Adaptive Rate Limiting

1. The Real Problem This Exists to Solve

Traditional rate limiting uses fixed thresholds (e.g., "100 requests per second") that don't account for system health, varying request costs, or changing traffic patterns. Adaptive rate limiting dynamically adjusts limits based on system capacity and observed behavior.

Real production scenario:

- API gateway with fixed limit: 1000 req/s per user
- Normal operation: Each request takes 10ms, system handles 1000 req/s easily
- Attacker exploits expensive endpoint: Search query taking 5 seconds per request
- With 200 search requests, system is saturated (200 × 5s = 1000s of CPU time)
- But rate limiter allows 1000 req/s (attacker only using 200 req/s)
- System overloaded despite being "under" rate limit
- Legitimate users experience failures
- Fixed rate limit doesn't prevent abuse

The core problem: Fixed rate limits assume all requests have equal cost. In reality, requests have vastly different resource consumption (10ms database lookup vs 5s complex search query vs 30s report generation).

Traditional fixed rate limiting:

- Same limit regardless of system health
- Doesn't account for request complexity
- Can't adapt to traffic patterns
- Allows abuse through expensive operations
- Rejects requests even when system healthy

Adaptive rate limiting:

- Adjusts limits based on system load (CPU, latency, error rate)
- Accounts for request cost (expensive operations get lower limits)
- Learns from traffic patterns
- Protects system from sophisticated abuse
- Maximizes throughput while preventing overload

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Fixed Per-User Rate Limit

```
// Incorrect: Same limit for all requests
const limiter = rateLimit({
  windowMs: 1000,
  max: 100,  // 100 requests per second
  keyGenerator: (req) => req.user.id,
});

app.use(limiter);

app.get('/quick', handleQuickQuery);     // 10ms
```

```
app.get('/search', handleSearch);          // 5s
app.post('/report', handleReport);         // 30s
```

Why it seems reasonable:

- Simple to configure
- Prevents individual users from overwhelming system
- Standard rate limiting pattern
- Easy to communicate to users

How it breaks:

- Attacker sends 100 expensive search requests (5s each) = 500s CPU time
- Rate limiter sees 100 req/s and allows it
- System overloaded from 100 requests
- Meanwhile legitimate user making quick queries gets rate limited
- One expensive request = same cost as one cheap request (wrong)

Production symptoms:

- System overloaded despite rate limits
- CPU at 100% from 50 users making expensive queries
- Quick endpoint users getting rate limited
- Attacker bypasses rate limit by using expensive endpoints
- Logs show: "Rate limit: 100/100 req/s" but system dying

❌ Incorrect Approach #2: Per-Endpoint Fixed Limits

```
// Incorrect: Different limits per endpoint but still fixed
app.get('/quick', rateLimit({ max: 1000 }), handleQuickQuery);
app.get('/search', rateLimit({ max: 100 }), handleSearch);
app.post('/report', rateLimit({ max: 10 }), handleReport);
```

Why it seems reasonable:

- Accounts for endpoint cost
- Expensive endpoints get lower limits
- Protects from expensive operations

How it breaks:

- Limits don't adjust to system health
- If system has spare capacity, artificially limits throughput
- If system overloaded, limits may still be too high
- Doesn't account for varying request complexity within endpoint
  - Search for "a" is fast
  - Search for "electronics under $100" is slow
- Hard to tune (requires manual adjustment)

Production symptoms:

- Search endpoint limited to 100 req/s even when system idle
- System overloaded when many users hit limit simultaneously
- Wasted capacity during low traffic

- Manual limit adjustments required after every incident

❌ Incorrect Approach #3: CPU-Based Global Rate Limit

```
// Incorrect: Adjust global limit based on CPU, no per-user fairness
import os from 'os';

class CPUAdjustedRateLimiter {
  private currentLimit = 1000;

  async checkLimit(req: Request): Promise<boolean> {
    const cpuUsage = os.loadavg()[0] / os.cpus().length;

    // Adjust limit based on CPU
    if (cpuUsage > 0.8) {
      this.currentLimit = 500;
    } else if (cpuUsage < 0.5) {
      this.currentLimit = 2000;
    }

    // But no per-user tracking!
    const currentRate = this.getGlobalRate();
    return currentRate < this.currentLimit;
  }
}
```

Why it seems reasonable:

- Adapts to system health
- Increases limit when system idle
- Decreases limit when system overloaded

How it breaks:

- No per-user fairness (single user can consume all quota)
- Attacker sends 500 req/s, consumes entire adjusted limit
- Legitimate users get nothing
- Global limit doesn't prevent individual abuse
- CPU may be high from background jobs, not user traffic

Production symptoms:

- One user consuming 80% of traffic
- Other users getting rate limited
- System appears "protected" but single user dominating
- No fairness between users

❌ Incorrect Approach #4: Latency-Based Rejection Without History

```
// Incorrect: Reject based on current latency, no smoothing
class LatencyBasedLimiter {
  async checkRequest(req: Request): Promise<boolean> {
    const currentLatency = await measureLatency();
```

```
    // Reject if latency high
    if (currentLatency > 1000) {
      return false;  // Rate limited
    }

    return true;
  }
}
```

Why it seems reasonable:

- Protects system when latency high
- Direct feedback from system health
- Prevents overload

How it breaks:

- Single slow query causes all requests to be rejected
- No smoothing (reacts to every spike)
- Doesn't identify which users causing high latency
- All users punished for one user's expensive query
- Oscillates (rejects → latency drops → allows → latency spikes → rejects)

Production symptoms:

- 503 errors spike after every slow query
- System oscillates between accepting and rejecting
- Legitimate users experience intermittent failures
- No way to identify culprit users

3. Correct Mental Model (How It Actually Works)

Adaptive rate limiting combines multiple signals to dynamically adjust limits based on system health and individual request costs.

The Feedback Loop

```
1. Measure system health (latency, CPU, error rate)
2. Calculate available capacity
3. Distribute capacity among users
4. Track per-user consumption
5. Adjust limits based on behavior
6. Repeat every second
```

Cost-Based Limiting

```
Request arrives
  ↓
Measure cost (estimate or measure)
  - CPU time
  - Memory usage
  - Database queries
```

```
  - External API calls
  ↓
Deduct cost from user's token bucket
  ↓
If tokens available: serve
If tokens exhausted: rate limit
```

The AIMD Algorithm (Additive Increase, Multiplicative Decrease)

```
Initial limit = baseline (e.g., 100 req/s)

Every second:
  If system healthy (latency < SLA):
    limit += additive_increase (e.g., +10 req/s)

  If system overloaded (latency > SLA):
    limit *= multiplicative_decrease (e.g., × 0.5)

Result: Converges to optimal throughput
```

Request Cost Model

```
Cost = base_cost + query_cost + computation_cost

Examples:
  GET /user/123
    base = 1, query = 1 (index lookup), compute = 0
    total cost = 2 units

  GET /search?q=electronics
    base = 1, query = 50 (full text search), compute = 10
    total cost = 61 units

  POST /report/generate
    base = 1, query = 500 (complex joins), compute = 1000
    total cost = 1501 units

Token bucket:
  User has 1000 tokens/second
  Can make: 500 quick queries OR 16 searches OR 1 report
```

4. Correct Design & Algorithm

Strategy 1: AIMD with Per-User Token Buckets

```
Per-user token bucket: capacity C, refill rate R

On request:
  cost = estimate_cost(request)
  if user.tokens >= cost:
```

```
    user.tokens -= cost
    process request
  else:
    return 429 (rate limited)

Every second:
  if system_healthy():
    R += additive_increase
  else if system_overloaded():
    R *= multiplicative_decrease

  for each user:
    user.tokens = min(C, user.tokens + R)
```

Strategy 2: Latency-Based Adaptive Limits

```
target_latency = 100ms
current_p99 = measure_p99_latency()

if current_p99 < target_latency:
  # System has headroom, increase limits
  rate_limit *= 1.1
else if current_p99 > target_latency * 2:
  # System overloaded, decrease limits
  rate_limit *= 0.5
```

Strategy 3: Cost-Based Token Consumption

```
request_cost = base_cost
if has_database_query:
  request_cost += query_complexity_score
if has_computation:
  request_cost += estimated_cpu_ms / 10

tokens_required = request_cost
if user.tokens >= tokens_required:
  allow
else:
  rate_limit
```

5. Full Production-Grade Implementation

```typescript
interface AdaptiveRateLimiterConfig {
  initialLimit: number;
  minLimit: number;
  maxLimit: number;
  targetLatencyMs: number;
  additiveIncrease: number;
  multiplicativeDecrease: number;
```

```
  costModel: CostModel;
}

interface CostModel {
  baseCost: number;
  costByEndpoint: Map<string, number>;
  costByComplexity: (req: Request) => number;
}

interface UserQuota {
  tokens: number;
  maxTokens: number;
  refillRate: number;
  lastRefill: number;
  requestCount: number;
  totalCost: number;
}

class AdaptiveRateLimiter {
  private config: AdaptiveRateLimiterConfig;
  private userQuotas: Map<string, UserQuota> = new Map();
  private currentGlobalLimit: number;
  private latencyHistory: number[] = [];
  private systemMetrics: SystemMetrics;

  constructor(config: Partial<AdaptiveRateLimiterConfig> = {}) {
    this.config = {
      initialLimit: config.initialLimit ?? 100,
      minLimit: config.minLimit ?? 10,
      maxLimit: config.maxLimit ?? 10000,
      targetLatencyMs: config.targetLatencyMs ?? 100,
      additiveIncrease: config.additiveIncrease ?? 10,
      multiplicativeDecrease: config.multiplicativeDecrease ?? 0.5,
      costModel: config.costModel ?? this.defaultCostModel(),
    };

    this.currentGlobalLimit = this.config.initialLimit;
    this.systemMetrics = { p99Latency: 0, requestRate: 0, errorRate: 0 };

    // Update limits every second
    setInterval(() => this.adaptLimits(), 1000);
    // Refill tokens every 100ms
    setInterval(() => this.refillTokens(), 100);
  }

  /**
   * Check if request should be rate limited
   */
  async checkLimit(userId: string, request: Request): Promise<RateLimitResult> {
    const cost = this.estimateCost(request);
    const quota = this.getOrCreateQuota(userId);
```

```
    // Refill tokens if needed
    this.refillUserTokens(quota);

    // Check if user has enough tokens
    if (quota.tokens >= cost) {
      quota.tokens -= cost;
      quota.requestCount++;
      quota.totalCost += cost;

      return {
        allowed: true,
        cost,
        remaining: quota.tokens,
        resetAt: this.getResetTime(quota),
      };
    }

    // Rate limited
    return {
      allowed: false,
      cost,
      remaining: 0,
      resetAt: this.getResetTime(quota),
      retryAfter: Math.ceil((cost - quota.tokens) / quota.refillRate),
    };
  }

  /**
   * Estimate cost of request
   */
  private estimateCost(request: Request): number {
    let cost = this.config.costModel.baseCost;

    // Add endpoint-specific cost
    const endpointCost = this.config.costModel.costByEndpoint.get(request.path);
    if (endpointCost) {
      cost += endpointCost;
    }

    // Add complexity-based cost
    const complexityCost = this.config.costModel.costByComplexity(request);
    cost += complexityCost;

    return Math.max(cost, 1);  // Minimum cost of 1
  }

  /**
   * Get or create user quota
   */
  private getOrCreateQuota(userId: string): UserQuota {
    let quota = this.userQuotas.get(userId);
```

```
  if (!quota) {
    quota = {
      tokens: this.currentGlobalLimit,
      maxTokens: this.currentGlobalLimit,
      refillRate: this.currentGlobalLimit,
      lastRefill: Date.now(),
      requestCount: 0,
      totalCost: 0,
    };
    this.userQuotas.set(userId, quota);
  }

  return quota;
}

/**
 * Refill tokens for user
 */
private refillUserTokens(quota: UserQuota): void {
  const now = Date.now();
  const elapsed = (now - quota.lastRefill) / 1000;  // seconds
  const tokensToAdd = elapsed * quota.refillRate;

  quota.tokens = Math.min(quota.maxTokens, quota.tokens + tokensToAdd);
  quota.lastRefill = now;
}

/**
 * Refill all user tokens (called every 100ms)
 */
private refillTokens(): void {
  for (const quota of this.userQuotas.values()) {
    this.refillUserTokens(quota);
  }
}

/**
 * Adapt limits based on system health (AIMD algorithm)
 */
private adaptLimits(): void {
  const p99 = this.getP99Latency();
  this.systemMetrics.p99Latency = p99;

  if (p99 < this.config.targetLatencyMs) {
    // System healthy, increase limit (additive)
    this.currentGlobalLimit = Math.min(
      this.config.maxLimit,
      this.currentGlobalLimit + this.config.additiveIncrease
    );
  } else if (p99 > this.config.targetLatencyMs * 2) {
    // System overloaded, decrease limit (multiplicative)
    this.currentGlobalLimit = Math.max(
```

```
      this.config.minLimit,
      this.currentGlobalLimit * this.config.multiplicativeDecrease
    );
  }

  // Update all user quotas with new limit
  for (const quota of this.userQuotas.values()) {
    quota.maxTokens = this.currentGlobalLimit;
    quota.refillRate = this.currentGlobalLimit;
  }
}

/**
 * Record request latency
 */
recordLatency(latency: number): void {
  this.latencyHistory.push(latency);
  if (this.latencyHistory.length > 1000) {
    this.latencyHistory.shift();
  }
}

/**
 * Get p99 latency
 */
private getP99Latency(): number {
  if (this.latencyHistory.length === 0) return 0;

  const sorted = [...this.latencyHistory].sort((a, b) => a - b);
  const index = Math.floor(sorted.length * 0.99);
  return sorted[index];
}

/**
 * Get reset time for user quota
 */
private getResetTime(quota: UserQuota): number {
  const secondsUntilFull = (quota.maxTokens - quota.tokens) / quota.refillRate;
  return Date.now() + (secondsUntilFull * 1000);
}

/**
 * Default cost model
 */
private defaultCostModel(): CostModel {
  return {
    baseCost: 1,
    costByEndpoint: new Map([
      ['/api/quick', 1],
      ['/api/search', 50],
      ['/api/report', 500],
    ]),
```

```
      costByComplexity: (req) => {
        // Estimate complexity from query params
        let complexity = 0;

        if (req.query.search) {
          complexity += req.query.search.length * 2;
        }

        if (req.query.limit) {
          complexity += parseInt(req.query.limit) / 10;
        }

        return complexity;
      },
    };
  }

  /**
   * Get statistics
   */
  getStats(): RateLimiterStats {
    const userStats = Array.from(this.userQuotas.entries()).map(([userId, quota]) =>
({
      userId,
      requestCount: quota.requestCount,
      totalCost: quota.totalCost,
      tokensRemaining: quota.tokens,
      avgCostPerRequest: quota.totalCost / Math.max(quota.requestCount, 1),
    }));

    return {
      currentLimit: this.currentGlobalLimit,
      p99Latency: this.systemMetrics.p99Latency,
      totalUsers: this.userQuotas.size,
      userStats: userStats.sort((a, b) => b.totalCost - a.totalCost).slice(0, 10),
    };
  }

  /**
   * Reset statistics
   */
  resetStats(): void {
    for (const quota of this.userQuotas.values()) {
      quota.requestCount = 0;
      quota.totalCost = 0;
    }
  }
}

interface Request {
  path: string;
  query: Record<string, any>;
```

```
  body: any;
  user: any;
}

interface RateLimitResult {
  allowed: boolean;
  cost: number;
  remaining: number;
  resetAt: number;
  retryAfter?: number;
}

interface SystemMetrics {
  p99Latency: number;
  requestRate: number;
  errorRate: number;
}

interface RateLimiterStats {
  currentLimit: number;
  p99Latency: number;
  totalUsers: number;
  userStats: Array<{
    userId: string;
    requestCount: number;
    totalCost: number;
    tokensRemaining: number;
    avgCostPerRequest: number;
  }>;
}

// Express middleware
const rateLimiter = new AdaptiveRateLimiter({
  initialLimit: 1000,
  minLimit: 100,
  maxLimit: 10000,
  targetLatencyMs: 100,
  additiveIncrease: 100,
  multiplicativeDecrease: 0.75,
  costModel: {
    baseCost: 1,
    costByEndpoint: new Map([
      ['/api/user/:id', 1],
      ['/api/search', 50],
      ['/api/analytics', 20],
      ['/api/report', 500],
    ]),
    costByComplexity: (req) => {
      let cost = 0;

      // Search queries more expensive for longer terms
      if (req.query.q) {
```

```
        cost += Math.min(req.query.q.length * 2, 100);
      }

      // Pagination: higher offsets = more expensive
      if (req.query.offset) {
        cost += parseInt(req.query.offset) / 100;
      }

      // Sorting adds cost
      if (req.query.sort) {
        cost += 10;
      }

      return cost;
    },
  },
});

app.use(async (req, res, next) => {
  const userId = req.user?.id || req.ip;
  const startTime = Date.now();

  const result = await rateLimiter.checkLimit(userId, req);

  // Add rate limit headers
  res.setHeader('X-RateLimit-Limit', result.cost.toString());
  res.setHeader('X-RateLimit-Remaining', result.remaining.toString());
  res.setHeader('X-RateLimit-Reset', result.resetAt.toString());

  if (!result.allowed) {
    res.status(429)
       .setHeader('Retry-After', result.retryAfter!.toString())
       .json({
         error: 'Rate limit exceeded',
         cost: result.cost,
         retryAfter: result.retryAfter,
       });
    return;
  }

  // Record latency after request completes
  res.on('finish', () => {
    const latency = Date.now() - startTime;
    rateLimiter.recordLatency(latency);
  });

  next();
});

// Expensive endpoint
app.get('/api/search', async (req, res) => {
  const results = await searchDatabase(req.query.q);
```

```
  res.json(results);
});

// Cheap endpoint
app.get('/api/user/:id', async (req, res) => {
  const user = await getUser(req.params.id);
  res.json(user);
});

// Very expensive endpoint
app.post('/api/report', async (req, res) => {
  const report = await generateReport(req.body);
  res.json(report);
});

// Stats endpoint
app.get('/api/stats/rate-limit', (req, res) => {
  const stats = rateLimiter.getStats();
  res.json({
    currentLimit: stats.currentLimit,
    p99LatencyMs: stats.p99Latency.toFixed(0),
    totalUsers: stats.totalUsers,
    topUsers: stats.userStats.map(u => ({
      userId: u.userId,
      requests: u.requestCount,
      totalCost: u.totalCost,
      avgCost: u.avgCostPerRequest.toFixed(2),
    })),
  });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: API with Mixed Request Costs

```
const rateLimiter = new AdaptiveRateLimiter({
  costModel: {
    baseCost: 1,
    costByEndpoint: new Map([
      ['/api/user/:id', 1],           // Simple lookup
      ['/api/search', 50],             // Full-text search
      ['/api/export', 100],            // Large data export
      ['/api/ml/predict', 200],        // ML inference
    ]),
    costByComplexity: (req) => {
      if (req.path === '/api/search' && req.query.fullText) {
        return 100;  // Full-text search is expensive
      }
      return 0;
    },
```

```
  },
});
```

Why this works:

- User can make 1000 simple lookups OR 20 searches OR 5 ML predictions
- Fair distribution based on resource consumption
- Prevents abuse of expensive endpoints

Pattern 2: Multi-Tenant SaaS with Tier-Based Limits

```
function getLimitForTier(tier: string): number {
  const limits = {
    free: 100,
    basic: 1000,
    premium: 10000,
    enterprise: 100000,
  };
  return limits[tier] || limits.free;
}

const rateLimiter = new AdaptiveRateLimiter({
  initialLimit: 1000,
});

app.use(async (req, res, next) => {
  const userId = req.user.id;
  const tierLimit = getLimitForTier(req.user.tier);

  // Override global limit with tier-specific limit
  const result = await rateLimiter.checkLimit(userId, req);

  if (!result.allowed && result.remaining < tierLimit * 0.1) {
    res.setHeader('X-Upgrade-Available', 'true');
    res.setHeader('X-Upgrade-Tier', 'premium');
  }

  // ... rest of handling
});
```

Why this works:

- Different limits for different customer tiers
- Encourages upgrades when approaching limit
- Adaptive behavior within tier constraints

Pattern 3: Burst Handling with Token Bucket

```
const rateLimiter = new AdaptiveRateLimiter({
  initialLimit: 100,  // 100 req/s sustained
  maxLimit: 1000,     // But allow bursts up to 1000
```

```
});

// User can burst to 1000 req/s for short periods
// Then rate drops back to 100 req/s sustained
```

Why this works:

- Handles traffic spikes gracefully
- Users not penalized for occasional bursts
- System protected from sustained high traffic

7. Failure Modes & Edge Cases

Cold Start Problem

Problem: New users have no history, system doesn't know their behavior.

Symptoms:

- Legitimate user makes expensive query as first request
- System doesn't know if it's abuse or legitimate
- May be too permissive or too restrictive

Mitigation:

- Start with conservative limit
- Gradually increase limit for good actors
- Decrease limit for suspicious behavior

Limit Oscillation

Problem: Limits increase and decrease rapidly, causing instability.

Symptoms:

- Limit swings between 100 and 10,000 rapidly
- Users experience inconsistent behavior
- Hard to predict available capacity

Mitigation:

- Smooth latency measurements (moving average)
- Add dampening to AIMD algorithm
- Set reasonable min/max bounds

Cost Estimation Errors

Problem: Estimated cost doesn't match actual cost.

Symptoms:

- User charged 50 tokens but request takes 500ms
- System overloaded despite rate limiting
- Unfair penalization for cheap requests estimated as expensive

Mitigation:

- Measure actual cost and update estimates

- Refund tokens if actual cost lower than estimated
- Feedback loop to improve cost model

Limit Starvation for Bursty Users

Problem: User makes burst of requests, exhausts tokens, can't make any requests.

Symptoms:

- User hits limit, must wait 10 seconds for refill
- User experiences complete outage
- No graceful degradation

Mitigation:

- Allow small burst even when tokens low
- Reserve minimum tokens for critical operations
- Graceful degradation (serve cached data)

8. Performance Characteristics & Tradeoffs

Latency Stability

Fixed rate limiting:

- Latency can grow unbounded under load
- No feedback mechanism

Adaptive rate limiting:

- Latency stays near target (100ms)
- System rejects requests to maintain SLA
- Served requests have predictable performance

Throughput Optimization

Fixed limit of 1000 req/s:

- System capacity 5000 req/s during off-peak
- Wastes 4000 req/s capacity
- During peak: 1000 req/s may overload system

Adaptive limit:

- Off-peak: Increases to 5000 req/s (uses full capacity)
- Peak: Decreases to 800 req/s (prevents overload)
- Optimal throughput at all times

Fairness

Fixed per-user limit:

- User making expensive requests same limit as cheap requests
- Unfair resource allocation

Cost-based adaptive limit:

- User charged based on actual resource consumption
- Fair allocation based on cost

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

Mistake 1: Not Measuring Actual Request Cost

Why engineers do it: Assume all requests equal cost.

What breaks: Expensive requests bypass rate limiting.

Fix: Profile endpoints, measure actual CPU/latency/query count.

Mistake 2: Adapting Too Quickly

Why engineers do it: Want fast response to overload.

What breaks: Limits oscillate wildly, unstable system.

Fix: Use moving average, add dampening, tune AIMD parameters.

Mistake 3: No Minimum Guaranteed Capacity

Why engineers do it: Trust adaptive algorithm completely.

What breaks: Limit decreases to near-zero during spike.

Fix: Set minLimit to ensure minimum service level.

Mistake 4: Ignoring Burst Traffic

Why engineers do it: Focus on sustained rate.

What breaks: Legitimate burst traffic gets rate limited.

Fix: Use token bucket with burst capacity.

Mistake 5: Not Returning Retry-After Header

Why engineers do it: Just return 429.

What breaks: Client doesn't know when to retry.

Fix: Calculate and return Retry-After based on token refill rate.

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

Anti-Pattern 1: Uniform Request Costs

If all requests have equal cost, use simple fixed rate limiting.

Anti-Pattern 2: Strict SLAs

If you must guarantee exact rate limits (compliance), use fixed limits.

Anti-Pattern 3: Unpredictable Backend

If backend latency varies randomly (not load-related), adaptive rate limiting won't help.

11. Related Concepts (With Contrast)

Fixed Rate Limiting

Difference: Adaptive adjusts limits based on system health. Fixed uses constant limits.

Load Shedding

Difference: Load shedding drops low-priority requests. Adaptive rate limiting throttles all users fairly.

Backpressure

Difference: Backpressure rejects when queue full. Adaptive rate limiting prevents queue from filling.

Circuit Breaker

Difference: Circuit breaker stops calling failing service. Adaptive rate limiting prevents overload before failure.

12. Production Readiness Checklist

Cost Model

- [ ] Profile endpoints to measure actual cost
- [ ] Define cost for each endpoint
- [ ] Add complexity-based cost calculation
- [ ] Validate cost estimates match actual resource usage

AIMD Parameters

- [ ] Set target latency based on SLA
- [ ] Tune additive increase (too high = overshoot)
- [ ] Tune multiplicative decrease (too high = unstable)
- [ ] Set min/max limits

Monitoring

- [ ] Track current adaptive limit
- [ ] Track per-user token consumption
- [ ] Alert on rapid limit decrease
- [ ] Dashboard showing limit over time

Testing

- [ ] Load test with mixed request types
- [ ] Verify limits adapt to load
- [ ] Test burst traffic handling
- [ ] Verify fairness (expensive vs cheap requests)

Headers

- [ ] X-RateLimit-Limit (current cost)
- [ ] X-RateLimit-Remaining (tokens left)
- [ ] X-RateLimit-Reset (when tokens refill)
- [ ] Retry-After (on 429)