

[Show Contents](#)[Filter](#)

Week 22.1

In this lecture, Harkirat explores **vertical** and **horizontal scaling**, **single-threaded** and **multi-threaded** architectures, implementing horizontal scaling in Node.js, and **capacity estimation** using examples. He covers scaling techniques, concurrency models, practical Node.js scaling implementations, and **resource estimation** based on traffic and performance needs.

[Vertical Scaling](#)

[Single-threaded Languages](#)

[Multi-threaded Languages](#)

[Implementing Horizontal Scaling in Node.js Project](#)

[Stickiness in the Browser](#)

[Capacity Estimation](#)



How Can You Support a Certain SLA Given Some Traffic?

Paper Math for Capacity Estimation

Example 1: PayTM App

Aggregate Requests and Auto Scaling

Example 2: Chess App

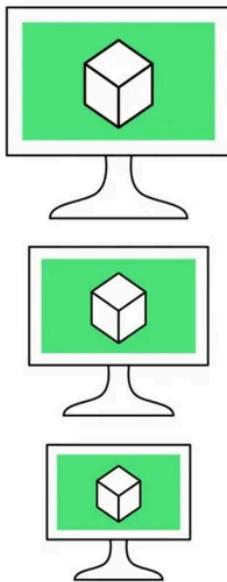
Aggregate Number of Active People

Vertical Scaling

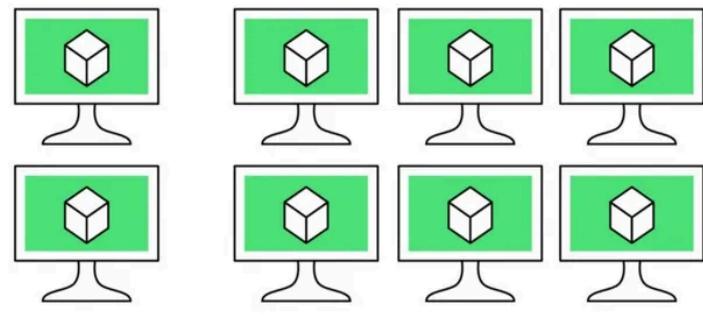
Vertical scaling, also known as scaling up, refers to increasing the capacity of a single machine or instance to handle more load or workload. This typically involves upgrading the hardware resources, such as CPU, RAM, or storage, of an existing server or virtual machine.

Scalability

Vertical scaling



Horizontal scaling

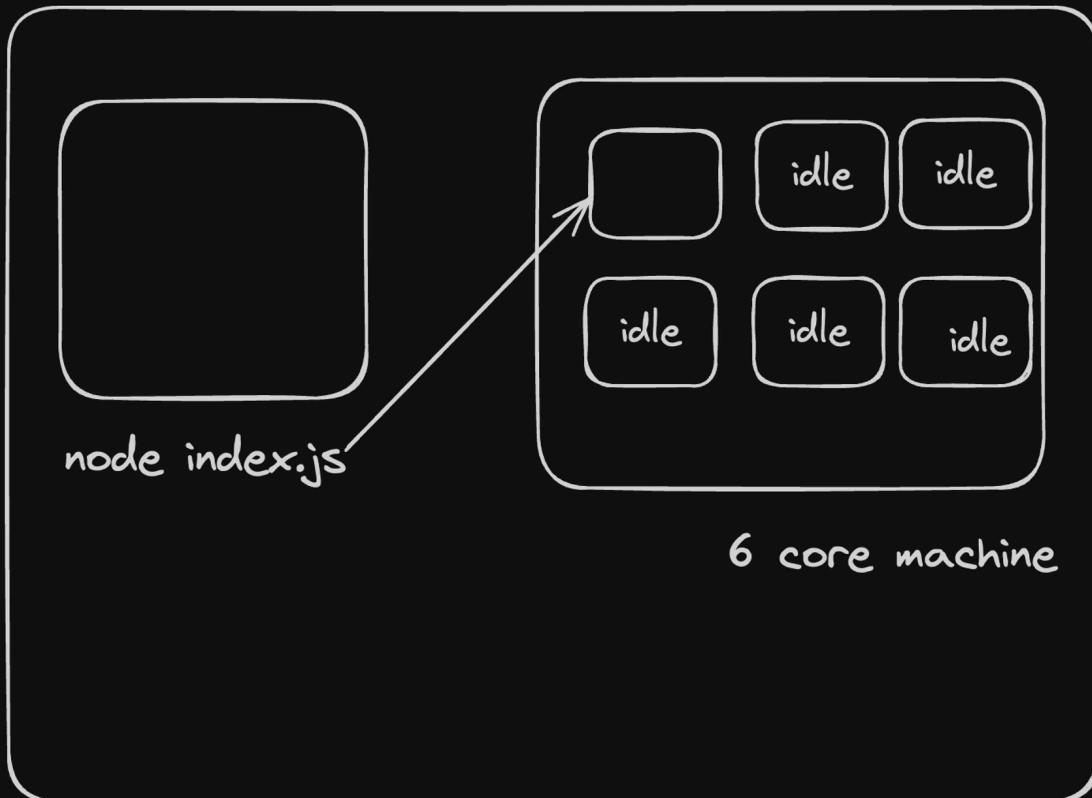


Single-threaded Languages

Single-threaded languages, like JavaScript, execute code in a single sequence or thread of execution. This means that at any given time, only one task or operation can be processed.



Mac Machine / AWS Machine



In the case of JavaScript, the language itself is single-threaded, but the runtime environment (e.g., Node.js) can leverage additional threads for specific tasks, such as I/O operations or computationally intensive tasks offloaded to worker threads.

Here's an example of an infinite loop in JavaScript that demonstrates how it utilizes a single CPU core:

```
let c = 0;  
while (1) {  
    c++;  
}
```



This code will continuously increment the `c` variable, effectively creating an infinite loop that consumes a single CPU core at 100% utilization.



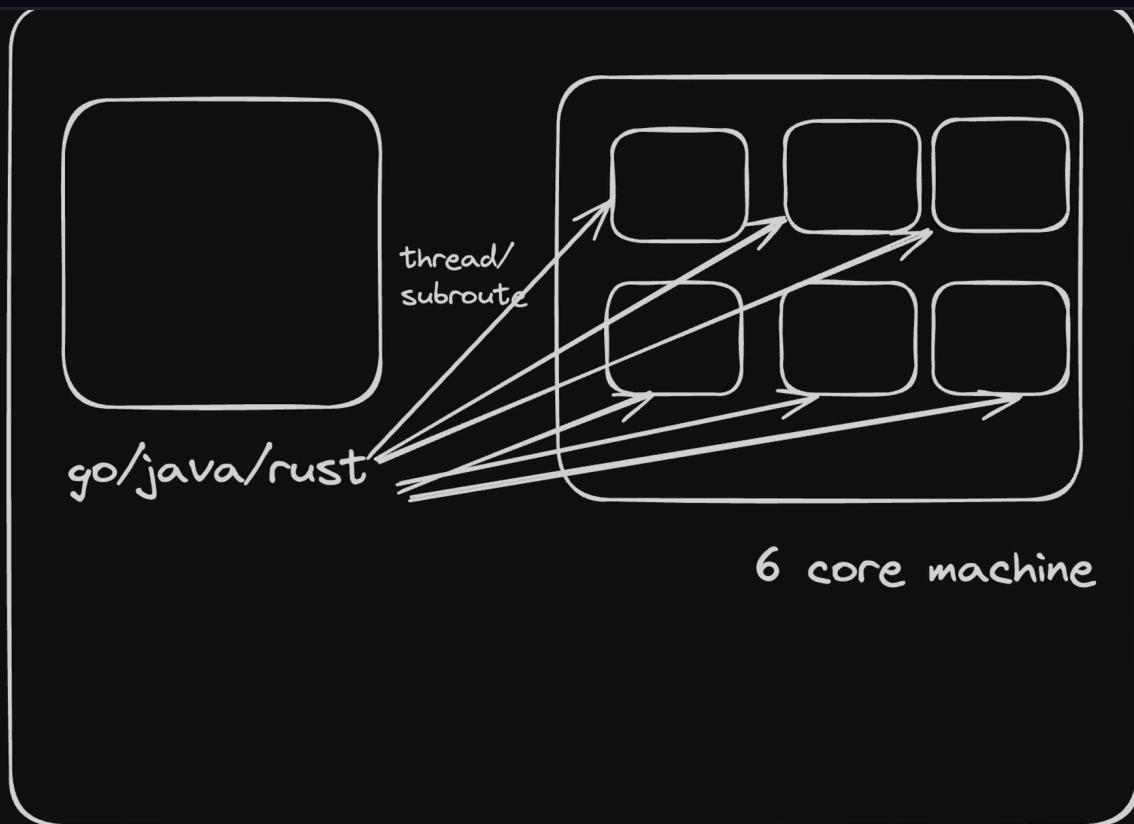
```
5918 horkrotsi 25 0 3900 36832 7 97.3 0.1 0:06.00 node index.js
75115 horkrotsi 17 0 3900 37082 7 97.0 0.1 0:21.00 node index.js
75109 horkrotsi 17 0 3900 36432 7 96.8 0.1 0:32.00 node index.js
63365 horkrotsi 17 0 3900 25124 7 95.0 0.5 49:21.00 /System/Applications/Utilities/System Information.app/Contents/Resources/Java/JRE.app/Contents/MacOS/JRE
63955 horkrotsi 17 0 3920 16468 7 34.3 2.5 31:56.00 /Applications/OSX.app/Contents/Auxiliary/OSX.app/Contents/Frameworks/Google Chrome Helper
72925 horkrotsi 24 0 1320 65917 7 34.0 2.5 0:00.00 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome
3110 horkrotsi 17 0 3900 16001 7 33.9 0.6 1:57.00 /Applications/Brave Browser.app/Contents/Frameworks/Brave Browser
72644 horkrotsi 17 0 1320 59792 7 15.8 0.8 14:59.00 /Applications/Brave Browser.app/Contents/Frameworks/Brave Browser
82124 horkrotsi 17 0 4216 28291 7 14.6 0.4 0:05:23 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome
8605 horkrotsi 17 0 4026 27504 7 11.6 4.2 12:12.00 /Applications/Adobe Premiere Pro 2023/Adobe Premiere Pro 2023
75921 horkrotsi 17 0 3900 23680 7 7.8 0.0 0:21.00 /usr/bin/screencapture -pd1 -z keyboard.selection
62427 horkrotsi 17 0 15206 29394 7 4.3 0.5 23:29.00 /Applications/Brave Browser.app/Contents/Frameworks/Brave Browser
45395 horkrotsi 17 0 4216 73680 7 4.1 0.1 49:32.00 /private/var/folders/zh/zapl7zj4d81gp5whwx16r/w000gn/T/AppTr
17 0 4216 21949 7 4.0 0.4 51:25.00 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome
13604 horkrotsi 17 0 3935 47392 7 3.6 0.0 0:00.00 /Applications/Postman.app/Contents/Frameworks/Postman Helper
67306 horkrotsi 24 0 3900 25124 7 3.5 2.0 0:15:20.00 /Applications/Utilities/Tasker.app/Contents/MacOS/Tasker
68092 horkrotsi 17 0 3900 16491 7 3.5 0.4 13:37.00 /Applications/Brave Browser.app/Contents/Frameworks/Brave Browser
2573 horkrotsi 17 0 3900 66964 7 3.2 1.0 50:48.00 /System/Library/Frameworks/Virtualization.framework/Versions/1.0/Support/VMware Fusion.app/Contents/MacOS/Fusion
2615 horkrotsi 17 0 3916 34864 7 2.4 0.1 1:14.00 /Applications/Docker.app/Contents/MacOS/Docker Desktop.app/Contents/MacOS/Docker Desktop
9347 horkrotsi 24 0 15206 45887 7 2.3 0.7 39:34.00 /Applications/Discord.app/Contents/Frameworks/Discord Helper
45399 horkrotsi 17 0 13206 53994 7 2.3 0.8 34:27.00 /private/var/folders/zh/zapl7zj4d81gp5whwx16r/w000gn/T/AppTr
9342 horkrotsi 17 0 4216 76704 7 2.0 0.1 45:21.00 /Applications/Discord.app/Contents/Frameworks/Discord Helper
855 horkrotsi 17 0 3916 52326 7 1.9 0.1 0:22.00 /Applications/Utilities/Adobe Creative Cloud/ACC/Creative Cloud
72590 horkrotsi 17 0 3930 56269 7 1.7 0.9 37:45.00 /Applications/WhatsApp.app/Contents/MacOS/WhatsApp
773 horkrotsi 24 0 3900 25124 7 1.6 0.9 0:00.00 /Applications/Utilities/Stream Deck.app/Contents/MacOS/Stream Deck
82120 horkrotsi 17 0 3900 16719 7 1.5 3.4 0:53:15.00 /Applications/Google Home.app/Contents/MacOS/Google Home
87526 horkrotsi 17 0 3900 19384 7 1.4 0.3 0:35:44.00 /Applications/zoom.us.app/Contents/MacOS/zoom
64000 horkrotsi 17 0 3900 56544 7 1.4 0.1 6:22.00 /Applications/OSX.app/Contents/Frameworks/OSX Helper (GPU).app
793 horkrotsi 17 0 3900 9792 7 1.3 0.9 51:03.00 /Applications/OllyDbg.app/Contents/Frameworks/OllyDbg Helper (GP)
70829 horkrotsi 17 0 13206 56101 7 1.3 0.9 2:23.00 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome
68903 horkrotsi 17 0 3916 12801 7 1.2 0.2 1:21.50 /System/Applications/Podcasts.app/Contents/MacOS/Podcasts
75920 horkrotsi 17 0 3900 23120 7 0.9 0.0 0:00.00 /System/Library/CoreServices/screencaptureutil.app/Contents/MacOS/screencaptureutil
87532 horkrotsi 17 0 3916 24088 7 0.8 0.0 49:18.00 /Applications/zoom.us.app/Contents/Frameworks/ZoomClient.app/Contents/MacOS/ZoomClient
7320 horkrotsi 24 0 13206 53994 7 0.8 0.0 0:00.00 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome
82344 horkrotsi 17 0 4216 18656 7 0.5 0.0 15:37.00 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome
3104 horkrotsi 17 0 4216 53994 7 0.3 0.8 57:43.00 /Applications/Brave Browser.app/Contents/MacOS/Brave Browser
73313 horkrotsi 24 0 3900 16864 7 0.2 0.0 0:00.00 /Applications/OSX.app/Contents/Frameworks/OSX Helper (CPU)
46063 horkrotsi 24 0 13116 42272 7 0.2 0.1 5:55.00 /private/var/folders/zh/zapl7zj4d81gp5whwx16r/w000gn/T/AppTr
517 horkrotsi 17 0 3936 14896 7 0.2 0.0 1:12.00 /System/Library/PrivateFrameworks/HomectlDaemon.framework/Supp
1397 horkrotsi 17 0 3916 11134 7 0.2 0.2 1:54.00 /Library/Application Support/Adobe/Creative Cloud Libraries/CC
521 horkrotsi 24 0 3936 36016 7 0.1 0.1 4:24.00 /usr/libexec/sharingd
9339 horkrotsi 17 0 13120 13881 7 0.1 0.2 8:34.00 /Applications/Discord.app/Contents/MacOS/Discord
799 horkrotsi 17 0 3936 6320 7 0.1 0.0 8:34.00 /Library/Application Support/Adobe/Adobe Desktop Common/IPCBox
72487 horkrotsi 17 0 13990 6958 7 0.1 1.0 0:51.00 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome
```

Multi-threaded Languages

Multi-threaded languages, like Rust, can create and manage multiple threads of execution

[Home](#) > 0-100 > Week 22 > 22.1 | Notes

achieve better performance for certain types of workloads, especially those that can be parallelized.



In Rust, you can create and manage threads using the `std::thread` module. Here's an example that spawns three threads, each running an infinite loop:

```
use std::thread;

fn main() {
    // Spawn three threads
    for _ in 0..3 {
        thread::spawn(|| {
            let mut counter: f64 = 0.00;
            loop {
                counter += 0.001;
            }
        });
    }

    loop {
        // Main thread does nothing but keep the program alive
    }
}
```



In this example, three threads are created, each incrementing a floating-point counter in an infinite loop. The main thread simply runs an empty loop to keep the program alive. This code will utilize multiple CPU cores, with each thread consuming a portion of the available resources.

The screenshot shows a terminal window with two panes. The left pane displays the output of the command `htop`, which monitors CPU usage across multiple cores. The right pane shows the output of the command `cargo run` for a Rust project named `rust-project`. The cargo output includes a warning about an unused variable and the generated assembly code for the local counter.

```
htop (htop)
CPU: 0[██████████] 1[██████████] 2[██████████] 3[██████████] 4[██████████] 5[██████████] 6[██████████] 7[██████████] 8[██████████] 9[██████████] Mem: 128.7G/64.00G Swap: 3.08G/4.00G tasks: 766, 3320 thr, 0 kthr; 9 running Load average: 19.15 14.93 15.02 Uptime: 8 days, 01:58:45

PID USER PRI NI VIRT RES S CPU% MEMS TIME+ Command
76225 harkiratsi 17 0 389G 1744 ? 399 3 0.0 0:41.00 target/debug/rust-project
73563 harkiratsi 17 0 391G 213M ? 53 5 0.3 58:37.00 /System/Applications/1It11t1es/System Information.app/Contents/MacOS/OBS
63995 harkiratsi 17 0 392G 1646M ? 38.3 2.5 51 57:30 /Applications/OBS.app/Contents/MacOS/OBS
8605 harkiratsi 24 0 402G 2739M ? 10.9 4.2 12 14:56 /Applications/Adobe Premiere Pro 2023/Adobe Premiere Pro 2023
73296 harkiratsi 17 0 1523G 855M ? 1.9 1.3 4:39.00 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework
87806 harkiratsi 24 0 392G 637M ? 2.2 1.0 1h14:21 /Applications/iTerm.app/Contents/MacOS/iTerm2
87526 harkiratsi 24 0 392G 198M ? 1.4 0.3 4h36:05 /Applications/zoom.us.app/Contents/MacOS/zoom.us
75993 harkiratsi 17 0 1521G 357M ? 19.9 0.5 0:33.00 /Applications/Brave Browser.app/Contents/Frameworks/Brave Browser Framework
64247 harkiratsi 17 0 1520G 295M ? 3.8 0.5 24:09.00 /Applications/Brave Browser.app/Contents/Frameworks/Brave Browser Framework
82125 harkiratsi 17 0 421G 294M ? 5 5 0.4 52:08.00 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework

cargo (rust-project)
+ rust-project git:(master) x cargo run
Compiling rust-project v0.1.0 (/Users/harkiratsingh/Projects/rust-project)
warning: variable `counter` is assigned to, but never used
--> src/main.rs:7:21
7 |     let mut counter: f64 = 0.00; // Local counter
   |           ^^^^^^
= note: consider using `&counter` instead
= note: #[warn(unused_variables)] on by default
warning: `rust-project` (bin "rust-project") generated 1 warning
Finished dev [unoptimized + debuginfo] target(s) in 0.23s
Running `target/debug/rust-project`
```

It's important to note that while multi-threaded languages can leverage multiple CPU cores, they also introduce additional complexity in terms of thread synchronization, shared memory access, and potential race conditions. Proper synchronization mechanisms, such as mutexes or message passing, must be employed to ensure thread safety and avoid data corruption or undefined behavior.

In summary, vertical scaling involves increasing the hardware resources of a single machine, while the choice between single-threaded and multi-threaded languages depends on the specific requirements of the application and the need for parallel processing or leveraging multiple CPU cores.

Implementing Horizontal Scaling in Node.js Project

Node.js, being a single-threaded language, can benefit from horizontal scaling to take advantage of multiple CPU cores and distribute the workload across multiple processes. However, manually starting multiple Node.js processes can be cumbersome and prone to issues like port conflicts and process management. This is where the `cluster` module comes into play.



Follow 100xDevs

Here's an example of how to implement horizontal scaling using the cluster module in a Node.js project:

```
import express from "express";
import cluster from "cluster";
import os from "os";

const totalCPUs = os.cpus().length;
const port = 3000;

if (cluster.isPrimary) {
  console.log(`Number of CPUs is ${totalCPUs}`);
  console.log(`Primary ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < totalCPUs; i++) {
    cluster.fork();
  }

  cluster.on("exit", (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
    console.log("Let's fork another worker!");
    cluster.fork();
  });
} else {
  const app = express();
  console.log(`Worker ${process.pid} started`);

  app.get("/", (req, res) => {
    res.send("Hello World!");
  });

  app.get("/api/:n", function (req, res) {
    let n = parseInt(req.params.n);
    let count = 0;

    if (n > 5000000000) n = 5000000000;
  });
}
```



```
    res.send(`Final count is ${count} ${process.pid}`);
  });

  app.listen(port, () => {
    console.log(`App listening on port ${port}`);
  });
}
```

In this example, we first check if the current process is the primary process using `cluster.isPrimary`. If it is, we fork worker processes equal to the number of available CPU cores using `cluster.fork()`. We also set up an event listener for the `exit` event, which forks a new worker process if an existing worker dies.

If the current process is a worker process, we create an Express app and define routes for handling requests. In this example, we have two routes: one for returning a simple "Hello World!" message, and another for performing a CPU-intensive task (summing up numbers from 0 to a specified value).

When you run this code, you'll see output similar to the following:

```
Number of CPUs is 8
Primary 12345 is running
Worker 67890 started
Worker 23456 started
Worker 78901 started
...
```



You can then send requests to the server using different tools like a web browser, Postman, or cURL, and you'll notice that the requests are being handled by different worker processes, as indicated by the different process IDs (PIDs) in the response.

Stickiness in the Browser



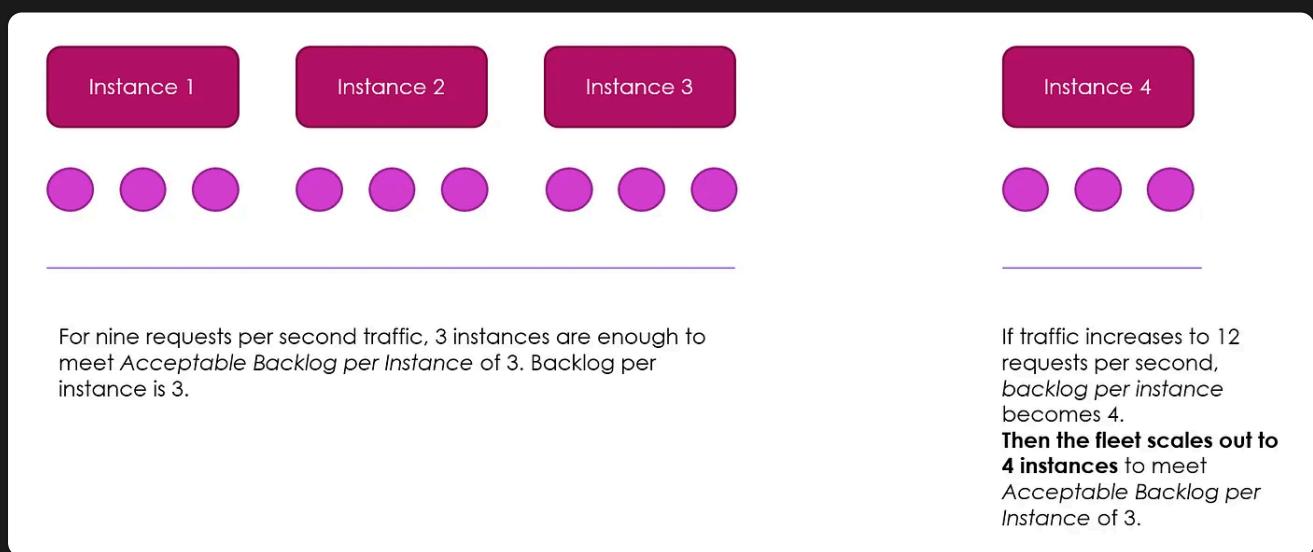
performance.

Browsers typically establish a persistent connection with the server and reuse it for subsequent requests to the same domain. This connection is associated with a specific worker process, leading to the observed stickiness.

To mitigate this behavior and distribute requests more evenly across worker processes, you can configure your server to use load balancing techniques or employ a reverse proxy server that manages the distribution of requests across multiple worker processes.

Capacity Estimation

Capacity estimation is a critical aspect of system design, especially when planning for scalability and reliability. In a system design interview, you may be asked how you would scale your server, handle traffic spikes, and support a certain Service Level Agreement (SLA) given specific traffic conditions. This section will elaborate on these points and explain the attached architecture diagrams in detail.



How Would You Scale Your Server?

Scaling a server involves increasing its capacity to handle more load. There are two primary methods of scaling:



2. **Horizontal Scaling (Scaling Out):** Adding more servers to distribute the load. This approach is more flexible and can handle larger increases in traffic.

How Do You Handle Spikes?

Handling traffic spikes requires a dynamic approach to scaling. Here are some strategies:

1. **Auto Scaling:** Automatically adding or removing servers based on the current load. This ensures that the system can handle sudden increases in traffic without manual intervention.
2. **Load Balancing:** Distributing incoming requests across multiple servers to ensure no single server is overwhelmed.
3. **Caching:** Storing frequently accessed data in a cache to reduce the load on the servers.

How Can You Support a Certain SLA Given Some Traffic?

Supporting a specific SLA (e.g., 99.9% uptime) requires careful planning and monitoring:

1. **Redundancy:** Having multiple servers and data centers to ensure that if one fails, others can take over.
2. **Monitoring and Alerts:** Continuously monitoring the system's performance and setting up alerts for any issues.
3. **Capacity Planning:** Estimating the required capacity based on current and projected traffic and ensuring that the system can handle it.

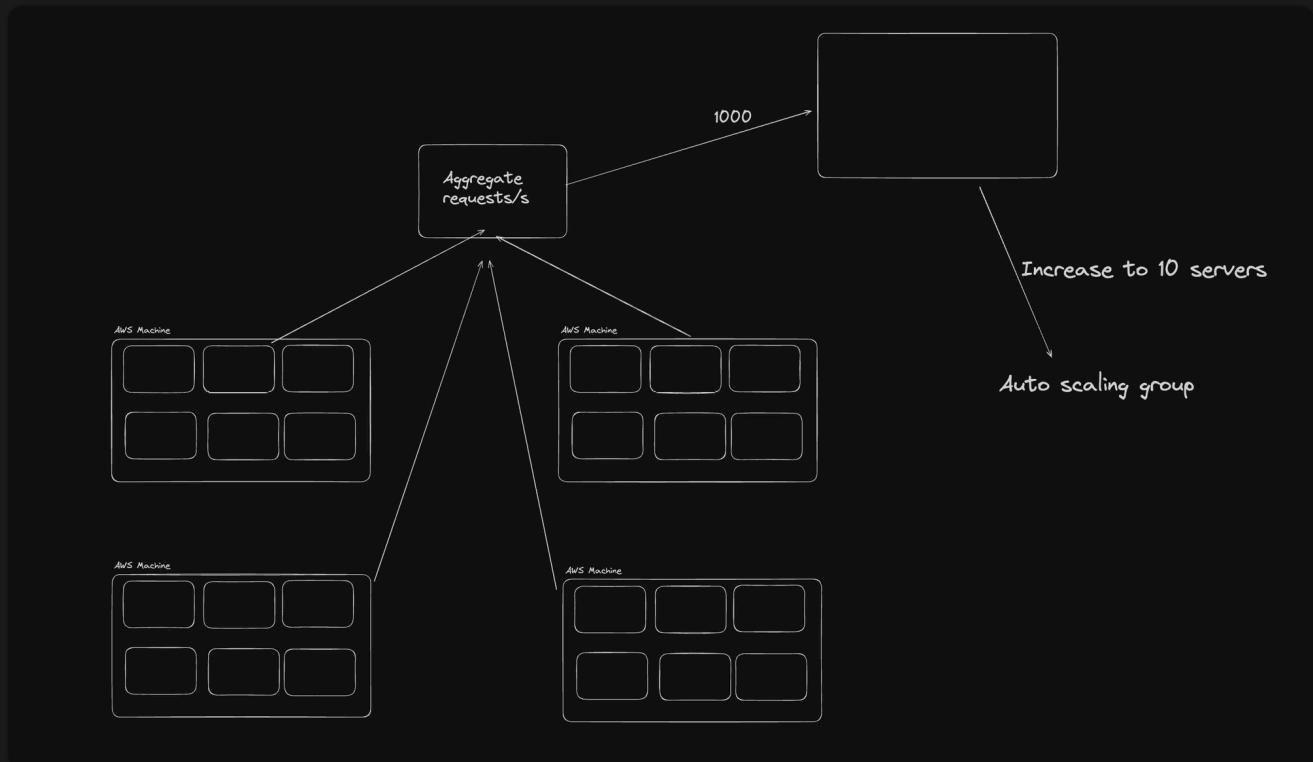
Paper Math for Capacity Estimation

Capacity estimation often involves some basic calculations:

1. **Estimating Requests per Second (RPS):** Based on the number of users and their activity patterns.
2. **Determining Machine Capacity:** Estimating how many requests a single machine can handle based on its specifications and the nature of the workload.
3. **Auto Scaling:** Setting up rules to add or remove machines based on the load.

Example 1: PayTM App

- **Auto Scaling Groups:** To handle varying loads.
- **Load Balancers:** To distribute traffic.
- **Database Sharding:** To manage large volumes of transaction data.



Aggregate Requests and Auto Scaling

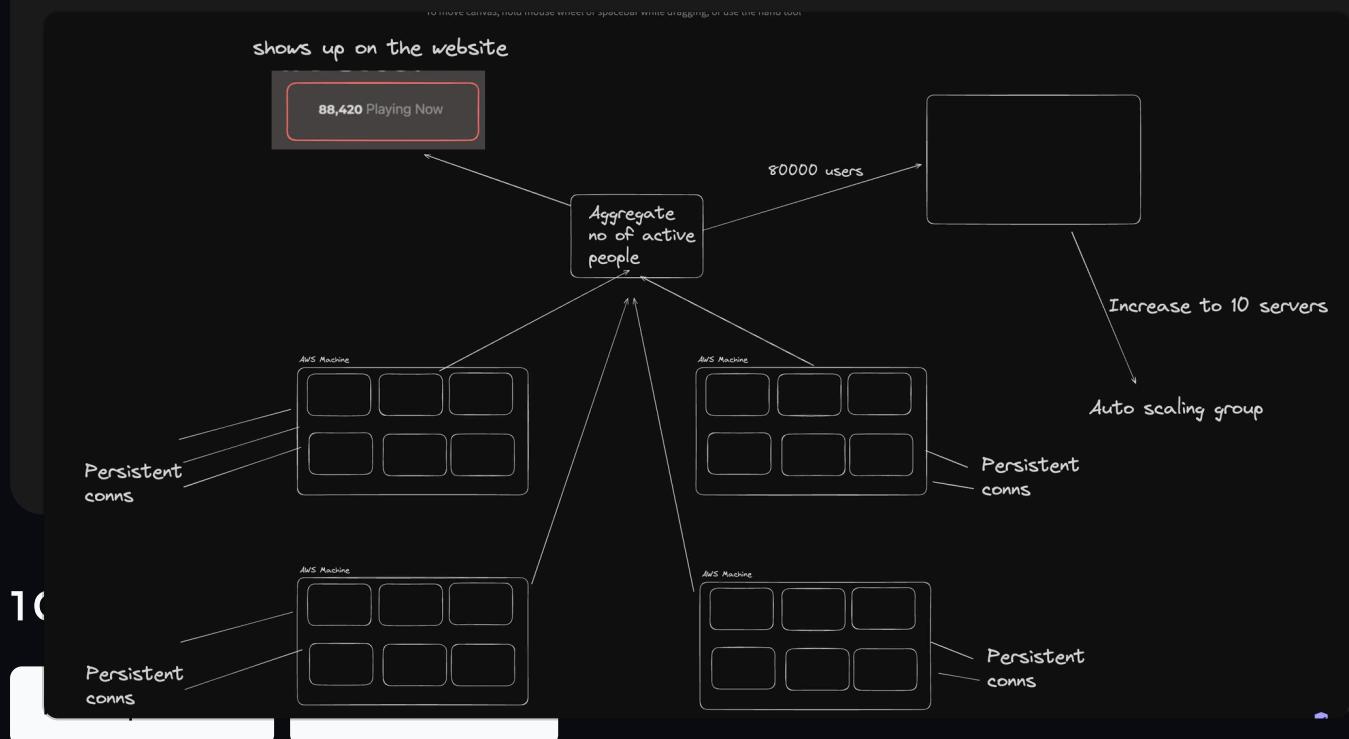
This diagram shows a system designed to handle aggregate requests with auto-scaling capabilities:

1. **Aggregate Requests:** The system aggregates incoming requests.
2. **Auto Scaling Groups:** These groups dynamically scale the number of servers based on the load.
3. **Auto Machines:** Individual processing units that handle the requests. The system can increase the number of these machines to handle higher loads, up to 10 servers

Example 2: Chess App

For a chess app, the focus might be on real-time interactions and maintaining game state.

- **Auto Scaling:** To handle peak times when many users are playing simultaneously.
- **Persistent Connections:** To maintain game state and player interactions.



Aggregate Number of Active People

Add a comment...

This diagram focuses on tracking the number of active users and scaling accordingly:

1. **Aggregate Number of Active People:** The system tracks the number of active users, shown as 90,000 users.

2. **Persistent Connections:** Each server maintains persistent connections with users.

Adarsh Rawat 2 months ago

3. **Auto Scaling Groups:** The system can increase the number of servers to handle the load, up to 10 servers.

1 0 0 Replies

4. **Playing Now Metric:** Shows the number of users currently active, e.g., 88,420.

Capacity estimation involves understanding the current and projected load, determining the capacity of individual machines, and setting up auto-scaling to handle varying traffic. By using techniques like load balancing, caching, and redundancy, you can ensure that your system meets its SLA and handles traffic spikes effectively. The provided



100xDevs

