Backpressure

1. The Real Problem This Exists to Solve

When a downstream system cannot keep up with the rate of incoming requests from upstream systems, unbounded buffering leads to memory exhaustion, latency explosion, and complete system collapse. Backpressure is the mechanism to signal upstream systems to slow down.

Real production scenario:

- API server receives 10,000 requests/second from load balancer
- Database can only handle 1,000 queries/second
- API server queues requests in memory while waiting for database
- Queue grows: 10,000 req/s in - 1,000 req/s out = net +9,000 req/s accumulation
- After 10 seconds: 90,000 requests queued in memory
- Memory usage: 90,000 × 10KB per request = 900MB
- After 60 seconds: 540,000 requests queued = 5.4GB memory
- Out of memory, process crashes
- All 540,000 in-flight requests lost
- Customers experience timeouts and data loss
- System cannot recover (restarts with same overload)

The fundamental problem: Without backpressure, fast producers overwhelm slow consumers. The buffer between them grows unbounded until system crashes.

Without backpressure:

- Systems accept all incoming load regardless of capacity
- Memory becomes the de-facto queue (until exhausted)
- Latency increases linearly with queue depth
- No signal to upstream that system is overloaded
- Crash is the only "feedback" mechanism

With backpressure:

- System signals "I'm full, slow down"
- Upstream respects signal (reduces send rate)
- Queue stays bounded
- System degrades gracefully under overload

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Unbounded Buffering

```
// Incorrect: Accept all requests, buffer indefinitely
class APIServer {
  private requestQueue: Request[] = [];

  async handleRequest(req: Request): Promise<Response> {
    // Always accept request, add to queue
    this.requestQueue.push(req);

    // Process asynchronously
    this.processQueue();

    return { status: 'accepted' };
  }

  private async processQueue() {
    while (this.requestQueue.length > 0) {
      const req = this.requestQueue.shift()!;
      await this.database.query(req.sql);
    }
  }
}
```

Why it seems reasonable:

- Never rejects requests (appears highly available)
- Simple to implement
- Async processing decouples request from execution
- Queue buffers traffic spikes

How it breaks:

- Queue grows unbounded when input > output rate
- Memory exhaustion after minutes of overload
- Process crashes, loses all buffered requests
- Latency becomes minutes/hours as queue grows
- Users wait indefinitely for response
- System cannot recover without fixing root cause (capacity)

Production symptoms:

- Memory usage grows linearly until OOM crash
- Request latency grows from 50ms to 60+ seconds
- Monitoring shows: 10k req/s in, 1k req/s out, queue depth 500k
- Process crashes every hour, requires restart
- Users report timeouts and lost requests
- Restart doesn't help (immediately overloaded again)

❌ Incorrect Approach #2: Timeout Without Rejection

```
// Incorrect: Timeout old requests but still accept new ones
class APIServer {
  private queue: QueuedRequest[] = [];
  private readonly TIMEOUT_MS = 30000;

  async handleRequest(req: Request): Promise<Response> {
    const promise = new Promise((resolve, reject) => {
      this.queue.push({
        req,
        resolve,
        reject,
        timestamp: Date.now(),
      });
    });

    // Timeout after 30s
    const timeout = setTimeout(() => {
      reject(new Error('Request timeout'));
    }, this.TIMEOUT_MS);

    try {
      return await promise;
    } finally {
      clearTimeout(timeout);
    }
  }

  private async processQueue() {
    while (true) {
      const item = this.queue.shift();
      if (!item) break;

      // Check if timed out
      if (Date.now() - item.timestamp > this.TIMEOUT_MS) {
        item.reject(new Error('Timeout'));
        continue;
      }

      const result = await this.database.query(item.req.sql);
      item.resolve(result);
    }
  }
}
```

Why it seems reasonable:

- Prevents requests from waiting forever
- Users get response (timeout error) within 30s
- Cleans up old requests from queue

How it breaks:

- Still accepts all incoming requests
- Queue still grows unbounded (timeouts don't prevent queuing)
- System wastes CPU processing timed-out requests
- Memory still grows until OOM (queue objects still in memory until processed)
- Users experience 100% timeout rate but system still drowning
- Doesn't signal upstream to slow down

Production symptoms:

- 100% timeout rate during overload
- Queue depth continues growing despite timeouts
- CPU at 100% processing requests that already timed out
- Memory still grows to OOM
- Users see only timeout errors

- Logs show: "Request took 45s, but client already disconnected"

❌ Incorrect Approach #3: Dropping Requests Silently

```
// Incorrect: Drop requests silently when overloaded
class APIServer {
  private queue: Request[] = [];
  private readonly MAX_QUEUE = 1000;

  async handleRequest(req: Request): Promise<Response> {
    if (this.queue.length >= this.MAX_QUEUE) {
      // Silently drop request
      return { status: 'ok' };  // Lie to client
    }

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

Why it seems reasonable:

- Prevents queue from growing unbounded
- Simple bounds checking
- Protects system from OOM

How it breaks:

- Drops requests without telling client
- Client thinks request succeeded (status 200)
- Data loss (client believes write succeeded but it was dropped)
- No signal to upstream to slow down (appears healthy)
- Load balancer continues sending full traffic
- Silent failures are worst kind of failure

Production symptoms:

- Users report: "I submitted order but it disappeared"
- 50% of requests silently dropped under load
- Metrics show 200 OK but database shows fewer writes
- No errors in logs (silent drops)
- Impossible to debug (no trace of dropped requests)
- Data integrity issues

❌ Incorrect Approach #4: Throwing Errors Without Rate Information

```
// Incorrect: Reject requests but don't provide retry guidance
class APIServer {
  private queue: Request[] = [];
  private readonly MAX_QUEUE = 1000;

  async handleRequest(req: Request): Promise<Response> {
    if (this.queue.length >= this.MAX_QUEUE) {
      throw new Error('Server overloaded');
    }

    return await this.processRequest(req);
  }
}

app.use((req, res) => {
  try {
    const result = await server.handleRequest(req);
    res.json(result);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

Why it seems reasonable:

- Rejects requests when overloaded
- Returns error to client
- Bounds queue depth

How it breaks:

- Returns HTTP 500 (Internal Server Error) instead of 503 (Service Unavailable)
- No `Retry-After` header to guide client backoff
- Clients don't know when to retry (immediately? 1s? 1min?)
- Clients retry immediately, amplifying overload
- Load balancer may interpret 500 as unhealthy, stop sending traffic
- Doesn't provide backpressure signal to upstream systems

Production symptoms:

- 50% error rate (500 errors)
- Clients retry immediately, making problem worse
- Retry storms (1 failed request → 10 retries)
- Load balancer removes server from pool (thinks it's down)
- Remaining servers receive more traffic, also fail
- Cascading failure across all servers

3. Correct Mental Model (How It Actually Works)

Backpressure is a control flow mechanism where downstream system signals upstream: "I'm at capacity, slow down."

The Flow Control Model

```
Producer (fast)  →  Buffer  →  Consumer (slow)
  10k req/s          Queue      1k req/s

Without backpressure:
  Buffer grows until memory exhausted
```

```
With backpressure:
  Buffer full → Signal producer → Producer slows to 1k req/s
```

Backpressure Signal Types

1. Explicit rejection (HTTP 503):

```
Client → Server
Request
        ← 503 Service Unavailable
        ← Retry-After: 5

Client waits 5 seconds before retrying
```

2. Implicit (TCP flow control):

```
Sender sends data faster than receiver can read
Receiver TCP window full → stops ACKing
Sender sees no ACKs → slows down automatically
```

3. Reactive Streams:

```
Subscriber requests N items
Publisher sends N items
Subscriber processes items
Subscriber requests N more items (pull model)
```

The Correct Response Chain

```
1. Database overloaded → query latency increases
2. API server detects: queue depth > threshold
3. API server returns: 503 + Retry-After header
4. Client receives 503, waits 5 seconds
5. Load balancer sees high 503 rate, reduces traffic
6. Queue depth decreases
7. System recovers
```

Key Properties

1. Bounded resources:

- Queue has maximum depth
- Reject requests beyond capacity
- Prevents memory exhaustion

2. Explicit signaling:

- Return 503 (not 500)
- Include Retry-After header
- Log rejection reason

3. Upstream propagation:

- Client respects backpressure

- Implements exponential backoff
- Load balancer adapts

4. Graceful degradation:

- System stays alive under overload
- Serves reduced capacity (1k req/s instead of 0)
- Can recover when load decreases

4. Correct Design & Algorithm

Strategy 1: Bounded Queue with Explicit Rejection

```
if queue.depth >= MAX_QUEUE:
  return 503 + Retry-After
else:
  queue.enqueue(request)
  process asynchronously
```

Strategy 2: Adaptive Backpressure Based on Latency

```
if p99_latency > SLA_THRESHOLD:
  start_rejecting = true
  rejection_probability = (p99_latency - SLA) / SLA

if start_rejecting && random() < rejection_probability:
  return 503 + Retry-After
```

Strategy 3: Token Bucket for Rate Limiting

```
tokens_available = capacity_per_second
every request:
  if tokens_available > 0:
    tokens_available--
    process request
  else:
    return 429 + Retry-After

every second:
  tokens_available = capacity_per_second
```

Strategy 4: Circuit Breaker on Downstream

```
if downstream_failures > threshold:
  circuit_open = true
  return 503 immediately (don't even try downstream)

after cooldown_period:
  circuit_half_open = true
  try one request
  if success:
    circuit_closed = true
```

5. Full Production-Grade Implementation

```
interface BackpressureConfig {
  maxQueueDepth: number;
  maxConcurrency: number;
  latencyThresholdMs: number;
  adaptiveRejection: boolean;
  retryAfterSeconds: number;
}

interface QueueMetrics {
  queueDepth: number;
  concurrency: number;
  p99Latency: number;
  rejectionRate: number;
  backpressureActive: boolean;
}

class BackpressureController {
  private queue: QueuedRequest[] = [];
  private inFlight = 0;
  private latencies: number[] = [];
  private rejectionCount = 0;
  private requestCount = 0;
  private config: BackpressureConfig;

  constructor(config: Partial<BackpressureConfig> = {}) {
    this.config = {
      maxQueueDepth: config.maxQueueDepth ?? 1000,
      maxConcurrency: config.maxConcurrency ?? 100,
      latencyThresholdMs: config.latencyThresholdMs ?? 1000,
      adaptiveRejection: config.adaptiveRejection ?? true,
      retryAfterSeconds: config.retryAfterSeconds ?? 5,
    };

    this.processQueue();
  }

  /**
   * Submit request with backpressure control.
   * Throws BackpressureError if system is overloaded.
   */
  async submit<T>(
    workFn: () => Promise<T>,
    priority: number = 0
  ): Promise<T> {
    this.requestCount++;
    // Check backpressure conditions
    const backpressure = this.shouldApplyBackpressure();
    if (backpressure.shouldReject) {
      this.rejectionCount++;
      throw new BackpressureError(
        backpressure.reason,
        this.config.retryAfterSeconds
      );
    }

    // Enqueue request
    return new Promise((resolve, reject) => {
      this.queue.push({
        workFn,
        priority,
        resolve,
        reject,
        enqueuedAt: Date.now(),
      });
    });
  }

  /**
   * Determine if backpressure should be applied
   */
  private shouldApplyBackpressure(): {
    shouldReject: boolean;
    reason: string;
  } {
    // Hard limit: queue depth exceeded
    if (this.queue.length >= this.config.maxQueueDepth) {
      return {
        shouldReject: true,
        reason: `Queue full (${this.queue.length}/${this.config.maxQueueDepth})`,
      };
    }

    // Adaptive backpressure based on latency
    if (this.config.adaptiveRejection) {
      const p99 = this.getP99Latency();
      if (p99 > this.config.latencyThresholdMs) {
        // Probabilistic rejection based on how far over threshold
        const overage = (p99 - this.config.latencyThresholdMs) /
                        this.config.latencyThresholdMs;
        const rejectionProb = Math.min(overage, 0.9); // Max 90% rejection

        if (Math.random() < rejectionProb) {
          return {
            shouldReject: true,
            reason: `High latency (p99=${p99}ms >
${this.config.latencyThresholdMs}ms)`,
          };
        }
      }
    }

    return { shouldReject: false, reason: '' };
  }

  /**
   * Process queue with concurrency control
   */
  private async processQueue(): Promise<void> {
    setImmediate(() => this.processQueue()); // Continuous processing

    if (this.inFlight >= this.config.maxConcurrency) {
      return; // At capacity
    }

    if (this.queue.length === 0) {
      return; // No work
    }

    // Sort by priority
    this.queue.sort((a, b) => b.priority - a.priority);
    const item = this.queue.shift()!;

    this.inFlight++;
    const startTime = Date.now();

    try {
      const result = await item.workFn();
      const latency = Date.now() - item.enqueuedAt;
      this.recordLatency(latency);
      item.resolve(result);
    } catch (error) {
      item.reject(error);
    } finally {
      this.inFlight--;
    }
  }

  /**
   * Record latency for percentile calculation
   */
  private recordLatency(latency: number): void {
    this.latencies.push(latency);
    if (this.latencies.length > 1000) {
      this.latencies.shift();
    }
  }

  /**
```

```
   * Calculate p99 latency
   */
  private getP99Latency(): number {
    if (this.latencies.length === 0) return 0;
    const sorted = [...this.latencies].sort((a, b) => a - b);
    const index = Math.floor(sorted.length * 0.99);
    return sorted[index];
  }

  /**
   * Get current metrics
   */
  getMetrics(): QueueMetrics {
    const p99 = this.getP99Latency();
    return {
      queueDepth: this.queue.length,
      concurrency: this.inFlight,
      p99Latency: p99,
      rejectionRate: this.rejectionCount / this.requestCount,
      backpressureActive: p99 > this.config.latencyThresholdMs ||
                          this.queue.length > this.config.maxQueueDepth * 0.8,
    };
  }

  /**
   * Reset metrics
   */
  resetMetrics(): void {
    this.rejectionCount = 0;
    this.requestCount = 0;
  }
}

class BackpressureError extends Error {
  constructor(
    public reason: string,
    public retryAfterSeconds: number
  ) {
    super(`Backpressure applied: ${reason}`);
    this.name = 'BackpressureError';
  }
}

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

// Express middleware with backpressure
```

```
const controller = new BackpressureController({
  maxQueueDepth: 1000,
  maxConcurrency: 100,
  latencyThresholdMs: 1000,
  adaptiveRejection: true,
  retryAfterSeconds: 5,
});

app.use(async (req, res, next) => {
  try {
    await controller.submit(async () => {
      await handleRequest(req, res);
    });
  } catch (error) {
    if (error instanceof BackpressureError) {
      res.status(503)
         .header('Retry-After', error.retryAfterSeconds.toString())
         .json({
           error: 'Service temporarily unavailable',
           reason: error.reason,
           retryAfter: error.retryAfterSeconds,
         });
    } else {
      next(error);
    }
  }
});

// Health check endpoint (should not apply backpressure)
app.get('/health', (req, res) => {
  const metrics = controller.getMetrics();

  if (metrics.backpressureActive) {
    res.status(503).json({
      status: 'degraded',
      metrics,
    });
  } else {
    res.status(200).json({
      status: 'healthy',
      metrics,
    });
  }
});

// Metrics endpoint
app.get('/metrics', (req, res) => {
  const metrics = controller.getMetrics();
  res.json({
    queue_depth: metrics.queueDepth,
    concurrency: metrics.concurrency,
    p99_latency_ms: metrics.p99Latency,
    rejection_rate: (metrics.rejectionRate * 100).toFixed(2) + '%',
    backpressure_active: metrics.backpressureActive,
  });
});

// Client-side retry logic with exponential backoff
async function makeRequestWithRetry<T>(
  fn: () => Promise<T>,
  maxRetries = 3
): Promise<T> {
  let attempt = 0;
  let delay = 1000;

  while (attempt < maxRetries) {
    try {
      return await fn();
    } catch (error) {
      if (error.response?.status === 503) {
        const retryAfter = parseInt(error.response.headers['retry-after'] || '5');
        const backoffDelay = Math.min(delay * Math.pow(2, attempt), retryAfter *
1000);

        console.log(`Backpressure detected, retrying after ${backoffDelay}ms`);
        await sleep(backoffDelay);

        attempt++;
      } else {
        throw error;
      }
    }
  }

  throw new Error('Max retries exceeded');
}

function sleep(ms: number): Promise<void> {
  return new Promise(resolve => setTimeout(resolve, ms));
}
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: API Gateway with Backpressure

```
app.post('/api/orders', async (req, res) => {
  try {
    const result = await controller.submit(async () => {
      return await orderService.create(req.body);
    }, 10); // High priority

    res.json(result);
  } catch (error) {
    if (error instanceof BackpressureError) {
      res.status(503)
         .header('Retry-After', '5')
         .json({ error: 'System overloaded, please retry' });
    } else {
      res.status(500).json({ error: 'Internal error' });
    }
  }
});
```

Why this works:

- Returns 503 when overloaded (not 500)
- Includes Retry-After header for client guidance
- Bounds queue depth (prevents OOM)
- System stays alive under overload

Pattern 2: Message Queue Consumer with Backpressure

```
async function consumeMessages() {
  while (true) {
    try {
      const messages = await queue.receive({ maxMessages: 10 });

      for (const msg of messages) {
        try {
          await controller.submit(() => processMessage(msg));
          await queue.deleteMessage(msg);
        } catch (error) {
          if (error instanceof BackpressureError) {
            // Don't delete message, it will be retried
            console.log('Backpressure active, message will be redelivered');
            await sleep(5000);
          } else {
            await queue.sendToDeadLetter(msg);
          }
        }
      }
    } catch (error) {
      console.error('Queue receive error:', error);
      await sleep(1000);
    }
  }
}
```

Why this works:

- Doesn't delete message when backpressure applied
- Message redelivered automatically after visibility timeout
- Natural backpressure (messages stay in queue)

- Consumer doesn't crash from overload

Pattern 3: Stream Processing with Backpressure

```typescript
import { Readable, Writable } from 'stream';

class BackpressureStream extends Writable {
  constructor(private processor: (chunk: any) => Promise<void>) {
    super({ objectMode: true, highWaterMark: 100 });
  }

  async _write(
    chunk: any,
    encoding: string,
    callback: (error?: Error | null) => void
  ): Promise<void> {
    try {
      await controller.submit(() => this.processor(chunk));
      callback();
    } catch (error) {
      if (error instanceof BackpressureError) {
        // Pause stream, will resume when ready
        setTimeout(() => callback(), 5000);
      } else {
        callback(error as Error);
      }
    }
  }
}

// Usage
const inputStream = fs.createReadStream('data.json');
const outputStream = new BackpressureStream(async (data) => {
  await database.insert(data);
});

inputStream.pipe(outputStream);
// Stream automatically pauses when backpressure applied
```

Why this works:

- Node.js streams have built-in backpressure support
- Write stream signals "not ready" when overloaded
- Read stream pauses automatically
- No unbounded buffering

7. Failure Modes & Edge Cases

Retry Storm

Problem: All clients retry simultaneously after backpressure, amplifying overload.

Symptoms:

- 503 rate drops, then spikes again
- System oscillates between working and overloaded
- Load never stabilizes

Mitigation:

- Clients use exponential backoff with jitter
- Vary Retry-After header (5-15s random)
- Rate limit retries separately

False Backpressure from Transient Spike

Problem: Brief latency spike triggers backpressure, but system actually healthy.

Symptoms:

- Backpressure triggers on single slow query
- 503 errors despite low queue depth
- Users experience unnecessary failures

Mitigation:

- Use moving average for latency (not single sample)
- Require sustained high latency before rejecting
- Probabilistic rejection (don't reject 100%)

Load Balancer Removing Healthy Servers

Problem: Load balancer sees high 503 rate, marks server unhealthy.

Symptoms:

- All traffic moves to other servers
- Those servers also overload
- Cascading failure

Mitigation:

- Configure load balancer to accept 503 as "degraded but healthy"
- Health check endpoint separate from traffic endpoint
- Return 200 on health check even when applying backpressure

Client Doesn't Respect Backpressure

Problem: Client ignores 503, retries immediately.

Symptoms:

- Server applies backpressure but load doesn't decrease
- Rejection rate climbs to 90%+
- System cannot recover

Mitigation:

- Rate limit per client (block misbehaving clients)
- Circuit breaker on server side
- Contact client owners to fix retry logic

8. Performance Characteristics & Tradeoffs

Availability Tradeoff

Without backpressure:

- 100% availability until crash
- Then 0% availability (complete outage)

With backpressure:

- 95% availability during overload (5% rejection)
- Never 0% (system stays alive)

Result: Better overall availability

Latency Impact

Without backpressure:

- Queue depth 100,000
- Latency: queue depth × service time = 100,000 × 100ms = 10,000 seconds

With backpressure:

- Queue depth bounded at 1,000
- Latency: 1,000 × 100ms = 100 seconds
- Rejected requests get immediate response (no latency)

Result: 100× better latency for accepted requests

Throughput

Without backpressure:

- Accept 10k req/s, process 1k req/s
- Effective throughput: 0 (all requests timeout)

With backpressure:

- Accept 1k req/s (reject 9k)
- Process 1k req/s
- Effective throughput: 1k req/s

Result: Better effective throughput

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

Mistake 1: Returning 500 Instead of 503

Why engineers do it: Don't distinguish between errors and overload.

What breaks: Load balancer marks server unhealthy.

Fix: Return 503 for backpressure, include Retry-After.

Mistake 2: No Retry-After Header

Why engineers do it: Forget to include it.

What breaks: Clients don't know when to retry.

Fix: Always include Retry-After with 503.

Mistake 3: Applying Backpressure to Health Checks

Why engineers do it: All endpoints share same queue.

What breaks: Health checks fail, load balancer removes server.

Fix: Health checks bypass backpressure.

Mistake 4: Not Propagating Backpressure Upstream

Why engineers do it: Only apply locally.

What breaks: Queue depth at upstream systems grows.

Fix: Propagate 503 to callers.

Mistake 5: Fixed Retry-After Value

Why engineers do it: Simple configuration.

What breaks: Synchronized retry storm.

Fix: Randomize Retry-After (5-15s).

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

Anti-Pattern 1: Critical Real-Time Systems

Don't apply backpressure to life-critical systems:

```
// WRONG for pacemaker control
if (queue.full()) return 503;
```

Anti-Pattern 2: Fire-and-Forget Operations

Don't apply to async operations where failure acceptable:

```
// WRONG for analytics logging
analytics.log(event); // Don't reject, drop is fine
```

Anti-Pattern 3: Low-Traffic Systems

Don't add complexity for systems with <100 req/s.

11. Related Concepts (With Contrast)

Rate Limiting

Difference: Rate limiting prevents abuse (per-user). Backpressure prevents system overload (global).

Circuit Breaker

Difference: Circuit breaker stops calling failing downstream. Backpressure stops accepting requests when downstream slow.

Load Shedding

Difference: Load shedding drops low-priority requests. Backpressure rejects requests with retry guidance.

12. Production Readiness Checklist

Metrics

- [ ] Queue depth
- [ ] Rejection rate (503 responses)
- [ ] p99 latency
- [ ] Retry-After values
- [ ] Backpressure active duration

Configuration

- [ ] maxQueueDepth set based on memory limits
- [ ] latencyThresholdMs set based on SLA
- [ ] retryAfterSeconds tuned (5-15s with jitter)

Testing

- [ ] Load test at 2× capacity
- [ ] Verify 503 returned with Retry-After
- [ ] Verify queue depth bounded
- [ ] Verify system doesn't crash
- [ ] Test client retry logic

Rollout

- [ ] Deploy with high thresholds (conservative)
- [ ] Monitor rejection rate
- [ ] Tune thresholds based on observations
- [ ] Document backpressure behavior for clients

Alerting

- [ ] Alert if rejection rate > 10%
- [ ] Alert if backpressure active > 5 minutes
- [ ] Alert if queue depth > 80% of max