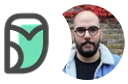
Capacity Planning



BYTEBYTEGO AND DIEGO BALLONA

JUN 29, 2023 · PAID

This newsletter is written by guest author Diego Ballona, who is a senior engineering manager at Spotify.

Follow Diego for more on Twitter.

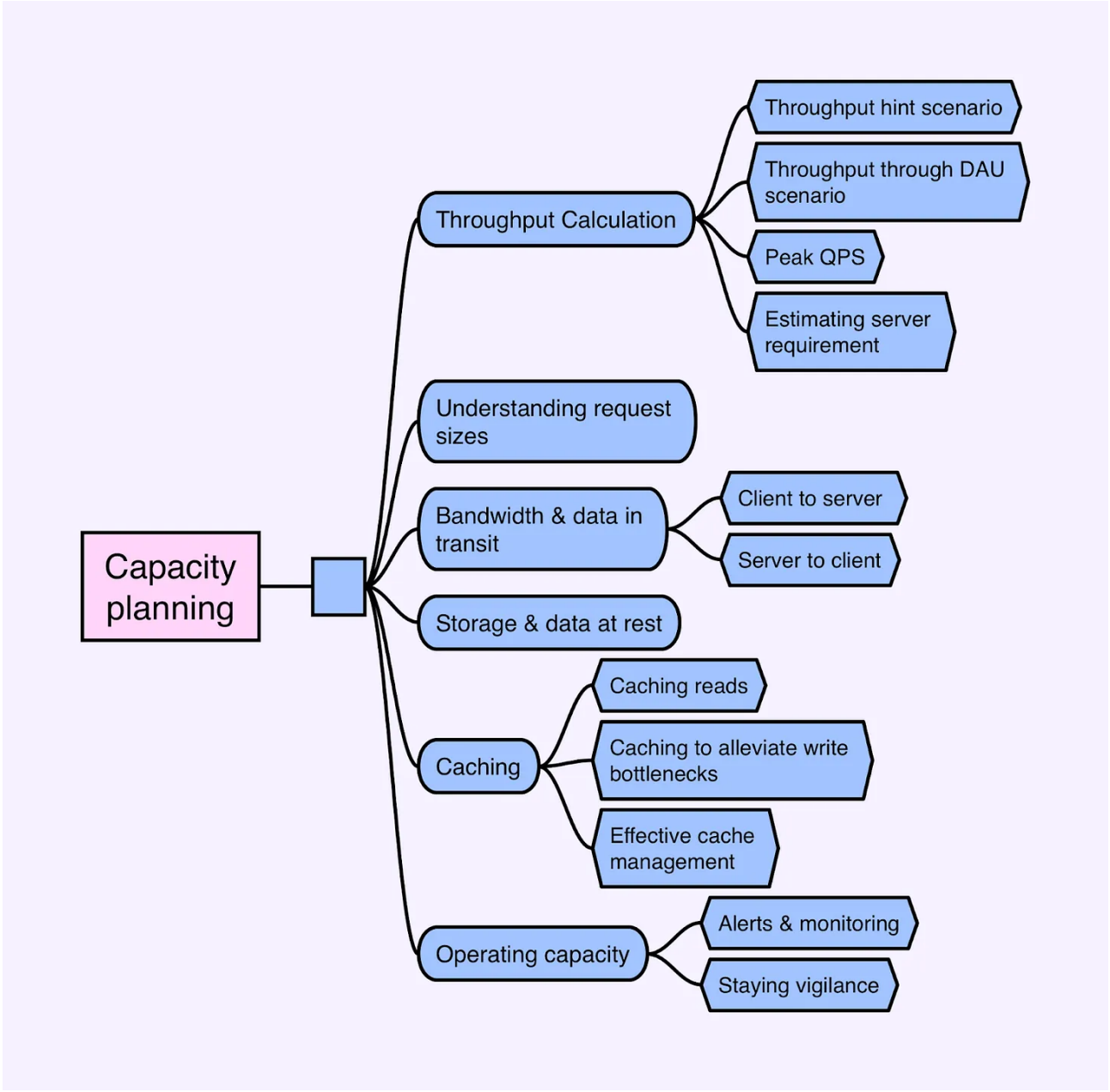
Capacity planning plays an integral role in the landscape of real-world system design. This complex exercise, far from being a mere theoretical consideration, is essential for engineers to accurately estimate the capacity needs of their proposed systems.

Several reasons underscore the significance of capacity planning in system design:

- Capacity planning can tell us if a proposed design is feasible. It allows engineers to foresee potential bottlenecks in system performance or scalability, assess data storage needs, network bandwidth requirements, and predict the projected cost of resources. This verifies that aspects like latency, throughput, and availability align with the non-functional requirements.
- A design might seem functionally feasible, but could have operational challenges. For example, if the system needs to handle surges in demand, it might need a lot of extra resources. This could make it prohibitively expensive to run.
- Effective capacity planning shows a deep understanding of system design. It indicates an engineer's knowledge of various design options and technologies, and the pros and cons of each. It also reflects their ability to anticipate potential problems and find appropriate solutions. This can lead to the development of systems that work better and are more efficient.

So, when designing a large-scale system, it's important to think about capacity planning from the very beginning. By understanding what the system needs to do and how powerful it needs to be, engineers can plan and build systems that are scalable and efficient.

Now, let's explore how capacity planning works in real-world system design.



Throughput Calculation

In designing large-scale systems, it is important to estimate the system's scale in the beginning. Requirements might hint at the throughput, or we may need to infer it from relevant metrics like daily active users.

Throughput hint scenario

Let's consider **Example 1 - Temperature sensors** for a monitoring system. The system needs to measure temperature changes across a county using about 10 million sensors. These sensors report changes every 5 seconds. Analysts use the system to forecast the weather using a dashboard that displays reports.

From these requirements, we can infer a few things about the system's capacity. We can estimate what the throughput is as follows:

- In the extreme case, if **all sensors are synchronized**, we'd have **10 million queries per second** (**QPS**). However, this is an unlikely edge case.
- Assuming each request from a **sensor to server lasts 200ms**, a more realistic average would be **2 million QPS.**

So, for this system, a good throughput estimate is about 2 million QPS.

Throughput through DAU scenario

Sometimes, throughput isn't evident in the requirements. Let's look at **Example 2 - Social Media News Feed**:

The system allows users to post text, images, and videos on their profile timeline, which is organized chronologically. Users can follow others and see relevant posts from those they follow on their timeline.

To inform capacity planning, we could ask:

- How many daily active users does this system have?
- What is the expected user base growth in 2-5 years?

Then, calculate the throughput based on the feature requirements. If the most relevant posts appear in the timeline on the home page, all daily active users likely interact with it. Assume each user interacts with it a certain number of times per day (e.g., 10 times), it means **5 billion page views per day, or roughly 60k QPS**.

```
500M users @ 10x pageviews/user = 5B timeline pageviews/day
5B pageviews / 86400 seconds (1 day = 24 * 60 * 60) = ~57.8k QPS
Round up to 60k QPS
```

Profile visits would likely be less frequent than homepage visits. If we assume that **each user visits two profile pages per day**, that results in **an average QPS of about 12k.**

```
Simplest way to calculate: 20% of 60k (previous example) = 12k QPS

Or, to expand:
```

```
500M users @ 2x pageviews/user = 1B profile pageviews/day
1B pageviews / 86400 seconds (1 day = 24 * 60 * 60) = ~11.5k QPS
Round up to 12k
```

For posts, let's assume that on average, **only 10% of the daily active users post once per day.** This would mean **an average QPS of approximately 6k.**

```
Simplest way to calculate: 10% of 60k (previous example) = 6k QPS

Or, to expand:

500M users * 10% = 50M new posts/day
50M new posts / 86400 seconds (1 day = 24 * 60 * 60) = ~5.7k QPS
Round up to 6k
```

Remember, these are rough estimates. For most system designs, this is good enough. A good tip is to think in round numbers and round up to ensure conservative estimates.

Peak QPS

Calculating peak QPS is important as it often dictates the capacity requirement of the design. Peak QPS refers to the highest rate at which a system will be expected to handle queries, often occurring during times of high usage or even traffic spikes. This can be much higher than the average rate. This is why it requires special attention.

One common method to determine peak QPS is through historical data analysis. This involves tracking the number of queries that the infrastructure handles over a specific timeframe, like days, weeks, or even months, and then choosing the highest value. This method relies on the availability of data and the system's historical performance.

Overprovisioning the infrastructure or utilizing autoscaling features can also help handle peak QPS. These strategies allow the system to increase its capacity temporarily to deal with unexpected surges in traffic. However, they come with their own costs and need careful cost-benefit analysis.

Peak QPS could also be influenced by business requirements or predictable usage patterns. For instance:

- Event-driven Peaks: A major product launch or marketing event might attract a surge of users signing up or using the system within a brief window. Predicting and preparing for these events can help maintain service quality during high demand.
- Time-driven Peaks: Some systems may experience predictable daily or weekly fluctuations in usage. For instance, a business-oriented application may see significantly higher traffic during working hours compared to late evenings or early mornings.

To estimate peak QPS, we often make calculations based on expected distribution characteristics. For example, we might assume that 80% of visits per day occur within 20% of the time (a variant of the Pareto Principle). We add some buffer capacity to handle unexpected surges and provide a smooth user experience.

In **Example 2 - Social Media News Feed**, if we anticipate that 80% of pageviews for timelines occur within an 8-hour time span, we'd calculate the peak QPS for this period to be around 138k. However, this is just a starting point - it's always a good practice to overprovision initially, monitor the data, and then adjust based on the actual usage patterns.

```
500M users @ 10x page views / user = 5B timeline pageviews/day

80% of pageviews = 4B timeline pageviews
Per hour over a period of 8 hours = 4B page views / 8 hours = 500M / hour

Average Peak QPS = 500M / 3600 (60 minutes in seconds) = 138k
```

Preparing for peak QPS helps ensure that the system remains stable and responsive even under the heaviest loads. This contributes to a better user experience and system reliability.

Estimating server requirement

With the estimated throughput and response time, we can estimate the number of servers needed to run the application.

In **Example 1 - Temperature sensors**, the system has an average response time of 200ms and needs to handle 2M QPS, and each application server can manage 32 workers handling 160 QPS, we'd need around 12.5k server instances.

```
2M QPS, 200ms avg response time per request
Each instance has 32 workers
Each worker can handle 5 queries per second (200ms*5)
Each instance can handle then 160 QPS
2M (average QPS) / 160 = 12.5k instances
```

Now that we estimated the scale of the overall system, let's focus on the specifics of the system we are designing, starting with request sizing.

Understanding request sizes

Assessing request sizes is crucial for determining bandwidth and storage requirements. In system design, we often need to accommodate a variety of request types that can significantly impact the load on our system. These could be as simple as GET requests retrieving data or as complex as POST requests that involve large multimedia files.

While initial requirements might not include specific request sizes, we can make informed assumptions based on the system's functionality and the nature of data it handles

When it comes to estimating request sizes, different types of systems will naturally have different expectations. Let's consider the two examples we've been discussing: Temperature Sensors and Social Media News Feed.

For a system like **Example 1 - Temperature Sensors**, the data sent may be relatively small. Assuming data is sent in JSON format, we can estimate the size of each field. The temperature is reported as a float (4 bytes), the sensor ID is a UUID (16 bytes), and we have three additional 4-byte fields. Accounting for the JSON format, the total request body size is less than 100 bytes. Including HTTP headers (typically between 200 to 400 bytes), we can conservatively estimate each request size to be around 0.5KB. This is quite small. However, even such small data requests can add up when dealing with millions of sensors, making this an important consideration.

For a system like **Example 2 - Social Media News Feed**, we're dealing with diverse content types - text, images, and videos. While text posts might only be a few KB, images files could be several hundred KB, and video files could be several MB. In these scenarios, an average request size needs to consider the distribution and size of various content types.

Let's make a conservative estimate of a text-only post being about 1KB, which includes a user ID (UUID) and a free-form string with 250 characters on average. For images and

videos, let's assume that for every 10 posts, there are three images (average size of 300KB after compression), and one is a video (averaging at 1MB). We can fold all these media types into an average to simplify our calculations. This leaves us with an average request size of around **200KB per post**.

```
For 10 posts:
- Every post has 1KB (text) = 10KB
- Three have an image = 300KB * 3 = 900KB
- One has a video = 1000KB

~191KB, round up to 200KB
```

Considerations for Request Sizing

Understanding request sizes is just part of the equation. We must also consider factors like data format, serialization/deserialization, and the impact of these on bandwidth and processing needs.

- **Data Format.** The format of data being sent can significantly impact the bandwidth required. More verbose data formats, like XML, can consume more bandwidth compared to less verbose formats like JSON. Even among similar formats, options like Binary JSON (BSON) can be more efficient.
- **Serialization/Deserialization:** Transforming data into a format that can be easily transmitted or stored (serialization) and then reverting it back to its original form (deserialization) can require significant resources. Different methods have different costs and benefits. For instance, Protobuf is faster than JSON for serialization and deserialization, but JSON is more human-readable and easier to debug.
- **Compression:** Implementing data compression can effectively reduce request size and conserve bandwidth. While it does come with additional processing overhead, the trade-off may be acceptable or even beneficial, especially for larger payloads.
- **Network Protocols:** The choice of network protocol (HTTP/1.1, HTTP/2, gRPC, etc.) can also affect the overall size of a request due to differences in headers, the ability to compress headers, and other factors.

Request sizing is an important aspect of capacity planning. Although it requires making assumptions, these informed guesses help anticipate the system's needs. It ensures that we're well-prepared to handle expected loads and traffic patterns.

Bandwidth & data in transit

Understanding each record's cost is fundamental. We need to consider the bandwidth per operation in both directions - client to server (ingress) and server to client (egress).

Client to server (ingress)

In **Example 1 - Temperature sensors**, the data sent by the sensors to the application is known as ingress: incoming data to the application's network. Suppose each sensor request is 0.5KB and the average QPS is 400k, our average ingress bandwidth is approximately 200MB/second. We can simplify the discussion by leaving out the read path for dashboards, etc., as it probably won't significantly impact capacity.

An interesting aspect to consider is how often we need to send data to the servers. It's possible that not all data requires immediate transmission. For example, a temperature sensor might only need to report data when a significant temperature change occurs. By storing the temperature state locally on the sensor and transmitting it only upon changes, we could potentially reduce the capacity load and costs.

However, it's important to set a limit for the maximum time a sensor can go without reporting to the server. This ensures that any malfunctioning or offline sensors are promptly identified.

Server to client (egress)

When considering a system like **Example 2 - Social Networking News Feed**, posting content would equate to ingress traffic. Based on our previous estimates, suppose each post is 1KB and we've got an average of 6k QPS, our ingress bandwidth is about 6MB/second.

The egress, or the data transmitted from the server to client, can be complex. Suppose our system paginates every 20 posts, and the server renders raw file versions each time. In this scenario, our average egress bandwidth would be very high at ~**250GB/second**:

```
500M users @ 10x pageviews/user = 5B timeline pageviews/day
Average post request size: 200KB
Assuming every time timelines are rendered the server sends everything to
the client.

20 posts * 200 KB = 4MB
```

```
5B pageviews * 4MB = 20PB
20PB / 86400 (1 day = 24 * 60 * 60) = ~231GB/s

Round up to ~250GB/s
```

But there are optimization strategies to reduce this egress traffic.For example:

- Trimming the content: We can reduce the data load by not sending the full text of the post content by default. This could reduce the text size by half.
- Image optimization: Low-quality image placeholders can be sent initially, and high-quality images can be progressively loaded based on user interactions and viewport visibility.
- Video optimization: Autoplay for videos can be turned off, with only a thumbnail loaded by default. The full video can be streamed on demand, significantly reducing the initial data load.

Here's a rough calculation:

- 80% of our user base won't get past the fifth post on the timeline;
- Sending only an excerpt of the text will reduce the text size by 50%;
- We reduce thumbnail size to the client to 30KB as placeholders and only load on demand (300KB) for what's visible in viewport
- Video will only be loaded on play (no autoplay), and by default will load the same thumbnails as images, progressively. Only 10% of the users will play videos.

We could achieve around 67% saving in our capacity, without caching:

```
Previous average post request size: 200KB
======
Now, for 10 posts (without counting progressive loading):
- Every post has 0.5KB (text) = 5KB
- Three have an image = 30KB * 3 = 90KB
- One has a video = 30KB
12.5KB, round up to 15KB

Timelines render 20 posts by default, 80% stops at 5th post (25%)
Assume users will interact with every single post in every possible way
for these 5 posts:
15 posts @ 15KB + 5 posts @ 200KB = (15*15+5*200) / 20 = 61.25KB
Round up to 65KB
```

```
Saving = ~135KB (-67%)
```

These optimizations, when combined, can lead to considerable savings in bandwidth and enhance user experience by reducing page load times. Always remember that these calculations, while useful for planning, can evolve based on the actual usage patterns and should be validated through continuous monitoring and adjusting.

Storage & data at rest

When considering the architecture of any system, it's essential to plan for storage and the management of data at rest - that is, data that's not currently moving through the network. There are several aspects to consider, including storage capacity, data lifecycle management, data accessibility, data security, and cost.

Capacity planning in the context of storage involves predicting the volume of data that the system will generate or manage, and ensuring we have sufficient storage capacity to handle it.

In a system like **Example 1 - Temperature sensors** where sensors are continuously collecting temperature data, the sheer volume of data generated could be immense. If we were to store every single data point, assuming each point is about 0.1KB in size, with 2 million data points per second, we'd be looking at about 17TB of data per day.

```
2M QPS @ 0.1KB (not storing headers) = 200MB/s
Storage per day = ~17TB
```

Storage requirements of this magnitude could quickly become unmanageable and costly. It's important to find ways to optimize storage. One approach is data deduplication - only storing unique data points or those that reflect a significant change.

In the context of **Example 1 - Temperature sensor**, rather than storing temperature data every time it's recorded, we could only store data when there's a significant change in temperature. This could reduce the storage requirements drastically. For example, if temperate changes only once a minute, daily storage is reduced to about 1.5 TB.

```
Sensors send every 5 seconds = 17TB
Sensors send every 60 seconds = 17TB / 12 = ~1.5TB
```

Even with deduplication, storing such large volumes of data can be challenging. This is where data retention policies and lifecycle management come into play.

Not all systems need to - or due to compliance issues, can - retain data indefinitely. An effective data retention policy can dramatically enhance system performance and reduce costs. A common strategy involves data granularity. For example, we could aggregate data by time or sensor ID.

To further optimize, we might keep raw events from the past month and increase the granularity for data older than this. Retain the end state of each hour, along with computations for properties that changed multiple times within that hour. If these computations require an additional 0.1KB, each hourly record will cost 0.2KB. Consequently, each year's worth of data will cost about 1.8MB, and the past month's raw events will amount to around 45TB (1.5TB * 30 days).

```
Per hour = 0.2KB
Per day = 24h * 0.2KB = ~5KB
Per month = 150KB
Per year = 1.8MB
```

Caching

Caching is a valuable technique that can help to ease overhead and mitigate infrastructure costs and capacity limits. This section will delve into several strategies for effective caching.

Caching reads

Often, a small fraction of the content accounts for the majority of the reads. In such cases, caching can significantly reduce bandwidth usage. In **Example 2 - Social Networking News Feed**, we could cache most of the content using a Content Delivery Network (CDN), such as Cloudfront.

By employing caching, we not only conserve bandwidth from our infrastructure, but also enhance user experience and reduce costs, particularly for static assets.

In this example where 20% of content accounts for 80% of the reads, caching this content can lead to a significant reduction in bandwidth consumption.

Caching to alleviate write bottlenecks

Caching is also a useful tool for systems with heavy write operations. It helps to reduce storage capacity needs. In **Example 1 - Temperature sensors,** we could cache the sensor/temperature data pairs and only commit them to long-term storage when a change occurs. Furthermore, we can aggregate writes in the cache, writing them into the long-term persistence layer only after a certain number of cache writes.

This strategy can help alleviate write bottlenecks and reduce the storage requirements.

Effective cache management

While caching can be a powerful tool, it requires effective management to avoid unexpected consequences. Here are three strategies on how to manage cache usage:

Write-through cache: In this method, data is written into the cache and the corresponding database at the same time. The cache serves as a read cache, with the database functioning as a longer-term storage layer. The trade-off here is an increase in latency as an operation is only confirmed once both cache and database writes are successful.

Write-around cache: This strategy involves writing directly to the long-term storage, bypassing the cache. While this can reduce cache overhead for write operations that won't be immediately accessed, it does have a downside. The data can only be read from the slower persistence layer, which leads to increased latency.

Write-back cache: In this approach, data is written solely to the cache, and confirmation is sent to the client immediately. Writes to the permanent storage are made under specific conditions, such as after every tenth record, or every set number of minutes. This method offers low latency, but the trade-off is the risk of data loss if the cache is flushed or if data is lost from the cache.

Operating capacity

Alerts & monitoring

One way to prevent issues that might arise from incorrect design assumptions is to code these assumptions as alerts in production. For instance, if we assumed a 20% cache hit in the design, but only achieved a 5% hit rate in practice, monitoring can help detect this discrepancy. While it's likely that monitoring is already in place, it's crucial to ensure these assumptions are tracked. Monitoring these previously made assumptions allows for proactive action, rather than waiting until the next infrastructure bill arrives or an incident occurs.

Staying vigilance

Despite our best efforts to use data for forecasting and understanding system loads, the reality is that not every scenario can be perfectly predicted. This unpredictable nature of systems and their loads demands constant vigilance to ensure the system remains robust and performant.

For instance, when making significant changes to the system, it's essential to consider potential risk factors. One effective strategy involves conducting thorough "what-if" analyses. What would happen if a certain parameter unexpectedly changed? What would be the consequence if a specific function or feature was suddenly in high demand?

Moreover, it's also useful to keep an eye on non-functional requirements like system responsiveness or uptime guarantees. Any changes in these areas could have ripple effects throughout the system, potentially causing bottlenecks or performance issues.

In short, staying vigilant means consistently questioning and reevaluating the system's readiness for change and unexpected scenarios. This constant reassessment can help us spot potential issues before they escalate, keeping our system resilient and ensuring it has the capacity to handle future challenges.

Conclusion

Capacity planning stands as a cornerstone of effective system design.

It allows engineers to navigate the intricate balance between resource usage, performance, and cost, thereby building systems that not only meet today's needs but can also adapt to the demands of tomorrow. Whether we're architecting a new system or optimizing an existing one, capacity planning is a critical process that brings visibility to the underlying complexities of system design, facilitating informed decisions, and proactive problem-solving.

While no capacity planning can perfectly predict the future, incorporating these principles and strategies into the design process can reduce the risk of over or underprovisioning, improve system performance, and ultimately lead to more resilient and efficient systems.

169 Likes · 4 Restacks

A guest post by

Diego Ballona

I like building products and teams to solve mission-critical challenges. I also write about engineering management, software engineering and product development on Twitter as @dballona and on my blog https://dballona.com

5 Comments

Write a comment...

Adam Johnston Jul 2

Could you explain where the average QPS is coming from for the temperature sensor example? Is that breaking up 10 million over a 5 second period equally (10/2)? Not sure how "Assuming each request from a sensor to server lasts 200ms" plays into the average.

LIKE (3) REPLY SHARE

3 replies

Dipanjan Jul 19

Request size calculation seems incorrect, it should be 2 Mb and not 200 kb

LIKE REPLY SHARE

3 more comments...