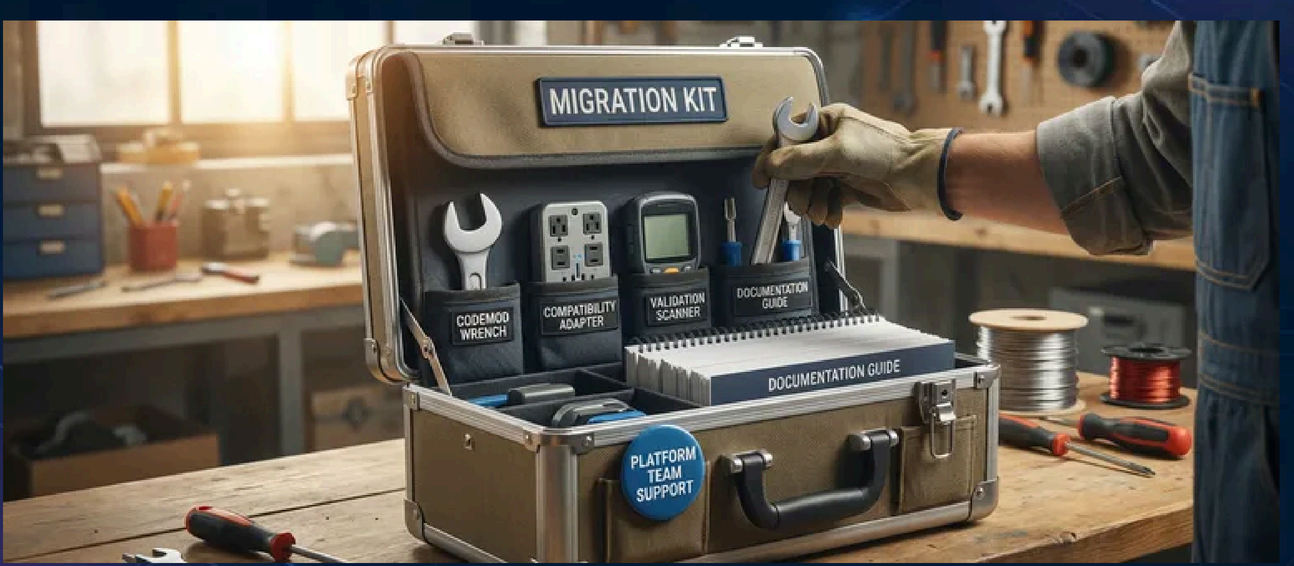
Versioning Internal Platform APIs



Published on April 8, 2023

Table of Contents

Versioning Strategies 3
Choosing a Versioning Scheme 3
Semantic Versioning for APIs 4
Breaking Change Management 5
Identifying Breaking Changes 5
The Breaking Change Process 7
Deprecation Communication 9
Deprecation Timeline 9
Multi-Channel Communication 9
Deprecation Headers and Response Warnings 10
Migration Support 11
Making Migration Easy 11
Migration Guide Structure 12
Automated Migration Tools 13
Measuring Success 15
Deprecation Metrics 15
The Migration Funnel 16
What Happens at Sunset 17
Conclusion 18
Quick Reference 19
Further Reading 19

Last week, a platform team I know shipped what they called a “small cleanup” to their deployment API. They renamed a field from `service_id` to `serviceId`, removed an endpoint that was “barely used,” and updated the response format to match their new schema. No version bump. No deprecation notice. Just a Friday afternoon deploy and a Slack message in `#platform-updates` that nobody read.

Monday morning, 40 CI pipelines failed. Three teams scrambled to update their deployment scripts. A critical security hotfix got blocked because the team couldn’t deploy. The platform team spent the entire week doing emergency migrations instead of planned work. And the trust they’d built over the previous year? Gone.

This happens constantly with internal APIs. There’s a temptation to treat them differently than external ones: “we can just tell people to update,” “everyone’s in the same building,” “we’ll coordinate in Slack.” But internal APIs deserve *more* versioning discipline than external ones, not less.

> **WARNING**
>
> Internal customers are captive customers. They can’t switch to a competitor’s platform. This makes breaking their workflows worse, not better - they have no recourse except escalating to leadership or building workarounds that create tech debt.

The same practices that make external APIs predictable - semantic versioning, deprecation policies, migration support - apply to internal platform APIs. The difference is that internal teams have higher expectations because they’re colleagues, and lower patience because they have their own roadmaps that don’t include “emergency migration of the deployment API.”

Versioning Strategies

Choosing a Versioning Scheme

Four versioning approaches dominate API design, each with tradeoffs that matter for internal platforms.

URL path versioning

/api/v1/deployments and /api/v2/deployments is explicit and visible - every request clearly shows which version it's using. Load balancers can route traffic easily, and deprecating old versions is straightforward. For internal platform APIs, this is usually the right choice because visibility trumps elegance. When debugging a failed

	pipeline at 2 AM, you want to immediately know which API version the client is calling.
Header versioning and query parameter versioning	Accept: application/vnd.platform.v2+json and /api/services?version=2 both hide version information where developers forget to set it and debugging tools don't show it. Skip these unless you have specific requirements for URL stability.
Date-based versioning	/api/2024-01-15/services works well for APIs with frequent, incremental changes - Stripe and AWS use this successfully. The downside is version accumulation: after a few years, you have dozens of dated endpoints. This works best when you have the tooling to manage many concurrent versions.

For most internal platform APIs, I recommend URL path versioning. Maximum visibility, easy auditing, simple routing, and clear communication about which versions are approaching sunset.

Semantic Versioning for APIs

Semantic versioning gives version numbers meaning. For APIs, the interpretation is straightforward:

Major versions (v1 → v2)

Indicate breaking changes that require consumer code updates. This includes removing endpoints, removing fields from responses, adding required fields to requests, changing field types, changing authentication mechanisms, or renaming fields without backward-compatible aliases. Major bumps need 90+ days advance notice and full deprecation process. Support the previous major version for at least 12 months.

Minor versions (v1.1 → v1.2)

These are backward-compatible additions. New optional endpoints, new optional request fields, new response fields, new enum values - anything that existing client code can safely ignore. No advance notice required, just release notes. Consumers can upgrade without code changes.

Patch versions (v1.2.1 → v1.2.2)

These are backward-compatible fixes. Bug fixes, performance improvements, documentation updates, security patches. These should be transparent to consumers - they shouldn't even notice. Apply patches to all supported versions.

#	Change Type	Version Bump	Consumer Action	Notice Required
1	Remove endpoint	Major	Update code	90 days
2	Remove field	Major	Update code	90 days
3	Add required field	Major	Update code	90 days
4	Add optional endpoint	Minor	None	Release notes
5	Add response field	Minor	None	Release notes
6	Bug fix	Patch	None	Changelog

Version bump decision guide.

> ⓘ INFO
>
> The key question for any change: “Will existing client code break?” If yes, it’s a major version bump with full deprecation process. If no, it’s minor or patch. When in doubt, assume breaking - consumers will tell you if you’re being too conservative.

Breaking Change Management

Identifying Breaking Changes

The hardest part of API versioning isn’t the mechanics - it’s deciding whether a change is breaking in the first place. The core question is simple: *will existing client code still work?* If no, it’s breaking. If yes, you need to dig deeper: is the behavior meaningfully different? Will consumers notice or care?

Some changes are obviously breaking. Removing an endpoint returns 404 to anyone still calling it. Changing an HTTP method from POST to PUT breaks existing clients with 405 errors. Renaming a URL path parameter from `{id}` to `{serviceId}` makes existing URLs invalid. Adding a required field to requests means all existing

requests fail validation. Changing a field's type from string to number breaks deserialization. These require full deprecation process, no exceptions.

Some changes are obviously safe. Adding an optional request field doesn't affect existing requests. Adding a field to responses is fine if clients ignore unknown fields (they should). New endpoints don't touch existing ones. Loosening validation - accepting 200 characters where you previously accepted 100 - doesn't break anything that worked before. Performance improvements and bug fixes are safe unless someone depends on the buggy behavior (it happens).

The tricky cases fall in between. Changing error codes from 400 to 422 is technically correct but breaks error handling logic. Changing pagination defaults affects clients that assumed a specific page size. Changing a field's *meaning* - like making `count` include deleted items when it previously didn't - is breaking even though the type and field name stay the same.

Breaking Change	Why It Breaks
Remove endpoint	Returns 404 to existing callers
Change HTTP method	Returns 405 to existing callers
Rename URL parameter	Existing URLs become invalid
Add required request field	Existing requests fail validation
Change field type	Deserialization fails
Change error codes	Error handling logic breaks
Change field meaning	Logic depending on value breaks

Changes that require full deprecation process.

Not every change demands that level of process. Many changes are additive or relaxing — they expand what the API accepts or returns without invalidating anything consumers already depend on. The distinction matters operationally because safe changes can ship continuously without coordination, while breaking changes gate on migration timelines and consumer readiness.

Misclassifying a safe change as a breaking change slows delivery unnecessarily; misclassifying a breaking change as safe causes outages. When the boundary is unclear — a changed default, a subtly different error shape — treat the change as breaking and let consumers confirm otherwise. It's easier to relax a deprecation notice than to roll back a broken deployment.

Safe Change	Why It's Safe
Add optional request field	Existing requests still valid
Add response field	Clients should ignore unknown fields
Add new endpoint	Existing endpoints unchanged
Loosen validation	Previously valid requests still work
Performance improvement	Same results, faster

Changes that can ship without deprecation.

When you're unsure, treat it as breaking. Consumers will tell you if you're being too conservative. Nobody complains about unnecessary deprecation warnings. They complain loudly about broken pipelines.

The Breaking Change Process

Once you've identified a breaking change, follow a structured process that gives consumers time to adapt.

Step 1: Identify affected consumers.
Before announcing anything, know who's using the affected endpoints. API gateway logs tell you which services call which endpoints. Service mesh telemetry shows call patterns. SDK analytics reveal version distribution. Query the last 30 days of traffic - this catches regular callers and weekly batch jobs.

Step 2: Create a proposal document.
Write down what's changing, why it's changing, which endpoints are affected, estimated migration effort, and the proposed timeline. This becomes the single source of truth for the deprecation.

Step 3: Open a feedback period.
Give affected teams 30 days to review the proposal and raise concerns. Maybe you're missing a use case. Maybe the timeline is too aggressive for their roadmap. Maybe there's a simpler migration path you hadn't considered. Listen to the feedback - internal customers have context you don't.

Step 4: Notify affected teams directly.
Broadcast announcements get ignored. Target the teams you identified in step 1. Send to their Slack channels. Email their tech leads. Make it impossible to miss. The notification should include what's changing, why, the timeline, the migration guide, and where to ask questions.

Step 5: Schedule the timeline.
Set milestones for feedback deadline, new version release, 90-day warning, 30-day warning, and sunset date. Put them in the team calendar. Configure automated reminders. Don't rely on anyone remembering.

✓ **SUCCESS**

Identify consumers before announcing changes. API gateway logs, service mesh telemetry, and SDK (Software Development Kit) analytics tell you who’s actually using what. Targeted notification to affected teams gets better response than broadcast announcements that everyone ignores.

Deprecation Communication

Deprecation Timeline

A well-run deprecation follows a predictable timeline that gives consumers multiple opportunities to act - and escalating urgency as the deadline approaches.

Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511
support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864

Major version sunsets

Needs 12 months total. Start with a 30-day feedback period where you announce the proposal and gather input. Release the new version, then run both versions in parallel for 12 months. Send the 90-day warning, the 30-day warning, and finally sunset the old version. This sounds like a long time, but internal teams have their own roadmaps. A migration that takes 2 days of work might not get scheduled for 3 months.

2

Minor feature deprecations

These can move faster - 6 months is usually sufficient. Same pattern: announcement, parallel operation, escalating warnings, sunset.

Security emergencies

These are the exception. When you need to remove something immediately for security reasons, the minimum is 30 days with direct contact to every known consumer. Get security team approval, document the risk, and communicate urgently. Don't use this as a shortcut for normal deprecations - it burns trust.

Multi-Channel Communication

People miss messages. The developer who needs to migrate might be on vacation when you post the announcement. Their email might filter your notice to a folder they never check. Their team lead might forget to forward it. Successful deprecation communication uses multiple channels with increasing urgency.

The key insight is that urgency should escalate as the deadline approaches. Early communication is broad and informational - changelog updates, Slack posts, email announcements. As sunset nears, communication becomes targeted and direct: personal outreach to remaining consumers, calendar invites for the sunset date, escalation to team leads. The table below shows the progression.

Timeline	Communication	Audience	Action Required
Announcement	Changelog, Slack, email	All engineers	Awareness
90 days	Slack reminder, dashboard	Affected teams	Plan migration
30 days	Direct email, Slack DM	Remaining consumers	Urgent migration
7 days	Personal outreach	Critical consumers	Emergency

Timeline	Communication	Audience	Action Required
Sunset day	Final notice	All	Complete

Deprecation communication timeline.

Deprecation Headers and Response Warnings

Your API should actively warn consumers about deprecations, not just document them. HTTP headers are the standard mechanism:

HTTP

```http
HTTP/1.1 200 OK
Deprecation: true
Sunset: Sat, 15 Aug 2024 00:00:00 GMT
Link: </docs/migration/v1-to-v2>; rel="deprecation"
```

Standard HTTP deprecation headers.

These headers are machine-readable - consumers can build automation that alerts when their dependencies are deprecated. Combine headers with response body warnings for visibility in logs and debugging:

JSON

```json
{
  "data": { },
  "_meta": {
    "deprecation": {
      "deprecated": true,
      "sunsetDate": "2024-08-15",
      "replacement": "/api/v2/deployments",
      "migrationGuide": "https://docs/migration/v1-to-v2"
    }
  }
}
```

Response body deprecation warning.

The `_meta` field convention keeps deprecation warnings separate from business data while ensuring they're visible in API responses.

WARNING

Never surprise people with deprecations. Even if policy says "90 days notice," earlier is better. Give consumers time to plan, prioritize, and execute migration - not scramble under deadline pressure.

Migration Support

Making Migration Easy

The faster consumers migrate, the sooner you can sunset the old version. Every friction point in migration extends the deprecation timeline. Invest in making migration as easy as possible - it pays back in reduced support load and faster adoption.

`Tier 1: Self-Service`

Most migrations should be completable without platform team involvement. Provide a comprehensive migration guide with before/after code examples, a comparison table of old vs new API, an FAQ covering common questions, and a self-service validator that consumers can run to check their migration status. Support comes through async channels like #platform-help.

`Tier 2: Assisted`

For consumers who need more help, offer scheduled office hours for questions, code review of migration PRs, and migration progress tracking. This tier catches teams who are stuck on edge cases or unclear about specific changes.

`Tier 3: Hands-On`

For high-impact consumers or complex migrations, the platform team creates the migration PRs directly. This includes pair programming sessions and production deployment support. Reserve this for critical paths where a delayed migration blocks the entire deprecation.

Support Tier	Platform Team Effort	When to Use
Self-service	Minimal (docs, tools)	Most consumers
Assisted	Moderate (office hours, reviews)	Stuck consumers
Hands-on	High (PRs, pairing)	Critical consumers, complex cases

Migration support tiers.

Migration Guide Structure

A good migration guide answers questions in the order developers ask them. Start with the overview: what's changing, why, and the timeline. Then provide a quick start - the minimal changes for basic migration, ideally with copy-paste code snippets. A 5-minute migration path for simple cases gets momentum going.

Follow with detailed changes: endpoint-by-endpoint mapping, request/response format changes, authentication changes. Include before/after code examples for each language your consumers use. Cover edge cases explicitly - they're where most migration bugs hide.

End with testing guidance: how to verify migration works, dual-write/dual-read patterns for gradual rollout, and rollback procedures if something goes wrong. Include a troubleshooting section addressing common errors.

A typical migration guide structure looks like this:

v1-to-v2-migration-guide.md

```
# Deployment API v1 → v2 Migration Guide

## Overview
- **What's changing**: Field naming conventions, response format
- **Timeline**: v1 sunset August 15, 2024
- **Estimated effort**: 30 minutes for most services

## Quick Start (5 minutes)
1. Update SDK: `npm install @platform/deploy-sdk@2`
```

```
10  2. Run the codemod: `npx @platform/deploy-codemod`
11  3. Verify in staging
12
13  ## Detailed Changes
14  | v1 | v2 | Notes |
15  |----|----|---------|
16  | service_id | serviceId | Codemod handles this |
17  | GET /status | GET /deployments/{id}/status | New endpoint path |
18
19  ## Testing Your Migration
20  - Run `platform-cli validate` to check for v1 usage
21  - Deploy to staging and verify deployment workflow
22
23  ## Troubleshooting
24  - **"Unknown field service_id"**: Run the codemod again
25  - **404 on /status**: Update to new endpoint path
```

Example migration guide structure.

Automated Migration Tools

Manual migration doesn't scale. When you have 50 consuming services, even a "simple" migration consumes hundreds of developer-hours. Automation tools reduce that dramatically.

Codemods automatically transform code from old API patterns to new ones. They parse source files, identify API calls matching v1 patterns, and rewrite them to v2. Good codemods flag changes that need manual review rather than silently transforming ambiguous cases.

For internal APIs, developing custom codemods is often worth the investment. You know exactly what patterns consumers use because you can grep the monorepo or scan consuming services. A codemod that handles 80% of cases automatically and flags the remaining 20% for manual review dramatically reduces migration friction. Facebook's jscodeshift makes this tractable for JavaScript/TypeScript:

deploy-api-v2-codemod.ts

```
1  // jscodeshift transform for deployment API v1 → v2 migration
2  export default function transformer(file, api) {
3    const j = api.jscodeshift;
```


Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511
support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864

```
4    const root = j(file.source);
5
6    // Transform service_id → serviceId in object properties
7    root.find(j.Property, { key: { name: 'service_id' } })
8      .forEach(path => {
9        path.node.key.name = 'serviceId';
10     });
11
12   // Flag ambiguous cases for manual review
13   root.find(j.CallExpression, { callee: { property: { name: 'deploy' } } })
14     .forEach(path => {
15       const args = path.node.arguments;
16       if (args.length > 2) {
17         // Complex call pattern - add review comment
18         j(path).insertBefore(
19           j.commentLine(' TODO: Review this deploy() call for v2 compatibility')
20         );
21       }
22     });
23
24   return root.toSource();
25 }
```

Custom jscodeshift codemod for internal API migration.

The key to good codemods is conservative transformation: change what you're certain about, flag what you're not. A codemod that silently breaks edge cases is worse than no codemod at all.

- **Compatibility adapters**
  Provides a translation layer that accepts v1 format and converts to v2 internally. This lets the platform team sunset the actual v1 implementation while giving consumers more time to migrate their code. The adapter logs usage for migration tracking, creating visibility into who still depends on the compatibility layer.
- **Migration validators**
  Scans codebases for v1 API usage and generate reports. Run them in CI to block PRs that introduce new v1 dependencies. Provide them as self-service tools so consumers can check their own migration progress.
- **Progress trackers**
  Aggregates data from API gateway logs, validator results, and team self-reports to show migration status across all consumers. The platform team uses this to identify holdouts and target support.

✓ SUCCESS

The easier migration is, the faster it happens. Codemods that automatically transform code, adapters that provide temporary compatibility, and validators that confirm migration completeness all reduce friction and accelerate adoption.

Measuring Success

Deprecation Metrics

You can’t manage what you can’t measure. Deprecation metrics tell you whether migration is on track, where consumers are stuck, and whether the deprecation is causing problems.

Adoption metrics

Tracks migration progress. New version adoption - the percentage of traffic on v2 versus v1 - is the headline number. Migration velocity shows how many consumers migrate per week; you want this linear or accelerating, not flat. Holdouts - consumers still on v1 near sunset - identify who needs urgent attention.

Health metrics

Catches problems. Monitor error rates during the migration period; they shouldn't increase versus baseline. Track migration-related support tickets to ensure the platform team isn't overwhelmed. Watch for rollbacks - services that migrated to v2 then reverted to v1 indicate something's wrong with the new version.

Timeline metrics keep the deprecation on schedule. Compare actual milestone dates to planned ones. The ultimate measure: did you successfully sunset the old version on the announced date?

The Migration Funnel

Think of migration as a funnel with stages: total consumers → aware → started migration → testing on v2 → production on v2 → v1 fully retired. Each stage has observable metrics.

Aware

Measured by announcement reach - did your Slack post get emoji reactions? Did affected teams acknowledge the notification?

Started migration

Shows up as PRs opened, branches created, or SDK upgrades in package files.

Testing on v2

Means staging or dev traffic hitting the new version.

Production on v2

The goal - production traffic on the new version, ideally with v1 traffic dropping to zero.

Here's what a healthy migration funnel looks like in practice. For a deployment API v2 migration with 50 consuming services:

- **Week 1:**
  50 total → 45 aware (90%) → 0 started. *Normal - teams are reading the announcement.*
- **Week 4:**
  50 total → 50 aware (100%) → 30 started (60%) → 5 testing. *Good velocity.*
- **Week 8:**
  50 total → 50 aware → 45 started (90%) → 35 testing (70%) → 20 production (40%). *On track.*
- **Week 12 (90 days out):**
  50 total → 48 production (96%) → 2 holdouts identified. *Target support at holdouts.*

When consumers are stuck at a particular stage, investigate why. Often it's a missing migration tool (stuck at "started"), unclear documentation (stuck at "testing"), or a blocking bug in v2 (stuck at "production"). Rarely is it resistance - most developers would rather move on than maintain compatibility with deprecated APIs.

Metric	90 Days Out	30 Days Out	Sunset Day
v2 Adoption	> 30%	> 80%	100%
Consumers Migrated	> 50%	> 95%	100%

Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511
support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864

Metric	90 Days Out	30 Days Out	Sunset Day
Migration Errors	Baseline	Baseline	N/A
Holdouts Identified	All known	Action plans	Resolved

Migration milestone targets.

What Happens at Sunset

When sunset day arrives, you need a clear cutover plan. The cleanest approach is returning HTTP 410 Gone with a response body pointing to the migration guide and v2 endpoint. This is unambiguous - the old version is dead, not broken.

Some teams prefer a softer approach: redirect v1 requests to v2 with a compatibility layer, logging each redirect for visibility. This buys stragglers more time but extends your maintenance burden. Only do this if you have genuine holdouts who can't migrate in time and have escalated appropriately.

Whatever you choose, communicate it in advance. "On August 15, v1 will return 410 Gone" is clear. "On August 15, v1 will stop working somehow" creates anxiety. The sunset mechanism should be as predictable as the deprecation timeline.

Conclusion

Remember that developer who spent a week scrambling because someone shipped a "small cleanup" without versioning? The 40 broken pipelines, the blocked security hotfix, the trust that took months to rebuild? That's what bad API versioning looks like. It's very visible.

Good API versioning is invisible. Consumers barely notice migrations because they're well-communicated, well-supported, and well-timed. The new version shows up with deprecation warnings months in advance. The migration guide makes the change trivial. By the time sunset arrives, everyone's already moved on.

Getting there requires treating internal APIs with the same discipline as external ones. Internal customers deserve predictable, well-communicated changes - arguably more so, because they can't switch providers when you break them. Use semantic versioning to signal intent clearly. Follow a structured deprecation process with real timelines and escalating communication. Invest in migration support that makes adoption easy: guides, codemods, adapters, validators.

✓ **SUCCESS**

Good API versioning is invisible - consumers barely notice migrations because they're well-communicated, well-supported, and well-timed. Bad versioning is very visible: broken pipelines, emergency meetings, and angry Slack messages. Invest in the former to avoid the latter.

The payoff is substantial. Trust with internal teams that makes future changes easier. Faster adoption of new versions because consumers know what to expect. Reduced support burden because migrations are self-service. And the platform team's time spent on planned work instead of emergency migrations.

Quick Reference

#	Parameter	Value
1	Major version sunset notice	12 months
2	Minor feature deprecation	6 months
3	Security emergency minimum	30 days
4	Breaking change advance notice	90 days
5	Feedback period for proposals	30 days
6	Target v2 adoption at 90 days	> 30%
7	Target v2 adoption at 30 days	> 80%
8	Target v2 adoption at sunset	100%

Key deprecation timeline thresholds.

Further Reading

- RFC 8594: The Sunset HTTP Header Field <https://datatracker.ietf.org/doc/html/rfc8594> - The IETF standard for machine-readable deprecation headers
- Stripe's API Versioning <https://stripe.com/docs/api/versioning> - Date-based versioning done well, with excellent migration tooling
- GitHub's Breaking Changes Policy <https://docs.github.com/en/rest/overview/breaking-changes> - Clear communication patterns for a large API surface
- jscodeshift <https://github.com/facebook/jscodeshift> - Facebook's toolkit for writing JavaScript/TypeScript codemods

Copyright © 2023 Webstack Builders, Inc.

The text, diagrams, and images in this work are licensed under CC BY-NC 4.0

All code samples in this article are licensed under the MIT License. Feel free to use, modify, and distribute them in any project.

https://www.webstackbuilders.com/articles/internal-platform-api-versioning-deprecation-breaking-changes

Webstack Builders
1032 E. Brandon Boulevard, Suite 5230
Brandon, FL 33511

support@webstackbuilders.com
Toll Free (888) 987-1881
Local (302) 608-6864