

# Building Scalable & Reliable Systems

Tejas Parikh ([t.parikh@northeastern.edu](mailto:t.parikh@northeastern.edu))

CSYE 7125: Advanced Cloud Computing  
Northeastern University

# Slashdot effect

- The Slashdot effect, also known as *slashdotting*, occurs when a popular website links to a smaller website, causing a massive increase in traffic.
- This overloads the smaller site, causing it to slow down or even temporarily become unavailable.
- This has the same effect as a denial-of-service attack, albeit accidentally.
- [https://en.wikipedia.org/wiki/Slashdot\\_effect](https://en.wikipedia.org/wiki/Slashdot_effect)

# Functional vs Non-Functional Requirements

# Functional vs Non-Functional Requirements

Functional Requirements	Non Functional Requirements
A functional requirement defines a system or its component.	A non-functional requirement defines the quality attribute of a software system.
It specifies "What should the software system do?"	It places constraints on "How should the software system fulfill the functional requirements?"
Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
It is mandatory.	It is not mandatory.
It is captured in use case.	It is captured as a quality attribute.
Defined at a component level.	Applied to a system as a whole.
Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.
Usually easy to define.	Usually more difficult to define.
<b>Example</b> <b>1)</b> Authentication of user whenever he/she logs into the system. <b>2)</b> System shutdown in case of a cyber attack. <b>3)</b> A Verification email is sent to user whenever he/she registers for the first time on some software system.	<b>Example</b> <b>1)</b> Emails should be sent with a latency of no greater than 12 hours from such an activity. <b>2)</b> The processing of each request should be done within 10 seconds <b>3)</b> The site should load in 3 seconds when the number of simultaneous users are > 10000

# Foundations of Scalable Systems

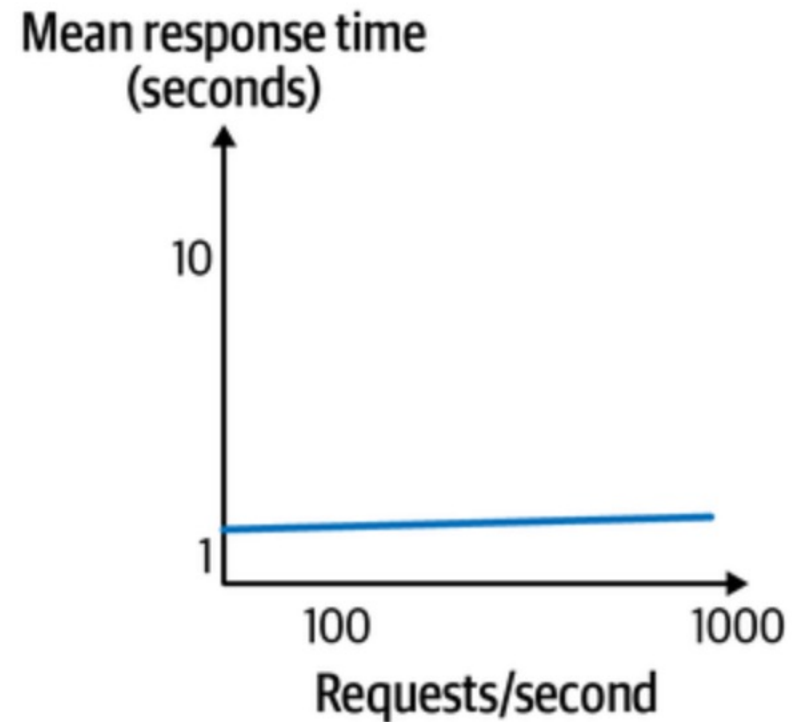
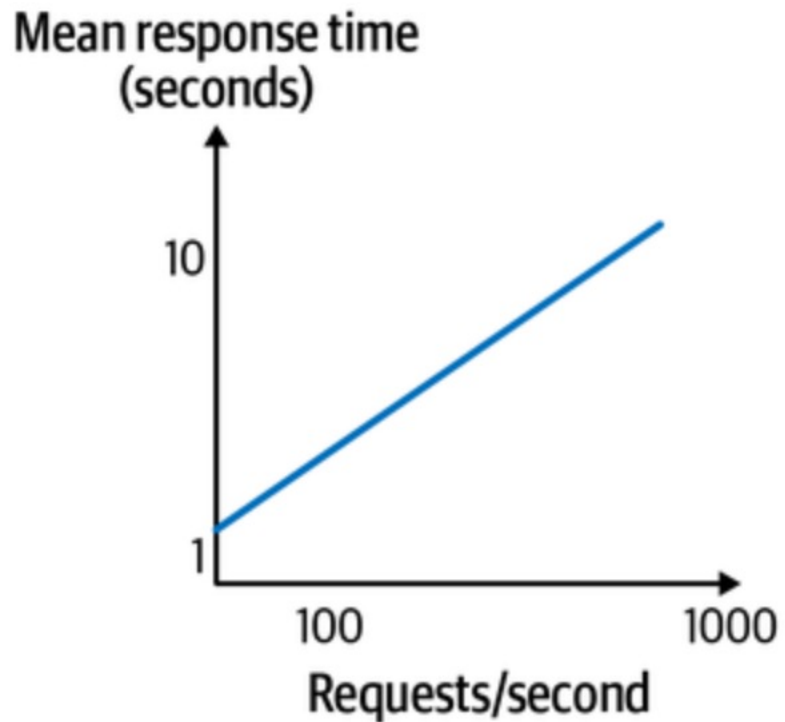
# What Is Scalability?

- Wikipedia page defines Scalability as the property of a system to handle a growing amount of work by adding resources to the system.
- Scalability defines a software system's capability to handle growth in some dimension of its operations. Examples of operational dimensions are:
  - The number of simultaneous user or external requests a system can process
  - The amount of data a system can effectively process and manage
  - The value that can be derived from the data a system stores through predictive analytics
  - The ability to maintain a stable, consistent response time as the request load grows

# Scalability Basic Design Principles

- The basic aim of scaling a system is to increase its capacity in some application-specific dimension.
- The two basic design principles
  - Replication
  - Optimization

# Scalability and Costs





# Scaling a system isn't easy

- The reason for this are many such as
  - The database becomes less responsive with 1,000 requests per second, requiring an upgrade to a new machine.
  - The web server generates a lot of content dynamically and this reduces response time under load. A possible solution is to alter the code to more efficiently generate the content, thus reducing processing time per request.
  - The request load creates hotspots in the database when many requests try to access and update the same records simultaneously. This requires a schema redesign and subsequent reloading of the database, as well as code changes to the data access layer.
  - The web server framework that was selected emphasized ease of development over scalability. The model it enforces means that the code simply cannot be scaled to meet the requested load requirements, and a complete rewrite is required. Use another framework? Use another programming language even?

# Scalability and Architecture Trade-Offs

- One of the enduring complexities of software architecture is the necessity of quality attribute trade-offs.
- Basically, a design that favors one quality attribute may negatively or positively affect others.
- Software architects constantly tread a fine line, crafting their designs to satisfy high-priority quality attributes, while minimizing the negative effects on other quality attributes.
- When designing a scalable system, we have to carefully consider how our design influences other highly desirable properties such as performance, availability, security, and the often-overlooked manageability.

# Performance

- When we target performance, we attempt to satisfy some desired metrics for individual requests. Example: The mean response time is less than 2 seconds.
- Improving performance is in general a good thing for scalability.

# Availability

- Availability and scalability are in general highly compatible partners.
- As we scale our systems through replicating resources, we create multiple instances of services that can be used to handle requests from any users.
- If one of our instances fails, the others remain available. The system just suffers from reduced capacity due to a failed, unavailable resource.

# Stateful Services & Availability

- Things get complicated with scalability and availability when state is involved.
- Consider scaling database. Easy to scale for READs, harder to scale for WRITEs.
- Whenever state is replicated for scalability and availability, we have to also deal with consistency.

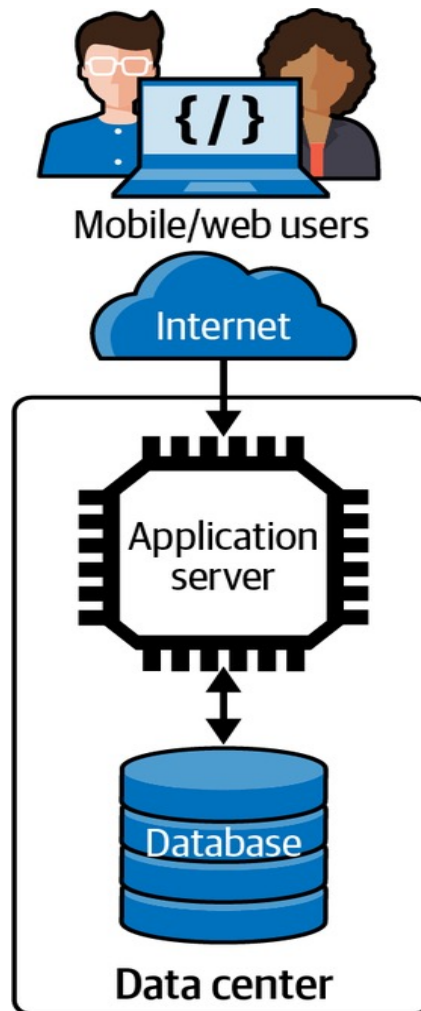
# Security

- The basic elements of a secure system are authentication, authorization, and integrity.
- [The New Enemy Problem](#)
- We need to ensure data cannot be intercepted in transit over networks, and data at rest (persistent store) cannot be accessed by anyone who does not have permission to access that data.
- In general, security and scalability are opposing forces. Security necessarily introduces performance degradation. The more layers of security a system encompasses, then a greater burden is placed on performance, and hence scalability. This eventually affects the bottom line—more powerful and expensive resources are required to achieve a system's performance and scalability requirements.

# Manageability

- As the systems we build become more distributed and complex in their interactions, their management and operations come to the fore.
- We need to pay attention to ensuring every component is operating as expected, and the performance is continuing to meet expectations.
- The term used for this sophisticated monitoring capability is observability.
- The only way to control the costs and complexity of manageability as we scale is through automation. This is where the world of DevOps enters the scene.
- DevOps is a set of practices and tooling that combine software development and system operations.

# Basic System Architecture



- Virtually all massive-scale systems start off small and grow due to their success.
- A typical very simple software architecture for “starter” systems comprises a client tier, application service tier, and a database tier.
- model–view–controller (MVC) pattern



# The model–view–controller (MVC) Pattern

- **Model** - The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.
- **View** - Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
- **Controller** - Accepts input and converts it to commands for the model or view

# Scaling Up / Vertical Scaling

- Add more resources to a system so that it meets demand.



Vertical Scaling  
(Scaling up)

# Scaling Out / Horizontal Scaling

- Scaling out relies on the ability to replicate a service in the architecture and run multiple copies on multiple server nodes.
- Requests from clients are distributed across the replicas so that in theory, if we have  $N$  replicas and  $R$  requests, each server node processes  $R/N$  requests.
- This simple strategy increases an application's capacity and hence scalability.



Horizontal Scaling  
(Scaling out)

# Scaling Out

- To successfully scale out an application, you need two fundamental elements in your design.
  - A load balancer
  - Stateless services

# Scaling Out - A load balancer

- All user requests are sent to a load balancer, which chooses a service replica target to process the request.
- Various strategies exist for choosing a target service, all with the core aim of keeping each resource equally busy.
- The load balancer also relays the responses from the service back to the client.
- Most load balancers belong to a class of internet components known as **reverse proxies**. These control access to server resources for client requests. As an intermediary, reverse proxies add an extra network hop for a request; they need to be extremely low latency to minimize the overheads they introduce.

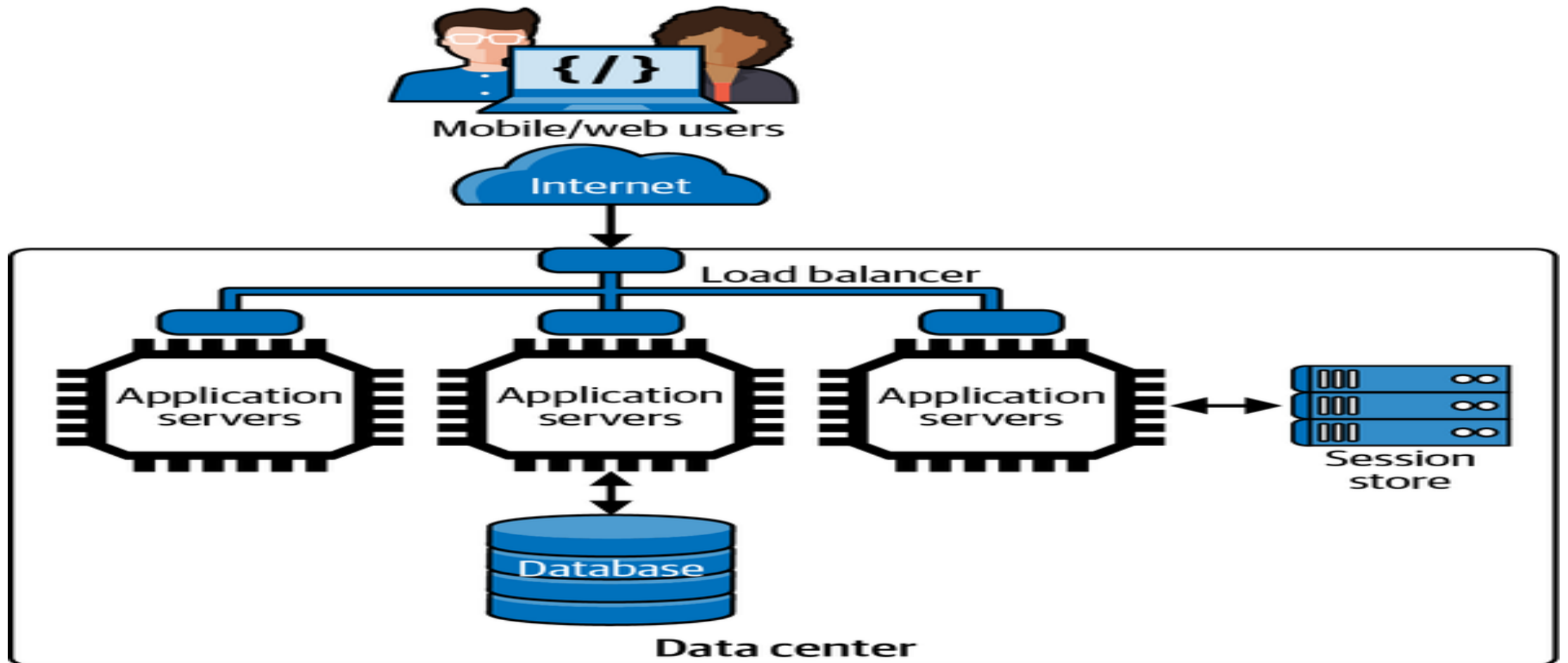
# Scaling Out - Stateless services

- For load balancing to be effective and share requests evenly, the load balancer must be free to send consecutive requests from the same client to different service instances for processing. This means the API implementations in the services must retain no knowledge, or state, associated with an individual client's session.
- When a user accesses an application, a user session is created by the service and a unique session is managed internally to identify the sequence of user interactions and track session state. A classic example of session state is a shopping cart. To use a load balancer effectively, the data representing the current contents of a user's cart must be stored somewhere—typically a data store—such that any service replica can access this state when it receives a request as part of a user session.

# Scaling Out

- Scaling out is attractive as, in theory, you can keep adding new (virtual) hardware and services to handle increased request loads and keep request latencies consistent and low. As soon as you see latencies rising, you deploy another server instance. This requires no code changes with stateless services and is relatively cheap as a result—you just pay for the hardware you deploy.
- Scaling out has another highly attractive feature. If one of the services fails, the requests it is processing will be lost. But as the failed service manages no session state, these requests can be simply reissued by the client and sent to another service instance for processing. This means the application is resilient to failures in the service software and hardware, thus enhancing the application's availability.
- Unfortunately, as with any engineering solution, simple scaling out has limits. As you add new service instances, the request processing capacity grows, potentially infinitely. At some stage, however, reality will bite and the capability of your single database to provide low-latency query responses will diminish. Slow queries will mean longer response times for clients. If requests keep arriving faster than they are being processed, some system components will become overloaded and fail due to resource exhaustion, and clients will see exceptions and request timeouts. Essentially, your database becomes a bottleneck that you must engineer away in order to scale your application further.

# Scale-out Architecture

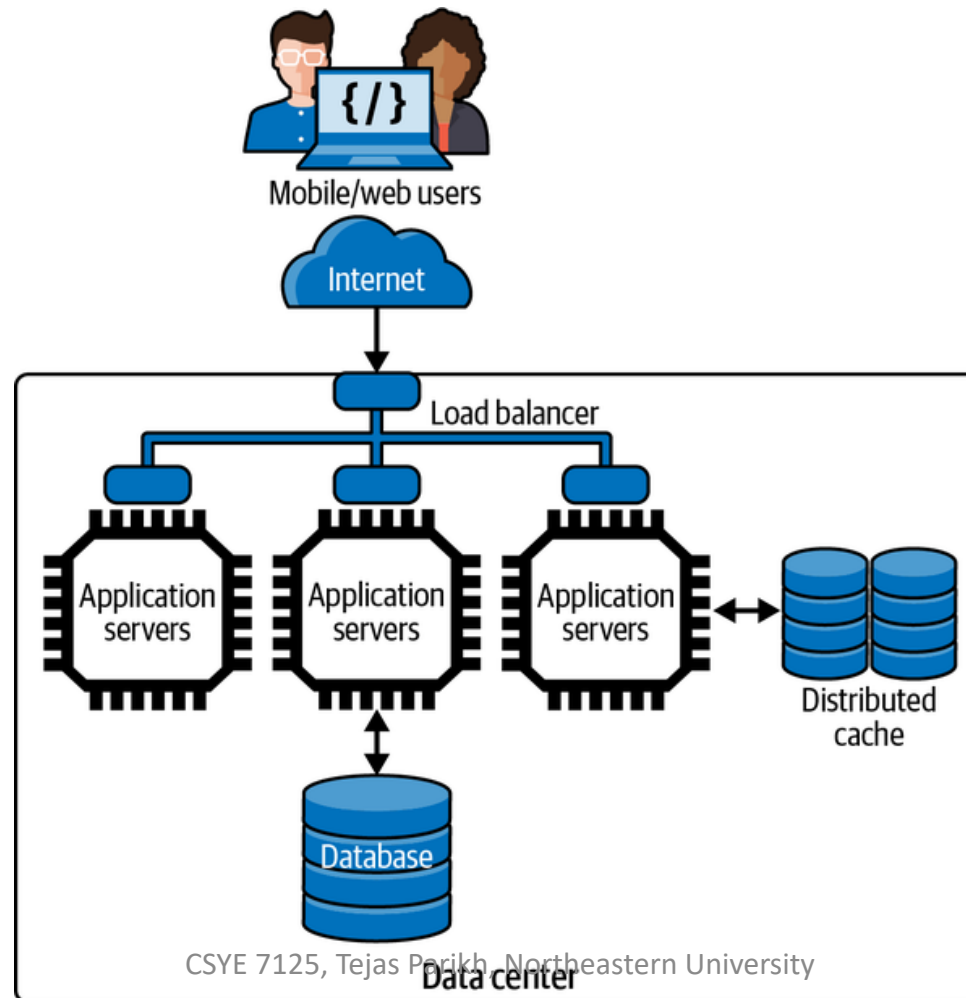




# Scaling the Database with Caching

- Scaling up by increasing the number of CPUs, memory, and disks in a database server can go a long way to scaling a system.
- In conjunction with scaling up, a highly effective approach is querying the database as infrequently as possible from your services.
- This can be achieved by employing ***distributed caching*** in the scaled-out service tier.
- Caching stores recently retrieved and commonly accessed database results in memory so they can be quickly retrieved without placing a burden on the database.

# Distributed Caching

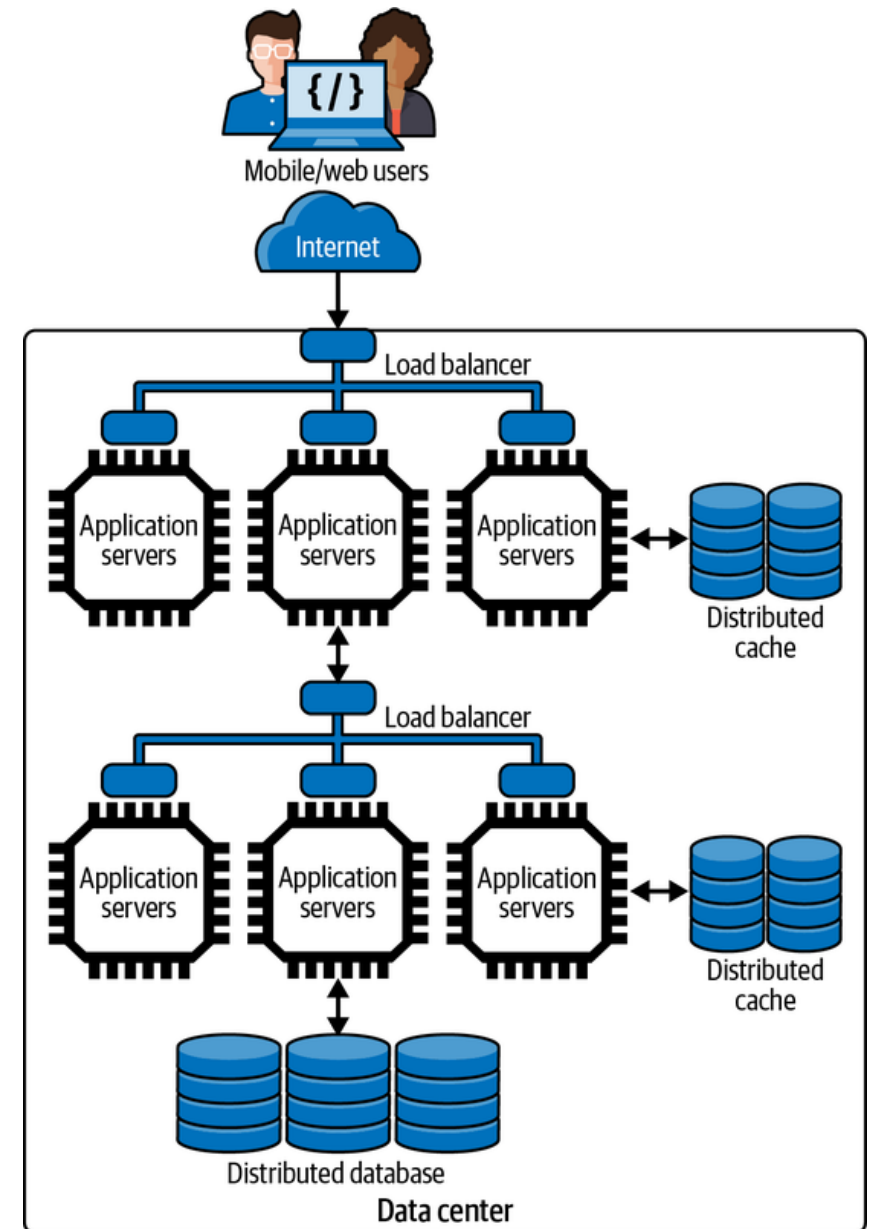


# Distributed Caching

- For data that is frequently read and changes rarely, your processing logic can be modified to first check a distributed cache.
- Introducing a caching layer also requires your processing logic to be modified to check for cached data.
- If what you want is not in the cache, your code must still query the database and load the results into the cache as well as return it to the caller.
- You also need to decide when to remove, or invalidate, cached results—your course of action depends on the nature of your data and your application's tolerance to serving out-of-date—also known as stale—results to clients.
- A well-designed caching scheme can be invaluable in scaling a system. Caching works great for data that rarely changes and is accessed frequently.

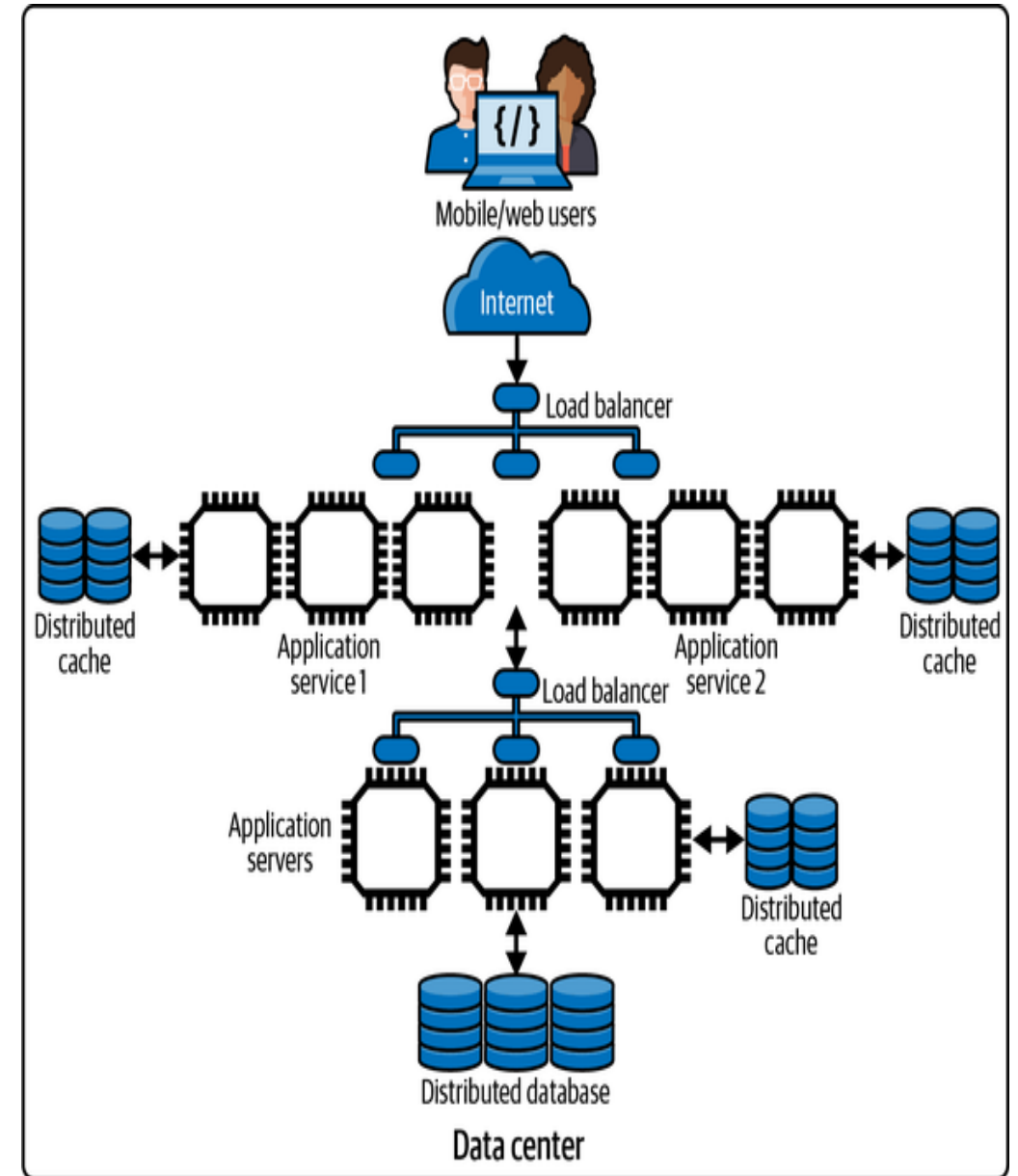
# Multiple Processing Tiers

Any realistic system that you need to scale will have many different services that interact to process a request.



# Multiple Processing Tiers

By breaking the application into multiple independent services, you can scale each based on the service demand.



# Increasing Responsiveness

- Most client application requests expect a response.
- The client sends a request and waits until a response is received. This time interval between sending the request and receiving the result is the response time of the request.
- We can decrease response times by using caching and precalculated responses, but many requests will still result in database accesses.
- Use queues for asynchronous processing - The data about the event is sent to the service, which acknowledges receipt and concurrently stores the data in a remote queue for subsequent writing to the database. Distributed queueing platforms can be used to reliably send data from one service to another, typically but not always in a first-in, first-out (FIFO) manner.
- Whenever the results of a write operation are not immediately needed, an application can use this approach to improve responsiveness and, as a result, scalability.

# Understanding, Measuring, and Improving Your Availability

# Availability

- No one cares whether your system has great features if they can't use it.
- One of the most important topics in architecting for scalable systems is availability.



# Availability Versus Reliability

- Availability and reliability are two similar yet very different concepts.
- Reliability - The ability of your system to perform the operations it is intended to perform without making a mistake.
- Availability - The ability of your system to be operational when needed in order to perform those operations.

# Causes of Poor Availability

- **Resource exhaustion** - Increase the number of users or increase the amount of data in use in a system and your application may fall victim to resource exhaustion, resulting in a slower and unresponsive application.
- **Unplanned load-based changes** - Increases in the popularity of your application might require code and application changes to handle the increased load. These changes, often implemented quickly and at the last minute with little or no forethought or planning, increase the likelihood of problems occurring.
- **Increased number of moving parts** - As an application gains popularity, it is often necessary to assign more and more developers, designers, testers, and other individuals to work on and maintain it. This larger number of individuals working on the application creates a large number of moving parts, whether those moving parts are new features, changed features, or just general application maintenance. The more individuals working on the application, the more moving parts within the application and the greater the chance for bad interactions to occur in it.
- **Outside dependencies** - The more dependencies your application has on external resources, such as SaaS services, infrastructure, or cloud-based services, the more it is exposed to availability problems caused by those resources.
- **Technical debt** - Increases in the applications complexity typically increase technical debt (i.e., the accumulation of desired software changes and pending bug fixes that often build up over time as an application grows and matures). Technical debt increases the likelihood of a problem occurring.

# Availability Problems

- All fast-growing applications have one, some, or all of these problems.
- As such, potential availability problems can begin occurring in applications that previously performed flawlessly.
- The problems can quietly creep up on you, or the problems may start suddenly without warning.
- But most growing applications will eventually begin having availability problems.
- Building applications designed to scale means building applications designed for high availability.

# Measuring Availability

- Measuring availability is important to keeping your system highly available.
- Only by measuring availability can you understand how your application is performing now and examine how your application's availability changes over time.
- The most widely held mechanism for measuring the availability of a web application is calculating the percent of time it's accessible for use by customers.

$$\text{Site availability percentage} = \frac{\text{total\_seconds\_in\_period} - \text{seconds\_system\_is\_down}}{\text{total\_seconds\_in\_period}}$$

# The Nines

- Often you will hear availability described as “the nines.”
- An application that has “2 nines” availability must be available 99% of the time. This means in a typical month it can be down for 432 minutes and still meet the 99% available goal.
- A “4 nines” application must be available 99.99% of the time, meaning it can be down a mere four minutes in a typical month.

Nines	Percentage	Monthly outage
2 nines	99%	432 minutes
3 nines	99.9%	43 minutes
4 nines	99.99%	4 minutes
5 nines	99.999%	26 seconds
6 nines	99.9999%	2.6 seconds

# Automate Your Manual Processes

- To maintain high availability, you need to remove unknowns and variables. Performing manual operations is a common way to insert variable results and/or unknown results into your system.
- DevOps anyone?

# Using only repeatable tasks gives you the following

- The ability to test a task before implementing it. Testing what happens when you make a specific change is critical to avoiding mistakes that cause outages.
- The ability to tweak the task to perform exactly what you want it to do. This lets you implement improvements to the change you are about to make before you actually make the change.
- The ability to have the task reviewed by a third party. This increases the likelihood that the task will have no unexpected side effects.
- The ability to put the task under version control. Version control systems allow you to determine when the task is changed, by whom, and for what reasons.
- The ability to apply the task to related resources. Making a change to a single server that improves how that server works is great. Being able to apply the same change to every affected server in a consistent way makes the task even more useful.
- The ability to have all related resources act consistently. If you continuously make “one-off” changes to resources such as servers, the servers will begin to drift and act differently from one another. This makes it difficult to diagnose problematic servers because there will be no baseline of operational expectation that you can use for comparison.
- The ability to implement repeatable tasks. Repeatable tasks are auditable tasks. Auditable tasks are tasks that you can analyze later for their impact, positive or negative, on the system as a whole.

# Five Focuses to Improve Application Availability

- Build with failure in mind
- Always think about scaling
- Mitigate risk
- Monitor availability
- Respond to availability issues in a predictable and defined way



# Focus #1: Build with Failure in Mind

- Design
  - What design constructs and patterns have you considered or are you using that will help improve the availability of your software?
  - Using design constructs and patterns, such as simple error catching deep within your application, retry logic, and circuit breakers, allows you to catch errors when they have affected the smallest available subset of functionality. This allows you to limit the scope of a problem and have your application still provide useful capabilities even if part of the application is failing.
- Dependencies
  - What do you do when a component you depend on fails? How do you retry? What do you do if the problem is an unrecoverable (hard) failure, rather than a recoverable (soft) failure?

# Focus #1: Build with Failure in Mind

- Customers
  - What do you do when a component that is a customer of your system behaves poorly? Can you handle excessive load on your system? Can you throttle excessive traffic? Can you handle garbage data passed in? What about excessive data?

# Focus #2: Always Think About Scaling

- As you build your system, don't build it for today's traffic; build it for tomorrow's traffic.
- Specifically, this might mean:
  - Architect in the ability to increase the size and capacity of your databases.
  - Think about what logical limits exist to your data scaling. What happens when your database tops out in its capabilities? Identify and remove these limits before your usage approaches them.
  - Build your application so that you can add additional application servers easily. This often involves being observant of where and how state is maintained and of how traffic is routed.
  - Redirect static traffic to offline providers. This allows your system to deal only with the dynamic traffic that it is designed to deal with. Using external content delivery networks (CDNs) not only can reduce the traffic your network has to handle but also allows the efficiencies of scale that CDNs provide to get that static content to your customers more quickly.
  - Think about whether specific pieces of dynamic content can actually be generated statically. Often, content that appears dynamic is actually mostly static, and the scalability of your application can be increased by making this content static. This “dynamic that can be static” data is sometimes hidden where you don't expect it.

# Focus #3: Mitigate Risk

- Keeping a system highly available requires removing risk from the system.
- Often the cause of a system failure could have been identified as a risk before the failure actually occurred.
- Identifying risk is a key method of increasing availability.
- All systems have risk in them. There is risk that:
  - A server will crash
  - A database will become corrupted
  - A returned answer will be incorrect
  - A network connection will fail
  - A newly deployed piece of software will fail
  - Keeping a system available requires removing risk.

# Hacker News Outage


**Hacker News Status**  @HNStatus · Jul 8 


HN is running again, on a new server created from backups. Will be monitoring closely. Sorry everyone for the inconvenience!

 6  14  172 

**Hacker News Status**  @HNStatus · Jul 8 

Working on restoring from backups the slow way, since failover server also failed. No ETA yet, but we're getting there.

 29  37  270 

**Hacker News Status**  @HNStatus · Jul 8 

HN is down again with what appears to be a second disk failure. This is on the fallback server, which we switched to last night when the primary server failed. Will update when we know more. Sorry all!

 57  126  587 

# Cloud Outages

## Top stories :

 Reuters

Google Cloud data center in London faces outage on UK's hottest day

1 day ago




 Data Center Knowledge

Google, Oracle Data Centers Knocked Offline by London Heat

1 day ago



 Data Center Dynamics

Cooling failure brings down Google Cloud data center in London on UK's hottest day

1 day ago



# Focus #4: Monitor Availability

- You can't know if there is a problem in your application unless you can see a problem.
- Proper monitoring depends on the specifics of your application and needs, but it usually entails some of the following capabilities:
  - Server monitoring - To monitor the health of your servers and make sure they keep operating efficiently.
  - Configuration change monitoring - To monitor your system configuration and identify if and when changes to your infrastructure impact your application.
  - Application performance monitoring - To look inside your application and services to make sure they are operating as expected.
  - Synthetic testing - To examine in real time how your application is functioning from the perspective of your users, in order to catch problems customers might see before they actually see them.
  - Alerting - To inform appropriate personnel when a problem occurs so that it can be quickly and efficiently resolved, minimizing the impact on your customers.

# Focus #5: Respond to Availability Issues in a Predictable and Defined Way

- Monitoring systems are useless unless you are prepared to act on the issues that arise.
- This means being alerted when problems occur so that you can take action.
- Additionally, you should establish processes and procedures that your team can follow to help diagnose issues and easily fix common failure scenarios.
- Playbooks / Runbooks



# Back of the Envelope Estimates

# Latency numbers every programmer should know

## Latency Comparison Numbers

L1 cache reference	0.5	ns		
Branch mispredict	5	ns		
L2 cache reference	7	ns		
Mutex lock/unlock	25	ns		
Main memory reference	100	ns		
Compress 1K bytes with Zippy	10,000	ns	10	us
Send 1 KB bytes over 1 Gbps network	10,000	ns	10	us
Read 4 KB randomly from SSD*	150,000	ns	150	us
Read 1 MB sequentially from memory	250,000	ns	250	us
Round trip within same datacenter	500,000	ns	500	us
Read 1 MB sequentially from SSD*	1,000,000	ns	1,000	us    1 ms
HDD seek	10,000,000	ns	10,000	us    10 ms
Read 1 MB sequentially from 1 Gbps	10,000,000	ns	10,000	us    10 ms
Read 1 MB sequentially from HDD	30,000,000	ns	30,000	us    30 ms
Send packet CA→Netherlands→CA	150,000,000	ns	150,000	us    150 ms

## Notes

1 ns =  $10^{-9}$  seconds

1 us =  $10^{-6}$  seconds = 1,000 ns

1 ms =  $10^{-3}$  seconds = 1,000 us = 1,000,000 ns

# Latency Numbers Every Programmer Should Know

■ 1 ns

■ L1 cache reference: 0.5 ns

■ Branch mispredict: 5 ns

■ L2 cache reference: 7 ns

■ Mutex lock/unlock: 25 ns

■ = 100 ns

■ Main memory reference: 100 ns

■ = 1  $\mu$ s

■ Compress 1KB with Zippy: 3  $\mu$ s

■ = 10  $\mu$ s

■ Send 1KB over 1Gbps network: 10  $\mu$ s

■ SSD random read (1Gb/s SSD): 150  $\mu$ s

■ Read 1MB sequentially from memory: 250  $\mu$ s

■ Round trip in same datacenter: 500  $\mu$ s

■ = 1 ms

■ Read 1MB sequentially from SSD: 1 ms

■ Disk seek: 10 ms

■ Read 1MB sequentially from disk: 20 ms

■ Packet roundtrip CA to Netherlands: 150 ms

# Additional Resources

See Lecture Page