Node.js

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Explain the Node.js event loop architecture and how it handles asynchronous operations across different phases.

Event Loop Phases

The Node.js event loop operates in distinct phases, each handling specific types of callbacks:

- **Timers:** Executes callbacks scheduled by setTimeout() and setInterval()
- **Pending callbacks:** Executes I/O callbacks deferred from the previous cycle
- **Idle, prepare:** Internal use only
- **Poll:** Retrieves new I/O events and executes I/O-related callbacks
- **Check:** Executes setImmediate() callbacks
- **Close callbacks:** Handles close events like socket.on('close')

Between each phase, Node.js checks for **process.nextTick()** and **microtasks** (Promises), which have higher priority than regular callbacks.

```
console.log('1');
setTimeout(() => console.log('2'), 0);
setImmediate(() => console.log('3'));
process.nextTick(() => console.log('4'));
Promise.resolve().then(() => console.log('5'));
console.log('6');
// Output: 1, 6, 4, 5, 2, 3
```

Understanding this execution order is critical for debugging race conditions and optimizing asynchronous workflows in production systems.

2. What are the differences between process.nextTick(), setImmediate(), and setTimeout() in terms of execution timing?

Execution Priority Comparison

process.nextTick() has the highest priority and executes immediately after the current operation completes, before the event loop continues. It can lead to I/O starvation if used recursively.

setImmediate() executes in the check phase of the event loop, after I/O events have been processed. It's designed for deferring execution until after I/O operations.

setTimeout(fn, 0) executes in the timers phase, which comes before the check phase but after the poll phase completes.

```
const fs = require('fs');

fs.readFile('file.txt', () => {
  setTimeout(() => console.log('timeout'), 0);
  setImmediate(() => console.log('immediate'));
});
// Output: immediate, timeout
```

Inside I/O callbacks, **setImmediate()** always executes before **setTimeout()** because the check phase comes after the poll phase. Use **setImmediate()** for better predictability in asynchronous flows.

3. How does Node.js handle multi-threading, and what are Worker Threads? When should you use them?

Node.js Threading Model

Node.js runs JavaScript in a **single-threaded event loop**, but uses a thread pool (libuv) for blocking operations like file I/O, DNS lookups, and crypto operations. The default pool size is 4 threads, configurable via UV_THREADPOOL_SIZE.

Worker Threads (introduced in Node.js 10.5.0) allow running JavaScript in parallel threads, each with its own V8 instance and event loop. They communicate via message passing.

```
const { Worker } = require('worker_threads');

const worker = new Worker('./worker.js', {
  workerData: { num: 5 }
});

worker.on('message', (result) => {
  console.log('Result:', result);
});
```

When to Use Worker Threads:

- CPU-intensive tasks (image processing, video encoding, complex calculations)
- Operations that would block the event loop for extended periods
- Parallel processing of independent data sets

Avoid for I/O-bound operations—the event loop already handles those efficiently.

4. Explain the differences between Streams and Buffers. How do you implement a custom Transform stream?

Buffers vs Streams

Buffers are fixed-size chunks of memory allocated outside the V8 heap for handling binary data. They load entire data into memory.

Streams process data in chunks, enabling memory-efficient handling of large datasets. Four types exist: Readable, Writable, Duplex, and Transform.

Custom Transform Stream Implementation:

```
const { Transform } = require('stream');

class UpperCaseTransform extends Transform {
  _transform(chunk, encoding, callback) {
    const upperChunk = chunk.toString().toUpperCase();
    this.push(upperChunk);
    callback();
  }
}

process.stdin.pipe(new UpperCaseTransform()).pipe(process.stdout);
```

Transform streams are ideal for data manipulation pipelines—compression, encryption, parsing—where you need to modify data as it flows through. They implement both Readable and Writable interfaces, making them perfect for middleware-style processing.

5. What is backpressure in Node.js streams, and how do you handle it properly?

Understanding Backpressure

Backpressure occurs when data is written to a stream faster than it can be consumed, causing memory buildup. This happens when a writable stream's internal buffer fills up.

The **write()** method returns **false** when the buffer is full, signaling you should pause writing. The **'drain'** event fires when it's safe to resume.

```
const fs = require('fs');
const readable = fs.createReadStream('large.txt');
const writable = fs.createWriteStream('output.txt');

readable.on('data', (chunk) => {
  const canContinue = writable.write(chunk);
  if (!canContinue) {
    readable.pause();
    writable.once('drain', () => readable.resume());
  }
});
```

Better approach: Use **pipe()** which handles backpressure automatically:

```
readable.pipe(writable);
```

Proper backpressure handling prevents memory leaks and ensures stable performance when processing large files or network streams.

6. How does the Node.js module caching mechanism work? What are the implications for singleton patterns?

Module Caching Behavior

Node.js caches modules in **require.cache** after first load. Subsequent require() calls return the same instance, making modules natural singletons.

```
// counter.js
let count = 0;
module.exports = {
  increment: () => ++count,
  getCount: () => count
};

// app.js
const counter1 = require('./counter');
const counter2 = require('./counter');
counter1.increment();
console.log(counter2.getCount()); // 1
```

Key Implications:

- Modules are cached by **resolved absolute path**—same module via different paths creates separate instances
- Cache is per-process—worker threads and cluster workers have separate caches
- Circular dependencies are handled via partial exports
- You can clear cache: **delete require.cache[require.resolve('./module')]**

This behavior is essential for implementing database connection pools, configuration managers, and other singleton patterns efficiently.

7. Explain the difference between clustering and worker threads for scaling Node.js applications.

Clustering vs Worker Threads

Cluster Module: Creates multiple Node.js processes (workers) that share the same server port. Each worker has its own event loop, memory, and V8 instance. The master process distributes incoming connections using round-robin (on most platforms).

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) cluster.fork();
} else {
  http.createServer((req, res) => {
    res.end('Hello');
  }).listen(8000);
}
```

Worker Threads: Run JavaScript in parallel within the same process, sharing memory via SharedArrayBuffer. Better for CPU-intensive tasks without I/O.

When to Use:

- **Clustering:** Scale HTTP servers, utilize all CPU cores, isolate crashes
- **Worker Threads:** CPU-heavy computations, parallel data processing, avoid blocking main thread

Often used together: clustering for horizontal scaling, worker threads for computational offloading.

8. What are the performance implications of synchronous vs asynchronous file operations? When would you use each?

Sync vs Async File Operations

Asynchronous operations (fs.readFile, fs.writeFile) don't block the event loop, allowing Node.js to handle other requests concurrently. They use the libuv thread pool.

Synchronous operations (fs.readFileSync, fs.writeFileSync) block the event loop completely until the operation completes, preventing all other processing.

```
// Blocks entire server during read
const data = fs.readFileSync('large.txt', 'utf8');

// Non-blocking, server remains responsive
fs.readFile('large.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

When to Use Synchronous:

- Application startup/initialization before accepting requests
- CLI tools and build scripts
- Configuration loading at boot time
- When the process has no other work to do

Never use sync operations in request handlers—they destroy throughput by blocking all concurrent requests. A single 100ms sync read can reduce server capacity from thousands to tens of requests per second.

9. How does Node.js handle memory management and garbage collection? What tools help diagnose memory leaks?

Memory Management in Node.js

Node.js uses V8's garbage collector with two main spaces:

- **New Space (Young Generation):** Short-lived objects, ~1-8MB, uses Scavenge algorithm (fast, frequent)
- **Old Space (Old Generation):** Long-lived objects, uses Mark-Sweep-Compact (slower, less frequent)

Common memory leak causes: global variables, closures retaining references, event listeners not removed, large caches without limits.

Diagnostic Tools:

```
// Heap snapshot
const v8 = require('v8');
const fs = require('fs');
const snapshot = v8.writeHeapSnapshot();
console.log('Snapshot:', snapshot);

// Memory usage
console.log(process.memoryUsage());
```

Tools for leak detection:

- **Chrome DevTools:** Connect with --inspect flag, take heap snapshots
- **clinic.js:** Comprehensive performance profiling
- **memwatch-next:** Monitors heap and detects leaks
- **heapdump:** Generate and compare snapshots

Monitor **heapUsed** over time—steady growth indicates leaks.

10. Explain how to implement graceful shutdown in a Node.js application with active connections and background jobs.

Graceful Shutdown Implementation

Graceful shutdown ensures in-flight requests complete, database connections close properly, and resources are released before process termination.

```
const server = http.createServer(app);
let isShuttingDown = false;

function gracefulShutdown(signal) {
  console.log(`${signal} received`);
  isShuttingDown = true;

  server.close(() => {
    console.log('HTTP server closed');
    dbConnection.close();
    process.exit(0);
  });

  setTimeout(() => process.exit(1), 30000);
}

process.on('SIGTERM', gracefulShutdown);
process.on('SIGINT', gracefulShutdown);
```

Best Practices:

- Stop accepting new requests immediately (return 503)
- Wait for active requests to complete with timeout
- Close database connections and external services
- Flush logs and metrics
- Cancel background jobs or wait for completion
- Use health check endpoints that return unhealthy during shutdown

For worker queues, implement drain mechanisms. In Kubernetes, configure **terminationGracePeriodSeconds** appropriately (typically 30-60s).

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement a Stack data structure in Node.js with push, pop, peek, and isEmpty methods. What is the time complexity of each operation?

Stack Implementation

A stack follows the **Last-In-First-Out (LIFO)** principle. Here's a clean implementation:

```
class Stack {
  constructor() { this.items = []; }
  push(element) { this.items.push(element); }
  pop() { return this.items.pop(); }
  peek() { return this.items[this.items.length - 1]; }
  isEmpty() { return this.items.length === 0; }
  size() { return this.items.length; }
}
```

Time Complexity:

- push(): O(1) - Constant time insertion at the end
- pop(): O(1) - Constant time removal from the end
- peek(): O(1) - Direct array access
- isEmpty(): O(1) - Length check is constant time

2. How would you implement an LRU (Least Recently Used) Cache in Node.js? Explain the approach and time complexity.

LRU Cache Implementation

An **LRU Cache** requires O(1) access and update operations. Use a **Map** (maintains insertion order) combined with key repositioning:

```
class LRUCache {
  constructor(capacity) {
    this.capacity = capacity;
    this.cache = new Map();
  }
  get(key) {
    if (!this.cache.has(key)) return -1;
    const val = this.cache.get(key);
    this.cache.delete(key);
    this.cache.set(key, val);
    return val;
  }
  put(key, value) {
    if (this.cache.has(key)) this.cache.delete(key);
    else if (this.cache.size >= this.capacity) {
      this.cache.delete(this.cache.keys().next().value);
    }
    this.cache.set(key, value);
  }
}
```

Time Complexity: O(1) for both get() and put() operations. **Space Complexity:** O(capacity)

3. Write a function to find all pairs in an array that sum to a target value. What's the optimal time complexity?

Pair Sum Problem

Use a **Set** to achieve O(n) time complexity instead of the naive O(n²) nested loop approach:

```
function findPairs(arr, target) {
  const seen = new Set();
  const pairs = [];
  for (const num of arr) {
    const complement = target - num;
    if (seen.has(complement)) {
      pairs.push([complement, num]);
    }
    seen.add(num);
  }
  return pairs;
}
```

Time Complexity: O(n) - Single pass through array

Space Complexity: O(n) - Set storage for seen elements

This approach uses the **hash table technique** to trade space for time efficiency.

4. Implement a Queue using two Stacks in Node.js. Explain the amortized time complexity.

Queue Using Two Stacks

This classic problem demonstrates **amortized analysis**:

```
class QueueWithStacks {
  constructor() {
    this.stack1 = [];
    this.stack2 = [];
  }
  enqueue(item) { this.stack1.push(item); }
  dequeue() {
    if (!this.stack2.length) {
      while (this.stack1.length) {
        this.stack2.push(this.stack1.pop());
      }
    }
    return this.stack2.pop();
  }
}
```

Time Complexity:

- enqueue(): O(1) - Direct push to stack1
- dequeue(): O(1) amortized - Each element is moved exactly once from stack1 to stack2

Space Complexity: O(n) for storing n elements

5. How do you implement a Trie (Prefix Tree) for autocomplete functionality? What are the time complexities?

Trie Implementation

A **Trie** is ideal for prefix-based searches like autocomplete:

```
class TrieNode {
  constructor() {
    this.children = {};
    this.isEndOfWord = false;
  }
}
class Trie {
  constructor() { this.root = new TrieNode(); }
  insert(word) {
    let node = this.root;
    for (const char of word) {
      if (!node.children[char]) node.children[char] = new TrieNode();
      node = node.children[char];
    }
    node.isEndOfWord = true;
  }
  search(word) {
    let node = this.root;
    for (const char of word) {
      if (!node.children[char]) return false;
      node = node.children[char];
    }
    return node.isEndOfWord;
  }
}
```

Time Complexity:

- Insert: O(m) where m is word length
- Search: O(m)
- Prefix search: O(p) where p is prefix length

Space Complexity: O(ALPHABET_SIZE * N * M) in worst case

6. Implement a sliding window maximum algorithm. Given an array and window size k, find the maximum in each window.

Sliding Window Maximum

Use a **deque (double-ended queue)** to maintain indices of useful elements in decreasing order:

```
function maxSlidingWindow(nums, k) {
  const result = [];
  const deque = [];
  for (let i = 0; i < nums.length; i++) {
    if (deque.length && deque[0] <= i - k) deque.shift();
    while (deque.length && nums[deque[deque.length-1]] < nums[i]) {
      deque.pop();
    }
    deque.push(i);
    if (i >= k - 1) result.push(nums[deque[0]]);
  }
  return result;
}
```

Time Complexity: O(n) - Each element is added and removed at most once

Space Complexity: O(k) - Deque stores at most k elements

This is more efficient than the naive O(n*k) approach.

7. How would you detect a cycle in a linked list? Implement Floyd's Cycle Detection Algorithm.

Floyd's Cycle Detection (Tortoise and Hare)

This algorithm uses **two pointers** moving at different speeds:

```
function hasCycle(head) {
  if (!head || !head.next) return false;
  let slow = head;
  let fast = head;
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow === fast) return true;
  }
  return false;
}
```

Time Complexity: O(n) - Fast pointer traverses at most 2n nodes

Space Complexity: O(1) - Only two pointers used

Why it works: If there's a cycle, the fast pointer will eventually catch up to the slow pointer inside the cycle. The distance closes by 1 each iteration.

8. Implement a function to find the kth largest element in an unsorted array. What's the most efficient approach?

Kth Largest Element

Use **QuickSelect algorithm** (based on QuickSort partitioning) for O(n) average time:

```
function findKthLargest(nums, k) {
  const targetIdx = nums.length - k;
  function quickSelect(left, right) {
    const pivot = nums[right];
    let p = left;
    for (let i = left; i < right; i++) {
      if (nums[i] <= pivot) [nums[p], nums[i]] = [nums[i], nums[p++]];
    }
    [nums[p], nums[right]] = [nums[right], nums[p]];
    if (p === targetIdx) return nums[p];
```

```
    return p < targetIdx ? quickSelect(p+1, right) : quickSelect(left, p-1);
  }
  return quickSelect(0, nums.length - 1);
}
```

Time Complexity:

- Average: O(n) - Partition reduces problem size by half
- Worst: O(n²) - Poor pivot selection

Alternative: Min-heap of size k gives O(n log k) with guaranteed performance.

9. How do you implement a HashMap from scratch in Node.js? Handle collisions using chaining.

HashMap with Chaining

Implement a **hash table** using an array of linked lists for collision resolution:

```
class HashMap {
  constructor(size = 53) {
    this.buckets = new Array(size).fill(null).map(() => []);
  }
  hash(key) {
    let total = 0;
    for (let char of key) total = (total + char.charCodeAt(0) * 31) % this.buckets.length;
    return total;
  }
  set(key, value) {
    const idx = this.hash(key);
    const bucket = this.buckets[idx];
    const existing = bucket.find(item => item[0] === key);
    if (existing) existing[1] = value;
    else bucket.push([key, value]);
  }
  get(key) {
    const idx = this.hash(key);
    const pair = this.buckets[idx].find(item => item[0] === key);
    return pair ? pair[1] : undefined;
  }
}
```

Time Complexity:

- Average: O(1) for get/set with good hash function
- Worst: O(n) if all keys hash to same bucket

10. Implement a binary search tree (BST) with insert, search, and inorder traversal. What are the time complexities?

Binary Search Tree

A **BST** maintains the property: left < node < right for efficient searching:

```
class TreeNode {
  constructor(val) { this.val = val; this.left = this.right = null; }
}
class BST {
  constructor() { this.root = null; }
  insert(val) {
    const node = new TreeNode(val);
    if (!this.root) { this.root = node; return; }
    let current = this.root;
    while (true) {
      if (val < current.val) {
        if (!current.left) { current.left = node; break; }
        current = current.left;
      } else {
        if (!current.right) { current.right = node; break; }
        current = current.right;
      }
    }
  }
  inorder(node = this.root, result = []) {
    if (node) {
      this.inorder(node.left, result);
      result.push(node.val);
      this.inorder(node.right, result);
    }
    return result;
  }
}
```

Time Complexity:

- Insert/Search: O(log n) average, O(n) worst (skewed tree)
- Inorder: O(n) - visits all nodes

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly. What are the key components and how would you handle high traffic?

Core Requirements

- **Functional:** Generate short URLs, redirect to original URLs, optional custom aliases
- **Non-functional:** High availability, low latency, scalability to billions of URLs

Architecture Components

- **API Gateway:** Rate limiting, authentication, load balancing
- **Application Servers:** Stateless Node.js services for URL generation and retrieval
- **Database:** NoSQL (e.g., Cassandra, DynamoDB) for horizontal scaling
- **Cache Layer:** Redis/Memcached for frequently accessed URLs (80/20 rule)
- **CDN:** Geographic distribution for faster redirects

URL Generation Strategy

```
// Base62 encoding for short URLs
function encodeBase62(num) {
  const chars = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
  let result = '';
  while (num > 0) {
    result = chars[num % 62] + result;
    num = Math.floor(num / 62);
  }
  return result;
}
```

Scaling Considerations

- **Sharding:** Partition by hash of short URL or range-based on counter
- **Replication:** Multi-region read replicas for high availability
- **Write optimization:** Use distributed counter service (Zookeeper/etcd) or pre-generate key ranges
- **Analytics:** Async message queue (Kafka) for click tracking without blocking redirects

CAP Theorem Trade-off

Choose AP (Availability + Partition Tolerance) over strong consistency. Eventual consistency is acceptable for analytics, but URL mappings should use strong consistency to avoid collisions.

2. How would you design a real-time chat application using Node.js? Discuss the architecture for supporting millions of concurrent users.

Architecture Overview

- **WebSocket Gateway:** Socket.io or native WebSocket servers for bidirectional communication
- **Message Broker:** Redis Pub/Sub or Kafka for message distribution across servers
- **Presence Service:** Track online/offline status using Redis with TTL
- **Message Storage:** MongoDB/Cassandra for chat history
- **Media Storage:** S3/CDN for images, videos, files

Connection Management

```
// Socket.io with Redis adapter
const io = require('socket.io')(server);
const redisAdapter = require('socket.io-redis');
io.adapter(redisAdapter({ host: 'localhost', port: 6379 }));

io.on('connection', (socket) => {
  socket.on('join', (roomId) => socket.join(roomId));
  socket.on('message', (data) => io.to(data.room).emit('message', data));
});
```

Scaling Strategy

- **Horizontal Scaling:** Multiple Node.js instances behind load balancer with sticky sessions or Redis adapter
- **Sharding:** Partition users by user_id or room_id across different clusters
- **Message Queue:** Decouple message processing from delivery using Kafka for durability
- **Read Replicas:** Separate databases for reads (message history) and writes (new messages)

Optimizations

- **Message Batching:** Aggregate multiple messages before broadcasting
- **Presence Heartbeat:** Client sends periodic pings, server updates Redis with expiry
- **Offline Messages:** Queue messages in database for offline users, deliver on reconnection
- **Rate Limiting:** Per-user message limits to prevent spam

3. Design a news feed system like Twitter or Facebook. How would you handle feed generation for millions of users?

Feed Generation Approaches

- **Fan-out on Write (Push):** Pre-compute feed when post is created, write to all followers' caches
- **Fan-out on Read (Pull):** Compute feed on-demand by querying followed users' posts
- **Hybrid:** Push for regular users, pull for celebrities with millions of followers

Architecture Components

- **Post Service:** Create and store posts in database (PostgreSQL/MySQL)
- **Feed Service:** Generate and cache personalized feeds
- **Graph Database:** Neo4j or adjacency list in Redis for follower relationships
- **Cache Layer:** Redis sorted sets for timeline storage (score = timestamp)
- **CDN:** Serve media content (images, videos)

Feed Generation Example

```
// Fan-out on write using Redis
async function fanoutPost(userId, postId, timestamp) {
  const followers = await getFollowers(userId);
```

```
  const pipeline = redis.pipeline();
  followers.forEach(followerId => {
    pipeline.zadd(`feed:${followerId}`, timestamp, postId);
    pipeline.zremrangebyrank(`feed:${followerId}`, 0, -1001);
  });
  await pipeline.exec();
}
```

Optimizations

- **Lazy Loading:** Load top 20 posts initially, infinite scroll for more
- **Ranking Algorithm:** Consider engagement (likes, comments), recency, and user preferences
- **Async Processing:** Use message queues (RabbitMQ/Kafka) for fan-out operations
- **Materialized Views:** Pre-aggregate trending posts, popular hashtags

Consistency Considerations

Eventual consistency is acceptable. Users can tolerate slight delays in seeing new posts. Use **write-through cache** to ensure newly created posts appear immediately in author's feed.

4. How would you design a distributed rate limiter service in Node.js? Discuss algorithms and implementation strategies.

Rate Limiting Algorithms

- **Token Bucket:** Tokens added at fixed rate, request consumes token
- **Leaky Bucket:** Requests processed at constant rate, excess queued or dropped
- **Fixed Window:** Count requests in fixed time windows (simple but has boundary issues)
- **Sliding Window Log:** Track timestamp of each request, count in sliding window
- **Sliding Window Counter:** Weighted count from current and previous window

Token Bucket Implementation with Redis

```
// Token bucket using Redis
async function allowRequest(userId, maxTokens, refillRate) {
  const key = `ratelimit:${userId}`;
  const now = Date.now();
  const result = await redis.eval(`
    local tokens = tonumber(redis.call('hget', KEYS[1], 'tokens')) or ARGV[1]
    local lastRefill = tonumber(redis.call('hget', KEYS[1], 'last')) or ARGV[2]
    local elapsed = ARGV[2] - lastRefill
    tokens = math.min(ARGV[1], tokens + elapsed * ARGV[3])
    if tokens >= 1 then tokens = tokens - 1 redis.call('hset', KEYS[1], 'tokens', tokens, 'last', ARGV[2]) return 1 else return 0 end
  `, 1, key, maxTokens, now, refillRate);
  return result === 1;
}
```

Distributed Considerations

- **Centralized Redis:** Single source of truth, but single point of failure
- **Redis Cluster:** Partition keys across nodes for horizontal scaling
- **Local + Global Limits:** In-memory counters per instance + Redis for global limit
- **Approximate Counting:** Use probabilistic data structures (Count-Min Sketch) for high scale

Architecture

- **API Gateway:** Enforce rate limits before routing to services
- **Middleware:** Express middleware for application-level limiting
- **Tiered Limits:** Different limits per user tier (free, premium, enterprise)
- **Circuit Breaker:** Fail open if Redis is down to maintain availability

5. Design a video streaming platform like YouTube. How would you handle video upload, processing, and delivery?

System Components

- **Upload Service:** Multipart upload to S3/Object Storage with resumable uploads
- **Transcoding Service:** FFmpeg workers to convert videos to multiple formats/resolutions
- **CDN:** CloudFront/Akamai for global content delivery
- **Metadata Service:** Store video info, thumbnails, captions in database
- **Recommendation Engine:** ML-based service for personalized suggestions

Upload Flow

```
// Chunked upload with progress tracking
const uploadVideo = async (file, onProgress) => {
  const chunkSize = 5 * 1024 * 1024; // 5MB chunks
  const uploadId = await initiateMultipartUpload();
  for (let i = 0; i < file.size; i += chunkSize) {
    const chunk = file.slice(i, i + chunkSize);
    await uploadPart(uploadId, chunk, i / chunkSize);
    onProgress((i / file.size) * 100);
  }
  return completeUpload(uploadId);
};
```

Video Processing Pipeline

- **Step 1:** Upload complete triggers Lambda/worker to start transcoding
- **Step 2:** Generate multiple resolutions (360p, 720p, 1080p, 4K) and formats (MP4, WebM)
- **Step 3:** Create HLS/DASH manifests for adaptive bitrate streaming
- **Step 4:** Extract thumbnails, generate preview clips
- **Step 5:** Run content moderation (ML models for NSFW detection)
- **Step 6:** Update metadata database and invalidate CDN cache

Streaming Strategy

- **Adaptive Bitrate:** HLS or MPEG-DASH for quality adjustment based on bandwidth
- **CDN Edge Caching:** Cache popular videos closer to users
- **Signed URLs:** Time-limited URLs for access control
- **Prefetching:** Preload next segment while current plays

Scaling Considerations

- **Distributed Transcoding:** Kubernetes cluster with auto-scaling based on queue depth
- **Storage Tiering:** Hot storage (recent videos) vs cold storage (old/rarely accessed)
- **Database Sharding:** Partition by video_id or user_id

6. How would you design a distributed caching system? Discuss cache invalidation strategies and consistency models.

Cache Architecture

- **Client-Side Cache:** Browser cache, service worker
- **CDN Cache:** Edge locations for static assets
- **Application Cache:** In-memory (Node.js Map, LRU cache)
- **Distributed Cache:** Redis Cluster, Memcached
- **Database Cache:** Query result cache, materialized views

Caching Patterns

- **Cache-Aside (Lazy Loading):** App checks cache, loads from DB on miss, writes to cache
- **Write-Through:** Write to cache and DB simultaneously
- **Write-Behind:** Write to cache immediately, async write to DB
- **Refresh-Ahead:** Proactively refresh cache before expiration

LRU Cache Implementation

```
class LRUCache {
  constructor(capacity) {
    this.capacity = capacity;
    this.cache = new Map();
  }
  get(key) {
    if (!this.cache.has(key)) return null;
    const val = this.cache.get(key);
    this.cache.delete(key);
    this.cache.set(key, val);
    return val;
  }
  put(key, val) { if (this.cache.has(key)) this.cache.delete(key); else if (this.cache.size >= this.capacity) this.cache.delete(this.cache.keys().next().value); this.cache.set(key, v
}
```

Cache Invalidation Strategies

- **TTL (Time-To-Live):** Expire cache after fixed duration
- **Event-Based:** Invalidate on data update using pub/sub
- **Version-Based:** Include version number in cache key
- **Tag-Based:** Group related cache entries, invalidate by tag

Consistency Models

- **Strong Consistency:** Synchronous cache invalidation (slower but accurate)
- **Eventual Consistency:** Async invalidation, temporary stale data acceptable
- **Read-Your-Writes:** Ensure user sees their own updates immediately

Distributed Cache Challenges

- **Cache Stampede:** Use mutex/semaphore to prevent multiple DB queries on cache miss
- **Hot Key Problem:** Replicate frequently accessed keys across nodes
- **Thundering Herd:** Stagger cache expiration times

7. Design a ride-sharing platform like Uber. Focus on real-time location tracking and driver-rider matching.

Core Components

- **Location Service:** Track real-time GPS coordinates of drivers and riders
- **Matching Service:** Algorithm to pair riders with nearby available drivers
- **Trip Service:** Manage trip lifecycle (request, accept, start, complete)
- **Pricing Service:** Dynamic pricing based on demand, distance, time
- **Notification Service:** Push notifications for status updates
- **Payment Service:** Process payments and payouts

Geospatial Indexing

```
// Using Redis Geospatial for driver location
async function updateDriverLocation(driverId, lat, lon) {
  await redis.geoadd('drivers:locations', lon, lat, driverId);
  await redis.expire(`drivers:locations:${driverId}`, 300);
}

async function findNearbyDrivers(lat, lon, radius) {
  return redis.georadius('drivers:locations', lon, lat, radius, 'km', 'WITHDIST', 'ASC', 'COUNT', 10);
}
```

Matching Algorithm

- **Step 1:** Find drivers within radius (e.g., 5km) using geospatial query
- **Step 2:** Filter by availability, vehicle type, rating
- **Step 3:** Calculate ETA using routing service (Google Maps API, OSRM)
- **Step 4:** Rank by distance, ETA, driver rating
- **Step 5:** Send request to top driver, timeout and cascade to next if no response

Real-Time Location Updates

- **WebSocket Connection:** Drivers send location every 3-5 seconds
- **Location Smoothing:** Interpolate between updates for smooth rider UI
- **Geofencing:** Trigger events when driver enters pickup/dropoff zones
- **Battery Optimization:** Reduce update frequency when idle

Scaling Considerations

- **Geohashing:** Partition map into grid cells for sharding
- **QuadTree/S2 Geometry:** Hierarchical spatial indexing for efficient queries
- **Event Streaming:** Kafka for location updates, separate hot path (real-time) from cold path (analytics)
- **Database:** Cassandra for trip history, PostgreSQL with PostGIS for geospatial queries

8. How would you design a notification service that supports multiple channels (email, SMS, push, in-app)? Discuss reliability and delivery guarantees.

Architecture Components

- **API Gateway:** Receive notification requests from services
- **Notification Service:** Orchestrate multi-channel delivery
- **Message Queue:** RabbitMQ/Kafka for reliable queuing
- **Channel Workers:** Specialized workers for email (SendGrid), SMS (Twilio), push (FCM/APNS)
- **Template Service:** Manage notification templates with variables
- **Preference Service:** User notification preferences and opt-outs
- **Analytics Service:** Track delivery, open, click rates

Notification Flow

```
// Publish notification to queue
async function sendNotification(userId, type, data) {
  const user = await getUserPreferences(userId);
  const channels = user.enabledChannels[type];
  for (const channel of channels) {
    await queue.publish('notifications', {
      userId, channel, type, data, retries: 0
    });
  }
}
```

Reliability Patterns

- **At-Least-Once Delivery:** Acknowledge message only after successful delivery
- **Idempotency:** Use unique notification ID to prevent duplicates
- **Retry with Exponential Backoff:** Retry failed deliveries with increasing delays
- **Dead Letter Queue:** Move permanently failed messages after max retries
- **Circuit Breaker:** Stop sending to failing providers temporarily

Priority and Rate Limiting

- **Priority Queues:** Critical (OTP, security) > transactional (orders) > promotional
- **Rate Limiting:** Respect provider limits (e.g., 100 SMS/second)
- **Batching:** Group emails for bulk sending APIs
- **Throttling:** Limit notifications per user to prevent spam

Delivery Guarantees

- **Webhook Callbacks:** Receive delivery status from providers
- **Status Tracking:** Store delivery state (pending, sent, delivered, failed, opened)
- **Fallback Channels:** Try SMS if push fails for critical notifications
- **Deduplication:** Check if notification already sent within time window

9. Design a search engine for an e-commerce platform. How would you implement autocomplete, filtering, and ranking?

Search Architecture

- **Search Engine:** Elasticsearch or Apache Solr for full-text search
- **Data Pipeline:** Stream product updates from database to search index
- **Query Service:** Node.js API to handle search requests
- **Cache Layer:** Redis for popular queries and autocomplete suggestions
- **Analytics:** Track search queries, click-through rates for ranking improvements

Autocomplete Implementation

```
// Trie-based autocomplete with Redis
async function autocomplete(prefix, limit = 10) {
  const cacheKey = `autocomplete:${prefix}`;
  let suggestions = await redis.get(cacheKey);
  if (!suggestions) {
    suggestions = await es.search({
      index: 'products',
      body: { suggest: { text: prefix, completion: { field: 'suggest' } } }
    });
    await redis.setex(cacheKey, 3600, JSON.stringify(suggestions));
  }
  return JSON.parse(suggestions).slice(0, limit);
}
```

Search Indexing Strategy

- **Document Structure:** Include product name, description, category, brand, attributes, price
- **Analyzers:** Tokenization, stemming, synonym expansion, n-grams for partial matching
- **Boosting:** Weight title matches higher than description matches
- **Sharding:** Distribute index across nodes for horizontal scaling
- **Replication:** Multiple replicas for high availability

Filtering and Faceting

- **Aggregations:** Count products per category, brand, price range
- **Multi-Select Filters:** Allow combining filters (category AND brand AND price)
- **Range Filters:** Price, rating, date ranges
- **Nested Filters:** Handle complex attributes (size, color variations)

Ranking Algorithm

- **Relevance Score:** TF-IDF or BM25 for text matching
- **Business Rules:** Boost in-stock items, promoted products, high-margin items
- **Personalization:** User's browsing history, past purchases, location
- **Popularity:** Sales volume, reviews, click-through rate
- **Freshness:** Boost newly added products

Performance Optimization

- **Query Caching:** Cache results for common queries
- **Async Indexing:** Update index asynchronously via message queue
- **Pagination:** Use search_after for deep pagination instead of offset

10. Design a distributed task scheduler system for running background jobs. How would you handle job prioritization, retry logic, and monitoring?

System Components

- **Job Queue:** Redis, RabbitMQ, or AWS SQS for task storage
- **Scheduler Service:** Cron-like service to enqueue recurring jobs
- **Worker Pool:** Node.js workers that consume and execute jobs
- **Coordinator:** Distribute jobs across workers, handle failures
- **State Store:** PostgreSQL or MongoDB for job metadata and history
- **Monitoring:** Prometheus + Grafana for metrics, alerting

Job Queue with Bull

```
// Using Bull for job queue with Redis
const Queue = require('bull');
const jobQueue = new Queue('tasks', { redis: { port: 6379 } });

jobQueue.process('email', async (job) => {
```

```
  await sendEmail(job.data.to, job.data.subject, job.data.body);
});

jobQueue.add('email', { to: 'user@example.com' }, {
  priority: 1, attempts: 3, backoff: { type: 'exponential', delay: 2000 }
});
```

Job Prioritization

- **Priority Levels:** Critical (1), high (2), normal (3), low (4)
- **Weighted Fair Queuing:** Process jobs proportionally by priority
- **Deadline Scheduling:** Prioritize jobs close to deadline
- **Resource-Based:** Assign jobs to workers based on resource requirements (CPU, memory)

Retry Logic and Error Handling

- **Exponential Backoff:** Increase delay between retries (1s, 2s, 4s, 8s...)
- **Max Attempts:** Move to dead letter queue after exhausting retries
- **Idempotency:** Ensure jobs can be safely retried without side effects
- **Timeout:** Kill jobs that exceed max execution time
- **Circuit Breaker:** Pause job type if failure rate exceeds threshold

Distributed Coordination

- **Leader Election:** Use Redis or Zookeeper to elect scheduler leader
- **Job Locking:** Distributed locks to prevent duplicate execution
- **Worker Registration:** Workers register with coordinator via heartbeat
- **Job Partitioning:** Shard jobs by hash to specific workers

Monitoring and Observability

- **Metrics:** Job throughput, latency, failure rate, queue depth
- **Tracing:** Track job lifecycle from enqueue to completion
- **Alerting:** Trigger alerts on high failure rates or queue buildup
- **Dashboard:** Visualize job status, worker health, performance trends

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a function to flatten a nested array in Node.js without using built-in flat() method.

Flattening Nested Arrays

Here's an efficient recursive solution:

```
function flattenArray(arr) {
  return arr.reduce((acc, item) => {
    if (Array.isArray(item)) {
      return acc.concat(flattenArray(item));
    }
    return acc.concat(item);
  }, []);
}

console.log(flattenArray([1, [2, [3, 4], 5], 6]));
// Output: [1, 2, 3, 4, 5, 6]
```

Key points:

- Uses **reduce()** to iterate through array elements
- Recursively processes nested arrays
- Handles arbitrary nesting depth
- Time complexity: O(n) where n is total elements

2. How do you debug memory leaks in Node.js applications? What tools and techniques do you use?

Memory Leak Debugging Strategies

Tools and Techniques:

- **Chrome DevTools:** Use --inspect flag and take heap snapshots to compare memory usage over time
- **process.memoryUsage():** Monitor heapUsed, heapTotal, external, and rss metrics
- **clinic.js:** Provides flame graphs and heap profiling for production apps
- **node --inspect:** Enables remote debugging with Chrome DevTools
- **heapdump module:** Generate heap snapshots programmatically

Common causes: Global variables, event listeners not removed, closures holding references, timers not cleared, and caching without limits.

```
// Monitor memory usage
setInterval(() => {
  const usage = process.memoryUsage();
  console.log(`Heap: ${usage.heapUsed / 1024 / 1024} MB`);
}, 5000);
```

3. Implement a function to check if a string is a palindrome, optimized for performance.

Palindrome Check Implementation

```
function isPalindrome(str) {
  const cleaned = str.toLowerCase().replace(/[^a-z0-9]/g, '');
  let left = 0;
  let right = cleaned.length - 1;

  while (left < right) {
    if (cleaned[left] !== cleaned[right]) return false;
    left++;
    right--;
  }
  return true;
}
```

Optimization features:

- Two-pointer approach with O(n/2) comparisons
- Early exit on mismatch
- Single pass for cleaning and validation
- Space complexity: O(n) for cleaned string
- Handles special characters and case insensitivity

4. What is monkey patching in Node.js? Provide an example and discuss when it's appropriate to use.

Monkey Patching Overview

Monkey patching is the practice of modifying or extending built-in objects or third-party modules at runtime.

```
// Example: Adding a method to Array prototype
Array.prototype.last = function() {
  return this[this.length - 1];
};

const arr = [1, 2, 3, 4];
console.log(arr.last()); // 4
```

Appropriate use cases:

- Polyfills for missing features in older Node.js versions
- Temporary fixes for third-party library bugs
- Testing and mocking dependencies

Risks:

- Can break code that depends on original behavior
- Conflicts with future native implementations
- Makes code harder to maintain and debug
- Better alternatives: wrapper functions, inheritance, composition

5. How do you handle uncaught exceptions and unhandled promise rejections in production Node.js applications?

Exception Handling Best Practices

Critical handlers to implement:

```
process.on('uncaughtException', (err) => {
  console.error('Uncaught Exception:', err);
  // Log to monitoring service
  process.exit(1); // Exit gracefully
});

process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled Rejection at:', promise, 'reason:', reason);
  // Log and handle appropriately
});
```

Production strategies:

- Use process managers like **PM2** for automatic restarts
- Implement centralized error logging (Sentry, DataDog)
- Graceful shutdown: close connections before exit
- Use try-catch for synchronous code
- Always handle promise rejections with .catch() or try-catch with async/await
- Set up health check endpoints for monitoring

6. Write a debounce function from scratch. Explain its use case in Node.js applications.

Debounce Implementation

```
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}

// Usage
const saveData = debounce((data) => {
  console.log('Saving:', data);
}, 300);
```

Node.js use cases:

- **API rate limiting:** Prevent excessive database writes
- **File system operations:** Batch file saves during file watching
- **Search functionality:** Reduce database queries on autocomplete
- **Webhook processing:** Handle burst events efficiently
- **Log aggregation:** Batch log writes to reduce I/O

7. What tools do you use for profiling CPU performance in Node.js? Show how to identify performance bottlenecks.

CPU Profiling Tools and Techniques

Primary profiling tools:

- **Node.js built-in profiler:** --prof flag generates V8 profiler output
- **Chrome DevTools:** --inspect for visual flame graphs
- **clinic.js flame:** Visualizes CPU usage patterns
- **0x:** Generates flame graphs with single command
- **perf (Linux):** System-level profiling

```
// Start profiling
node --prof app.js
// Process output
node --prof-process isolate-0x*.log > profile.txt

// Or use inspector
node --inspect app.js
// Open chrome://inspect
```

Identifying bottlenecks: Look for functions consuming high CPU time, synchronous operations blocking event loop, inefficient algorithms, and excessive JSON parsing/stringification.

8. Implement a memoization function for expensive computations. When should you use memoization in Node.js?

Memoization Implementation

```
function memoize(fn) {
  const cache = new Map();
  return function(...args) {
    const key = JSON.stringify(args);
    if (cache.has(key)) {
      return cache.get(key);
    }
    const result = fn.apply(this, args);
    cache.set(key, result);
    return result;
  };
}
```

Use cases in Node.js:

- **Expensive calculations:** Fibonacci, factorial, complex algorithms
- **API responses:** Cache frequently requested data
- **Database queries:** Store results of repeated queries
- **File system reads:** Cache configuration files

Considerations: Memory usage grows with cache size, implement cache invalidation strategies, use LRU cache for bounded memory, not suitable for functions with side effects.

9. How do you debug asynchronous code in Node.js? What are the common pitfalls and how do you avoid them?

Debugging Asynchronous Code

Debugging techniques:

- **async_hooks module:** Track async operation lifecycle
- **Chrome DevTools:** Set breakpoints in async functions

- **console.trace():** Show call stack for async operations
- **Promise chains:** Add .catch() handlers at each step
- **async/await:** Easier to debug than callbacks

```
// Using async_hooks for tracking
const async_hooks = require('async_hooks');

const hook = async_hooks.createHook({
  init(asyncId, type, triggerAsyncId) {
    console.log(`Async operation: ${type}`);
  }
});
hook.enable();
```

Common pitfalls: Swallowed errors in promises, callback hell, race conditions, not handling rejections, mixing callbacks and promises.

10. Write a function to deep clone an object in Node.js. What are the edge cases you need to handle?

Deep Clone Implementation

```
function deepClone(obj, hash = new WeakMap()) {
  if (obj === null || typeof obj !== 'object') return obj;
  if (obj instanceof Date) return new Date(obj);
  if (obj instanceof RegExp) return new RegExp(obj);
  if (hash.has(obj)) return hash.get(obj);

  const clone = Array.isArray(obj) ? [] : {};
  hash.set(obj, clone);

  Object.keys(obj).forEach(key => {
    clone[key] = deepClone(obj[key], hash);
  });
  return clone;
}
```

Edge cases handled:

- **Circular references:** WeakMap prevents infinite recursion
- **Date objects:** Create new Date instances
- **RegExp objects:** Preserve pattern and flags
- **Arrays vs Objects:** Maintain correct type
- **Null values:** Return as-is
- **Functions:** Not cloned (by design)
- **Symbols:** Need Object.getOwnPropertySymbols() for full support

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you optimized the performance of a Node.js application.

Situation: Our e-commerce API was experiencing high response times (>2s) during peak traffic, causing customer complaints and cart abandonments.

Task: I was tasked with reducing API response times to under 500ms and improving throughput to handle 10,000 concurrent users.

Action: I implemented several optimizations: added Redis caching for frequently accessed product data, introduced connection pooling for database queries, replaced synchronous operations with async/await patterns, and implemented clustering to utilize all CPU cores.

```
const cluster = require('cluster');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
} else {
  require('./server');
}
```

Result: Response times dropped to an average of 350ms, we successfully handled Black Friday traffic with 15,000 concurrent users, and customer satisfaction scores improved by 23%.

2. Describe a situation where you had to debug a critical production issue in Node.js.

Situation: Our payment processing service started crashing randomly in production, affecting approximately 15% of transactions and causing significant revenue loss.

Task: I needed to identify the root cause quickly and implement a fix without causing further downtime.

Action: I analyzed application logs and discovered memory leak patterns. Using heap snapshots and the clinic.js tool, I identified that event listeners weren't being properly removed after WebSocket connections closed. I implemented proper cleanup:

```
socket.on('close', () => {
  socket.removeAllListeners();
  clearInterval(heartbeatInterval);
  connectionPool.delete(socket.id);
});
```

I also added monitoring with PM2 and set up automatic restarts as a safety measure.

Result: The crashes stopped completely, transaction success rate returned to 99.8%, and we recovered approximately $50,000 in daily revenue. I also documented the issue and created guidelines for proper event listener management.

3. Give an example of how you handled a disagreement with a team member about technical implementation.

Situation: During a microservices migration, a senior colleague wanted to use REST APIs for inter-service communication, while I advocated for implementing message queues with RabbitMQ for better decoupling.

Task: We needed to reach consensus on the architecture that would best serve our scalability and reliability requirements.

Action: I organized a technical discussion where we both presented our approaches with pros and cons. I created a proof-of-concept demonstrating how message queues would handle service failures gracefully and prevent cascading failures. I also showed performance benchmarks comparing both approaches under load. Rather than insisting on my solution, I proposed a hybrid approach: using message queues for asynchronous operations and REST for synchronous queries.

Result: The team agreed on the hybrid approach, which provided the best of both worlds. The implementation proved successful, reducing service coupling by 60% and improving system resilience. My colleague and I strengthened our working relationship through respectful technical debate.

4. Tell me about a time when you had to learn a new Node.js technology or framework quickly.

Situation: Our company acquired a startup whose codebase was built entirely with NestJS, a framework I had never used, and I was assigned to lead the integration team.

Task: I needed to become proficient in NestJS within two weeks to effectively lead the team and make architectural decisions for integrating their services with our existing Node.js/Express infrastructure.

Action: I created a structured learning plan: spent the first three days reading official documentation and building a sample microservice, then reviewed the acquired codebase to understand their patterns, paired with their senior developer for knowledge transfer, and participated in their code reviews. I also created documentation comparing NestJS concepts with Express patterns for my team.

Result: Within two weeks, I successfully led the integration project, identified optimization opportunities in their existing code, and completed the migration one week ahead of schedule. The hybrid system now handles 50% more traffic, and I became the internal expert on NestJS, later conducting training sessions for other team members.

5. Describe a situation where you improved code quality or development practices in your Node.js team.

Situation: Our Node.js team was experiencing frequent production bugs, inconsistent code styles, and difficult code reviews that were slowing down our release cycle.

Task: I was asked to improve our development practices and reduce the defect rate by at least 40% within three months.

Action: I introduced several improvements systematically: implemented ESLint and Prettier for consistent code formatting, established TypeScript for better type safety, created comprehensive testing standards with Jest requiring 80% coverage, set up Husky for pre-commit hooks, and implemented CI/CD pipelines with automated testing. I also organized weekly code review sessions to share best practices:

```
// .eslintrc.js
module.exports = {
  extends: ['airbnb-base', 'plugin:@typescript-eslint/recommended'],
  rules: {
    'no-console': 'error',
    'max-len': ['error', { code: 100 }]
  }
};
```

Result: Production bugs decreased by 65%, code review time reduced by 50%, and team velocity increased by 30%. Developer satisfaction improved significantly, and other teams adopted our practices as company standards.

6. Tell me about a time when you had to make a trade-off between code quality and meeting a deadline.

Situation: We had a critical feature required for a major client demo in three days, but implementing it properly with full test coverage and optimal architecture would

take at least five days.

Task: I needed to deliver a working feature for the demo while ensuring we wouldn't create long-term technical debt that would slow future development.

Action: I proposed a phased approach to stakeholders: build a functional MVP with core features for the demo, but architect it properly with clear interfaces and separation of concerns. I focused on critical path functionality, wrote integration tests for main flows (skipping some unit tests temporarily), and used feature flags to isolate the new code. I documented all shortcuts taken and created tickets for technical debt:

```
// Feature flag approach
if (featureFlags.isEnabled('new-payment-flow')) {
  return await newPaymentService.process(order);
}
return await legacyPaymentService.process(order);
```

I also scheduled a two-day refactoring sprint immediately after the demo.

Result: The demo was successful and we secured the client contract worth $500K. We completed the refactoring as planned, achieving 85% test coverage and zero production issues. Stakeholders appreciated the transparency and realistic planning.

7. Describe a time when you mentored a junior developer or helped a team member grow their Node.js skills.

Situation: A junior developer joined our team with basic JavaScript knowledge but no experience with Node.js, asynchronous programming, or backend development.

Task: I was assigned as their mentor with the goal of bringing them to a productive mid-level capability within six months.

Action: I created a personalized learning path starting with Node.js fundamentals, then progressively introducing more complex topics. We had weekly pairing sessions where I explained concepts like the event loop, promises, and streams. I assigned increasingly challenging tasks with detailed code reviews, focusing on teaching rather than just correcting. For example, when they struggled with callback hell, I walked them through refactoring:

```
// Before: callback hell
fs.readFile('file.txt', (err, data) => {
  processData(data, (err, result) => {
    saveResult(result, (err) => { /* ... */ });
  });
});
```

I encouraged them to present their work in team meetings and nominated them for their first production deployment.

Result: Within five months, they were independently handling medium-complexity features and contributing to architectural discussions. They successfully led a microservice implementation that processed 100K daily requests. Their confidence grew significantly, and they later became a mentor themselves.

8. Tell me about a time when you had to refactor legacy Node.js code without breaking existing functionality.

Situation: Our authentication service was built three years ago with callback-based code, no tests, and tightly coupled dependencies. It was becoming a bottleneck for new features and had accumulated significant technical debt.

Task: I needed to refactor the service to use modern async/await patterns, add proper error handling, and implement dependency injection—all while maintaining 100% uptime for our 200,000 active users.

Action: I adopted a strangler fig pattern: first, I wrote comprehensive integration tests to document existing behavior. Then I refactored incrementally, one module at a time, starting with the least critical components. I used TypeScript interfaces to define contracts and implemented the adapter pattern to maintain backward compatibility:

```
class AuthService {
  constructor(private db: IDatabase, private cache: ICache) {}

  async authenticate(credentials: Credentials): Promise {
    const user = await this.db.findUser(credentials.email);
    return this.generateToken(user);
  }
}
```

I deployed changes using feature flags and monitored metrics closely.

Result: Completed the refactor over six weeks with zero downtime. Code coverage increased from 0% to 92%, authentication response time improved by 40%, and the new architecture enabled us to add OAuth2 support in just two days—something previously estimated at two weeks.

9. Describe a situation where you had to handle a security vulnerability in a Node.js application.

Situation: A security audit revealed our API was vulnerable to NoSQL injection attacks through user input in MongoDB queries, and we had several endpoints accepting unsanitized user data.

Task: I needed to patch the vulnerability immediately across all affected endpoints, implement proper input validation, and ensure similar issues wouldn't occur in the future.

Action: I first conducted a comprehensive code review to identify all vulnerable endpoints. I implemented input sanitization using validator.js and parameterized queries, replaced direct string concatenation in queries with proper MongoDB operators, and added rate limiting to prevent abuse:

```
// Fixed vulnerable code
const sanitize = require('mongo-sanitize');

const findUser = async (email) => {
  const clean = sanitize(email);
  return await User.findOne({ email: clean });
};
```

I also established security guidelines, integrated Snyk for dependency scanning in CI/CD, and conducted a security training session for the team.

Result: All vulnerabilities were patched within 48 hours with no exploitation detected. We prevented potential data breaches affecting 500,000 user accounts. The security practices I established became part of our development standards, and subsequent audits showed zero critical vulnerabilities.

10. Tell me about a time when you designed and implemented a scalable architecture for a Node.js application.

Situation: Our monolithic Node.js application was struggling to handle growing traffic (from 10K to 100K daily users in six months), experiencing frequent slowdowns and occasional crashes during peak hours.

Task: I was tasked with redesigning the architecture to handle 500K daily users while maintaining high availability and enabling faster feature development.

Action: I proposed and led a migration to microservices architecture. I identified service boundaries based on business domains, implemented an API gateway with rate limiting, introduced Redis for caching and session management, set up RabbitMQ for asynchronous processing, and containerized services with Docker and Kubernetes for orchestration:

```
// API Gateway pattern
const gateway = express();

gateway.use('/users', createProxyMiddleware({
  target: process.env.USER_SERVICE_URL,
  changeOrigin: true
}));
```

I implemented circuit breakers to prevent cascading failures and established comprehensive monitoring with Prometheus and Grafana.

Result: The new architecture successfully handled 600K daily users during launch. Response times improved by 60%, deployment frequency increased from weekly to daily, and system availability reached 99.95%. The modular design reduced time-to-market for new features by 40%.