

Assume you are expert in k6 for performance testing using javascript, and a good guide to explain step by step understanding all its key features then Answer the following question to a beginner "What is K6? Why K6 for performance testing? "

Absolutely! Let's break it down in a beginner-friendly and clear way 📌

🧠 What is K6?

K6 is an **open-source performance testing tool** built using **JavaScript**. It helps you check how your website, API, or backend system performs under **load or stress**—meaning when many users are using it at the same time.

It's **developer-friendly**, fast, scriptable, and designed for modern DevOps and CI/CD pipelines.

🚀 Why Use K6 for Performance Testing?

Here are **simple and strong reasons** why K6 is a great choice:

1. ✅ Script with JavaScript

- K6 uses plain JavaScript for writing test scripts.
- If you know basic JavaScript, you're already ahead!

2. ⚡ High Performance

- It's built in **Go** (a super-fast language), so it can simulate **thousands of virtual users** with low resource usage.

3. 🛠️ Great Developer Experience

- You write tests in code, which means you can version control, reuse, and customize tests easily.
- No clunky UI—everything is script-based and programmable.

4. 📦 CI/CD Integration

- Easy to plug into pipelines (like GitHub Actions, Jenkins, GitLab, etc.).
- Automate performance tests on every deployment.

5. 📊 Rich Results and Reporting

- K6 provides detailed output on performance stats like:
 - Response times
 - Errors
 - Request success rate
 - Custom thresholds and trends

6. ☁️ k6 Cloud (Optional)

- If you want advanced visualization and large-scale load testing, K6 offers a **cloud platform** to complement the local open-source tool.

💡 Example Use Case

You built a login API and want to know:

“Can it handle 1000 users trying to log in at once?”

K6 lets you simulate those 1000 users with a simple JavaScript script and tells you if the API can handle the load, or if it crashes, slows down, or misbehaves.

📌 Summary:

Feature	Benefit
JavaScript Scripting	Easy and flexible test writing
Open Source & Free	No cost to get started
CLI Tool	Lightweight and fast
CI/CD Friendly	Automate performance testing
Modern Reports	Get deep insights & trends



If you're interested, I can show you a simple K6 script next! Want to try writing your first one? 😊







Advantages and disadvantages of k6

Absolutely! Let's break it down in a beginner-friendly way.

Question: What are the advantages and disadvantages of k6 in performance testing?

Advantages of k6

-  **Developer-Friendly (JavaScript-based)**
 - k6 uses JavaScript to write test scripts, which is easy to learn and very familiar for web developers.
 - This lowers the entry barrier for writing performance tests.
-  **High Performance and Speed**
 - k6 is written in **Go** (a high-performance language), making it **lightweight and fast**.

- It can generate thousands of virtual users (VUs) from a single machine.
- 3.  **Supports Load, Stress, Spike, and Soak Testing**
 - You can simulate a variety of real-world performance scenarios using flexible test configurations.
- 4.  **Built-in Metrics & Reporting**
 - k6 provides detailed metrics like response time, requests per second, errors, etc.
 - These can be viewed in the terminal or sent to external systems (like InfluxDB, Grafana, or JSON files).
- 5.  **CI/CD Integration**
 - Easily integrates with Jenkins, GitHub Actions, GitLab CI, etc.
 - Enables **automation** of performance testing in pipelines.
- 6.  **HTTP and WebSocket Support**
 - You can test **REST APIs**, **GraphQL**, **WebSockets**, etc.
- 7.  **Open Source and Extensible**
 - Free to use, and has a growing community.
 - You can even write extensions if needed.
- 8.  **Scalable via k6 Cloud (Optional)**
 - If you need more power, k6 Cloud allows you to scale your tests across multiple machines.

Disadvantages of k6

1.  **No GUI (Command-line Only)**

- k6 is CLI-based. Beginners who prefer graphical tools (like JMeter) may find it challenging at first.

2. 🧩 Limited Protocol Support

- Only supports **HTTP(s)** and **WebSockets** out of the box.
- No built-in support for protocols like FTP, SMTP, or SAP unless you extend it manually.

3. 🚫 No Built-in Recorder

- Unlike tools like JMeter or Postman, k6 doesn't have a test script recorder.
- You need to manually write your test scripts in JavaScript.

4. 📖 Learning Curve for Advanced Features

- While basic scripting is easy, advanced scenarios (e.g., custom thresholds, scenarios, parameterization) require deeper understanding of JavaScript and k6's APIs.

5. 💰 Cloud is Paid

- The open-source tool is free, but **k6 Cloud** (for distributed testing, advanced dashboards) requires a subscription.

🔍 Summary Table

Feature	Advantage ✅	Disadvantage ❌
Language	Uses JavaScript – easy for devs	Requires scripting knowledge
Performance	Lightweight & fast (written in Go)	
Protocol Support	HTTP/WebSocket support	Limited to HTTP/WebSocket
Usability	Script-based (CI/CD friendly)	No GUI or script recorder
Reporting	Detailed built-in metrics, exportable	Basic reporting in CLI

Extensibility	Open source & extendable	Extending needs coding expertise
Cloud Option	Can scale tests via k6 Cloud	Cloud version is paid

Would you like me to show a small beginner-friendly example of a basic **k6 test script** with explanations?

Assume you are expert in k6 for performance testing using javascript, and a good guide to explain step by step understanding all its key features then Answer the following question to a beginner "CLI Basics (k6 run, k6 archive, k6 inspect)"

Absolutely! Let's break down the **CLI Basics** of **k6** in a beginner-friendly way, covering:

- `k6 run`
- `k6 archive`
- `k6 inspect`

These commands are essential for running, preparing, and inspecting your k6 performance test scripts.

1. **k6 run**

♦ **What it does:**

This command **executes** your k6 script locally.

Syntax:

k6 run script.js

✓ Example:

```
k6 run test.js
```

📌 What happens:

- It will start executing your test script (`test.js`)
- You'll see real-time output in the terminal:
 - number of virtual users (VUs)
 - duration
 - passed/failed requests
 - performance metrics (e.g., HTTP response times)

🔧 Options:

```
k6 run --vus 10 --duration 30s script.js
```

This means:

- Use **10 virtual users**
 - Run the test for **30 seconds**
-

✓ 2. k6 archive

♦ What it does:

This command **packages** your k6 script and its dependencies into a **.tar** file (called a bundle/archive). It's useful for running the test later or on a remote machine (like k6 Cloud or Docker).

📦 Syntax:

```
k6 archive script.js -o bundle.tar
```

✅ Example:

```
k6 archive test.js -o test-bundle.tar
```

📌 What happens:

- It creates a `.tar` file that includes:
 - your script
 - imported modules
 - any external files (like JSON, data, etc.)

You can now move this archive and run it anywhere with `k6 run`.

✅ 3. `k6 inspect`

♦ What it does:

It shows a **summary** of what's inside a script or an archive without actually running the test.

📦 Syntax:

```
k6 inspect script.js
```

or

```
k6 inspect archive.tar
```

✅ Example:

```
k6 inspect test.js
```

📌 What it tells you:

- Which default function is being exported
- What scenarios, stages, VUs, and duration are defined
- Useful for **debugging or previewing** before actual test execution



Summary Table

Command	Purpose	Example
<code>k6 run</code>	Runs a script	<code>k6 run test.js</code>
<code>k6 archive</code>	Creates a bundle (.tar)	<code>k6 archive test.js -o test.tar</code>
<code>k6 inspect</code>	Shows script or archive info	<code>k6 inspect test.js</code>

Let me know if you want a sample test file to try these commands on, or want to explore `k6 cloud` next!

Assume you are expert in k6 for performance testing using javascript, and a good guide to explain step by step understanding all its key features when necessary and explain well about code in form of comments then Answer the following question to a beginner "Write a simple script to get started"

Absolutely! Let's get you started with a **simple K6 performance testing script using JavaScript**, explained step-by-step with comments for a complete beginner.

✓ Step 1: What is K6?

K6 is an open-source **performance testing tool** written in Go but scripts are written in **JavaScript**. It helps simulate **virtual users** (VUs) hitting your APIs or web services and measures performance like **response time, success rate, and more**.

✓ Step 2: Simple Script to Get Started with K6

```
import http from 'k6/http';    // Import the HTTP module to make requests
import { check } from 'k6';    // Import the check module to assert test conditions

// Options block: defines how the test will run
export const options = {
  vus: 10,          // Number of virtual users (VUs) - simulate 10 users
  duration: '10s',  // Test duration - run the test for 10 seconds
};

// The default function is like the "main" function - it's called by each virtual user
export default function () {
  // Make an HTTP GET request to the test URL
  const response = http.get('https://test-api.k6.io');

  // Check if the response status is 200 OK
  check(response, {
    'status is 200': (r) => r.status === 200,
  });
}
```

✓ Step 3: How to Run This Script

1. **Save the file** as `script.js`

Run it using the terminal:

```
k6 run script.js
```

- 2.
-

✓ Output You'll See

You'll get a report in the terminal with:

- **Checks passed/failed**
- **Request duration**
- **Throughput (requests/sec)**
- **HTTP status codes**

✓ Explanation of Key Concepts

Concept	Meaning
<code>vus</code>	Virtual Users - how many simulated users hit the system at once
<code>duration</code>	Total time the test runs
<code>http.get()</code>	Makes a GET request to a URL
<code>check()</code>	Used to validate the response (like asserting in unit tests)
<code>response</code>	The data received from the HTTP request

Let me know if you'd like to test **POST requests**, add **metrics**, or explore **stages (ramping up users)** next!

Assume you are:

Expert in k6 for performance testing using javascript

Good guide to explain step by step understanding all its key features when necessary

Explain well about code in form of comments

then Answer or Provide details about the following question to a beginner "VUs (Virtual Users) vs Iterations"

Absolutely! Let's break it down for you in a beginner-friendly, clear way.

We're discussing **K6**, a powerful open-source tool used for **performance testing** written in **JavaScript**, and the topic is:

◆ VUs (Virtual Users) vs Iterations in K6

These two concepts are **central to how K6 works**, but they mean very different things.

✓ 1. What is a VU (Virtual User)?

- A **VU** is a *simulated user* created by K6.
- It **executes your script just like a real user would**, but virtually.
- Each VU runs **independently** in a loop.
- VUs are **not real system users**—they're lightweight, simulated threads that generate traffic.

Think of 10 VUs as 10 people opening your website and doing the same task.



✓ 2. What is an Iteration?

- An **iteration** is a *single complete execution* of the default function by a VU.
- Each time a VU finishes executing your test script, that counts as **1 iteration**.
- A VU can perform **multiple iterations** depending on the duration of the test.

If 1 VU runs your script 5 times, that's 5 iterations.

Simple Analogy

Imagine you're testing a checkout flow of an online store.

-  Each **VU** is like a **customer**.
-  Each **iteration** is like **one shopping trip**.

So if:

- You have **5 customers (VUs)**
- Each customer shops **3 times (iterations)**

You get: **15 total iterations**

Sample K6 Code with Explanation

```
import http from 'k6/http';
import { sleep } from 'k6';

export let options = {
  vus: 2, // Number of Virtual Users
  duration: '10s', // How long to keep the test running
};

// This function is executed by each VU
export default function () {
  http.get('https://test-api.k6.io'); // VU makes HTTP GET request
  sleep(1); // VU waits 1 second (simulate user think time)
}
```

What happens in this script?

- 2 VUs are created.
- They run the test for 10 seconds.
- Each VU runs the `default` function in a **loop** for 10 seconds.
- Each time a VU finishes the loop, **1 iteration is counted**.

So you may get around:

- ~5 iterations per VU (depending on sleep time and server response),
- ~10 total iterations in 10 seconds.



Summary Table

Concept	Description	Example Value
VU	Virtual simulated user	10
Iteration	One full run of your script by a VU	100
Duration	How long VUs should run the script	'30s'
Loop Behavior	VUs repeat your script in a loop	Until duration ends



Key Takeaways

- **VUs = number of users**, like how many people are using the system.
 - **Iterations = how many times** each user repeats the test script.
 - **Duration controls how long** they keep repeating.
-

If you'd like, I can show you how changing VUs or duration affects performance test results. Want a deeper example or test plan?

Assume you are: Expert in k6 for performance testing using javascript Good guide to explain step by step understanding all its key features when necessary Explain well about code in form of comments and Ensure code must be from beginner to Advance then Answer or Provide details about the following question to a beginner "Options block – setting up: ○ vus ○ duration ○ iterations "

Great! Since you're a beginner diving into **k6 for performance testing**, let's take a step-by-step approach to understand the **options block**, which is the heart of configuring your test scenarios.

What is the **options** block in k6?

The **options** block in a k6 script is used to define how the test should run — like how many **virtual users (VUs)** to simulate, how long to run the test, or how many total **iterations** (executions of the default function) to perform.

You can define it like this:

```
export const options = {  
  vus: 10,      // Number of virtual users  
  duration: '30s', // Duration the test will run  
};
```

Now let's break down each component you asked about.

✓ 1. **vus** – Virtual Users

What it does:

- It defines **how many virtual users** will run concurrently.
- Each virtual user executes the **default** function independently.

Beginner Code Example:

```
import http from 'k6/http';

export const options = {
  vus: 5,           // Simulate 5 users at the same time
  duration: '10s', // Run the test for 10 seconds
};

export default function () {
  http.get('https://test.k6.io'); // Each user sends this GET request repeatedly
}
```

Explanation:

- In the 10 seconds of test time, all 5 users will send as many requests as they can.
 - This is a good basic **load test** to see how the server handles concurrent traffic.
-

✓ 2. **duration** – How Long the Test Runs

What it does:

- It specifies the **total time** the test will keep running.
- Works together with **vus**.

Duration Formats:

- You can write: '10s', '1m', '1h30m', etc.

Example Recap:

```
export const options = {  
  vus: 5,  
  duration: '30s', // Run test for 30 seconds  
};
```

✓ 3. **iterations** – Total Number of Executions

What it does:

- Instead of running for a time duration, you can **specify the number of times** to execute the **default** function **in total**.

Important: When you use **iterations**, k6 ignores **vus** and **duration** unless explicitly defined.

Beginner Code Example:

```
import http from 'k6/http';  
  
export const options = {  
  iterations: 10, // Only run 10 total requests  
  vus: 1,        // Use only 1 virtual user to run them  
};  
  
export default function () {  
  http.get('https://test.k6.io');  
}
```

Explanation:

- This will simulate **a total of 10 requests**, run sequentially (or concurrently if you increase **vus**).
- This is useful for a **simple functional test** or **smoke test** to check if the endpoint works.

Summary Table

Option	Meaning	Use Case Example
<code>vus</code>	Number of users running in parallel	Simulating multiple users at once
<code>duration</code>	How long to run the test	Load testing over time
<code>iterations</code>	Total number of executions	Limited run for test or debugging

Bonus: Combining `vus` and `iterations`

You can also do:

```
export const options = {  
  vus: 2,  
  iterations: 10  
};
```

This will split the 10 iterations among 2 users — for example, each doing 5 iterations (roughly).

Would you like me to show how to write this using a **staged load test** (e.g., ramping up users gradually)? Or shall we go into advanced topics next like thresholds, checks, and performance metrics?

Let me know your pace 😊

Assume you are:

Expert in k6 for performance testing using javascript

Good guide to explain step by step understanding all its key features when necessary

Explain well about code in form of comments and

Ensure code must be from beginner to Advance

then Answer or Provide details about the following question to a beginner "● Basic checks and thresholds"

Absolutely, Routh Kiran! Let's break down "**Basic checks and thresholds**" in **k6** for **performance testing using JavaScript** from a **beginner to advanced** level — step-by-step, with detailed explanations and **comments directly inside the code**. 💡

✅ What are "Checks" and "Thresholds" in k6?

- **Checks** are test assertions written in your script to **validate the correctness** of the response data (like HTTP status codes, response content, etc.).
 - **Thresholds** are **pass/fail criteria** that you set at the test level to define **acceptable performance metrics** like response time, success rate, etc.
-

🧠 Think of it like this:

- **Checks** = *"Is the API response correct?"*
 - **Thresholds** = *"Is the system fast and reliable under load?"*
-

Let's start with a basic script that uses both checks and thresholds:

Beginner Script — Basic Check and Threshold

```
import http from 'k6/http';
import { check, sleep } from 'k6';

// Define performance thresholds
export let options = {
  thresholds: {
    // 95% of requests must finish in under 500ms
    http_req_duration: ['p(95)<500'],

    // 100% of checks should pass
    checks: ['rate==1.0'],
  },
};

export default function () {
  // Make an HTTP GET request
  let res = http.get('https://test-api.k6.io/public/crocodiles/');

  // Perform a basic check on the response
  check(res, {
    'status is 200': (r) => r.status === 200, // Must return HTTP 200
    'body is not empty': (r) => r.body.length > 0, // Body should contain data
  });

  // Pause for 1 second between iterations
  sleep(1);
}
```

Code Breakdown (with comments):

options block:

```
export let options = {
  thresholds: {
    http_req_duration: ['p(95)<500'], // 95% of requests must be under 500ms
    checks: ['rate==1.0'],           // All checks must pass
  }
}
```

```
},  
};
```

- **http_req_duration**: Built-in metric that measures request time.
- **p(95)<500**: 95th percentile should be < 500 milliseconds.
- **checks: ['rate==1.0']**: Ensures **all checks** (assertions) passed.

✅ **check()** function:

```
check(res, {  
  'status is 200': (r) => r.status === 200,  
  'body is not empty': (r) => r.body.length > 0,  
});
```

- Each line is an assertion.
- Returns **true** if it passes, otherwise **false**.
- k6 tracks the **percentage of passed checks** for reporting and threshold enforcement.

Output Sample:

When you run this with **k6 run script.js**, you'll get:

- ✓ status is 200
- ✓ body is not empty

```
checks.....: 100.00% ✓ 2 ✗ 0  
http_req_duration.....: p(95)=420ms
```

If the thresholds fail, k6 marks the test as **FAILED**.

Advance Usage

You can define **custom thresholds**, for example:

```
thresholds: {  
  'http_req_failed': ['rate<0.01'], // Less than 1% of requests should fail  
  'http_req_duration{staticAsset:yes}': ['p(90)<300'], // Custom tag usage  
}
```

And add **tags to metrics**:

```
http.get('https://site.com/script.js', { tags: { staticAsset: 'yes' } });
```

Summary Table:

Feature	Purpose	Example
<code>check()</code>	Validates response (assertion)	<code>status === 200</code>
<code>thresholds</code>	Sets pass/fail rules on metrics	<code>p(95)<500,</code> <code>rate==1.0</code>
<code>http_req_duration</code>	Response time metric	Measures speed
<code>checks</code>	Built-in metric for check pass rate	Used in <code>thresholds</code>
<code>tags</code>	Custom label for advanced thresholds	<code>staticAsset:yes</code>

Best Practices:

- Always use **checks** to validate business logic (e.g., correct content, status code).
- Always define **thresholds** to enforce performance SLAs.
- Use **tags** when you want separate thresholds for different request types.
- Use **summary reports** to analyze failures after the test.

Would you like the **next level** with **scenarios, stages, or custom metrics** in k6? Or a real-world example like load testing a login API?

Assume you are: Expert in k6 for performance testing using javascript Good guide to explain step by step understanding all its key features when necessary Explain well about code in form of comments and Ensure code must be from beginner to Advance then Answer or Provide details about the following question to a beginner "🎯 Milestone: Run a basic test with VUs and thresholds."

Absolutely! Let's break down the milestone "🎯 Run a basic test with VUs and thresholds" in **K6 (a modern performance testing tool written in JavaScript)** — step by step, like you're learning it from scratch.

🎯 Milestone: Run a Basic Test with VUs and Thresholds

📌 Goal:

- Learn how to simulate users (VUs - Virtual Users).
- Add **thresholds** to define performance goals.
- Understand what the test is doing **line by line**.

✅ 1. Setup

You should have:

- K6 installed → <https://k6.io/docs/getting-started/installation/>
- Any code editor like VS Code

Run this command to check installation:

k6 version

Concept Explanation

✓ VUs (Virtual Users):

They simulate **real users** hitting your system **concurrently**.

✓ Thresholds:

These are **pass/fail conditions** for your test. For example:

"Make sure 95% of requests finish in under 500ms."

If your app doesn't meet that, the test **fails**. Great for CI/CD!

Basic K6 Test with VUs and Thresholds

Here's the full script:

```
import http from 'k6/http'; // Import the HTTP module to make requests
import { check, sleep } from 'k6'; // Import helper functions

// Test configuration: VUs and duration
export const options = {
  vus: 5, // Number of Virtual Users
  duration: '10s', // How long the test will run
  thresholds: {
    // Define performance goals
    http_req_duration: ['p(95)<500'], // 95% of requests must complete below 500ms
    http_req_failed: ['rate<0.01'], // Less than 1% requests should fail
```



```

    },
  };

  // Default function: executed by each VU repeatedly for the duration
  export default function () {
    const res = http.get('https://test.k6.io'); // Make a GET request

    // Check if the response status is 200
    check(res, {
      'status is 200': (r) => r.status === 200,
    });

    sleep(1); // Wait for 1 second before next iteration (simulates user think-time)
  }

```

Step-by-Step Explanation:

Line	Explanation
<code>import http...</code>	Load the HTTP module to make requests
<code>options</code> block	Configures your test: VUs, duration, thresholds
<code>vus: 5</code>	5 users will run simultaneously
<code>duration: '10s'</code>	Each user will run test logic for 10 seconds
<code>thresholds</code>	Define SLAs (performance targets)
<code>default function</code>	Main logic each VU runs
<code>http.get(...)</code>	Sends a GET request to the test site
<code>check(...)</code>	Assertion to verify response is successful
<code>sleep(1)</code>	Simulate real user wait time

How to Run This Test

Save as: `basic-test.js`


Then run in terminal:

```
k6 run basic-test.js
```

You'll see output like this:






```
running (10.0s), 5/5 VUs, 50 complete and 0 interrupted iterations
```

```
...
```

```
http_req_duration...: avg=200ms p(95)=300ms =>  Passed
```

```
http_req_failed.....: rate=0.00 =>  Passed
```

What You've Learned (Milestone Achieved!):

-  Simulated concurrent users
 -  Defined performance thresholds
 -  Ran a full test and got metrics
 -  Passed if thresholds met,  failed if not
-

Next Steps (Coming Soon):

- Ramp up/down VUs
- Test different HTTP methods
- Set up load profiles (stages)
- Export test results (JSON, HTML)
- Integrate with CI tools (like GitHub Actions)

Let me know if you'd like me to turn this into a **PDF**, create a **flowchart**, or move on to the **next milestone**!