Head-of-Line Blocking

1. The Real Problem This Exists to Solve

When a single slow request blocks an entire queue or connection, all subsequent requests must wait for the slow one to complete, even though they could be processed independently. This creates cascading delays where one problem request affects thousands of healthy requests.

Real production scenario:

- WebSocket server handling 1000 concurrent connections
- Each connection processes messages sequentially from a queue
- Client sends normal messages (10ms to process) followed by one expensive message (30s to process)
- The 30s message enters the queue
- All subsequent messages for that connection wait 30 seconds
- User sends 100 more messages during that time
- All 100 messages queued behind the slow one
- User experiences 30+ second delay for simple operations
- Other connections work fine, but this connection is completely blocked

The fundamental problem: A single slow operation at the head of a queue/stream blocks all operations behind it, even when those operations are independent and could be processed in parallel or on different resources.

This occurs in multiple contexts:

- HTTP/1.1: One slow request blocks all requests on same TCP connection
- Message queues: One poison message blocks queue processing
- Database connections: One long query blocks connection for other queries
- Stream processing: One slow record blocks entire stream partition

Without understanding head-of-line blocking:

- Engineers create single queues that serialize everything
- One slow operation degrades entire system
- Users experience inconsistent latency (sometimes fast, sometimes blocked)
- Debugging is hard (why is fast operation slow?)

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Single Queue for All Requests

```
// Incorrect: All requests share single queue, processed sequentially
class RequestProcessor {
  private queue: Request[] = [];
  private processing = false;

  async enqueue(request: Request): Promise<Response> {
    return new Promise((resolve, reject) => {
      this.queue.push({ request, resolve, reject });
      this.processQueue();
    });
  }
  private async processQueue() {
    if (this.processing) return;
    this.processing = true;

    while (this.queue.length > 0) {
      const { request, resolve, reject } = this.queue.shift()!;
      try {
        const response = await this.processRequest(request);
        resolve(response);
      } catch (error) {
        reject(error);
      }
    }

    this.processing = false;
  }

  private async processRequest(request: Request): Promise<Response> {
    // Could take 10ms or 30s depending on request
    return await database.query(request.sql);
  }
}
```

Why it seems reasonable:

- Guarantees order (FIFO)
- Simple to implement
- No concurrency issues
- Single queue easy to reason about

How it breaks:

- One slow request (30s database query) blocks entire queue
- Fast requests (10ms lookup) wait 30s behind slow one
- Queue depth grows as requests pile up behind slow one
- All users experience slow response, not just the user with slow request
- System throughput drops to 1/3000 of normal (30s per request vs 10ms)

Production symptoms:

- Latency spikes affect all users simultaneously
- Queue depth grows from 0 to 1000+ when one slow request enters
- Slow query log shows one 30s query followed by thousands of fast queries
- Users report entire system becoming slow periodically
- p99 latency equals the slowest query time (30s)

❌ Incorrect Approach #2: HTTP/1.1 Connection Pooling Without Pipelining

```
// Incorrect: Single connection handles one request at a time
class HTTPClient {
  private connection: Connection;
  async request(url: string): Promise<Response> {
    // Wait for connection to be free
    await this.waitForConnection();

    try {
      const response = await this.connection.send(url);
      return response;
    } finally {
      this.releaseConnection();
    }
  }

  private async waitForConnection() {
    while (this.connection.inUse) {
      await sleep(10);
    }
    this.connection.inUse = true;
  }

  private releaseConnection() {
    this.connection.inUse = false;
  }
}
```

Why it seems reasonable:

- Standard HTTP/1.1 pattern
- Connection reuse is efficient
- Prevents creating too many connections

How it breaks:

- If first request takes 10s (slow backend), all subsequent requests on this connection wait 10s
- Even though requests are independent (different URLs)
- Client has 6 connections to server, but one slow request blocks 1/6 of capacity
- Other connections are fine, but requests randomly hit slow connection
- Users experience unpredictable latency (fast on some connections, slow on others)

Production symptoms:

- Latency varies wildly for identical requests (10ms or 10s)
- Connection pool has idle connections while requests are queued
- Network traces show gaps where connection is idle after slow request
- Some users experience fast responses, others slow, seemingly random
- Increasing connection pool size helps but doesn't eliminate problem

❌ Incorrect Approach #3: Stream Processing Without Parallelism

```
// Incorrect: Process stream records sequentially
async function processStream(stream: ReadableStream) {
  for await (const record of stream) {
    await processRecord(record);  // Could be fast or slow
  }
}

async function processRecord(record: Record): Promise<void> {
  if (record.type === 'fast') {
    await fastOperation(record);  // 10ms
  } else if (record.type === 'slow') {
    await slowOperation(record);  // 30s
  }
}
```

Why it seems reasonable:

- Preserves order
- Simple iteration pattern
- Low memory usage (one record at a time)

How it breaks:

- One slow record blocks entire stream
- Fast records behind slow one wait 30s
- Stream throughput drops to 1/3000 when slow record appears
- Backpressure builds up in stream
- Upstream producers are throttled even though downstream could handle more fast records

Production symptoms:

- Stream lag increases dramatically when slow records appear
- Consumer lag goes from 0 to hours
- Fast records delayed by unrelated slow records
- Stream partition becomes "stuck"
- Rebalancing doesn't help (record order preserved)

❌ Incorrect Approach #4: Database Connection Waiting for Long Transaction

```
// Incorrect: Connection pool with one connection blocked by long transaction
async function executeQuery(sql: string) {
  const conn = await pool.getConnection();

  try {
    // Could be fast query or long transaction
    return await conn.query(sql);
  } finally {
    conn.release();
  }
}

// Long transaction blocks connection for 30s
await executeQuery('BEGIN');
await executeQuery('SELECT ... FOR UPDATE');  // Holds locks
await sleep(30000);  // Long processing
await executeQuery('COMMIT');
```

```
// Meanwhile, all other queries wanting this connection wait
```

Why it seems reasonable:

- Standard connection pooling pattern
- Transactions require connection affinity
- Pool size matches database capacity

How it breaks:

- Long transaction holds connection for 30s
- Other queries needing a connection wait for one to become available
- Pool exhaustion: all connections blocked by long transactions
- Simple queries (1ms) wait 30s for connection
- Database has spare capacity but clients can't use it

Production symptoms:

- Connection pool exhausted errors
- Database CPU at 30% but clients timing out
- Connection acquisition time high (10-30s)
- Short queries show long "waiting for connection" time
- Increasing pool size helps temporarily but fills up again

3. Correct Mental Model (How It Actually Works)

Head-of-line blocking occurs when:

1. Operations are processed sequentially from a queue/stream
2. Operations have variable processing time
3. Operations are independent (don't require serialization)

The Blocking Pattern

```
Queue: [Fast, Fast, SLOW, Fast, Fast, Fast, ...]

Time 0ms:    Fast (10ms) ▓
Time 10ms:   Fast (10ms) ▓
Time 20ms:   SLOW (30s)  ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓
Time 30020ms: Fast (10ms) ▓
Time 30030ms: Fast (10ms) ▓
Time 30040ms: Fast (10ms) ▓

Result: 3 fast operations (30ms total) took 30,060ms
```

The slow operation at the head blocks all operations behind it.

The Solution: Parallel Processing

```
Queue: [Fast, Fast, SLOW, Fast, Fast, Fast, ...]

Process in parallel with concurrency limit:
```

```
Worker 1: Fast (10ms)  ▓
Worker 2: Fast (10ms)  ▓
Worker 3: SLOW (30s)   ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓
Worker 4: Fast (10ms)  ▓
Worker 5: Fast (10ms)  ▓
Worker 6: Fast (10ms)  ▓

Result: 5 fast operations complete in 10ms, slow one takes 30s but doesn't block
others
```

Key Insights

Serialization is not always required:

- HTTP requests are independent (can process in parallel)
- Message queue messages often independent
- Database queries on different tables are independent

HTTP/2 multiplexing solves HTTP/1.1 HOL blocking:

- HTTP/1.1: One request at a time per connection
- HTTP/2: Multiple parallel streams over one connection
- Slow response on stream 1 doesn't block stream 2

Parallelism introduces complexity:

- Order is not preserved
- Need to handle concurrent operations safely
- May need request prioritization

4. Correct Design & Algorithm

Strategy 1: Parallel Processing with Concurrency Limit

Don't process sequentially. Process N operations in parallel:

```
while (queue.notEmpty() && activeWorkers < maxConcurrency) {
  task = queue.dequeue()
  startWorker(task)  // Process in parallel, don't wait
}
```

Strategy 2: Separate Queues by Priority/Latency

Fast operations and slow operations use different queues:

```
fastQueue = Queue()  // Expected latency: <100ms
slowQueue = Queue()  // Expected latency: >1s

Classify request:
  if (isExpectedSlow(request))
    slowQueue.enqueue(request)
  else
    fastQueue.enqueue(request)
```

```
Process both queues concurrently
```

Strategy 3: Timeout and Skip

Set timeout for operations. If one times out, skip it and continue:

```
for task in queue:
  try:
    await withTimeout(process(task), timeout)
  catch TimeoutError:
    log("Task timed out, skipping")
    continue  // Don't let one slow task block others
```

Strategy 4: Use HTTP/2 or HTTP/3

For HTTP clients, use HTTP/2+ which multiplexes streams:

```
HTTP/1.1: 6 connections × 1 request each = 6 concurrent
HTTP/2:   1 connection × 100 streams = 100 concurrent (no HOL blocking)
```

5. Full Production-Grade Implementation

```typescript
interface Task<T> {
  id: string;
  fn: () => Promise<T>;
  priority: number;
  timeout: number;
  resolve: (value: T) => void;
  reject: (error: Error) => void;
}

interface WorkerPoolOptions {
  concurrency: number;
  enablePriority: boolean;
  defaultTimeout: number;
  separateSlowLane: boolean;
  slowThreshold: number;
}

class ParallelProcessor {
  private fastQueue: Task<any>[] = [];
  private slowQueue: Task<any>[] = [];
  private activeWorkers = 0;
  private readonly options: WorkerPoolOptions;
  private taskLatencies: Map<string, number[]> = new Map();

  constructor(options: Partial<WorkerPoolOptions> = {}) {
    this.options = {
      concurrency: options.concurrency ?? 10,
      enablePriority: options.enablePriority ?? true,
      defaultTimeout: options.defaultTimeout ?? 30000,
      separateSlowLane: options.separateSlowLane ?? true,
      slowThreshold: options.slowThreshold ?? 1000,
    };

    // Start processing loop
    this.processLoop();
  }

  /**
   * Submit task for parallel processing (no head-of-line blocking)
   */
  async submit<T>(
    taskFn: () => Promise<T>,
    options: {
      id?: string;
      priority?: number;
      timeout?: number;
      expectedSlow?: boolean;
    } = {}
  ): Promise<T> {
    const {
      id = crypto.randomUUID(),
      priority = 0,
      timeout = this.options.defaultTimeout,
      expectedSlow = false,
    } = options;

    return new Promise((resolve, reject) => {
      const task: Task<T> = {
        id,
        fn: taskFn,
        priority,
        timeout,
        resolve,
        reject,
      };

      // Classify task: fast lane or slow lane
      if (this.options.separateSlowLane && expectedSlow) {
        this.slowQueue.push(task);
      } else {
        this.fastQueue.push(task);
      }

      // Wake up processing loop
      this.processLoop();
    });
  }

  /**
   * Processing loop: continuously pull tasks and execute in parallel
```

```typescript
   */
  private async processLoop(): Promise<void> {
    while (this.activeWorkers < this.options.concurrency) {
      const task = this.dequeueTask();
      if (!task) break;

      // Start worker (don't await - parallel execution)
      this.activeWorkers++;
      this.executeTask(task).finally(() => {
        this.activeWorkers--;
        // Check for more work
        setImmediate(() => this.processLoop());
      });
    }
  }

  /**
   * Dequeue next task with priority and slow/fast lane separation
   */
  private dequeueTask(): Task<any> | null {
    // Prioritize fast queue over slow queue (80/20 split)
    const useFastQueue = Math.random() < 0.8 || this.slowQueue.length === 0;

    const queue = useFastQueue ? this.fastQueue : this.slowQueue;

    if (queue.length === 0) {
      // Try other queue
      const otherQueue = useFastQueue ? this.slowQueue : this.fastQueue;
      if (otherQueue.length === 0) return null;
      return this.dequeueFromQueue(otherQueue);
    }

    return this.dequeueFromQueue(queue);
  }

  /**
   * Dequeue with priority sorting
   */
  private dequeueFromQueue(queue: Task<any>[]): Task<any> | null {
    if (queue.length === 0) return null;

    if (this.options.enablePriority) {
      // Sort by priority (higher first)
      queue.sort((a, b) => b.priority - a.priority);
    }

    return queue.shift()!;
  }

  /**
   * Execute single task with timeout
   */
```

```typescript
private async executeTask<T>(task: Task<T>): Promise<void> {
  const startTime = Date.now();

  try {
    // Execute with timeout to prevent indefinite blocking
    const result = await this.withTimeout(task.fn(), task.timeout);
    const latency = Date.now() - startTime;

    this.recordLatency(task.id, latency);
    task.resolve(result);

    console.log(`[Worker] Task ${task.id} completed in ${latency}ms`);
  } catch (error) {
    const latency = Date.now() - startTime;
    this.recordLatency(task.id, latency);

    if (error instanceof TimeoutError) {
      console.warn(`[Worker] Task ${task.id} timed out after ${latency}ms`);
      task.reject(new Error(`Task timeout after ${task.timeout}ms`));
    } else {
      console.error(`[Worker] Task ${task.id} failed:`, error);
      task.reject(error as Error);
    }
  }
}

/**
 * Execute promise with timeout
 */
private async withTimeout<T>(
  promise: Promise<T>,
  timeoutMs: number
): Promise<T> {
  let timeoutId: NodeJS.Timeout;

  const timeoutPromise = new Promise<never>((_, reject) => {
    timeoutId = setTimeout(() => {
      reject(new TimeoutError(`Operation timed out after ${timeoutMs}ms`));
    }, timeoutMs);
  });

  try {
    return await Promise.race([promise, timeoutPromise]);
  } finally {
    clearTimeout(timeoutId!);
  }
}

/**
 * Record task latency for analysis
 */
private recordLatency(taskId: string, latency: number): void {
    if (!this.taskLatencies.has(taskId)) {
      this.taskLatencies.set(taskId, []);
    }
    this.taskLatencies.get(taskId)!.push(latency);

    // Keep only recent samples
    const samples = this.taskLatencies.get(taskId)!;
    if (samples.length > 100) {
      samples.shift();
    }
  }

  /**
   * Get metrics for monitoring
   */
  getMetrics() {
    return {
      fastQueueDepth: this.fastQueue.length,
      slowQueueDepth: this.slowQueue.length,
      activeWorkers: this.activeWorkers,
      maxConcurrency: this.options.concurrency,
      utilization: this.activeWorkers / this.options.concurrency,
    };
  }
}

class TimeoutError extends Error {
  constructor(message: string) {
    super(message);
    this.name = 'TimeoutError';
  }
}

// Example: API request processor without HOL blocking
const processor = new ParallelProcessor({
  concurrency: 50,
  enablePriority: true,
  separateSlowLane: true,
  slowThreshold: 1000,
});

app.get('/api/:endpoint', async (req, res) => {
  try {
    const result = await processor.submit(
      async () => {
        return await processAPIRequest(req);
      },
      {
        id: req.id,
        priority: req.headers['x-priority'] ? 10 : 0,
        expectedSlow: req.params.endpoint === 'analytics',
        timeout: 10000,
      }
    );

    res.json(result);
  } catch (error) {
    if (error.message.includes('timeout')) {
      res.status(504).json({ error: 'Request timeout' });
    } else {
      res.status(500).json({ error: 'Internal error' });
    }
  }
});

// Example: Message queue consumer without HOL blocking
async function consumeMessages(queue: MessageQueue) {
  while (true) {
    const messages = await queue.poll(10); // Get 10 messages

    // Submit all messages in parallel (no HOL blocking)
    await Promise.all(
      messages.map(msg =>
        processor.submit(
          () => processMessage(msg),
          {
            id: msg.id,
            timeout: 30000,
            expectedSlow: msg.type === 'batch-report',
          }
        )
      )
    );
  }
}

// Example: HTTP/2 client (no HOL blocking at protocol level)
import http2 from 'http2';

const client = http2.connect('https://api.example.com');

async function makeRequest(path: string): Promise<any> {
  return new Promise((resolve, reject) => {
    const req = client.request({ ':path': path });

    let data = '';
    req.on('data', chunk => { data += chunk; });
    req.on('end', () => resolve(JSON.parse(data)));
    req.on('error', reject);
    req.end();
  });
}

// Multiple requests in parallel over single connection (no HOL blocking)
```

```
const results = await Promise.all([
  makeRequest('/fast'),
  makeRequest('/slow'),  // Doesn't block /fast
  makeRequest('/fast'),
]);

// Monitoring
app.get('/metrics/processor', (req, res) => {
  const metrics = processor.getMetrics();
  res.json({
    fast_queue_depth: metrics.fastQueueDepth,
    slow_queue_depth: metrics.slowQueueDepth,
    active_workers: metrics.activeWorkers,
    utilization: `${(metrics.utilization * 100).toFixed(1)}%`,
  });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Message Queue Consumer

Process messages in parallel to avoid one slow message blocking others:

```
async function processQueue(queueUrl: string) {
  while (true) {
    const messages = await sqs.receiveMessage({ QueueUrl: queueUrl,
MaxNumberOfMessages: 10 });

    // Process in parallel instead of sequentially
    await Promise.all(
      messages.Messages.map(msg =>
        processor.submit(() => handleMessage(msg), {
          timeout: 30000,
          expectedSlow: msg.body.includes('slow-operation'),
        })
      )
    );
  }
}
```

Why this works:

- One slow message (30s) doesn't block other 9 messages
- Fast messages complete in 10ms while slow one runs
- Queue lag doesn't spike when slow message appears
- Throughput maintained even with mixed fast/slow messages

Pattern 2: HTTP API Gateway

Route requests through parallel processor:

```
app.use(async (req, res, next) => {
  try {
    await processor.submit(
      () => handleRequest(req, res),
      {
        id: req.id,
        timeout: 30000,
        priority: req.user.isPremium ? 10 : 0,
      }
    );
  } catch (error) {
    next(error);
  }
});
```

Why this works:

- One slow database query doesn't block other API calls
- Users with fast operations get fast responses
- Premium users get priority (processed first)
- Slow operations isolated in slow lane

Pattern 3: WebSocket Message Handler

Handle WebSocket messages without blocking:

```
ws.on('message', async (data) => {
  const msg = JSON.parse(data);

  // Don't await - process in parallel
  processor.submit(
    async () => {
      const result = await handleWebSocketMessage(msg);
      ws.send(JSON.stringify({ id: msg.id, result }));
    },
    { timeout: 5000 }
  ).catch(err => {
    ws.send(JSON.stringify({ id: msg.id, error: err.message }));
  });
});
```

Why this works:

- One expensive message doesn't block subsequent messages
- User can send 100 messages, all processed in parallel
- Slow messages timeout, don't indefinitely block
- Fast messages complete immediately

7. Failure Modes & Edge Cases

Timeout Too Aggressive

Problem: Timeout shorter than legitimate operation time, causes failures.

Symptoms:

- High timeout rate (20%+)
- Operations partially complete then fail
- Database shows completed queries but client shows timeout

Mitigation:

- Set timeout to p99 latency × 3
- Monitor timeout rate, adjust timeout if >1%
- Separate timeouts for fast lane vs slow lane

Memory Exhaustion from Large Queues

Problem: Parallel processing allows unbounded queue growth.

Symptoms:

- Fast queue grows to 10,000+ tasks
- Memory usage GB+
- OOM killer terminates process

Mitigation:

- Set max queue depth
- Reject new tasks when queue full (backpressure)
- Monitor queue depth

Order Violations

Problem: Parallel processing breaks ordering assumptions.

Symptoms:

- Messages processed out of order
- Database writes race (later write happens first)
- Business logic depends on order, breaks

Mitigation:

- Don't use parallel processing when order matters
- Use partitioning (same key → same worker, order preserved within partition)
- Explicitly sequence dependent operations

Resource Exhaustion

Problem: Too many parallel operations exhaust resources (connections, memory).

Symptoms:

- Database connection pool exhausted
- Out of memory
- Too many open file descriptors

Mitigation:

- Set concurrency limit
- Use bulkhead pattern (separate pools for different operations)

- Monitor resource usage

8. Performance Characteristics & Tradeoffs

Latency Improvement

Sequential processing:

- One slow operation (30s) blocks 100 fast operations (10ms each)
- Total time: 30,000ms + (100 × 10ms) = 31,000ms
- Average latency: 310ms per operation

Parallel processing (concurrency=10):

- Slow operation takes 30s
- 100 fast operations complete in 10 batches × 10ms = 100ms
- Total time: max(30,000ms, 100ms) = 30,000ms
- Average latency for fast operations: 10ms (99% of operations)

Improvement: 31× better average latency for fast operations

Throughput Improvement

Sequential:

- 101 operations / 31 seconds = 3.3 ops/sec

Parallel:

- 100 fast operations / 0.1 seconds = 1000 ops/sec (for fast operations)
- 1 slow operation / 30 seconds = 0.03 ops/sec (for slow operations)

Improvement: 300× better throughput overall

Memory Tradeoff

Sequential:

- One operation in memory at a time
- Memory: O(1)

Parallel:

- Concurrency=50: 50 operations in memory
- Memory: O(concurrency)

Cost: 50× more memory per request type

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: Parallel Processing When Order Matters

Why engineers do it: "Parallelism is always faster."

What breaks: Operations processed out of order, violates business logic.

```
// WRONG: Parallel processing breaks order
await Promise.all([
  updateBalance(userId, -100),  // Could execute second
  updateBalance(userId, +50),   // Could execute first
]);
// Balance incorrect!
```

Fix: Serialize operations when order matters.

Mistake 2: No Timeout

Why engineers do it: "Let operations complete naturally."

What breaks: One hung operation blocks worker forever.

Fix: Always set timeout.

```
await processor.submit(() => operation(), { timeout: 30000 });
```

Mistake 3: Same Priority for All Tasks

Why engineers do it: "All tasks are equal."

What breaks: Critical operations wait behind non-critical.

Fix: Use priority for critical operations.

```
processor.submit(() => criticalOp(), { priority: 10 });
processor.submit(() => batchJob(), { priority: 1 });
```

Mistake 4: No Slow Lane Separation

Why engineers do it: "One queue is simpler."

What breaks: Known-slow operations consume workers, starve fast operations.

Fix: Separate slow lane.

```
processor.submit(() => slowOp(), { expectedSlow: true });
```

Mistake 5: HTTP/1.1 with Low Connection Limit

Why engineers do it: "Don't want too many connections."

What breaks: Head-of-line blocking on each connection.

Fix: Use HTTP/2 or increase connection pool size.

```
// Use HTTP/2
const client = http2.connect('https://api.example.com');

// Or increase HTTP/1.1 pool size
const agent = new https.Agent({ maxSockets: 100 });
```

10. When NOT to Use This (Anti-Patterns)

Anti-Pattern 1: Operations Must Be Ordered

Don't parallelize when order is required:

```
// WRONG
await Promise.all([
  createUser(data),
  assignRole(userId, 'admin'),  // Depends on createUser!
]);
```

Anti-Pattern 2: Resource-Constrained Operations

Don't parallelize operations that exhaust shared resources:

```
// WRONG: Exhaust database connections
await Promise.all(
  Array(1000).fill(0).map(() => db.query('SELECT ...'))
);
// Database connection pool only has 50 connections!
```

Anti-Pattern 3: Operations with Side Effects

Don't parallelize when operations have non-idempotent side effects:

```
// WRONG
await Promise.all([
  incrementCounter(key),  // Race condition
  incrementCounter(key),  // Race condition
]);
// Counter only incremented once instead of twice
```

11. Related Concepts (With Contrast)

Queueing Theory

Relationship: HOL blocking is a specific failure mode of queues. Parallel processing reduces queue wait time.

Connection Pooling

Relationship: HOL blocking happens within connection pools. HTTP/2 multiplexing eliminates it.

Backpressure

Difference: Backpressure limits incoming rate. HOL blocking mitigation processes existing load in parallel.

Load Balancing

Difference: Load balancing distributes across servers. HOL blocking mitigation distributes within a server.

12. Production Readiness Checklist

Metrics to Monitor

- [ ] Fast queue depth
- [ ] Slow queue depth
- [ ] Active workers
- [ ] Worker utilization
- [ ] Timeout rate
- [ ] Average latency (fast vs slow operations)

Configuration

- [ ] Set concurrency limit based on resources
- [ ] Set default timeout based on p99 latency
- [ ] Enable slow lane separation
- [ ] Configure priority levels

Testing

- [ ] Test mixed fast/slow operations
- [ ] Verify slow operation doesn't block fast ones
- [ ] Test timeout behavior
- [ ] Test queue full rejection
- [ ] Load test at max concurrency

Rollout

- [ ] Deploy with high timeout (conservative)
- [ ] Monitor timeout rate
- [ ] Tune timeout and concurrency
- [ ] Enable slow lane separation

Alerting

- [ ] Alert if timeout rate > 1%
- [ ] Alert if queue depth > 1000
- [ ] Alert if worker utilization > 90%
- [ ] Alert if slow operations blocking fast lane