Load Shedding

1. The Real Problem This Exists to Solve

When a system is overwhelmed and cannot serve all incoming requests, it must decide which requests to drop and which to serve. Without intelligent load shedding, systems waste resources processing low-value requests while critical operations fail.

Real production scenario:

- E-commerce site during Black Friday sale
- 100,000 requests/second hitting checkout API
- System capacity: 10,000 requests/second
- Without load shedding:
  - Server attempts to process all 100k req/s
  - Queue depth grows to millions
  - Memory exhausted, system crashes
  - ALL users (including paying customers) experience failures
  - Revenue impact: $100,000/minute in lost sales
- With load shedding:
  - Priority 1: Checkout/payment requests (5k req/s) - **always served**
  - Priority 2: Add to cart (3k req/s) - served
  - Priority 3: Product browse (2k req/s) - served
  - Priority 4: Search (shed completely) - temporarily disabled
  - Priority 5: Recommendations (shed completely)
  - System serves 10k req/s within capacity
  - Paying customers complete checkout successfully
  - Browse/search users see "temporarily unavailable" message
  - Revenue preserved: $95,000/minute (vs $0 in crash scenario)

The fundamental problem: During overload, serving all requests equally means serving no requests well. Strategic shedding of low-priority traffic preserves service for critical operations.

Without load shedding:

- All requests treated equally
- System attempts to serve everything
- Queue grows unbounded
- Latency increases for all requests
- Eventually crashes, serves nothing
- Critical and non-critical requests both fail

With load shedding:

- Requests prioritized by business value
- Low-priority traffic dropped immediately (no resource consumption)
- Critical traffic served within SLA
- System stays operational
- Graceful degradation of features

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Random Request Dropping

```
// Incorrect: Drop requests randomly
class APIServer {
  async handleRequest(req: Request): Promise<Response> {
    const systemLoad = this.getSystemLoad();

    if (systemLoad > 0.8) {
      // Drop 50% of requests randomly
      if (Math.random() < 0.5) {
        throw new Error('Service overloaded');
      }
    }

    return await this.processRequest(req);
  }
}
```

Why it seems reasonable:

- Simple to implement
- Reduces incoming load by 50%
- No complex priority logic needed

How it breaks:

- Drops critical requests (checkout, payment) same as non-critical (recommendations)
- Paying customers experience failures
- Low-value requests consume resources while high-value requests fail
- Revenue impact same as serving nothing
- No way to protect business-critical operations

Production symptoms:

- 50% of checkout requests fail (lost revenue)
- 50% of recommendation requests fail (acceptable)
- Same failure rate across all endpoints
- Customers complain: "Couldn't complete purchase during sale"
- Millions in lost revenue
- Metrics show: high-value endpoints same error rate as low-value

❌ Incorrect Approach #2: Rate Limiting Without Priority

```
// Incorrect: Apply same rate limit to all requests
const limiter = rateLimit({
  windowMs: 1000,
  max: 10000, // 10k req/s total
});

app.use(limiter);
```

```
app.post('/checkout', handleCheckout);
app.get('/recommendations', handleRecommendations);
```

Why it seems reasonable:

- Caps incoming load at system capacity
- Prevents overload
- Standard rate limiting pattern

How it breaks:

- First 10k requests served regardless of priority
- If 9k low-priority requests arrive first, only 1k high-priority requests served
- No differentiation between critical and non-critical
- Recommendation requests can consume all quota
- Checkout requests rejected despite being critical

Production symptoms:

- Checkout API returns 429 (rate limited) during peak
- Recommendation API serving successfully (consumed quota first)
- Users can browse products but can't checkout
- Shopping carts abandoned (can't complete purchase)
- Revenue lost despite system not being overloaded

❌ Incorrect Approach #3: CPU-Based Shedding

```
// Incorrect: Shed load based only on CPU usage
import os from 'os';

class APIServer {
  async handleRequest(req: Request): Promise<Response> {
    const cpuUsage = os.loadavg()[0] / os.cpus().length;

    if (cpuUsage > 0.8) {
      throw new Error('High CPU, shedding load');
    }

    return await this.processRequest(req);
  }
}
```

Why it seems reasonable:

- CPU usage is measurable indicator of overload
- Protects system from CPU exhaustion
- Simple threshold-based logic

How it breaks:

- High CPU could be from batch processing (acceptable)
- Low CPU could hide memory or I/O bottlenecks
- Doesn't consider request priority
- All requests shed equally when CPU high

- May shed load when system not actually overloaded for request traffic

Production symptoms:

- Load shedding triggers during scheduled report generation (high CPU but not from user traffic)
- All user requests rejected during batch job
- System has plenty of memory/network capacity but shedding due to CPU
- Critical requests rejected unnecessarily

❌ Incorrect Approach #4: Queue-Based Shedding Without Priorities

```
// Incorrect: Drop requests when queue full, no priority
class RequestQueue {
  private queue: Request[] = [];
  private readonly MAX_QUEUE = 10000;

  enqueue(req: Request): void {
    if (this.queue.length >= this.MAX_QUEUE) {
      throw new Error('Queue full');
    }
    this.queue.push(req);
  }

  dequeue(): Request | undefined {
    return this.queue.shift();
  }
}
```

Why it seems reasonable:

- Bounds queue depth (prevents memory exhaustion)
- Simple FIFO queue
- Clear capacity limit

How it breaks:

- When queue full, new requests (potentially high-priority) rejected
- Old requests (potentially low-priority) still being processed
- No way to preempt low-priority work for high-priority
- Checkout request arriving at full queue rejected while recommendation requests processed

Production symptoms:

- Queue fills with low-priority requests during spike
- High-priority requests arriving later get rejected (queue full)
- System processing old, low-value requests
- New critical requests can't get in
- FIFO processing during overload hurts most important requests

3. Correct Mental Model (How It Actually Works)

Load shedding is the strategic dropping of low-priority requests to preserve resources for high-priority requests during overload.

The Priority Hierarchy

```
Priority 1 (Critical): Never shed
  - Checkout, payment processing
  - Account login, authentication
  - System health checks

Priority 2 (High): Shed last
  - Add to cart
  - View order history
  - Customer support tickets

Priority 3 (Medium): Shed early
  - Product browse
  - Search queries
  - Inventory checks

Priority 4 (Low): Shed first
  - Recommendations
  - Related products
  - Analytics tracking
  - A/B test logging
```

The Decision Flow

```
Request arrives
  ↓
Extract priority (header, endpoint, user tier)
  ↓
Check system load
  ↓
If load < 60%: Serve all requests
If load 60-80%: Shed priority 4 (low)
If load 80-90%: Shed priority 3-4 (medium + low)
If load 90-95%: Shed priority 2-4 (high + medium + low)
If load > 95%: Only serve priority 1 (critical)
```

Resource Protection Model

```
Total capacity: 10,000 req/s

Reserved capacity allocation:
  Priority 1: 5,000 req/s (50%) - always available
  Priority 2: 3,000 req/s (30%) - available if not used by P1
  Priority 3: 1,500 req/s (15%) - available if not used by P1/P2
  Priority 4: 500 req/s (5%) - available if not used by P1/P2/P3

During overload (15k req/s incoming):
  Priority 1: 6k req/s → serve 6k (within reserved + borrowed)
  Priority 2: 4k req/s → serve 3k (remaining capacity)
```

```
  Priority 3: 3k req/s → serve 1k
  Priority 4: 2k req/s → shed all (no capacity)

Result: Critical operations protected, graceful degradation
```

4. Correct Design & Algorithm

Strategy 1: Priority-Based Admission Control

```
priority = extract_priority(request)
current_load = measure_load()
threshold = get_threshold(priority)

if current_load > threshold:
  return 503 Service Unavailable
else:
  process request
```

Strategy 2: Token Bucket Per Priority

```
for each priority level:
  tokens[priority] = capacity[priority]

on request with priority:
  if tokens[priority] > 0:
    tokens[priority]--
    process request
  else:
    return 503
```

Strategy 3: Weighted Fair Queueing

```
Multiple queues (one per priority)
Serve from queues with weights:
  Priority 1: weight 50 (serve 50% of time)
  Priority 2: weight 30
  Priority 3: weight 15
  Priority 4: weight 5
```

5. Full Production-Grade Implementation

```typescript
interface LoadSheddingConfig {
  priorities: {
    [key: string]: {
      level: number;  // 1 (critical) to 4 (low)
      reservedCapacity: number;  // req/s
      shedThreshold: number;  // load percentage to start shedding
    };
  };
```

```
  loadMetric: 'queue_depth' | 'latency' | 'error_rate';
  loadThresholds: {
    low: number;
    medium: number;
    high: number;
    critical: number;
  };
}

interface LoadMetrics {
  queueDepth: number;
  p99Latency: number;
  errorRate: number;
  requestRate: number;
  capacityUtilization: number;
}

class LoadShedder {
  private config: LoadSheddingConfig;
  private metrics: LoadMetrics;
  private tokenBuckets: Map<number, TokenBucket>;
  private shedCount: Map<number, number> = new Map();
  private servedCount: Map<number, number> = new Map();

  constructor(config: LoadSheddingConfig) {
    this.config = config;
    this.metrics = this.initMetrics();
    this.tokenBuckets = this.initTokenBuckets();

    // Update metrics every second
    setInterval(() => this.updateMetrics(), 1000);
  }

  /**
   * Determine if request should be admitted or shed.
   * Returns undefined if admitted, error if shed.
   */
  async admit(
    priority: number,
    requestType: string
  ): Promise<SheddingDecision> {
    const currentLoad = this.getCurrentLoad();
    const priorityConfig = this.getPriorityConfig(priority);

    // Update counters
    this.incrementCounter(priority, 'total');

    // Critical priority (1) never shed
    if (priority === 1) {
      this.incrementCounter(priority, 'served');
      return { admitted: true };
    }
```

```
    // Check if this priority should be shed based on load
    if (currentLoad > priorityConfig.shedThreshold) {
      this.incrementCounter(priority, 'shed');
      return {
        admitted: false,
        reason: `Load ${currentLoad.toFixed(2)} > threshold
${priorityConfig.shedThreshold}`,
        priority,
        requestType,
      };
    }

    // Check token bucket for rate limiting per priority
    const bucket = this.tokenBuckets.get(priority);
    if (bucket && !bucket.tryConsume(1)) {
      this.incrementCounter(priority, 'shed');
      return {
        admitted: false,
        reason: `Priority ${priority} rate limit exceeded`,
        priority,
        requestType,
      };
    }

    this.incrementCounter(priority, 'served');
    return { admitted: true };
  }

  /**
   * Get current system load (0.0 to 1.0)
   */
  private getCurrentLoad(): number {
    switch (this.config.loadMetric) {
      case 'queue_depth':
        return this.metrics.queueDepth / 10000;  // Normalize to 0-1
      case 'latency':
        return Math.min(this.metrics.p99Latency / 5000, 1.0);  // 5s = 100%
      case 'error_rate':
        return this.metrics.errorRate;
      default:
        return this.metrics.capacityUtilization;
    }
  }

  /**
   * Get priority configuration
   */
  private getPriorityConfig(priority: number): {
    shedThreshold: number;
  } {
    // Default thresholds per priority
```

```
    const defaults = {
      1: { shedThreshold: 1.0 },   // Never shed (100% load)
      2: { shedThreshold: 0.95 },  // Shed only at 95%+ load
      3: { shedThreshold: 0.80 },  // Shed at 80%+ load
      4: { shedThreshold: 0.60 },  // Shed at 60%+ load
    };

    return defaults[priority as keyof typeof defaults] || defaults[4];
  }

  /**
   * Initialize token buckets for rate limiting per priority
   */
  private initTokenBuckets(): Map<number, TokenBucket> {
    const buckets = new Map<number, TokenBucket>();

    for (const [_, config] of Object.entries(this.config.priorities)) {
      buckets.set(
        config.level,
        new TokenBucket(config.reservedCapacity, config.reservedCapacity)
      );
    }

    return buckets;
  }

  /**
   * Initialize metrics
   */
  private initMetrics(): LoadMetrics {
    return {
      queueDepth: 0,
      p99Latency: 0,
      errorRate: 0,
      requestRate: 0,
      capacityUtilization: 0,
    };
  }

  /**
   * Update metrics (called every second)
   */
  private updateMetrics(): void {
    // Implementation would pull from actual monitoring system
    // Placeholder for example
  }

  /**
   * Increment counter
   */
  private incrementCounter(priority: number, type: 'total' | 'served' | 'shed'):
void {
```

```
    if (type === 'served') {
      this.servedCount.set(priority, (this.servedCount.get(priority) || 0) + 1);
    } else if (type === 'shed') {
      this.shedCount.set(priority, (this.shedCount.get(priority) || 0) + 1);
    }
  }

  /**
   * Get shedding statistics
   */
  getStats(): SheddingStats {
    const stats: SheddingStats = {
      currentLoad: this.getCurrentLoad(),
      byPriority: {},
    };

    for (let priority = 1; priority <= 4; priority++) {
      const served = this.servedCount.get(priority) || 0;
      const shed = this.shedCount.get(priority) || 0;
      const total = served + shed;

      stats.byPriority[priority] = {
        served,
        shed,
        total,
        shedRate: total > 0 ? shed / total : 0,
      };
    }

    return stats;
  }

  /**
   * Reset statistics
   */
  resetStats(): void {
    this.shedCount.clear();
    this.servedCount.clear();
  }
}

interface SheddingDecision {
  admitted: boolean;
  reason?: string;
  priority?: number;
  requestType?: string;
}

interface SheddingStats {
  currentLoad: number;
  byPriority: {
    [priority: number]: {
```

```
      served: number;
      shed: number;
      total: number;
      shedRate: number;
    };
  };
}

class TokenBucket {
  private tokens: number;
  private lastRefill: number;

  constructor(
    private capacity: number,
    private refillRate: number  // tokens per second
  ) {
    this.tokens = capacity;
    this.lastRefill = Date.now();
  }

  tryConsume(tokens: number): boolean {
    this.refill();

    if (this.tokens >= tokens) {
      this.tokens -= tokens;
      return true;
    }

    return false;
  }

  private refill(): void {
    const now = Date.now();
    const elapsed = (now - this.lastRefill) / 1000;
    const tokensToAdd = elapsed * this.refillRate;

    this.tokens = Math.min(this.capacity, this.tokens + tokensToAdd);
    this.lastRefill = now;
  }
}

// Express middleware for load shedding
const shedder = new LoadShedder({
  priorities: {
    checkout: { level: 1, reservedCapacity: 5000, shedThreshold: 1.0 },
    addToCart: { level: 2, reservedCapacity: 3000, shedThreshold: 0.95 },
    browse: { level: 3, reservedCapacity: 1500, shedThreshold: 0.80 },
    recommendations: { level: 4, reservedCapacity: 500, shedThreshold: 0.60 },
  },
  loadMetric: 'queue_depth',
  loadThresholds: {
    low: 0.5,
```

```
    medium: 0.7,
    high: 0.85,
    critical: 0.95,
  },
});

// Middleware to extract priority from request
function extractPriority(req: express.Request): number {
  // Method 1: From header
  const headerPriority = req.header('X-Priority');
  if (headerPriority) return parseInt(headerPriority);

  // Method 2: From endpoint
  if (req.path.includes('/checkout')) return 1;
  if (req.path.includes('/cart')) return 2;
  if (req.path.includes('/browse')) return 3;
  if (req.path.includes('/recommendations')) return 4;

  // Method 3: From user tier (premium users = higher priority)
  const userTier = req.user?.tier;
  if (userTier === 'premium') return 2;

  // Default: medium priority
  return 3;
}

app.use(async (req, res, next) => {
  const priority = extractPriority(req);
  const requestType = req.path;

  const decision = await shedder.admit(priority, requestType);

  if (!decision.admitted) {
    res.status(503)
       .header('X-Load-Shed', 'true')
       .header('X-Priority', priority.toString())
       .header('Retry-After', '10')
       .json({
         error: 'Service temporarily unavailable',
         reason: decision.reason,
         priority,
         retryAfter: 10,
       });
    return;
  }

  // Add priority to request for downstream use
  (req as any).priority = priority;
  next();
});

// Critical endpoint (priority 1)
```

```
app.post('/checkout', async (req, res) => {
  const order = await processCheckout(req.body);
  res.json({ orderId: order.id });
});

// Medium priority endpoint (priority 3)
app.get('/browse', async (req, res) => {
  const products = await getProducts();
  res.json(products);
});

// Low priority endpoint (priority 4)
app.get('/recommendations', async (req, res) => {
  const recs = await getRecommendations(req.user.id);
  res.json(recs);
});

// Stats endpoint
app.get('/stats/load-shedding', (req, res) => {
  const stats = shedder.getStats();
  res.json({
    currentLoad: (stats.currentLoad * 100).toFixed(1) + '%',
    priorities: Object.entries(stats.byPriority).map(([priority, data]) => ({
      priority: parseInt(priority),
      served: data.served,
      shed: data.shed,
      shedRate: (data.shedRate * 100).toFixed(2) + '%',
    })),
  });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: E-Commerce During Sales

```
// Priority 1: Revenue-generating operations
app.post('/api/checkout', priority(1), handleCheckout);
app.post('/api/payment', priority(1), handlePayment);

// Priority 2: User engagement
app.post('/api/cart', priority(2), handleAddToCart);
app.get('/api/orders/:id', priority(2), handleGetOrder);

// Priority 3: Browsing
app.get('/api/products', priority(3), handleBrowseProducts);

// Priority 4: Non-essential
app.get('/api/recommendations', priority(4), handleRecommendations);
app.post('/api/analytics', priority(4), handleAnalytics);
```

During Black Friday spike:

- System at 150% capacity
- Priority 4 shed completely (recommendations, analytics)
- Priority 3 shed 70% (some browsing works)
- Priority 2 shed 20% (most cart operations work)
- Priority 1 never shed (all checkouts succeed)
- Revenue preserved

Pattern 2: User Tier-Based Shedding

```
function getPriorityForUser(user: User): number {
  if (user.tier === 'enterprise') return 1;
  if (user.tier === 'premium') return 2;
  if (user.tier === 'free') return 3;
  return 4;  // Anonymous
}

app.use((req, res, next) => {
  const priority = getPriorityForUser(req.user);
  req.priority = priority;
  next();
});
```

During overload:

- Enterprise customers always served
- Premium customers mostly served
- Free users degraded experience
- Anonymous users shed first
- Protects paying customers

Pattern 3: Feature Flags for Dynamic Shedding

```
const featureFlags = {
  recommendations: true,
  relatedProducts: true,
  userReviews: true,
};

app.get('/api/product/:id', async (req, res) => {
  const product = await getProduct(req.params.id);

  // Conditionally include non-critical data
  const response = { ...product };

  if (featureFlags.recommendations && req.priority <= 3) {
    response.recommendations = await getRecommendations(product.id);
  }

  if (featureFlags.relatedProducts && req.priority <= 2) {
```

```
    response.relatedProducts = await getRelatedProducts(product.id);
  }

  res.json(response);
});

// During incident: disable non-critical features
app.post('/admin/shed-load', (req, res) => {
  featureFlags.recommendations = false;
  featureFlags.relatedProducts = false;
  res.json({ message: 'Load shedding activated' });
});
```

During overload:

- Disable recommendations (saves DB queries)
- Disable related products
- Core product data still served
- Response time improves
- System stays up

7. Failure Modes & Edge Cases

Priority Inversion

Problem: High-priority request depends on low-priority service that's being shed.

Symptoms:

- Checkout endpoint (priority 1) calls inventory service (priority 3)
- Inventory service shedding requests due to overload
- Checkout fails despite being priority 1

Mitigation:

- Propagate priority downstream (include X-Priority header)
- Dependencies respect caller priority
- Critical services never fully shed

Starvation of Low-Priority Traffic

Problem: Low-priority requests never served during sustained high load.

Symptoms:

- Recommendations endpoint (priority 4) shed for hours
- Users never see recommendations
- A/B tests can't collect data

Mitigation:

- Guarantee minimum bandwidth per priority (e.g., 5%)
- Time-based admission (allow 1% of low-priority every minute)
- Graceful degradation messaging

Incorrect Priority Assignment

Problem: Critical operation accidentally assigned low priority.

Symptoms:

- Password reset endpoint marked priority 4
- Users can't reset passwords during overload
- Support tickets flood in

Mitigation:

- Code review for priority assignments
- Automated tests for critical endpoints
- Monitor shedding rate per endpoint

Load Spike from Single Priority

Problem: All traffic is priority 1, shedding doesn't help.

Symptoms:

- DDoS attack on checkout endpoint
- All requests marked priority 1
- No low-priority traffic to shed

Mitigation:

- Rate limiting per priority
- Authentication required for critical endpoints
- DDoS protection layer

8. Performance Characteristics & Tradeoffs

Throughput Preservation

Without shedding (during 200% overload):

- Attempt to serve 200k req/s
- System capacity: 100k req/s
- Queue grows, memory exhausted
- System crashes
- Effective throughput: 0

With shedding:

- Shed 100k req/s (priority 4)
- Serve 100k req/s (priority 1-3)
- System stays up
- Effective throughput: 100k req/s

Latency Impact

Without shedding:

- All requests queued
- Average latency: 30 seconds
- p99 latency: 300 seconds

With shedding:

- Priority 1 requests: 50ms (no queue)
- Priority 2-3 requests: 200ms (some queue)
- Priority 4 requests: immediate 503 (no latency)
- Served requests have good latency

Availability by Priority

Priority	Without Shedding	With Shedding
1 (Critical)	0% (crash)	99.9%
2 (High)	0% (crash)	95%
3 (Medium)	0% (crash)	60%
4 (Low)	0% (crash)	10%

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

Mistake 1: Not Propagating Priority Downstream

Why engineers do it: Forget to pass priority to downstream services.

What breaks: High-priority request calls low-priority service, gets shed.

Fix: Include X-Priority header in all outbound requests.

Mistake 2: Too Many Priority Levels

Why engineers do it: Want fine-grained control.

What breaks: Complex configuration, hard to reason about, priority inversion.

Fix: Use 3-4 priority levels maximum.

Mistake 3: Shedding at Wrong Layer

Why engineers do it: Apply shedding at load balancer.

What breaks: Load balancer doesn't know business logic priorities.

Fix: Apply at application layer where context available.

Mistake 4: No Monitoring of Shedding

Why engineers do it: Focus on serving successful requests.

What breaks: Don't notice when shedding critical traffic.

Fix: Alert on shed rate per priority level.

Mistake 5: Static Priority Assignment

Why engineers do it: Simple configuration.

What breaks: Business priorities change, configuration doesn't.

Fix: Dynamic priority from feature flags or database.

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

Anti-Pattern 1: Low-Traffic Systems

Don't add complexity for systems with consistent low traffic (<1000 req/s).

Anti-Pattern 2: All Traffic Equally Critical

Don't use load shedding if all operations have same priority (use backpressure instead).

Anti-Pattern 3: Life-Critical Systems

Don't shed requests in healthcare/safety systems where all requests critical.

11. Related Concepts (With Contrast)

Backpressure

Difference: Backpressure rejects all traffic equally. Load shedding rejects selectively by priority.

Rate Limiting

Difference: Rate limiting prevents abuse (per-user). Load shedding handles system overload (global).

Circuit Breaker

Difference: Circuit breaker stops calling failing service. Load shedding selectively drops requests to prevent overload.

12. Production Readiness Checklist

Priority Assignment

- [ ] Critical operations assigned priority 1
- [ ] Business impact documented per endpoint
- [ ] Code review for priority assignments
- [ ] Tests for critical endpoint priorities

Configuration

- [ ] Shedding thresholds per priority level
- [ ] Reserved capacity per priority
- [ ] Load metric selected (queue depth, latency, or error rate)

Monitoring

- [ ] Shed rate by priority level
- [ ] Served rate by priority level
- [ ] Current load percentage
- [ ] Alerts on shedding critical traffic (priority 1-2)

Testing

- [ ] Load test at 200% capacity

- [ ] Verify priority 1 never shed
- [ ] Verify priority 4 shed first
- [ ] Test graceful degradation

Communication

- [ ] User-facing error messages for shed requests
- [ ] Retry-After headers included
- [ ] Documentation for client retry logic
- [ ] Status page updates during incidents