Presentation #3

THE NODE JS RUNTIME

Github Organization

CREATED BY

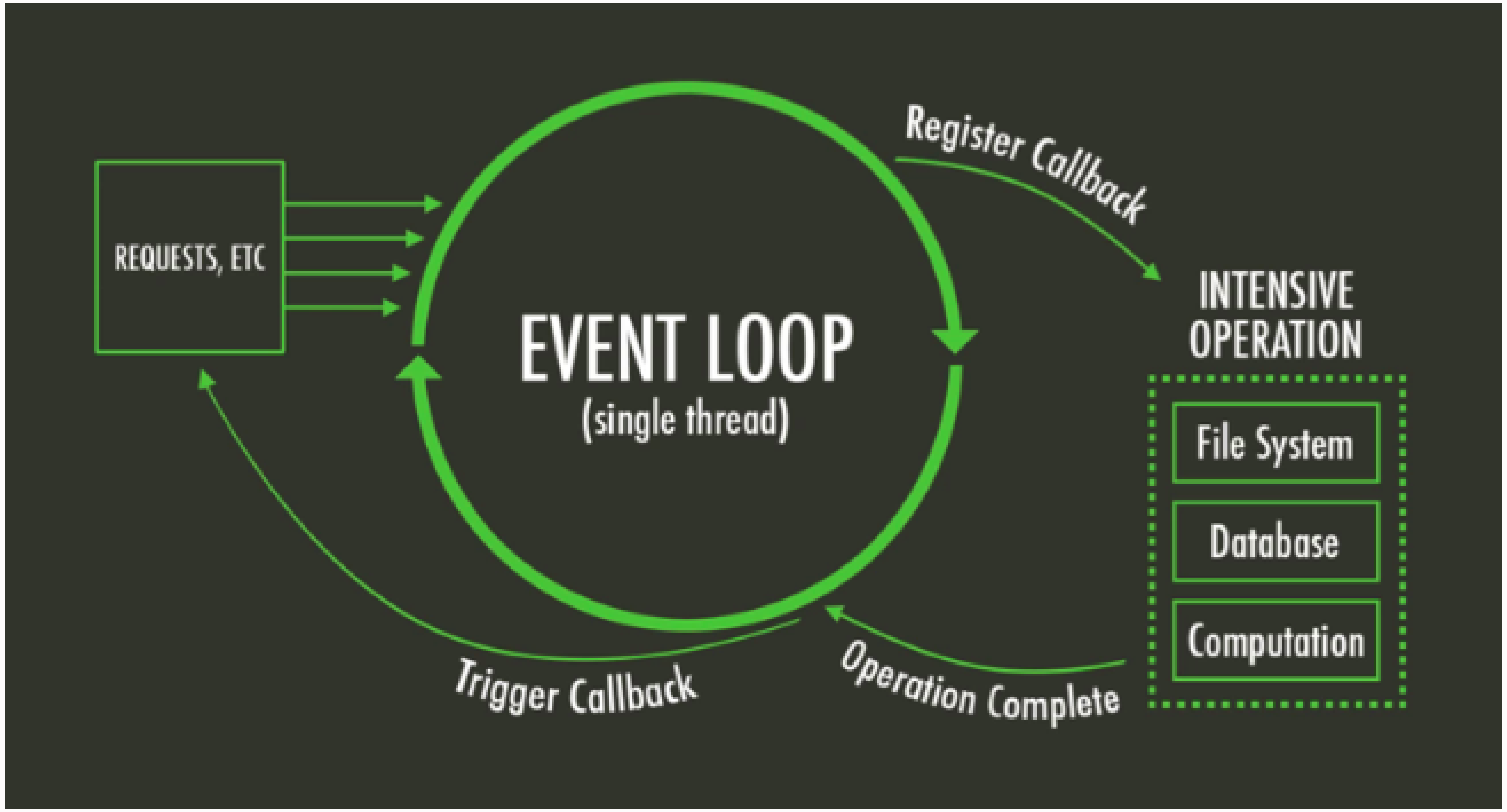
KETI ELIZBARASHVILI

Agenda

- Introduction to Node.js Runtime
- Brief overview and significance
- The Event Loop Explained
- Core concept and functionality of the event loop
- Phases of the Event Loop
- Detailed look into various phases
- Event Loop: Node.js vs Traditional Models
- Comparison with multi-threaded environments
- Blocking vs Non-blocking I/O
- Understanding the differences
- Non-blocking I/O in Node.js
- Demonstration and examples
- The Asynchronous Nature of Node.js
- Detailed explanation with examples
- Event-Driven Architecture Basics
- Introduction to event handling in Node.js
- Building an Event-Driven Application
- Step-by-step guide with example
- Understanding and Handling Back-Pressure
- Concept and practical scenarios
- Strategies for Managing Back-Pressure
- Practical approaches with examples
- Common Misconceptions
- Clarifying misunderstandings about Node.js
- Pitfalls: Blocking the Event Loop
- - Explanation and avoidance strategies
- Best Practices in Node.js Runtime Management
- Tips for efficient coding
- Further Resources
- Books, websites, tutorials for deep exploration
- Q&A Session
- Interactive discussion with the audience
- Closing Remarks
- Key takeaways and conclusion

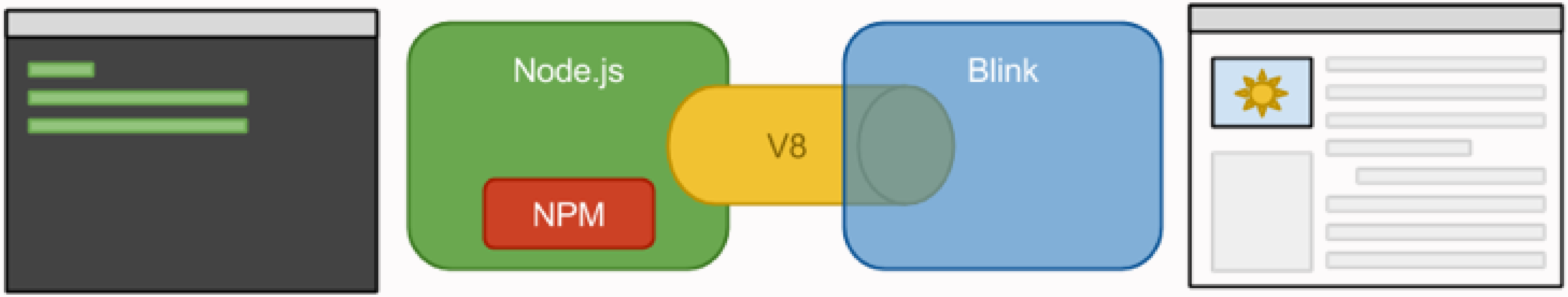
Introduction to Node.js Runtime

- What is Node.js?
- Why Node.js?
- The Node.js Runtime
- Key Features
- Use Cases



Why Understanding the Runtime Matters

Central to Performance and Efficiency
Improves Debugging Skills
Enables Scalable Application Design
Fosters Best Practices
Enables Scalable Application Design
Fosters Best Practices
Enhances Learning and Adaptability

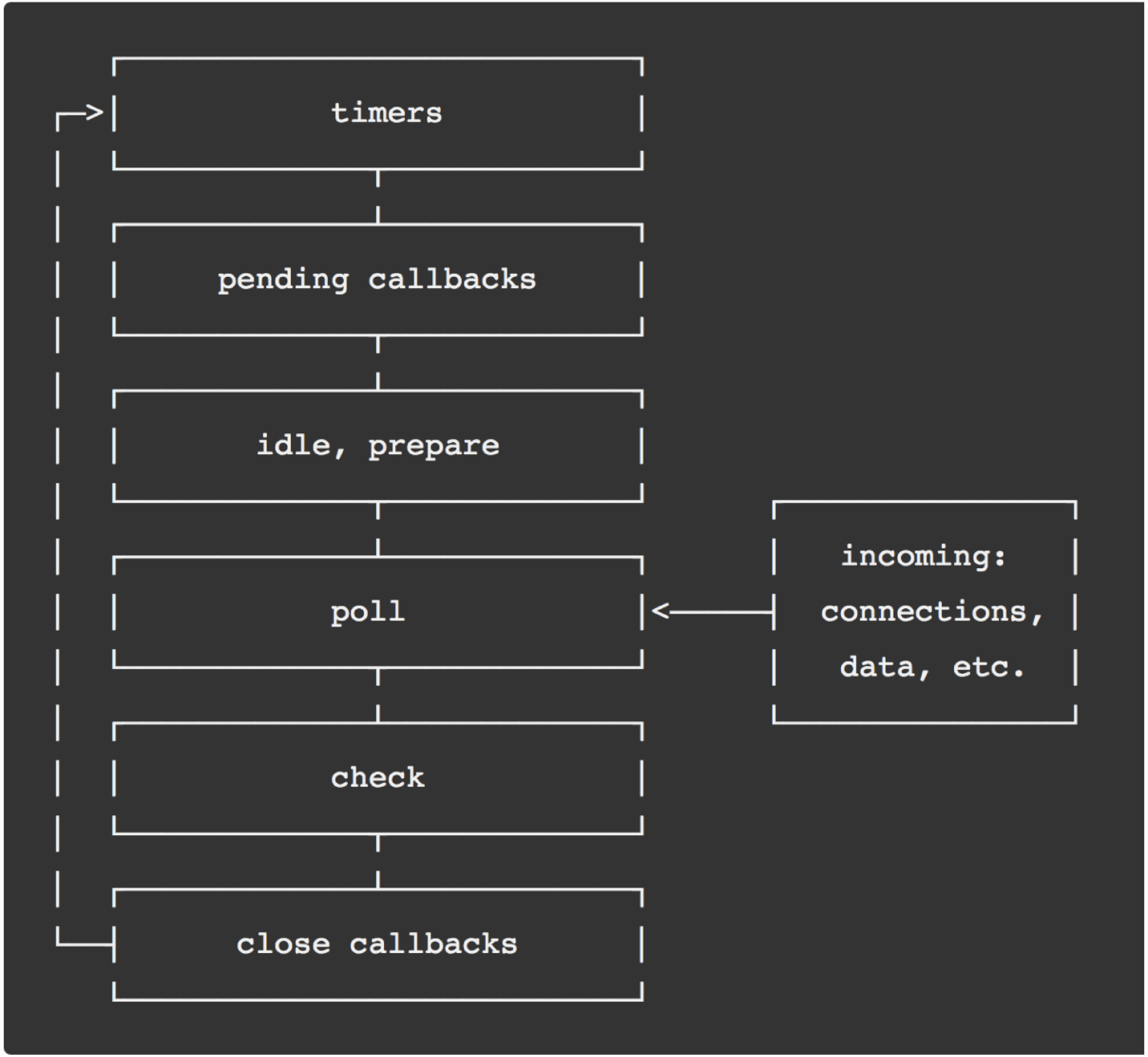


Access to computer
resources in Node.js

Visual rendering of app
in Blink Browser Component

The Event Loop Explained: Basic concept of the event loop.

1. Core of Node.js Architecture
    - The Event Loop is a fundamental part of the Node.js architecture, enabling its non-blocking, asynchronous behavior.
2. Single-Threaded Yet Powerful
    - Despite being single-threaded, the Event Loop effectively handles multiple concurrent operations, making Node.js efficient for I/O-heavy tasks.
3. How it Works
    - The Event Loop continuously checks for and executes tasks (callbacks) from an event queue.
    - Operations like I/O, timers, or network requests are executed outside the Event Loop and queued upon completion.
4. Non-Blocking Operations
    - I/O operations do not block the Event Loop. While waiting for a response, the loop can continue executing other tasks, significantly improving performance and scalability.
5. Event-Driven Programming
    - Node.js leverages event-driven programming, where the flow of the application is determined by events such as user input, file read completion, or a timer.
6. Why It's Important
    - Understanding the Event Loop is crucial for writing efficient Node.js applications. It helps in avoiding performance issues like blocking the loop and understanding asynchronous programming patterns.


timers
pending callbacks
idle, prepare
poll
check
close callbacks
incoming:
connections,
data, etc.

Phases of the Event Loop

Timers Phase

Handles callbacks scheduled by setTimeout() and setInterval().
Executes callbacks with the oldest timers first.

I/O Callbacks Phase

Processes callbacks from most system operations like network, file, and database I/O.
This phase excludes close callbacks, timers, and setImmediate().

Poll Phase

Retrieves new I/O events; execute I/O-related callbacks (almost all with the exception of close callbacks, timers, and setImmediate()).
Node.js will block here if appropriate, waiting for new events.

Check Phase

SetImmediate() callbacks are invoked here.
This phase allows execution of callbacks immediately after the poll phase has completed.

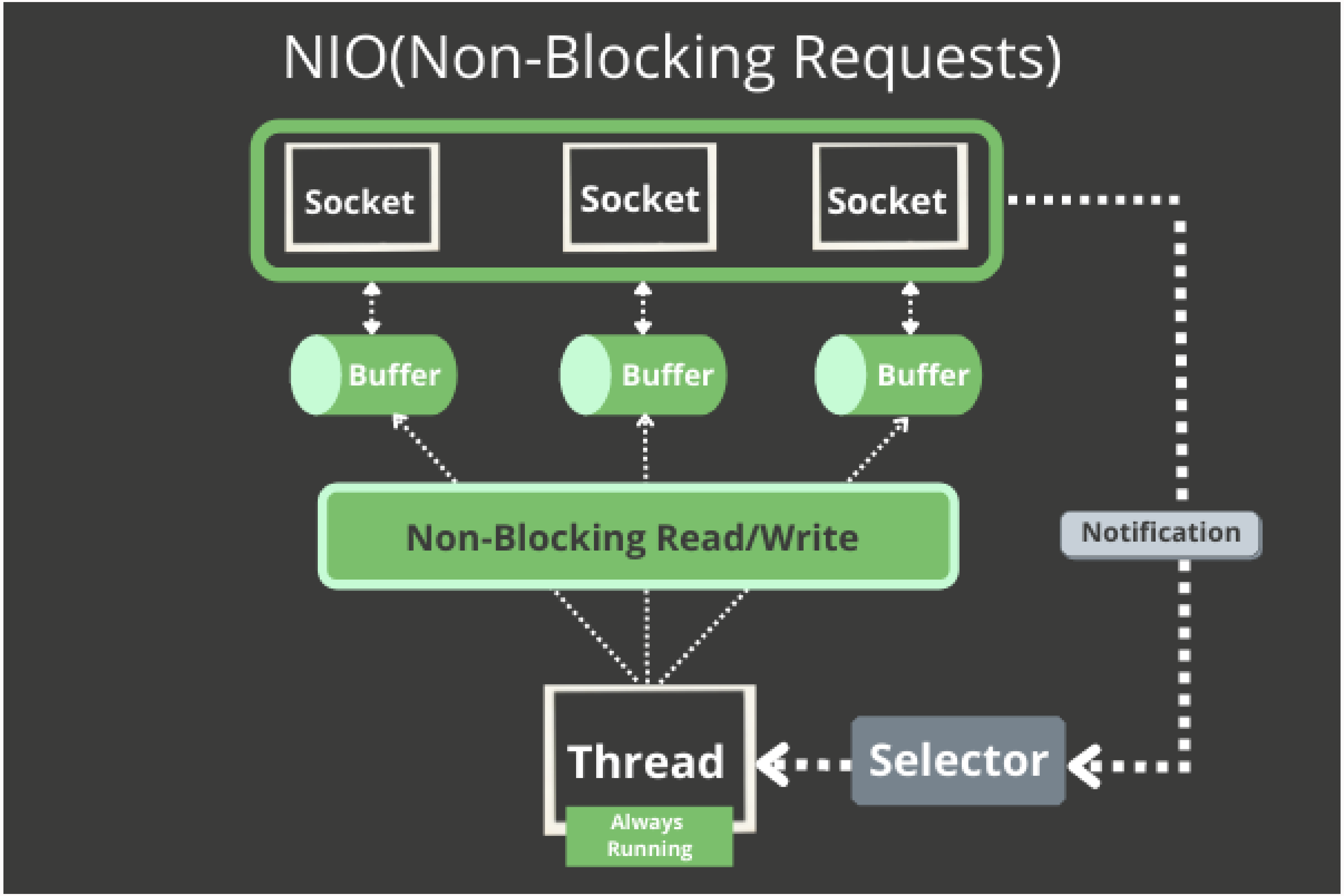
Close Callbacks Phase

Executes callbacks for some shutdown actions like socket.on('close', ...).
Important for cleanup and closing connections.

Blocking vs Non-blocking I/O: Conceptual Differences

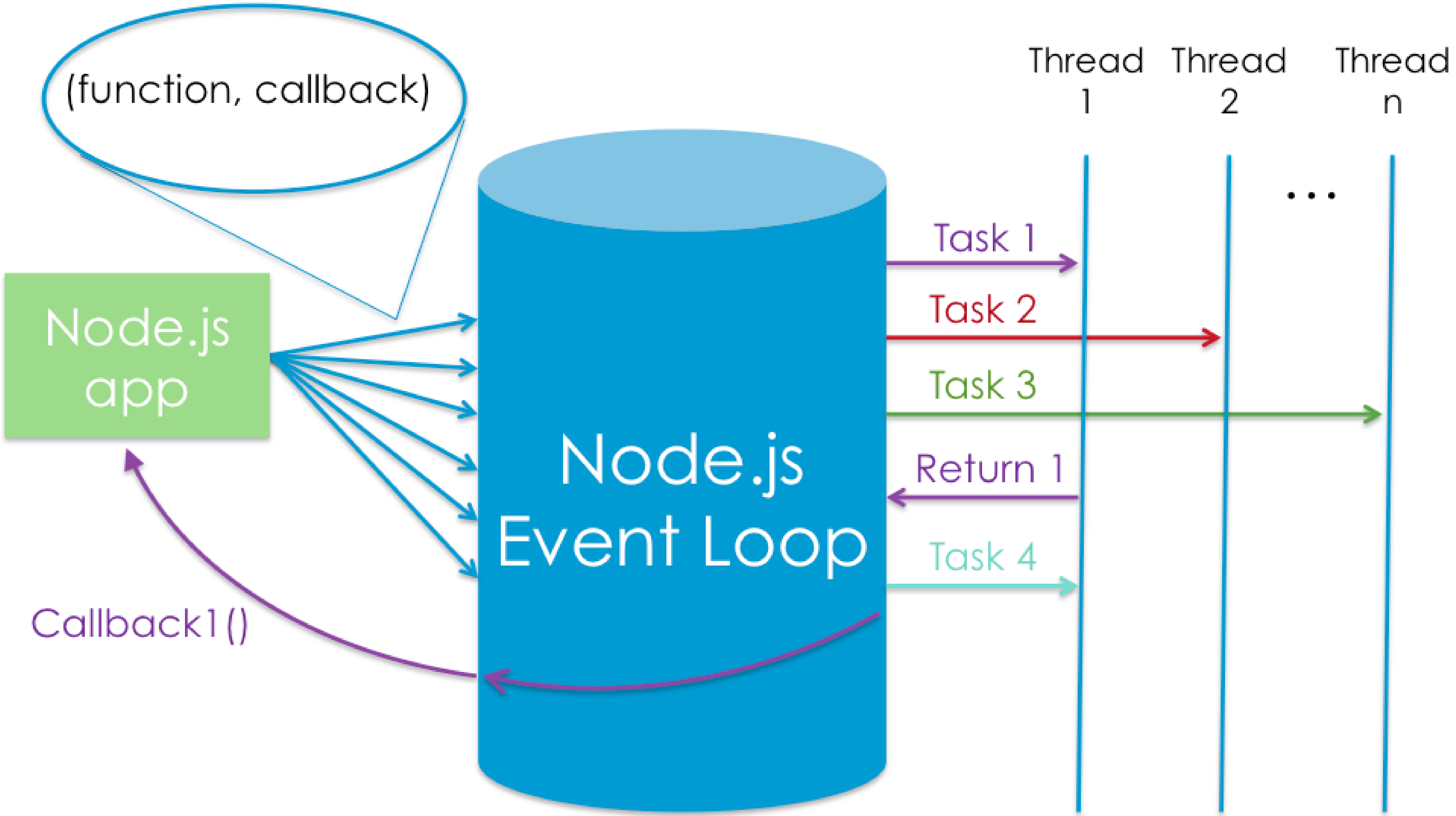
Blocking I/O
Non-blocking I/O

- Throughput and Responsiveness: Non-blocking I/O often results in higher throughput and better responsiveness in applications.
- Programming Complexity: Asynchronous programming (non-blocking) can add complexity due to callbacks, promises, and async/await patterns.
- Use Case Suitability: The choice between blocking and non-blocking I/O depends on the application's nature and performance requirements.


NIO(Non-Blocking Requests)
Socket
Socket
Socket
Buffer
Buffer
Buffer
Non-Blocking Read/Write
Notification
Thread
Always Running
Selector

1 Node apps pass async tasks to the event loop, along with a callback

2 The event loop efficiently manages a thread pool and executes tasks efficiently…



3 …and executes each callback as tasks complete

What is Event-Driven Architecture?

Event-driven architecture is a design paradigm where the flow of the program is determined by events.
In this context, an event can be anything significant that happens within the system, like a user action, system-generated event, or a message from another program.

Events in Node.js

Node.js is inherently event-driven. Most of its core API is built around an asynchronous, event-driven architecture.
Events are used extensively, from handling HTTP requests in a web server to dealing with file operations.

Event Emitters

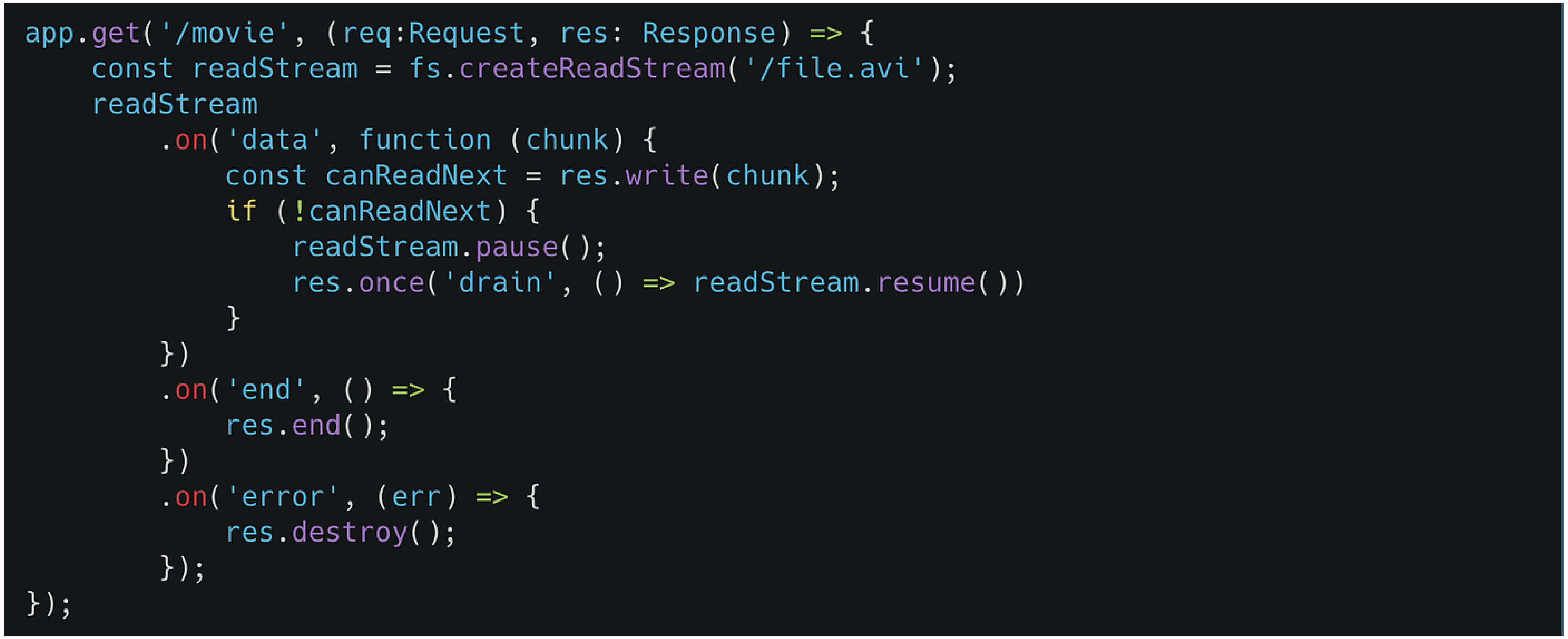
The EventEmitter class, part of the events module in Node.js, is key to working with events.
It provides the ability to create and handle custom events.

Understanding Back-Pressure

Back-pressure occurs when a data stream generates data at a faster rate than it can be processed, leading to potential bottlenecks in the system.

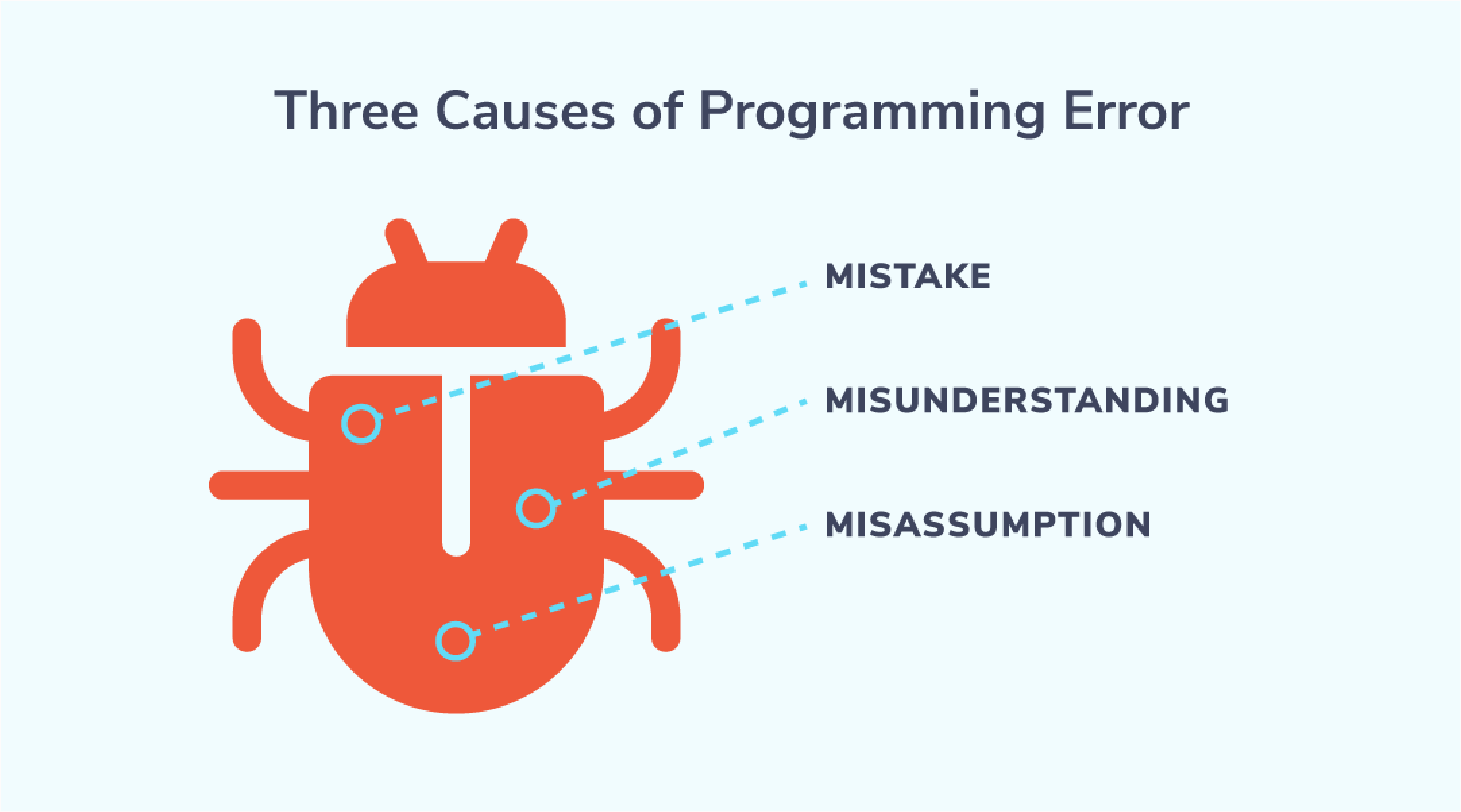
In Node.js, this is often encountered in scenarios involving streams, like reading from a fast data source and writing to a slower sink.

Using Stream's .pause() and .resume()



```
app.get('/movie', (req:Request, res: Response) => {
    const readStream = fs.createReadStream('/file.avi');
    readStream
        .on('data', function (chunk) {
            const canReadNext = res.write(chunk);
            if (!canReadNext) {
                readStream.pause();
                res.once('drain', () => readStream.resume())
            }
        })
        .on('end', () => {
            res.end();
        })
        .on('error', (err) => {
            res.destroy();
        });
});
```

Common Misconceptions: Clarifying asynchronous execution and other misunderstandings.



Questions & Answers