Hedged Requests

1. The Real Problem This Exists to Solve

In distributed systems, request latency follows a heavy-tailed distribution where a small percentage of requests are dramatically slower than the median. Even with healthy services, these slow outliers create poor user experience at the tail percentiles (p95, p99).

Real production scenario:

- Payment processing service has p50 latency of 50ms
- p99 latency is 2000ms (40× slower than median)
- One in 100 checkout requests takes 2+ seconds
- Users perceive the system as slow, abandon checkouts
- Slow requests are caused by: GC pauses, network packet loss, database lock contention, hitting slow replica
- Timeouts don't help (waiting 2s doesn't prevent user from experiencing 2s)
- Retrying after timeout doubles latency (2s timeout + 50ms retry = 2.05s total)

The fundamental problem: **you cannot predict which request will be slow**. The first attempt might hit a slow path, and you only discover this after waiting. Traditional retry (wait for failure, then retry) adds latency rather than reducing it.

Hedged requests solve this by **issuing duplicate requests proactively** before the first request completes. If the primary request is slow, the duplicate completes faster. Take the first successful response and cancel the others.

Key insight: **Slight increase in backend load (10-20%) can dramatically improve tail latency (50-90% reduction at p99)**.

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Retry After Timeout

```
// Incorrect: Wait for timeout, then retry
async function callPaymentService(data: Payment): Promise<Result> {
  try {
    return await withTimeout(paymentService.charge(data), 2000);
  } catch (timeoutError) {
    console.log('Timeout, retrying...');
    return await paymentService.charge(data);  // Retry after timeout
  }
}
```

Why it seems reasonable:

- Standard error handling pattern
- Only retries on timeout (avoids unnecessary load)
- Simple to implement

How it breaks:

- User waits 2 seconds (timeout) before retry even starts
- Total latency: 2000ms (timeout) + 50ms (retry) = 2050ms
- p99 latency actually **worse** than without retry
- Timeout is arbitrary (too short = false positives, too long = doesn't help)
- For critical requests (payment, checkout), 2s+ latency is unacceptable

Production symptoms:

- p99 latency spikes to timeout value (2s)
- User complaints about slowness despite "successful" retries
- Retry logs show most retries succeed instantly (indicates first request was just slow, not failed)
- Conversion rate drops during latency spikes

❌ Incorrect Approach #2: Parallel Requests Without Coordination

```
// Incorrect: Issue requests to all replicas, race them
async function callService(data: Request): Promise<Result> {
  const replicas = ['replica1', 'replica2', 'replica3'];

  const promises = replicas.map(replica =>
    fetch(`https://${replica}.api.com/endpoint`, {
      method: 'POST',
      body: JSON.stringify(data),
    })
  );

  // Return first successful response
  return Promise.race(promises);
}
```

Why it seems reasonable:

- Guarantees fast response (fastest replica wins)
- Simple Promise.race pattern
- No timeout configuration needed

How it breaks:

- Sends 3× traffic to backend for every request (300% load)
- All replicas do work, only one result is used (waste)
- For writes (payment, order creation), creates duplicates
- Losing requests continue running, holding resources
- Database sees 3× normal query load

Production symptoms:

- Backend CPU 3× higher than necessary
- Connection pool exhaustion (3× connections per request)
- Duplicate charges, duplicate orders (writes are not idempotent)
- Database slow query log shows 3× identical queries
- 3× increase in cloud infrastructure costs

❌ Incorrect Approach #3: Fixed-Delay Hedge

```
// Incorrect: Always hedge after fixed delay
async function callServiceWithHedge(data: Request): Promise<Result> {
  const primary = callReplica('replica1', data);

  // Always issue hedge after 100ms
  const hedge = delay(100).then(() => callReplica('replica2', data));

  return Promise.race([primary, hedge]);
}

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

Why it seems reasonable:

- Hedges all requests after predictable delay
- Simple to implement
- Improves tail latency by issuing duplicate

How it breaks:

- Issues hedge even when primary is fast
- If p50 latency is 50ms, 50% of requests issue unnecessary hedge
- Backend load increases by 50% (half of requests send duplicate)
- Slow replicas receive same proportion of traffic (hedge doesn't prefer fast replicas)
- Fixed delay doesn't adapt to changing latency

Production symptoms:

- Backend traffic 150% of normal (50% hedges)
- Hedge rate is 100% (every request hedges)
- Most hedges are cancelled because primary completes first (wasted work)
- During low-latency periods, hedging is pure overhead

❌ Incorrect Approach #4: Hedging Without Cancellation

```
// Incorrect: Issue hedge but don't cancel losing request
async function hedge(data: Request): Promise<Result> {
  const primary = callReplica('replica1', data);
  const backup = delay(95).then(() => callReplica('replica2', data));

  // Promise.race returns first, but both continue running
  const result = await Promise.race([primary, backup]);
  return result;
  // Primary and backup both complete, holding connections
}
```

Why it seems reasonable:

- Uses standard Promise.race
- Gets first result quickly

- Simple code

How it breaks:

- Losing request continues to completion, consuming resources
- Holds database connections, HTTP connections, memory
- For writes, both requests may succeed (duplicate side effects)
- Over time, accumulates thousands of "zombie" requests
- Connection pool exhaustion

Production symptoms:

- Connection pool grows to maximum despite low actual QPS
- Memory usage grows unbounded
- Logs show requests completing after response was already sent
- Duplicate database writes (double charges, double inserts)
- File descriptor exhaustion

3. Correct Mental Model (How It Actually Works)

Hedged requests work by issuing duplicate requests at percentile-based delay:

```
t=0ms:    Primary request starts
          ↓
t=0-50ms: Wait...
          ↓
t=50ms:   Check if primary completed
          ├─ Yes → return result (no hedge)
          └─ No → issue hedge request (primary is slow)
                  ↓
                  Both requests race
                  ↓
                  First to complete wins
                  ↓
                  Cancel losing request
```

Key Properties

1. Adaptive hedge timing:

- Hedge delay = p95 or p99 latency of primary
- If p95 is 50ms, hedge after 50ms
- Only 5% of requests are hedged (those slower than p95)
- 95% of requests complete before hedge is issued

2. Cancellation:

- When first request completes, cancel the other
- Use AbortController or equivalent
- Prevent duplicate side effects
- Free up resources immediately

3. Result selection:

- Take first successful response

- If first response is error, wait for second
- If both error, propagate error

4. Idempotency requirement:

- Hedging only safe for idempotent operations
- Reads: always safe
- Writes: require idempotency tokens

Load-Latency Tradeoff

```
Without hedging:
- Backend load: 100%
- p99 latency: 2000ms

With hedging (p95 delay):
- Backend load: 105% (5% of requests hedged)
- p99 latency: 100ms (20× improvement)

Tradeoff: 5% more load → 95% better tail latency
```

When Hedge Fires

```
Fast request (p50): 50ms
 Primary: ▓▓▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ (completes before hedge)
 Hedge:   ──────────── (not issued)
 Result: 50ms latency, no hedge

Slow request (p99): 2000ms
 Primary: ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ (slow)
 Hedge:   ──────────▓▓ (issued at 50ms, completes at 100ms)
 Result: 100ms latency (hedge wins)
```

4. Correct Design & Algorithm

Core Algorithm

```
function hedgedRequest(requestFn, p95Latency):
    primary = start request
    hedgeTimer = setTimeout(p95Latency)

    winner = race([
        primary,
        hedgeTimer.then(() => start hedged request)
    ])

    on winner complete:
        cancel loser
        return winner result
```

Percentile Tracking

Hedge delay should be adaptive based on recent latency:

```
Track last 1000 request latencies
Calculate p95 latency every 10 seconds
Use p95 as hedge delay
Update delay as latency changes
```

Idempotency Handling

For writes, include idempotency token:

```
idempotencyKey = generateUUID()
primary = callService(data, idempotencyKey)
hedge = callService(data, idempotencyKey)  // Same key

Backend deduplicates based on idempotencyKey
```

5. Full Production-Grade Implementation

```typescript
interface HedgeOptions {
  hedgeDelay?: number;
  hedgePercentile?: number;
  enableAdaptive?: boolean;
  maxHedgeRate?: number;
  idempotencyKeyFn?: () => string;
}

interface HedgeMetrics {
  totalRequests: number;
  hedgedRequests: number;
  primaryWins: number;
  hedgeWins: number;
  bothFailed: number;
  currentP95: number;
}

class HedgedRequestExecutor {
  private latencies: number[] = [];
  private readonly maxLatencySamples = 1000;
  private metrics: HedgeMetrics = {
    totalRequests: 0,
    hedgedRequests: 0,
    primaryWins: 0,
    hedgeWins: 0,
    bothFailed: 0,
    currentP95: 100,
  };

  /**
   * Execute request with hedging to reduce tail latency.
   *
```

```
   * Issues a duplicate request if primary is slow (exceeds p95 latency).
   * Takes first successful response, cancels the other.
   *
   * WARNING: Only use for idempotent operations (reads or writes with idempotency
tokens).
   */
  async execute<T>(
    requestFn: (signal: AbortSignal, idempotencyKey?: string) => Promise<T>,
    options: HedgeOptions = {}
  ): Promise<T> {
    const {
      hedgeDelay,
      hedgePercentile = 95,
      enableAdaptive = true,
      maxHedgeRate = 0.2,
      idempotencyKeyFn,
    } = options;

    this.metrics.totalRequests++;

    // Determine hedge delay
    const delay = hedgeDelay ??
      (enableAdaptive ? this.getPercentileLatency(hedgePercentile) : 100);

    // Check hedge rate limiting
    const currentHedgeRate = this.metrics.hedgedRequests /
this.metrics.totalRequests;
    const shouldSkipHedge = currentHedgeRate > maxHedgeRate;

    if (shouldSkipHedge) {
      console.warn('[Hedge] Skipping hedge due to high hedge rate:',
currentHedgeRate);
      return this.executeWithoutHedge(requestFn);
    }

    const startTime = Date.now();
    const idempotencyKey = idempotencyKeyFn?.();

    // Create abort controllers for cancellation
    const primaryController = new AbortController();
    const hedgeController = new AbortController();

    // Start primary request
    const primaryPromise = this.wrapRequest(
      'primary',
      requestFn(primaryController.signal, idempotencyKey),
      primaryController,
      hedgeController,
      startTime
    );

    // Start hedge timer
```

```
    const hedgePromise = this.sleep(delay).then(async () => {
      // Check if primary already completed
      if (primaryController.signal.aborted) {
        return Promise.reject(new Error('Primary completed before hedge'));
      }

      this.metrics.hedgedRequests++;
      console.log(`[Hedge] Issuing hedge after ${delay}ms`);

      return this.wrapRequest(
        'hedge',
        requestFn(hedgeController.signal, idempotencyKey),
        hedgeController,
        primaryController,
        startTime
      );
    });

    try {
      const result = await Promise.race([primaryPromise, hedgePromise]);
      return result;
    } catch (error) {
      // Both failed or only primary failed before hedge
      this.metrics.bothFailed++;
      throw error;
    } finally {
      // Ensure both controllers are aborted
      primaryController.abort();
      hedgeController.abort();
    }
  }

  /**
   * Wrap request with cancellation and metrics
   */
  private async wrapRequest<T>(
    source: 'primary' | 'hedge',
    promise: Promise<T>,
    myController: AbortController,
    otherController: AbortController,
    startTime: number
  ): Promise<T> {
    try {
      const result = await promise;
      const latency = Date.now() - startTime;

      // I won - cancel the other request
      otherController.abort();

      // Record metrics
      if (source === 'primary') {
        this.metrics.primaryWins++;
```

```
    } else {
      this.metrics.hedgeWins++;
      console.log(`[Hedge] Hedge won, latency: ${latency}ms`);
    }

    this.recordLatency(latency);
    return result;
  } catch (error) {
    // Check if I was cancelled (the other request won)
    if (myController.signal.aborted) {
      throw new Error(`${source} cancelled`);
    }
    throw error;
  }
}

/**
 * Execute without hedging (fallback for high hedge rate)
 */
private async executeWithoutHedge<T>(
  requestFn: (signal: AbortSignal) => Promise<T>
): Promise<T> {
  const controller = new AbortController();
  const startTime = Date.now();

  try {
    const result = await requestFn(controller.signal);
    const latency = Date.now() - startTime;
    this.recordLatency(latency);
    return result;
  } finally {
    controller.abort();
  }
}

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

  // Update p95 periodically
  if (this.latencies.length % 100 === 0) {
    this.metrics.currentP95 = this.getPercentileLatency(95);
  }
}

/**
 * Calculate percentile latency from recent samples
```

```
   */
  private getPercentileLatency(percentile: number): number {
    if (this.latencies.length === 0) {
      return 100; // Default
    }

    const sorted = [...this.latencies].sort((a, b) => a - b);
    const index = Math.ceil((percentile / 100) * sorted.length) - 1;
    return sorted[Math.max(0, index)];
  }

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

  /**
   * Get current metrics for monitoring
   */
  getMetrics(): HedgeMetrics {
    return { ...this.metrics };
  }

  /**
   * Reset metrics
   */
  resetMetrics(): void {
    this.metrics = {
      totalRequests: 0,
      hedgedRequests: 0,
      primaryWins: 0,
      hedgeWins: 0,
      bothFailed: 0,
      currentP95: this.metrics.currentP95,
    };
  }
}

// Example usage with payment service
const hedgeExecutor = new HedgedRequestExecutor();

async function chargePayment(payment: Payment): Promise<PaymentResult> {
  return hedgeExecutor.execute(
    async (signal, idempotencyKey) => {
      const response = await fetch('https://api.payment.com/charge', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Idempotency-Key': idempotencyKey || crypto.randomUUID(),
        },
        body: JSON.stringify(payment),
        signal,
      });
```

```
      if (!response.ok) {
        throw new Error(`Payment failed: ${response.status}`);
      }

      return response.json();
    },
    {
      enableAdaptive: true,
      hedgePercentile: 95,
      maxHedgeRate: 0.1, // Max 10% hedge rate
      idempotencyKeyFn: () => crypto.randomUUID(),
    }
  );
}

// Example usage with database query (read-only)
async function getUserData(userId: string): Promise<User> {
  return hedgeExecutor.execute(
    async (signal) => {
      // Query multiple replicas, hedge if primary is slow
      const replicas = ['replica1', 'replica2'];
      const replica = replicas[Math.floor(Math.random() * replicas.length)];

      const query = db.query(
        `SELECT * FROM users WHERE id = ?`,
        [userId],
        { signal }
      );

      return query;
    },
    {
      enableAdaptive: true,
      hedgePercentile: 99, // Hedge slowest 1%
    }
  );
}

// Monitoring endpoint
app.get('/metrics/hedge', (req, res) => {
  const metrics = hedgeExecutor.getMetrics();

  res.json({
    hedge_rate: metrics.hedgedRequests / metrics.totalRequests,
    primary_win_rate: metrics.primaryWins / metrics.totalRequests,
    hedge_win_rate: metrics.hedgeWins / metrics.totalRequests,
    current_p95_ms: metrics.currentP95,
    total_requests: metrics.totalRequests,
  });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Critical User-Facing Operations

Payment, checkout, authentication where tail latency directly impacts conversion:

```
app.post('/api/checkout', async (req, res) => {
  try {
    const result = await hedgeExecutor.execute(
      async (signal, idempotencyKey) => {
        return await paymentService.processCheckout(req.body, {
          signal,
          idempotencyKey,
        });
      },
      { hedgePercentile: 95, idempotencyKeyFn: () => req.body.orderId }
    );

    res.json({ success: true, result });
  } catch (error) {
    res.status(500).json({ error: 'Checkout failed' });
  }
});
```

Why this works:

- Checkout latency directly impacts conversion rate
- 95% of requests complete before hedge (low overhead)
- 5% slow requests get hedge (dramatic improvement)
- User experiences sub-100ms checkout instead of 2s

Pattern 2: Database Queries Across Replicas

Reading from database replicas where some may be slow:

```
async function getProduct(productId: string): Promise<Product> {
  return hedgeExecutor.execute(
    async (signal) => {
      // Randomly select replica
      const replica = selectReplica();
      return await replica.query(
        'SELECT * FROM products WHERE id = ?',
        [productId],
        { signal }
      );
    },
    { hedgePercentile: 99 }
  );
}
```

Why this works:

- Replicas may have varying latency (replication lag, load, network)
- Hedge automatically routes around slow replica
- No manual replica health checking needed
- Self-healing: slow replica excluded via hedge wins

Pattern 3: External API Calls

Third-party APIs with unpredictable latency:

```
async function fetchWeatherData(city: string): Promise<Weather> {
  return hedgeExecutor.execute(
    async (signal) => {
      const response = await fetch(
        `https://api.weather.com/v1/${city}`,
        { signal }
      );
      return response.json();
    },
    { hedgePercentile: 95, maxHedgeRate: 0.15 }
  );
}
```

Why this works:

- External APIs have variable latency (network, load)
- Can't control API performance
- Hedging reduces user-perceived latency
- maxHedgeRate prevents excessive API usage

Pattern 4: Microservice Communication

Service-to-service calls where tail latency compounds:

```
async function aggregateDashboard(userId: string): Promise<Dashboard> {
  const [profile, orders, recommendations] = await Promise.all([
    hedgeExecutor.execute(
      signal => profileService.get(userId, { signal }),
      { hedgePercentile: 95 }
    ),
    hedgeExecutor.execute(
      signal => orderService.getRecent(userId, { signal }),
      { hedgePercentile: 95 }
    ),
    hedgeExecutor.execute(
      signal => recommendationService.get(userId, { signal }),
      { hedgePercentile: 95 }
    ),
  ]);

  return { profile, orders, recommendations };
}
```

Why this works:

- Fan-out to multiple services amplifies tail latency
- Each service benefits from hedging independently
- Aggregate latency improves dramatically
- User sees fast dashboard load

7. Failure Modes & Edge Cases

Duplicate Side Effects from Write Operations

Problem: Hedged write request both succeed, creating duplicate charges/orders.

Symptoms:

- Users charged twice
- Duplicate orders in database
- Idempotency not implemented

Mitigation:

- Always include idempotency token for writes
- Backend deduplicates based on token
- Both requests succeed but only one side effect occurs

```
async function createOrder(order: Order): Promise<string> {
  const idempotencyKey = crypto.randomUUID();

  return hedgeExecutor.execute(
    async (signal) => {
      return await orderService.create(order, {
        signal,
        idempotencyKey,  // Backend deduplicates
      });
    },
    { idempotencyKeyFn: () => idempotencyKey }
  );
}
```

Hedge Rate Spiraling

Problem: High hedge rate creates backend overload, which increases latency, which increases hedge rate further.

Symptoms:

- Hedge rate climbs from 5% to 50%+
- Backend load doubles
- Latency increases, making problem worse

Mitigation:

- Set maxHedgeRate (e.g., 20%)
- Circuit breaker: disable hedging when backend is overloaded

- Adaptive hedge delay: increase delay when hedge rate is high

```
const currentHedgeRate = hedgedRequests / totalRequests;
if (currentHedgeRate > maxHedgeRate) {
  // Skip hedging, use primary only
  return executeWithoutHedge(requestFn);
}
```

Connection Pool Exhaustion

Problem: Hedged requests double connection usage, exhaust pool.

Symptoms:

- Connection pool max reached
- New requests queued or rejected
- Pool size: 100, hedged requests: 200 connections

Mitigation:

- Size connection pool for hedge overhead (120% of normal)
- Cancel losing request immediately (free connection)
- Monitor active connections

Losing Request Not Cancelled

Problem: Cancellation not implemented, both requests complete.

Symptoms:

- Connection pool grows despite low QPS
- Both requests appear in logs
- Duplicate side effects

Mitigation:

- Use AbortController for HTTP requests
- For database queries, check signal before executing
- Ensure finally block aborts both controllers

```
finally {
  primaryController.abort();
  hedgeController.abort();
}
```

Fixed Hedge Delay Too Aggressive

Problem: Hedge delay set too low (e.g., 10ms), hedges issued for most requests.

Symptoms:

- Hedge rate 80%+
- Backend load 180%
- No latency improvement (primary usually wins)

Mitigation:

- Use adaptive delay based on p95 or p99
- Monitor hedge rate, adjust delay if >20%
- Start conservative (p99), tune down to p95

8. Performance Characteristics & Tradeoffs

Latency Impact

Without hedging:

- p50: 50ms
- p95: 200ms
- p99: 2000ms
- p99.9: 10000ms

With hedging (p95 delay):

- p50: 50ms (unchanged)
- p95: 200ms (unchanged)
- p99: 250ms (8× improvement)
- p99.9: 500ms (20× improvement)

Why: Slowest 5% of requests get hedge, which completes faster than slow primary.

Load Impact

Hedge rate vs load increase:

- p99 hedging (1% hedge rate): +1% load
- p95 hedging (5% hedge rate): +5% load
- p90 hedging (10% hedge rate): +10% load

Typical: p95 hedging adds 5% load for 10× tail latency improvement.

Cost-Benefit Analysis

```
5% more backend load = 5% more infrastructure cost
10× better p99 latency = 10-20% higher conversion rate

ROI: 5% cost increase → 15% revenue increase
```

For user-facing services, tail latency improvement pays for itself.

When Hedge Fires

```
Distribution of request latencies:
- 50% complete in <50ms (no hedge)
- 45% complete in 50-200ms (no hedge)
- 4% complete in 200ms-1s (hedge fired but primary wins)
- 1% complete in >1s (hedge wins)

Result:
- 95% of requests: no hedge overhead
```

```
- 4% of requests: hedge issued but cancelled (minimal overhead)
- 1% of requests: hedge wins (dramatic improvement)
```

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

Mistake 1: Hedging Write Operations Without Idempotency

Why engineers do it: "Hedging improves latency for all operations."

What breaks: Duplicate charges, duplicate orders, data corruption.

```
// WRONG
async function createOrder(order: Order) {
  return hedgeExecutor.execute(
    () => orderService.create(order)  // No idempotency key!
  );
}
// Both primary and hedge succeed → 2 orders created
```

Detection: Users report double charges, duplicate orders in database.

Fix: Include idempotency token for all writes.

```
// Correct
async function createOrder(order: Order) {
  return hedgeExecutor.execute(
    (signal, idempotencyKey) => {
      return orderService.create(order, { idempotencyKey });
    },
    { idempotencyKeyFn: () => crypto.randomUUID() }
  );
}
```

Mistake 2: Not Cancelling Losing Request

Why engineers do it: Forget to implement cancellation, or assume Promise.race cancels automatically.

What breaks: Connection pool exhaustion, duplicate side effects.

```
// WRONG
const result = await Promise.race([primary, hedge]);
return result;
// Both requests continue running!
```

Detection: Connection pool maxed out, memory leak, logs show completed requests after response sent.

Fix: Use AbortController and cancel explicitly.

```
// Correct
try {
  const result = await Promise.race([primary, hedge]);
```

```
  return result;
} finally {
  primaryController.abort();
  hedgeController.abort();
}
```

Mistake 3: Fixed Hedge Delay Instead of Adaptive

Why engineers do it: Simpler to configure.

What breaks: Either hedges too often (high load) or too rarely (no benefit).

```
// WRONG: Fixed 50ms delay
{ hedgeDelay: 50 }
// If p99 is 2000ms, hedge fires at 2.5% (too often)
// If p99 is 20ms, hedge never helps (too late)
```

Detection: Hedge rate is too high (>20%) or tail latency doesn't improve.

Fix: Use adaptive delay based on percentiles.

```
// Correct
{ enableAdaptive: true, hedgePercentile: 95 }
```

Mistake 4: Hedging All Operations

Why engineers do it: "If hedging helps critical path, use everywhere."

What breaks: 2× backend load for non-critical operations.

```
// WRONG: Hedge analytics query
async function runAnalytics() {
  return hedgeExecutor.execute(() => db.query('SELECT ...'));
}
// Analytics is not user-facing, doesn't need tail latency optimization
```

Detection: Backend load doubled, hedge rate >50%.

Fix: Only hedge user-facing, latency-sensitive operations.

Mistake 5: Not Monitoring Hedge Rate

Why engineers do it: Deploy hedging, don't track metrics.

What breaks: Hedge rate spirals, backend overload.

Detection: Backend CPU 2×, hedge rate 80%.

Fix: Alert on hedge rate >20%, disable hedging if spiraling.

```
if (metrics.hedgedRequests / metrics.totalRequests > 0.2) {
  console.error('[Hedge] Hedge rate too high, circuit breaker opened');
```

```
  disableHedging();
}
```

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

Anti-Pattern 1: Background Jobs

Don't hedge async operations where latency doesn't matter:

```
// WRONG
async function processEmailQueue() {
  const email = await queue.pop();
  await hedgeExecutor.execute(() => sendEmail(email));
}
// Email sending is async, user doesn't wait, hedging wastes resources
```

Anti-Pattern 2: Write Operations Without Idempotency

Never hedge writes without idempotency support:

```
// NEVER
await hedgeExecutor.execute(() =>
  db.query('INSERT INTO orders VALUES (?)', [order])
);
// Creates duplicate orders
```

Anti-Pattern 3: Low-Latency Operations

Don't hedge operations faster than hedge delay:

```
// WRONG: Operation is 5ms, hedge delay is 50ms
await hedgeExecutor.execute(() => cache.get(key));
// Hedge never fires, pure overhead
```

Anti-Pattern 4: Operations with Side Effects

Don't hedge operations that can't be cancelled safely:

```
// WRONG
await hedgeExecutor.execute(() => {
  logToFile(message);
  incrementCounter();
  return fetchData();
});
// Both requests execute side effects
```

11. Related Concepts (With Contrast)

Retry After Timeout

Difference: Retry waits for timeout before second attempt. Hedging issues second attempt proactively.

Latency:

- Retry: timeout + retry_latency (e.g., 2000ms + 50ms = 2050ms)
- Hedge: max(primary, hedge) (e.g., max(2000ms, 100ms) = 100ms)

Request Coalescing (Singleflight)

Difference: Coalescing deduplicates identical concurrent requests. Hedging duplicates requests to reduce latency.

Opposite goals: Coalescing reduces load, hedging increases load (but improves latency).

When to combine: Use both. Coalesce identical requests, hedge unique requests.

Load Balancing

Difference: Load balancing distributes requests across replicas. Hedging issues duplicates to multiple replicas.

When to combine: Load balancing + hedging = intelligent routing. Hedge to multiple replicas, take fastest.

Circuit Breaker

Difference: Circuit breaker stops calling failing services. Hedging improves latency of working services.

When to combine: Circuit breaker for failures, hedging for slowness.

12. Production Readiness Checklist

Metrics to Monitor

- [ ] Hedge rate (hedged requests / total requests)
- [ ] Primary win rate
- [ ] Hedge win rate
- [ ] Current p95/p99 latency (adaptive delay)
- [ ] Backend load increase
- [ ] Connection pool utilization

Logging

- [ ] Log when hedge is issued with latency
- [ ] Log which request won (primary vs hedge)
- [ ] Log hedge rate periodically
- [ ] Log when hedge rate exceeds threshold

Configuration

- [ ] Set hedgePercentile (start with p99, tune to p95)
- [ ] Set maxHedgeRate (recommend 0.1-0.2)
- [ ] Enable adaptive delay

- [ ] Configure idempotency for writes

Load Testing

- [ ] Test latency improvement (p99 before/after)
- [ ] Test backend load increase (should be ~hedge rate)
- [ ] Test with slow replicas (hedge should improve)
- [ ] Test cancellation (losing request should abort)
- [ ] Test idempotency (duplicate requests shouldn't duplicate side effects)

Rollout

- [ ] Deploy to 1% traffic, monitor metrics
- [ ] Verify tail latency improvement
- [ ] Verify backend load acceptable
- [ ] Gradually increase to 100%
- [ ] Have kill switch to disable instantly

Alerting

- [ ] Alert if hedge rate > 20%
- [ ] Alert if backend load > 120%
- [ ] Alert if tail latency doesn't improve
- [ ] Alert if idempotency violations detected