or serverless function. Figure 4.2 shows an example of a transaction supervisor.

A transaction supervisor should generally be first implemented as an interface for manual review of inconsistencies, and manual executions of compensating transactions. Automating

compensating transactions is generally risky and should be approached with caution. Before automating a compensating transaction, it must first be extensively tested. Also ensure that there are no other distributed transaction mechanisms, or they may interfere with each other, leading to data loss or situations which are difficult to debug.

A compensating transaction must always be logged, regardless of whether it was manually or automatically run.

4.3 Change Data Capture (CDC)

Change Data Capture (CDC) is about logging data change events to a change log event stream and providing this event stream through an API.

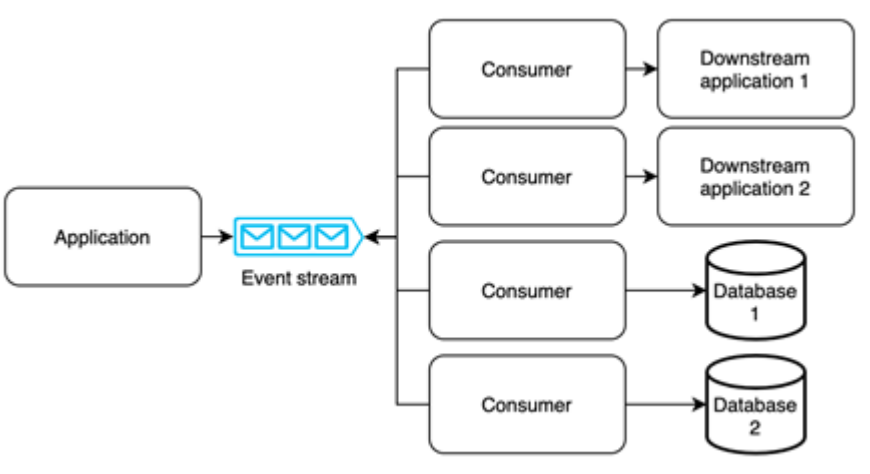

Figure 4.3 Using a change log event stream to synchronize data changes. Besides consumers, serverless functions can also be used to propagate changes to downstream applications or databases.

Figure 4.3 illustrates CDC. A single change or group of changes can be published as a single event to a change log event stream. This event stream has multiple consumers, each corresponding to a service/application/database. Each consumer consumes the event and provides it to its downstream service to be processed. Without CDC, the publisher will have to make a separate request to each downstream service, which carries the disadvantages discussed in the previous section on event sourcing.

The transaction log tailing pattern[1] is another system design pattern to prevent possible inconsistency when a process needs to write to a database and produce to Kafka. One of the two writes may fail, causing inconsistency.

[1] Chris Richardson. Microservices Patterns: With Examples in Java. Manning Publications, 2019. Page 99-100.

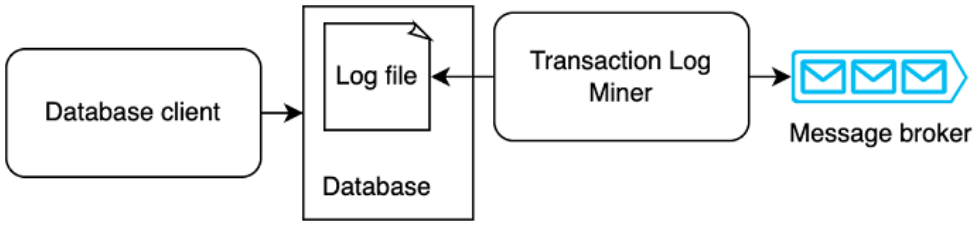

Figure 4.4 Illustration of the transaction log tailing pattern. A service does a write query to a database, which records this query in its log file. The transaction log miner tails the log file and picks up this query, then produces an event to the message broker.

Figure 4.4 illustrates the transaction log tailing pattern. In transaction log tailing, a process called the transaction log miner tails a database's transaction log and produces each update as an event. Transaction log tailing is a form of Change Data Capture (CDC), discussed in the previous section.

CDC platforms include Debezium (https://debezium.io/), Databus (https://github.com/linkedin/databus), DynamoDB Streams (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html), and Eventuate CDC Service (https://github.com/eventuate-foundation/eventuate-cdc). They can be used as transaction log miners.

Transaction log miners may generate duplicate events. One way to handle duplicate events is to use the message broker's mechanisms for exactly-once delivery. Another way is for the events to be defined and processed idempotently.

Davis (Cornelia Davis. Cloud Native Patterns, chapter 12.4, page 349-356. Manning Publications, 2019) refers to CDC as "event sourcing". The idea of event sourcing is that the event log is the source of truth, and all other databases are projections of the event log. Any write must first be made to the event log. After this write succeeds, one or more event handlers consume this new event and writes it to the other databases.

4.4 Saga

A saga is a long-lived transaction that can be written as a sequence of transactions. All transactions must complete successfully, or compensating transactions are run to roll back the executed transactions. A saga is a pattern to help manage failures. A saga itself has no state.

A typical saga implementation involves services communicating via a message broker like Kafka or RabbitMQ. In our discussions in this book that involve saga, we will use Kafka.

An important use case of sagas is to carry out a distributed transaction only if certain services satisfy certain requirements. For example, in booking a tour package, a travel service may make a write request to an airline ticket service and another write request to a hotel room service. If there are either no available flights or hotel rooms, the entire saga should be rolled back.

The airline ticket service and hotel room service may also need to write to a payments service, which is separate from the airline ticket service and hotel service for possible reasons including the following:

- The payment service should not process any payments until the airline ticket service confirms that the ticket is available and hotel room service confirms that the room is available. Otherwise, it may collect money from the user before confirming her entire tour package.
- The airline ticket and hotel room services may belong to other companies, and we cannot pass the user's private payment information to them. Rather, our company needs to handle the user's payment, and our company should make payments to other companies.

If a transaction to the payments service fails, the entire saga should be rolled back in reverse order using compensating transactions on the other two services.

There are two ways to structure the coordination: choreography (parallel) or orchestration (linear). In the rest of this section, we discuss one example of choreography and one example of orchestration, then compare choreography vs. Orchestration. Refer to https://microservices.io/patterns/data/saga.html for another example.

4.4.1 Choreography

In choreography, the service that begins the saga communicates with 2 Kafka topics. It produces to 1 Kafka topic to start the distributed transaction, and consumes from another Kafka topic to perform any final logic. Other services in the saga communicate directly with each other via Kafka topics.

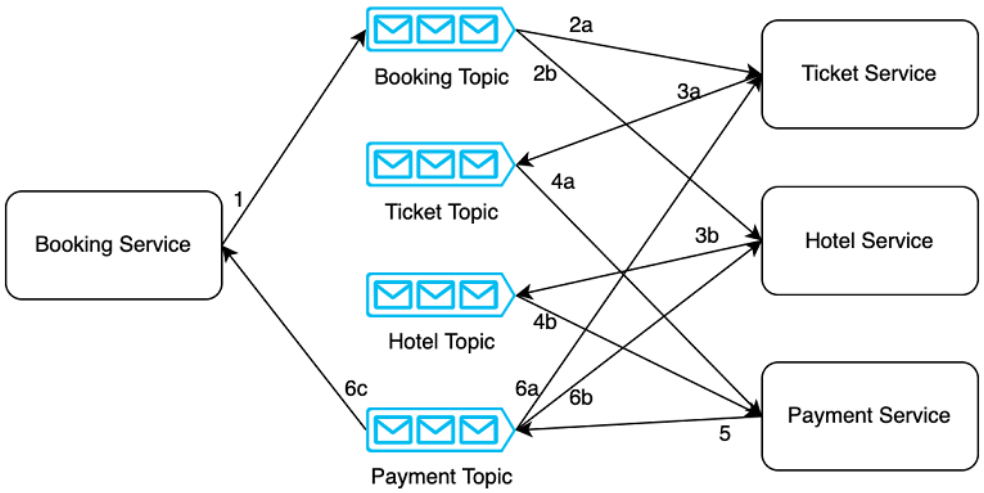

Figure 4.5 A choreography saga to book an airline ticket and a hotel room for a tour package. Two labels with

the same number but different letters represent steps that occur in parallel.

Figure 4.5 illustrates a choreography saga to book a tour package. In this chapter, the figures that include Kafka topics illustrate event consumption with the line arrowheads pointing away from the topic. In the other chapters of this book, an event consumption is illustrated with the line arrowhead pointing to the topic. The reason for this difference is that the diagrams in this chapter may be confusing if we follow the same convention as the other chapters. The diagrams in this chapter illustrate multiple services consuming from multiple certain topics and producing to multiple other topics, and it is clearer to display the arrowhead directions in the manner that we chose.

The steps of a successful booking are as follows.

1. A user may make a booking request to the Booking Service. The Booking Service produces a booking request event to the Booking Topic.
2. The Ticket Service and Hotel Service consume this booking request event. They both confirm that their requests can be fulfilled. Both services may record this event in their respective databases, with the booking ID and a state like "AWAITING_PAYMENT".
3. The Ticket Service and Hotel Service each produce a payment request event to the Ticket Topic and Hotel Topic respectively.
4. The Payment Service consumes these payment request events from the Ticket Topic and Hotel Topic topics. As these two events are consumed at different times and likely by different hosts, the Payment Service needs to record the receipt of these events in a database, so the service's hosts will know when all the required events have been received. When all required events are received, the Payment Service will process the payment.
5. If the payment is successful, the Payment Service produces a payment success event to the Payment Topic.
6. The Ticket Service, Hotel Service, and Booking Service consume this event. The Ticket Service and Hotel Service both confirm this booking, which may involve changing the state of that booking ID to CONFIRMED, or other processing and business logic as necessary. The Booking Service may inform the user that her booking is confirmed.

Step 1-4 are compensable transactions, which can be rolled back by compensating transactions. Step 5 is a pivot transaction. Transactions after the pivot transaction can be retried until they succeed. The step 6 transactions are retriable transactions; this is an example of CDC as discussed in section 2.2. The Booking Service doesn't need to wait for any responses from the Ticket Service or Hotel Service.

A question that may be asked is how does an external company subscribe to our company's Kafka topics. The answer is that it doesn't. For security reasons, we never allow direct external access to our Kafka service. We had simplified the details of this discussion for clarity. The Ticket Service and Hotel Service actually belong to our company. They

communicate directly with our Kafka service/topics and make requests to external services. Figure 4.5 did not illustrate these details so they don't clutter the design diagram.

If the Payment Service responds with an error that the ticket cannot be reserved (maybe because the requested flight is fully-booked or cancelled), step 6 will be different. Rather than confirming the booking, the Ticket Service and Hotel Service will cancel the booking, and the Booking Service may return an appropriate error response to the user. Compensating transactions made by error responses from the Hotel Service or Payment Service will be similar to the above situation, so we will not discuss them.

Other points to note in choreography:

- There are no bidirectional lines i.e. a service does not both produce to and subscribe to the same topic.
- No two services produce to the same topic.
- A service can subscribe to multiple topics. If a service needs to receive multiple different events from multiple topics before it can perform an action, it needs to record in a database that it has received certain events, so it can read the database to determine if all the required events have been received.
- The relationship between topics and services can be 1:many or many:1, but not many:many.
- There may be cycles. Notice the cycle in figure 4.4 (Hotel Topic -> Payment Service -> Payment Topic -> Hotel Service -> Hotel Topic).

In figures 4.5, there are many lines between multiple topics and services. Choreography between a larger number of topics and services can become overly-complex, error-prone, and difficult to maintain.

4.4.2 Orchestration

In orchestration, the service that begins the saga is the orchestrator. The orchestrator communicates with each service via a Kafka topic. In each step in the saga, the orchestrator must produce to a topic to request this step to begin, and it must consume from another topic to receive the step's result.

An orchestrator is a finite state machine that reacts to events and issues commands. The orchestrator must only contain the sequence of steps. It must not contain any other business logic.

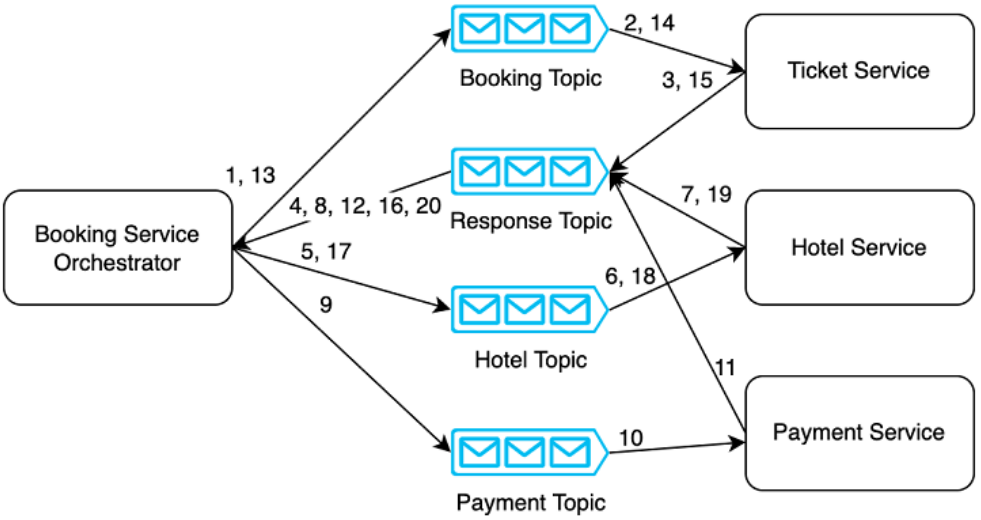

Figure 4.6 An orchestration saga to book an airline ticket and a hotel room for a tour package.

Figure 4.6 illustrates an orchestration saga to book a tour package. The steps in a successful booking process are as follows.

1. The orchestrator produces a ticket request event to the Booking Topic.
2. The Ticket Service consumes this ticket request event, and reserves the airline ticket for the booking ID with the state "AWAITING_PAYMENT".
3. The Ticket Service produces a "ticket pending payment" event to the Response Topic.
4. The orchestrator consumes the "ticket pending payment" event.
5. The orchestrator produces a hotel reservation request event to the Hotel Topic.
6. The Hotel Service consumes the hotel reservation request event, and reserves the hotel room for the booking ID with the state "AWAITING_PAYMENT".
7. The Hotel Service produces a "room pending payment" event to the Response Topic.
8. The orchestrator consumes the "room pending payment" event.
9. The orchestrator produces a payment request event to the Payment Topic.
10. The Payment Service consumes the payment request event.
11. The Payment Service processes the payment, then produces a payment confirmation event to the Response Topic.
12. The orchestrator consumes the payment confirmation event.
13. The orchestrator produces a payment confirmation event to the Booking Topic.
14. The Ticket Service consumes the payment confirmation event, and changes the state corresponding to that booking to "CONFIRMED".
15. The Ticket Service produces a ticket confirmation event to the Response Topic.
16. The orchestrator consumes this ticket confirmation event from Response Topic.

17. The orchestrator produces a payment confirmation event to the Hotel Topic.
18. The Hotel Service consumes this payment confirmation event, and changes the state corresponding to that booking to "CONFIRMED".
19. The Hotel Service produces a hotel room confirmation event to the Response Topic.
20. The Booking Service orchestrator consumes the hotel room confirmation event. It can then perform next steps, such as sending a success response to the user, or any further logic internal to the Booking Service.

Steps 18 and 19 appear unnecessary, as step 18 will not fail; it can keep retrying until it succeeds. Steps 18 and 20 can be done in parallel. However, we carry out these steps linearly to keep the approach consistent.

Steps 1-13 are compensable transactions. Step 14 is the pivot transaction. Steps 15 onward are retriable transactions.

If any of the three services produces an error response to the Booking topic, the orchestrator can produce events to the various other services to run compensating transactions.

4.4.3 Comparison

Table 4.1 compares choreography vs. orchestration.

Table 4.1 Choreography saga vs. orchestration saga.

Cheoreography	Orchestration
Requests to services are made in parallel. This is the observer object-oriented design pattern.	Requests to services are made linearly. This is the controller object-oriented design pattern.
The service that begins the saga communicates with 2 Kafka topics. It produces to 1 Kafka topic to start the distributed transaction, and consumes from another Kafka topic to perform any final logic.	The orchestrator communicates with each service via a Kafka topic. In each step in the saga, the orchestrator must produce to a topic to request this step to begin, and it must consume from another topic to receive the step's result.
The service that begins the saga only has code that produces to the saga's first topic and consumes from the saga's last topic. A developer must read the code of every service involved in the saga to understand its steps.	The orchestrator has code which produces and consumes to Kafka topics that correspond to steps in the saga, so reading the orchestrator's code allows one to understand the services and steps in the distributed transaction.
A service may need to subscribe to multiple Kafka topics, such as the Accounting Service in figure 4.5 of Richardson's book. This is because it may produce a certain event only when it has consumed certain other events from multiple services. This means that it must record in a database which events it has already consumed.	Other than the orchestrator, each service only subscribes to one other Kafka topic (from one other service). The relationships between the various services are easier to understand. Unlike choreography, a service never needs to consume multiple events from separate services before it can produce a certain event, so it may be possible to reduce the number of database writes.

Less resource-intensive. less chatty and less network traffic, hence lower latency overall.	Since every step must pass through the orchestrator, the number of events is double that of choreography. Overall effect is that orchestration is more resource-intensive; chattier and has more network traffic, hence higher latency overall.
Parallel requests also result in lower latency.	Requests are linear, so latency is higher.
Services have a less independent software development lifecycle, because developers must understand all services to change any one of them.	Services are more independent. A change to a service only affects the orchestrator, and does not affect other services.
No such single point of failure as in orchestration i.e. no service needs to be highly available except the Kafka service.	If the orchestration service fails, the entire saga cannot execute. I.e., the orchestrator and the Kafka service must be highly available.
Compensating transactions are triggered by the various services involved in the saga.	Compensating transactions are triggered by the orchestrator.

4.5 Other transaction types

The following consensus algorithms are typically more useful for achieving consensus for large numbers of nodes, typically in distributed databases. We will not discuss them in this book. Refer to Martin Kleppman's book "Designing Data-Intensive Applications" for more details.

- Quorum writes.
- Paxos and EPaxos.
- Raft
- Zab (ZooKeeper atomic broadcast protocol) - Used by Apache ZooKeeper.

4.6 Further reading

- Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems (2017) by Martin Kleppmann
- Boris Scholl, Trent Swanson, Peter Jausovec. Cloud Native: Using Containers, Functions, and Data to Build Next-Generation Applications. O'Reilly, 2019.
- Cornelia Davis. Cloud Native Patterns. Manning Publications, 2019.
- Chris Richardson. Microservices Patterns: With Examples in Java. Manning Publications, 2019. Chapter 3.3.7 discusses the transaction log tailing pattern. Chapter 4 is a detailed chapter on saga.

4.7 Summary

- A distributed transaction writes the same data to multiple services, with either eventual consistency or consensus.
- In event sourcing, write events are stored in a log (such as an event stream), and subscribers tail or process the log to obtain these events.
- In Change Data Capture (CDC), an event stream has multiple consumers, each

corresponding to a downstream service.

- A saga is a series of transactions that are either all completed successfully or all rolled back.
- Choreography (parallel) or orchestration (linear) are two ways to coordinate sagas.

5

Common services for functional partitioning

This chapter covers:

- Centralization of cross-cutting concerns with an API gateway, service mesh or sidecar pattern.
- Using a metadata service to minimize network traffic, by storing data required by multiple components within a system.
- Considering various web and mobile frameworks to fulfill one's specific requirements.
- Implementing functionality as libraries vs services.
- Selecting an appropriate API paradigm for a particular functionality, between REST, RPC, and GraphQL.

Earlier in this book, we discussed functional partitioning as a scalability technique that partitions out specific functions from our backend to run on their own dedicated clusters. This chapter first discusses the API Gateway, followed by the Sidecar Pattern (also called Service Mesh), which was a recent innovation. Next, we discuss centralization of common data into a metadata service. A common theme of these services is that they contain functionalities common to many backend services, and which we can partition from those services into shared common services.

Istio, a popular service mesh implementation, had its first production release in 2018.

Lastly, we discuss some frameworks that can be used to develop the various components in a system design.

5.1 Some common functionalities of various services

A service can have many non-functional requirements, and many services with different functional requirements can share the same non-functional requirements. For example, a service that calculates sales taxes and a service to check hotel room availability may both take advantage of caching to improve performance, or must only accept requests from registered users.

If engineers implement these functionalities separately for each service, there may be duplication of work or duplicate code. Errors or inefficiencies are more likely, as scarce engineering resources are spread out across a larger amount of work.

One possible solution is to place this code into libraries, and various services can use these common libraries. However, this solution has the disadvantages discussed in chapter 10.4. Library updates are controlled by users, so the services may continue to run old versions that contain bugs or security issues fixed in newer versions. Each host running the service also runs the libraries, so the different functionalities cannot be independently scaled.

A solution is to centralize these cross-cutting concerns with an API Gateway. The functionalities of an API Gateway include the following, which can be grouped into categories.

SECURITY

These functionalities prevent unauthorized access to a service's data.

- Authentication - Verifies that a request is from an authorized user.
- Authorization - Verifies that a user is allowed to make this request.
- SSL termination - Termination is usually not handled by the API Gateway itself, but by a separate HTTP proxy that runs as a process on the same host. We do termination on the API Gateway because termination on a load balancer is expensive. Although the term "SSL termination" is commonly used, the actual protocol is TLS, which is the successor to SSL.
- Server-side data encryption - If we need to store data securely on backend hosts or on a database, the API Gateway can encrypt data before storage, and decrypt data before it is sent to a requestor.

ERROR-CHECKING

Error-checking prevents invalid or duplicate requests from reaching service hosts, allowing them to process only valid requests.

- Request validation – One validation step is to ensure the request is properly formatted. For example, a POST request body should be valid JSON. Ensures that all required parameters are present in the request and their values honor constraints. We can configure these requirements on our service on our API Gateway.
- Request deduplication - Duplication may occur when a response with success status fails to reach the requestor/client, as the requestor/client may reattempt this request. Caching is usually used to store previously seen request IDs to avoid duplication. If our service is idempotent/stateless/"at least once" delivery, it can handle duplicate requests, and request duplication will not cause errors. However, if our service expects "exactly once" or "at most once" delivery, request duplication may cause

errors.

PERFORMANCE AND AVAILABILITY

An API Gateway can improve the performance and availability of services by providing caching, rate limiting, and request dispatching.

- Caching - The API Gateway can cache common requests to the database or other services such as:
    - In our service architecture, the API Gateway may make requests to a Metadata Service (refer to section 5.4). It can cache information on the most actively used entities.
    - Identity information to save calls to authentication and authorization services.
- Rate Limiting (also called throttling) - Prevents our service from being overwhelmed by requests. (Refer to chapter 12 for a discussion on a sample Rate Limiting service.)
- Request dispatching - The API Gateway makes remote calls to other services. It creates HTTP clients for these various services and ensures that requests to these services are properly isolated. When 1 service experiences slowdown, requests to other services are not affected. Common patterns like bulkhead and circuit breaker help implement resource isolation and make services more resilient when remote calls fail.

LOGGING AND ANALYTICS

Another common functionality provided by an API Gateway is request logging or usage data collection, which is the gathering real-time information for various purposes such as analytics, auditing, billing, and debugging.

5.2 API gateway

An *API Gateway* is a lightweight web service that consists of stateless machines located across several data centers. It provides common functionalities to our organization's many services, for centralization of cross-cutting concerns across various services, even if they are written in different programming languages. It should be kept as simple as possible despite its many responsibilities.

If an API Gateway only serves a single service rather than many services, it may be called a "Frontend Service" or may be a frontend module implemented in that service and not a separate service. This term "frontend" shares the same term commonly used to refer to a web UI service in the system, but is a completely different concept. This is an unfortunate overloaded term that causes confusion when we view it in various books, technical videos, or other technical documentation. The interviewer may not have heard of this "frontend" concept so we may need to describe it to her, as always in a clear and organized manner.

Amazon API Gateway (https://aws.amazon.com/api-gateway/) and Kong (https://konghq.com/kong) are examples of cloud-provided API gateways.

5.3 Service Mesh / Sidecar pattern

Section 1.4.6 briefly discussed using a service mesh to address the disadvantages of an API Gateway, repeated below.

- Additional latency in each request, from having to route the request through an additional service.
- Large cluster of hosts, which requires scaling to control costs.

Figure 5.1 is a repeat of figure 1.8, illustrating a service mesh. A slight disadvantage of this design is that a service's host will be unavailable if its sidecar is unavailable, even if the service is up; this is the reason we generally do not run multiple services or containers on a single host.

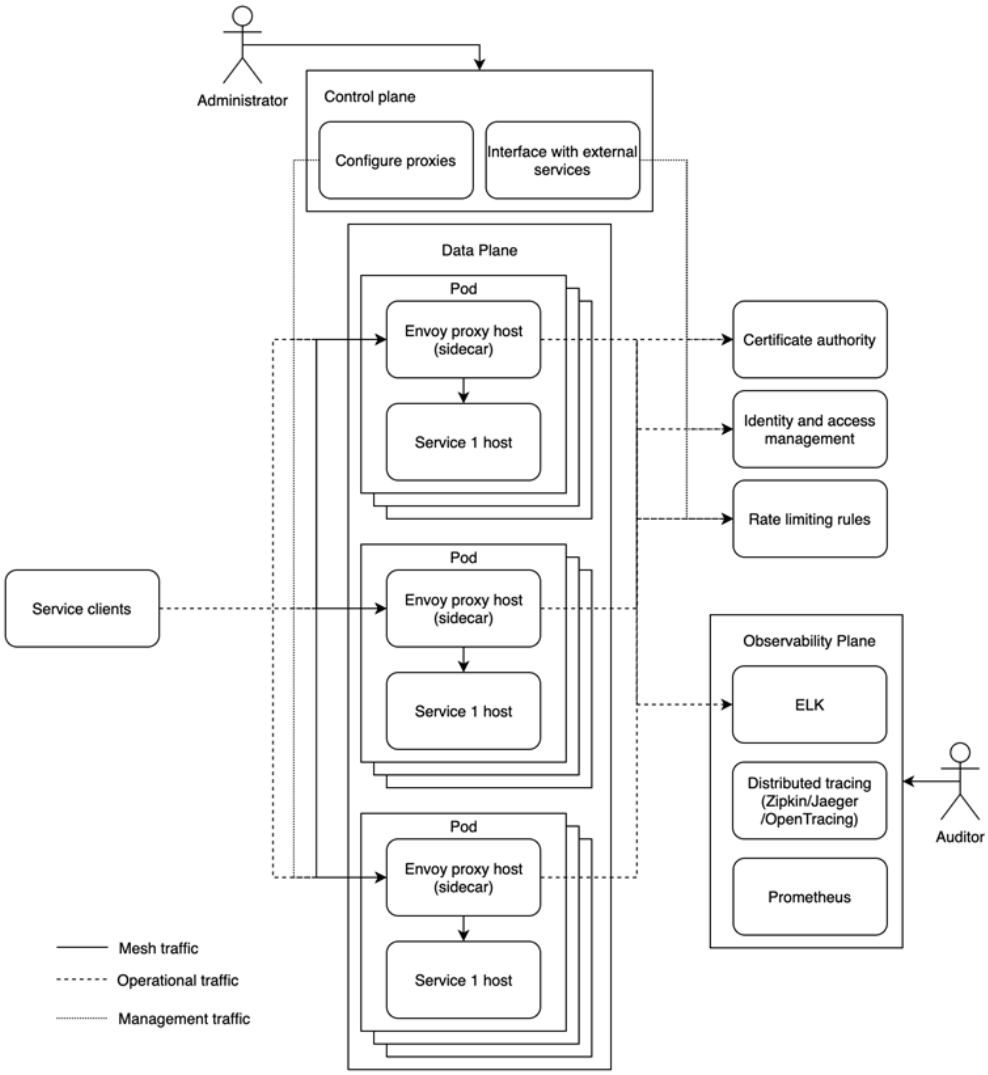

Figure 5.1 Illustration of a service mesh, repeated from figure 1.8.

Istio's documentation states that a service mesh consists of a Control Plane and a Data Plane (https://istio.io/latest/docs/ops/deployment/architecture/), while Jenn Gile from Nginx also described an Observability Plane (https://www.nginx.com/blog/how-to-choose-a-service-mesh/). Figure 1.8 contains all 3 types of planes.

An administrator can use the Control Plane to manage proxies, and interface with external services. For example, the Control Plane can connect to a certificate authority to

obtain a certificate, or to an identity and access control service to manage certain configurations. It can also push the certificate ID or the identify and access control service configurations to the proxy hosts. Interservice and intraservice requests occur between the Envoy (https://www.envoyproxy.io/) proxy hosts, which we refer to as mesh traffic. Sidecar proxy interservice communication can use various protocols including HTTP and gRPC (https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/service-mesh-communication-infrastructure). The Observability Plane provides logging, monitoring, alerting, and auditing.

Rate limiting is another example of a common shared service that can be managed by a service mesh. Chapter 12 discusses this in more detail.

AWS App Mesh (https://aws.amazon.com/app-mesh) is a cloud-provided service mesh.

Refer to section 1.4.6 for a brief discussion on sidecarless service mesh.

5.4 Metadata Service

A Metadata Service stores information that is used by multiple components within a system. If these components pass this information between each other, they can pass IDs rather than all the information. A component that receives an ID can request the Metadata Service for the information that corresponds to that ID. There is less duplicate information in the system, analogous to SQL normalization, so there is better consistency.

One example is ETL pipelines. Consider an ETL pipeline for sending welcome emails for certain products that users have signed up for. The email message may be a HTML file of several MB that contains many words and images, which are different according to product. Referring to figure 5.2, when a producer produces a message to the pipeline queue, instead of including an entire HTML file in the message, the producer can only include the ID of the file. The file can be stored in a Metadata Service. When a consumer consumes a message, it can request the Metadata Service for the HTML file that corresponds to that ID. This approach saves the queue from containing large amounts of duplicate data.

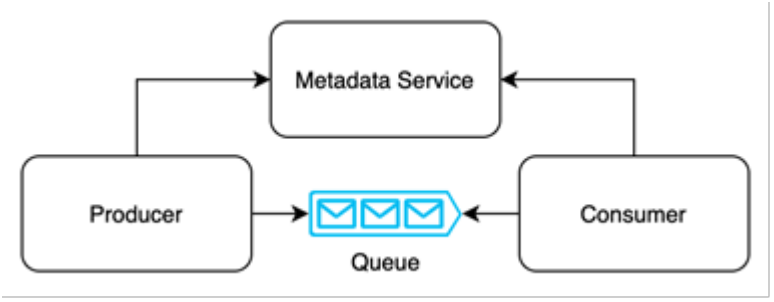

Figure 5.2 We can use a Metadata Service to reduce the size of individual messages in a queue, by placing large objects in the Metadata Service, and enqueuing only IDs in individual messages.

A tradeoff of using a metadata service is increased complexity and overall latency. Now the producer must write both to the Metadata Service and the queue. In certain designs, we may populate the Metadata Service in an earlier step, so the producer does not need to write to the Metadata Service.

If the producer cluster experiences traffic spikes, it will make a high rate of read requests to the Metadata Service, so the Metadata Service should be capable of supporting high read volumes.

In summary, a Metadata Service is for ID lookups. We will use metadata services in many of our sample question discussions in part 2.

Figure 5.3 illustrates the architecture changes from introducing the API Gateway and Metadata Services. Instead of making requests to the backend, clients will make requests to the API Gateway, which perform some functions and may send requests to either the Metadata Service and/or the backend. Figure 1.8 illustrates a Service Mesh.

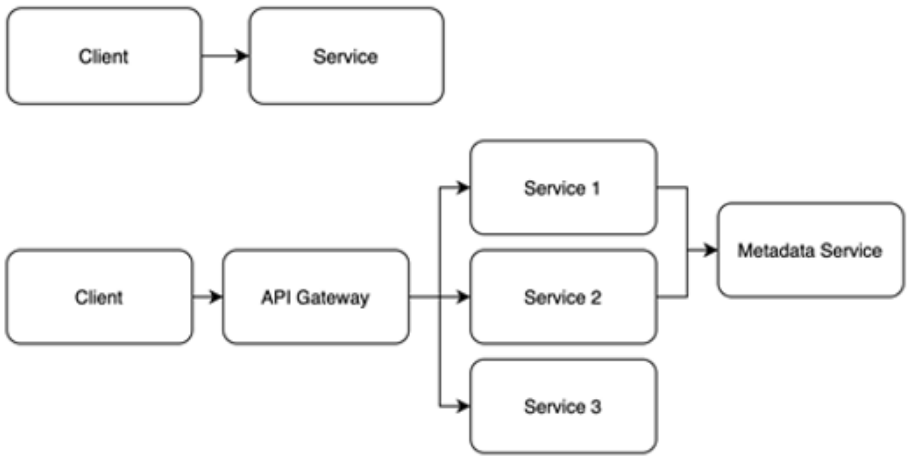

Figure 5.3 Functional partitioning of a service (top) to separate out the API Gateway and Metadata Service (bottom). Before this partitioning, clients query the service directly. With the partitioning, clients query the API Gateway, which perform some functions and may route the request to one of the services, which in turn may query the Metadata service for certain shared functionalities.

5.5 Service discovery

Service discovery is a microservices concepts that might be briefly mentioned during an interview in the context of managing multiple services. Service discovery is done under the hood, and most engineers do not need to understand their details. Most engineers only need to understand that each internal API service is typically assigned a port number via which it is accessed. External API services and most UI services are assigned URLs via which they are accessed. Service discovery may be covered in interviews for teams that develop

infrastructure. It is unlikely that the details of service discovery will be discussed for other engineers, as it provides little relevant interview signal.

Very briefly, service discovery is a way for clients to identify which service hosts are available. A service registry is a database that keeps track of the available hosts of a service. Refer to sources such as https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/service-discovery.html for details on service registries in Kubernetes and AWS. Refer to https://microservices.io/patterns/client-side-discovery.html and https://microservices.io/patterns/server-side-discovery.html for details on client-side discovery and server-side discovery.

5.6 Functional partitioning and various frameworks

In this section, we discuss some of the countless frameworks that can be used to develop the various components in a system design diagram. New frameworks are continuously being developed, and various frameworks fall in and out of favor with the industry. The sheer number of frameworks can be confusing to a beginner. Moreover, certain frameworks can be used for more than one component, making the overall picture even more confusing. This section is a broad discussion of various frameworks, in particular the following types of frameworks.

- Web
- Mobile, including Android and iOS.
- Backend
- PC

The universe of languages and frameworks is far bigger than can be covered in this chapter, and it is not the aim of this chapter to discuss them all. The purpose of this chapter is to provide some awareness of some frameworks and languages. By the end of this chapter, you should be able to more easily read the documentation of a framework to understand its purposes and where it fits into a system design.

5.6.1 Basic system design of an app

Figure 1.1 introduced a basic system design of an app. In almost all cases today, a company that develops a mobile app that makes requests to a backend service will have an iOS app on the iOS app store and an Android app on the Google Play Store. It may also develop a browser app that has the same features of the mobile apps, or may be a simple page that directs users to download the mobile apps. There are many variations. For example, a company may also develop a PC app. But attempting to explain every possible combination is counterproductive, and we will not do so.

We will start with discussing the following questions regarding figure 1.1, then expand our discussion to various frameworks and their languages.

- Why is there a separate web server application from the backend and browser app?
- Why does the browser app make requests to this node.js app, which then make requests to the backend that is shared with the Android and iOS apps?

5.6.2 Purposes of a web server app

The purposes of a web server app include the following:

- When someone using a web browser accesses the URL (e.g. https://google.com/), the browser downloads the browser app from the node.js app. As stated on section 1.4.1, the browser app should preferably be small so it can be downloaded quickly.
- When the browser makes a specific URL request (e.g. with a specific path like https://google.com/about), node.js handles the routing of the URL, and serves the corresponding page.
- The URL may include certain path and query parameters that require specific backend requests. The node.js app processes the URL and makes the appropriate backend requests.
- Certain user actions on the browser app, such as filling then submitting forms, or clicking on buttons, may require backend requests. A single action may correspond to multiple backend requests, so the node.js app exposes its own API to the browser app. Referring to figure 6.1, for each user action, the browser app makes an API request to the node.js app/server, which then makes one or more appropriate backend requests and returns the requested data.

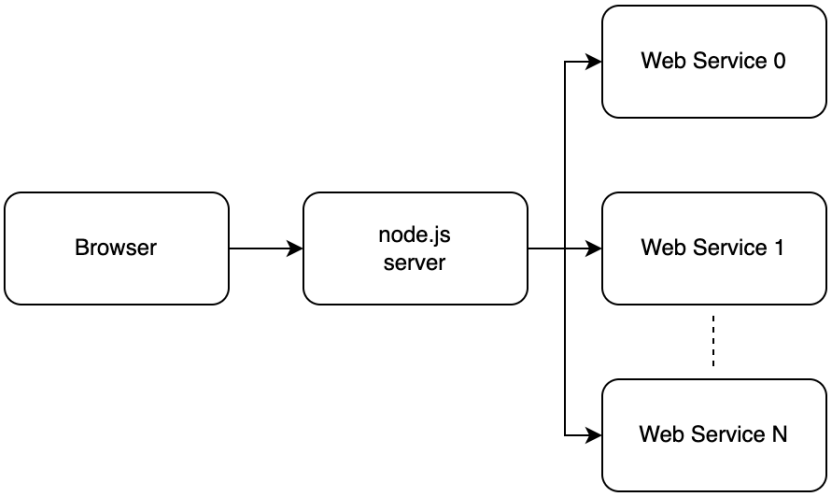

Figure 5.4 A node.js server can serve a request from a browser by making appropriate requests to one or more web services, aggregate and process their responses, and return the appropriate response to the browser.

Why doesn't a browser make requests directly to the backend? If the backend was a REST app, its API endpoints may not return the exact data required by the browser. The browser may have to make multiple API requests and fetch more data than required. This data transmission occurs over the Internet, between a user's device and a data center, which is

inefficient. It is more efficient for the node.js app to make these large requests, as the data transmission will likely happen between adjacent hosts in the same data center. The node.js app can then return the exact data required by the browser.

GraphQL apps allow users to request the exact data required, but securing GraphQL endpoints is more work than a REST app, causing more development time and possible security breaches. Other disadvantages include the following. Refer to section 7.4 for more discussion on GraphQL.

- Flexible queries mean that more work is required to optimize performance.
- More code on the client.
- More work needed to define the schema.
- Larger requests.

5.6.3 Web and mobile frameworks

This section contains a list of frameworks, classified into the following.

- Web/browser app development.
- Mobile app development.
- Backend app development.
- PC aka desktop app development i.e. for Windows, Mac, and Linux.

A complete list will be very long, include many frameworks that one will unlikely encounter or even read about during her career, and in the author's opinion will not be useful to the reader. This list only states some of the frameworks that are prominent or used to be prominent.

The flexibility of these space makes a complete and objective discussion difficult. Frameworks and languages are developed in countless ways, some of which make sense and others which do not.

BROWSER APP DEVELOPMENT

Browsers accept only HTML, CSS, and JavaScript, so browser apps must be in these languages for backward compatibility. A browser is installed on a user's device, so it must be upgraded by the user herself, and it is difficult and impractical to persuade or force users to download a browser that accepts another language. It is possible to develop a browser app in vanilla JavaScript (i.e. without any frameworks), but this is impractical for all but the smallest browser apps, because frameworks contain many functions that one will otherwise have to reimplement in vanilla JavaScript, e.g. animation, or data rendering like sorting tables or drawing charts.

Although browser apps must be in these 3 languages, a framework can offer other languages. The browser app code written in these languages are transpiled to HTML, CSS and JavaScript.

The most popular browser app frameworks include React, Vue.js, and Angular. Other frameworks include Meteor, jQuery, Ember.js, and Backbone.js. A common theme of these frameworks is that developers mix the markup and logic in the same files, rather than having separate HTML files for markup and JavaScript files for logic. These frameworks may also contain their own languages for markup and logic. For example, React introduced JSX, which

is a HTML-like markup language. A JSX file can include both markup and JavaScript functions and classes. Vue.js has the template tag, which is similar to HTML.

Some of the more prominent web development languages (which are transpiled to JavaScript) include the following:

- TypeScript (https://www.typescriptlang.org/) is a statically-typed language. It is a wrapper/superset around JavaScript. Virtually any JavaScript framework can also use TypeScript, with some setup work.
- Elm (https://elm-lang.org/) can be directly transpiled to HTML, CSS and JavaScript, or it can also be used within other frameworks like React.
- PureScript (https://www.purescript.org/) aims for similar syntax as Haskell.
- Reason (https://reasonml.github.io/).
- ReScript (https://rescript-lang.org/).
- Clojure (https://clojure.org/) is a general-purpose language. The ClojureScript (https://clojurescript.org/) framework transpiles to Clojure to JavaScript.
- CoffeeScript (https://coffeescript.org/).

These browser app frameworks are for the browser/client side. Here are some server-side frameworks. Any server-side framework can also make requests to databases, and be used for backend development. In practice, a company often chooses one framework for server development and another framework for backend development. This is a common point of confusion for beginners trying to distinguish between "server-side frontend" frameworks and "backend" frameworks. There is no strict division between them.

- Express (https://expressjs.com/) is a Node.js (https://nodejs.org/) server framework. Node.js is a JavaScript runtime environment built on Chrome's V8 JavaScript engine. The V8 JavaScript engine was originally built for Chrome, but can also run on an operating system, like Linux or Windows. The purpose of Node.js is for JavaScript code to run on an operating system. Most frontend or full-stack job postings that state Node.js as a requirement are actually referring to Express.
- Deno (https://deno.land/) supports JavaScript and TypeScript. It was created by Ryan Dahl, the original creator of node.js, to address his regrets about node.js.
- Goji (https://goji.io/) is a Golang framework.
- Rocket (https://rocket.rs/) is a Rust framework. Refer to https://blog.logrocket.com/the-current-state-of-rust-web-frameworks/ for more examples of web server and backend (section 6.3.4 below) frameworks.
- Vapor (https://vapor.codes/) is a framework for the Swift language.
- Vert.x (https://vertx.io/) offers development in Java, Groovy, and Kotlin.
- PHP (https://www.php.net/). (There is no universal agreement on whether PHP is a language or a framework. The author's opinion is that there is no practical value in debating this semantic.) A common solution stack is the LAMP (Linux, Apache, MySQL, PHP/Perl/Python) acronym. PHP code can be run on an Apache (https://httpd.apache.org/) server, which in turn runs on a Linux host. PHP was popular before ~2010 (https://www.tiobe.com/tiobe-index/php/), but in the author's experience, PHP code is seldom directly used for new projects. PHP remains prominent for web development via the WordPress platform, which is useful for

building simple websites. More sophisticated user interfaces and customizations are more easily done by web developers, using frameworks that require considerable coding, such as React and Vue.js. Meta (formerly known as Facebook) was a prominent PHP user. The Facebook browser app was formerly developed in PHP. In 2014, Facebook introduced the Hack language (https://hacklang.org/) and HipHop Virtual Machine (HHVM) (https://hhvm.com/). Hack is a PHP-like language that does not suffer from the bad security and performance of PHP. It runs on HVVM. Meta is an extensive user of Hack and HHVM.

MOBILE APP DEVELOPMENT

The dominant mobile operating systems are Android and iOS, developed by Google and Apple respectively. Google and Apple each offer their own Android or iOS app development platform, which are commonly referred to as "native" platforms. The native Android development languages are Kotlin and Java, while the native iOS development languages are Swift and Objective-C.

CROSS-PLATFORM DEVELOPMENT

Cross-platform development frameworks in theory reduce duplicate work by running the same code on multiple platforms. In practice, there may be additional code required to code the app for each platform, which will negate some of this benefit. Such situations occur when the UI (user interface) components provided by operating systems are too different from each other.

Frameworks which are cross-platform between Android and iOS include the following.

- React Native. React Native is distinct from React. The latter is for web development only. There is also a framework called React Native for Web (https://github.com/necolas/react-native-web), which allows web development using React Native.
- Flutter (https://flutter.dev/) is cross-platform across Android, iOS, web, and PC.
- Ionic (https://ionicframework.com/) is cross-platform across Android, iOS, web, and PC.
- Xamarin (https://dotnet.microsoft.com/en-us/apps/xamarin) is cross-platform across Android, iOS, and Windows.

Electron (https://www.electronjs.org/) is cross-platform between web and PC.

Cordova (https://cordova.apache.org/) is a framework for mobile and PC development using HTML, CSS and JavaScript. With Cordova, cross-platform development with web development frameworks like Ember.js is possible.

BACKEND DEVELOPMENT

Here is a list of backend development frameworks. As stated in chapter 7, backend frameworks can be classified into RPC, REST, and GraphQL. Some backend development frameworks are full-stack i.e. they can be used to develop a monolithic browser application that makes database requests. We can also choose to use them for browser app development, and make requests to a backend service developed in another framework, but the author has never heard of these frameworks being used this way.

- gRPC (https://grpc.io/) is a RPC framework that can be developed in C#, C++, Dart, Golang, Java, Kotlin, Node, Objective-C, PHP, Python, or Ruby. It may be extended to other languages in the future.
- Thrift (https://thrift.apache.org/) and Protocol Buffers (https://developers.google.com/protocol-buffers) are used to serialize data objects, compressing them to reduce network traffic. An object can be defined in a definition file. We can then generate client and server (backend, not web server) code from a definition file. Clients can use the client code to serialize requests to the backend, which uses the backend code to deserialize the requests, and vice versa for the backend's responses. Definition files also help to maintain backward and forward compatibility by placing limitations on possible changes.
- Spring Boot (https://spring.io/projects/spring-boot) and Dropwizard (https://www.dropwizard.io/) are examples of Java REST frameworks.
- Flask (https://flask.palletsprojects.com/) and Django (https://www.djangoproject.com/) are 2 examples of REST frameworks in Python. They can also be used for web server development.

Here are some examples of full-stack frameworks.

- Dart (https://dart.dev) is a language that offers frameworks for any solution. It can be used for full-stack, backend, server, browser, and mobile apps.
- Rails (https://rubyonrails.org/) is a Ruby full-stack framework that can also be used for REST. Ruby on Rails is often used as a single solution, rather than using Ruby with other frameworks or Rails with other languages.
- Yesod (https://www.yesodweb.com/) is a Haskell framework that can also be used just for REST. Browser app development can be done with Yesod using its Shakespearen template languages https://www.yesodweb.com/book/shakespearean-templates, which transpile to HTML, CSS and JavaScript.
- Integrated Haskell Platform (https://ihp.digitallyinduced.com/) is another Haskell framework.
- Phoenix (https://www.phoenixframework.org/) is a framework for the Elixir language.
- JavaFX (https://openjfx.io/) is a Java client application platform for desktop, mobile, and embedded systems. It is descended from Java Swing (https://docs.oracle.com/javase/tutorial/uiswing/), for developing GUI for Java programs.
- Beego (https://beego.vip/) and Gin (https://gin-gonic.com/) are Golang frameworks.

5.7 Library vs Service

After determining our system's components, we can discuss the pros and cons of implementing each component on the client-side vs server-side, as a library vs service. Do not immediately assume that a particular choice is best for any particular component. In most situations, there is no obvious choice between using a library vs service, so we need to be able to discuss design and implementation details and tradeoffs for both options.

A library may be an independent code bundle, may be a thin layer that forwards all requests and responses between clients and servers respectively, or may contain elements of

both i.e. some of the API logic is implemented within the library while the rest may be implemented by services called by the library. In this chapter, for the purpose of comparing libraries vs services, the term "library" refers to an independent library.

Table 5.1 summarizes a comparison of libraries vs services. Most of these points are discussed in detail in the rest of this chapter.

Table 5.1 Summary comparison of libraries vs services.

Library	Service
Users choose which version/build to use, and have more choice on upgrading to new versions. A disadvantage is that users may continue to use old versions of libraries that contain bugs or security issues fixed in newer versions. Users who wish to always use the latest version of a frequently-updated library have to implement programmatic upgrades themselves.	Developers select the build and control when upgrades happen.
No communication or data sharing between devices limits applications. If the user is another service, this service is horizontally scaled, and data sharing between hosts is needed, the customer service's hosts must be able to communicate with each other to share data. This communication must be implemented by the user service's developers.	No such limitation. Data synchronization between multiple hosts can be done via requets to each other or to a database. Users need not be concerned about this.
Language-specific.	Technology-agnostic.
Predictable latency.	Less predictable latency due to dependence on network conditions.
Predictable reproducible behavior.	Network issues are unpredictable and difficult to reproduce, so behavior may be less predictable and less reproducible.
If we need to scale up the load on the library, the entire application must be scaled up with it. Scaling costs are borne by the user's service.	Independently scalable. Scaling costs are borne by the service.
Users may be able to decompile the code to steal intellectual property.	Code is not exposed to users. (Though APIs can be reverse engineered. This is outside the scope of this book.)

5.7.1 Language specific vs technology-agnostic

For ease of use, a library should be in the client's language, so the same library must be reimplemented in each supported language.

Most libraries are optimized to perform a well-defined set of related tasks, so they can be optimally implemented in a single language. However, certain libraries may be partially or completely written in another language, as certain languages and frameworks may be better suited for specific purposes. Implementing this logic entirely in the same language may cause inefficiencies during use. Moreover, while developing our library, we may want to utilize libraries written in other languages. There are various utility libraries that one can use to develop a library that contains components in other languages. This is outside the scope of this book. A practical difficulty is that the team or company that develops this library will require engineers fluent in all of these languages.

A service is technology agnostic, as a client can utilize a service regardless of the former's or latter's technology stacks. A service can be implemented in the language and frameworks best suited for its purposes. There is a slight additional overhead for clients, who will need to instantiate and maintain HTTP, RPC, or GraphQL connections to the service.

5.7.2 Predictability of latency

A library has no network latency, has guaranteed and predictable response time, and can be easily profiled with tools such as flame graphs.

A service has unpredictable and uncontrollable latency as it depends on numerous factors such as:

- Network latency, which depends on the user's Internet connection quality.
- The service's ability to handle its current traffic volume.

5.7.3 Predictability and reproducibility of behavior

A service has less predictable and reproducible behavior than a library, as its behavior has more dependencies such as:

- A deployment rollout is usually gradual i.e. the build is deployed to a few service hosts at a time. Requests may be routed by the load balancer to hosts running different builds, resulting in different behavior.
- Users do not have complete control of the service's data, and it may be changed between requests by the service's developers. This is unlike a library, where users have complete control of their machine's file system.
- A service may make requests to other services and be affected by their unpredictable and unreproducible behavior.

Despite these factors, a service is often easier to debug than a library, because:

- A service's developers have access to its logs, while a library's developers do not have access to the logs on the users' devices.
- A service's developers control its environment, and can set up a uniform environment using tools like virtual machines and Docker for its hosts. A library users run it on a diversity of environments such as variations in hardware, firmware, and OS (Android vs iOS). A user may choose to send her crash logs to the developers, but it may still be difficult to debug without access to the user's device and exact environment.

5.7.4 Scaling considerations for libraries

A library cannot be independently scaled up, since it is contained within the user's application. It does not make sense to discuss scaling up a library on a single user device. If the user's application runs in parallel on multiple devices e.g. a user can scale up the library by scaling the application that uses it. To scale just the library alone, the user can create her own service that is a wrapper around that library, and scale that service. But this won't be a library anymore, but simply a service that is owned by the user, so scaling costs are borne by the user.

5.7.5 Other considerations

This section briefly described some anecdotal observations from the author personal experiences.

Some engineers have psychological hesitations in bundling their code with libraries, but are open to connecting to services. They may be concerned that a library will inflate their build size, especially for JavaScript bundles. They are also concerned about the possibility of malicious code in libraries, while this is not a concern for services since the engineers control the data sent to services and have full visibility of the service's responses.

People expect breaking changes to occur in libraries, but are less tolerant of breaking changes in services, particular internal services. Service developers may be forced to adopt clumsy API endpoint naming conventions such as including terms like "/v2/", "/v3" etc. in their endpoint names.

Anecdotal evidence suggests that Adapter Pattern is more followed more often when using a library instead of a service.

5.8 Common API paradigms

This section introduces and compares the following common communication paradigms. One should consider the tradeoffs in selecting a paradigm for her service.

- REST (Representational State Transfer)
- RPC (Remote Procedure Call)
- GraphQL
- WebSocket

5.8.1 The Open Systems Interconnection (OSI) model

The *7-layer OSI model* is a conceptual framework/model that characterizes the functions of a networking system without regard to its underlying internal structure and technology. Table 5.2 briefly describes each layer. A convenient way to think of this model is that the protocols of each level are implemented using protocols of the lower level.

Actor, GraphQL, REST, and WebSocket are implemented on top of HTTP. RPC is classified as layer 5, because it handles connections, ports, and sessions directly, rather than relying on a higher-level protocol like HTTP.

Table 5.2 The OSI model.

Layer no.	Name	Description	Examples
7	Application	User interface.	FTP, HTTP, Telnet.
6	Presentation	Presents data. Encryption occurs here.	UTF, ASCII, JPEG, MPEG, TIFF.
5	Session	Distinction between data of separate applications. Maintains connections. Controls ports and sessions.	RPC, SQL, NFX, X Windows.
4	Transport	End-to-end connections. Defines reliable or unreliable delivery and flow control.	TCP, UDP.
3	Network	Logical addressing. Defines the physical path the data uses. Routers work at this layer.	IP, ICMP.
2	Data link	Network format. May correct errors at the physical layer.	Ethernet, WiFi.
1	Physical	Raw bits over physical medium.	Fiber, coax, repeater, modem, network adapter, USB.

5.8.2 REST

We assume the reader is familiar with the basics of REST as a stateless communication architecture that uses HTTP methods and request/response body most commonly encoded in JSON or XML. In this book, we use REST for APIs and JSON for POST request and response body. We can represent JSON schema with the specification by the JSON Schema organization (https://json-schema.org/), but we don't do so in this book because it is usually too verbose and low-level to discuss JSON schemas in detail in a 50-minute system design interview.

REST is simple to learn, set up, and experiment and debug with (using curl or a web browser). Its other advantages include its hypermedia and caching capabilities, which we discuss below.

HYPERMEDIA

Hypermedia controls (HATEOAS) or hypermedia is about providing a client with information about "next available actions" within a response. This takes the form of a field such as "links" within a response JSON, which contain API endpoints that the client may logically query next.

For example, after an ecommerce app displays an invoice, the next step is for the client to make payment. The response body for an invoice endpoint may contain a link to a payment endpoint, such as this:

```
{
  “data”: {
    “type”: “invoice”,
    “id”: “abc123”,
  },
  “links”: {
    “pay”: “https://api.acme.com/payment/abc123”
  }
}
```

where the response contains an invoice ID, and the next step is to POST a payment for that invoice ID.

There is also the OPTIONS HTTP method, which is for fetching metadata about an endpoint, such as available actions, fields that can be updated, or what data do certain fields expect.

In practice, hypermedia and OPTIONS are difficult for client developers to use, and it makes more sense to provide a client developer with API documentation of each endpoint or function, such as using OpenAPI (https://swagger.io/specification/) for REST or the built-in documentation tools of RPC and GraphQL frameworks.

Refer to https://jsonapi.org/ for conventions on request/response JSON body specification.

Other communication architectures like RPC or GraphQL do not provide hypermedia.

CACHING

Developers should declare REST resources as cacheable whenever possible, a practice which carries advantages such as the following:

- Lower latency, as some network calls are avoided.
- Higher availability, as the resource is available even if the service is not.
- Better scalability, since there is lower load on the server.

Use the “Expires”, “Cache-Control”, “ETag”, and “Last-Modified” HTTP headers for caching.

The Expires HTTP header specifies an absolute expiry time for a cached resource. A service can set a time value up to 1 year ahead of its current clock time. An example header is “Expires: Mon, 11 Dec 2021 18:00 PST”.

The Cache-Control header consists of comma-separated directives (instructions) for caching in both requests and responses. An example header is “Cache-Control: max-age=3600”, which means the response is cacheable for 3600 seconds. A POST or PUT request (noun) may include a Cache-Control header as a directive to the server to cache this data, but this does not mean that the server will follow this directive, and this directive might not be contained in responses for this data. Refer to https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control for all cache request and response directives.

An ETag value is a opaque string token59 that is an identifier for a specific version of a resource. A client can refresh its resource more efficiently by including the ETag value in the GET request. The server will only return the resource’s value if the latter’s ETag is different i.e. the resource’s value changed, so it does not unnecessarily return the resource’s value if the client already has it.

The Last-Modified header contains the date and time the resource was last modified, and can be used as a fallback for the ETag header if the latter is unavailable.

Related headers are If-Modified-Since and If-Unmodified-Since.

DISADVANTAGES OF REST

A disadvantage is that it has no integrated documentation mechanisms, other than hypermedia or OPTIONS endpoints, which developers can choose not to provide. One must add an OpenAPI documentation framework to a service implemented using a REST framework. Otherwise, clients have no way of knowing the available request endpoints, or their details such as path or query parameters or the request and response body fields. REST also has no standardized versioning procedure; a common convention is to use a path like "/v2" or "/v3" etc for versioning. Another disadvantage of REST is that it does not have a universal specification, which leads to confusion. OData and JSON-API are 2 popular specifications.

5.8.3 RPC (Remote Procedure Call)

RPC is a technique to make a procedure execute in a different address space i.e. another host, without the programmer having to handle the network details. Popular open-source RPC frameworks include Google's gRPC, Facebook's Thrift, and RPyC in Python.

For an interview, you should be familiar with the following common encoding formats. Understand how encoding (also called serialization or marshalling) and decoding (also called parsing, deserialization, or unmarshalling) are done.

- CSV, XML, JSON
- Thrift
- Protocol Buffers (protobuf)
- Avro

The main advantages of RPC frameworks like gRPC over REST are:

- RPC is designed for resource optimization, so it is the best communication architecture choice for low-power devices, such as IoT devices like smart home devices. For a large web service, its lower resource consumption compared to REST or GraphQL becomes significant with scale.
- Efficient. protobuf is an efficient encoding. JSON is repetitive and verbose, causing requests and responses to be large. Network traffic savings become significant with scale.
- Developers define the schemas of their endpoints in files. Common formats include Avro, Thrift, and protobuf. Clients use these files to create requests and interpret responses. As schema documentation is a required step in developing the API, client developers will always have good API documentation. These encoding formats also have schema modification rules, which make it clear to developers how to maintain backward and/or forward compatibility in schema modifications.

The main disadvantages of RPC are also from its nature as a binary protocol. It is troublesome for clients to have to update to the latest version of a schema files, especially

outside an organization. Also, if an organization wishes to monitor its internal network traffic, it is easier to do so with text protocols like REST than with binary protocols like RPC.

5.8.4 GraphQL

GraphQL is a flexible communication architecture. It provides an API data query and manipulation language for pinpoint requests. It also provides an integrated API documentation tool that essential for navigating this flexibility.

Main benefits:

- The client decides what data they want and its format.
- Efficiency. The server delivers exactly what the client requests without under fetching (which necessitates multiple requests) or over fetching (which inflates response size).

Tradeoffs:

- May be too complex for simple APIs.
- Higher learning curve than RPC and REST, including security mechanisms.
- Smaller user community than RPC and REST.
- Encoding in JSON only, which carries all the tradeoffs of JSON.
- User analytics may be more complicated, as each API user perform slightly different queries. In REST and RPC, we can easily see how many queries were made to each API endpoint, but this is less obvious in GraphQL.
- We should be cautious when using GraphQL for external APIs. It is similar to exposing a database and allowing clients to make SQL queries.

Many of the benefits of GraphQL can be done in REST. A simple API can begin with simple REST HTTP methods (GET, POST, PUT, DELETE) with simple JSON bodies. As its requirements become more complex, it can use more REST capabilities such as OData https://www.odata.org/, or use JSON-API capabilities like https://jsonapi.org/format/#fetching-includes to combine related data from multiple resources into a single request. GraphQL may be more convenient than REST in addressing complex requirements, because it provides a standard implementation and documentation of its capabilities. REST on the other hand has no universal standard.

5.8.5 WebSocket

WebSocket is a communications protocol for full-duplex communication over a persistent TCP connection, unlike HTTP which creates a new connection for every request and closes it with every response. REST, RPC, GraphQL, and Actor model are design patterns or philosophies, while WebSocket and HTTP are communication protocols. However, it makes sense to compare WebSocket to the rest as API architectural styles, because we can choose to implement our API using WebSocket rather than the other 4 choices.

To create a WebSocket connection, a client sends a WebSocket request to the server. WebSocket uses a HTTP handshake to create an initial connection, and requests the server to upgrade to WebSocket from HTTP. Subsequent messages can use WebSocket over this persistent TCP connection.

WebSocket keeps connections open, which increases overhead on all parties. This means that WebSocket is stateful (compared to REST and HTTP, which are stateless). A request must be handled by the host that contains the relevant state/connection, unlike in REST where any host can handle any request. Both the stateful nature of WebSocket and the resource overhead of maintaining connections means that WebSocket is less scalable.

WebSocket allows p2p communication, so no backend is required. It trades off scalability for lower latency and higher performance.

5.8.6 Comparison

During an interview, we may need to evaluate the tradeoffs between these architectural styles and the factors to consider in choosing a style and protocol. REST and RPC are the most common. Startups usually use REST for simplicity, while large organizations can benefit from RPC's efficiency and backward and forward compatibility. GraphQL is a relatively new philosophy. WebSocket is useful for bidirectional communication, including p2p communication.

Other references include https://phil.tech/2018/picking-an-api-paradigm-implementation/ and https://www.baeldung.com/rest-vs-websockets.

5.9 Summary

- An API Gateway is a web service designed to be stateless and lightweight, yet fulfill many cross-cutting concerns across various services, which can be grouped into security, error-checking, performance and availability, and logging.
- A service mesh or sidecar pattern is an alternative pattern. Each host gets its own sidecar, so no service can consume an unfair share.
- To minimize network traffic, we can consider using a metadata service to store data that is processed by multiple components within a system.
- Service discovery is for clients to identify which service hosts are available.
- A browser app can have 2 or more backend services. One of them is a web server service that intercepts requests and responses from the other backend services.
- A web server service minimizes network traffic between the browser and data center, by performing aggregation and filtering operations with the backend.
- Browser app frameworks are for browser app development. Server-side frameworks are for web service development. Mobile app development can be done with native or cross-platform frameworks.
- There are cross-platform or full-stack frameworks for developing browser apps, mobile apps, and web servers. They carry tradeoffs which may make them unsuitable for one's particular requirements.
- Backend development frameworks can be classified into RPC, REST, and GraphQL frameworks.
- Some components can be implemented as either libraries or services. Each approach has its tradeoffs.
- Most communication paradigms are implemented on top of HTTP. RPC is a lower-level protocol for efficiency.
- REST is simple to learn and use. We should declare REST resources as cacheable

whenever possible.

- REST requires a separate documentation framework like OpenAPI.
- RPC is a binary protocol designed for resource optimization. It's schema modification rules also allow backward and forward compatibility.
- GraphQL allows pinpoint requests and has an integrated API documentation tool. However, it is complex and more difficult to secure.
- WebSocket is a stateful communications protocol for full-duplex communication. It has more overhead on both the client and server than other communication paradigms.

6

A typical interview flow

This chapter covers:

- Clarifying the system's requirements and discussing possible tradeoffs to optimize for the requirements.
- Drafting your system's API specification.
- Designing the data models of your system.
- Discussing other concerns like logging, monitoring, and alerting, or search.
- How to reflect on your interview experience, and evaluate the company.

In the chapters of part 1 up to this point, we discussed some of the most common concerns and topics in designing a scalable web application. To end part 1, and before we begin the detailed discussions of sample system design questions in part 2, we will discuss a few principles of system design interviews that must be followed during your 1-hour systems design interview. When you complete this book, refer to this list again. Keep these principles in mind during your interviews.

1. Clarify functional and non-functional requirements, such as QPS (queries per second) and P99 latency. Ask if the interviewer desires starting the discussion from a simple system, then scaling up or designing more features, or immediately design a scalable system.
2. Everything is a tradeoff. There is almost never any characteristic of a system that is entirely positive and without tradeoffs. Any new addition to a system to improve scalability, consistency or latency also increases complexity, cost, requires security, and requires logging, monitoring, and alerting.
3. Drive the interview. Keep the interviewer's interest strong. Discuss what she wants. Keep suggesting topics of discussion to her.
4. Be mindful of time. As just stated, there is too much to discuss in 1 hour.

5. Discuss logging, monitoring, alerting, and auditing.
6. Discuss testing and maintainability including debuggability, complexity, and security and privacy.
7. Consider and discuss graceful degradation and failure in the overall system and every component, including silent and disguised failures. Errors can be silent. Never trust anything. Don't trust external or internal systems. Don't trust your own system.
8. Draw system diagrams, flowcharts, and sequence diagrams. Use them as visual aids for your discussions.
9. The system can always be improved. There is always more to discuss.

A discussion of any system design interview question can last for many hours. You will need to focus on certain aspects by suggesting to the interviewer various directions of discussion and asking which direction to go. You have less than 1 hour to communicate or hint the full extent of your knowledge. You must possess the ability to consider and evaluate relevant details, to smoothly zoom up and down to discuss high level architecture and relationships and low-level implementation details of every component. If you forget or neglect to mention something, the interviewer will assume you don't know it. One should practice discussing system design questions with fellow engineers to improve oneself in this art. Prestigious companies interview many polished candidates, and every candidate who passes is well-drilled and speaks the language of system design fluently.

The question discussions in this section are examples of the approaches you can take to discuss various topics in a System Design interview. Many of these topics are common, so you will see some repetition between the discussions. Pay attention to the use of common industry terms, and how many of the sentences uttered within the time-limited discussion are filled with useful information.

The following list is a rough guide. A System Design discussion is dynamic, and we should not expect it to progress in the order of this list.

1. Clarify the requirements. Discuss trade-offs.
2. Draft the API specification.
3. Design the data model. Discuss possible analytics.
4. Discuss failure design, graceful degradation, monitoring, and alerting. Other topics include bottlenecks, load balancing, removing single points of failure, high availability, disaster recovery, and caching.
5. Discuss complexity and tradeoffs, maintenance and decommissioning processes and costs.

6.1 Clarify requirements and discuss trade-offs

Clarifying the requirements of the question is the first checkbox to tick off during an interview. Section Requirements describes the details and importance of discussing functional and non-functional requirements.

We end this chapter with a general guide to discussing requirements in an interview. We will go through this exercise in each question of part 2. We emphasize that you keep in mind that your particular interview may be a unique situation, and you should deviate from this guide as required by your situation.

Discuss functional requirements within 10 minutes, as that is already >=20% of the interview time. Nonetheless, attention to detail is critical. Do not write down the functional requirements 1 at a time and discuss them. You may miss certain requirements. Rather, quickly brainstorm and scribble down a list of functional requirements then discuss them. We can tell the interviewer that we want to ensure we have captured all crucial requirements, but we also wish to be mindful of time.

We can begin by spending 30 seconds or 1 minute to discuss the overall purpose of the system and how it fits into the big picture business requirements. We then discuss these details of some common functional requirements.

1. Consider users categories/roles.
    a. Who will use this system and how? Discuss and scribble down user stories. Consider various combinations of user categories, such as manual vs programmatic, or consumer vs enterprise. For example, a manual/consumer combination involves requests from our consumers, via our mobile or browser apps. A programmatic/enterprise combination involves requests from other services or companies.
    b. Technical or non-technical? Design platforms or services for developers or non-developers. Technical examples include a database service like key-value store, libraries for purposes like consistent hashing, or analytics services. Non-technical questions are typically in the form of "Design this well-known consumer app.". In such questions, we must discuss all categories of users, not just the non-technical consumers of the app.
    c. List the user roles, e.g. buyer, seller, poster, viewer, developer, manager.
    d. NUMBERS. Every functional and nonfunctional requirement must have a number. Fetch news items? How many news items? How much time? How many milliseconds/seconds/hours/days?
    e. Any communication between users, or between users and operations staff?
    f. Ask about i18n and L10n support. National or regional languages. Postal address. Price. Ask if multiple currency support is required.
2. Based on the user categories, clarify the scalability requirements. Estimate the number of daily active users, then estimate the daily or hourly request rate. For example, if a search service has 1 billion daily users each submitting 10 search requests, there are 10 billion daily requests or 420 million hourly requests.
3. Which data should be accessible to which users? Discuss the authentication and authorization roles and mechanisms. Discuss the contents of the response body of API endpoint. Next, discuss how often is data retrieved. Real-time, monthly reports, or another frequency?
4. Search. What are possible use cases that involve search?
5. Analytics is a typical requirement. Discuss possible machine learning requirements, including support for experimentation such as A/B testing or multi-armed bandit.

Refer to https://www.optimizely.com/optimization-glossary/ab-testing/ and https://www.optimizely.com/optimization-glossary/multi-armed-bandit/ for introductions to these topics.

6. Scribble down pseudocode function signatures e.g. fetchPosts(userId) to fetch posts by a certain user, and match them to the user stories. Discuss with the interviewer which requirements are needed and which are out of scope.

Always ask "Are there other user requirements?", and also brainstorm these possibilities. Do not allow the interviewer to do the thinking for you. Do not give the interviewer the impression that you want her to do the thinking for you, or want her to tell you all the requirements.

Requirements are subtle, and one often misses details even if she thinks she has clarified them. One reason software development follows agile practices is that requirements are difficult or impossible to communicate. New requirements or restrictions are constantly discovered through the development process. With experience, one learns the clarifying questions to ask.

Display your awareness that a system can be expanded to serve other functional requirements in the future, and brainstorm such possibilities.

The interviewer should not expect you to possess all domain knowledge, so you may not think of certain requirements that require certain domain knowledge. What you do need is to demonstrate your critical thinking, attention to detail, humility, and willingness to learn.

Next, discuss non-functional requirements. Refer to chapter 2 for a detailed discussion on non-functional requirements. We may need to design our system to serve the entire world population, and assume that our product has complete global market dominance. Clarify with your interviewer if we should design immediately for scalability. If not, she may be more interested in how we consider complicated functional requirements. This includes the data models we design.

After we discuss requirements, we can proceed to discuss our system design.

6.2 Draft the API specification

Based on the functional requirements, determine the data that the system's users expect to receive from and send to the system. We will generally spend less than 5 minutes to scribble down a draft of the GET, POST, PUT and DELETE endpoints, including path and query parameters. It is generally inadvisable to linger on drafting the endpoints. Inform the interviewer that there is much more to discuss within our 50 minutes, so we will not use much time here.

You should have already clarified the functional requirements before scribbling these endpoints; you are past the appropriate section of the interview to clarify functional requirements, and should not do so here unless you had missed anything.

Propose an API specification and describe how it satisfies the functional requirements. Briefly discuss and identify any functional requirements that you may have missed.

6.2.1 Common API endpoints

These are common endpoints of most systems. You can quickly go over these endpoints, and clarify that they are out of scope. It is very unlikely that you will need to discuss them in detail, but it never hurts to display that you are detail-oriented while also see the big picture.

HEALTH

GET /health - This is a test endpoint. A 4xx or 5xx response indicates the system has production issues. It may just do a simple database query, or may return health information such as disk space, statuses of various other endpoints, and application logic checks.

SIGNUP AND LOGIN (AUTHENTICATION)

An app user will typically need to signup (POST /signup) and login (POST /login) prior to submitting content to the app. OpenID Connect is a common authentication protocol, discussed in appendix C.

USER AND CONTENT MANAGEMENT

We may need endpoints to get, modify, and delete user details.

Many consumer apps provide channels for users to flag/report inappropriate content, such as content that is illegal or violates community guidelines.

6.3 Connections and processing between users and data

In section 6.2, we discussed the types of users and data, and which data should be accessible to which users. In section 6.3, we designed API endpoints for users to CRUD data. We can now draw diagrams to represent the connections between user and data, and to illustrate various system components and the data processing that occurs between them.

Phase 1

- Draw a box to represent each type of user.
- Draw a box to represent each system that serves the functional requirements.
- Draw the connections between users and systems.

Phase 2

- Break up request processing and storage.
- Different designs based on the non-functional requirements, such as real-time vs eventual consistency.
- Consider shared services.

Phase 3

- Break up the systems into components.
- Draw the connections.
- Consider Logging, Monitoring, Alerting.
- Consider Security.

Phase 4

- Summary of our system design.
- New additional requirements.
- Fault-tolerance. What can go wrong with each component? Network delays, inconsistency, no linearizability. What can we do to prevent and/or mitigate each situation, and improve the fault-tolerance of this component and the overall system?

Refer to appendix C for an overview of the *C4 model*, which is a system architecture diagram technique to decompose a system into various levels of abstraction.

6.4 Design the data model

We should discuss if we are designing the data model from scratch, or using existing databases. Sharing databases between services is commonly regarded as an antipattern, so if we are using existing databases, we should build more API endpoints designed for programmatic customers, as well as batch and/or streaming ETL pipelines from and to those other databases as required.

The following are common issues that may occur with shared databases:

- Queries from various services on the same tables may contend for resources. Certain queries, such as UPDATE on many rows, or transactions that contain other long-running queries, may lock a table for an extended period of time.
- Schema migrations are more complex. A schema migration that benefits one service may break the DAO code of other services. This means that although an engineer of may work only on that service, she needs to keep up to date with the low-level details of the business logic and perhaps even the source code of other services that she does not work on, which may be an unproductive use of both her time and the time of other engineers who made those changes and need to communicate it to her and others. More time will be spent in writing and reading documentation and presentation slides and in meetings. Various teams may take time to agree on proposed schema migrations, which may be an unproductive use of engineering time. Teams may not be able to agree on schema migrations, or may compromise on certain changes, which will introduce technical debt and decrease overall productivity.
- All services must use the same database technology e.g. MySQL, HDFS, Cassandra, Kafka, etc. A service cannot use a different database technology if there are other services that depend on this service's data. Services cannot pick the database technology that best suits their requirements.

This means that in either case, we will need to design a new schema for our service. We can use the request and response bodies of the API endpoints we discussed in the previous section as starting points to design our schema, closely mapping each body to a table's schema, and probably combine the bodies of read (GET) and write (POST and PUT) requests of the same paths to the same table.

6.4.1 Example - Adding a new service related to an existing service

If we were designing an ecommerce system, we may want a service that can retrieve business metric data, such as the total number of orders in the last 7 days. Our teams had found that without a source of truth for business metric definitions, different teams had been computing metrics differently. For example, should the total number of orders include cancelled or refunded orders? What timezone should be used for the cutoff time of "7 days ago"? Does "last 7 days" include the present day? The communication overhead between multiple teams to clarify metric definitions was costly and error-prone.

Although computing business metrics uses order data from the Orders service, we decide to form a new team to create a dedicated Metrics service, since metric definitions can be modified independently of order data.

The Metrics service will depend on the Orders service for order data. A request for a metric will be processed as follows:

1. Retrieve the metric.
2. Retrieve the related data from the Orders service.
3. Compute the metric.
4. Return the metric's value.

If both services shared the same database, the computation of a metric make SQL queries on Orders service's tables. Schema migrations become more complex. For example, if the Orders team decides that users of the Order table have been making too many large queries on it. After some analysis, the team determined that queries on recent orders are more important and require higher latency than queries on older orders. The team proposes that the Order table to contain only orders from the last year and older orders will be moved to an Archive table. The Order table can be allocated a larger number of followers/read replicas than the Archive table.

The Metrics team must understand this proposed change and change metric computation to occur on both tables. The Metrics team may object to this proposed change, so the change may not go ahead, and the organizational productivity gain from faster queries on recent order data cannot be achieved.

If the Orders team wishes to move the Order table to Cassandra to leverage the low write latency of Cassandra, while the Metrics service continues using SQL because of its simplicity and because Metrics service has a low write rate, the services can no longer share the same database.

6.4.2 Preventing concurrent user updates

There are many situations where a client application allows multiple users to edit a shared configuration. If an edit to this shared configuration is nontrivial for a user (if a user needs to spend more than a few seconds to enter some information before submitting her edit), it may be a frustrating UX if multiple users simultaneously edit this configuration, then overwrite each other's changes when they save them. Source control management prevents this for source code, but most other situations involve non-technical users, and we obviously cannot expect them to learn git.

For example, a hotel room booking service may require users to spend some time to enter their check-in and check-out dates, and their contact and payment information, then submit their booking request. We should ensure that multiple users do not overbook the room.

Another example may be configuring the contents of a push notification. For example, our company may provide a browser app for employees to configure push notifications sent to our Beigel app (refer to chapter 1). A particular push notification configuration may be owned by a team. We should ensure that multiple team members do not edit the push notification simultaneously, then overwrite each other's changes.

To prevent such situations, we can lock a configuration when it is being edited. Our service may contain a SQL table to store these configurations. We can add a timestamp column to the relevant SQL table that we can name "unix_locked" and string columns "edit_username" and "edit_email" (This schema design is not normalized, but it is usually ok in practice. Ask your interviewer if she insists on a normalized schema.), and expose a PUT endpoint that our UI can use to notify our backend when a user clicks on an edit icon or button to start editing the query string. Referring to figure 6.1, here are a series of steps that may occur when 2 users decide to edit a push notification at approximately the same time. One user can lock a configuration for a certain period (e.g. 10 minutes) and another user finds that it is locked.

1. Alice and Bob are both viewing the push notification configuration on our Notifications browser app. She decides to update the title from "Celebrate National Bagel Day!" to "20% off on National Bagel Day!". She clicks on the "edit" button. The following steps occur.
    a) The click event sends a PUT request which sends her username and email to the backend. The backend's load balancer assigns this request to a host.
    b) Alice's backend host makes 2 SQL queries one at a time. Firstly, it determines the current unix_locked time.

```
SELECT unix_locked FROM table_name WHERE config_id = {config_id}.
```

    c) The backend checks if the "edit_start" timestamp is less than 12 minutes ago. (2-minute buffer in case the countdown timer in step 2 started late, and also because hosts' clocks cannot be perfectly synchronized.) If so, it updates the row to indicate to lock the configuration. The UPDATE query sets "edit_start" to the backend's current Unix time, and overwrites the "edit_username" and "edit_email" with Alice's username and email. We need the "unix_locked" filter just in case another user has changed it in the meantime. The UPDATE query returns a boolean if that indicates if it ran successfully.

```
UPDATE table_name SET unix_locked = {new_time}, edit_username = {username}, edit_email =
       {email} WHERE config_id = {config_id} AND unix_locked = {unix_locked}
```

    d) *If the UPDATE query was successful, the backend* returns 200 success to the UI with a response body like `{"can_edit": "true"}`.
2. The UI opens a page where she can make this edit, and displays a 10-minute countdown timer. Alice erases the old title and starts to type the new title.

3. *In between the SQL queries of steps 1b and 1c*, Bob decides to edit the configuration too.
   a) He clicks on the "edit" button, triggering a PUT request, which is assigned to a different host.
   b) The first SQL query returns the same unix_locked time as in step 1b.
   c) The second SQL query is sent just after the query in step 1c. Referring to section 3.3.2, SQL DML queries are sent to the same host. This means this query cannot run until the query in step 1c completes. When the query runs, the unix_time value had changed, so the row is not updated, and the SQL service returns false to the backend. The backend returns a 200 success to the UI with a response body like `{"can_edit": "false", "edit_start": "1655836315", "edit_username": "Alice", "edit_email": "alice@beigel.com"}`.
   d) The UI computes the number of minutes Alice has left, and displays a banner notification that states "Alice (alice@beigel.com) is making an edit. Try again in 8 minutes".
4. Alice finishes her edits, and clicks on the "save" button. This triggers a PUT request to the backend, which saves her edited values, and also erases "unix_locked", "edit_start", "edit_username", and "edit_email".
5. Bob clicks on the "edit" button again, and now he can make edits. If Bob had clicked on the "edit" button at least 12 minutes after the "edit_start" value, he can also make edits. If Alice had not saved her changes before her countdown expires, the UI will display a notification to inform her that she cannot save her changes anymore.

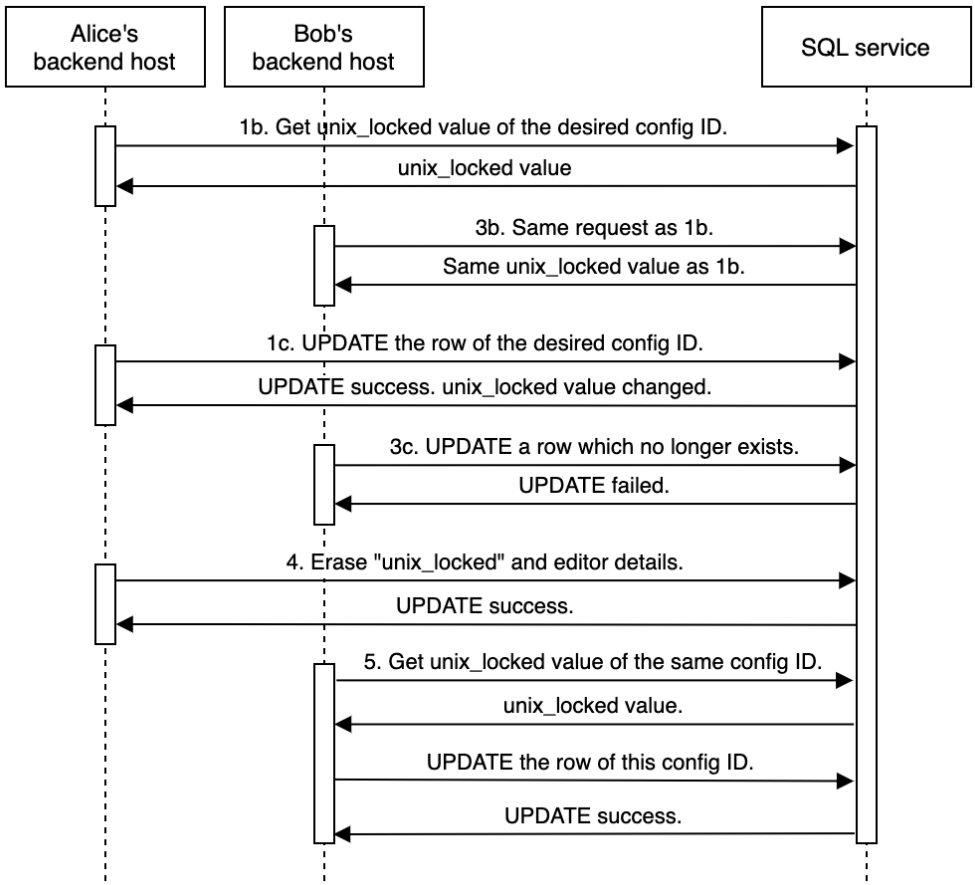

Figure 6.1 Illustration of a locking mechanism that uses SQL. Here, 2 users request to update the same SQL row that corresponds to the same configuration ID. Alice's host first gets the unix_locked timestamp value of the desired configuration ID, then sends an UPDATE query to update that row, so Alice has locked that specific configuration ID. Right after her host sent that query in step 1c, Bob's host sends an UPDATE query too, but Alice's host had changed the unix_locked value, so Bob's UPDATE query cannot run successfully, and Bob cannot lock that configuration ID.

What if Bob visits the push notification configuration page after Alice started editing the configuration? A possible UI optimization at this point is to disable the "edit" button and display the banner notification, so Bob knows he cannot make edits because Alice is doing so. To implement this optimization, we can add the 3 fields to the GET response for the push notification configuration, and the UI should process those fields and render the "edit" button as "enabled" or "disabled".

Refer to https://vladmihalcea.com/jpa-entity-version-property-hibernate/ for an overview of version tracking with Jakarta Persistence API and Hibernate.

6.5 Logging, monitoring, and alerting

There are many books on logging, monitoring, and alerting. In this chapter, we will discuss key concepts that one must mention in an interview, and dive into specific concepts that one may be expected to discuss. Never forget to mention monitoring to the interviewer.

6.5.1 The importance of monitoring

Monitoring is critical for every system to provide visibility into the customer's experience. We need to identify bugs, degradations, unexpected events, and other weaknesses in our system's ability to satisfy current and possible future functional and non-functional requirements.

Web services may fail at any time. We may categorize these failures by urgency and how quickly they need attention. High-urgency failures must be attended to immediately. Low-urgency failures may wait until we complete higher priority tasks. Our requirements and discretion determine the multiple levels of urgency that we define.

If our service is a dependency of other services, every time those services experience degradations, their teams may identify our service as a potential source of those degradations, so we need a logging and monitoring setup that will allow us to easily investigate possible degradations and answer their questions.

6.5.2 Observability

This leads us to the concept of observability. The observability of our system is a measure of how well-instrumented it is, and how easily we can find out what's going on inside it (John Arundel & Justin Domingus, Cloud Native DevOps with Kubernetes, page 272, O'Reilly Media Inc, 2019.). Without logging, metrics, and tracing, our system is opaque. We will not easily know how well a code change meant to decrease P99 of a particular endpoint by 10% works in production. If P99 decreased by much less than 10% or much more than 10%, we should be able to derive relevant insights from our instrumentation on why our predictions fell short.

Refer to Google's SRE book (https://sre.google/sre-book/monitoring-distributed-systems/#xref_monitoring_golden-signals) for a detailed discussion of the 4 golden signals of monitoring: latency, traffic, errors, and saturation.

1. Latency: We can set up alerts for latency that exceeds our service-level agreement (SLA), such as over 1 second. Our SLA may be for any individual request over 1 second, or alerts that trigger for a P99 over a sliding widow (e.g. 5 seconds, 10 seconds, 1 minute, 5 minutes).
2. Traffic: Measured in HTTP requests per second. We can set up alerts for various endpoints that trigger if there is too much traffic. We can set appropriate numbers based on the load limit determined in our load testing.
3. Errors: Set up high-urgency alerts for 4xx or 5xx response codes, that must be immediately addressed. Trigger low-urgency (or high depending on your requirements) alerts on failed audits.
4. Saturation: Depending on whether our system's constraint is CPU, memory, or I/O, we can set utilization targets that should not be exceeded. We can set up alerts that

trigger if utilization targets are reached. Another example is storage utilization. We can set up an alert that triggers if storage (due to file or database usage) may run out within hours or days.

The 3 instruments of monitoring and alerting are metrics, dashboards, and alerts. A *metric* is a variable we measure, like error count, latency, or processing time. A dashboard provides a summary view of a service's core metrics. An alert is a notification service sent to service owners in a reaction to some issue happening in the service. Metrics, dashboards, and alerts are populated by processing log data. We may provide a common browser UI to create and manage them more easily.

OS metrics like CPU utilization, memory utilization, disk utilization and network I/O can be included in our dashboard, and used to tune the hardware allocation for our service as appropriate, or detect memory leaks.

On a backend application, our backend framework may log each request by default, or provide simple annotations on the request methods to turn on logging. We can put logging statements in our application code. We can also manually log the values of certain variables within our code to help us understand how the customer's request was processed.

Scholl et al. (Boris Scholl, Trent Swanson & Peter Jausovec. Cloud Native: Using Containers, Functions, and Data to build Next-Generation Applications. O'Reilly, 2019. Page 145.) states the following general considerations in logging:

- Log entries should be structured, to be easy to parse by tools and automation.
- Each entry should contain a unique identifier to trace requests across services and share between users and developers.
- Log entries should be small, easy to read, and useful.
- Timestamps should use the same time zone and time format. A log that contains entries with different time zones and time formats is difficult to read or parse.
- Categorize log entries. Start with debug, info, and error.
- Do not log private or sensitive information like passwords or connection strings. A common term to refer to such information is Personally Identifiable Information (PII).

Logs which are common to most services include the following. Many request-level logging tools have default configurations to log these details.

- Host logging:
    - CPU and memory utilization on the hosts.
    - Network I/O.
- Request-level logging capture the details of every request.
    - Latency.
    - Who and when made the request.
    - Function name and line number.
    - Request path and query parameters, headers, and body.
    - Return status code and body (including possible error messages).

In a particular system, we may be particularly interested in certain user experiences, such as errors. We can place log statements within our application and set up customized metrics,

dashboards, and alerts that focus on these user experiences. For example, to focus on 5xx errors due to application bugs, we can create metrics, dashboards and alerts that process certain details like request parameters and return status code and error messages if any.

We should also log events to monitor how well our system satisfies our unique functional and non-functional requirements. For example, if we build a cache, we want to log cache faults, hits, and misses. Metrics should include the counts of faults, hits, and misses.

In enterprise systems, we may wish to give users some access to monitoring, or even build monitoring tools specifically for users e.g. customers can create dashboards to track the state of their requests, and filter and aggregate metrics and alerts by categories such as URL paths.

We should also discuss how to address possible silent failures. These may be due to bugs in our application code or dependencies such as libraries and other services that allow the response code to be 2xx when it should be 4xx or 5xx, or may indicate your service requires logging and monitoring improvements.

Besides logging, monitoring, and alerting on individual requests, we may also create batch and/streaming audit jobs to validate our system's data. This is akin to monitoring our system's data integrity. We can create alerts that trigger if the results of the job indicate failed validations. Such a system is discussed in chapter 14.

6.5.3 Responding to alerts

A team that develops and maintains a service may typically consist of a few engineers. This team may set up an on-call schedule for the service's high-urgency alerts. An on-call engineer may not be intimately familiar with the cause of a particular alert, so we should prepare a runbook that contains a list of the alerts, possible causes, and procedures to find and remediate the cause.

As we prepare the runbook, if we find that certain runbook instructions consist of a series of commands that can easily be copy-pasted to solve the issue e.g. restarting a host, these steps should be automated in the application, along with logging that these steps were run (Mike Julian, Practical Monitoring, chapter 3, O'Reilly Media Inc, 2017). Failure to implement automated failure recovery when possible is runbook abuse. If certain runbook instructions consist of running commands to view certain metrics, we should display these metrics on our dashboard.

A company may have a Site Reliability Engineer (SRE) team, which consists of engineers who develop tools and processes to ensure high reliability of critical services and often on-call for these critical services. If our service obtains SRE coverage, a build of our service may have to satisfy the SRE team's criteria before it can be deployed. This criteria typically consists of high unit test coverage, a functional test suite that passes SRE review, and a well-written runbook that has good coverage and description of possible issues and has been vetted by the SRE team.

After the outage is resolved, we should author a postmortem that identifies what went wrong, why, and how the team will ensure it does not recur. Postmortems should be blameless, or employees may attempt to downplay or hide issues instead of addressing them.

Based on identifying patterns in the actions that are taken to resolve the issue, we can identify ways to automate mitigation of these issues, introducing self-healing characteristics to the system.

6.5.4 Application-level logging tools

The open-source ELK (Elasticsearch, Logstash, Kinaba) suite and the paid-service Splunk are common application-level logging tools. Logstash is used to collect and manage logs. Elasticsearch is a search engine, useful for storage, indexing, and searching through logs. Kibana is for visualization and dashboarding of logs, using Elasticsearch as a data source and for users to search logs.

In this book, whenever we state that we are logging any event, it is understood that we log the event to a common ELK service used for logging by other services in our organization.

There are numerous monitoring tools, which may be proprietary or FOSS (Free and open-source software). We will briefly discuss a few of these tools, but an exhaustive list, detailed discussion, or comparison are outside the scope of this book.

These tools differ in characteristics such as:

- Features. Various tools may offer all or a subset of logging, monitoring, alerting, and dashboarding.
- Support for various operating systems and other types of equipment besides servers, such as load balancers, switches, modems, routers, or network cards etc.
- Resource consumption.
- The ease of finding engineers familiar with the system.
- Developer support, such as the frequency of updates.

They also differ in subjective characteristics like:

- Learning curve.
- Difficulty of manual configuration.
- Ease of integration with other software and services.
- Number and severity of bugs.
- UX. Some of the tools have browser or desktop UI clients, and various users may prefer the UX of one UI over another.

FOSS monitoring tools include the following.

- Prometheus + Grafana - Prometheus for monitoring, Grafana for visualization and dashboarding.
- Sensu – A monitoring system that uses Redis to store data. We can configure Sensu to send alerts to a third-party alerting service.
- Nagios – A monitoring and alerting system.
- Zabbix – A monitoring system that includes a monitoring dashboard tool.

Proprietary tools include Splunk, Datadog, and New Relic.

A time series database (TSDB) is a system that is optimized for storing and serving time series, such as the continuous writes that happen with logging time series data. Examples include the following. Most queries may be made on recent data, so old data will be less valuable and we can save storage by configuring down sampling on TSDB. This rolls up old data by computing averages over defined intervals. Only these averages are saved, and the original data is deleted, so less storage is used. Data retention period and resolution depend on our requirements and budget.

To further reduce the cost of storing old data, we can compress it, or use cheap storage medium like tape or optical disks. Refer to https://www.zdnet.com/article/could-the-tech-beneath-amazons-glacier-revolutionise-data-storage/ or https://arstechnica.com/information-technology/2015/11/to-go-green-facebook-puts-petabytes-of-cat-pics-on-ice-and-likes-windfarming/ for examples of custom setups such as hard disk storage servers that slow down or stop when not in use.

- Graphite – Commonly used to log OS metrics (though it can monitor other setups like websites and applications), which are visualized with the Grafana web application.
- Prometheus – Also typically visualized with Grafana.
- OpenTSDB – A distributed, scalable TSDB that uses HBase.
- InfluxDB – An open-source TSDB written in Go.

Prometheus is an open-source monitoring system built around a time series database. Prometheus pulls from target HTTP endpoints to request metrics, and a Pushgateway pushes alerts to Alertmanager, which we can configure to push to various channels such as email and PagerDuty. We can use Prometheus query language (PromQL) to explore metrics and draw graphs.

Nagios is a proprietary legacy IT infrastructure monitoring tool that focuses on server, network, and application monitoring. It has hundreds of 3rd party plugins, web interfaces, and advanced visualization dashboarding tools.

6.5.5 Streaming and batch audit of data quality

Data Quality is an informal term that refers to ensuring that data represents the real-world construct to which it refers, and can be used for its intended purposes. For example, if a particular table that is updated by an ETL job is missing some rows that were produced by that job, the data quality is poor.

Database tables can be continuously and/or periodically audited to detect data quality issues. We can implement such auditing by defining streaming and batch ETL jobs to validate recently added and modified data.

This is particularly useful to detect silent errors, which are errors which were undetected by earlier validation checks, such as validation checks which occur during processing of a service request.

We can extend this concept to a hypothetical shared service for database batch auditing, discussed in chapter 14.

6.5.6 Anomaly detection to detect data anomalies

Anomaly detection is a machine learning concept to detect unusual datapoints. A full description of machine learning concepts is outside the scope of this book. This section briefly describes anomaly detection to detect unusual datapoints. This is useful both to ensure data quality and for deriving analytical insights, as an unusual rise or fall of a particular metric can indicate issues with data processing or changing market conditions.

In its most basic form, anomaly detection consists of feeding a continuous stream of data into an anomaly detection algorithm. After it processes a defined number of datapoints, referred to in machine learning as the training set, the anomaly detection algorithm develops

a statistical model. The model's purpose is to accept a datapoint and assign a probability that the datapoint is anomalous. We can validate that this model works by using it on a set of datapoints called the validation set, where each datapoint has been manually-labelled as normal or anomalous. Finally, we can quantify accuracy characteristics of the model by testing it on another manually-labeled set, called the test set.

Many parameters are manually tunable, such as which machine learning models are used, the number of datapoints in each of the 3 sets, and the model's parameters to adjust characteristics such as precision vs recall. Machine learning concepts such as precision and recall are outside the scope of this book.

In practice, this approach to detecting data anomalies is complex and costly to implement, maintain, and use. It is reserved for critical datasets.

6.5.7 Silent errors and auditing

Silent errors may occur due to bugs where an endpoint may return status code 200 even though errors occurred. We can write batch ETL jobs to audit recent changes to our databases, and raise alerts on failed audits. Further details are provided in chapter 14.

6.5.8 Further reading on observability

- Michael Hausenblas, Cloud Observability in Action, Manning Publications, 2023. - A guide to apply observability practices to cloud-based serverless and Kubernetes environments.
- https://www.manning.com/liveproject/configure-observability - A hands on course in implementing a service observability-related features.
- Mike Julian, Practical Monitoring, O'Reilly Media Inc, 2017. – A dedicated book on observability best practices, incident response, and antipatterns.
- Boris Scholl, Trent Swanson & Peter Jausovec. Cloud Native: Using Containers, Functions, and Data to build Next-Generation Applications. O'Reilly, 2019. – Emphasizes that observability is integral to cloud native applications.
- John Arundel & Justin Domingus, Cloud Native DevOps with Kubernetes, chapters 15-16, O'Reilly Media Inc, 2019. – These chapters discuss observability, monitoring, and metrics in cloud native applications.

6.6 Search bar

Search is a common feature of many applications. Most frontend applications provide users with search bars to rapidly find their desired data. The data can be indexed in an Elasticsearch cluster.

6.6.1 Search bar

A search bar is a common feature in many apps. It can be just a single search bar, or may contain other frontend components for filtering. Figure 6.2 is an example of a search bar.

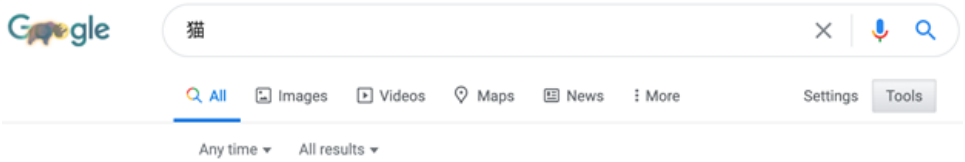

Figure 6.2 Google search bar with dropdown menus for filtering results. Image from Google.

Common techniques of implementing search are:

1. Search on a SQL database with the LIKE operator and pattern matching. A query may resemble something like "SELECT <column> FROM <table> WHERE Lower(<column>) LIKE "%Lower(<search_term>)%"".
2. Use a library such as match-sorter (https://github.com/kentcdodds/match-sorter), which is a JavaScript library that accepts search terms and does matching and sorting on the records. Such a solution needs to be separately implemented on each client application. This is a suitable and technically simple solution for up to a few GB of text data i.e up to millions of records. A web application usually downloads its data from a backend, and this data is unlikely to be more than a few MB, or the application will not be scalable to millions of users. A mobile application may store data locally, so it is theoretically possible to have GBs of data, but it is practically impossible for a person to enter GBs of text data into her phone.
3. Use a search engine such as Elasticsearch. This solution is scalable and can handle PBs of data.

The first technique has numerous limitations and should only be used as a quick temporary implementation that will soon be either discarded or changed to a proper search engine. Disadvantages include:

- Difficult to customize the search query.
- No sophisticated features like boosting, weight, fuzzy search, or text preprocessing such as stemming or tokenization.

This discussion assumes that individual records are small, e.g. they are text records, not video records. For video records, the indexing and search operations are not directly on the video data, but on accompanying text metadata. The implementation of indexing and search in search engines is outside the scope of this book.

We will reference these techniques in the question discussions in part 2, paying much more attention using Elasticsearch.

6.6.2 Elasticsearch

An organization can have a shared Elasticsearch cluster to serve the search requirements of many of its services. In this section, we first describe a basic Elasticsearch full-text search query, then the basic steps for adding Elasticsearch to your service given an existing Elasticsearch cluster. We will not discuss Elasticsearch cluster setup in this book, or describe

Elasticsearch concepts and terminology in detail. We will use our Beigel app (from chapter 1) for our examples.

6.6.3 Search bar implementation

To provide basic full-text search with fuzzy matching, we can attach our search bar to a GET endpoint which forwards the query to our Elasticsearch service. An Elasticsearch query is done against an Elasticsearch index (akin to a database in a relational database). If the GET query returns a 2xx response with a list of search results, the frontend loads a results page that displays the list.

For example, if our Beigel app provides a search bar, and a user may search for the term "sesame", the Elasticsearch request may resemble either of the following.

The search term may be contained in a query parameter, which allows exact matches only.

```
GET /beigel-index/_search?q=sesame
```

We can also use a JSON request body, which allows us to use the full Elasticsearch DSL, which is outside the scope of this book.

```
GET /beigel-index/_search
{
  "query": {z6
    "match": {
      "query": "sesame",
      "fuzziness": "AUTO"
    }
  }
}
```

"fuzziness": "AUTO" is to allow fuzzy (approximate) matching, which has many use cases, such as if the search term or search results contain misspellings.

The results are returned as a JSON array of hits sorted by decreasing relevance, such as the following example. Our backend can pass the results back to the frontend, which can parse and present them to the user.

6.6.4 Elasticsearch index and ingestion

Creating an Elasticsearch index consists of ingesting the documents that should be searched when the user submits a search query from a search bar, followed by the indexing operation.

We can keep the index updated with periodic or event-triggered indexing or delete requests using the Bulk API.

To change the index's mapping (refer to table 10.2), one way is to create a new index and drop the old one. Another way is to use Elasticsearch's reindexing operation, but this is expensive because the internal Lucene commit operation occurs synchronously after every write request (https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-translog.html#index-modules-translog).

Creating an Elasticsearch index requires all data that you wish to search on to be stored in the Elasticsearch document store, which increases our overall storage requirements. There are various optimizations that involve sending only a subset of data to be indexed.

Table 6.1 is an approximate mapping between SQL and Elasticsearch terminology.

Table 6.1 Approximate mapping between SQL and Elasticsearch terminology. There are differences between the mapped terms and this table should not to be taken at face value. This mapping is meant for an Elasticsearch beginner with SQL experience to use as a starting point for further learning.

SQL	Elasticsearch
Database	Index
Partition	Shard
Table	Type (deprecated without replacement)
Column	Field
Row	Document
Schema	Mapping
Index	Everything is indexed.

6.6.5 Using Elasticsearch in place of SQL

Elasticsearch can be used like SQL. Elasticsearch has the concept of query context vs filter context (https://www.elastic.co/guide/en/elasticsearch/reference/current/query-filter-context.html). From the documentation, in a filter context, a query clause answers the question "Does this document match this query clause?". The answer is a simple yes or no; no scores are calculated. In a query context, a query clause answers the question "How well does this document match this query clause?". The query clause determined whether the document matches and calculates a relevance score. Essentially, query context is analogous to SQL queries, while filter context is analogous to search.

Using Elasticsearch in place of SQL will allow both searching and querying, eliminate duplicate storage requirements, and eliminate the maintenance overhead of the SQL database. I have seen services that use only Elasticsearch for data storage.

However, Elasticsearch is often used to complement relational databases instead of replacing them. It is a schemaless database, and does not have the concept of normalization or relations between tables such as primary key and foreign key.

Moreover, the Elasticsearch Query Language (EQL) is a JSON-based language, and it is verbose and presents a learning curve. SQL is familiar to non-developers such as data analysts and to non-technical personnel. Non-technical users can easily learn basic SQL within a day.

Elasticsearch SQL was introduced in June 2018 in the release of Elasticsearch 6.3.0 (https://www.elastic.co/blog/an-introduction-to-elasticsearch-sql-with-practical-examples-part-1 and https://www.elastic.co/what-is/elasticsearch-sql). It supports all common filter and aggregation operations (https://www.elastic.co/guide/en/elasticsearch/reference/current/sql-functions.html). This is a promising development. SQL's dominance is well-established, but in the coming years it is possible that more services will use Elasticsearch for all their data storage as well as search.

6.6.6 Implementing search in our services

Mentioning search during the user stories and functional requirements discussion of the interview demonstrates customer focus. Unless the question is to design a search engine, it is unlikely that we will describe implementing search beyond creating Elasticsearch indexes, ingesting and indexing, making search queries, and processing results. Most of the question discussions of part 2 discuss search in this manner.

6.6.7 Further reading on search

Here are more resources on Elasticsearch and indexing:

- https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html - The official Elasticsearch guide.
- Madhusudhan Konda, Elasticsearch in Action (Second Edition), Manning Publications, 2023. – A hands-on guide to developing fully functional search engines with Elasticsearch and Kibana.
- https://www.manning.com/livevideo/elasticsearch-7-and-elastic-stack - A course on Elasticsearch 7 and Elastic Stack.
- https://www.manning.com/liveproject/centralized-logging-in-the-cloud-with-elasticsearch-and-kibana - A hands-on course on logging in the cloud with Elasticsearch and Kibana
- https://stackoverflow.com/questions/33858542/how-to-really-reindex-data-in-elasticsearch - This is a good alternative to the official Elasticsearch guide regarding how to update an Elasticsearch index.
- https://developers.soundcloud.com/blog/how-to-reindex-1-billion-documents-in-1-hour-at-soundcloud - A case study of a large reindexing operation.

6.7 Other discussions

When we reach a point in our discussion where our system design satisfies our requirements, we can discuss other topics. This section briefly discusses a few possible topics of further discussion.

6.7.1 Maintaining and extending the application

We discuss the requirements at the beginning of the interview, and have system design for them. We can continue to improve our design to better serve our requirements.

We can also expand the discussion to other possible requirements. Anyone who works in the tech industry knows that application development is never complete. There are always new and competing requirements. Users submit feedback they want developed or changed. We monitor the traffic and request contents of our API endpoints to make scaling and development decisions. There is constant discussion on what features to develop, maintain, deprecate, and decommission. We can discuss these topics:

- Maintenance may already be discussed during the interview. Which system components rely on technology (such as software packages) which are developed fastest and require the most maintenance work? How will we handle upgrades that introduce breaking changes in any component?
- Features we may need to develop in the future, and the system design.
- Features which may not be needed in the future, and how to gracefully deprecate and decommission them, what is an adequate level of user support to provide during this process and how to best provide it.

6.7.2 Supporting other types of users

We can extend the service to support other types of users. If we focused on either consumer or enterprise, manual or programmatic, we may discuss extending the system to support the other user categories. We can discuss extending the current services, building new services, and the tradeoffs of both approaches.

6.7.3 Alternative architectural decisions

During the earlier part of the interview, we should have discussed alternative architectural decisions. We can revisit them in greater detail.

6.7.4 Usability and feedback

Usability is a measure of how well our users can use our system to effectively and efficiently achieve the desired goals. It is an assessment of how easy our user interface is to use.

We can define usability metrics, log the required data, and implement a batch ETL job to periodically compute these metrics and update a dashboard that displays them.

Usability metrics can be defined based on how we intend our users to use our system.

For example, if we made a search engine, we will like our users to find their desired result quickly. One possible metric can be the average index of the result list that a user clicks on. We will like the results to be ordered in decreasing relevance, and we assume that a low average chosen index indicates that a user found her desired result close to the top of the list.

Another example metric is the amount of help users need from our support department when they use our application. It is ideal for our application to be self-service i.e. a user can perform her desired tasks entirely within the application, without having to ask for help. If our application has a helpdesk, this can be measured by number of helpdesk tickets created per day or week. A high number of helpdesk tickets indicates that our application is not self-service.

Usability can also be measured with user surveys. A common usability survey metric is Net Promoter Score (NPS). NPS is defined as the percentage of customers rating their likelihood to recommend our application to a friend or colleague as 9 or 10 minus the percentage rating this at 6 or below on a scale of 0 to 1083.

We can create UI components within our application for users to submit feedback. For example, our web UI may have a HTML link or form for users to email feedback and comments. If we do not wish to use email, because of reasons such as possible spam, we can create an API endpoint to submit feedback and attach our form submission to it.

Good logging will aid the reproducibility of bugs, by helping us match the user's feedback with her logged activities.

6.7.5 Edge cases and new constraints

Near the end of the interview, the interviewer may introduce edge cases and new constraints, limited only to the imagination. They may consist of new functional requirements, or pushing certain nonfunctional requirements to the extreme. You may have anticipated some of these edge cases during requirements planning. We can discuss if we can make tradeoffs to fulfill them, or can redesign our architecture to support our current requirements as well as these new requirements. Here are some examples.

New functional requirements:

- We designed a sales service that supports credit card payments. What if our payments system needs to be customizable to support different credit card payments requirements in each country? What if we also need to support other payment types like store credit? What if we need to support coupon codes?
- We designed a text search service. How may we extend it to images, audio, and video?
- We designed a hotel room booking service. What if the user needs to change rooms? We'll need to find an available room for her, perhaps in another hotel.
- What if we decide to add social networking features to our news feed recommendation service?

Scalability and performance:

- What if a user has 1 million followers or 1 million recipients of her messages? Can we accept a long P99 message delivery time? Or do we need to design for better performance?
- What if we need to perform an accurate audit of our sales data for the last 10 years?

Latency and throughput:

- What if our P99 message delivery time needs to be within 500 milliseconds?
- If we designed a video streaming service that does not accommodate live streaming, how may we modify the design to support live streaming? How may we support simultaneously streaming of a million high resolution videos across 10 billion devices?

Availability and fault-tolerance:

- We designed a cache that didn't require high availability, since all our data is also in the database. What if we want high availability, at least for certain data?
- What if our sales service was used for high frequency trading? How may we increase its availability?
- How may each component in your system fail? How may we prevent or mitigate these failures?

Cost:

- We may have made expensive design decisions to support low latency and high performance. What may we tradeoff for lower costs?
- How may we gracefully decommission our service if required?
- Did we consider portability? How may we move our application to the cloud (or off the cloud)? What are the tradeoffs in making our application portable? (Higher costs and complexity.) Consider MinIO84 for portable object storage.

Every question in part 2 of this book ends with a list of topics for further discussion.

6.7.6 Cloud Native concepts

We may discuss addressing the nonfunctional requirements via cloud native concepts like microservices, service mesh and sidecar for shared services (Istio), containerization (Docker), orchestration (Kubernetes), automation (Skaffold, Jenkins), and infrastructure as code (Terraform, Helm). A detailed discussion of these topics is outside the scope of this book. Interested readers can easily find dedicated books or online materials.

6.8 Post-interview reflection and assessment

You will improve your interview performance as you go through more interviews. To help you learn as much as possible from each interview, you should write a post-interview reflection as soon as possible after each interview. Then you will have the best possible written record of your interview, and you can write your honest critical assessment of your interview performance.

6.8.1 Write your reflection as soon as possible after the interview

To help with this process, at the end of an interview, politely ask for permission to take photos of your diagrams, but do not persist if permission is denied. Carry a pen and a paper notebook with you in your bag. If you cannot take photos, use the earliest possible opportunity to redraw your diagrams from memory into your notebook. Next, scribble down as much detail you can recall.

You should write your reflection as soon as possible after the interview, when you can still remember many details. You may be tired after your interview, but it is counterproductive to relax and possibly forget information valuable to improving your future interview performances. Immediately go home or back to your hotel room and write your reflection, so you may write it in a comfortable and distraction-free environment.

Your reflection may have the following outline:

1. Header:
    a. The company and group the interview was for.
    b. The interview's date.
    c. Your interviewer's name and job title.
    d. The question that the interviewer asked.
    e. Were diagrams from your photos or redrawn from memory?

2. Divide the interview into approximate 10-minute sections. Place your diagrams within the sections when you started drawing them. Your photos may contain multiple diagrams, so you may need to split your photos into their separate diagrams.
3. Fill in as much detail of the interview as you can recall into the sections.
    a. What you said.
    b. What you drew.
    c. What the interviewer said.
4. Write your personal assessment and reflections, such as to the following. Your assessments may be imprecise, so you should aim to improve them with practice.
    a. Try to find the interviewer's resume or LinkedIn profile.
    b. Put yourself in the interviewer's shoes. Why do you think the interviewer chose that system design question? What did you think the interviewer expected?
    c. The interviewer's expressions and body language. Did the interviewer seem satisfied or unsatisfied with your statements and your drawings? Which were they? Did the interviewer interrupt or was eager to discuss any statements you made? What statements were they?
5. In the coming days, if you happen to recall more details, append them to these as separate sections, so you do not accidentally introduce inaccuracies into your original reflection.

While you are writing your reflection, ask yourself questions such as the following:

- What questions did the interviewer ask you about your design?
- Did the interviewer question your statements, such as by asking "Are you sure?"?
- What did the interviewer not tell you? Do you believe this was done on purpose to see if you will mention it, or might the interviewer have lacked this knowledge?

When you have finished your reflection and recollections, take a well-deserved break.

6.8.2 Writing your assessment

Writing your assessment serves to help you learn as much as possible about your areas of proficiency and deficiency that you demonstrated at the interview. Begin writing your assessment within a few days of the interview.

Before you start researching the question that you were asked, you may first write down any additional thoughts on the following. The purpose is for you to be aware of the current limit of your knowledge, and how polished you are at a systems design interview.

6.8.3 Details you didn't mention

It is impossible to comprehensively discuss a system within the 50 minutes. You choose which details to mention within that time. Based on your current knowledge i.e. before you begin your research, what other details do you think you could you have added? Why didn't you mention them during the interview?

Did you consciously choose not to discuss them? Why? Did you think those details were irrelevant, too low level, or were there other reasons that you decided to use the interview time to discuss other details?

Was it due to insufficient time? How could you have managed the interview time better so you had time to discuss it?

Were you unfamiliar with the material? Now you are clearly aware of this shortcoming. Study the material so you can describe it better.

Were you tired? Was it due to lack of sleep? Should you have rested more the day before instead of cramming too much? Was it from the interviews before this one? Should you have requested a short break before the interview? Perhaps the aroma of a cup of coffee on the interview table will improve your alertness?

Were you nervous? Were you intimidated by the interviewer or other aspects of the situation? Look up the numerous online resources on how to keep calm.

Were you burdened by the weight of expectations of yourself or others? Remember to keep things in perspective. There are numerous good companies. Or you may be lucky and enter a company that wasn't prestigious, but has excellent business performance in the future so your experience and equity become valuable. You know that you are humble and determined to keep learning every day, and no matter what, this will be one of many experiences which you are determined to learn as much from as possible to improve your performance in the many interviews to come.

Which details which were probably incorrect? This indicates concepts that you are unfamiliar with. Do your research, and learn these concepts better.

Now, you should find resources on the question that was asked. You may search in books and in online resources such as the following.

- Google.
- Websites such as http://highscalability.com/.
- YouTube videos.

As emphasized throughout this book, there are many possible approaches to a system design question. The materials you find will share similarities and also have numerous differences from each other. Compare your reflection to the materials that you found. Examine how each of those resources did the following compared to you:

- Clarifying the question. Did you ask intelligent questions? What points did you miss?
- Diagrams. Did the materials contain understandable flow charts? Compare the high-level architecture diagrams and low-level component design diagrams with your own.
- How well does their high-level architecture address the requirements? What tradeoffs were made? Do you think the tradeoffs were too costly? What technologies were chosen and why?
- Communication proficiency.
    - How much of the material did you understand the first time you read or watched it?
    - What did you not understand? Was it due to your lack of knowledge, or was the presentation unclear? What can be changed such that you will understand it the

first time? Answering these questions improves your ability to clearly and concisely communicate complex and intricate ideas.

You can always add on more material to your assessment anytime in the future. Even months after the interview, you may have new insights in all manner of topics ranging from areas of improvement to alternative approaches you could have suggested, and you can add these insights to your assessment then. Extract as much value as possible from your interview experiences.

You can and should discuss the question with others, but never disclose the company where you were asked this question. Respect the privacy of your interviewers and the integrity of the interview process. We are all ethically and professionally obliged to maintain a level playing field so companies can hire on merit, and we can work and learn from other competent engineers. Industry productivity and compensation will benefit from all of us doing our part.

6.8.4 Interview feedback

Ask for interview feedback. You may not receive much feedback if the company has a policy of not providing specific feedback, but it never hurts to ask.

The company may request feedback by email or over the phone. You should provide interview feedback if asked. Remind yourself that even though there will be no impact on the hiring decision, you can help the interviewers as a fellow engineer.

6.9 Interviewing the company

In this book, we have been focused on how to handle a Systems Design Interview as the candidate. This section discusses some questions that you as the candidate may wish to ask, to decide if this company is where you wish to invest the next few years of your finite life.

The interview process goes both ways. The company wants to understand your experience, expertise, and suitability for the role, to fill the role with the best candidate they can find. You will spend at least a few years of your life at this company, so you must work at the company with the best people and development practices and philosophy that you can find, that will allow you will develop your engineering skills as much as possible.

Here are some questions you may ask to estimate how you can develop your engineering skills

Before the interview, read the company's engineering blog to understand more about the following. If there are too many articles, read the top 10 most popular ones and the ones which are most relevant to your position. For each article about a tool, understand the following:

1. What is this tool?
2. Who uses it?
3. What does it do? How does it do these things? How does it do certain things similarly or differently from other similar tools? What can it do that other tools cannot? How does it do these things? What can't it do that other tools can?

Consider writing down at least 2 questions about each article. Before your interview, look through your questions and plan which ones to ask during the interview.

Some points to understand about the company include the following:

- The company's technology stack in general.
- The data tools and infrastructure the company uses.
- Which tools were bought, and which were developed? How are these decisions made?
- Which tools are open source?
- What other open-source contributions has the company made?
- The history and development of various engineering projects.
- The quantity and breakdown of engineering resources the projects consumed. The VP and director overseeing the project, and the composition, seniority, expertise, and experience of the engineering managers, project managers, and engineers (frontend, backend, data engineers and scientists, mobile, security etc.).
- The status of the tools. How well did the tools anticipate and address their users' requirements? What are the best experiences and pain points with the company's tools, as reflected in frequent feedback? Which ones were abandoned, and why? How do these tools stack up to competitors, to the state of the art?
- What has the company or the relevant teams within the company done to address these points?
- What are the experiences of engineers with the company's CI/CD tools? How often do engineers run into issues with CI/CD? Are there incidents where CI builds succeed but CD deployments fail? How much time do they spend to troubleshoot these issues? How many messages were sent to the relevant helpdesk channels in the last month, divided by the number of engineers?
- What projects are planned, and what needs do they fulfill? What is the engineering department's strategic vision?
- What were the organizational-wide migrations in the last 2 years? Examples of migrations:
    - Shift services from bare metal to a cloud vendor or between cloud vendors.
    - Stop using certain tools e.g. a database like Cassandra, a particular monitoring solution.
- Have there been sudden U-turns, e.g. migrating from bare metal to Google Cloud Platform followed by migrating to AWS just a year later? How much were these U-turns motivated by unpredictable versus overlooked or political factors?
- Have there been any security breaches in the history of the company, how serious were they, and what is the risk of future breaches?
- Have any systems found to violate any laws or regulations?
- The overall level of the company's engineering competence.
- The management track record, both in the current and previous roles.

Be especially critical of your prospective manager's technical background. As an engineer or engineering manager, never accept a non-technical engineering manager, especially a charismatic one. An engineering manager who cannot critically evaluate engineering work, cannot make good decisions on sweeping changes in engineering processes or lead the execution of such changes e.g. cloud native processes like moving from manual deployments to continuous deployment, and may prioritize fast feature development at the cost of technical debt that she cannot recognize. Such a manager has typically been in the same company (or

an acquired company) for many years, has established a political foothold that enabled her to get her position, and is unable to get a similar position in other companies that have competent engineering organizations. Large companies that breed the growth of such managers have or are about to be disrupted by emerging startups. Working at such companies may be more lucrative in the short term than alternatives currently available to you, but may set back your long-term growth as an engineer by years, and perhaps even be financially worse for you as companies that you rejected for short-term financial gain end up performing better in the market, with higher growth in the valuation of your equity. Proceed at your own peril.

Overall, what can I learn and cannot learn from this company in the next 4 years? When you have your offers, you can go over this information that you have collected, and make a thoughtful decision.

https://blog.pragmaticengineer.com/reverse-interviewing/ is an article on interviewing your prospective manager and team.

6.10 Summary

- Everything is a tradeoff. Low latency and high availability increase cost and complexity. Every improvement in certain aspects is a regression in others.
- Be mindful of time. Clarify the important points of the discussion and focus on them.
- Start the discussion by clarifying the system's requirements, and discuss possible tradeoffs in the system's capabilities to optimize for the requirements.
- The next step is to draft the API specification to satisfy the functional requirements.
- Draw the connections between users and data. What data do users read and write to the system, and how is data modified as it moves between system components?
- Discuss other concerns like logging, monitoring, alerting, search, and others that come up in the discussion.
- After the interview, write your self-assessment to evaluate your performance, and learn your areas of strength and weakness. It is a useful future reference to track your improvement.
- Know what you want to achieve in the next few years, and interview the company to determine if it is where you wish to invest your career.
- Logging, monitoring, and alerting are critical to alert us to unexpected events quickly and provide useful information to resolve them.
- Use the 4 golden signals and 3 instruments to quantify your service's observability.
- Log entries should be easy to parse, small, useful, categorized, have standardized time formats, and not contain private information.
- Follow the best practices of responding to alerts, such as runbooks that are useful and easy to follow, and continuously refine your runbook and approach based on the common patterns you identify.

7 Craigslist

This chapter covers:

- **Designing an application for where there are 2 distinct types of users.**
- **Considering geolocation routing to partition users by geography.**
- **Designing read-heavy vs write-heavy applications.**
- **Minor deviations during the interview.**

We want to design a web application for classifieds posts. Craigslist is an example of a typical web application that may have over a billion users. It is partitioned by geography. We can discuss the overall system, which includes browser and mobile apps, a stateless backend, simple storage requirements, and analytics. More use cases and constraints can be added for an open-ended discussion. This chapter is also unique in that it is the only system in this book where we discuss a monolith architecture as a possible system design.

7.1 User stories and requirements

Let's discuss the user stories for Craigslist. We distinguish 2 primary user types - viewer and poster.

A poster should be able to create and delete a post, and search her posts as she may have many, especially if they were programmatically generated. This post should contain the following information:

- Title.
- Some paragraphs of description.
- Price. Assume a single currency. Ignore currency conversions.
- Location.
- Up to 10 photos of 1 MB each.
- Video, though this may be added to a later iteration of our application.

A poster can renew her post every 7 days. She will receive an email notification with a click-through to renew her post.

A viewer should be able to:

1. View all posts or search on posts within any city made in the last 7 days. View a list of results, possibly as an endless scroll.
2. Apply filters on the results.
3. Click on an individual post to view its details.
4. Contact the poster, such as by email.
5. Report fraud and misleading posts e.g. a possible clickbait technique is to state a low price on the post but a higher price in the description.

The non-functional requirements are as follows.

- Scalable - Up to 10 million users in a single city.
- High availability - 99.9% uptime.
- High performance - Viewers should be able to view posts within seconds of creation.

Most of the required storage will be for Craiglist posts. The amount of required storage is low.

- We may show a Craigslist user only the posts in her local area. This means that a data center serving any individual user only needs to store a fraction of all the posts (though it may also back up posts from other data centers).
- Posts are manually (not programmatically) created, so storage growth will be slow.
- We do not handle any programmatically generated data.
- A post may be automatically deleted after 1 week.

A low storage requirement means that all the data can fit into a single host, so we do not require distributed storage solutions. Let's assume an average post contains 1000 letters or 1 KB of text. If we assume that a big city has 10 million people and 10% of them are posters creating an average of 10 posts/day i.e. 10 GB/day, our SQL database can easily store months of posts.

7.2 API

Let's scribble down some API endpoints. (In an interview, we have no time to write down a formal API specification such as in OpenAPI format or GraphQL schema, so we can tell the interviewer that we can use a formal specification to define our API, but in the interest of time will use rough scribbles during the interview. We will not mention this again in the rest of the book.)

- GET and DELETE /post/{id}
- POST and PUT /post
- POST /contact
- POST /report
- DELETE /old_posts
- POST /signup (We need not discuss user account management.)
- POST /login

- DELETE /user
- GET /health (Usually automatically generated by the framework. Our implementation can be as simple as making a small GET request and verifying it returns 200, or can be detailed and include statistics like P99 and availability of various endpoints.)

There are various filters, which may vary by the product category. For simplicity, we assume a fixed set of filters. Filters can be implemented both on the frontend and backend.

- neighborhood - enum
- minimum price
- maximum price
- item condition - enum. Values include NEW, EXCELLENT, GOOD, ACCEPTABLE.

The GET /post endpoint can have a "search" query parameter to search on posts.

7.3 SQL database schema

We can design the following SQL schema for our Craigslist user and post data.

- User - id PRIMARY KEY, first_name text, last_name text, signup_ts integer
- Post - id PRIMARY KEY, ts integer, poster_id, location_id int, title, description, price integer, condition
- Images - id PRIMARY KEY, ts integer, post_id, image_address
- Location - id PRIMARY KEY, country_code, state, city, street_number, street_name, zip_code
- Report - id PRIMARY KEY, ts integer, post_id, user_id, abuse_type text, message text
- We can store images on an object store. AWS S3 and Azure Blob Storage are popular because they are reliable, simple to use and maintain, and cost-efficient.
- image_address is the identifier used to retrieve an image from the object store.

When low latency is required, such as responding to user queries, we usually use SQL or in-memory databases with low latency such as Redis. NoSQL databases that use distributed file systems such as HDFS are for large data processing jobs.

7.4 Initial high level architecture

Referring to figure 7.1, we can discuss multiple possibilities for our initial Craigslist design, in order of complexity. We will discuss these 2 designs in the next 2 sections.

1. A monolith that uses a user authentication service for authentication, and an object store for storing posts.
2. A client frontend service, a backend service, a SQL service, an object store, and a user authentication service.

In all designs, we also include a logging service, as logging is almost always a must, so we can effectively debug our system. For simplicity, we can exclude monitoring and alerting. However, most cloud vendors provide logging, monitoring, and alerting tools that are easy to set up, and we should tend to use them.

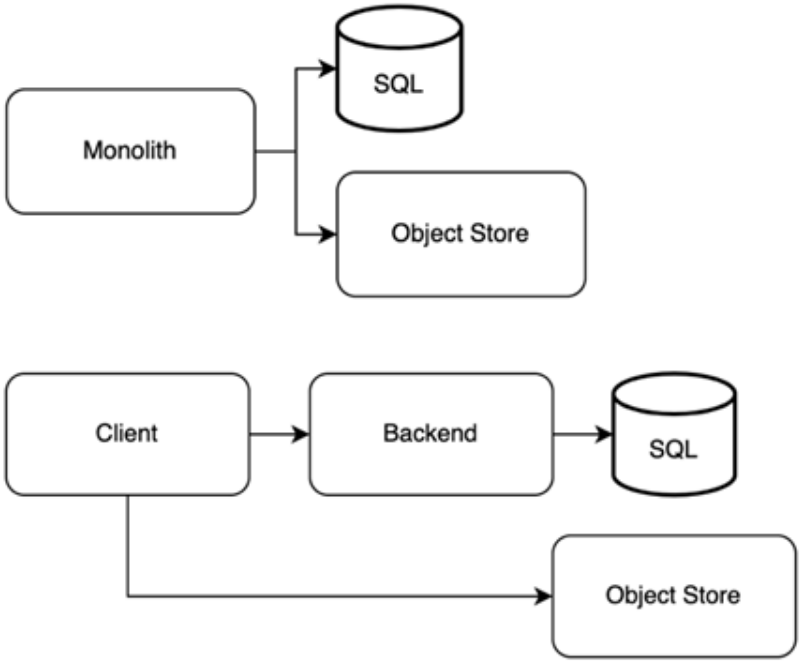

Figure 7.1 Simple initial designs for our high level architecture. (Top) Our high level architecture consists of just a monolith and an object store. (Bottom) A conventional high-level architecture with a UI frontend service and a backend service. Image files are stored in an object store, which clients make requests to. The rest of a post is stored in SQL.

7.5 A monolith architecture

Our first suggested design to use a monolith is unintuitive, and the interviewer may even be taken aback. One is unlikely to use monolith architecture for web services in her career. However, we should keep in mind that every design decision is about tradeoffs, and not be afraid to suggest such designs and discuss tradeoffs.

We can implement our application as a monolith that contains both the UI and the backend functionality, and store entire webpages in our object store. A key design decision is that we can store a post's webpage in its entirety in the object store, including the post's photos. Such a design decision means that we may not use many of the columns of the Post table discussed in section 7.3; we will use this table for the high level architecture illustrated in the bottom of figure 7.1, which we discuss later in this chapter.

Referring to figure 7.2, the home page is static, except for the location navigation bar (containing a regional location such as "SF bay area" and links to more specific locations such as "sfc", "sby", etc.), and the "nearby cl" section that has a list of other cities. Other

sites are static, including the sites on the left navigation bar such as “craigslist app” and “about craigslist”, and the sites on the bottom navigation bar such as “help”, “safety”, “privacy” etc. are static.

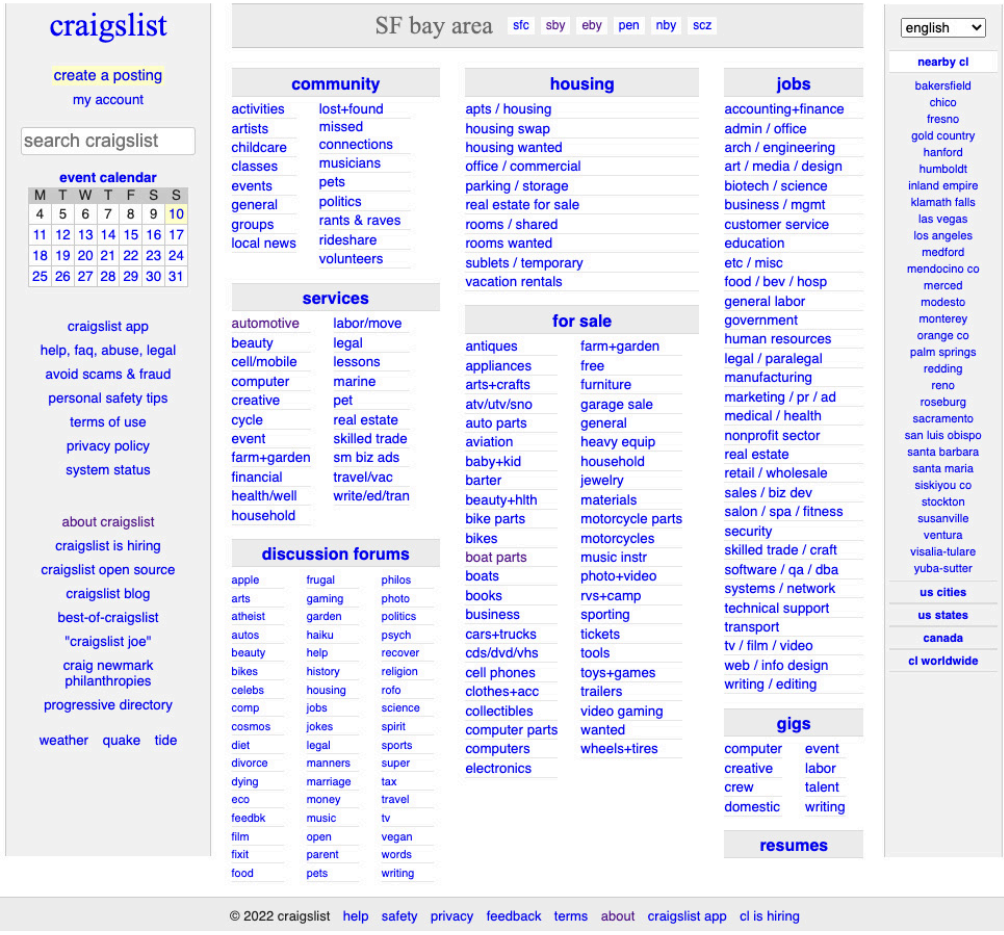

Figure 7.2 A Craigslist homepage. Image from https://sfbay.craigslist.org/.

This approach is simple to implement and maintain. Its main tradeoffs are:

1. HTML tags, CSS, and JavaScript are duplicated on every post.
2. If we develop a mobile app, it cannot share a backend with the browser app.
3. Any analytics on posts will require us to parse the HTML. This is only a minor disadvantage. We can develop and maintain our own utility scripts to fetch post pages and parse the HTML.

A disadvantage of the first tradeoff is that additional storage is required to store the duplicate page components. Another disadvantage is that new features or fields cannot be applied to old posts, though since posts are automatically deleted after 1 week, this may be acceptable depending on our requirements. We can discuss this with our interviewer as an example of how we should not assume any requirement when designing a system.

We can partially mitigate the second tradeoff by writing the browser app using a responsive design approach, and implement the mobile app as a wrapper around the browser app using WebViews. https://github.com/react-native-webview/react-native-webview is a WebView library for React Native. https://developer.android.com/reference/android/webkit/WebView is the WebView library for native Android, and https://developer.apple.com/documentation/webkit/wkwebview is the WebView library for native iOS. We can use CSS media queries (https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design#media_queries) to display different page layouts for phone displays, tablet displays, and laptop and desktop displays. This way, we do not need to use UI components from a mobile framework. A comparison of UX between using this approach versus the conventional approach of using the UI components in mobile development frameworks is outside the scope of this book.

For authentication on the backend service and Object Store Service, we can use a 3rd party user authentication service, or maintain our own. We can download the appropriate libraries. Refer to appendix B for a detailed discussion of Simple Login and OpenID Connect authentication mechanisms.

7.6 Using a SQL database and object store

The bottom diagram of figure 7.1 shows a more conventional high-level architecture. We have a UI frontend service that makes requests to a backend service and an object store service. The backend service makes requests to a SQL service.

In this approach, the object store is for image files, while the SQL database stores the rest of a post's data as discussed in section 7.4. We could have started without the object store, and store image files on SQL. However, this will mean that a client must download image files through the backend host. This is an additional burden on the backend host, increases image download latency, and increases the overall possibility of download failures from sudden network connectivity issues.

If we wish to keep our initial implementation simple, we can consider going without the feature to have images on posts, and add the object store when we wish to implement this feature.

That being said, since each post is limited to 10 image files of 1 MB each, and we will not store large image files, we can discuss with the interviewer on whether this requirement may change in the future. We can suggest that if we are unlikely to require larger images, we can store the images in SQL. The Image table can have a post_id text column and an image blob column. An advantage of this design is its simplicity.

7.7 Migrations are troublesome

While we are on the subject of choosing the appropriate data stores for our non-functional requirements, let's discuss the issue of data migrations before we proceed with discussing other features and requirements.

Another disadvantage of storing image files on SQL is that in the future we will have to migrate to storing them on an object store. Migration from one data store to another is generally troublesome and tedious.

Let's discuss a possible simple migration process, assuming the following:

1. We can treat both data stores as single entities i.e. replication is abstracted away from us, and we do not need to consider how data is distributed across various data centers to optimize for non-functional requirements like latency or availability.
2. Downtime is permissible. We can disable writes to our application during the data migration, so users will not add new data to the old data store while data is being transferred from the old data store to the new data store.
3. We can disconnect/terminate requests in progress when the downtime begins, so users who are making write (POST, PUT, DELETE) requests will receive 500 errors. We can give users advance notification of this downtime via various channels, such as email, browser and mobile push notifications, or a banner notification on the client.

We can write a Python script that runs on a developer's laptop to read records from one store and write it to another store. Referring to figure 7.3, this script will make GET requests to our backend to get the current data records, and POST requests to our new object store. Generally, this simple technique is suitable if the data transfer can complete within hours and only needs to be done once. It will take a developer a few hours to write this script, so it is not worth spending more time to speed up the data transfer.

We should expect our migration job may stop suddenly due to bugs or network issues, and we will need to restart the script execution. The write endpoints should be idempotent to prevent duplicate records from being written to our new data store. The script should do checkpointing, so it does not reread and rewrite records that have already been transferred. A simple checkpointing mechanism will suffice; after each write, we can save the object's ID to our local machine's hard disk. If the job fails midway, the job can resume from the checkpoint when we restart it (after fixing bugs if necessary).

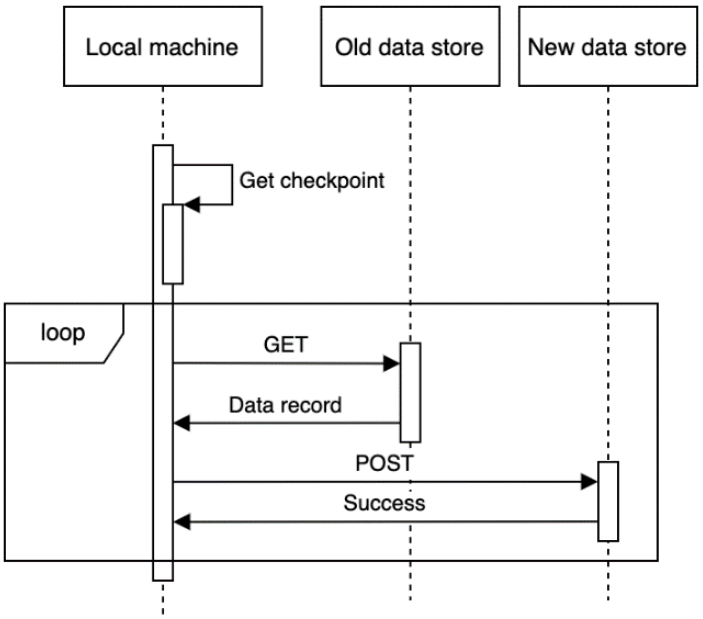

Figure 7.3 Sequence diagram of a simple data migration process. The local machine first finds the checkpoint if there is any, then makes the relevant requests to move each record from the old data store to the new data store.

An alert reader will notice that for this checkpoint mechanism to work, the script needs to read the records in the same order each time it is run. There are many possible ways to achieve this, including the following:

- If we can obtain a complete and sorted list of our records' IDs and store it in our local machine's hard disk, our script can load this list into memory before commencing the data transfer. Our script fetches each record by its ID, writes it to the new data store, and records on our hard disk that this ID has been transferred. As hard disk writes are slow, we can write/checkpoint these completed IDs in batches. With this batching, a job may fail before a batch of IDs have been checkpointed, so these objects may be reread and rewritten, and our idempotent write endpoints prevent duplicate records.
- If our data objects have any ordinal (indicates position) fields such as timestamps, our script can checkpoint using this field. For example, if we checkpoint by date, our script first transfers the records with the earliest date, checkpoints this date, increments the date, transfers the records with this date, and so on, until the transfer is complete.

This script must read/write the fields of the data objects to the appropriate tables and columns. The more features we add before a data migration, the more complex the migration script will be. More features mean more classes and properties. There will be more

database tables and columns, we will need to author a larger number of ORM/SQL queries, and these query statements will also be more complex and may have JOINs between tables.

If the data transfer is too big to complete with the above technique, we will need to run the script within the data center. We can run it separately on each host if the data is distributed across multiple hosts. Using multiple hosts allows the data migration to occur without downtime. If our data store is distributed across many hosts, it is because we have many users, and in these circumstances, downtime is too costly to revenue and reputation.

To decommission the old data store one host at a time, we can follow these steps on each host.

1. Drain the connections on the host. *Connection draining* refers to allowing existing requests to complete while not taking on new requests. Refer to sources like https://cloud.google.com/load-balancing/docs/enabling-connection-draining, https://aws.amazon.com/blogs/aws/elb-connection-draining-remove-instances-from-service-with-care/, and https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/config-conn-drain.html for more information on connection draining.
2. After the host is drained, run the data transfer script on the host.
3. When the script has finished running, we no longer need this host.

How should we handle write errors? If this migration takes many hours or days to complete, it will be impractical if the transfer job crashes and terminates each time there is an error with reading or writing data. Our script should instead log the errors and continue running. Each time there is an error, log the record that is being read or written, and continue reading and writing the other records. We can examine the errors, fix bugs if necessary, then rerun the script to transfer these specific records.

A lesson to take away from this is that a data migration is a complex and costly exercise that should be avoided if possible. When deciding on which data stores to use for a system, unless we are implementing this system as a proof-of-concept that will handle only a small amount of data (preferably unimportant data that can be lost or discarded without consequences), we should set up the appropriate data stores at the beginning, rather than set them up later then have to do a migration.

7.8 Writing and reading posts

Figure 7.4 is a sequence diagram of a poster writing a post, using the architecture in section 7.7. Although we are writing data to more than one service, we will not require distributed transaction techniques for consistency. The following steps occur:

1. The client makes a POST request to the backend with the post, excluding the images. The backend writes the post to the SQL database, and returns the post ID to the client.
2. The client can upload the image files one at a time to the object store, or fork threads to make parallel upload requests.

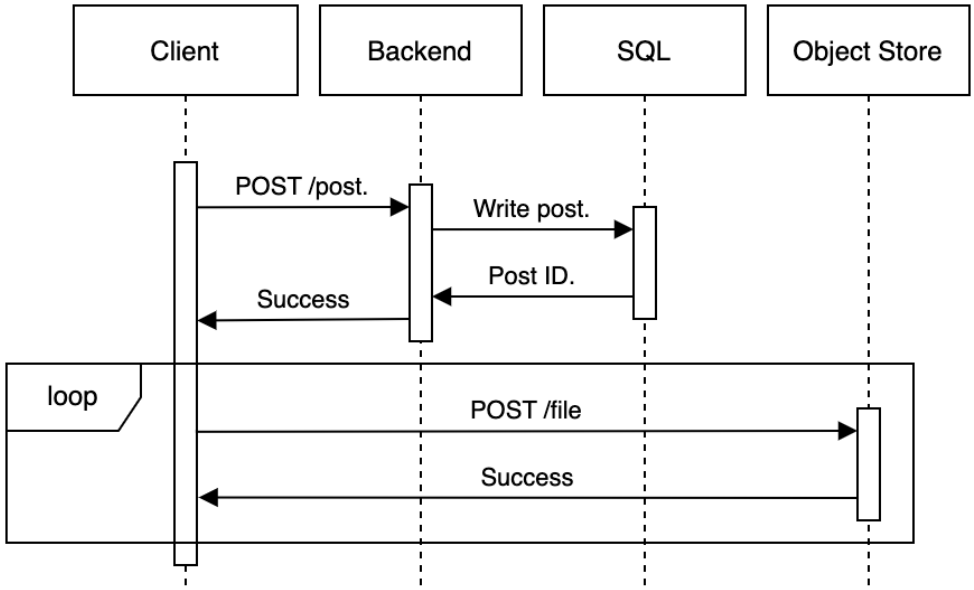

Figure 7.4 Sequence diagram of writing a new post, where the client handles image uploads.

In this approach, the backend returns 200 success without knowing if the image files were successfully uploaded to the object store. For the backend to ensure that the entire post is uploaded successfully, it must upload the images to the object store itself. Figure 7.5 illustrates such an approach. The backend can only return 200 success to the client after all image files are successfully uploaded to the object store, just in case image file uploads are unsuccessful. This may occur due to reasons such as the backend host crashing during the upload process, network connectivity issues, or if the object store is unavailable.

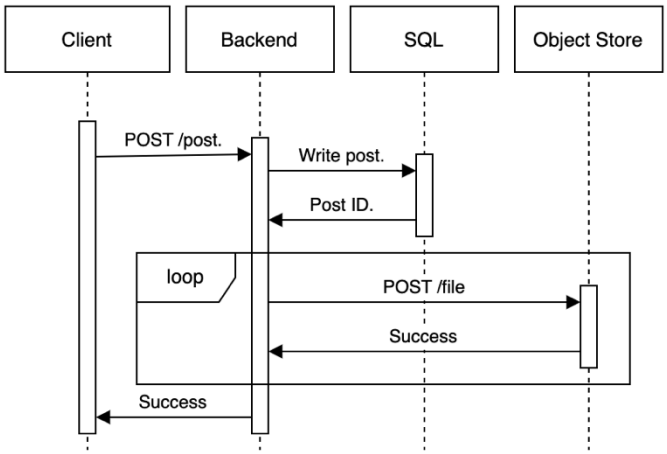

Figure 7.5 Sequence diagram of writing a new post, where the backend handles image uploads.

Let's discuss the tradeoffs of either approach.

Advantages of excluding the backend from the image uploads:

- Our system requires fewer resources. We push the burden of uploading images onto the client. If image file uploads went through our backend, our backend must scale up with our object store.
- Lower overall latency, as the image files do not need to go through an additional host. If we decide to use a CDN to store images, this latency issue will become even worse, as clients cannot take advantage of CDN edges close to their locations.

Advantages of including the backend in the image uploads:

- We will not need to implement and maintain authentication and authorization mechanisms on our object store. As the object store is not exposed to the outside world, our system has smaller overall attack surface.
- Viewers are guaranteed to be able to view all images of a post. In the previous approach, if some or all images are not successfully uploaded, viewers will not see them when they view the post. We can discuss with the interviewer if this is an acceptable tradeoff.

One way to capture most of the advantages of both approaches is for clients to write image files to the backend but read image files from the CDN.

What are the tradeoffs of uploading each image file in a separate request versus uploading all the files in a single request?

Does the client really need to upload each image file in a separate request? This complexity may be unnecessary. The maximum size of a write request will be slightly over

10 MB, which is small enough to be uploaded in seconds. But this means that retries will also be more expensive. Discuss these tradeoffs with the interviewer.

Should the object store rather than the backend assign the image file's URL? This way, we don't need to make the GET request in step 2. If the object store assigns the URL, a file must be fully uploaded to the object store before the latter can return the URL in the 200 response. An algorithm where the object store assigns the URL before the file upload is complete may be too complex. It will have more requests, and has to handle retries for all combinations of error responses.

The sequence diagram of a viewer reading a post is identical to figure 7.4, except that we have GET instead of POST requests. When a viewer reads a post, the backend fetches the post from the SQL database and returns it to the client. Next, the client fetches and displays the post's images from the object store. The image fetch requests can be parallel, so the files are stored on different storage hosts and are replicated, the files can be downloaded in parallel from separate storage hosts.

7.9 Functional Partitioning

The first step in scaling up can be to employ functional partitioning by geographical region such as city. This is commonly referred to as *geolocation routing*, serving traffic based on the location DNS queries originate from i.e. the geographic location of our users. We can deploy our application into multiple data centers, and route each user to the data center that serves her city, which will usually also be the closest data center. So the SQL cluster in each data center contains only the data of the cities that it serves. We can implement replication of each SQL cluster to 2 other SQL services in different data centers as described with MySQL's binlog-based replication (refer to section 3.3.2).

Craigslist does this geographical partitioning by assigning a subdomain to each city e.g. sfbay.craigslist.com, shanghai.craiglist.com etc. If we go to craigslist.com in our browser, the following steps occur. An example is shown on figure 7.6.

1. Our Internet Service Provider does a DNS lookup for craigslist.com and returns its IP address. (Browsers and OS have DNS caches, so the browser can use its DNS cache or the OS's DNS cache for future DNS lookups, faster than sending this DNS lookup request to the ISP.)
2. Our browser makes a request to the IP address of craigslist.com. The server determines our location based on our IP address, which is contained in the address, and returns us a 3xx response with the subdomain that corresponds to our location. This returned address can be cached by the browser and other intermediaries along the way, such as the user's OS and ISP.
3. Another DNS lookup is required to obtain the IP address of this subdomain.
4. Our browser makes a request to the IP address of the subdomain. The server returns the webpage and data of that subdomain.

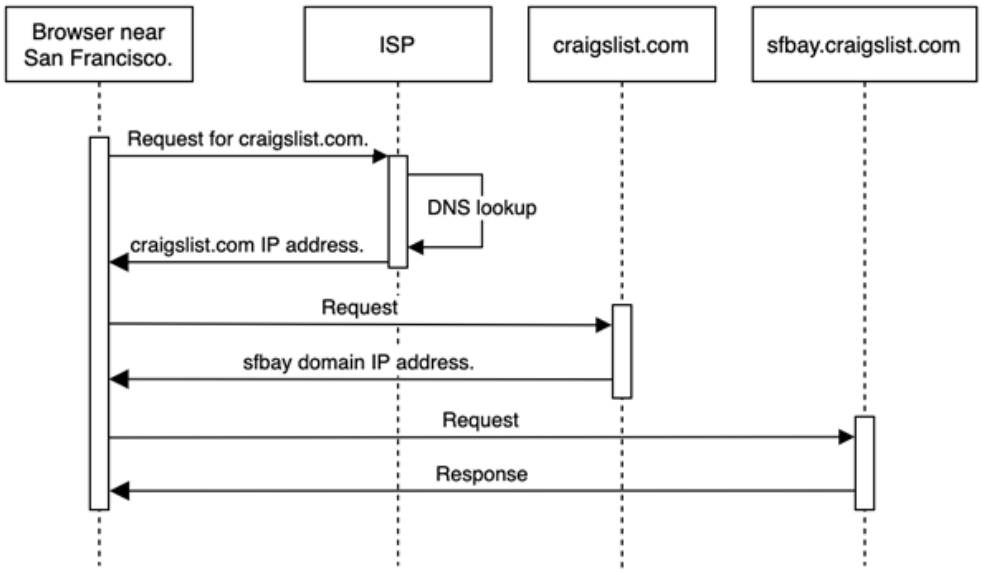

Figure 7.6 Sequence diagram for using GeoDNS to direct user requests to the appropriate IP address.

We can use GeoDNS for our Craigslist. Our browser only needs to do a DNS lookup once for craigslist.com, and the IP address returned will be the data center that corresponds to our location. Our browser can then make a request to this data center to get our city's posts. Instead of having a subdomain specified in our browser's address bar, we can state the city in a dropdown menu on our UI. The user can select a city in this dropdown menu to make a request to the appropriate datacenter and view that city's posts. Our UI can also provide a simple static webpage page that contains all Craigslist cities, where a user can click through to her desired city.

Cloud services such as AWS (https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-policy-geo.html) provide guides to configuring geolocation routing.

7.10 Caching

Certain posts may become very popular, and receive a high rate of read requests, for example a post which shows an item with a much lower price than its market value. To ensure compliance to our latency SLA e.g. 1 second p99, and prevent 504 timeout errors, we can cache popular posts.

We can implement an LRU cache using Redis. The key can be a post ID and value is the entire HTML page of a post. We may implement an image service in front of the object store, so it can contain its own cache mapping object identifiers to images.

The static nature of posts limits potential cache staleness, though a poster may update her post. If so, the host should refresh the corresponding cache entry.

7.11 CDN

Referring to figure 7.7, we can consider using a CDN, although Craigslist has very little static media (i.e. images and video) that are shown to all users. The static content it does have are CSS and JavaScript files, which are only a few MB in total. We can also use the browser cache for the CSS and JavaScript files.

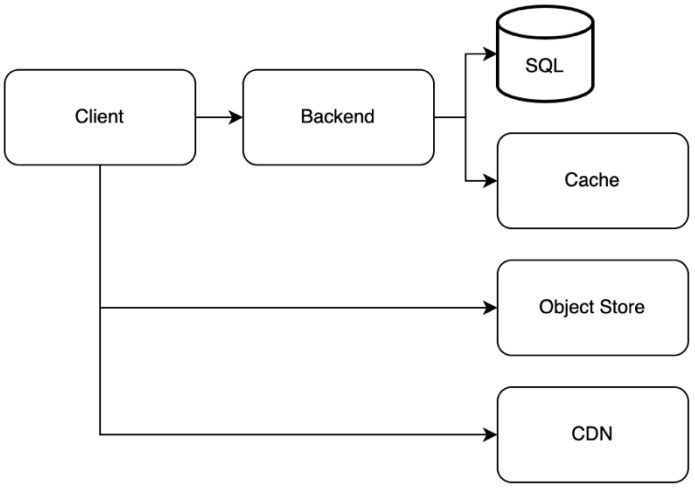

Figure 7.7 Our Craigslist architecture diagram after adding our cache and CDN.

7.12 Scaling reads with a SQL cluster

It is unlikely that we will need to go beyond functional partitioning and caching.

If we do need to scale reads, we can follow the approaches discussed in chapter 3, such as SQL replication.

7.13 Scaling write throughput

At the beginning of this chapter, we stated that this is a read-heavy application. So it is unlikely that we will need to allow programmatic creation of posts. This section is a hypothetical situation where we do allow it and perhaps expose a public API for post creation.

If there are traffic spikes in inserting and updating to the SQL host, the required throughput may exceed its maximum write throughput. Referring to https://stackoverflow.com/questions/2861944/how-to-do-very-fast-inserts-to-sql-server-2008, certain SQL implementations offer methods for fast INSERT e.g. SQL Server's ExecuteNonQuery achieves thousands of INSERTs per second. Another solution is to use batch commits instead of individual *INSERT statements*, so there is no log flush overhead for each *INSERT statement*.

USE A MESSAGE BROKER LIKE KAFKA

If the average throughput e.g. over a few hours is below the its write throughput, we can use a streaming solution like Kafka, by placing a Kafka service in front of the SQL services.

Figure 7.8 shows a possible design. When a poster submits a new or updated posts, the hosts of the Post Writer Service can produce to the Post topic. The service is stateless and horizontally scalable. We can create a new service we name "Post Writer" that continuously consumes from the Post topic and writes to the SQL service. This SQL service can use leader-follower replication that was discussed in chapter 3.

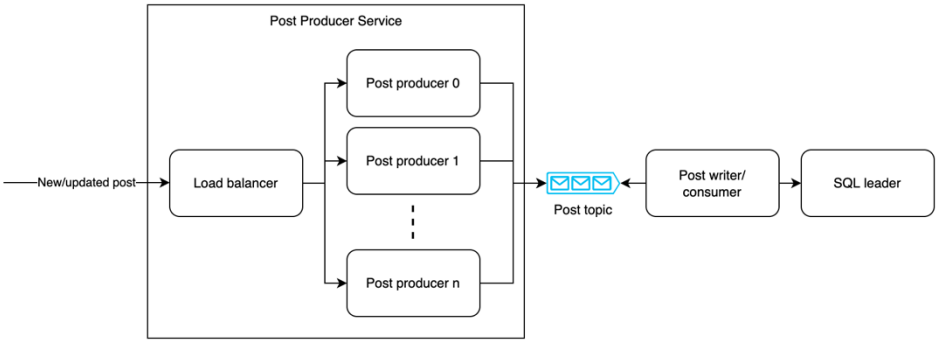

Figure 7.8 Using horizontal scaling and a message broker to handle write traffic spikes.

The main tradeoffs of this approach are complexity and eventual consistency. Our organization likely has a Kafka service which we can use, so we don't have to create our own Kafka service, somewhat negating the complexity. Eventual consistency duration increases as writes will take longer to reach the SQL followers.

If the required write throughput exceeds the average write throughput of a single SQL host, we can do further functional partitioning of SQL clusters, and have dedicated SQL clusters for categories with heavy write traffic. This solution is not ideal because the application logic for viewing posts will need to read from particular SQL clusters depending on category. Querying logic is no longer encapsulated in the SQL service, but present in the application too. Our SQL service is no longer independent on our backend service, and maintenance of both services become more complex.

If we need higher write throughput, we can use a NoSQL database such as Cassandra or Kafka with HDFS.

We may also wish to discuss adding a rate limiter (refer to chapter 8) in front of our backend cluster to prevent DDoS attacks.

7.14 Email Service

Our backend can send requests to a shared Email Service for sending email.

To send a renewal reminder to posters when a post is 7 days old, this can be implemented as a batch ETL job that queries our SQL database for posts that are 7 days old, then requests the Email Service to send an email for each post.

The notification service for other apps may have requirements such as handling unpredictable traffic spikes, low latency, and notifications should be delivered within a short time. Such as Notification Service is discussed in the next chapter.

7.15 Search

Referring to section 6.7.2, we create an Elasticsearch index on the Post table for users to search posts. We can discuss if we wish to allow the user to filter the posts before and after searching, such as by user, price, condition, location, recency of post etc., and we make the appropriate modifications to our index.

7.16 Removing old posts

Craigslist posts expire after a certain number of days, upon which the post is no longer accessible. This can be implemented with a cron job or Airflow, calling the `DELETE /old_posts` endpoint daily. `DELETE /old_posts` may be its own endpoint separate from `DELETE /post/{id}`, as the latter is a single simple database delete operation, while the former contains more complex logic to first compute the appropriate timestamp value then delete posts older than this timestamp value. Both endpoints may also need to delete the appropriate keys from the Redis cache.

This job is simple and non-critical, as it is acceptable for posts that were supposed to be deleted to continue to be accessible for days, so a cron job may be sufficient, and Airflow may introduce unneeded complexity. We must be careful not to accidentally delete posts before they are due, so any changes to this feature must be thoroughly tested in staging before a deployment to production. The simplicity of cron over a complex workflow management platform like Airflow improves maintainability, especially if the engineer who developed the feature has moved on and maintenance is being done by a different engineer.

Removing old posts or deletion of old data in general has the following advantages:

- Monetary savings on storage provisioning and maintenance.
- Database operations such as reads and indexing are faster.
- Maintenance operations which require copying all data to a new location are faster, less complex and lower cost; such as adding or migrating to a different database solution.
- Fewer privacy concerns for the organization and limits the impact of data breaches, though this advantage is not strongly felt since this is public data.

Disadvantages:

- Prevents analytics and useful insights that may be gained from keeping the data.
- Government regulations may make it necessary to keep data for a certain period.
- A tiny probability that the deleted post's URL may be used for a newer post, and a viewer may think she is viewing the old post. The probability of such events is higher if one is using a link shortening service. However, the probability of this is so low, and

it has little user impact, so this risk is acceptable. This risk will be unacceptable if sensitive personal data may be exposed.

If cost is an issue, and old data is infrequently accessed, an alternative to data deletion may be compression followed by storing on low-cost archival hardware such as tape, or an online data archival service like AWS Glacier or Azure Archive Storage. When certain old data is required, it can be written onto disk drives prior to data processing operations.

7.17 Monitoring and alerting

Besides what was discussed in chapter 9, we should monitor and send alerts for the following:

- Our database monitoring solution (discussed in chapter 14) should trigger a low-urgency alert if old posts were not removed.
- Anomaly detection for:
    - Number of posts added or removed.
    - High number of searches for a particular term.
    - Number of posts flagged as inappropriate.

7.18 Summary of our architecture discussion so far

Figure 7.9 shows our Craigslist architecture with many of the services we have discussed, namely the client, backend, SQL, cache, Notification Service, Search Service, object store, CDN, logging, monitoring, alerting, and batch ETL.

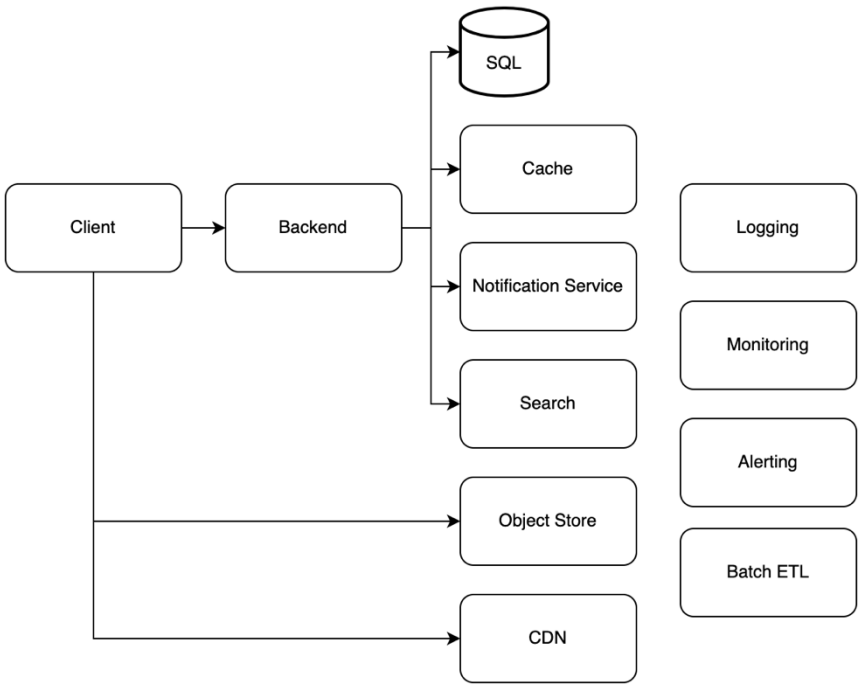

Figure 7.9 Our Craigslist architecture with Notification Service, Search, Logging, Monitoring, and Alerting. Logging, Monitoring, and Alerting can serve many other components, so on our diagram they are soon as loose components. We can define jobs on the Batch ETL Service for purposes such as to periodically remove old posts.

7.19 Other possible discussion topics

Our system design fulfills the requirements stated at the beginning of this chapter. The rest of the interview may be on new constraints and requirements.

7.19.1 Reporting posts

We have not discussed the functionality for users to report posts, as it is straightforward. Such a discussion may include a system design to fulfill such requirements:

- If a certain number of reports are made, the post is taken down, and the poster receives an email notification.
- A poster's account and email may be automatically banned, so the poster cannot log in to Craigslist or create posts. However, she can continue viewing posts without logging in, and can continue sending emails to other posters.
- A poster should be able to contact an admin to appeal this decision. We may need to discuss with the interviewer if we need a system to track and record these interactions and decisions.

- If a poster wishes to block emails, she will need to configure her own email account to block the sender's email address. Craigslist does not handle this.

7.19.2 Graceful degradation

How may we handle a failure on each component? What are the possible corner cases that may cause failures and how may we handle them?

7.19.3 Complexity

Craigslist is designed to be a simple classifieds app that is optimized for simplicity of maintenance by a small engineering team. The feature set is deliberately limited and well-defined, and new features are seldomly introduced. We may wish to discuss strategies to achieve this.

MINIMIZE DEPENDENCIES

Any app that contains dependencies to libraries and/or services naturally atrophies over time, and requires developers to maintain it just to keep providing its current functionality. Old library versions and occasionally entire libraries are deprecated, and services can be decommissioned, necessitating our developers to install a later version or find alternatives. New library versions or service deployments may also break our application. Library updates may also be necessary if bugs or security flaws are found in the currently used libraries. Minimizing our system's feature set minimizes its dependencies, which simplifies debugging, troubleshooting and maintenance.

This approach requires an appropriate company culture that focuses on providing the minimal useful set of features that do not require extensive customization for each market. For example, possibly the main reason that Craigslist does not provide payments is that the business logic to handle payments can be different in each city. We must consider different currencies, taxes, payment processors (Mastercard, Visa, PayPal, WePay etc.), and constant work is required to keep up with changes in these factors. Many big tech companies have engineering cultures that reward program managers and engineers for conceptualizing and building new services; such a culture is unsuitable for us here.

USE CLOUD SERVICES

In figure 11.5, other than the client and backend, every service can be coded on a cloud services. For example, we can use the following AWS services for each of the services in figure 11.5. Other cloud vendors like Azure or GCP provide similar services.

- SQL – RDS (https://aws.amazon.com/rds/)
- Object Store – S3 (https://aws.amazon.com/s3/)
- Cache – ElastiCache (https://aws.amazon.com/elasticache/)
- CDN – CloudFront (https://www.amazonaws.cn/en/cloudfront/)
- Notification Service – Simple Notification Service (https://aws.amazon.com/sns)
- Search – CloudSearch (https://aws.amazon.com/cloudsearch/)
- Logging, monitoring, and alerting – CloudWatch (https://aws.amazon.com/cloudwatch/)
- Batch ETL – Lambda functions with rate and cron expressions

(https://docs.aws.amazon.com/lambda/latest/dg/services-cloudwatchevents-expressions.html)

STORING ENTIRE WEBPAGES AS HTML DOCUMENTS

A webpage usually consists of a HTML template with interspersed JavaScript functions that make backend requests to fill in details. In the case of Craigslist, a post's HTML page template may contain fields such as title, description, price, photo etc., and each field's value can be filled in with JavaScript.

The simple and small design of Craigslist's post webpage allows the simpler alternative we first discussed in section 7.5, and we can discuss it further here. A post's webpage can be stored as a single HTML document in our database or CDN. This can be as simple as a key-value pair where the key is the post's ID and the value is the HTML document. This solution trades off some storage space since there will be duplicate HTML in every database entry. Search indexes can be built against this list of post IDs.

This approach also makes it less complex to add or remove fields from new posts. *If we decide to add a new required field e.g. "subtitle", we can* change the fields without a SQL database migration. We don't need to modify the fields in old posts, which anyway have a retention period and will be automatically deleted. The Post table is simplified, replacing a post's fields with the post's CDN URL. The columns become "id, ts, poster_id, location_id, post_url".

OBSERVABILITY

Any discussion of maintainability must emphasize the importance of observability, discussed in detail in section 6.5. We must invest in logging, monitoring, and alerting, automated testing, and adopt good SRE practices, including good monitoring dashboards, runbooks, and automation of debugging.

7.19.4 Item categories/tags

We can provide item categories/tags, such as "automotive", "real estate", "furniture" etc. and allow posters to place up to a certain number of tags (e.g. 3) on a listing. We can create a SQL dimension table for tags. Our Post table can have a column for a comma-separated tag list. An alternative is to have an associative/junction table "post_tag", as shown on figure 7.10.

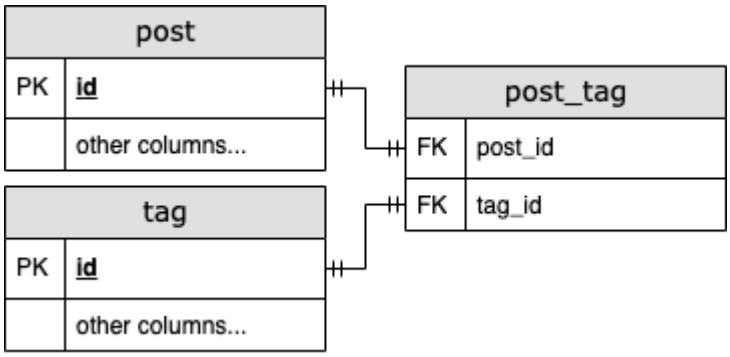

Figure 7.10 Associative/junction table for posts and tags. This schema normalization maintains consistency by avoiding duplicate data. If the data was in a single table, there will be duplicate values across rows.

We can expand this from a flat list to a hierarchical list, so users can apply more precise filters to view posts that are more relevant to their interests. For example, "real estate" may have the following nested subcategories.

- Real estate -> Transaction type -> Rent
- Real estate -> Transaction type -> Sale
- Housing type -> Apartment
- Housing type -> Single-family house
- Housing type -> Townhouse

7.19.5 Analytics and recommendations

We can create daily batch ETL jobs that query our SQL database and populate dashboards for various metrics:

- Number of items by tag.
- Tags which received the most clicks.
- Tags which got the highest number of viewers contacting posters.
- Tags with the fastest sales, which may be measured by how soon the poster removed the post after posting it.
- Numbers and geographical and time distributions of reported, suspected, and confirmed fraudulent posts.

Craigslist does not offer personalization, and posts are ordered starting from most recent. We may discuss personalization, which includes tracking user activity and recommending posts.

7.19.6 A/B testing

As briefly discussed in section 1.4.5, as we develop new features and aesthetic designs in our application, we may wish to gradually roll them out to an increasing percentage of users, rather than to all users at once.

7.19.7 Subscriptions and saved searches

We may provide an API endpoint for viewers to save search terms (with a character limit) and notify them (by email, text message, in-app message, etc.) of any new posts that match their saved searches. A POST request to this endpoint can write a row (timestamp, user_id, search_term) to a SQL table we name "saved_search".

This saved search subscription service can be a complex system in itself, as described in this section.

A user should receive a single daily notification that covers all her saved searches. This notification may consist of a list of search terms and up to 10 corresponding results for each search term. Each result in turn consists of a list of post data for that search term. The data can include a link to the post and some summary information (title, price, first 100 characters of the description) to display in the notification.

For example, if a user has the 2 saved search terms "san francisco studio apartment" and "systems design interview book", the notification may contain the following. (You certainly do not write down all of this during an interview. You can scribble down some quick snippets and verbally describe what they mean.)

```
[
  {
    "search_term": "san francisco studio apartment",
    "results": [
      {
        "link": "sfbay.craigslist.com/12345",
        "title": "Totally remodeled studio",
        "price": 3000,
        "description_snippet": "Beautiful cozy studio apartment in the Mission. Nice views
       in a beautiful and safe neighborhood. Clo"
      },
      {
        "link": "sfbay.craigslist.com/67890"
        "title": "Large and beautiful studio",
        "price": 3500,
        "description_snippet": "Amenities\nComfortable, open floor plan\nIn unit
       laundry\nLarge closets\nPet friendly\nCeiling fan\nGar"
      },
      ...
    ]
  },
  {
    "search_term": "systems design interview book",
    "results": [
      ...
    ]
  }
]
```

To send the users new results for their saved searches, we can implement a daily batch ETL job. We can suggest at least 2 ways to implement this job, a simpler way which allows duplicate requests to the search service, and another more complex way which avoids these duplicate requests.

7.19.8 Allow duplicate requests to the search service

Elasticsearch caches frequent search requests (https://www.elastic.co/blog/elasticsearch-caching-deep-dive-boosting-query-speed-one-cache-at-a-time), so frequent requests with the same search terms do not waste much resources. Our batch ETL job can process users and their individual saved search terms one at a time. Each process consists of sending a user's search terms as separate requests to the search service, consolidating the results, then sending a request to a notification service (the subject of chapter 9).

7.19.9 Avoid duplicate requests to the search service

Our batch ETL job runs the following steps:

5. Deduplicate the search terms, so we only need to run a search on each term once. We can run a SQL query like `SELECT DISTINCT LOWER(search_term) FROM saved_search WHERE timestamp >= UNIX_TIMESTAMP(DATEADD(CURDATE(), INTERVAL -1 DAY)) AND timestamp < UNIX_TIMESTAMP(CURDATE())` to deduplicate yesterday's search terms. Our search can be case-insensitive, so we lowercase the search terms as part of deduplication. Since our Craigslist design is partitioned by city, we should not have more than 100M search terms. Assuming an average of 10 characters per search term, there will be 1GB of data, which easily fits into a single host's memory.
6. For each search term:
    a) Send a request to the (Elasticsearch) search service and get the results.
    b) Query the "saved_search" table for the user IDs associated with this search term.
    c) For each (user ID, search term, results) tuple, send a request to a Notification Service.

What if the job fails during step 2? How do we avoid resending notifications to users? We can use a distributed transaction mechanism described in chapter 4. Or we can implement logic on the client that checks if a notification has already been displayed (and possibly dismissed) before displaying the notification. This is possible for certain types of clients like a browser or mobile app, but not for email or texting.

If saved searches should expire, we can clean up old table rows with a daily batch job that runs a SQL DELETE statement on rows older than the expiry date.

7.19.10 Rate limiting

All requests to our service can pass through a rate limiter to prevent any individual user from sending requests too frequently and so consume too much resources. The design of a rate limiter is discussed in chapter 8.

7.19.11 Large number of posts

What if we would like to provide a single URL where all listings are accessible to anyone (regardless of location)? Then the Post table may be too big for a single host, and the Elasticsearch index for posts may also be too big for a single host. However, we should

continue to serve a search query from a single host. Any design where a query is processed by multiple hosts and results aggregated in a single host before returning them to the viewer will have high latency and cost. How can we continue to serve search queries from a single host? Possibilities include:

- Impose a post expiry (retention period) of 1 week, and implement a daily batch job to delete expired posts. A short retention period means there is less data to search and cache, reducing the system's costs and complexity.
- Reduce the amount of data stored in a post.
- Do functional partitioning on categories of posts. Perhaps create separate SQL tables for various categories. But the application may need to contain the mappings to the appropriate table. Or this mapping can be stored in a Redis cache, and the application will need to query the Redis cache to determine which table to query.
- We may not consider compression, because it is prohibitively expensive to search compressed data.

7.19.12 Local regulations

Each jurisdiction (country, state, county, city, etc.) may have its own regulations that affect Craigslist. Examples include:

- The types of products or services permitted on Craigslist may differ by jurisdiction. How may our system handle this requirement? Section 15.10.1 discusses a possible approach.
- Customer data and privacy regulations. The company may not be allowed to export customer data outside of the country. It may be required to delete customer data on customer demand, or share data with governments. These considerations are likely outside the scope of the interview.

We will need to discuss the exact requirements. Is it sufficient to selectively display certain products and services sections on the client applications based on the user's location, or do we also need to prevent users from viewing or posting about banned products and services?

An initial approach to selectively display sections will be to add logic in the clients to display or hide sections based on the country of the user's IP address. Going further, if these regulations are numerous or frequently changing, we may need to create a Regulations Service that Craigslist admins can use to configure regulations, and the clients will make requests to this service to determine which HTML to show or hide. As this service will receive heavy read traffic and much lighter write traffic, we can apply CQRS techniques to ensure that writes succeed. For example, we can have separate regulation services for admins and viewers that scale separately, and periodic synchronization between them.

If we need to ensure that no forbidden content is posted on our Craigslist, we may need to discuss systems that detect forbidden words or phrases, or perhaps machine learning approaches.

A final thought is that Craigslist does not attempt to customize its listings based on country. A good example was how it removed its Personals section in 2018 in response to

new regulations passed in the United States. It did not attempt to keep this section in other countries. We can discuss the tradeoffs of such an approach.

7.20 Summary

- We discuss the users and their various required data types (like text, images or video) to determine the non-functional requirements, which in our Craigslist system were scalability, high availability, and high performance.
- A CDN is a common solution for serving images or video, but don't assume it is always the appropriate solution. Use an object store if these media will be served to a small fraction of users.
- Functional partitioning by GeoDNS is the first step in discussing scaling up.
- Next are caching and CDN, mainly to improve the scalability and latency of serving posts.
- Our Craigslist service is read-heavy. If we use SQL, consider leader-follower replication for scaling reads.
- Consider horizontal scaling of our backend and message brokers to handle write traffic spikes. Such a setup can serve write requests by distributing them across many backend hosts, and buffer them in a message broker. A consumer cluster can consume requests from the message broker and process them accordingly.
- Consider batch or streaming ETL jobs for any functionality that don't require real time latency. This is slower, but more scalability and lower cost.
- The rest of the interview may be on new constraints and requirements. In this chapter, the new constraints and requirements we mentioned were reporting posts, graceful degradation, decreasing complexity, adding categories/tags of posts, analytics and recommendations, A/B testing, subscriptions and saved searches, rate limiting, serving more posts to each user, and local regulations.

8

Rate limiting service

This chapter covers:

- Using rate limiting.
- Discussing a rate limiting service.
- Various rate limiting algorithms.

Rate limiting is a common service that we should almost always mention during a system design interview, and is mentioned in most of the example questions in this book. This chapter aims to address situations where 1) the interviewer may ask for more details when we mention rate limiting during an interview, and 2) the question itself is to design a rate limiting service.

Rate limiting can be implemented as a library or as a separate service called by a frontend, API gateway, or service mesh. In this question, we implement it as a service to gain the advantages discussed in chapter 6. Figure 8.1 illustrates a rate limiter design that we will discuss in this chapter.

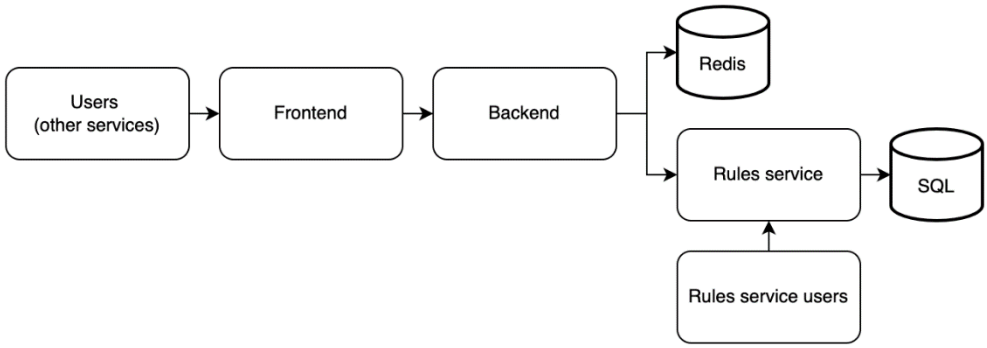

Figure 8.1 Initial high-level architecture of rate limiter. The frontend, backend, and Rules service all also log to a shared logging service; this is not shown here. The Redis database is usually implemented as a shared Redis service, rather than our service provisioning its own Redis database. The Rules Service users may make API requests to the Rules Service via a browser app. We can store the rules in SQL.

Rate limiting defines the rate at which consumers can make requests to API endpoints. Rate limiting prevents inadvertent or malicious overuse by clients, especially bots. In this chapter, we refer to such clients as "excessive clients".

Examples of inadvertent overuse include the following:

- Our client is another web service that experienced a (legitimate or malicious) traffic spike.
- The developers of that service decided to run a load test on their production environment.

Such inadvertent overuse causes a "noisy neighbor" problem, where a client utilizes too much resource on our service, so our other clients will experience higher latency or higher rate of failed requests.

Malicious attacks include the following. There are other bot attacks that rate limiting does not prevent. Refer to https://www.cloudflare.com/learning/bots/what-is-bot-management/ for more information.

- *Denial-of-service* (DoS) or *distributed denial-of-service* (DDoS) attacks. DoS floods a target with requests, so normal traffic cannot be processed. DoS uses a single machine, while DDoS is the use of multiple machines for the attack. This distinction is unimportant in this chapter, and we refer to them collectively as "DoS".
- Brute force attack, which is repeated trial-and-error to find sensitive data, such as cracking passwords, encryption keys, API keys, and SSH login credentials.
- Web scraping, using bots to make GET requests to many webpages of a web application to obtain a large amount of data. An example is scraping Amazon product pages for prices and product reviews.

8.1 Alternatives to a rate limiting service, and why they are

infeasible

Why don't we scale out the service by monitoring the load and adding more hosts when needed, instead of doing rate limiting? We can design our service to be horizontally scalable, so it will be straightforward to add more hosts to serve the additional load. We can consider auto-scaling.

The process of adding new hosts when a traffic spike is detected may be too slow. Adding a new host involves steps that take time, like provisioning the host hardware, downloading the necessary Docker containers, starting up the service on the new host, then updating our load balancer configuration to direct traffic to the new host. This process may be too slow, and our service may already have crashed by the time the new hosts are ready to serve traffic. Even auto-scaling solutions may be too slow.

A load balancer can limit the number of requests sent to each host. Why don't we use our load balancer to ensure that hosts are not overloaded, and drop requests when our cluster has no more spare capacity?

We should not serve malicious requests as we mentioned above. Rate limiting guards against such requests by detecting their IP addresses and simply dropping them. As discussed later, our rate limiter can usually return 429 Too Many Requests, but if we are sure that certain requests are malicious, we can choose either of these options:

1. Drop the request and not return any response, and allow the attacker to think that our service is experiencing an outage.
2. Shadow ban the user by returning 200 with an empty or misleading response.

Why does rate limiting need to be a separate service? Can each host independently track the request rate from its requestors, and rate limit them?

The reason is that certain requests are more expensive than others. Certain users may tend to make requests that return more data, require more expensive filtering and aggregation, or involve JOIN operations between larger datasets. A host may become slow from processing expensive requests from particular clients.

A level 4 load balancer cannot process a request's contents. We will need a level 7 load balancer for sticky sessions (to route requests from a user to the same host), which introduces cost and complexity. If we do not have other use cases for a level 7 load balancer, it may not be worth it to use level 7 load balancer just for this purpose, and a dedicated and shared rate limiting service may be a better solution.

Table 8.1 summarizes our discussion.

Table 8.1 Comparison of rate limiting to its alternatives.

Rate Limiting	Add new hosts	Use level 7 load balancer
Handles traffic spikes by returning 429 Too Many Requests responses to the users with high request rates.	Adding new hosts may be too slow to respond to traffic spikes. Our service may crash by the time the new hosts are ready to serve traffic.	Not a solution to handle traffic spikes.

Handles DoS attacks by providing misleading responses.	Processes malicious requests, which we should not do.	Not a solution.
Can rate limit users who make expensive requests.	Causes our service to incur the costs of processing expensive requests.	Can reject expensive requests, but may be too costly and complex as a standalone solution.

8.2 When not to do rate limiting

Rate limiting is not necessarily the appropriate solution for any kind of client overuse.

For example, consider a social media service we designed. A user may subscribe to updates associated with a particular hashtag. If a user makes too many subscription requests within a certain period, the social media service may respond to the user "you have made too many subscription requests within the last few minutes". If we did rate limiting, we will simply drop the user's requests and return 429 (Too Many Requests), or return nothing and the client decides the response is 500. This will be a poor user experience. If the request is sent by a browser or mobile app, the app can display to the user that she sent too many requests, providing a good user experience.

Another example is services that charge subscription fees for certain request rates (e.g. different subscription fees for 1000 or 10000 hourly requests). If a client exceeds its quota for a particular time interval, further requests should not be processed until the next time interval. A shared rate limiting service is not the appropriate solution to prevent clients from exceeding their subscriptions. As discussed in more detail below, the shared rate limiting service should be limited to supporting simple use cases.

8.3 Functional requirements

Our rate limiting service is a shared service, and our users are services (mainly external-facing services). We refer to such services as "user services". A user should be able to set a maximum request rate over which further requests from the same requestor will be delayed or rejected, with 429 response. We can assume the interval can be 10 seconds or 60 seconds. We can set a minimum of 10 requests in 10 seconds. Other functional requirements are as follows.

- We assume that each user service must rate limit its requestors across its hosts, but we do not need to rate limit the same users across services. Rate limiting is independent on each user service.
- A user can set multiple rate limits, one per endpoint. We do not need more complicated user-specific configurations, such as allowing different rate limits to particular requestors/users. We want our rate limiter to be a cheap and scalable service that is easy to understand and use.
- Our users should be able to view which users were rate limited and the timestamps these rate limiting events began and ended. We provide an endpoint for this.
- We can discuss with the interviewer whether we should log every request, as we will need a large amount of storage to do so, which is expensive. We will assume that this is needed, and discuss techniques to save storage to reduce cost.

- We should log the rate-limited requestors, for manual follow-up and analytics. This is especially required for suspected attacks.

8.4 Non-functional requirements

Rate limiting is a basic functionality required by virtually any service. It must be scalable, have high performance, be as simple as possible, secure and private. Rate limiting is not essential to a service's availability, so we can trade off high-availability and fault tolerance. Accuracy and consistency are fairly important but not stringent.

8.4.1 Scalability

Our service should be scalable to billions of daily requests that query whether a particular requestor should be rate limited. Requests to change rate limits will only be manually made by internal users in our organization, so we do not need to expose this capability to external users.

How much storage is required? Assume 1 billion users, and we need to store up 100 requests per user. Only the user IDs and a queue of 100 timestamps per user need to be recorded; each is 64 bits. Our rate limiter is a shared service, so we will need to associate requests with the service that is being rate limited. A typical big organization has thousands of services. Let's assume up to 100 of them need rate limiting.

We should ask whether our rate limiter actually needs to store data for 1 billion users. What is the retention period? A rate limiter usually should only need to store data for 10 seconds, as it makes a rate limiting decision based on the user's request rate for the last 10 seconds. Moreover, we can discuss with the interviewer whether there will be more than 1-10 million users within a 10-second window. Let's make a conservative worst-case estimate of 10 million users. Our overall storage requirement is 100 * 64 * 101 * 10M = 808 GB. If we use Redis and assign a key to each user, a value's size will be just 64 * 100 = 800 bytes. It may be impractical to delete data immediately after it is older than 10 seconds, so the actual amount of required storage depends on how fast our service can delete old data.

8.4.2 Performance

When another service receives a request from its user (We refer to such requests as *user requests*.), it makes a request to our rate limiting service (We refer to such requests as *rate* limiter requests.) to determine if the user request should be rate-limited. The rate limiter request is blocking; the other service cannot respond to its user before the rate limiter request is completed. The rate limiter request's response time adds to the user request's response time. So our service needs very low latency, perhaps a P99 of 100ms. The decision to rate-limit or not rate-limit the user request must be quick.

We don't require low latency for viewing or analytics of logs.

8.4.3 Complexity

Our service will be a shared service, used by many other services in our organization. Its design should be simple to minimize the risk of bugs and outages, aid troubleshooting, allow

it to focus on its single functionality as a rate limiter, and minimize costs. Developers of other services should be able to integrate our rate limiting solution as simply and seamlessly as possible.

8.4.4 Security and privacy

Chapter 2 discussed security and privacy expectations for external and internal services. Here, we can discuss some possible security and privacy risks. The security and privacy implementations of our user services may be inadequate to prevent external attackers from accessing our rate limiting service. Our (internal) user services may also attempt to attack our rate limiter e.g. by spoofing requests from another user service to rate limit it. Our user services may also violate privacy by requesting data about rate limiter requestors from other user services.

For these reasons, we will implement security and privacy in our rate limiter's system design.

8.4.5 Availability and fault-tolerance

We may not require high availability or fault-tolerance. If our service has less than three nines availability and is down for an average of a few minutes daily, user services can simply process all requests during that time, and not impose rate limiting. As discussed later in this chapter, we can design our service to use a simple highly-available cache to cache the IP addresses of excessive clients. If the rate limiting service had identified excessive clients just prior to the outage, this cache can continue to serve rate limiter requests during the outage, so these excessive clients will continue to be rate limited. It is statistically unlikely that an excessive client will occur during the few minutes the rate limiting service has an outage.

8.4.6 Accuracy

To prevent poor user experience, we should not erroneously identify excessive clients and rate limit them. In case of doubt, we should not rate limit the user. The rate limit value itself does not need to be precise. For example, if the limit is 10 requests in 10 seconds, it is acceptable to occasionally rate limit a user at 8 or 12 requests in 10 seconds. If we have an SLA that requires us to provide a minimum request rate, we can set a higher rate limit e.g. 12+ requests in 10 seconds.

8.4.7 Consistency

The previous discussion on accuracy leads us to the related discussion on consistency. We do not need strong consistency for any of our use cases. When a user service updates a rate limit, this new rate limit need not immediately apply to new requests; a few seconds of inconsistency may be acceptable. Eventual consistency is also acceptable for viewing logged events such as which users were rate limited, or performing analytics on these logs. Eventual rather than strong consistency will allow a simpler and cheaper design.

8.5 Discuss user stories and required service components

A rate limiter request contains a required user ID (in a HTTP request query parameter or body) and a user service ID (in the HTTP origin field). Since rate limiting is independent on each user service, the ID format can be specific to each user service. The ID format for a user service is defined and maintained by the user service, not by our rate limiting service. We can use the user service ID to distinguish possible identical user IDs from different user services. As each user service has a different rate limit, our rate limiter also uses the user service ID to determine the rate limit value to apply.

Our rate limiter will need to store this (user ID, service ID) data for 60 seconds, since it must use this data to compute the user's request rate, to determine if it is higher than the rate limit. To minimize the latency of retrieving any user's request rate or any service's rate limit, these data must be stored (or cached) on in-memory storage. As consistency and latency are not required for logs, we can store logs on an eventually-consistent storage like HDFS, which has replication to avoid data loss from possible host failures.

Lastly, user services can make infrequent requests to our rate limiting service to create and update rate limits for their endpoints. This request can consist of a user service ID, endpoint ID, and the desired rate limit e.g. a maximum of 10 requests in 10 seconds.

Putting these requirements together, we need:

1. A database with fast reads and writes for counts. The schema will be simple; unlikely to be much more complex than (user ID, service ID). We can use an in-memory database like Redis.
2. A service where rules can be defined and retrieved, which we call the Rules Service.
3. A service that makes requests to the Rules Service and the Redis database, which we can call the Backend Service.

The 2 services are separate, because requests to the Rules Service for adding or modifying rules should not interfere with requests to the rate limiter that determine if a request should be rate limited.

8.6 High-level architecture

Figure 8.2 (repeated from figure 8.1) illustrates our high-level architecture considering these requirements and stories. When a client makes a request to our rate limiting service, this request initially goes through the frontend or service mesh. If the frontend's security mechanisms allow the request, the request goes to the backend, where the following steps occur:

1. Get the service's rate limit from the Rules Service. This can be cached for lower latency and lower request volume to the Rules Service.
2. Determine the service's current request rate, including this request.
3. Return a response that indicates if the request should be rate limited.

Steps 1 and 2 can be done in parallel to reduce overall latency, by forking a thread for each step or using threads from a common thread pool.

The frontend and Redis (distributed cache) services in our high-level architecture in figure 8.2 are for horizontal scalability. This is the distributed cache approach discussed in section 2.5.3.

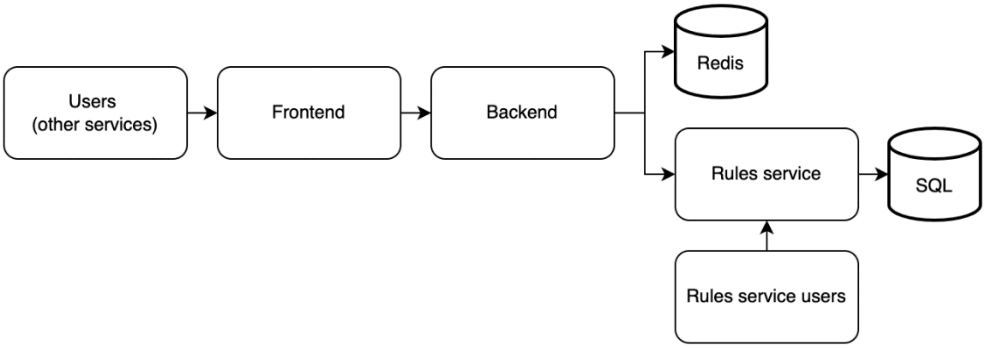

Figure 8.2 Initial high-level architecture of rate limiter. The frontend, backend, and Rules service all also log to a shared logging service; this is not shown here. The Redis database is usually implemented as a shared Redis service, rather than our service provisioning its own Redis database. The Rules Service users may make API requests to the Rules Service via a browser app.

We may notice in figure 8.2 that our Rules Service has users from 2 different services (Backend and Rules service users) with very different request volumes, and 1 of which (Rules Service users) does all the writes.

Referring back to the leader-follower replication concepts in sections 3.3.2 and 3.3.3, and illustrated in figure 8.3, the Rules Service users can make all their SQL queries, both reads and writes, to the leader node. The backend should make its SQL queries, which are only SELECT queries, to the follower nodes. This way, the Rules Service users have high consistency and experience high performance.

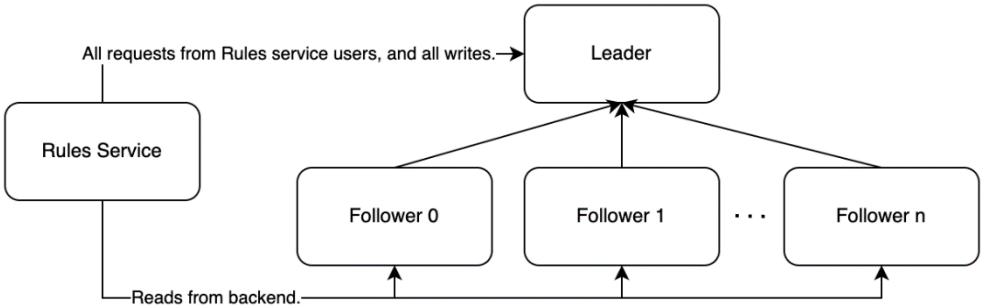

Figure 8.3 The leader host should process all requests from Rules service users, and all write operations. Reads from the backend can be distributed across the follower hosts.

Referring to figure 8.4, as we do not expect rules to change often, we can add a Redis cache to the Rules Service to improve its read performance even further. Figure 8.4 displays cache-aside caching, but we can also use other caching strategies from section 3.8. Our Backend Service can also cache rules in Redis. As discussed earlier in section 8.4.5, we can also cache the IDs of excessive users. As soon as a user exceeds its rate limit, we can cache its ID along with an expiry time where a user should no longer be rate limited. Then our backend need not query the Rules Service to deny a user's request.

If we are using AWS (Amazon Web Services), we can consider DynamoDB instead of Redis and SQL. DynamoDB can handle millions of requests per second (https://aws.amazon.com/dynamodb/), and it can be either eventually consistent or strongly consistent (https://docs.aws.amazon.com/whitepapers/latest/comparing-dynamodb-and-hbase-for-nosql/consistency-model.html), but using it subjects us to vendor lock-in.

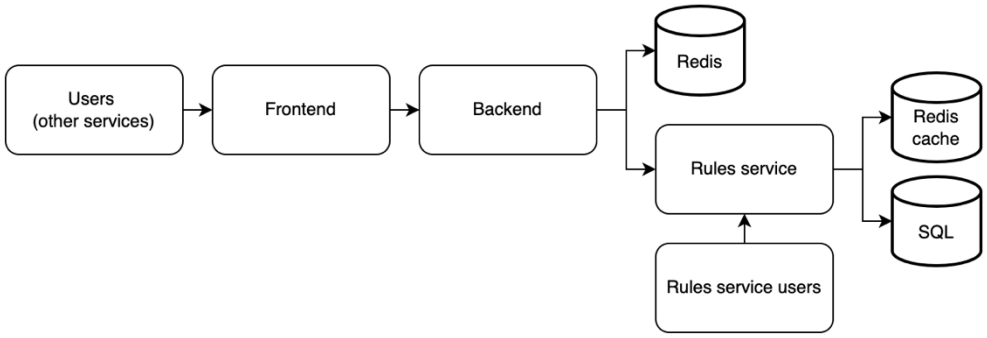

Figure 8.4 Rate limiter with Redis cache on the Rules Service. Frequent requests from the backend can be served from this cache instead of the SQL database.

The backend has all our non-functional requirements. It is scalable, has high performance, not complex, is secure and private, and eventually consistent. The SQL database with its leader-leader replication is highly available and fault-tolerant, which goes beyond our requirements. We will discuss accuracy in a later section. This design is not scalable for the Rules Service users, which is acceptable as discussed in section 8.4.1.

Considering our requirements, our initial architecture may be overengineered, and overly complex and costly. This design is highly accurate and strongly consistent. Can we trade off some accuracy and consistency for lower cost? Let's first discuss 2 possible approaches to scaling up our rate limiter.

1. A host can serve any user, by not keeping any state and fetching data from a shared database. This is the stateless approach we have been following for most questions in this book.
2. A host serves a fixed set of users and stores its user's data. This is a stateful approach that we discuss in the next section.

8.7 Stateful approach / sharding

Figure 8.5 illustrates the backend of a stateful solution that is closer to our non-functional requirements. When a request arrives, our load balancer routes it to its host. Each host stores the counts of its clients in its memory. The host determines if the user has exceeded her rate limit, and returns true or false. If a user makes a request and its host is down, our service will return a 500 error, and the request will not be rate limited.

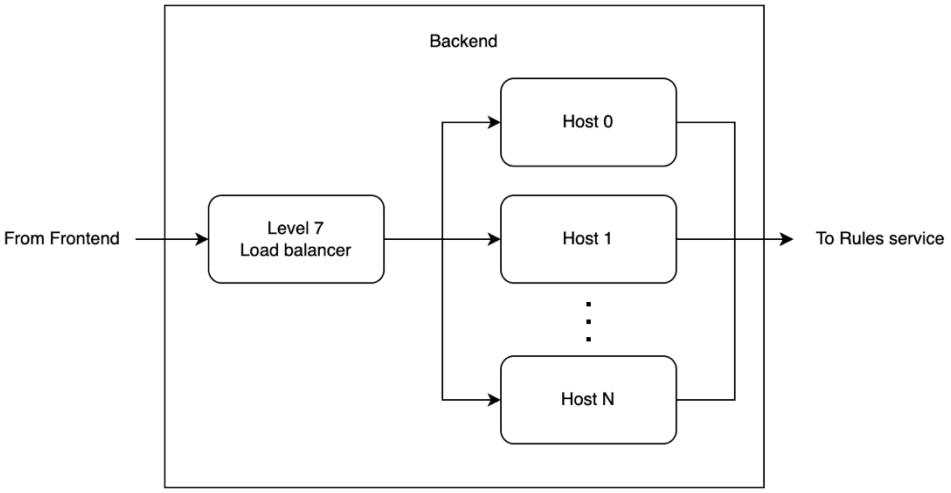

Figure 8.5 Backend architecture of rate limiter that employs a stateful sharded approach. The counts are stored in the hosts' memory, rather than in a distributed cache like Redis.

A stateful approach requires a level 7 load balancer. This may seem to contradict what we discussed in section 8.1 about using a level 7 load balancer, but note that we are now discussing using it in a distributed rate limiting solution, not just for sticky sessions to allow each host to perform its own rate limiting.

A question that immediately arises in such an approach is fault-tolerance, whether we need to safeguard against data loss when a host goes down. If so, this leads to discussions on topics like replication, failover, and hot shards and rebalancing. As briefly discussed in section 3.1, we can use sticky sessions in replication. But in our requirements discussion above, we discussed that we don't need consistency, high availability, or fault-tolerance. If a host that contains certain users' data goes down, we can simply assign another host to those users and restart the affected users' request rate counts from 0. Instead, the relevant discussion will be on detecting host outages, assigning and provisioning replacement hosts, and rebalancing traffic.

The 500 error should trigger an automated response to provision a new host. Our new host should fetch its list of addresses from the configuration service, which can be a simple manually-updated file stored on a distributed object storage solution like AWS S3 (For high

availability, this file must be stored on a distributed storage solution and not on a single host.), or a complex solution like ZooKeeper. When we develop our rate limiting service, we should ensure that the host setup process does not exceed a few minutes. We should also have monitoring on the host setup duration, and trigger a low-urgency alert if the setup duration exceeds a few minutes.

We should monitor for hot shards, and periodically rebalance traffic across our hosts. We can periodically run a batch ETL job that reads the request logs, identifies hosts that receive large numbers of requests, determines an appropriate load balancing configuration, then writes this configuration to a configuration service. The ETL job can also push the new configuration to the load balancer service. We write to a configuration service in case any load balancer host goes down. When the host recovers or a new load balancer host is provisioned, it can read the configuration from the configuration service.

Figure 8.6 illustrates our backend architecture with the rebalancing job. This rebalancing prevents a large number of heavy users from being assigned to a particular host and cause it to go down. Since our solution does not have failover mechanisms that distribute the users of a failed host over other hosts, we do not have the risk of a death spiral, where a host fails because of excessive traffic, then its traffic is redistributed over the remaining hosts and increases their traffic, which in turn causes them to fail.

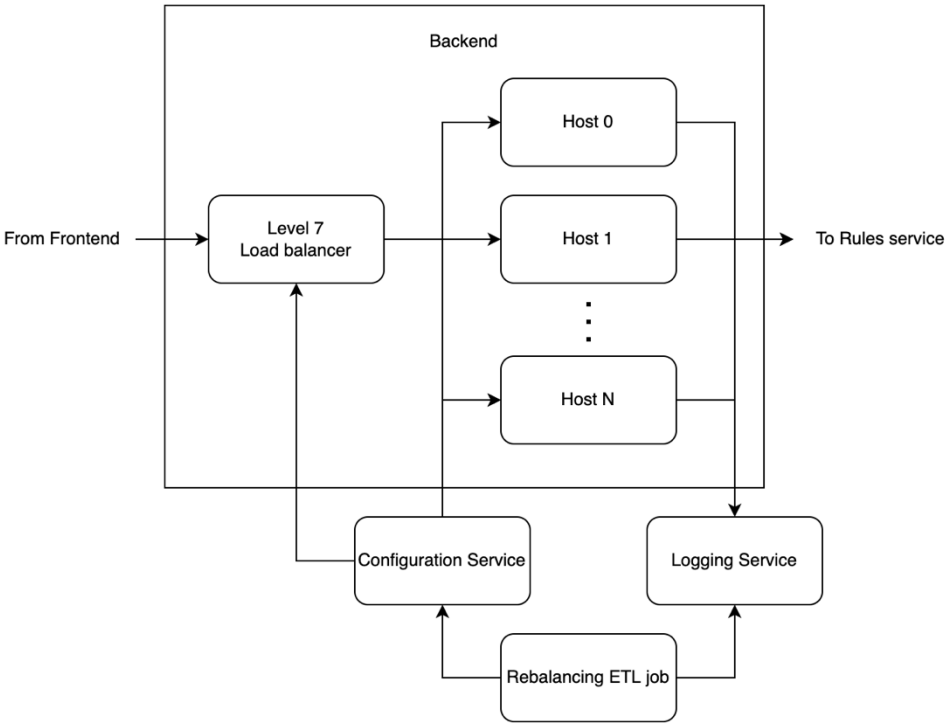

Figure 8.6 Backend architecture with rebalancing ETL job.

A tradeoff of this approach is that it is less resilient to DoS/DDoS attacks. If a user has a very high request rate, such as hundreds of requests per second, its assigned host cannot handle this, and all users assigned to this host cannot be rate limited. We may choose to have an alert for such cases, and we should block requests from this user across all services. Load balancers should drop requests from this IP address i.e. do not send the requests to any backend host, and do not return any response, but do log the request.

Compared to the stateless approach, the stateful approach is more complex and has higher consistency and accuracy, but has lower:

- Cost.
- Availability.
- Fault-tolerance.

Overall, this approach is a poor cousin of a distributed database. We attempted to design our own distributed storage solution, and it will not be as sophisticated or mature as widely-used distributed databases. It is optimized for simplicity and low cost, and has neither strong consistency nor high availability.

8.8 Storing all counts in every host

The stateless backend design discussed in section 8.6 used Redis to store request timestamps. Redis is distributed, and it is scalable and highly available. It also has low latency, and will be an accurate solution. However, this design requires us to use a Redis database, which is usually implemented as a shared service. Can we avoid our dependency on an external Redis Service, which will expose our rate limiter to possible degradation on that service?

The stateful backend design discussed in section 8.7 avoids this lookup by storing state in the backend, but it requires the load balancer to process every request to determine which host to send it to, and it also requires reshuffling to prevent hot shards. What if we can reduce the storage requirement such that all user request timestamps can fit in memory on a single host?

8.8.1 High-level architecture

How can we reduce the storage requirement? We can reduce our 808 GB storage requirement to 8.08 GB ≅ 8 GB by creating a new instance of our Rate Limiting Service for each of the ~100 services that uses it, and use the frontend to route requests by service to the appropriate service. 8 GB can fit into a host's memory. Due to our high request rate, we cannot use a single host for rate limiting. If we use 128 hosts, each host will store only 64 MB.

Figure 8.7 is the backend architecture of this approach. When a host receives a request, it does the following in parallel:

- Makes a rate limiting decision and returns it.
- Asynchronously synchronizes its timestamps with other hosts.

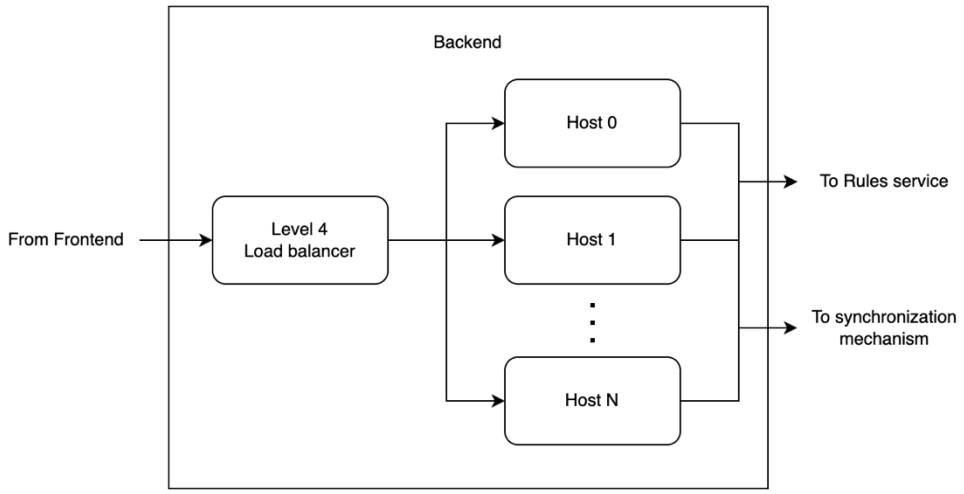

Figure 8.7: High-level architecture of a Rate Limiting Service where all user request timestamps can fit into a single backend host, requests are randomly load balanced across hosts, and each host can respond to a user request with a rate limiting decision without first making requests to other services or hosts. Hosts synchronize their timestamps with each other in a separate process.

Our level 4 load balancer randomly balances requests across hosts, so a user may be directed to different hosts in each rate limiting request. For rate limits to accurately computed, the rate limits on our hosts need to kept synchronized. There are multiple possible ways to synchronize the hosts. We discuss them in detail in the next section, except to say that we will use streaming instead of batch update, as batch update will be too infrequent, and cause users to be rate limited at a much higher request rate than the set request rate.

Compared to the other 2 designs discussed earlier (the stateless backend design and stateful backend design), this design trades off consistency and accuracy for lower latency and higher performance (it can process a higher request rate). As a host may not have all the timestamps in memory before making a rate limiting decision, it may compute a lower request rate than the actual value. It also has these characteristics:

- Use a level 4 load balancer to direct requests to any host like a stateless service. (Though the frontend)
- A host can make a rate limiting decisions with its data in memory.
- Data synchronization can be done in an independent process.

What if a host goes down and its data is lost? Certain users who will be rate limited will be permitted to make more requests before they are rate limited. As discussed earlier, this is acceptable. Refer to page 157-158 of Martin Kleppmann's book "Designing Data-Intensive Systems" for a brief discussion on leader failover and possible issues.

Table 8.2 summarizes our comparison between the 3 approaches that we have discussed.

Table 8.2 Comparison of our stateless backend design, stateful backend design, and design that stores counts in every host.

Stateless backend design	Stateful backend design	Storing counts in every host
Stores counts in a distributed database.	Stores each user's count in a backend host.	Store every user's counts in every host.
Stateless, so a user can be routed to any host.	Requires a level 7 load balancer to route each user to its assigned host.	Every host has every user's counts, so a user can be routed to any host.
Scalable. We rely on the distributed database to serve both high read and high write traffic.	Scalable. A load balancer is an expensive and vertically-scalable component that can handle high request rate.	Not scalable, because each host needs to store the counts of every user. Need to divide users into separate instances of this service, and require another component (such as a frontend) to route users to their assigned instances.
Efficient storage consumption. We can configure our desired replication factor in our distributed database.	Lowest storage consumption, because there is no backup by default. We can design a storage service with an in-cluster or out-cluster approach, as discussed in section 13.4. Without backup, it is the cheapest approach.	Most expensive approach. High storage consumption. Also high network traffic from n-n communication between hosts to synchronize counts.
Eventually consistent. A host making a rate limiting decision may do so before synchronization is complete, so this decision may be slightly inaccurate.	Most accurate and consistent, since a user always makes requests to the same hosts.	Least accurate and consistent approach, as it takes time to synchronize counts between all hosts.
Backend is stateless, so we leverage the highly-available and fault-tolerant properties of the distributed database.	Without backup, any host failure will result in data loss of all the user counts it contains. This is the lowest availability and fault-tolerant of the 3 designs. However, these factors may be inconsequential as they are not non-functional requirements. If the rate limiter cannot obtain an accurate count, it can simply let the request through.	Hosts are interchangeable, so this is the most highly-available and fault-tolerant of the 3 designs.
Dependent on external database service. Outages of such services may impact our service, and	Not dependent on external database services. Load balancer needs to process every request to	Not dependent on external database services like Redis. Avoids risk of service outage from

remediating such outages may be outside our control.	determine which host to send it to. Also requires reshuffling to prevent hot shards.	outages of such downstream services. Also easier to implement, particularly in big organizations where provisioning or modifying database services may involve considerable bureaucracy.

8.8.2 Synchronizing counts

How can the hosts synchronize their user request counts? In this section, we discuss a few possible algorithms. All the algorithms except all-to-all are feasible for our rate limiter.

Should the synchronization mechanism be pull or push? We can choose push to trade off consistency and accuracy for higher performance, lower resource consumption, and lower complexity. If a host goes down, we can simply disregard its counts, and allow users to make more requests before they are rate limited. With these considerations, we can decide that hosts should asynchronously share their timestamps using UDP instead of TCP.

We should consider that hosts must be able to handle their traffic from these 2 main kinds of requests.

1. Request to make a rate limiting decision. Such requests are limited by the load balancer, and by provisioning a larger cluster of hosts as necessary.
2. Request to update the host's timestamps in memory. Our synchronization mechanism must ensure that a host does not receive a high rate of requests, especially as we increase the number of hosts in our cluster.

ALL-TO-ALL

All-to-all means every node transmits messages to every other node in a group. It is more general than *broadcasting*, which refers to simultaneous transfer of the message to recipients. Referring to figure 2.3 (repeated in figure 8.7), all-to-all requires a *full mesh* topology, where every node in a network is connected to every other node. All-to-all scales quadratically with the number of nodes, so it is not scalable. If we use all-to-all communication with 128 hosts, each all-to-call communication will require 128*128*64 MB which is > 1 TB, which is infeasible.

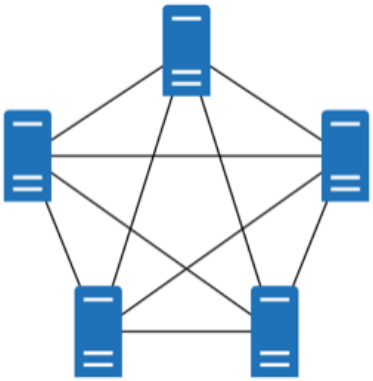

Figure 8.8: A full mesh topology. Every node in a network is connected to every other node. In our rate limiter, every node receives user requests, computes request rate, and approves or denies the request.

GOSSIP PROTOCOL

In *gossip protocol*, referring to figure 2.6 (repeated in figure 8.8), nodes periodically randomly pick other nodes and send them messages. Yahoo's distributed rate limiter uses gossip protocol to synchronize its hosts (https://yahooeng.tumblr.com/post/111288877956/cloud-bouncer-distributed-rate-limiting-at-yahoo). This approach trades off consistency and accuracy for higher performance and lower resource consumption. It is also more complex.

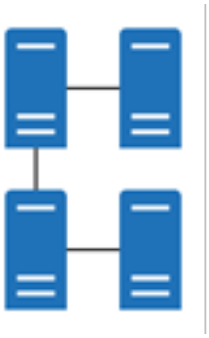

Figure 8.9: Gossip protocol. Each node periodically randomly picks other nodes and sends them messages.

In this section, all-to-all and gossip protocol are the synchronization mechanisms that require all nodes to send messages directly to each other. This means that all nodes must know the IP addresses of the other nodes. Since nodes are continuously added and removed from the cluster, each node will make requests to the configuration service (such as ZooKeeper) to find the other nodes' IP addresses.

In the other synchronization mechanisms, hosts make requests to each other via a particular host or service.

EXTERNAL STORAGE OR COORDINATION SERVICE

Referring to figure 8.9 (almost identical to figure 2.4), these 2 approaches use external components for hosts to communicate between each other.

Hosts can communicate with each other via a leader host. This host is selected by the cluster's configuration service (such as ZooKeeper). Each host only needs to know the IP address of the leader host, while the leader host needs to periodically update its list of hosts.

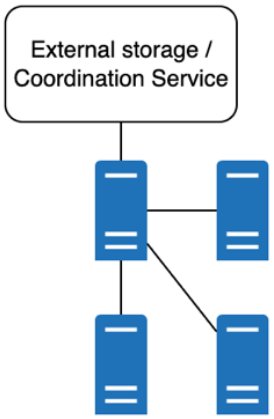

Figure 8.10: Hosts can communicate through an external component, such as an external storage service or coordination service.

RANDOM LEADER SELECTION

We can trade off higher resource consumption for lower complexity by using a simple algorithm to elect a leader. Referring to figure 2.7 (repeated in figure 8.10), this may cause multiple leaders to be elected. As long as each leader communicates with all other hosts, every host will be updated with the all the request timestamps. There will be unnecessary messaging overhead.

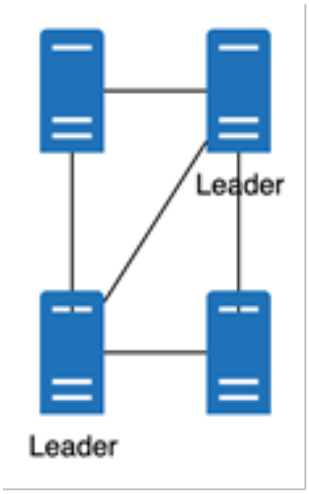

Figure 8.11: Random leader selection may cause multiple leaders to be elected. This will cause unnecessary messaging overhead, but does not present other issues.

8.9 Rate limiting algorithms

Up to this point, we have assumed that a user's request rate is made from its request timestamps, but have not actually discussed possible techniques to compute the request rate. At this point, one of the main questions is how our distributed rate limiting service determines the requestor's current request rate. Common rate limiting algorithms include the following.

- Token bucket
- Leaky bucket
- Fixed window counter
- Sliding window log
- Sliding window counter

Before we continue, we note that certain system design interview questions may seem to involve specialized knowledge and expertise that most candidates will not have prior experience with. The interviewer may not expect us to be familiar with rate limiting algorithms. This is an opportunity for her to assess the candidate's communication skills and learning ability. The interviewer may describe a rate limiting algorithm to us, and assess our ability to collaborate with her to design a solution around it that satisfies our requirements.

The interviewer may even make sweeping generalizations or erroneous statements, and assess our ability to critically evaluate them, and tactfully, firmly, clearly, and concisely ask intelligent questions and express our technical opinions.

We can consider implementing more than 1 rate limiting algorithm in our service, and allow each user of this service to choose a rate limiting algorithm to select the algorithm that

most closely suits the user's requirements. In this approach, a user selects the desired algorithm and sets the desired configurations in the Rules Service.

For simplicity of discussion, the discussions of the rate limiting algorithms in this section assume that the rate limit is 10 requests in 10 seconds.

8.9.1 Token bucket

Referring to figure 8.11, the token bucket algorithm is based on an analogy of a bucket filled with tokens. A bucket has 3 characteristics:

- A maximum number of tokens.
- Number of currently available tokens.
- A refill rate at which tokens are added to the bucket.

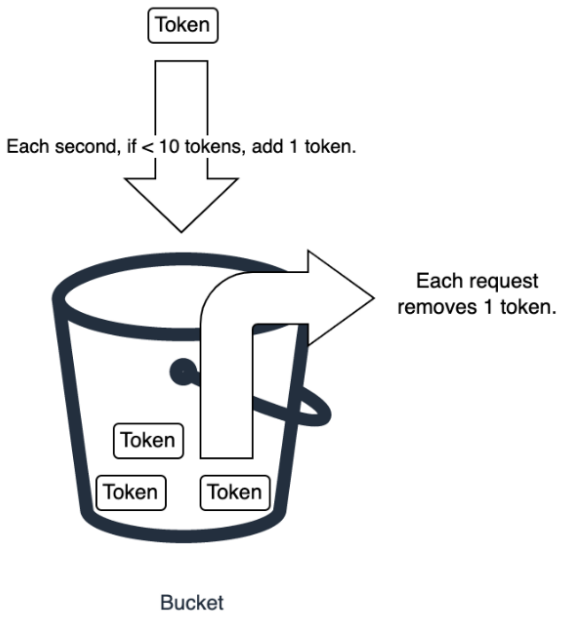

Figure 8.12: A token bucket that enqueues once per second.

Each time a request arrives, we remove a token from the bucket. If there are no tokens, the request is rejected or rate limited. The bucket is refilled at a constant rate.

In a straightforward implementation of this algorithm, the following occurs with each user request. A host can store key-value pairs using a hash map. If the host does not have a key for this user ID, initialize an entry with a user ID and a token count of 9 (10 – 1). Return false i.e. the user should not be rate limited. If the host has a key for this user ID and its value is more than 0, decrement its count. If the count is 0, return true i.e. the user should

be rate limited. Our system also needs to increment every value by 1 each second if it is less than 10.

The advantages of token bucket are that it is easy to understand and implement, and it is memory efficient (each user only needs a single integer variable to count tokens).

One obvious consideration with this implementation is that each host will need to increment every key in the hash map. It is feasible to do this on a hash map in a host's memory. If the storage is external to the host, such as on a Redis database. Redis provides the MSET (https://redis.io/commands/mset/) command to update multiple keys, but there may be a limit on the number of keys that can be updated in a single MSET operation (https://stackoverflow.com/questions/49361876/mset-over-400-000-map-entries-in-redis). (Stack Overflow is not an academically credible source, and the official Redis documentation on MSET does not state an upper limit on the number of keys in a request. However, when designing a system, we must always ask reasonable questions, and should not completely trust even official documentation.) Moreover, if each key is 64 bits, a request to update 10 million keys will be 8.08 GB in size, which is much too big.

If we need to divide the update command into multiple requests, each request incurs resource overhead and network latency.

Moreover, there is no mechanism to delete keys i.e. remove users who have not made recent requests, so the system doesn't know when to remove users to reduce the token refill request rate, or make room in the Redis database for other users who made recent requests. Our system will need a separate storage mechanism to record the last timestamp of a user's request, and a process to delete old keys.

In a distributed implementation like section 8.8, every host may contain its own token bucket, and use this bucket to make rate limiting decisions. Hosts may synchronize their buckets using techniques discussed in section 8.8.2. If a host makes a rate limiting decision using its bucket before synchronizing this bucket with other hosts, a user may be able to make requests at a higher rate than the set rate limit. For example, if 2 hosts each receive requests close in time, each one will subtract a token and have 9 tokens left, then synchronize with other hosts. Even though there were 2 requests, all hosts will synchronize to 9 tokens.

8.9.2 Leaky bucket

A leaky bucket has a maximum number of tokens, leaks at a fixed rate, and stops leaking when empty. Each time a request arrives, we add a token to the bucket. If the bucket is full, the request is rejected or rate limited.

Referring to figure 8.12, a common implementation of a leaky bucket is to use a FIFO queue with a fixed size. The queue is dequeued periodically. When a request arrives, a token is enqueued if the queue has spare capacity. Due to the fixed queue size, this implementation is less memory-efficient than token bucket.

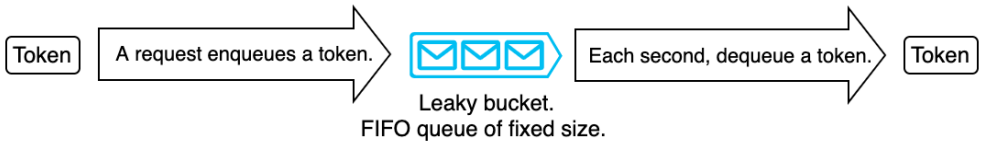

Figure 8.13: A leaky bucket that dequeues once per second.

This algorithm has some of the same issues as token bucket.

- Every second, a host needs to dequeue every queue in every key.
- We need a separate mechanism to delete old keys.
- A queue cannot exceed its capacity, so in a distributed implementation, there may be multiple hosts that simultaneously fill their buckets/queues fully before they synchronize. This means that the user exceeded its rate limit.

Another possible design is to enqueue timestamps instead of tokens. When a request arrives, we first dequeue timestamps until the remaining timestamps in the queue are older than our retention period, then enqueue the request's timestamp if the queue has space. Return false if the enqueue was successful and true otherwise. This approach avoids the requirement to dequeue from every single queue every second.

Do you notice any possible consistency issues?

An alert reader will immediately notice 2 possible consistency issues that will introduces inaccuracy into a rate limiting decision.

1. A race condition can occur where a host writes a key-value pair to the leader host, and it is immediately overwritten by another host.
2. Hosts' clocks are not synchronized, and a host may make a rate limiting decision using timestamps written by other hosts with slightly different clocks.

This slight inaccuracy is acceptable. These 2 issues also apply to all the distributed rate limiting algorithms discussed in this section that use timestamps, namely fixed window counter and sliding window log, and we will not mention them again.

8.9.3 Fixed window counter

Fixed window counters are implemented as key-value pairs. A key can be a combination of a client ID and a timestamp e.g. user0_1628825241, while the value is the request count. When a client makes a request, its key is incremented if it exists or created if it does not exist. The request is accepted if the count is within the set rate limit, and rejected if the count exceeds the set rate limit.

The window intervals are fixed. For example, a window can be between the [0, 60) seconds of each minute. After a window has passed, all keys expire. For example, the key

"user0_1628825241" is valid from 3:27:00 AM GMT to 3:27:59 AM GMT, because 1628825241 is 3:27:21 AM GMT, which is within the minute of 3:27 AM GMT.

How much may the request rate exceed the set rate limit?

A disadvantage of fixed window counter is that it may allow a request rate of up to twice the set rate limit. For example, referring to figure 8.13, if the rate limit is 5 requests in 1 minute, a client can make up to 5 requests in [8:00:00 AM, 8:01:00 AM) and up to another 5 requests in [8:01:00 AM, 8:01:30 AM). The client has actually made 10 requests in a 1-minute interval, twice the set rate limit of 5 requests per minute.

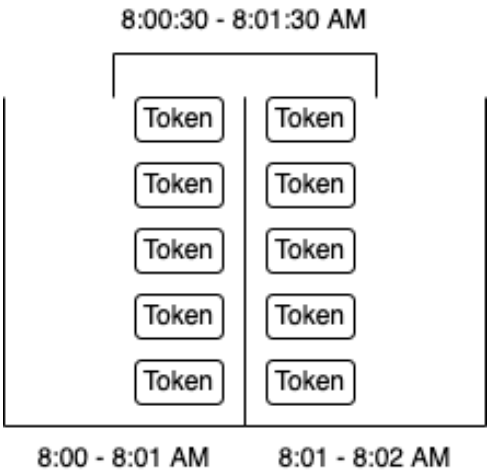

Figure 8.14: The user made 5 requests in [8:00:30 AM, 8:01:30 AM) and another 5 in [8:01:00 AM, 8:01:30 AM). Even though it was within the limit of 5 requests per fixed window, it actually made 10 requests in 1 minute.

Adapting this approach for our rate limiter, each time a host receives a user request, it takes these steps with its hash map. Refer to figure 8.14 for a sequence diagram of these steps.

1. Determine the appropriate keys to query. For example, if our rate limit had a 10-second expiry, the corresponding keys for user0 at 1628825250 will be ["user0_1628825241", "user0_1628825242", …, "user0_1628825250"].
2. Make requests for these keys. If we are storing key-value pairs in Redis instead of the host's memory, we can use the MGET (https://redis.io/commands/mget/) command to return the value of all specified keys. Although the MGET command is O(N) where N is the number of keys to retrieve, making a single request instead of multiple requests has lower network latency and resource overhead.

3. If no keys are found, create a new key-value pair e.g. (user0_1628825250, 1). If 1 key is found, increment its value. If more than 1 key is found (due to race conditions), sum the values of all the returned keys and increment this sum by 1. This is the number of requests in the last 10 seconds.
4. In parallel:
    a) Write the new or updated key-value pair to the leader host (or Redis database). If there were multiple keys, delete all keys except the oldest one.
    b) Return true if the count is more than 10 and false otherwise.

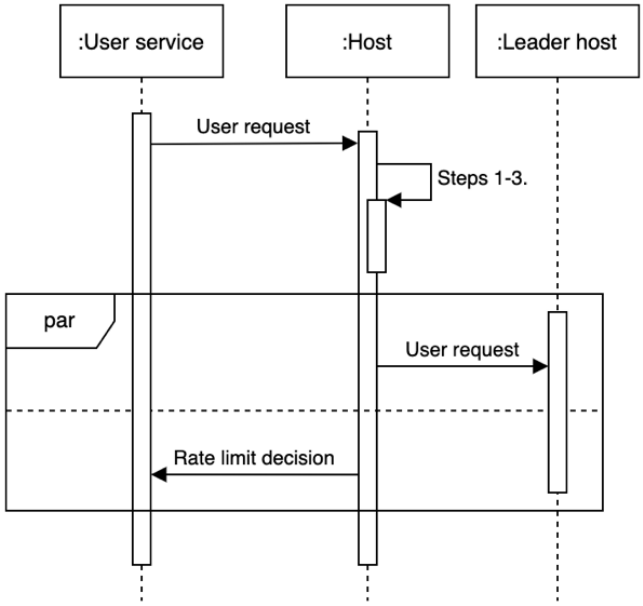

Figure 8.15: Sequence diagram of our fixed window counter approach. This diagram illustrates the approach of using the host's memory to store the request timestamps, instead of Redis. The rate limiting decision is made immediately on the host, using only data stored in the host's memory. The subsequent steps on the leader host for synchronization are not illustrated.

How may race conditions cause multiple keys to be found in step 5?

Redis keys can be set to expire (https://redis.io/commands/expire/), so we should set the keys to expire after 10 seconds. Otherwise, we will need to implement a separate process to continuously find and delete expired keys. If this process is needed, it is an advantage of fixed window counter that the key deletion process is independent from the hosts. This independent deletion process can be scaled separately from the host, and it can be developed independently, making it easier to test and debug.

8.9.4 Sliding window log

A sliding window log is implemented as a key-value pair for each client. The key is the client ID and the value is a sorted list of timestamps. A sliding window log stores a timestamp for each request.

Figure 8.15 is a simple illustration of sliding window log. When a new request comes in, we append its timestamp and check if the first timestamp is expired. If so, perform a binary search to find the last expired timestamp, then remove all timestamps before that. Use a list instead of a queue, because a queue does not support binary search. Return true if the list has more than 10 timestamps, and false otherwise.

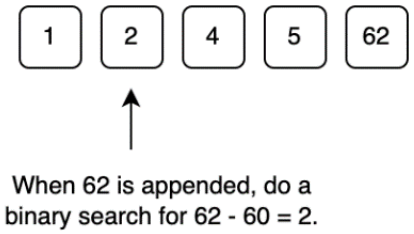

Figure 8.16: Simple sliding window log illustration. A timestamp is appended when a new request is made. Next, the last expired timestamp is found using binary search, then all expired timestamps are removed. The request is allowed if the size of the list does not exceed the limit.

Sliding window log is accurate (except in a distributed implementation, due to the factors discussed in the last paragraph of section 8.9.2), but storing a timestamp value for every request consumes more memory than a token bucket.

The sliding window log algorithm counts requests even after the rate limit is exceeded, so it also allows us to measure the user's request rate.

8.9.5 Sliding window counter

Sliding window counter is a further development of fixed window counter and sliding window log. It uses multiple fixed window intervals, and each interval is 1/60th the length of the rate limit's time window.

For example, if the rate limit interval is 1 hour, we use 60 1-minute windows instead of 1 1-hour window. The current rate is determined by summing the last 60 windows. It may slightly undercount requests. For example, counting requests at 11:00:35 will sum the 60 1-minute windows from the [10:01:00, 10:01:59] window to the [11:00:00, 11:00:59] window, and ignore the [10:00:00, 10:00:59] window. This approach is still more accurate than a fixed window counter.

8.10 Employing a sidecar pattern

This brings us to the discussion of applying sidecar pattern to rate limiting policies. Figure 1.8 illustrates our rate limiting service architecture using a sidecar pattern. Like what was discussed in section 1.4.6, an administrator can configure a user service's rate limiting policies in the Control Plane, which distributes them to the sidecar hosts. With this design where user services contain their rate limiting policies in their sidecar hosts, user service hosts do not need to make requests to our rate limiting service to look up their rate limiting policies, saving the network overhead of these requests.

8.11 Logging, monitoring, and alerting

Besides the logging, monitoring, and alerting practices discussed in section 6.5, we should configure monitoring and alerting for the following. We can configure monitoring tasks in our shared Monitoring Service on the logs in our shared Logging Service, and these tasks should trigger alerts to our shared Alerting Service to alert developers about these issues:

- Signs of possible malicious activity, such as users which continue to make requests at a high rate despite being shadow banned.
- Signs of possible DDoS attempts, such as an unusually high number of users being rate limited in a short interval.

8.12 Providing functionality in a client library

Does a user service need to query the Rate Limiter Service for every request? An alternative approach is for the user service to aggregate user requests, then query the Rate Limiter Service in certain circumstances such as when it:

- Accumulates a batch of user requests.
- *Notices a sudden increase in request rate.*

Generalizing this approach, can rate limiting be implemented as a library instead of a service? Section 5.7 is a general discussion of a library vs service. If we implement it entirely as a library, we will need to use the approach in section 8.7, where a host can contain all user requests in memory, and synchronize the user requests with each other. Hosts must be able to communicate with each other to synchronize their user request timestamps, so the developers of the service using our library must configure a configuration service like ZooKeeper. This may be overly complex and error-prone for most developers, so as an alternative, we can offer a library with features to improve the performance of the Rate Limiting Service, by doing some processing on the client hence allowing a lower rate of requests to the service.

This pattern of splitting processing between client and server can generalize to any system, but it may cause tight-coupling between the client and server, which is in general an antipattern. The development of the server application must continue to support old client versions for a long time. For this reason, a client SDK (software development kit) is usually just a layer on a set of REST or RPC endpoints, and does not do any data processing. If we

wish to do any data processing in the client, at least one of the following conditions should be true.

- The processing should be simple, so it is easy to continue to support this client library in future versions of the server application.
- The processing is resource-intensive, so the maintenance overhead of doing such processing on the client is a worthy tradeoff for the significant reduction in the monetary cost of running the service.
- There should be a stated support lifecycle that clearly informs users when the client will no longer be supported.

Regarding batching of requests to the rate limiter, we can experiment with batch size to determine the best balance between accuracy and network traffic.

What if the client also measures the request rate, and only uses the Rate Limiting Service if the request rate exceeds a set threshold? An issue with this is that since clients do not communicate with each other, a client can only measure the request rate on the specific host its installed on, and cannot measure the request rate of specific users. This means that rate limiting is activated based on the request rate across all users, not on specific users. Users may be accustomed to a particular rate limit, and may complain if they are suddenly rate limited at a particular request rate where they were not rate limited before.

An alternative approach is for the client to use anomaly detection to notice a sudden increase in the request rate, then start sending rate limiting requests to the server.

8.13 Further reading

- Smarshchok, Mikhail (2019) System Design Interview YouTube channel, https://youtu.be/FU4WlwfS3G0
- The discussions of Fixed Window Counter, Sliding Window Log, and Sliding Window Counter were adapted from https://www.figma.com/blog/an-alternative-approach-to-rate-limiting/.
- Madden, Neil (2020) API Security in Action. Manning Publications.
- Posta, Christian E. and Maloku, Rinor (2022) Istio in Action. Manning Publications.
- Bruce, Morgan and Pereira, Paulo A. (2018) Microservices in Action, chapter 3.5. Manning Publications

8.14 Summary

- Rate limiting prevents service outages and unnecessary costs.
- Alternatives such as adding more hosts or using the load balancer for rate limiting are infeasible. Adding more hosts to handle traffic spikes may be too slow, while using a level 7 load balancer just for rate limiting may add too much cost and complexity.
- Do not use rate limiting if it results in poor user experience, or for complex use cases such as subscriptions.
- The non-functional requirements of a rate limiter are scalability, performance, and lower complexity. To optimize for these requirements, we can trade off availability, fault-tolerance, accuracy, and consistency.

- The main input to our rate limiter service is user ID and service ID, which will be processed according to rules defined by our admin users to return a "yes" or "no" response on rate limiting.
- There are various rate limiting algorithms, each with its own tradeoffs. Token bucket is easy to understand and implement, and is memory-efficient, but synchronization and cleanup are tricky. Leaky bucket is easy to understand and implement, but is slightly inaccurate. Fixed window log is easy to test and debug, but inaccurate and more complicated to implement. Sliding window log is accurate, but requires more memory. Sliding window counter uses less memory than sliding window log, but is less accurate than sliding window log.
- We can consider a sidecar pattern for our rate limiting service.