Queueing Theory in Practice

1. The Real Problem This Exists to Solve

Engineers vastly underestimate how queue depth and latency explode as systems approach capacity. A service running at 70% utilization can have acceptable latency, but at 90% utilization, latency becomes unbounded and the system approaches collapse.

Real production scenario:

- API server has capacity of 1000 requests/second
- Current load: 700 req/s (70% utilization)
- Average response time: 50ms
- Everything seems fine, plenty of headroom (30%)
- Marketing launches campaign, traffic increases to 900 req/s (90% utilization)
- Average response time suddenly jumps to 500ms (10× increase)
- Queue depth grows from 5 to 50 requests
- Engineers add more servers, but damage is done (users already frustrated)

The non-intuitive reality: The last 20% of capacity increase causes 10× latency increase. This is not a bug—it's fundamental mathematics of queueing systems.

Without understanding queueing theory, engineers:

- Overestimate system capacity ("we have 30% headroom")
- Don't understand why latency explodes at high utilization
- Add capacity too late (after users experience slowness)
- Can't predict when system will collapse under load
- Set wrong SLAs (promise p99 < 100ms at utilization where it's mathematically impossible)

The problem: **Queues hide capacity limits until you're dangerously close to collapse**.

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Ignoring Queue Buildup

```
// Incorrect: No queue depth tracking or limits
class APIServer {
  async handleRequest(req: Request): Promise<Response> {
    // Just process every incoming request
    return await processRequest(req);
  }
}

app.use(async (req, res) => {
  const result = await apiServer.handleRequest(req);
  res.json(result);
});
```

Why it seems reasonable:

- Simple implementation

- Process all incoming requests
- Node.js event loop handles concurrency automatically
- No explicit queue management needed

How it breaks:

- As load increases, event loop queue grows unbounded
- Each request waits for all previous requests to complete
- At 90% utilization, queue depth reaches 50-100 requests
- Request that arrives sees 50 requests ahead of it
- Wait time = 50 × 50ms = 2500ms just in queue
- Total latency = queue wait + processing = 2.5s + 50ms
- Users timeout, retry, making queue even worse

Production symptoms:

- Latency increases exponentially as traffic increases
- Queue depth grows from 5 to 500 during traffic spikes
- Memory usage spikes (queued request objects)
- Old requests timeout while waiting in queue
- CPU usage is fine (<80%) but latency is terrible
- Adding servers helps, but response is slow (queue already long)

❌ Incorrect Approach #2: Linear Capacity Planning

```
// Incorrect: Assume capacity scales linearly
class CapacityPlanner {
  calculateRequiredServers(expectedQPS: number): number {
    const maxQPSPerServer = 1000;
    const utilizationTarget = 0.8; // 80% utilization

    // Assume we can run at 80% capacity with good latency
    const effectiveCapacity = maxQPSPerServer * utilizationTarget;
    return Math.ceil(expectedQPS / effectiveCapacity);
  }
}

// Plan for 8000 req/s
// Calculation: 8000 / (1000 * 0.8) = 10 servers
```

Why it seems reasonable:

- Standard capacity planning approach
- Leave 20% headroom
- Based on max throughput tests

How it breaks:

- At 80% utilization, latency is already 4-5× base latency
- At 90% utilization during burst, latency is 10×+ base latency
- Utilization target of 80% is way too high for predictable latency
- Linear math ignores queueing effects
- "Headroom" is consumed by queue buildup, not available for burst

Production symptoms:

- SLA violations despite "sufficient" capacity
- p99 latency is 10× higher than p50
- System feels slow even at designed capacity
- Traffic bursts cause disproportionate latency spikes
- Engineers confused: "We're only at 70% CPU!"

❌ Incorrect Approach #3: Setting Timeouts Without Queue Limits

```
// Incorrect: Timeout old requests but don't limit queue
app.use(async (req, res) => {
  const timeout = setTimeout(() => {
    res.status(504).json({ error: 'Timeout' });
  }, 5000);

  try {
    const result = await processRequest(req);
    clearTimeout(timeout);
    res.json(result);
  } catch (err) {
    clearTimeout(timeout);
    res.status(500).json({ error: 'Error' });
  }
});
```

Why it seems reasonable:

- Prevent requests from hanging forever
- User gets response (even if error) within 5s
- Simple to implement

How it breaks:

- Timeout doesn't prevent request from entering queue
- Request waits in queue for 5s, times out, but still uses resources
- Server spent CPU on request that timed out (wasted work)
- Doesn't prevent queue from growing to 1000+ requests
- All 1000 requests timeout, but server still processes them all
- Next batch of requests also times out (queue never drains)

Production symptoms:

- 100% timeout rate during overload
- Server CPU at 100% processing timed-out requests
- Queue length continues growing
- Users see nothing but timeouts
- Server cannot recover (drowning in timed-out work)

❌ Incorrect Approach #4: Vertical Scaling Based on Average Latency

```
// Incorrect: Auto-scale based on average latency
class AutoScaler {
```

```
  async checkAndScale() {
    const avgLatency = await metrics.getAverageLatency();

    if (avgLatency > 100) {
      // Average latency high, add server
      await addServer();
    }
  }
}

setInterval(() => autoScaler.checkAndScale(), 60000);
```

Why it seems reasonable:

- Latency indicates overload
- Automatically add capacity when slow
- Average latency is easy to measure

How it breaks:

- Average latency is lagging indicator (problem already happened)
- By the time average is high, p99 is catastrophic
- Queue buildup happens in seconds, scaling takes minutes
- Average hides distribution (p50 fine, p99 terrible)
- Scaling based on symptoms, not root cause (utilization)

Production symptoms:

- Scaling triggers after users already experienced slowness
- By time new server is ready, traffic spike is over
- System oscillates (scale up, scale down, repeat)
- Users report intermittent slowness
- Autoscaling doesn't prevent latency spikes

3. Correct Mental Model (How It Actually Works)

Queueing theory describes the relationship between arrival rate, service rate, and latency.

Little's Law

```
L = λ × W

L = average number of requests in system (queue + being processed)
λ = arrival rate (requests/second)
W = average time in system (latency)
```

Example:

- Arrival rate: 900 req/s
- Latency: 500ms
- Requests in system: L = 900 × 0.5 = 450 requests

If you measure 450 requests in-flight, and you know arrival rate is 900 req/s, latency MUST be 500ms. This is mathematical law, not a guess.

Utilization and Latency (M/M/1 Queue)

For a single-server queue with random arrivals and service times:

```
ρ = λ / μ  (utilization)

λ = arrival rate
μ = service rate

Average latency: W = 1/(μ - λ) = S/(1 - ρ)

S = service time (time to process one request)
ρ = utilization
```

Critical insight: As ρ approaches 1 (100% utilization), latency approaches infinity.

Latency vs Utilization Curve

```
Utilization | Latency multiplier
------------|-------------------
0%          | 1×   (baseline)
50%         | 2×
70%         | 3.3×
80%         | 5×
90%         | 10×
95%         | 20×
99%         | 100×
```

At 90% utilization, latency is 10× the baseline service time.

Example:

- Service time: 50ms
- At 50% utilization: 50ms × 2 = 100ms average latency
- At 90% utilization: 50ms × 10 = 500ms average latency
- At 95% utilization: 50ms × 20 = 1000ms average latency

Queue Depth Distribution

Queue depth is not constant—it fluctuates:

```
Utilization 50%:
Queue depth: 0 ▓▓▓▓▓▓▓▓▓▓
             1 ▓▓▓▓▓▓
             2 ▓▓▓
             3 ▓

Utilization 90%:
Queue depth: 0-5   ▓▓▓
             5-10  ▓▓▓▓
             10-20 ▓▓▓▓▓
             20-50 ▓▓▓▓
             50+   ▓▓
```

At high utilization, queue depth varies wildly, creating unpredictable latency.

Why the Last 10% is Expensive

```
Capacity: 0%  -------- 50% ------------ 90% --> 100%
Latency:  50ms        100ms            500ms    ∞

Cost to serve last 10% of capacity:
- 50% → 70% capacity: 1.5× latency increase
- 70% → 90% capacity: 3× latency increase
- 90% → 95% capacity: 2× latency increase
```

The relationship is non-linear. Each additional 10% utilization costs exponentially more in latency.

4. Correct Design & Algorithm

Strategy 1: Target Low Utilization

Don't run at 80-90% utilization. Target 50-60% for predictable latency:

```
Utilization target: 50-60%
Latency multiplier: 2-2.5×
Headroom for bursts: 40-50%
```

This seems wasteful but provides:

- Predictable latency (2× instead of 10×)
- Room for traffic bursts
- Tolerance for server failures

Strategy 2: Implement Queue Depth Limits

Reject requests when queue is too deep:

```
if (queueDepth > MAX_QUEUE_DEPTH) {
  reject request with 503 (Service Unavailable)
  return immediately (don't add to queue)
}
```

This prevents:

- Unbounded queue growth
- Wasted work on requests that will timeout
- Complete system collapse

Strategy 3: Monitor Utilization, Not Just Latency

Primary metric: server utilization ($\rho = \lambda / \mu$)

```
if (utilization > 0.6) {
  alert: approaching capacity limit
  action: add servers NOW (before latency degrades)
}
```

Utilization is leading indicator; latency is lagging indicator.

Strategy 4: Use Multiple Workers (M/M/c Queue)

Single server at 90% = 10× latency 10 servers at 90% each = better latency (queue shared across servers)

Multi-server queues have better latency characteristics than single-server queues at same total utilization.

5. Full Production-Grade Implementation

```
import { EventEmitter } from 'events';

interface QueueMetrics {
  queueDepth: number;
  inFlight: number;
  utilization: number;
  arrivalRate: number;
  serviceRate: number;
  avgLatency: number;
  queueWaitTime: number;
}

interface QueueOptions {
  maxQueueDepth: number;
  maxConcurrency: number;
  targetUtilization: number;
  sampleWindowMs: number;
}

class AdaptiveQueue extends EventEmitter {
  private queue: Array<QueuedRequest> = [];
  private inFlight: number = 0;
  private readonly maxQueueDepth: number;
  private readonly maxConcurrency: number;
  private readonly targetUtilization: number;

  // Metrics tracking
  private requestTimestamps: number[] = [];
  private completionTimestamps: number[] = [];
  private latencies: number[] = [];
  private readonly sampleWindowMs: number;

  constructor(options: QueueOptions) {
    super();
    this.maxQueueDepth = options.maxQueueDepth;
    this.maxConcurrency = options.maxConcurrency;
    this.targetUtilization = options.targetUtilization;
    this.sampleWindowMs = options.sampleWindowMs;

    // Emit metrics periodically
    setInterval(() => this.emitMetrics(), 1000);
```

```
}

/**
 * Add request to queue with admission control.
 * Returns Promise that resolves when request is processed.
 */
async enqueue<T>(
  workFn: () => Promise<T>,
  priority: number = 0
): Promise<T> {
  // Admission control: reject if queue too deep
  if (this.queue.length >= this.maxQueueDepth) {
    throw new Error('Queue full - request rejected (503)');
  }

  // Admission control: reject if utilization too high
  const metrics = this.getMetrics();
  if (metrics.utilization > 0.95) {
    throw new Error('System overloaded - request rejected (503)');
  }

  const arrivalTime = Date.now();
  this.recordArrival();

  return new Promise<T>((resolve, reject) => {
    const request: QueuedRequest = {
      workFn,
      priority,
      arrivalTime,
      resolve,
      reject,
    };

    this.queue.push(request);
    this.processQueue();
  });
}

/**
 * Process queued requests up to concurrency limit
 */
private async processQueue(): Promise<void> {
  while (this.inFlight < this.maxConcurrency && this.queue.length > 0) {
    // Sort by priority (higher priority first)
    this.queue.sort((a, b) => b.priority - a.priority);

    const request = this.queue.shift()!;
    this.inFlight++;

    const queueWaitTime = Date.now() - request.arrivalTime;

    // Process request
```

```
    this.processRequest(request, queueWaitTime).finally(() => {
      this.inFlight--;
      this.processQueue(); // Process next request
    });
  }
}

/**
 * Execute single request and track metrics
 */
private async processRequest(
  request: QueuedRequest,
  queueWaitTime: number
): Promise<void> {
  const startTime = Date.now();

  try {
    const result = await request.workFn();
    const totalLatency = Date.now() - request.arrivalTime;
    const serviceTime = Date.now() - startTime;

    this.recordCompletion(totalLatency);
    request.resolve(result);

    // Emit detailed metrics for monitoring
    this.emit('request-complete', {
      queueWaitTime,
      serviceTime,
      totalLatency,
      queueDepth: this.queue.length,
    });
  } catch (error) {
    request.reject(error);
    this.emit('request-error', error);
  }
}

/**
 * Record request arrival for rate calculation
 */
private recordArrival(): void {
  const now = Date.now();
  this.requestTimestamps.push(now);
  this.cleanOldSamples(this.requestTimestamps, now);
}

/**
 * Record request completion for rate and latency calculation
 */
private recordCompletion(latency: number): void {
  const now = Date.now();
  this.completionTimestamps.push(now);
```

```
    this.latencies.push(latency);
    this.cleanOldSamples(this.completionTimestamps, now);
    this.cleanOldSamples(this.latencies, now);
  }

  /**
   * Remove samples outside time window
   */
  private cleanOldSamples(samples: number[], now: number): void {
    const cutoff = now - this.sampleWindowMs;
    while (samples.length > 0 && samples[0] < cutoff) {
      samples.shift();
    }
  }

  /**
   * Calculate current system metrics
   */
  getMetrics(): QueueMetrics {
    const arrivalRate = this.requestTimestamps.length / (this.sampleWindowMs /
1000);
    const serviceRate = this.completionTimestamps.length / (this.sampleWindowMs /
1000);

    // Utilization: ρ = λ / μ (arrival rate / service rate)
    const utilization = serviceRate > 0 ? arrivalRate / (serviceRate *
this.maxConcurrency) : 0;

    // Average latency from recent samples
    const avgLatency = this.latencies.length > 0
      ? this.latencies.reduce((a, b) => a + b, 0) / this.latencies.length
      : 0;

    // Queue wait time using Little's Law: W = L / λ
    const queueWaitTime = arrivalRate > 0 ? this.queue.length / arrivalRate : 0;

    return {
      queueDepth: this.queue.length,
      inFlight: this.inFlight,
      utilization: Math.min(utilization, 1.0),
      arrivalRate,
      serviceRate,
      avgLatency,
      queueWaitTime: queueWaitTime * 1000, // Convert to ms
    };
  }

  /**
   * Get health status based on queueing theory
   */
  getHealthStatus(): {
    status: 'healthy' | 'warning' | 'critical';
```

```
    reason: string;
  } {
    const metrics = this.getMetrics();

    if (metrics.utilization > 0.9) {
      return {
        status: 'critical',
        reason: `Utilization ${(metrics.utilization * 100).toFixed(1)}% - latency
will be 10×+ baseline`,
      };
    }

    if (metrics.utilization > this.targetUtilization) {
      return {
        status: 'warning',
        reason: `Utilization ${(metrics.utilization * 100).toFixed(1)}% exceeds
target ${(this.targetUtilization * 100).toFixed(1)}%`,
      };
    }

    if (this.queue.length > this.maxQueueDepth * 0.5) {
      return {
        status: 'warning',
        reason: `Queue depth ${this.queue.length} approaching limit
${this.maxQueueDepth}`,
      };
    }

    return {
      status: 'healthy',
      reason: `Utilization ${(metrics.utilization * 100).toFixed(1)}%, queue depth
${this.queue.length}`,
    };
  }

  /**
   * Predict latency at given utilization using M/M/c formula
   */
  predictLatency(targetUtilization: number): number {
    const metrics = this.getMetrics();
    const baseServiceTime = metrics.serviceRate > 0
      ? 1 / metrics.serviceRate
      : 0.05; // Default 50ms

    // Simplified M/M/1 formula: W = S / (1 - ρ)
    return baseServiceTime / (1 - targetUtilization);
  }

  private emitMetrics(): void {
    this.emit('metrics', this.getMetrics());
  }
}
```

```typescript
interface QueuedRequest {
  workFn: () => Promise<any>;
  priority: number;
  arrivalTime: number;
  resolve: (value: any) => void;
  reject: (error: Error) => void;
}

// Example usage in Express server
const queue = new AdaptiveQueue({
  maxQueueDepth: 100,
  maxConcurrency: 50,
  targetUtilization: 0.6,
  sampleWindowMs: 10000,
});

// Monitor queue health
queue.on('metrics', (metrics: QueueMetrics) => {
  console.log('[Queue Metrics]', {
    utilization: `${(metrics.utilization * 100).toFixed(1)}%`,
    queueDepth: metrics.queueDepth,
    avgLatency: `${metrics.avgLatency.toFixed(0)}ms`,
    arrivalRate: `${metrics.arrivalRate.toFixed(1)} req/s`,
  });

  // Alert if approaching capacity
  const health = queue.getHealthStatus();
  if (health.status !== 'healthy') {
    console.warn(`[Queue Health] ${health.status}: ${health.reason}`);
  }
});

// Middleware to use adaptive queue
app.use(async (req, res, next) => {
  try {
    await queue.enqueue(async () => {
      // Process request
      await processRequest(req, res);
    });
  } catch (error) {
    if (error.message.includes('Queue full') ||
error.message.includes('overloaded')) {
      res.status(503).json({
        error: 'Service temporarily unavailable',
        retryAfter: 5,
      });
    } else {
      next(error);
    }
  }
});
```

```
// Metrics endpoint
app.get('/metrics/queue', (req, res) => {
  const metrics = queue.getMetrics();
  const health = queue.getHealthStatus();

  res.json({
    ...metrics,
    health: health.status,
    healthReason: health.reason,
    predictedLatencyAt70Pct: queue.predictLatency(0.7),
    predictedLatencyAt90Pct: queue.predictLatency(0.9),
  });
});

// Capacity planning helper
function calculateRequiredCapacity(
  expectedQPS: number,
  serviceTimeMs: number,
  targetUtilization: number = 0.6
): number {
  // Each server can handle: μ requests/sec at target utilization
  const serviceRate = 1000 / serviceTimeMs; // Convert ms to req/s
  const effectiveCapacity = serviceRate * targetUtilization;

  return Math.ceil(expectedQPS / effectiveCapacity);
}

console.log('Capacity planning:');
console.log(`Expected load: 10,000 req/s`);
console.log(`Service time: 50ms`);
console.log(`Target utilization: 60%`);
console.log(`Required servers: ${calculateRequiredCapacity(10000, 50, 0.6)}`);
// Output: Required servers: 84
// Each server: 20 req/s capacity × 0.6 = 12 req/s effective
// Total: 84 servers × 12 = 1008 req/s per server
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: API Gateway with Admission Control

```
const apiQueue = new AdaptiveQueue({
  maxQueueDepth: 200,
  maxConcurrency: 100,
  targetUtilization: 0.6,
  sampleWindowMs: 10000,
});

app.post('/api/:endpoint', async (req, res) => {
  try {
```

```
    const result = await apiQueue.enqueue(async () => {
      return await processAPIRequest(req);
    });
    res.json(result);
  } catch (error) {
    if (error.message.includes('Queue full')) {
      res.status(503).json({ error: 'System overloaded, try again' });
    } else {
      res.status(500).json({ error: 'Internal error' });
    }
  }
});
```

Why this works:

- Rejects requests when queue too deep (prevents collapse)
- Monitors utilization (leading indicator)
- Provides backpressure to clients
- Prevents wasted work on timed-out requests

Pattern 2: Background Job Processor with Priority

```
const jobQueue = new AdaptiveQueue({
  maxQueueDepth: 1000,
  maxConcurrency: 20,
  targetUtilization: 0.7,
  sampleWindowMs: 30000,
});

async function enqueueJob(job: Job, priority: 'high' | 'normal' | 'low') {
  const priorityMap = { high: 10, normal: 5, low: 1 };

  await jobQueue.enqueue(
    () => processJob(job),
    priorityMap[priority]
  );
}
```

Why this works:

- High-priority jobs processed first
- Queue depth limited (prevents memory exhaustion)
- Utilization monitoring for capacity planning
- Can tolerate higher utilization (70%) for batch processing

Pattern 3: Database Connection Pool

```
const dbQueue = new AdaptiveQueue({
  maxQueueDepth: 500,
  maxConcurrency: 50, // Pool size
  targetUtilization: 0.5,
```

```
  sampleWindowMs: 5000,
});

async function query(sql: string, params: any[]): Promise<any> {
  return dbQueue.enqueue(async () => {
    const conn = await pool.getConnection();
    try {
      return await conn.query(sql, params);
    } finally {
      conn.release();
    }
  });
}
```

Why this works:

- Limits concurrent database queries
- Prevents connection pool exhaustion
- Queue provides buffering for burst traffic
- Low utilization target (50%) for predictable latency

7. Failure Modes & Edge Cases

Convoy Effect

Problem: One slow request blocks queue, subsequent requests wait.

Symptoms:

- All requests slow, even though most work is fast
- Queue depth grows even at low utilization
- One database query taking 30s blocks everything

Mitigation:

- Timeout individual requests
- Use separate queues for different request types
- Implement request prioritization

Self-Inflicted Retry Storm

Problem: Clients retry on 503, increasing load further.

Symptoms:

- Queue rejection rate 50%
- Actual arrival rate 2× expected (retries)
- Utilization increases due to retries
- System cannot recover

Mitigation:

- Return `Retry-After` header
- Clients implement exponential backoff
- Circuit breaker on client side

Bursty Traffic

Problem: Traffic arrives in bursts, not steady rate.

Symptoms:

- Utilization averages 50% but queue depth varies wildly
- Periodic queue depth spikes
- Latency spikes despite low average utilization

Mitigation:

- Use queue depth limits
- Track p99 queue depth, not average
- Size for burst capacity (lower utilization target)

Memory Exhaustion from Queue

Problem: Queue grows to thousands of requests, exhausts memory.

Symptoms:

- Memory usage grows to GB
- OOM killer terminates process
- Queue depth in thousands

Mitigation:

- Set strict maxQueueDepth
- Monitor queue memory usage
- Reject early rather than queue indefinitely

8. Performance Characteristics & Tradeoffs

Utilization vs Latency

Utilization	Latency Multiplier	Queue Depth (avg)
30%	1.4×	0.4
50%	2×	1
70%	3.3×	2.3
80%	5×	4
90%	10×	9
95%	20×	19
99%	100×	99

Cost of Low Utilization

Running at 50% utilization seems wasteful:

- 50% of resources "unused"
- 2× more servers than running at 100%

But provides:

- Predictable latency (2× instead of ∞)
- Room for traffic bursts
- Tolerance for server failures (can lose 25% of servers)

ROI: 2× infrastructure cost → 10× better tail latency → higher conversion rate

Queue Depth as Leading Indicator

Queue depth indicates impending latency problems:

```
Queue depth < 5: Healthy
Queue depth 5-20: Warning (approaching capacity)
Queue depth 20+: Critical (latency already degraded)
```

Monitor queue depth to predict problems before latency degrades.

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

Mistake 1: Running at 80%+ Utilization

Why engineers do it: "We have headroom, why waste resources?"

What breaks: Latency becomes 5-10× baseline, unpredictable.

Detection: p99 latency is 10× p50, user complaints.

Fix: Target 50-60% utilization for user-facing services.

Mistake 2: Not Rejecting Requests Early

Why engineers do it: "We should try to process every request."

What breaks: Queue grows to thousands, system collapses completely.

Detection: Memory exhaustion, 100% timeout rate.

Fix: Set maxQueueDepth, return 503 when exceeded.

```
if (queue.length > maxQueueDepth) {
  return 503;  // Reject immediately
}
```

Mistake 3: Monitoring Average Latency Only

Why engineers do it: Average is easy to calculate.

What breaks: Average looks fine (100ms) but p99 is terrible (10s).

Detection: Users complain but metrics look okay.

Fix: Monitor p95, p99, p99.9 latency and queue depth.

Mistake 4: Auto-Scaling Based on CPU

Why engineers do it: CPU indicates load.

What breaks: At high utilization, queue grows before CPU maxes out.

Detection: CPU at 70%, latency at 10×.

Fix: Scale based on utilization (arrival rate / service rate), not CPU.

Mistake 5: Believing "We Have 30% Headroom"

Why engineers do it: Running at 70% = 30% unused capacity.

What breaks: That 30% is consumed by queueing effects, not available.

Detection: Traffic increases 20%, latency increases 300%.

Fix: Understand that usable headroom is much less than arithmetic headroom.

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

Anti-Pattern 1: Low-Traffic Services

Don't add queueing complexity for services with <100 req/s:

```
// Unnecessary for low traffic
const queue = new AdaptiveQueue({ maxConcurrency: 50 });
// Service only gets 10 req/s, no queueing needed
```

Anti-Pattern 2: Batch Processing

Batch jobs don't need bounded queues:

```
// WRONG for batch processing
queue.enqueue(() => processBatchJob());
// Batch jobs should use work queues (Kafka, SQS), not admission control
```

Anti-Pattern 3: WebSocket/Streaming

Don't use request queue for long-lived connections:

```
// WRONG
await queue.enqueue(() => handleWebSocket(conn));
// WebSocket holds connection for minutes/hours, not milliseconds
```

11. Related Concepts (With Contrast)

Rate Limiting

Difference: Rate limiting restricts requests per time window. Queueing manages concurrency.

When to combine: Use both. Rate limit to prevent abuse, queue to manage load.

Circuit Breaker

Difference: Circuit breaker stops calling failing services. Queueing prevents overloading working services.

When to combine: Use both. Circuit breaker for failures, queue for capacity.

Load Balancing

Difference: Load balancing distributes requests across servers. Queueing manages requests within a server.

When to combine: Use both. Load balancer fans out to servers, each server has queue.

Backpressure

Difference: Backpressure is the mechanism (reject requests). Queueing theory explains when/why to apply it.

Relationship: Queueing theory informs backpressure policy (reject at 60% utilization).

12. Production Readiness Checklist

Metrics to Monitor

- [ ] Queue depth (current, p50, p95, p99)
- [ ] Utilization (arrival rate / service rate)
- [ ] Arrival rate (requests/second)
- [ ] Service rate (completions/second)
- [ ] Queue wait time (time in queue before processing)
- [ ] Total latency (queue wait + service time)
- [ ] Rejection rate (503 responses)

Alerting

- [ ] Alert if utilization > 70%
- [ ] Alert if queue depth > 50
- [ ] Alert if p99 latency > 5× p50
- [ ] Alert if rejection rate > 1%

Configuration

- [ ] Set maxQueueDepth based on memory limits
- [ ] Set maxConcurrency based on resource limits (connections, threads)
- [ ] Set targetUtilization to 0.5-0.6 for user-facing services
- [ ] Set sampleWindowMs to 10-30 seconds

Load Testing

- [ ] Test at 50%, 70%, 90% utilization
- [ ] Measure latency at each level
- [ ] Verify latency matches queueing theory predictions
- [ ] Test rejection behavior (maxQueueDepth exceeded)
- [ ] Test burst traffic (sudden 2× load increase)

Capacity Planning

- [ ] Calculate servers needed for expected load at target utilization
- [ ] Account for traffic growth (plan for 2× current load)
- [ ] Account for server failures (N+2 redundancy)

- [ ] Document: "At 10k req/s, need 84 servers at 60% utilization"

Rollout

- [ ] Deploy queue with high limits (low rejection risk)
- [ ] Monitor metrics for 1 week
- [ ] Tune maxQueueDepth and targetUtilization
- [ ] Enable admission control (rejection)
- [ ] Document observed utilization vs latency relationship

Documentation

- [ ] Document target utilization and reasoning
- [ ] Document capacity calculation formula
- [ ] Create runbook for overload situations
- [ ] Define SLAs based on realistic utilization (don't promise p99 < 100ms at 90% utilization)