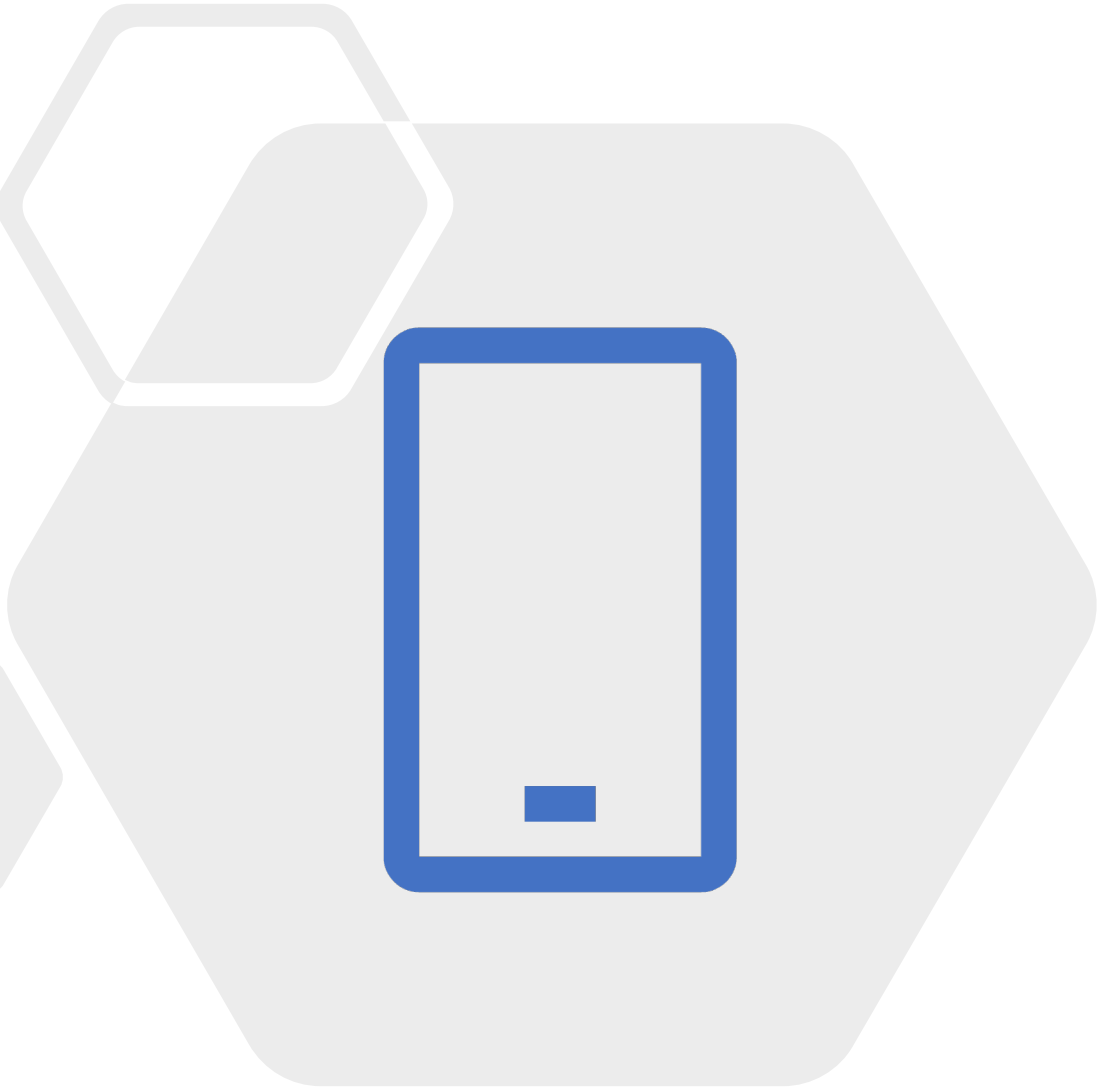
Performance Optimization for Mobile Applications

Written by Thapanapong Rukkanchanunt



Outline

- Mobile Performance
- Performance Metrics
- Optimizing Application Design
- Code Optimization

The Need of Mobile Applictions

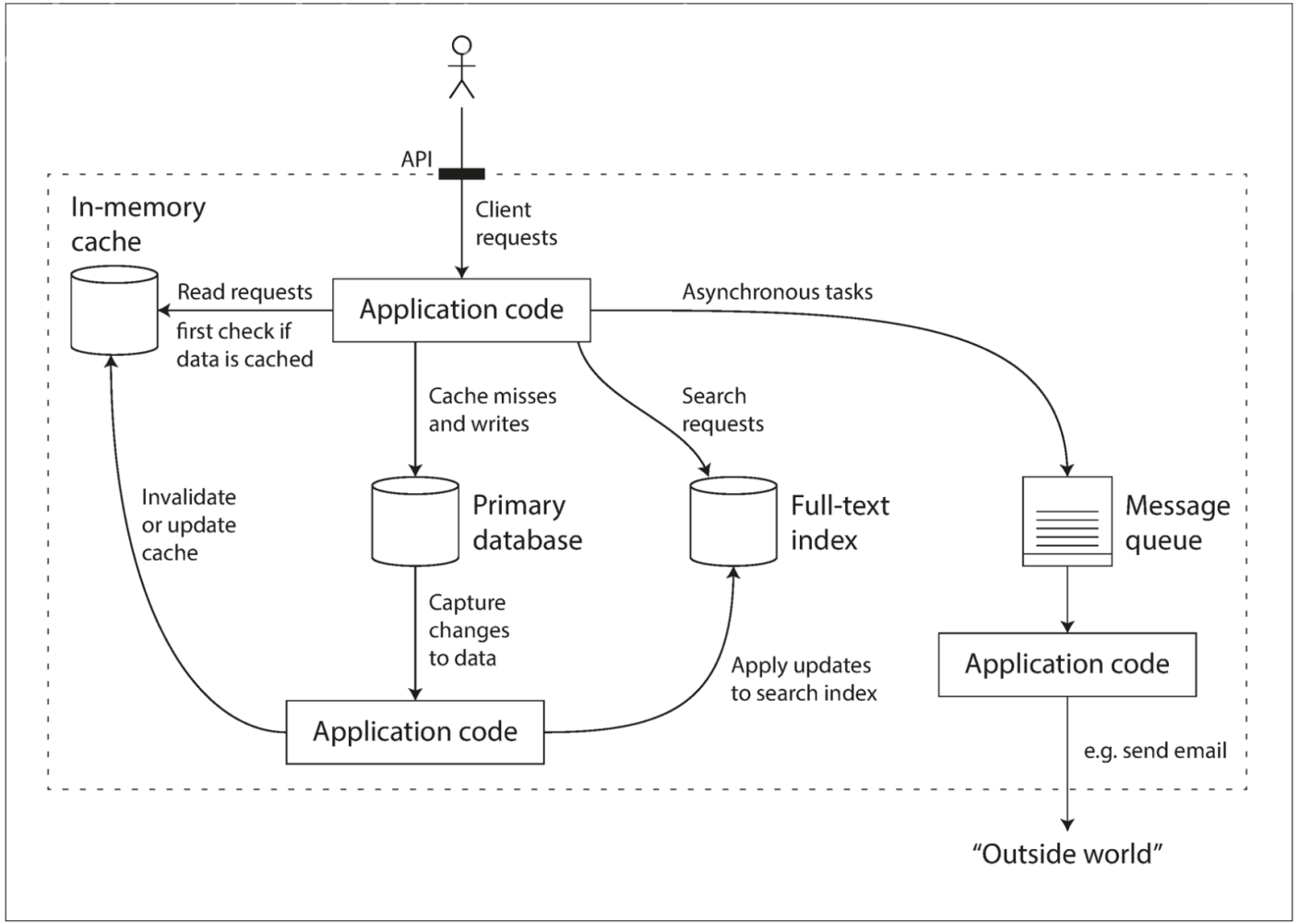
- Most applications are data-intensive
- CPU is rarely an issue (unless it’s a game)
- Commonly shared functions in all application
  - Store data to be searched later (databases)
  - Remember the result of an expensive operation (caches)
  - Allow users to search data by keyword (search indexes)
  - Send a message to another process, to be handled asynchronously (stream)
  - Periodically process a large amount of accumulated data (batch)

Reality is not simple

Database systems for data storage	• Each application has its own requirements
Many data storage tools	• Each database system has its own characteristics
No one-fit-all solution	• Several ways to cache data • Several ways to index data

One Possible Architecture

- What kind of code do we need to write?



Mobie Performance

Reliability

- Work correctly even if other fails

Scalability

- Dealing with growth

Maintainability

- For future generation

Reliability

Work correctly even if other fails

Reliability

Reliability means “continuing to work correctly, even when things go wrong.”

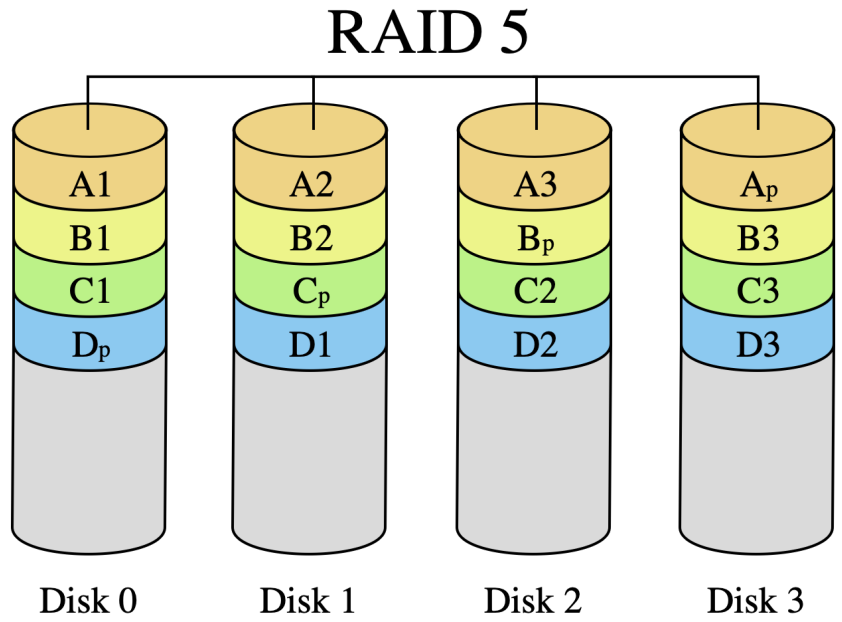
Things that go wrong are called faults.

Systems that anticipate faults are called fault-tolerant or resilient.

Faults are not failures

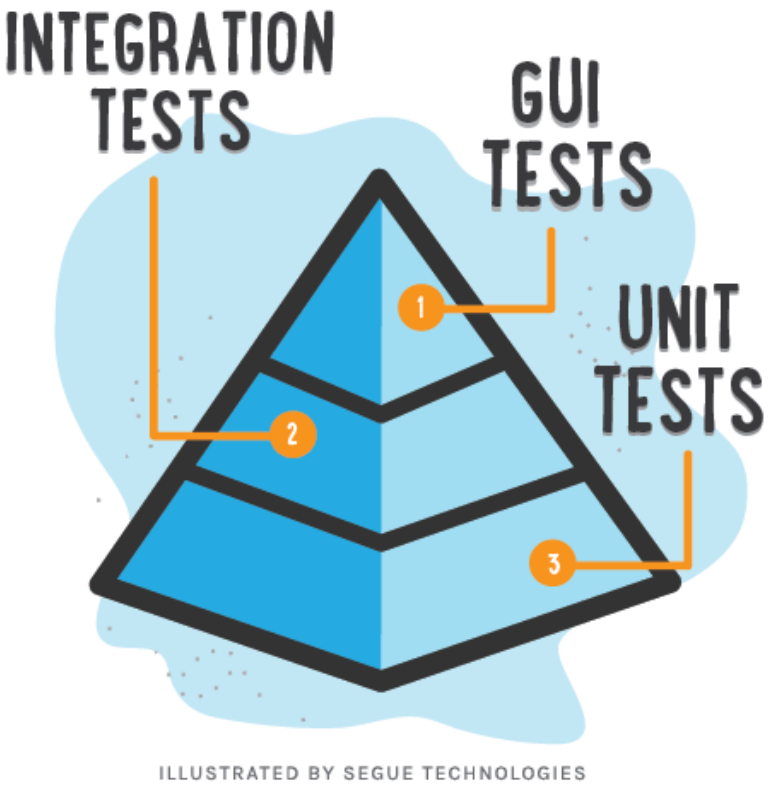
Hardware Faults

- Hard disks can crash. RAM can become faulty. Blackout can happen.
- Hard disks have mean time to failure (MTTF) of 10 to 50 years
  - A cluster of 10,000 disks is expected to have one faulty disk everyday
- Remedied by Redundancy
  - RAID configuration.
  - Dual power supplies.
  - Hot-swappable CPUs.
  - Backup power.
- More in 204441.



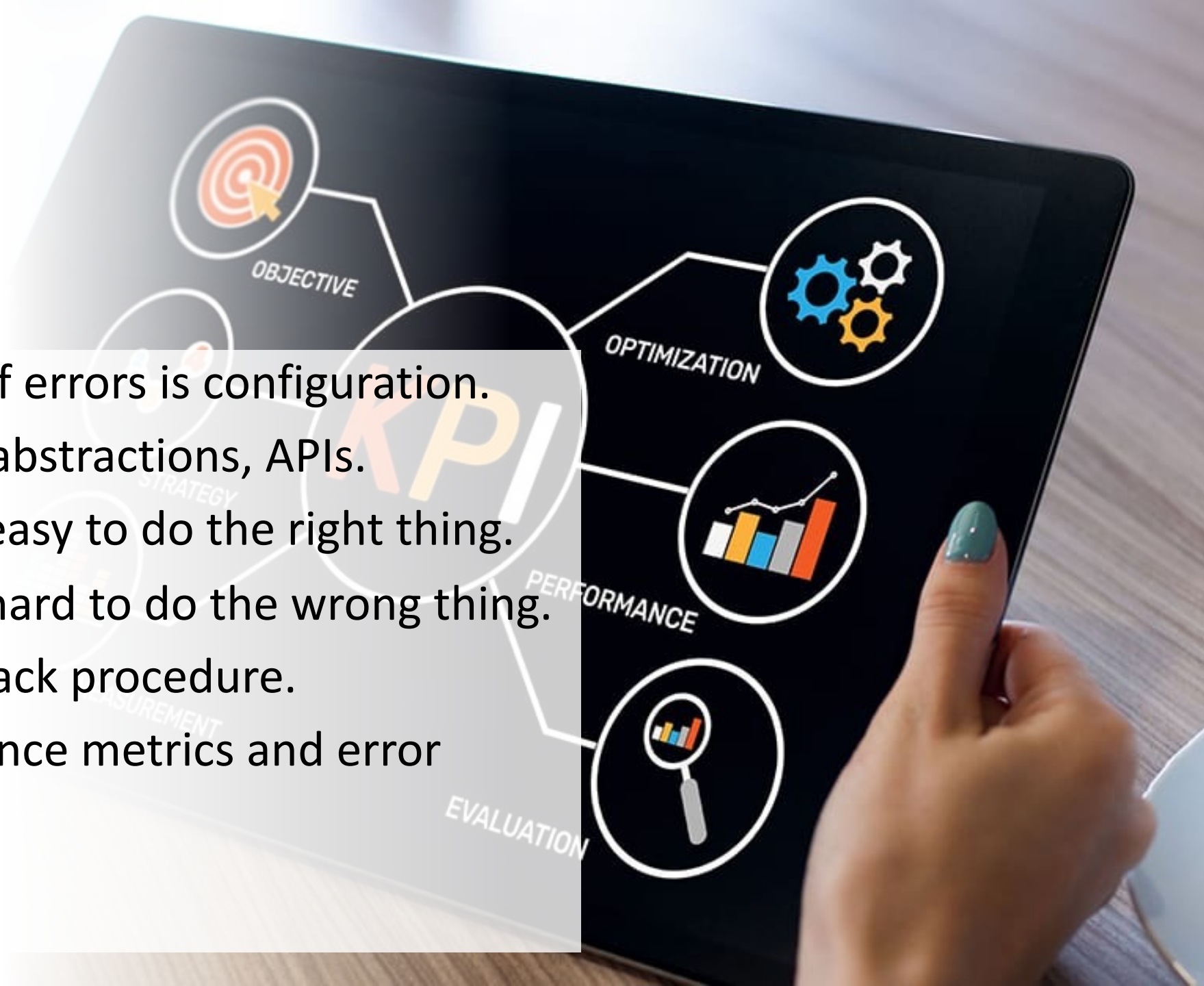
Software Errors

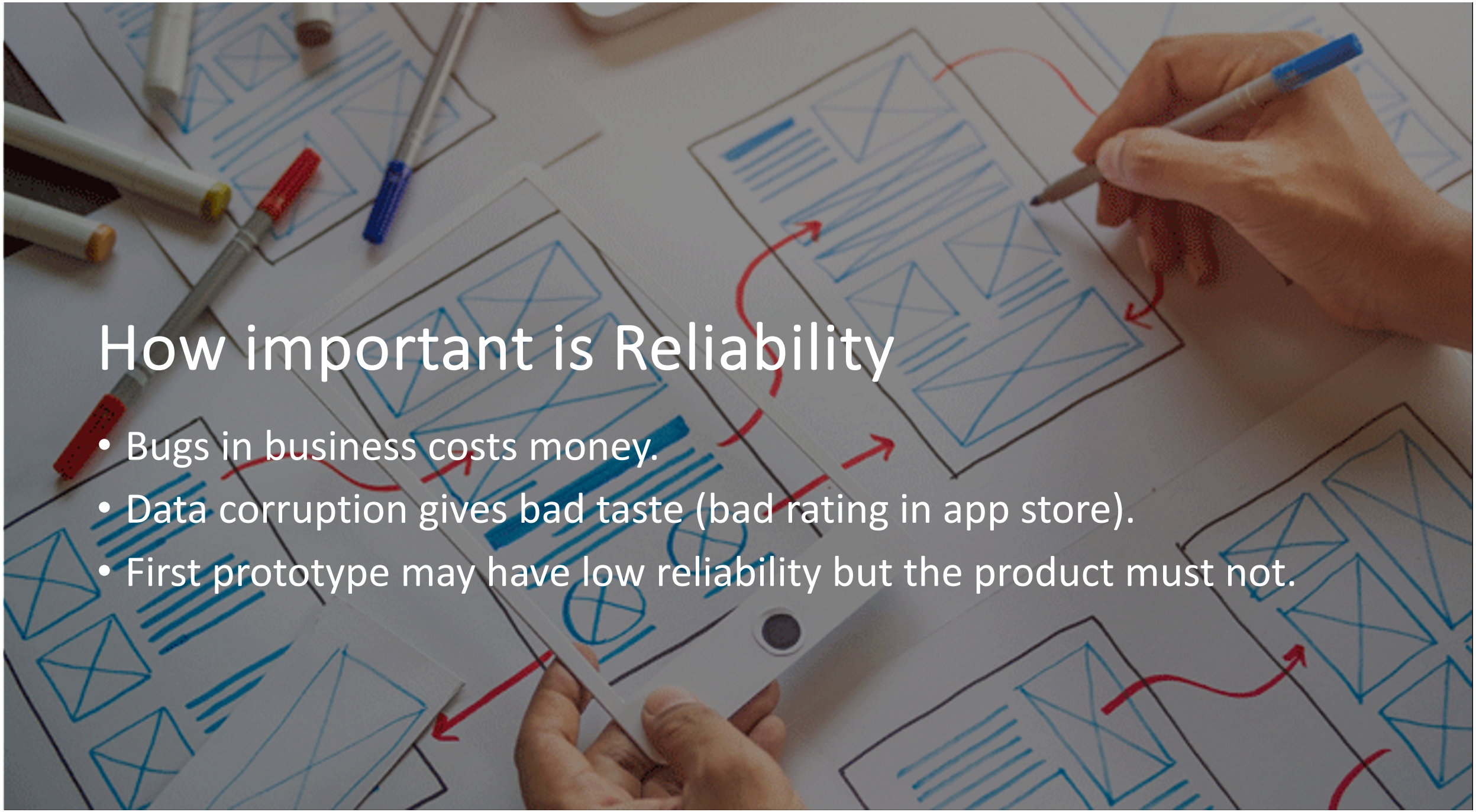
- Crashing from bad input is the most common.
- Memory leak. runaway process.
- A service requests data from slow system.
- Cascading failure: one error leads to another.
- Unit Testing is the key (Luckily, we will do it in this course!!).



Human Errors

- Common source of errors is configuration.
- Need well-design abstractions, APIs.
- Make the system easy to do the right thing.
- Make the system hard to do the wrong thing.
- Always have roll back procedure.
- Monitor performance metrics and error rates.
- More in 204365



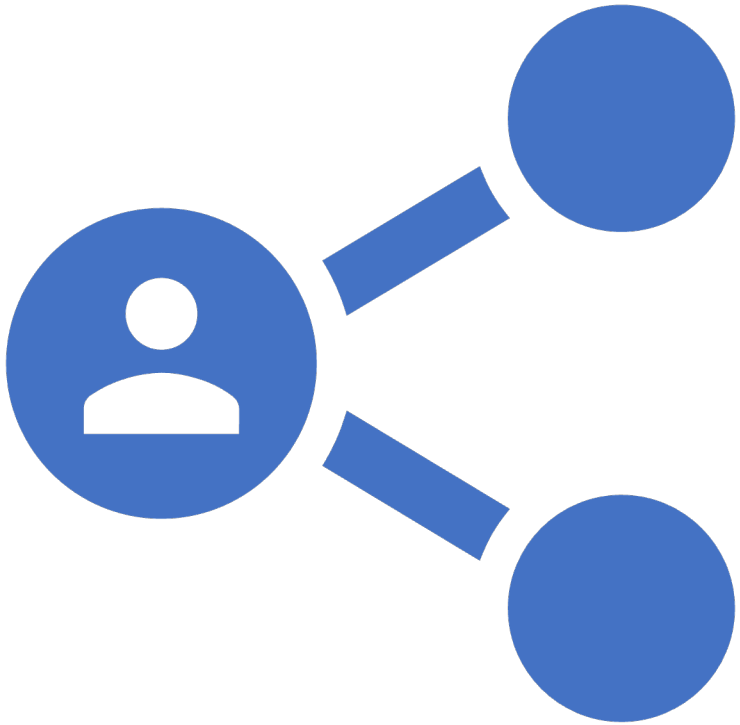


How important is Reliability

- Bugs in business costs money.
- Data corruption gives bad taste (bad rating in app store).
- First prototype may have low reliability but the product must not.

Scalability

Dealing with growth



Scalability

- App working today doesn’t mean it will work in the future.
- Users will increase (from 100 to 1,000 to 1,000,000).
- Data will be accumulated for several years. Need bigger space.
- Scalability is the ability to cope with increased load.

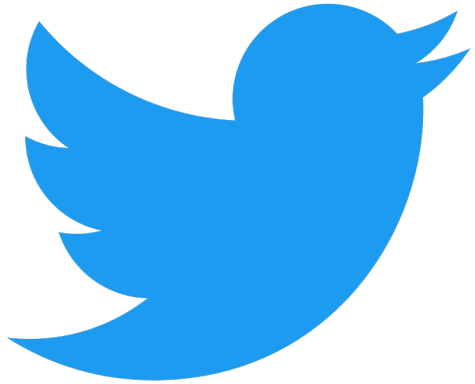
Describing Load

Load is a combination of load parameters, basically numbers.

- Requests per second.
- Read/Write ratio to database.
- Concurrent users in chat room.
- Hit rate on a cache.

These will depend on your application.

Twitter (becomes X in 2023) Example

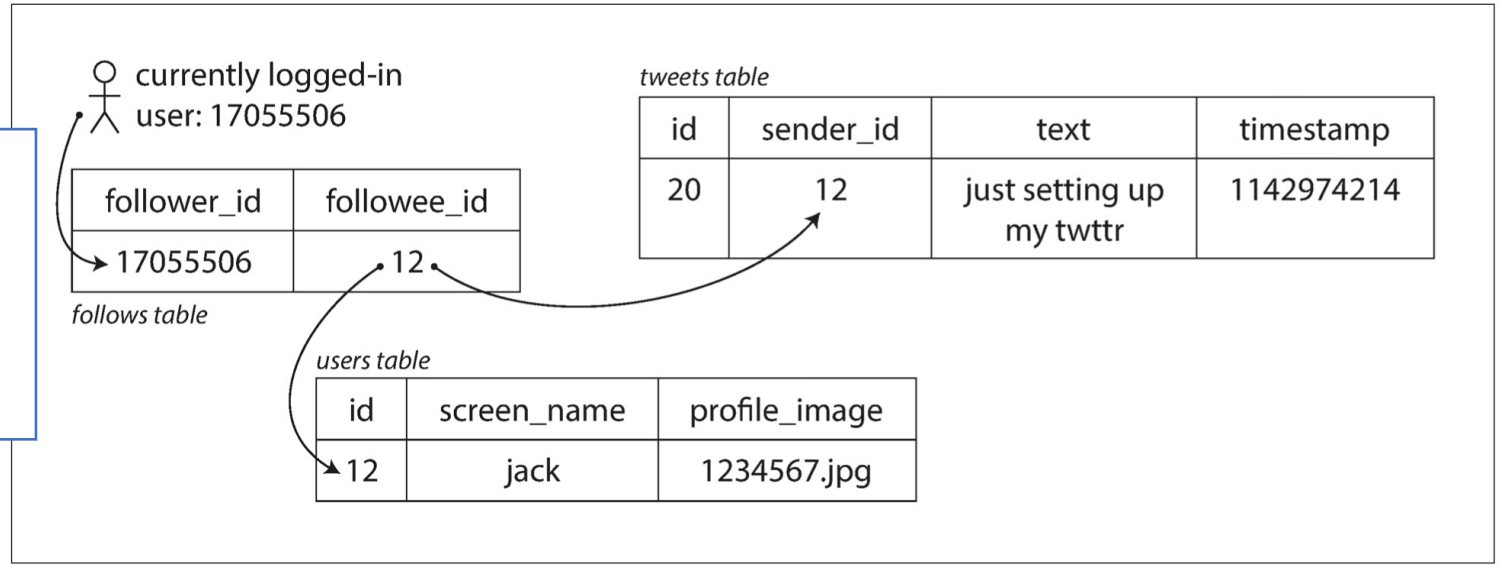


- Data from 2012
- Post Tweet
  - A user can publish a new message to their followers (4.6k requests/sec on average, over 12k requests/sec at peak).
- Home Timeline
  - A user can view tweets posted by the people they follow (300k requests/sec).
- What is the challenge?
  - Writing 12k tweets to database is easy.
  - Imagine each user follow many people and each user is followed by many people.

Approach 1: Global Collection

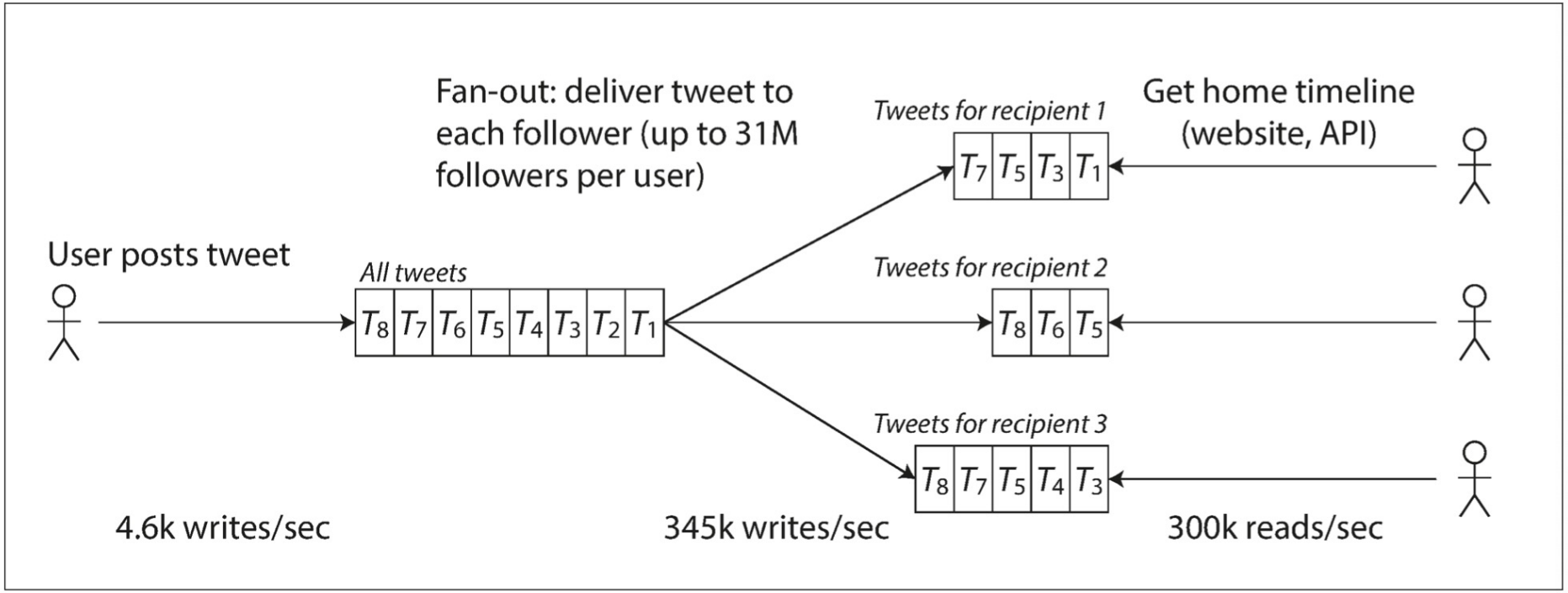
- New tweet inserted to global collection of tweets (insert top).
- When a user requests home timeline, look up all the followings.
- For each following, grab all tweets.
- Merge time and sorted by time.

```
SELECT tweets.*, users.* FROM tweets
JOIN users ON tweets.sender_id = users.id
JOIN follows ON follows.followee_id = users.id
WHERE follows.follower_id = current_user
```



Approach 2: Home Timeline's Cache

- When a user posts a tweet, it will be saved to all tweets table.
- Then, look up all the people who follow that user and insert the new tweet into each of their home timeline cache.



Which is better?

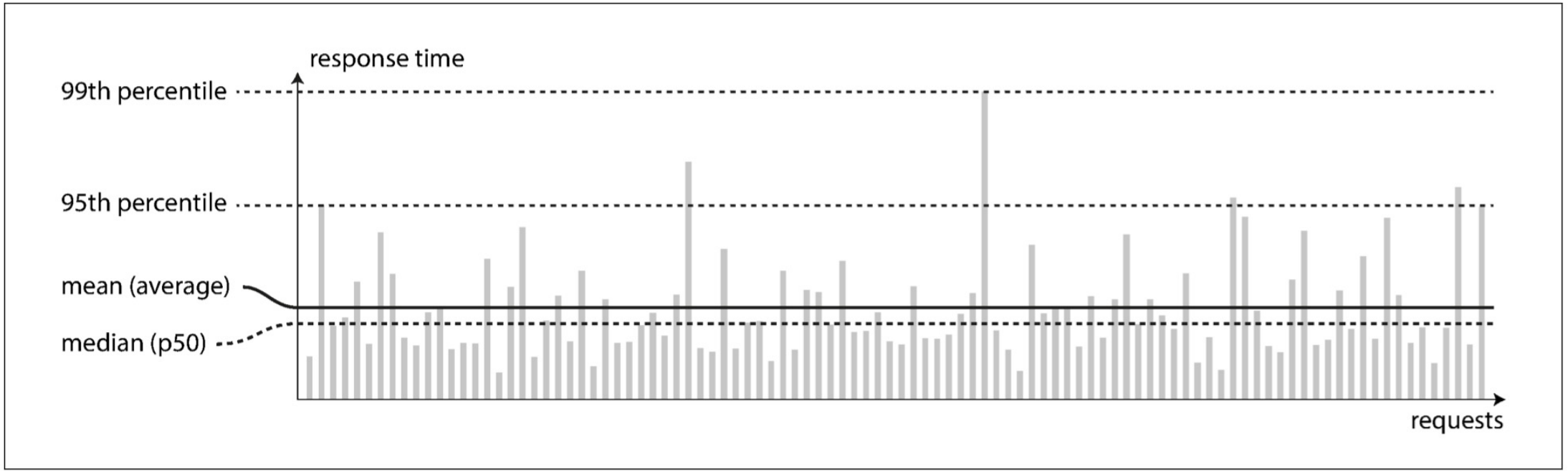
- First version of Twitter used approach 1.
- They switched to approach 2 to reduce load of home timeline query.
- What is the downside of approach 2 if their goal is to deliver tweet within 5 seconds?
  - Number of followers is the key load parameter for scalability
- What is current approach for Twitter now?

Describing Performance

- There are two ways to investigate when happened when loads are increased.
  - Keep system resource unchanged, how is performance affect?
  - How much do you need to increase the resource to keep performance unchange?
- The response time is what the client sees: besides the actual time to process the request (the service time), it includes network delays and queueing delays.
- Latency is the duration that a request is waiting to be handled during which it is latent, awaiting service.

Distribution over Value

- Even if you only make the same request repeatedly, you'll get a slightly different response time on every attempts.
- We therefore need to think of response time not as a single number, but as a distribution of values that you can measure.



Average vs Percentile

- Average is summing up all value divided by the number of users.
- However, the average is not a very good metric if you want to know your “typical” response time, because it doesn’t tell you how many users experienced that delay.
- It is better to use percentile.
- Median is halfway point or 50^{th} percentile (p50).
- If your median response time is 200 ms, that means half your requests return in less than 200 ms, and half your requests take longer than that.

High Percentile

- High percentiles of response times (tail latencies) are important because they directly affect users' experience of the service.
- Amazon describes response time requirements for internal services in terms of the p99.9, even though it only affects 1 in 1,000 requests.
- The customers with the slowest requests have the most data on their accounts because they have made many purchases
  - 100 ms increase in response time reduces sales by 1%
  - 1-second slowdown reduces a customer satisfaction metric by 16%

Maintainability

For future generation

Maintainability

- It is well known that most of the cost of software is not in its initial development, but in its ongoing maintenance.
  - fixing bugs.
  - keeping its systems operational.
  - adapting it to new platforms.
  - modifying it for new use cases.
  - adding new features.
- Many people don’t like to work on legacy systems.
- We should design software that reduce the pain of maintenance.

Three Design Principles for Software Systems

Operability - Make it easy for operations teams to keep the system running smoothly.

Simplicity - Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system.

Evolvability - Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change.

Operability: Making Life Easy for Operations

- Good operations can work around the limitations of bad software, but good software cannot run reliably with bad operations.
- Good operability means making routine tasks easy
  - Providing visibility into the runtime behavior and internals of the system, with good monitoring.
  - Providing good support for automation and integration with standard tools.
  - Avoiding dependency on individual machines.
  - Providing good documentation and an easy-to-understand operational model ("If I do X, Y will happen").
  - Providing good default behavior, but also giving administrators the freedom to override defaults when needed.

Simplicity: Managing Complexity

- Small software projects can have delightfully simple and expressive code, but as projects get larger, they often become very complex and difficult to understand.
- Possible symptoms of complexity are:
  - explosion of the state space.
  - tight coupling of modules.
  - tangled dependencies.
  - inconsistent naming and terminology.
  - hacks aimed at solving performance problems.
- Complexity introduces more bugs in the future!

Abstraction

- One of the best tools we have for removing accidental complexity is abstraction.
- A good abstraction can hide a great deal of implementation detail behind a clean, simple-to-understand facade.
- For example, SQL is an abstraction that hides complex on-disk and in-memory data structures, concurrent requests from other clients, and inconsistencies after crashes.

Evolvability: Making Change Easy

- Your system's requirements will change over time.
  - Unanticipated use cases emerge.
  - Business priorities change.
  - Users request new feature.
  - New platforms replace old platforms.
- In practice, agile software development is adapted to change.
  - Test-driven development (TDD)
  - Refactoring, process of restructuring code without changing behavior

Optimization Techniques

Best coding practices

Efficient Memory Management

- Performance: Efficient memory usage ensures faster app operation and smooth animations.
- Resource Conservation: Mobile devices have limited memory. Optimizing memory use ensures that other apps on the device run smoothly.
- User Experience: An app that efficiently manages memory reduces the chances of crashes and enhances user satisfaction.

Best Practices (for Flutter)

- **Leverage Dart's Garbage Collection:** Dart automatically deallocates objects that are no longer in use. Ensure that references to unused objects are nullified so that the garbage collector can reclaim the memory.
- **Use const Constructors for Widgets:** When widgets remain unchanged across rebuilds, using const constructors can prevent unnecessary object instantiations, conserving memory.
- **Opt for Stateless Widgets:** Stateless widgets are lighter on memory as they don't store mutable state. Use them whenever the widget doesn't need to retain mutable information.

Best Practices (for Flutter)

- **Streamline Assets:** Large image or data files can be memory-intensive. Compress images, use appropriate resolutions, and consider using asset bundling or lazy loading.
- **Dispose of Controllers and Streams:** Objects like TextEditingController or subscriptions to streams should be disposed of when they're no longer needed, typically in the dispose() method.
- **Be Careful with Caches:** While caching can speed up operations, excessive caching can bloat memory. Use caching judiciously and set limits.

Best Practices (for Flutter)

- **Leverage Widgets that Automatically Release Resources:** Widgets like ListView.builder and GridView.builder only render items that fit the screen, effectively managing memory by not rendering off-screen objects.
- **Use Weak References:** If you want to reference an object without preventing it from being collected, consider using weak references available in packages like dart:ffi.
- **Global Variables:** Excessive use of global variables can lead to memory leaks as these variables can persist and hold onto memory throughout the app's lifecycle.

Best Practices (for Flutter)

- **Avoid Circular References:** If two objects reference each other, they may not be garbage collected, leading to memory retention.
- **Moderate Animations:** While animations are visually appealing, they can consume significant memory, especially if multiple animations run simultaneously.
- **Testing on Low-end Devices:** Just because your app runs smoothly on the latest devices doesn’t mean it will perform well on older, less capable ones.

Best Practices (for Flutter)

- **Efficient Use of Databases:** Continually querying a local database or holding large amounts of data in memory after a fetch can consume significant amounts of memory. Fetch only the data you need and release memory after operations.
- **Third-party Packages:** Some packages may not be optimized for memory usage. Always vet and monitor third-party packages for potential memory inefficiencies.
- **Use Profiling Tools:** Consider using Flutter DevTools.

Quiz 1: Understanding Memory Management

- In a mobile application, you notice that the memory usage consistently grows as the user navigates through different screens, without decreasing even when returning to previous screens. What is the most likely cause of this issue?
  - A) Inefficient network usage
  - B) Memory leak
  - C) Unoptimized image resources
  - D) Excessive logging

Quiz 2: Data Structures and Performance

- Which of the following data structure choices would generally offer the fastest retrieval time for a frequently accessed, fixed-size list of elements where the order of elements is not important?
  - A) Linked List
  - B) Array
  - C) Hash Map
  - D) Binary Tree

Quiz 3: Asynchronous Operations

- In mobile app development, why are asynchronous operations preferred for tasks such as fetching data from a server?
  - A) They are simpler to implement
  - B) They reduce the app size
  - C) They keep the UI responsive by not blocking the main thread
  - D) They are more secure

Quiz 4: Optimization Scenario

- You are optimizing an e-commerce mobile app. The app has a feature where it displays recommended products based on user preferences. This feature is currently causing the app to slow down. Which of the following is the most appropriate optimization technique?
  - A) Implement lazy loading for recommended products
  - B) Use a simpler algorithm for recommendations
  - C) Increase the number of server requests for product data
  - D) Store all product images locally on the device

References

- Kleppmann, Martin. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.
- https://decode.agency/article/mobile-app-requirement-document/