Put Back-of-the-envelope Numbers in Perspective

Learn to use appropriate numbers in back-of-the-envelope calculations.

Back-of-the-envelope calculations (BOTECs) involve swift, approximate, and simplified estimations or computations typically done on paper or, figuratively, on the back of an envelope. While these calculations are not intended to yield precise results, they function as a quick and preliminary evaluation of crucial parameters and the feasibility of a system.

For example, let's say we're in a city and want to estimate the population of a particular neighborhood. We could count the number of houses in a sample area, estimate the average number of people per household, and then extrapolate to the whole neighborhood. Similar calculations can be used to check the validity of census data for some neighborhoods.

BOTECs in system design#

A modern system is a complex web of computational resources connected via a network. Different kinds of nodes, such as load balancers, web servers, application servers, caches, in-memory databases, and storage nodes, collectively serve the clients. Such a system might be architected in different ways, including a monolithic architecture, a modular monolith architecture, or a microservices architecture. Precisely considering such richness at the design level (especially in an interview) isn't advisable, and sometimes, it's impossible.

BOTECs help us ignore the nitty-gritty details of the system (at least at the design level) and focus on more important aspects, such as finding the feasibility of the service in terms of computational resources.

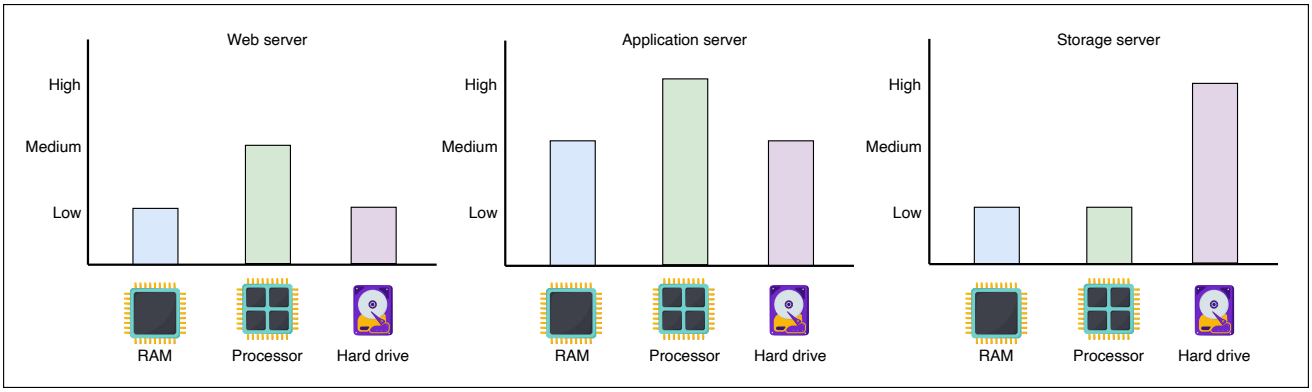
Some examples where we often need BOTECs are the following estimations:

- The number of concurrent TCP connections a server can support
- The number of requests per second (RPS) a web, database, or cache server can handle
- The storage requirements of a service

Using BOTECs, we abstract away the messy details specific to different kinds of servers used in the actual system, the different access latencies of system components, different throughput rates, and the different types of requests. As we move forward, we'll first look into these different server types, access latencies, throughput numbers, and request types to know the reality of the systems and see how complex they are. Then, abstracting away these details, we'll learn to estimate the number of RPS a server can handle. Finally, we'll practice bandwidth, servers, and storage estimation examples.

Types of data center servers#

Data centers don't have a single type of server. Enterprise solutions use commodity hardware to save costs and develop scalable solutions. Below, we discuss the types of servers that are commonly used within a data center to handle different workloads.



An approximation of the resources required on the web, application, and storage layer of the server, where the y-axis is a categorical axis with data points indicating levels of low, medium, and high resource requirements

Web servers#

For scalability, web servers are decoupled from application servers. **Web servers** are the first point of contact after load balancers. Data centers have racks full of web servers that usually handle API calls from clients. Depending on the service that's offered, the memory and storage resources in web servers can be small to medium. However, such servers require good processing resources. For example, Facebook has used a web server with 32 GB of RAM and 500 GB of storage space in the past.

Application servers#

Application servers run the core application software and business logic. The difference between web servers and application servers is somewhat fuzzy. Application servers primarily provide dynamic content, whereas web servers mostly serve static content to the client. They can require extensive computational and storage resources. Storage resources can be volatile and nonvolatile. Facebook has used application servers with a RAM of up to 256 GB and two types of storage—traditional rotating disks and flash—with a capacity of up to 6.5 TB.

Storage servers#

With the explosive growth of Internet users, the amount of data stored by giant services has multiplied. Additionally, various types of data are now being stored in different storage units. For instance, YouTube uses the following data stores:

1. **Blob storage:** This is used for its encoded videos.
2. **Temporary processing queue storage:** This can hold a few hundred hours of video content uploaded daily to YouTube for processing.

3. **Bigtable:** This is a specialized storage used for storing a large number of thumbnails of videos.
4. **Relational database management system (RDBMS):** This is for users' and videos' metadata (comments, likes, user channels, and so on).

Other data stores are still used for analytics, for example, Hadoop's HDFS. Storage servers mainly include structured (for example, SQL) and nonstructured (NoSQL) data management systems.

Returning to the example of Facebook: they've used servers with a storage capacity of up to 120 TB. With the number of servers in use, Facebook is able to house exabytes of storage. (One exabyte is 10^{18} bytes. By convention, we measure storage and network bandwidth in base 10, and not base 2.) However, the RAM of these servers is only 32 GB.

> **Note:** The servers described above aren't the only types of servers in a data center. Organizations also require servers for services like configuration, monitoring, load balancing, analytics, accounting, caching, and so on.

We need a reference point to ground our calculations. In the table below, we depict the capabilities of a typical server that can be used in the data centers of today:

Typical Server Specifications

Component	Count
Processor	Intel Xeon (Sapphire Rapids 8488C)

Number of cores	64 cores
RAM	256 GB
Cache (L3)	112.5 MB
Storage capacity	16 TB

Standard numbers to remember#

A lot of effort goes into the planning and implementation of a service. But without any basic knowledge of the kinds of workloads machines can handle, this planning isn't possible. Latencies play an important role in deciding the amount of workload a machine can handle. The table below depicts some of the important numbers system designers should know in order to perform resource estimation.

Important Latencies

Component	Time (nanoseconds)
L1 cache reference	0.9
L2 cache reference	2.8
L3 cache reference	12.9
Main memory reference	100

Compress 1KB with Snzip	3,000 (3 microseconds)
Read 1 MB sequentially from memory	9,000 (9 microseconds)
Read 1 MB sequentially from SSD	200,000 (200 microseconds)
Round trip within same datacenter	500,000 (500 microseconds)
Read 1 MB sequentially from SSD with speed ~1GB/sec SSD	1,000,000 (1 milliseconds)
Disk seek	4,000,000 (4 milliseconds)
Read 1 MB sequentially from disk	2,000,000 (2 milliseconds)
Send packet SF->NYC	71,000,000 (71 milliseconds)

Remember the order of magnitude difference between different components and operations is more important than remembering the exact numbers. For example, we should know that doing IO-bound work (for example, reading 1 MB data sequentially from the SSD disk) is two orders of magnitude slower than CPU-bound work (for example, compressing 1 KB data as snzip). You might be wondering why the data sizes are different in the comparison!

As long as the data to compress is readily available to the processor from L1, L2, or L3 caches, the time to compress will be relatively consistent. The data up to the size of the L3 cache of the server (which is normally a few MBs—45 MBs for a typical server, as mentioned above) fits entirely within the cache, and therefore, compressing data up to this limit will take almost the same time. This is because the processor can quickly access the data from the cache without incurring the additional latency associated with fetching data from slower levels of memory or storage.

Apart from the latencies listed above, throughput numbers are measured as queries per second (QPS) that a typical single-server datastore can handle.

Important Rates

QPS handled by MySQL	1000
QPS handled by key-value store	10,000
QPS handled by cache server	100,000–1 M

The numbers above are approximations and vary greatly depending on a number of reasons, like the type of query (**point↴** and **range↴**), the specification of the machine, the design of the database, the indexing, the load on the server, and so on.

> **Note:** For real projects, initial designs use BOTECs similar to the ones we use in a system design interview. As the design goes through iterations for real products, we might use reference numbers from some synthetic workload that match our requests (for example, spec int for CPU characterizations and TPC-C for measuring database transactions per unit time). Initial prototypes are used to validate design-level assumptions. Later on, built-in monitoring of resources and demand is carefully analyzed to find any bottlenecks and for future capacity planning.

Points to ponder

1. **With reference to the throughput numbers given above, what will be your reply if an interviewer says that they think that for a MySQL database, the average count of queries per second handled is 2000?**

Show Answer

Q1 / Q2

Request types#

We'll see in the next section that while estimating the number of requests a server can handle, we don't get into the details of what kind of requests we're going to calculate for. But in reality, all requests are not the same. Workloads (clients' requests) can be broadly classified into three categories: CPU-bound, memory-bound, and IO-bound.

CPU-bound requests: These primarily depend on the processor of a node. An example of a CPU-bound request is compressing 1 KB of data as snzip. From the table above, we see that this operation takes 3 microseconds.

Memory-bound requests: These are primarily bottlenecked by the memory subsystem. An example is reading 1 MB of data sequentially from the RAM of a node. From the table above, we see that such an operation takes 9 microseconds (that's *three times* slower than a CPU-bound operation!).

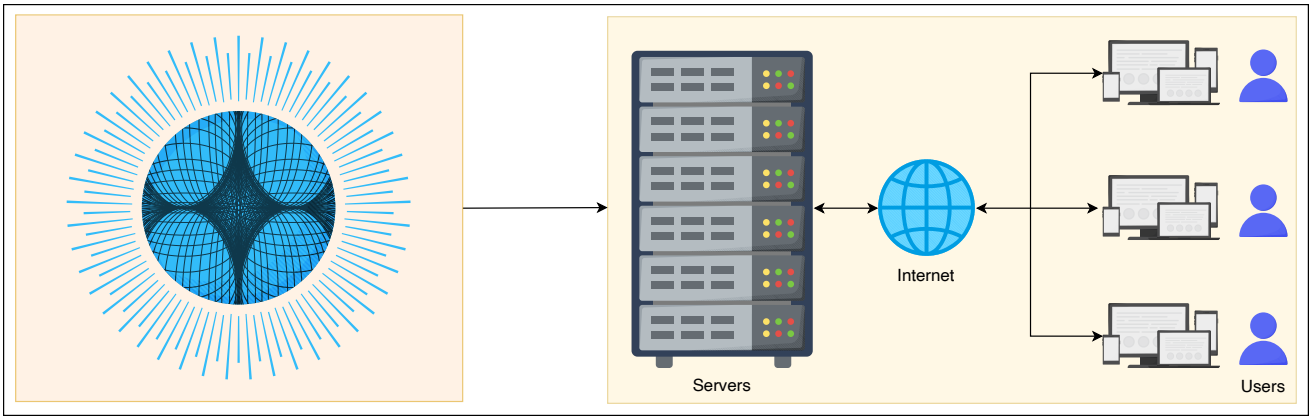
IO-bound requests: These are primarily bottlenecked by the IO subsystem (such as disks or the network). An example is reading 1 MB of data sequentially from a disk. From the table above, we see that such an operation takes 200 microseconds (a whopping *66 times* slower than CPU-bound operations!)

Similar to BOTECs, we can say that if a CPU-bound request takes X time units to complete some work on a node, then memory-bound workloads are an order of magnitude slower ($10X$), and IO-bound workloads are two orders of magnitude slower ($100X$) than the CPU-bound workload. We do such simplifications to make any further calculations easier.

Abstracting away the complexities of real system#

Above, we've seen the complexities involved in real systems. You might have realized that considering all such complexities at the design level, especially in a limited time frame such as an interview, is impractical.

BOTECs are a valuable tool for making quick, high-level estimates and decisions in the early stages of system design or when a rapid assessment is needed. So, moving forward, we'll learn to perform back-of-the-envelope calculations.



A real service is complex, where requests flow through many microservices, as shown on the left side of the image (which is an abstraction of the right side)

Request estimation in system design#

This section discusses the number of requests a *typical server* can handle in a second. A real request will touch many nodes in a data center for different kinds of processing before a reply can be sent back to the client, and we'll accumulate all such work for our estimations.

The following equation calculates the CPU time to execute a program (request). For simplicity, we assume that each instruction can be executed in one clock cycle; therefore, CPI (clock cycles per instruction) is 1 in the following equation. Let's assume the average clock rate for our servers' processor is $3.5\ GHz$ (3,500,000,000 cycles per second). It's reasonable to assume that a request will consume a few million instructions for full processing. For simplicity, let's assume that, on average, each request needs $3.5\ million$ instructions.

$$CPU_{time\ per\ program} = Instruction_{per\ program} \times CPI \times CPU_{time\ per\ clock\ cycle}$$

Let's see that the units match on both sides of the equation. On the right side, we have the following:

- $Instruction_{per\ program}$: This is a count of the instructions the program (request) consists of, so it has no unit.

- CPI: This is a count of the clock cycles required to process one instruction, so it has no unit.
- $CPU_{time\ per\ clock\ cycle}$: This is the time the CPU takes to complete one clock cycle, measured in seconds.

From this, we can see that on the right side, we have the result in seconds, which is the time taken by the CPU per program (request).

Now, we'll put the assumed values in the equation above. But before that, we'll find the CPU time per clock cycle given the CPU frequency equalling $3.5\ GHz$.

$$Clock\ cycles\ per\ second\ for\ a\ CPU\ with\ a\ 3.5\ GHz\ clock\ rate = 3.5 \times 10^9$$

$$CPU_{time\ per\ clock\ cycle} = \frac{1}{3.5 \times 10^9}$$

Putting all the values together, we get:

$$CPU_{time\ per\ program} = (3.5 \times 10^6) \times 1 \times \frac{1}{3.5 \times 10^9} = 0.001\ second$$

$$Total\ requests\ a\ CPU\ executes\ in\ 1\ second = \frac{1}{10^{-3}} = 1000\ requests$$

$$Total\ requests\ a\ 64\ core\ server\ executes\ in\ 1\ second = 64000\ requests$$

Note that by changing the assumptions (for example, the number of instructions in a request), we'll get different final answers. In the absence of more information from these measurements, our estimates are reasonable.

Note how we avoided the complexities related to CPU, memory, or io-bound requests and system architecture. Doing so is the hallmark of BOTECs.

In the next lesson, we'll use RPS numbers for server estimation with other resources, such as storage and network bandwidth.