Architectural Resilience in Multimodal AI Integration: A Deep Dive into Gemini 2.5 Flash Image, Cloudflare Edge, and React 19

1. Introduction: The Convergence of Constraints in Modern AI Systems

The integration of generative artificial intelligence into modern web application architectures represents a fundamental shift in software engineering paradigms. We are moving from a deterministic era, where inputs reliably produce identical outputs via predictable logical paths, into a probabilistic era governed by stochastic models, fluctuating inference costs, and opaque, dynamic resource constraints. The specific challenge encountered—a 429 RESOURCE_EXHAUSTED error when attempting a single image editing operation using Google's gemini-2.5-flash-image model within a Cloudflare Pages and React 19 environment—is not merely a configuration oversight. Rather, it is a symptomatic manifestation of a complex interplay between distributed edge network topologies, the evolving economics of large language model (LLM) providers, and the strict concurrency patterns of modern frontend frameworks.

To understand why a single request triggers a resource exhaustion event, one must dissect the layers of the technology stack: the frontend lifecycle management of React 19, the request routing mechanics of the Cloudflare edge, the SDK implementation details of @google/genai, and the underlying quota management algorithms employed by Google's Gemini API infrastructure. This report provides an exhaustive technical analysis of these components, establishing a "correct knowledge" baseline for implementing production-grade multimodal AI features in a constrained environment.

The context of this analysis is defined by the significant restructuring of Google's API offerings in late 2025. The era of "loss-leader" strategy—where tech giants subsidized massive free tiers to capture developer mindshare—has effectively concluded. The resulting landscape is one where free-tier access is no longer a guaranteed utility but a volatile, dynamically throttled resource pool. For developers architecting applications on this substrate, particularly those utilizing heavy modalities like image editing, understanding the microscopic details of request lifecycles and token consumption is no longer optional; it is a prerequisite for system stability.

2. The Macro-Economic and Technical Landscape of

Gemini API Limits

The first step in diagnosing the 429 error is to contextualize the operating environment of the Gemini API as of late 2025. The error message RESOURCE_EXHAUSTED is often misinterpreted as a simple counter-reset notification. However, in the context of Google's evolved infrastructure, it represents a multifaceted signal derived from token consumption, request frequency, and regional capacity availability.

2.1 The December 2025 Quota Restructuring

In December 2025, the developer ecosystem experienced a seismic shift in the availability of free computational resources. Historical data indicates that prior to this period, developers utilizing Google's generative models enjoyed relatively generous allowances, often ranging from 250 to 1,000 requests per day (RPD) on Flash-tier models. This allowance was sufficient for robust prototyping, automated testing pipelines, and even low-volume production applications.

However, the rapid commoditization of generative models and the astronomical costs associated with inference—particularly for multimodal tasks involving image generation and processing—forced a correction. Reports and telemetry from the developer community confirm a reduction in free tier capacity by approximately 92% across the board.[2] The specific implications for the gemini-2.5-flash family of models are profound.

The standard gemini-2.5-flash text model saw its quota reduced from ~250 RPD to approximately **20 RPD**.[1] This reduction alone creates a precarious environment for development. A developer refreshing a user interface ten times during a debugging session, with each refresh triggering two API calls (a common behavior in React strict mode, discussed later), would exhaust the entire daily allocation in under five minutes.

For image generation and editing models, the constraints are even more severe. The gemini-2.5-flash-image model (internally codenamed "Nano Banana") is a specialized diffusion-transformer hybrid designed for low-latency visual output. Unlike text inference, which can be batched and optimized via KV caching in the attention layers, image generation requires substantial, dedicated GPU/TPU memory for the diffusion process. Consequently, the effective limit for image generation on free tiers has been reported to be as low as **2 to 10 images per day**.[1] This creates a scenario where the margin for error is effectively zero. A single misconfigured loop or a retry caused by network jitter can trigger a lockout for 24 hours.

2.2 The Mechanics of Quota Enforcement: Token Buckets vs. Leaky Buckets

To understand why the API returns a 429 error instantly, one must examine the algorithm likely used for rate limiting. While Google does not publicly disclose the exact source code of its

rate limiters, industry standard patterns and observed behaviors suggest the use of a hierarchical **Token Bucket** algorithm with additional IP-based reputation filtering.

In a standard Token Bucket system, a user is assigned a "bucket" of tokens. Each API request consumes a specific number of tokens. The bucket refills at a fixed rate. If the bucket is empty, the request is rejected.

- **The Consumption Spike:** Multimodal requests are not created equal. A text prompt might consume 50 tokens. An image generation request, however, consumes a fixed cost—historically **1,290 tokens** per image.[5]
- **The "Burst" Limit:** API quotas often define a burst limit (Requests Per Minute or RPM) alongside a sustained limit (Requests Per Day or RPD). If the free tier allows for 2 RPM, and the application sends three requests in quick succession (e.g., due to a frontend race condition), the third request will fail regardless of the daily allowance remaining.[7]

Crucially, specifically for *image editing*, the cost calculation involves both the input tokens (the reference image provided for editing) and the output tokens (the generated result). Recent updates suggest the input token weight for images was optimized to 258 tokens [8], but the output remains heavy. If the system treats the "editing" operation as a distinct, higher-cost transaction than simple text generation, a single request can consume a disproportionate amount of the minute-level "burst" allowance, triggering an immediate throttle.

2.3 The "Noisy Neighbor" Effect in Serverless Edge Computing

The user's backend environment—Cloudflare Pages Functions—introduces a critical variable into this equation: the network topology of the egress traffic. Cloudflare Pages Functions execute on Cloudflare's global edge network. Unlike a traditional Virtual Private Server (VPS) or a dedicated container which might have a static IP address, edge functions operate on a shared infrastructure.

When a Cloudflare Worker or Pages Function initiates an outbound HTTP request to Google's API, that request exits Cloudflare's network via a NAT (Network Address Translation) gateway. This gateway uses a pool of public IP addresses that are shared among thousands of other Cloudflare customers.[9]

This architecture interacts poorly with IP-based rate limiting strategies often employed for free-tier services.

1. **Shared Reputation:** Google's abuse prevention systems likely track request volume per IP address to prevent denial-of-service attacks and scraping.
2. **The Collision:** If a malicious actor (or simply a high-volume user) on the same Cloudflare node is spamming the Gemini API, the shared egress IP may be flagged or throttled by Google.
3. **The Result:** When the user's legitimate, low-volume request exits that same IP, Google's

gatekeeper rejects it immediately with 429 RESOURCE_EXHAUSTED, attributing the "exhaustion" not to the user's API key, but to the origin IP's aggregated traffic.[10]

This phenomenon explains the user's observation of hitting the limit after sending "only 1 image." In reality, from Google's perspective, the *IP address* sending the image may have sent thousands of requests in the preceding minute. This is a classic "noisy neighbor" problem inherent to serverless edge computing when interacting with rate-limited downstream services.[11]

3. Technical Anatomy of the Gemini 2.5 Flash Image Model

To implement the "correct knowledge" as requested, we must dissect the specific model being invoked. gemini-2.5-flash-image is not merely a text model with a vision encoder; it is a unified multimodal agent capable of native pixel generation. This distinction dictates the structure of the API payload and the handling of the response.

3.1 Model Architecture and Capabilities

The gemini-2.5-flash-image model is engineered for high-throughput, low-latency tasks. It represents a departure from the "Pro" series, utilizing a smaller parameter count optimized for speed—hence the internal moniker "Nano Banana".[12]

Key Specifications:

- **Modality Support:** The model accepts text and image inputs and can output text and images interleaved. This capability is critical for "editing" workflows, where the input is (Image A + Instruction) and the output is (Image B).
- **Token Economics:**
  - **Input Cost:** $0.30 per 1 million tokens. This applies to the text prompt and the base64-encoded reference image.[13]
  - **Output Cost:** $30.00 per 1 million tokens for image output. This is significantly higher than text output, reflecting the computational intensity of diffusion processes.[6]
  - **Fixed Consumption:** A generated image typically accounts for a fixed token count (e.g., 1,290 tokens), simplifying quota calculations compared to variable-length text generation.[5]
- **Latency Profile:** The model aims for sub-4-second generation times, contrasting with the 10-40 second wait times typical of older diffusion models like Midjourney v6.[5]

3.2 The Confusion: generateContent vs. generateImages

A critical source of implementation error lies in the SDK method selection. The @google/genai SDK unifies access to Gemini (multimodal) and Imagen (pure image) models, but the methods

are distinct.

- **The Imagen Path:** Google's dedicated image models (e.g., imagen-3.0) are accessed via generateImages. These models are purely text-to-image generators and do not support the rich reasoning capabilities of Gemini.[15]
- **The Gemini Path:** The gemini-2.5-flash-image model is accessed via the **generateContent** method. Because it is fundamentally a language model that *can* generate images, the request structure must explicitly request image modalities.

Implementation Imperative:
When performing image editing, the developer must use generateContent. Using generateImages with a Gemini model ID will likely result in a 404 Not Found or 400 Invalid Argument error because the endpoint expectations do not match the model's capabilities. Furthermore, the config object passed to generateContent must explicitly include responseModalities:. Without this flag, the model may default to text-only output, describing the edits it would have made rather than performing them.16

3.3 The Mechanics of Image Editing

Image editing in Gemini 2.5 is not a traditional pixel-manipulation process (like Photoshop's programmatic filters). It is a **semantic generation** process.

1. **Ingestion:** The model ingests the base64-encoded reference image. This is tokenized by the vision encoder into a sequence of vector embeddings.
2. **Reasoning:** The text prompt ("Add a hat to the cat") is processed to understand the intent. The model attends to the specific regions of the image embeddings relevant to the prompt.
3. **Generation:** The model generates new image tokens that reconstruct the scene with the requested modifications, effectively "dreaming" a new image that matches both the visual consistency of the input and the semantic constraints of the text.

This process is computationally expensive. The "editing" operation effectively doubles the token load compared to simple generation: the system must process the high-density input tokens of the reference image *and* generate the high-cost output tokens of the result. This multiplier effect contributes significantly to the rapid exhaustion of free-tier quotas.

4. Architectural Analysis of the User's Stack

The user's environment is a modern, high-performance stack: React 19 on the frontend and Cloudflare Pages Functions on the backend. While performant, this combination introduces specific synchronization challenges that can inadvertently trigger rate limits.

4.1 React 19: The Frontend Lifecycle and Double-Invocation

React 19 introduces significant changes to the rendering lifecycle and state management, specifically with the introduction of Concurrent features being enabled by default and the

new "Strict Mode" behaviors.

4.1.1 Strict Mode and the Double-Effect

In a development environment (NODE_ENV=development), React's Strict Mode intentionally invokes certain lifecycle methods and hooks twice.

- **The Mechanism:** To help developers find impure side effects, React will mount, unmount, and remount components. It will also run the body of useEffect hooks twice.
- **The Trap:** If the developer initiates the API call within a useEffect (e.g., "generate image on component load"), React will trigger the API call **twice** in rapid succession.
- **Quota Impact:** For a user with a 20 RPD limit, a single page load in development mode consumes 10% of the daily quota. Five page refreshes exhaust the entire day's allowance.

4.1.2 The useActionState and Form Submission

React 19 encourages the use of useActionState (formerly useFormState) for handling form submissions. This hook integrates with Server Actions or client-side async functions.

- **Button Mashing:** If the "Generate" button does not correctly implement the isPending state provided by useActionState or useTransition, a user might click the button multiple times while waiting for the image to generate.
- **Network Latency:** Given that image generation takes ~3-5 seconds, a user perceiving a lack of feedback might click "Generate" three or four times.
- **Result:** The frontend dispatches multiple parallel requests to the Cloudflare backend. Each request consumes quota. Even if the backend cancels the processing of earlier requests, the initial handshake with the Gemini API may have already decremented the token bucket.

4.2 Cloudflare Pages Functions: The Backend Execution Model

Cloudflare Pages Functions are built on Cloudflare Workers, a serverless platform running on the V8 engine.

4.2.1 Execution Limit and Retries

Cloudflare functions have a default timeout (typically 10ms CPU time, but much longer wall-clock time for I/O). However, the interaction with upstream APIs introduces retry logic.

- **Automatic Retries:** If the fetch request to Google's API fails with a transient error (e.g., a network blip), the Cloudflare runtime or the fetch implementation might attempt a retry depending on the specific configuration or libraries used.
- **Queueing:** If the user is employing Cloudflare Queues to decouple the image generation (a common pattern), the default behavior on failure is to retry the batch. A single 429 from Google could trigger a retry loop in Cloudflare, where the worker hammers the Google API repeatedly trying to process the message, creating a localized DDoS effect

against the user's own quota.[18]

4.2.2 The Base64 Bloat

Image editing requires sending the image data to the server.

- **Encoding Overhead:** Base64 encoding increases the file size by approximately 33%. A 7MB image (the limit for inline data [19]) becomes ~9.3MB of text payload.
- **Transmission Time:** Uploading this payload from the client to Cloudflare, and then from Cloudflare to Google, takes time.
- **Timeout Risks:** If the upload to Google takes too long, the Cloudflare function might time out and retry, sending the payload again. This "invisible retry" consumes quota without the developer knowing the first request was actually received by Google.

4.3 The SDK: @google/genai

The user is using the newer @google/genai SDK. This is the correct choice, as legacy SDKs (@google/generative-ai) are being deprecated.[20] However, the SDK's abstraction layer can hide the raw HTTP responses.

- **Error Masking:** The SDK might throw a generic "Client Error" wrapper around the 429 response. Debugging requires inspecting the raw error object to differentiate between "Project Quota Exceeded" (your key is out of limit) and "IP Rate Limit Exceeded" (Cloudflare's IP is blocked).

5. Implementation Strategy: The "Correct Knowledge"

Based on the analysis above, we can now formulate the definitive implementation guide. This solution addresses the quota limits, the frontend concurrency, and the backend architecture.

5.1 Step 1: Solving the Quota Constraint

The most direct solution to the "20 requests per day" limit and the "shared IP" blocking is to transition the project's billing status.

The Economic Reality:
The free tier is strictly for "hello world" exploration. It is functionally incapable of supporting an image editing workflow due to the heavy token weights.

- **Action:** Enable "Pay-as-you-go" billing in Google Cloud Console for the project.
- **Cost Implication:** At ~$0.04 per image, the cost is trivial for development. A budget of $5.00 allows for ~125 image generations—far more than the free tier's daily allowance, and crucially, **paid requests are not subject to the shared-IP rate limits** in the same way free requests are. Paid tier requests are prioritized and rate-limited by project ID, bypassing the Cloudflare "noisy neighbor" problem.[7]

5.2 Step 2: Backend Implementation (Cloudflare Pages)

The backend code must be robust against failures and explicitly handle the multimodal payload.

Key Technical Details:

- **Streaming vs. Blocking:** For image generation, blocking requests (waiting for the full response) is often simpler than streaming, as the image arrives as a single binary blob.
- **Security:** The API key must be stored in Cloudflare Environment Variables, never in the frontend code.

Annotated Implementation (functions/api/edit-image.js):

JavaScript

```
import { GoogleGenAI } from "@google/genai";

export async function onRequestPost(context) {
  const { request, env } = context;

  // 1. Validation and Setup
  const apiKey = env.GEMINI_API_KEY;
  if (!apiKey) return new Response("Missing API Key", { status: 500 });

  // 2. Parse the Incoming Request
  // Expecting JSON with prompt and base64 image
  let body;
  try {
    body = await request.json();
  } catch (e) {
    return new Response("Invalid JSON", { status: 400 });
  }

  const { prompt, imageBase64 } = body;

  if (!prompt ||!imageBase64) {
    return new Response("Missing prompt or image data", { status: 400 });
  }

  // 3. Initialize the SDK
```

```
// We use the unified SDK pattern.
const ai = new GoogleGenAI({ apiKey });

try {
  // 4. Construct the Multimodal Payload
  // The gemini-2.5-flash-image model expects 'generateContent'
  // We must pass the image as inline data.
  const model = "gemini-2.5-flash-image";

  const contents =
    }
  ];

  // 5. Execute the Generation
  // CRITICAL: responseModalities must include IMAGE
  const result = await ai.models.generateContent({
    model: model,
    contents: contents,
    config: {
      responseModalities:,
      temperature: 0.4, // Lower temperature stabilizes editing consistency
    }
  });

  // 6. Extract the Image
  // The response structure needs careful parsing.
  const parts = result.candidates?.?.content?.parts ||;
  const imagePart = parts.find(p => p.inlineData);

  if (!imagePart) {
    throw new Error("Model returned no image data");
  }

  // 7. Return to Client
  return new Response(JSON.stringify({
    image: imagePart.inlineData.data
  }), {
    headers: { "Content-Type": "application/json" }
  });

} catch (error) {
  // 8. Robust Error Handling
  console.error("Gemini API Error:", error);
```

```
    // Pass through the specific Google error code if available
    const status = error.status |

| 500;
    const message = error.message |

| "Internal Server Error";

    return new Response(JSON.stringify({ error: message }), { status });
  }
}
```

5.3 Step 3: Frontend Implementation (React 19)

The frontend must prevent double-submissions and handle the upload efficiently.

Key Technical Details:

- **useTransition:** This hook is essential in React 19 to mark the API call as a transition, allowing the UI to remain responsive while preventing race conditions.
- **Base64 Cleaning:** The frontend must strip the prefix before sending to the backend if the backend expects raw base64.

Annotated Implementation (src/components/ImageEditor.jsx):

```
    JavaScript


import { useState, useTransition } from 'react';

export default function ImageEditor() {
  const = useTransition();
  const [generatedImage, setGeneratedImage] = useState(null);
  const [error, setError] = useState(null);

  const handleEdit = async (formData) => {
    const file = formData.get('imageFile'); // From <input type="file" />
    const prompt = formData.get('prompt');

    if (!file ||!prompt) return;
```

```
  // Convert file to Base64
  const reader = new FileReader();
  reader.onloadend = () => {
    const base64String = reader.result.split(','); // Remove prefix

    startTransition(async () => {
      setError(null);
      try {
        const response = await fetch('/api/edit-image', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({
            prompt,
            imageBase64: base64String
          }),
        });

        if (response.status === 429) {
          throw new Error("Quota exceeded. Please try again later.");
        }

        if (!response.ok) {
          throw new Error(`Server Error: ${response.statusText}`);
        }

        const data = await response.json();
        setGeneratedImage(data.image); // Display result
      } catch (e) {
        setError(e.message);
      }
    });
  };
  reader.readAsDataURL(file);
};

return (
  <form action={handleEdit}>
    <input name="imageFile" type="file" accept="image/*" />
    <input name="prompt" type="text" placeholder="Describe edits..." />
    <button type="submit" disabled={isPending}>
      {isPending? 'Processing...' : 'Generate Edit'}
    </button>
    {error && <p className="error">{error}</p>}
```

```
    {generatedImage && <img src={`${generatedImage}`} />}
  </form>
 );
}
```

6. Detailed Analysis of 429 Errors and Mitigation Strategies

When the 429 RESOURCE_EXHAUSTED error occurs, it is rarely a binary "stop" signal. It carries specific metadata that informs the retry strategy.

6.1 Deconstructing the 429 Response

Google's API often provides headers or body details with the error:

- retry-after: A header indicating the number of seconds to wait.
- **Granularity:** The error message typically distinguishes between "User Rate Limit" (API key) and "Shared Rate Limit" (Project/IP).
  - *Case A: "User rate limit exceeded"* -> You have hit the 20 RPD cap or the 2 RPM burst cap. **Solution:** Wait 24 hours or upgrade to paid.
  - *Case B: "Quota exceeded for metric... on project..."* -> The project-level budget or free tier bounds are hit. **Solution:** Enable billing.
  - *Case C: "The service is currently overloaded"* -> This is a server-side capacity issue (common with "Nano Banana" models). **Solution:** Exponential backoff retry.

6.2 The "Proxy & Cache" Architectural Pattern

To survive in a constrained environment without upgrading to paid tiers immediately, one can implement a caching layer using Cloudflare KV. This is particularly effective for testing, where developers often retry the *same* prompt multiple times.

The Logic:

1. **Hash the Input:** Calculate SHA256(prompt + imageBase64).
2. **KV Lookup:** Before calling Google, check await env.IMAGE_CACHE.get(hash).
3. **Cache Hit:** If found, return the cached image immediately. Cost: 0 Google tokens.
4. **Cache Miss:** Call Google API. On success, write the result to KV with a TTL (e.g., 24 hours).
5. **Result:** Repeated pressing of "Generate" on the same input incurs no API cost and triggers no 429s.

6.3 Handling Cold Starts and Latency

The gemini-2.5-flash-image model is fast, but Cloudflare Workers have a CPU limit. Large image uploads (base64 processing) can consume significant CPU time.

- **Optimization:** Use the stream: true option in the SDK if the Cloudflare function acts as a pass-through. This allows the function to pipe the response directly to the client byte-by-byte, reducing memory pressure on the Worker and preventing timeouts for large generations.

7. Operationalizing for Production

Moving from a prototype to a production app requires monitoring and resilience.

7.1 Data Tables: Quotas and Pricing

The following table summarizes the operational constraints as of late 2025/early 2026.

Metric	Free Tier (Gemini 2.5 Flash)	Paid Tier (Tier 1)	Impact on Image Editing
Requests Per Day (RPD)	~20 - 50 (Soft Cap)	Unlimited (Pay-as-you-go)	Free tier is non-viable for iteration.
Requests Per Minute (RPM)	2 - 10	~1,000+	High RPM needed for concurrent users.
Input Cost	Free	$0.30 / 1M tokens	Input images are cheap (~$0.0001).
Output Cost	Free	$30.00 / 1M tokens	Output images are costly (~$0.04/image).
Rate Limit Scope	IP / API Key	Project ID / Billing Account	Paid tier eliminates Cloudflare IP blocks.

6

7.2 Safety Settings and Filtering

Image editing models are subject to strict safety filters (e.g., no generation of copyrighted characters, no photorealistic deepfakes of real people).

- **The Trap:** A safety block often manifests as a 400 or a generic error, which can be confused with a 429.
- **Fix:** Inspect the finishReason in the API response. If it is SAFETY, the prompt was rejected, not the quota. The application should handle this gracefully by informing the user.

7.3 Multi-Provider Redundancy

Given the volatility of the 429 errors on Google's side (even for paid tiers during high traffic), a production app should implement a fallback pattern.

- **Primary:** Gemini 2.5 Flash Image.
- **Secondary:** Vertex AI Imagen 3 (via separate SDK/Endpoint).
- Tertiary: Open-source model (e.g., Flux or SDXL) hosted on a separate inference provider (e.g., Hugging Face or Replicate).
  This ensures that if Google's multimodal cluster is overloaded, the user still receives an image, maintaining application reliability.

8. Conclusion and Strategic Outlook

The difficulty in implementing image editing with gemini-2.5-flash-image stems from a collision of three factors: the recent decimation of free-tier allowances, the network opacity of serverless edge functions, and the intricacies of multimodal API payloads.

The "correct knowledge" required to solve this is not merely syntactical. While using the correct SDK method (generateContent with responseModalities) is essential, code correctness alone cannot overcome the "20 requests per day" barrier or the "noisy neighbor" IP blocks. The definitive solution requires an architectural acknowledgment that high-fidelity AI inference is a paid utility. By enabling billing to bypass shared rate limits, implementing strict frontend debounce/transition logic in React 19, and utilizing backend caching, the application can achieve the resilience required for a professional environment.

The future outlook suggests that "Flash-Lite" models may eventually support image generation at lower costs, but for the immediate term, developers must architect their systems assuming that free access is ephemeral and unstable. The transition to paid, authenticated, and cached architectures is the only viable path for consistent application performance.

Works cited

1. Is Free Gemini 2.5 Pro API fried? Changes to the free quota in 2025 - CometAPI, accessed December 24, 2025, https://www.cometapi.com/is-free-gemini-2-5-pro-api-fried-changes-to-the-free-quota-in-2025/
2. Do they really think we wouldn't notice a 92% free tier quota? - Gemini API,

accessed December 24, 2025, https://discuss.ai.google.dev/t/do-they-really-think-we-wouldnt-notice-a-92-free-tier-quota/111262
3. Gemini has slashed free API limits, here's what to use instead - How-To Geek, accessed December 24, 2025, https://www.howtogeek.com/gemini-slashed-free-api-limits-what-to-use-instead/
4. Gemini only letting me generate >10 images a day - Google Help, accessed December 24, 2025, https://support.google.com/gemini/thread/373278659/gemini-only-letting-me-generate-10-images-a-day?hl=en
5. Gemini 2.5 Flash Image Free Limit Complete Guide 2025: Cost Calculator & China Access, accessed December 24, 2025, https://www.cursor-ide.com/blog/gemini-2-5-flash-image-free-limit
6. accessed December 24, 2025, https://developers.googleblog.com/introducing-gemini-2-5-flash-image/#:~:text=Gemini%202.5%20Flash%20Image%20is,follow%20Gemini%202.5%20Flash%20pricing.
7. Rate limits | Gemini API - Google AI for Developers, accessed December 24, 2025, https://ai.google.dev/gemini-api/docs/rate-limits
8. Release notes | Gemini API - Google AI for Developers, accessed December 24, 2025, https://ai.google.dev/gemini-api/docs/changelog
9. Gemini API Error 429: Causes, Fixes & Prevention Tips - HostingSeekers, accessed December 24, 2025, https://www.hostingseekers.com/blog/gemini-api-error-429-causes-fixes-prevention/
10. "429 Too Many Requests" Error with gemini-2.5-flash-image-preview in N8n Workflow, accessed December 24, 2025, https://discuss.ai.google.dev/t/429-too-many-requests-error-with-gemini-2-5-flash-image-preview-in-n8n-workflow/103761
11. Gemini AI and Cloudflare Pages | Experience - Reddit, accessed December 24, 2025, https://www.reddit.com/r/CloudFlare/comments/1p991zt/gemini_ai_and_cloudflare_pages_experience/
12. Image generation with Gemini (aka Nano Banana & Nano Banana Pro) | Gemini API | Google AI for Developers, accessed December 24, 2025, https://ai.google.dev/gemini-api/docs/image-generation
13. Google AI Studio Free Plans, Trials, and Subscriptions: access tiers, limits, and upgrade paths, accessed December 24, 2025, https://www.datastudios.org/post/google-ai-studio-free-plans-trials-and-subscriptions-access-tiers-limits-and-upgrade-paths
14. Introducing Gemini 2.5 Flash Image, our state-of-the-art image model, accessed December 24, 2025, https://developers.googleblog.com/introducing-gemini-2-5-flash-image/
15. Generate images using Imagen | Gemini API | Google AI for Developers, accessed

December 24, 2025, https://ai.google.dev/gemini-api/docs/imagen
16. Generate & edit images using Gemini (aka "nano banana") | Firebase AI Logic - Google, accessed December 24, 2025, https://firebase.google.com/docs/ai-logic/generate-images-gemini
17. Guides: Google Gemini Image Generation - AI SDK, accessed December 24, 2025, https://ai-sdk.dev/cookbook/guides/google-gemini-image-generation
18. Batching, Retries and Delays - Queues - Cloudflare Docs, accessed December 24, 2025, https://developers.cloudflare.com/queues/configuration/batching-retries/
19. Gemini 2.5 Flash | Generative AI on Vertex AI - Google Cloud Documentation, accessed December 24, 2025, https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-flash
20. Migrate to the Google GenAI SDK | Gemini API, accessed December 24, 2025, https://ai.google.dev/gemini-api/docs/migrate
21. Gemini Developer API pricing, accessed December 24, 2025, https://ai.google.dev/gemini-api/docs/pricing