Vamshi Jandhyala

Taste in API design: Why developer experience demands aesthetic judgment

December 2024

APIs aren't just functional contracts, they're user interfaces for developers. Great API design requires the same aesthetic judgment and taste that defines excellent product design.

Introduction

When we talk about "taste" in product design, we usually think of visual interfaces, typography, color palettes, and interaction patterns. But taste matters just as much in API design, perhaps even more. An API is a user interface for developers, and the difference between a well-designed API and a poorly designed one can determine whether developers love or abandon your platform.

Taste in API design isn't about subjective preference. It's about making thousands of small decisions that collectively create an experience that feels inevitable, elegant, and correct. It's the difference between Stripe's API and most payment processors. It's why developers will pay a premium for services with better APIs even when cheaper alternatives exist.

What is taste in API design?

Taste in API design manifests in how the API *feels* to use:

- **Naming consistency**: Do concepts map to intuitive names? Do similar operations follow similar patterns?
- **Predictability**: Can developers guess how to accomplish something they haven't done before?
- **Error handling**: Are error messages helpful? Do they guide developers toward solutions?
- **Discoverability**: Can developers explore the API and understand its capabilities without constant documentation lookup?
- **Aesthetic coherence**: Does the API feel like it was designed by one mind with a clear vision?

Good taste means saying ”no” to features that would make the API more powerful but less elegant. It means choosing consistency over clever optimization. It means understanding that the API’s surface area is itself a design constraint.

Examples of taste in action

Stripe: The gold standard

Stripe’s API is often cited as exemplary because it demonstrates taste in countless small decisions:

Good naming: ‘javascript // Creating a customer feels natural const customer = await stripe.customers.create({ email: ’customer@example.com’, source: token.id });

// Charging feels like what you’re actually doing const charge = await stripe.charges.create({ amount: 2000, currency: ’usd’, customer: customer.id }); ‘

The verbs are consistent (`create`, `retrieve`, `update`, `delete`). Resource names are plural and intuitive (`customers`, `charges`, `subscriptions`). The hierarchy makes sense.

Contrast with poor taste: ‘javascript // Inconsistent verbs api.createCustomer() // create api.getCharge() // retrieve (why not retrieveCharge?) api.modifyPlan() // update (why not updatePlan?) api.deleteCard() // delete (at least this is consistent)

// Unclear resource naming api.charges.new() // is this creating or retrieving? api.customer.fetch() // why fetch here but get elsewhere? ‘

Twilio: Predictable patterns

Twilio’s API demonstrates taste through predictability:

‘javascript // Creating resources follows the same pattern const message = await client.messages.create({...}); const call = await client.calls.create({...}); const recording = await client.recordings.create({...});

// Listing resources is always the same const messages = await client.messages.list({limit: 20}); const calls = await client.calls.list({limit: 20}); ‘

Once you learn the pattern, you can use any part of the API without looking at documentation. This predictability is a hallmark of good taste, the designer resisted the temptation to optimize each endpoint individually and instead maintained consistency.

GitHub: Discoverable resources

GitHub’s API shows taste in how it handles resource relationships:

‘javascript // Resources expose their relationships clearly GET /repos/:owner/:repo/issues GET /repos/:owner/:repo/pulls GET /repos/:owner/:repo/commits

// Nested resources make sense GET /repos/:owner/:repo/issues/:issue_number/comments GET /repos/:owner/:repo/pulls/:pull_number/reviews ‘

The URL structure itself is documentation. You can guess endpoints you’ve never used because the patterns are consistent and intuitive.

Where taste breaks down

Bad taste in API design usually comes from:

1. Premature optimization

‘javascript // Bad: Optimizing for fewer API calls at the expense of clarity api.createCustomerAndCharge({ customer: {...}, charge: {...} });
// Good: Clear, composable operations const customer = await api.customers.create({...}); const charge = await api.charges.create({ customer: customer.id, ... }); ‘
The first approach saves one API call but creates a Frankenstein operation that doesn’t map to how developers think. Good taste chooses clarity.

2. Exposing internal structure

‘javascript // Bad: Exposing database field names api.users.update({ usr_email_addr: ’new@example.com’, usr_phone_nbr: ’555-1234’ });
// Good: Developer-friendly naming api.users.update({ email: ’new@example.com’, phone: ’555-1234’ }); ‘
Your database schema is an implementation detail. Good taste means designing the API for developers, not for your database.

3. Inconsistent error handling

‘javascript // Bad: Inconsistent error responses // Some endpoints return: { error: ”Invalid email” }
// Others return: { message: ”Email validation failed”, code: 400 }
// Others throw HTTP errors with no body
// Good: Consistent error structure { error: { type: ”validation_error”, message: ”Invalid email address”, param: ”email”, code: ”invalid_email” } } ‘
Good taste means developers can write error handling once and have it work everywhere.

4. Feature creep without discipline

‘javascript // Bad: Adding parameters that break the mental model api.customers.create({ email: ’user@example.com’, autoChargeOnCreate: true, // Wait, I thought I was creating a customer? sendWelcomeEmail: false, // Now I’m managing email campaigns? syncToWarehouse: true, // And database syncing? notifySlackChannel: ’#sales’ // And Slack integrations? });
// Good: Focused, single-purpose operations api.customers.create({ email: ’user@example.com’ }); // Other concerns handled separately ‘
Good taste means resisting the temptation to make every operation do everything. Each endpoint should have a clear, focused purpose.

More examples of good taste

Slack: Thoughtful pagination

Slack's API handles pagination elegantly:
'javascript // Consistent cursor-based pagination across all list endpoints const result = await slack.conversations.list({ limit: 100, cursor: 'dXNlcjpVMDYxTkZUVDI=' });
// Response structure is predictable { ok: true, channels: [...], response_metadata: { next_cursor: 'dGVhbTpDMUg5UkVTRow=' } } '
Good taste: Every list endpoint uses the same pagination pattern. Developers write pagination logic once and it works everywhere.

Shopify: Expressive filtering

Shopify's GraphQL API demonstrates taste in query design:
'graphql # Natural, expressive filtering { products(first: 10, query: "tag:summer AND inventory_quantity:>0") { edges { node { title variants { price } } } } } '
Good taste: Filtering uses natural language syntax. Complex queries remain readable. The query structure mirrors the response structure.

Django REST Framework: Sensible defaults

Django REST Framework shows taste in its default behaviors:
'python # Minimal code for common patterns class ProductViewSet(viewsets.ModelViewSet): queryset = Product.objects.all() serializer_class = ProductSerializer # That's it - pagination, filtering, ordering all work automatically '
Good taste: Common patterns require minimal configuration. Framework provides sensible defaults while allowing customization when needed.

Notion API: Clear versioning

Notion's API versioning demonstrates taste:
'javascript // Version specified in header, not URL fetch('https://api.notion.com/v1/pages', { headers: { 'Notion-Version': '2022-06-28' } }); '
Good taste: Version in header keeps URLs clean. Date-based versions are self-documenting. No breaking changes within a version.

AI agents as API consumers: Why taste matters more than ever

The rise of AI agents as API consumers adds a new dimension to why taste matters. When AI agents interact with APIs, good design becomes critical in ways we're only beginning to understand.

Predictability enables AI reasoning

AI agents rely on pattern recognition to understand how to use APIs. Well-designed APIs with consistent patterns are significantly easier for AI models to learn and use

correctly:

‘javascript // AI-friendly: Consistent patterns // The AI can infer that all resources follow the same CRUD pattern GET /customers/:id POST /customers PATCH /customers/:id DELETE /customers/:id

GET /orders/:id POST /orders PATCH /orders/:id DELETE /orders/:id

// AI-unfriendly: Inconsistent patterns // Each resource uses different verbs and structures GET /customers/:id POST /create-customer PUT /customers/:id/modify DELETE /remove-customer/:id

GET /fetch-order/:id POST /orders/new POST /orders/:id/update POST /orders/:id/cancel ‘

When patterns are consistent, AI agents can correctly use endpoints they’ve never seen in their training data by applying learned patterns. Inconsistent APIs force the AI to memorize each endpoint individually, increasing error rates.

Self-describing APIs reduce hallucination

AI agents sometimes ”hallucinate” API capabilities that don’t exist. Well-named, self-describing APIs reduce this problem:

‘javascript // Clear, self-describing api.customers.subscriptions.cancel({ subscription_id: ’sub_123’, cancel_at_period_end: true });

// Ambiguous - AI might hallucinate parameters api.cancelSub(’sub_123’); // Does this cancel immediately or at period end? // Can I provide a cancellation reason? // The AI has to guess or hallucinate parameters ‘

Good taste means designing APIs that communicate their capabilities through their structure and naming, not just through documentation that AI agents may not have access to.

Error messages guide AI self-correction

When AI agents make mistakes, good error messages help them self-correct:

‘javascript // AI-friendly error: Guides correction { error: { type: ”invalid_request”, message: ”Missing required field: customer_id”, param: ”customer_id”, suggestion: ”Include customer_id in the request body” } } // AI can parse this, understand the issue, and retry correctly

// AI-unfriendly error: Forces guessing { error: ”Bad request” } // AI doesn’t know what went wrong or how to fix it ‘

As AI agents become more prevalent as API consumers, the distinction between good and bad API design will manifest in agent reliability. Agents using well-designed APIs will succeed at tasks; agents using poorly designed APIs will fail repeatedly.

The impact on developer experience

APIs with good taste create measurably better developer experiences:

Faster onboarding: Developers can learn one pattern and apply it everywhere. Time to first successful API call drops dramatically.

Fewer support tickets: When the API is predictable and well-named, developers don’t need to ask basic questions. Error messages guide them to solutions.

Higher retention: Developers who enjoy using your API become advocates. They'll recommend it, write tutorials, and defend it in technical discussions.

Lower maintenance costs: Consistent APIs are easier to maintain, test, and extend. Future additions feel natural rather than bolted-on.

Competitive advantage: In crowded markets, developer experience can be the primary differentiator. Developers will pay more for APIs that respect their time and intelligence.

Cultivating taste in API design

Good taste in API design isn't innate, it's developed through:

1. Study great APIs

Use Stripe, Twilio, GitHub, and Shopify's APIs. Notice the patterns. Ask yourself why certain decisions feel right. What makes them predictable? What makes them pleasant to use?

2. Write against your own API

Don't just design in the abstract. Write real client code that uses your API. If you find yourself fighting the API, that's a design smell.

3. Get feedback early

Share API designs with developers before implementation. Watch them try to use it. Where do they get confused? What do they expect to work that doesn't?

4. Maintain consistency religiously

Create and document patterns for naming, error handling, pagination, filtering, and versioning. Enforce them in code review. Consistency is the foundation of predictability.

5. Say no to features

Every additional parameter, endpoint, or special case adds cognitive load. Good taste means ruthlessly eliminating anything that doesn't pull its weight.

6. Iterate based on usage

Monitor which endpoints generate the most support tickets. Where do developers make mistakes? Use that data to refine the design.

Conclusion

Taste in API design is the accumulated wisdom of thousands of small decisions made in service of developer experience. It's choosing consistency over cleverness, clarity over brevity, and long-term elegance over short-term convenience.

Great API design, like great product design, is invisible when done right. Developers simply use it without friction, without confusion, without frustration. They don't notice the design, they notice that everything just works the way they expect.

That's taste. And it matters just as much for APIs as it does for visual interfaces.

Because at the end of the day, an API is a user interface. And like all interfaces, it deserves to be designed with care, consistency, and aesthetic judgment.