Rate Limiting & Throttling

Overview

Genesys Cloud API has rate limits to ensure fair usage and platform stability. Understanding and respecting these limits is critical for production integrations.

Standard Rate Limit: 600 requests per minute per organization

---

How Rate Limiting Works

Time Windows

Rate limits are enforced in **1-minute rolling windows**:

```
Window 1: 00:00-00:59 → 600 requests allowed
Window 2: 00:01-01:00 → 600 requests allowed
  (overlaps with Window 1)
Window 3: 00:02-01:01 → 600 requests allowed
```

Each minute, the window slides forward. Old requests drop off, new requests are added.

Rate Limit Headers

Every API response includes rate limit information:

```
X-Rate-Limit-Limit:     600      (max requests per window)
X-Rate-Limit-Remaining: 450      (requests left)
X-Rate-Limit-Reset:     1679491234  (Unix timestamp)
```

Example

You make request 150 at timestamp 1679491200

Header: X-Rate-Limit-Remaining: 450

This means: 150 requests made, 450 more allowed before limit.

Detecting Rate Limit Conditions

Proactive Detection (Before hitting limit)

Monitor remaining requests:

```
async function makeRequest(endpoint) {
  const response = await fetch(endpoint);
  const remaining = parseInt(
    response.headers.get('X-Rate-Limit-Remaining')
  );
  const limit = parseInt(response.headers.get('X-Rate-Limit-Limit'));

  const percentRemaining = (remaining / limit) * 100;

  if (percentRemaining < 20) {
    console.warn('⚠ Only 20% of rate limit remaining. Slowing down...');
    // Reduce request frequency
  }

  if (percentRemaining < 5) {
    console.error('□□ Critical: <5% remaining. STOP requests immediately.');
    // Pause all requests
  }

  return response;
}
```

Reactive Detection (After hitting limit)

Watch for 429 responses:

```
async function makeRequest(endpoint) {
  const response = await fetch(endpoint);

  if (response.status === 429) {
    console.error('⛔ Rate limit exceeded!');

    const retryAfter = response.headers.get('Retry-After');
    const resetTime = response.headers.get('X-Rate-Limit-Reset');

    if (retryAfter) {
      // API tells you when to retry
      const seconds = parseInt(retryAfter);
      console.warn(`Wait ${seconds} seconds`);
    } else if (resetTime) {
      // Calculate wait time from reset timestamp
      const now = Math.floor(Date.now() / 1000);
      const waitSeconds = parseInt(resetTime) - now;
      console.warn(`Wait until ${new Date(resetTime * 1000).toISOString()}`);
    }
  }

  return response;
}
```

Strategy 1: Exponential Backoff (Reactive)

Only for when you hit the limit:

```
async function requestWithBackoff(endpoint, maxAttempts = 4) {
  const delays = [3000, 9000, 27000, 300000]; // 3s, 9s, 27s, 5min
```

```
  for (let attempt = 0; attempt < maxAttempts; attempt++) {
    const response = await fetch(endpoint);

    if (response.status !== 429) {
      return response; // Success (or other error)
    }

    // Rate limited
    if (attempt >= maxAttempts - 1) {
      throw new Error('Rate limit exceeded after max retries');
    }

    const delayMs = delays[attempt];
    console.warn(`Rate limited. Waiting ${delayMs / 1000}s...`);
    await sleep(delayMs);
  }
}
```

Strategy 2: Request Throttling (Proactive)

Limit request rate to stay well below limit:

```
class ThrottledClient {
  constructor(requestsPerSecond = 8) {
    // 8 requests/sec = 480/min (80% of 600 limit)
    this.requestsPerSecond = requestsPerSecond;
    this.minIntervalMs = 1000 / requestsPerSecond;
    this.lastRequestTime = 0;
  }

  async makeRequest(endpoint) {
    const now = Date.now();
    const timeSinceLastRequest = now - this.lastRequestTime;

    if (timeSinceLastRequest < this.minIntervalMs) {
```

```
      const waitMs = this.minIntervalMs - timeSinceLastRequest;
      await sleep(waitMs);
    }

    this.lastRequestTime = Date.now();
    return fetch(endpoint);
  }
}

// Usage
const client = new ThrottledClient(8); // 8 req/sec (safe limit)
await client.makeRequest('/contacts');
await client.makeRequest('/contacts');
// These will be spaced 125ms apart
```

Strategy 3: Request Queue with Batch Processing

Buffer requests and process in batches:

```
class RequestQueue {
  constructor(batchSize = 50, batchIntervalMs = 5000) {
    this.queue = [];
    this.batchSize = batchSize;
    this.batchIntervalMs = batchIntervalMs;
    this.processing = false;
  }

  async add(endpoint, body) {
    return new Promise((resolve, reject) => {
      this.queue.push({ endpoint, body, resolve, reject });

      if (!this.processing) {
        this.processBatch();
      }
    });
```

```
  }

  async processBatch() {
    this.processing = true;

    while (this.queue.length > 0) {
      const batch = this.queue.splice(0, this.batchSize);

      // Process batch in parallel (but within rate limit)
      const promises = batch.map(item =>
        fetch(item.endpoint, { body: JSON.stringify(item.body) })
          .then(res => item.resolve(res))
          .catch(err => item.reject(err))
      );

      await Promise.all(promises);

      // Wait between batches
      if (this.queue.length > 0) {
        console.log(`Processed ${batch.length} requests. Waiting ${this.batchIntervalMs}ms...`);
        await sleep(this.batchIntervalMs);
      }
    }

    this.processing = false;
  }
}

// Usage
const queue = new RequestQueue(50, 5000); // 50 requests per 5 seconds

for (const contact of millionContacts) {
  queue.add('/contacts', contact);
}
```

Strategy 4: Bulk Operations

Most efficient: use bulk endpoints instead of individual requests.

Without Bulk (SLOW - 100 requests)

```
// Creating 100 contacts individually
for (const contact of contacts) {
  await fetch('/contacts', {
    method: 'POST',
    body: JSON.stringify(contact)
  });
}
// Uses 100 API calls!
```

With Bulk (FAST - 1 request)

```
// Creating 100 contacts in one batch
await fetch('/contacts/bulk', {
  method: 'POST',
  body: JSON.stringify({
    contacts: contacts  // Array of 100
  })
});
// Uses 1 API call!
```

Benefit: 100x reduction in API calls.

Strategy 5: Caching

Avoid repeated requests for same data:

```
class CachedClient {
  constructor(cacheTtlSeconds = 3600) {
    this.cache = new Map();
    this.cacheTtlSeconds = cacheTtlSeconds;
  }
```

```
  async getContact(contactId) {
    const cacheKey = `contact:${contactId}`;

    // Check cache
    const cached = this.cache.get(cacheKey);
    if (cached && Date.now() - cached.timestamp < this.cacheTtlSeconds * 1000) {
      console.log(`Cache hit: ${cacheKey}`);
      return cached.data;
    }

    // Not cached, fetch from API
    console.log(`Cache miss: ${cacheKey}`);
    const response = await fetch(`/contacts/${contactId}`);
    const data = await response.json();

    // Store in cache
    this.cache.set(cacheKey, { data, timestamp: Date.now() });

    return data;
  }

  clearCache() {
    this.cache.clear();
  }
}

// Usage
const client = new CachedClient(3600); // Cache for 1 hour
const contact1 = await client.getContact('c1'); // API call
const contact1Again = await client.getContact('c1'); // Cache hit!
```

Strategy 6: WebSocket Subscriptions (Real-time, Low Overhead)

Instead of polling, use WebSocket for real-time updates:

Polling (Uses many requests)

```
// Poll every 10 seconds = 6 requests/min per user
setInterval(async () => {
  const status = await fetch('/users/user-123?expand=presence');
  // Expensive for many users!
}, 10000);
```

WebSocket (1 connection)

```
// Single WebSocket connection for real-time updates
const ws = new WebSocket('wss://...');

ws.onmessage = (event) => {
  const { userId, presence } = JSON.parse(event.data);
  console.log(`User ${userId} is now ${presence}`);
};

// Can handle thousands of users on one connection!
```

Benefit: Eliminates polling entirely.

Rate Limit Calculation

Scenario 1: Simple API calls

```
10,000 contacts to sync
1 API call per contact = 10,000 calls
Rate limit: 600/minute
Time needed: 10,000 / 600 = 16.67 minutes

Strategy: Use bulk endpoint (1 call) or batch in 600-request chunks
```

Scenario 2: Contact lookup every second

100 concurrent agents

Each looks up contact every second

= 100 requests/second = 6,000/minute

Rate limit: 600/minute

You'd exceed limit 10x over!

Strategy: Add 100ms delay between requests

= 10 requests/sec = 600/minute (perfect!)

Scenario 3: Mixed workload

- Sync contacts: 1000 requests (use bulk)
- Agent presence updates: 100 requests/minute (natural rate)
- Search queries: variable (depends on usage)
- Reporting: 50 requests/day (batch at night)

Total: 1000 + 100 + variable + ~2 = Safe if variable < 500/min

---

Monitoring & Alerting

Track rate limit over time

```
class RateLimitMonitor {
  constructor() {
    this.history = [];
  }

  record(remaining, limit) {
    const now = new Date();
    const percentUsed = ((limit - remaining) / limit) * 100;

    this.history.push({ now, remaining, percentUsed });
```

```
    // Alert if trend is concerning
    if (this.history.length > 10) {
      const recentAverage = this.history
        .slice(-10)
        .reduce((sum, h) => sum + h.percentUsed, 0) / 10;

      if (recentAverage > 80) {
        console.warn('⚠ Average usage >80%. Trending toward limit!');
      }
    }
  }

  report() {
    const avg = this.history.reduce((sum, h) => sum + h.percentUsed, 0) / this.history.length;
    const max = Math.max(...this.history.map(h => h.percentUsed));

    console.log(`
      Rate Limit Usage Report:
      Average: ${avg.toFixed(1)}%
      Peak: ${max.toFixed(1)}%
      Samples: ${this.history.length}
    `);
  }
}
```

Recommended Approach: Tiered Strategy

Tier 1 - Proactive (Always do this)

- Monitor `X-Rate-Limit-Remaining` on every request
- Implement throttling (8 req/sec = 480/min, safe)
- Batch operations when possible

Tier 2 - Reactive (If approaching limit)

- Reduce request frequency further
- Implement caching
- Queue requests instead of fire-and-forget

Tier 3 - Emergency (If hitting limit)

- Implement exponential backoff
- Stop new requests
- Alert operations team

Best Practices

1. **Monitor proactively** - Don't wait for 429 errors
2. **Use bulk endpoints** - Single call for multiple records
3. **Implement throttling** - Spread requests evenly
4. **Cache aggressively** - Don't re-fetch same data
5. **Use WebSockets** - For real-time subscriptions
6. **Batch requests** - Process in groups, not individually
7. **Set timeouts** - Don't retry forever
8. **Alert operations** - Know when limit is approached

Common Mistakes

 Fire-and-forget requests - No rate limit awareness
 Monitor headers, throttle proactively

 Polling instead of WebSocket - Wastes requests
 Use WebSocket for real-time data

 Individual API calls in loop - 1000 calls instead of 1
 Use bulk endpoints, batch in groups

 Ignore rate limit warnings - Hit limit unexpectedly
 Monitor, reduce frequency before limit

 Unlimited retry - Keep hammering API
 Exponential backoff, respect Retry-After

Related Topics

- Chapter 11: Error Handling & Retry Strategy
- Chapter 11: API Endpoints Reference
- Chapter 5: Data Actions (rate limiting in flows)

Revision #1
Created 15 March 2026 00:42:24 by Cesar Gzz
Updated 15 March 2026 00:42:35 by Cesar Gzz