PuigFlows

IA, automatizacion e ingenieria de software

Versionado de APIs y Contract Testing: Cambios Compatibles, OpenAPI en CI, Pact y Retirada con Sunset

API Versioning and Contract Testing: Safe Changes, OpenAPI in CI, Pact, and Retiring Endpoints with Sunset

Autor: Josué Puig

Tipo: Guia · Nivel: Intermedio · Categoria: Proyectos Reales
Tiempo de lectura estimado: 55 min por idioma
Edicion: 14 de septiembre de 2026
Contenido bilingue: Espanol (ES) y English (EN)

VERSION EN ESPANOL

Versionado de APIs y Contract Testing: Cambios Compatibles, OpenAPI en CI, Pact y Retirada con Sunset

El problema: coordinar despliegues no escala

Dos servicios y un cambio. El equipo de *orders-api* renombra un campo, lo avisa por Slack, y el equipo de *checkout-web* despliega el mismo martes a las seis. Funciona. Con cinco servicios ya hace falta una hoja de cálculo. Con veinte, la coordinación se convierte en el cuello de botella: cada cambio pequeño espera a una ventana común y, cuando algo se rompe, no hay forma barata de saber quién rompió a quién.

El contract testing invierte el planteamiento. En lugar de descubrir la incompatibilidad en un entorno de integración —o en producción, a las tres de la mañana— la conviertes en un fallo de CI del commit que la introduce. El versionado de API resuelve la otra mitad: qué haces cuando el cambio incompatible es inevitable. No cómo evitarlo, sino cómo darle a tus consumidores un camino de salida con fecha.

Esta guía cubre las tres piezas que rara vez se implementan juntas: saber qué cambios rompen de verdad, detectarlos automáticamente antes del merge, y retirar lo viejo con un procedimiento en lugar de con un correo esperanzado.

Qué es de verdad un cambio incompatible

La intuición habitual —«añadir es seguro, quitar no»— acierta menos de la mitad de las veces. La regla real depende de la dirección del dato: **lo que el servidor recibe** y **lo que el servidor devuelve** se rigen por reglas opuestas.

En la petición, el servidor es el consumidor

Relajar es seguro; restringir rompe.

- Añadir un campo **opcional**: seguro.
- Añadir un campo **obligatorio**: rompe. Todos los clientes existentes empiezan a recibir 400 de golpe.
- Convertir en opcional un campo antes obligatorio: seguro.
- Ampliar el conjunto de valores aceptados en un enum: seguro. Reducirlo: rompe.
- Relajar una validación (subir un `maxLength`, ampliar un rango): seguro. Endurecerla: rompe, y además rompe en silencio para el subconjunto concreto de clientes que ya enviaba valores fuera del nuevo límite.

En la respuesta, las reglas se invierten

- Quitar un campo o renombrarlo: rompe.
- Añadir un campo: *casi* siempre seguro. Rompe si algún cliente valida con un esquema estricto o deserializa con `additionalProperties: false`.
- Ampliar un enum devuelto: **rompe**. Este es el que más se cuela. Si devuelves `status` con tres valores y añades un cuarto, cualquier cliente con un `switch` exhaustivo, un `match` sin rama por defecto o un

enum tipado se cae la primera vez que ve el valor nuevo.
- Hacer `nullable` un campo que nunca lo fue: rompe.
- Cambiar el tipo, incluso entre representaciones «equivalentes» como un id numérico que pasa a string: rompe.

Y después está la capa que ningún esquema captura: cambiar el *significado* de un campo sin cambiar su forma, devolver códigos de error distintos ante el mismo fallo, bajar el máximo de paginación de 1000 a 100, o convertir un endpoint de 200 ms en uno de 4 s que dispara los timeouts río arriba. Son incompatibilidades reales que ninguna herramienta de diff detecta. Volvemos sobre ellas en la sección de errores recurrentes.

El lector tolerante

Buena parte de lo anterior deja de doler si los clientes siguen el patrón *tolerant reader*: leer sólo los campos que se usan, ignorar los desconocidos y no reventar ante un valor de enum no previsto.

```
from enum import Enum
from pydantic import BaseModel, ConfigDict, field_validator


class OrderStatus(str, Enum):
    PENDING = "pending"
    SHIPPED = "shipped"
    DELIVERED = "delivered"
    UNKNOWN = "unknown"  # valvula de escape para estados que aun no conocemos


class Order(BaseModel):
    # Ignorar campos desconocidos ya es el comportamiento por defecto de Pydantic,
    # pero dejarlo explicito evita que alguien lo cambie a "forbid" sin medir el impacto.
    model_config = ConfigDict(extra="ignore")

    id: str
    status: OrderStatus
    total_cents: int

    @field_validator("status", mode="before")
    @classmethod
    def tolerate_unknown_statuses(cls, value: str) -> str:
        known = {member.value for member in OrderStatus}
        return value if value in known else OrderStatus.UNKNOWN.value
```

Ese `UNKNOWN` es toda la diferencia entre «el proveedor añadió un estado» y «el checkout devolvió 500 durante cuarenta minutos». Ojo: tolerar no es adivinar. Si la lógica de negocio necesita distinguir el estado nuevo, seguir arrancando sólo compra tiempo. Pero tiempo es exactamente lo que necesitas: tiempo para desplegar el soporte real sin prisa y sin incidente.

Conviene además que el validador emita una métrica cuando cae en la rama `UNKNOWN`. Un contador con etiqueta por campo y valor desconocido convierte un fallo silencioso en una alerta suave: «el proveedor está devolviendo algo que no modelamos».

OpenAPI como contrato ejecutable

Un fichero OpenAPI escrito a mano y actualizado «cuando da tiempo» no es un contrato: es documentación obsoleta con formato YAML. Para que sirva de algo tiene que cumplir dos condiciones: generarse desde el código (o generar el código), y compararse contra la versión anterior en cada pull request.

Con FastAPI la primera condición sale gratis. Lo que falta es exportar el esquema como un artefacto versionado en el repositorio:

```
# scripts/export_openapi.py
import json
import sys

from app.main import app


def main() -> int:
    spec = app.openapi()
    # sort_keys hace que el diff de git sea legible: sin el, cualquier reordenacion
    # interna del framework produce un diff de 400 lineas que nadie revisa.
    with open("openapi.json", "w", encoding="utf-8") as fh:
        json.dump(spec, fh, indent=2, sort_keys=True, ensure_ascii=False)
        fh.write("\n")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

Con `openapi.json` committeado, cualquier cambio de la API aparece en el diff del pull request, donde un humano puede verlo antes de que lo vea un cliente.

Detectar los cambios incompatibles en CI

oasdiff compara dos especificaciones y clasifica cada diferencia. Soporta OpenAPI 3.0, 3.1 y 3.2, y cubre cientos de reglas de compatibilidad, incluidas las contraintuitivas de la sección anterior —sí, detecta la ampliación de un enum de respuesta—.

```
# .github/workflows/api-contract.yml
name: API contract
on: pull_request

jobs:
  breaking-changes:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Especificacion de la rama base
        run: git show origin/${{ github.base_ref }}:openapi.json > /tmp/base.json

      - name: Regenerar la especificacion desde el codigo
        run: |
          pip install -e .
          python scripts/export_openapi.py

      # Sin este paso todo lo demas es teatro: si el fichero committeado no refleja
      # el codigo, el diff sale vacio y el check pasa en verde mintiendo.
      - name: La especificacion committeada esta al dia
        run: git diff --exit-code openapi.json

      - name: Buscar cambios incompatibles
        uses: oasdiff/oasdiff-action/breaking@main
        with:
          base: /tmp/base.json
```

```
        revision: openapi.json
        fail-on: ERR
```

Tres detalles separan un check decorativo de uno útil:

- `git diff --exit-code openapi.json` impide que el contrato se quede atrás. Es el paso que todo el mundo omite y el que hace que el resto funcione.
- `fail-on: ERR` bloquea sólo lo incompatible. Los cambios menores pasan y quedan registrados como aviso, que es lo que quieres: la herramienta no debe convertirse en algo que se ignora por ruidoso.
- Cuando el cambio incompatible es intencionado y ya está negociado con los consumidores, se aprueba de forma explícita: una etiqueta en el PR que salta el gate, o `--exclude-elements` acotado a ese path concreto. El objetivo no es que romper sea imposible, sino que requiera una decisión visible y firmada por alguien.

Donde OpenAPI no llega: contract testing dirigido por el consumidor

OpenAPI describe lo que la API *puede* devolver. No describe lo que cada consumidor *usa*. La diferencia es enorme en ambas direcciones.

Si tu respuesta tiene cuarenta campos y ningún cliente lee `legacy_shipping_code`, quitarlo es un cambio incompatible según el esquema y una limpieza inofensiva en la realidad. Al revés también ocurre: un consumidor depende de que `items` venga ordenado por fecha de creación, algo que el esquema no expresa en ninguna parte y que puedes romper sin tocar una línea del OpenAPI.

El contract testing dirigido por el consumidor (CDC, y en la práctica Pact) resuelve esto invirtiendo quién escribe el contrato. Cada consumidor declara, en forma de test ejecutable, exactamente lo que necesita. La unión de esas expectativas es el contrato que el proveedor debe cumplir.

Lado consumidor

```
# checkout/tests/contract/test_orders_api.py
import pytest
from pact.v3 import Pact, match

from checkout.clients.orders import OrdersClient


@pytest.fixture(scope="module")
def pact() -> Pact:
    return Pact("checkout-web", "orders-api").with_specification("V4")


def test_get_order_returns_the_fields_checkout_needs(pact: Pact) -> None:
    # Solo los campos que checkout lee de verdad. Los otros 36 de la respuesta
    # real no entran en el contrato: el proveedor debe poder cambiarlos.
    expected = {
        "id": match.str("ord_42"),
        "status": match.regex("shipped", regex=r"pending|shipped|delivered"),
        "total_cents": match.int(1999),
        "currency": match.str("EUR"),
    }

    (
        pact.upon_receiving("una peticion del pedido ord_42")
```

```
        .given("el pedido ord_42 existe y esta enviado")
        .with_request("GET", "/v1/orders/ord_42")
        .will_respond_with(200)
        .with_body(expected, content_type="application/json")
    )

    with pact.serve() as srv:
        order = OrdersClient(base_url=str(srv.url)).get_order("ord_42")

    assert order.total_cents == 1999
    assert order.status == "shipped"
```

Dos cosas que hacen que este test valga algo:

- Los `match.*` comprueban **tipo y forma**, no valores literales. El proveedor puede devolver cualquier identificador válido sin romper nada. Un contrato escrito con valores fijos degenera en un test de snapshot y falla cada semana por motivos que no son incompatibilidades.
- `given(...)` declara el estado que el proveedor tendrá que preparar antes de verificar. No es decorativo: es el punto donde la mayoría de las implementaciones se tuercen, porque acaba siendo un endpoint del proveedor que inserta filas a mano. Merece la pena diseñarlo bien desde el principio, con estados pequeños y parametrizados en lugar de un fixture gigante compartido.

Lado proveedor

El proveedor descarga los contratos de todos sus consumidores desde el broker y los reproduce contra una instancia real del servicio:

```
# orders/tests/contract/test_provider.py
import os

from pact.v3 import Verifier


def test_verify_consumer_contracts(running_app_url: str) -> None:
    verifier = (
        Verifier("orders-api")
        .add_transport(url=running_app_url)
        # Endpoint interno que monta el estado pedido con given(...) y lo deshace
        # despues. Solo se registra en los entornos de test, nunca en produccion.
        .set_state(f"{running_app_url}/_pact/provider-state", teardown=True)
    )

    (
        verifier.broker_source(
            os.environ["PACT_BROKER_URL"],
            token=os.environ["PACT_BROKER_TOKEN"],
            selector=True,
        )
        .consumer_versions(deployed_or_released=True)
        .build()
    )

    verifier.verify()
```

La selección de versiones es la parte que casi todo el mundo se salta y luego echa de menos. `deployed_or_released=True` significa «verifica contra las versiones de los consumidores que ahora mismo están desplegadas o publicadas en algún entorno». Sin ese filtro tienes dos malas opciones: verificar contra el último contrato subido, y que te bloquee un consumidor que aún está experimentando en una rama; o

verificar contra todo el histórico, y que te bloquee una versión que nadie ejecuta desde hace ocho meses.

La pieza que lo hace seguro: can-i-deploy

Una verificación en verde dice que *esas dos versiones concretas* son compatibles. No dice si la versión que estás a punto de desplegar es compatible con lo que hay corriendo ahora mismo en producción. Eso sólo lo sabe el broker:

```
# Antes de desplegar el proveedor
pact-broker can-i-deploy \
  --pacticipant orders-api \
  --version "$GIT_SHA" \
  --to-environment production \
  --retry-while-unknown 12 \
  --retry-interval 10

# Y cuando el despliegue termina, se registra el hecho
pact-broker record-deployment \
  --pacticipant orders-api \
  --version "$GIT_SHA" \
  --environment production
```

`record-deployment` parece burocracia y es justo lo contrario. Sin ese registro el broker no sabe qué versión corre en cada entorno, y `can-i-deploy` no puede responder nada útil. Un despliegue que no se registra convierte toda la infraestructura de contratos en teatro caro.

`--retry-while-unknown` cubre una carrera muy habitual: el pipeline del proveedor llega a la comprobación antes de que el pipeline del consumidor haya terminado de publicar su resultado de verificación. Sin reintentos, esa carrera se manifiesta como fallos intermitentes que la gente acaba resolviendo relanzando el job, que es la peor forma posible de usar un gate de seguridad.

Cómo introducirlo sin parar la fábrica

Si lo activas todo de golpe, el primer consumidor que añada una expectativa nueva dejará rojo el pipeline de un proveedor que no ha hecho nada mal. Para eso existen dos mecanismos, y conviene encenderlos desde el día uno:

- **Pending pacts.** Un contrato que el proveedor no ha verificado nunca con éxito se reporta en la salida del build, pero no lo hace fallar. En cuanto el proveedor lo verifica una primera vez, deja de estar pendiente y pasa a ser vinculante. Es lo que permite que el consumidor escriba primero la expectativa y el proveedor la implemente después.
- **WIP pacts.** El proveedor recoge además contratos procedentes de ramas de consumidores en desarrollo, también sin bloquear. Sirve para que el consumidor sepa, antes incluso de abrir el pull request, si lo que necesita ya funciona.

El orden de adopción que funciona en la práctica: empieza por el par de servicios que más incidentes de integración genera, deja los contratos en pending durante un par de sprints para que el equipo vea el valor sin sufrir el gate, y sólo entonces pon `can-i-deploy` como paso obligatorio del pipeline.

¿Y si el proveedor es de terceros?

CDC exige que el proveedor ejecute la verificación, así que no sirve para APIs que no controlas. La alternativa son los **contratos bidireccionales**: el proveedor publica su OpenAPI, el consumidor publica su pacto, y el broker comprueba que el segundo esté contenido en el primero. Es una garantía más débil

—compara esquemas, no comportamiento real— pero funciona sin ninguna cooperación del otro lado, y es mucho mejor que nada.

Estrategias de versionado

Cuando el cambio incompatible es inevitable, hay que ofrecer las dos versiones a la vez durante un tiempo. Cómo el cliente elige cuál quiere es la decisión de versionado, y hay tres opciones reales:

- **En la ruta** (`/v1/orders`, `/v2/orders`). Visible en logs, trazas, dashboards y CDN; trivial de enrutar en el gateway; trivial de probar con curl. A cambio, viola la idea de que una URI identifica un recurso: el mismo pedido tiene dos direcciones. En la práctica es lo que usa casi todo el mundo, y con razón.
- **En una cabecera propia** (`X-API-Version: 2026-09-14`). Mantiene las URIs estables y permite versionar por fecha, que escala mucho mejor que los enteros cuando tienes decenas de cambios pequeños. El coste es operativo: la versión desaparece de los logs de acceso salvo que la añadas a mano, y la caché necesita `Vary` para no servir la respuesta equivocada.
- **En el tipo de contenido** (`Accept: application/vnd.acme.order+json;version=2`). Es lo más correcto según HTTP y lo más incómodo de usar. Reserva esta opción para APIs con consumidores sofisticados y necesidad real de negociar representaciones.

Cualquiera de las tres funciona. La que no funciona es no elegir ninguna y confiar en que nunca habrá un cambio incompatible.

Versiona lo mínimo posible

El error caro no suele ser elegir mal el mecanismo, sino elegir mal la **granularidad**. Saltar toda la API a `/v2` porque cambió un endpoint obliga a todos los consumidores a migrar todo, incluido lo que no ha cambiado. El resultado previsible: nadie migra, mantienes dos superficies completas durante años, y cada funcionalidad nueva hay que implementarla dos veces.

La alternativa aburrida y correcta es evolucionar en su sitio y versionar sólo el recurso afectado. Añade el campo nuevo junto al viejo y rellena ambos durante la transición. Si de verdad necesitas dos formas incompatibles, expón `/v2/orders` y deja el resto de la API en `/v1`. Un consumidor que sólo usa `/v1/invoices` no debería enterarse siquiera de que hubo una migración.

Retirar una versión: Deprecation, Sunset y datos

Publicar `/v2` es la parte fácil. La difícil es apagar `/v1`, y casi nadie lo consigue porque lo intenta con un correo en lugar de con un procedimiento.

HTTP tiene dos cabeceras estándar para esto, y son cosas distintas:

- `Deprecation` (RFC 9745): «este recurso está deprecado». Su valor es un *structured field* de tipo Date, es decir una arroba seguida de segundos Unix. Puede estar en el pasado (ya está deprecado) o en el futuro (lo estará). Deprecar no cambia el comportamiento del recurso: sigue funcionando igual.
- `Sunset` (RFC 8594): «a partir de esta fecha es probable que el recurso deje de responder». Su valor es una fecha HTTP en formato IMF-fixdate. La RFC 9745 es explícita: la fecha de `Sunset` nunca puede ser anterior a la de `Deprecation`.

Además, un `Link` con `rel="deprecation"` apunta a la documentación de la migración, y `rel="successor-version"` a lo que hay que usar en su lugar. Entre las tres cabeceras, un cliente puede detectar y diagnosticar la situación sin que nadie lea un changelog.

```
from datetime import datetime, timezone
from email.utils import format_datetime

from fastapi import FastAPI, Request

app = FastAPI()

DEPRECATED_AT = datetime(2026, 9, 14, tzinfo=timezone.utc)
SUNSET_AT = datetime(2027, 3, 14, tzinfo=timezone.utc)
DEPRECATED_PREFIXES = ("/v1/orders",)

DOCS = "https://api.acme.com/docs/migrations/orders-v2"
SUCCESSOR = "https://api.acme.com/v2/orders"


@app.middleware("http")
async def deprecation_headers(request: Request, call_next):
    response = await call_next(request)
    if request.url.path.startswith(DEPRECATED_PREFIXES):
        # RFC 9745: structured field Date = "@" + segundos unix
        response.headers["Deprecation"] = f"@{int(DEPRECATED_AT.timestamp())}"
        # RFC 8594: HTTP-date en IMF-fixdate (usegmt produce "GMT", no "+0000")
        response.headers["Sunset"] = format_datetime(SUNSET_AT, usegmt=True)
        response.headers["Link"] = (
            f'<{DOCS}>; rel="deprecation"; type="text/html", '
            f'<{SUCCESSOR}>; rel="successor-version"'
        )
    return response
```

Una respuesta deprecada queda así:

```
HTTP/1.1 200 OK
Content-Type: application/json
Deprecation: @1789344000
Sunset: Sun, 14 Mar 2027 00:00:00 GMT
Link: <https://api.acme.com/docs/migrations/orders-v2>; rel="deprecation"; type="text/html",
      <https://api.acme.com/v2/orders>; rel="successor-version"
```

La parte que decide si podrás apagarlo: medir

Una fecha de sunset puesta a ojo se incumple siempre. Lo que permite apagar de verdad es saber, con nombres y apellidos, quién sigue llamando:

```
from prometheus_client import Counter

DEPRECATED_CALLS = Counter(
    "deprecated_api_calls_total",
    "Llamadas a endpoints deprecados, por consumidor",
    ["consumer", "route", "api_version"],
)


@app.middleware("http")
async def track_deprecated_usage(request: Request, call_next):
    response = await call_next(request)
    if request.url.path.startswith(DEPRECATED_PREFIXES):
        DEPRECATED_CALLS.labels(
            # El consumidor sale del token, no del User-Agent: el User-Agent
            # es "python-requests/2.32" en el 80% de los casos y no identifica a nadie.
            consumer=getattr(request.state, "client_id", "unknown"),
```

```
            # La plantilla de ruta, no la URL concreta: /v1/orders/{order_id}
            # y no /v1/orders/ord_42, o la cardinalidad de la metrica explota.
            route=request.scope.get("route").path if request.scope.get("route") else "unmatched",
            api_version="v1",
        ).inc()
    return response
```

Con eso, la consulta que contesta la única pregunta importante:

```
# Quien ha llamado a /v1/orders en los ultimos 7 dias
sum by (consumer) (
  increase(deprecated_api_calls_total{api_version="v1", route=~"/v1/orders.*"}[7d])
) > 0
```

Con esa lista puedes hacer lo único que funciona: hablar con cada equipo por su nombre, en lugar de anunciar en un canal general y esperar. Y cuando la lista baja a cero o sólo quedan rezagados, añade **brownouts** antes del apagado definitivo: ventanas programadas y anunciadas en las que el endpoint devuelve `410 Gone` durante una hora. Un brownout encuentra en un día a los consumidores que ninguna métrica atribuyó bien, y lo hace en horario laboral en lugar de a las tres de la mañana del día del apagado.

Los contratos no son sólo HTTP

Si publicas eventos, el esquema del evento es un contrato igual de vinculante, con una diferencia importante: no puedes desplegar a productores y consumidores a la vez, y los mensajes viejos siguen en el topic después de que cambies el esquema. Por eso los registros de esquemas ofrecen modos de compatibilidad, y elegir el correcto es una decisión sobre *quién actualiza primero*:

- **BACKWARD**: un consumidor con el esquema nuevo puede leer datos escritos con el viejo. Actualizas primero los consumidores. Es el modo por defecto y el que quieres casi siempre.
- **FORWARD**: un consumidor con el esquema viejo puede leer datos escritos con el nuevo. Actualizas primero los productores. Útil cuando no controlas a todos los consumidores.
- **FULL**: ambas cosas. Sólo admite cambios muy conservadores (añadir o quitar campos opcionales con valor por defecto), y es lo que necesitas si el orden de despliegue no está garantizado.

El equivalente de `can-i-deploy` aquí es ejecutar la comprobación de compatibilidad del registro en CI, contra el esquema registrado en producción, antes de hacer merge. Mismo principio: que el fallo aparezca en el pull request y no en el consumidor.

Los errores también son contrato: RFC 9457

Casi todo lo que se escribe sobre versionado habla del camino feliz: el esquema de la respuesta 200. Y luego, en producción, el cambio que rompe a un cliente es otro. Alguien reescribe el manejador de errores y lo que antes era un objeto con una clave `error` pasa a ser una lista bajo la clave `errors`. El 200 no ha cambiado ni una coma, oasdiff no protesta porque los errores nunca se documentaron, y el consumidor que decidía si reintentar leyendo `error` empieza a reintentar peticiones que no van a funcionar nunca.

Los errores son parte del contrato exactamente igual que los éxitos. Merecen el mismo trato: un formato estable, documentado en OpenAPI y vigilado en CI. RFC 9457, *Problem Details for HTTP APIs* (julio de 2023, deja obsoleto el RFC 7807), define ese formato y evita que cada equipo invente el suyo. El cuerpo va con `Content-Type: application/problem+json` y tiene cinco campos estándar:

- `type`: una URI que identifica la *clase* de problema. Es el único campo pensado para que lo lea una máquina, y por tanto el que de verdad forma parte del contrato.
- `title`: resumen legible y corto de esa clase de problema. Se puede reescribir o traducir sin romper nada.
- `status`: el código HTTP, duplicado en el cuerpo para que sobreviva a proxies y a logs que sólo guardan el body.
- `detail`: explicación de *esta* ocurrencia concreta. También para humanos.
- `instance`: URI de la ocurrencia. Sirve para correlacionar con trazas si le metes el identificador de la petición.

Además puedes añadir campos propios —las *extensiones*—, y ahí es donde el formato se vuelve útil de verdad: `balance` y `required` en un error de saldo, `retry_after_seconds` en un 429, `field` y `rule` en un error de validación. Las extensiones siguen la misma regla aditiva que el resto de la respuesta: añadir una es seguro, quitarla rompe.

Una implementación mínima en FastAPI que centraliza el formato en un solo sitio:

```
# app/errors.py
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

PROBLEM_BASE = "https://api.example.com/problems/"


class ProblemError(Exception):
    """Error de dominio que ya sabe cómo serializarse."""

    def __init__(self, slug: str, title: str, status: int, detail: str = "", **extensions):
        self.slug = slug
        self.title = title
        self.status = status
        self.detail = detail
        self.extensions = extensions


def install(app: FastAPI) -> None:
    @app.exception_handler(ProblemError)
    async def handle(request: Request, exc: ProblemError):
        body = {
            "type": PROBLEM_BASE + exc.slug,
            "title": exc.title,
            "status": exc.status,
            "detail": exc.detail,
            "instance": request.url.path,
        }
        body.update(exc.extensions)
        return JSONResponse(
            status_code=exc.status,
            content=body,
            media_type="application/problem+json",
        )
```

Lanzarlo desde el dominio queda razonablemente limpio:

```
raise ProblemError(
    slug="insufficient-funds",
    title="Saldo insuficiente",
    status=422,
    detail="La cuenta no cubre el importe del pedido.",
```

```
    balance="12.40",
    required="49.90",
)
```

Y el cliente recibe algo que puede tratar sin adivinar:

```
HTTP/1.1 422 Unprocessable Content
Content-Type: application/problem+json

{
  "type": "https://api.example.com/problems/insufficient-funds",
  "title": "Saldo insuficiente",
  "status": 422,
  "detail": "La cuenta no cubre el importe del pedido.",
  "instance": "/v1/orders",
  "balance": "12.40",
  "required": "49.90"
}
```

Tres reglas que convierten esto en un contrato y no en un formato bonito:

- **La URI de `type` es inmutable.** Cambiar `insufficient-funds` por `payment-declined` es un cambio incompatible tan real como renombrar un campo de la respuesta, porque hay clientes ramificando sobre ese valor. Si necesitas una clase nueva, añádela; la vieja se deprecia y se retira con el mismo procedimiento que un endpoint.
- **No pongas nunca en `detail` algo que el cliente necesite parsear.** En cuanto alguien extrae un número de ahí con una expresión regular, `detail` deja de ser texto para humanos y se convierte en API. Si el dato hace falta, es una extensión con su propio campo.
- **Documenta los errores en OpenAPI.** Si no están en el esquema, oasdiff no puede ver que has cambiado uno, y todo el trabajo de la sección anterior se queda a medias.

```
from pydantic import BaseModel


class Problem(BaseModel):
    type: str
    title: str
    status: int
    detail: str = ""
    instance: str = ""


PROBLEM_RESPONSES = {
    409: {"model": Problem, "content": {"application/problem+json": {}}},
    422: {"model": Problem, "content": {"application/problem+json": {}}},
}


@app.post("/v1/orders", responses=PROBLEM_RESPONSES)
def create_order(...):
    ...
```

Un detalle que ahorra disgustos: mantén en el repositorio una tabla con todas las URIs de `type` que publica el servicio, y un test que recorra el código buscando `ProblemError` y falle si aparece un `slug` que no está en esa tabla. Es cinco minutos de trabajo y es lo que impide que en seis meses haya cuarenta clases de problema, ocho de ellas duplicadas y ninguna documentada.

Paginación, filtros y ordenación: el contrato invisible

La paginación es el sitio donde más contratos se rompen sin que nadie lo llame romper un contrato. El esquema de la respuesta no cambia: sigue habiendo una lista de pedidos y un total. Lo que cambia es el *comportamiento*, y no hay fichero OpenAPI que lo capture.

Empecemos por lo que casi nadie considera un cambio incompatible y lo es:

- **Bajar el `limit` máximo.** Si aceptabas `limit=500` y ahora devuelves 400 por encima de 100, has roto a todos los que paginaban de 500 en 500. Que lo hicieras por proteger la base de datos es una razón excelente para el cambio y ninguna razón para saltarte el procedimiento de deprecación.
- **Cambiar el tamaño de página por defecto.** Un cliente que llama sin `limit` y asume 50 empieza a recibir 20. Si además paraba de paginar al recibir menos elementos de los esperados, ahora procesa el 40 % de los datos y no falla: los pierde en silencio, que es peor.
- **Cambiar el orden por defecto.** Si nunca documentaste el orden, técnicamente eres libre. En la práctica hay un cliente que muestra «los últimos pedidos» confiando en que el primero es el más reciente, y ese cliente se rompe sin un solo error en los logs.
- **Volver estricto un filtro que era laxo.** Rechazar con 400 un parámetro desconocido que antes ignorabas es endurecer la validación de la petición: el mismo caso que añadir un campo obligatorio.

Regla práctica: los valores por defecto de la paginación y del orden son parte del contrato desde el primer día, estén documentados o no. Escríbelos en el esquema, incluidos `default` y `maximum`, y trátalos como cualquier otro campo.

De offset a cursor sin romper a nadie

El otro problema de `offset`/`limit` no es de contrato sino de corrección: con escrituras concurrentes, la página 2 se solapa con la 1 o se salta filas, porque el desplazamiento se calcula sobre un conjunto que se mueve entre peticiones. Y el coste crece linealmente con el offset, porque la base de datos tiene que leer y descartar todas las filas anteriores. La alternativa es la paginación por *keyset*: en lugar de decir «sáltate 10.000», dices «dame lo que va después de este punto exacto».

```
import base64
import json
from datetime import datetime


def encode_cursor(created_at: datetime, order_id: str) -> str:
    raw = json.dumps({"t": created_at.isoformat(), "id": order_id}, separators=(",", ":"))
    return base64.urlsafe_b64encode(raw.encode()).decode().rstrip("=")


def decode_cursor(cursor: str) -> tuple[datetime, str]:
    padded = cursor + "=" * (-len(cursor) % 4)
    data = json.loads(base64.urlsafe_b64decode(padded))
    return datetime.fromisoformat(data["t"]), data["id"]
```

La consulta usa la tupla completa para desempatar, que es el detalle que hace que funcione con timestamps repetidos:

```
-- El indice tiene que coincidir exactamente con el ORDER BY
CREATE INDEX CONCURRENTLY idx_orders_customer_created
  ON orders (customer_id, created_at DESC, id DESC);

SELECT id, created_at, total
FROM orders
WHERE customer_id = $1
  AND (created_at, id) < ($2, $3)
ORDER BY created_at DESC, id DESC
LIMIT $4;
```

La migración se hace igual que cualquier otra evolución compatible: el endpoint acepta las dos formas durante un tiempo y la vieja se marca como deprecada.

```
@app.get("/v1/orders")
def list_orders(
    customer_id: str,
    limit: int = Query(default=25, ge=1, le=100),
    cursor: str | None = None,
    offset: int | None = Query(default=None, deprecated=True),
):
    if cursor is not None and offset is not None:
        raise ProblemError(
            slug="conflicting-pagination",
            title="Parámetros de paginación incompatibles",
            status=400,
            detail="Usa cursor u offset, no ambos.",
        )
    if offset is not None:
        DEPRECATED_PARAM.labels(param="offset", consumer=consumer_of(request)).inc()
        rows = query_by_offset(customer_id, limit, offset)
    else:
        rows = query_by_cursor(customer_id, limit, cursor)

    next_cursor = encode_cursor(rows[-1].created_at, rows[-1].id) if len(rows) == limit else None
    return {"items": [serialize(r) for r in rows], "next_cursor": next_cursor}
```

Fíjate en el contador `DEPRECATED_PARAM`: es el mismo truco de la sección de sunset aplicado a un parámetro en vez de a una ruta entera. Sin él, la fecha en la que puedes borrar la rama de `offset` es una conjetura.

Y una advertencia sobre el cursor: documenta explícitamente que es **opaco**, y cúmplelo tú también. En cuanto un cliente descubre que es base64 de un JSON, lo decodifica, se inventa cursores y te ata a ese formato interno para siempre. Si te preocupa, fírmalo con HMAC o cífralo; una cadena que no se puede construir desde fuera es una cadena que puedes cambiar cuando quieras.

Expand / contract: renombrar un campo sin parar la fábrica

Hasta aquí hemos hablado de detectar cambios incompatibles y de avisar de ellos. Falta la parte mecánica: cómo se ejecuta de verdad un cambio que afecta a la vez a la base de datos, al código y al contrato público, sin ventana de mantenimiento y sin que un rollback te deje con datos que el código anterior no entiende.

El patrón se llama *expand / contract* (o *parallel change*) y su idea es simple: entre el estado viejo y el nuevo hay un periodo en el que conviven los dos, y ese periodo dura tanto como haga falta. Lo que lo hace seguro no es el patrón en sí, sino una regla que conviene grabar en piedra: **cada despliegue debe poder revertirse sin revertir el esquema**. Si para hacer rollback de un release tienes que deshacer una migración, no tienes rollback, tienes una esperanza.

Veamos el caso concreto de sustituir `shipping_country` (texto libre: «Spain», «ES», «españa») por `shipping_country_code` (ISO 3166-1 alpha-2). Seis fases, cada una en su propio despliegue.

Fase 1 — Expandir el esquema

Una migración estrictamente aditiva. Columna nueva, nullable, sin valor por defecto y sin restricciones. El código antiguo sigue funcionando porque no sabe que existe.

```
ALTER TABLE orders ADD COLUMN shipping_country_code char(2);

-- CONCURRENTLY no bloquea escrituras; a cambio no puede ir dentro
-- de una transaccion, asi que va en su propia migracion.
CREATE INDEX CONCURRENTLY idx_orders_shipping_country_code
  ON orders (shipping_country_code);
```

Nada de `NOT NULL` todavía, y nada de `DEFAULT` con una función volátil: en tablas grandes, según la versión del motor, eso puede reescribir la tabla entera y dejarte el servicio parado justo cuando pensabas que ibas a hacer un cambio inocuo.

Fase 2 — Escritura doble

El servicio escribe los dos campos en cada operación. Sigue leyendo del viejo. Este despliegue es completamente reversible: si lo revertimos, el campo nuevo simplemente deja de actualizarse.

```
COUNTRY_ALIASES = {"spain": "ES", "españa": "ES", "es": "ES", "france": "FR", "fr": "FR"}


def normalize_country(raw: str | None) -> str | None:
    if not raw:
        return None
    return COUNTRY_ALIASES.get(raw.strip().lower())


def save_order(conn, order) -> None:
    conn.execute(
        "UPDATE orders SET shipping_country = %s, shipping_country_code = %s WHERE id = %s",
        (order.shipping_country, normalize_country(order.shipping_country), order.id),
    )
```

Fase 3 — Backfill

Las filas históricas siguen con el campo nuevo a `NULL`. Se rellenan en lotes, fuera del camino de la petición, con pausas para no ahogar la replicación.

```
import time

BATCH = 5_000


def backfill(conn) -> None:
    while True:
        rows = conn.execute(
            "SELECT id, shipping_country FROM orders "
            "WHERE shipping_country_code IS NULL AND shipping_country IS NOT NULL "
            "ORDER BY id LIMIT %s FOR UPDATE SKIP LOCKED",
```

```
            (BATCH,),
        ).fetchall()
        if not rows:
            return
        conn.executemany(
            "UPDATE orders SET shipping_country_code = %s WHERE id = %s",
            [(normalize_country(country), row_id) for row_id, country in rows],
        )
        conn.commit()
        time.sleep(0.2)
```

`FOR UPDATE SKIP LOCKED` permite que varios procesos de backfill avancen a la vez sin pisarse y sin bloquear a las escrituras normales. Y conviene registrar cuántas filas no se pudieron normalizar: ese número es el que te dice si el mapeo está completo o si hay veinte variantes de «Reino Unido» que nadie había visto.

Fase 4 — Leer del nuevo, con red

El servicio pasa a leer el campo nuevo, pero con una vuelta al viejo si el nuevo está vacío. Esa rama de respaldo lleva su propia métrica: cuando el contador lleve días a cero, el backfill está completo de verdad.

```
code = row.shipping_country_code
if code is None:
    FALLBACK_READS.inc()
    code = normalize_country(row.shipping_country)
```

Fase 5 — Publicar los dos y deprecar el viejo

Ahora, y sólo ahora, el cambio llega al contrato público. La respuesta lleva los dos campos, el viejo se marca como deprecado en OpenAPI, y las llamadas que lo usan se cuentan como vimos en la sección de sunset. Añadir un campo es compatible, así que este despliegue no rompe a nadie.

```
class OrderOut(BaseModel):
    id: str
    shipping_country_code: str | None = None
    shipping_country: str | None = Field(
        default=None,
        deprecated=True,
        description="Deprecado. Usa shipping_country_code (ISO 3166-1 alpha-2). Retirada:
          2026-12-01.",
    )
```

Fase 6 — Contraer

Cuando ningún consumidor lee el campo viejo —y eso lo dicen los contratos de Pact y el contador de uso, no una intuición—, se retira: primero del API, luego de la escritura doble, y por último de la base de datos. Los tres pasos son despliegues separados, en ese orden. El `DROP COLUMN` es el último y el único irreversible, así que va semanas después del resto, cuando ya nadie se acuerda del tema.

```
-- Semanas despues de que el API deje de exponerlo y el codigo de escribirlo
ALTER TABLE orders ALTER COLUMN shipping_country_code SET NOT NULL;
ALTER TABLE orders DROP COLUMN shipping_country;
```

Lo bonito de encajar esto con lo anterior es que las fases no dependen de que alguien se acuerde. La fase 5 pasa el gate de oasdiff porque sólo añade. La fase 6 falla en oasdiff —quitar un campo de la respuesta rompe— y ahí es donde entra `can-i-deploy`: si algún consumidor sigue esperando `shipping_country` en su pact, la retirada no llega a producción. El patrón te dice qué hacer; el pipeline te impide adelantarte.

Gobernanza con Spectral: que el contrato además sea coherente

oasdiff contesta a «¿he roto algo?». No contesta a «¿esto se parece al resto de nuestra API?». Y esa segunda pregunta importa más de lo que parece, porque la incoherencia se paga en forma de cambios incompatibles futuros: el endpoint que devolvió el identificador como número entero porque nadie lo revisó acabará migrando a string el día que los ids dejen de caber en un entero, y esa migración sí romperá a alguien.

Spectral es un linter de JSON/YAML mantenido por Stoplight que valida especificaciones OpenAPI y AsyncAPI contra reglas que tú defines. Trae un conjunto base (`spectral:oas`) con las comprobaciones estructurales, y encima pones las convenciones de tu casa.

```
# .spectral.yaml
extends: ["spectral:oas"]

rules:
  operation-operationId: error
  operation-description: warn
  operation-tags: error

  paths-kebab-case:
    description: Las rutas usan kebab-case y sustantivos en plural.
    severity: error
    given: $.paths[*]~
    then:
      function: pattern
      functionOptions:
        match: "^(/[a-z0-9-]+|/\{[a-zA-Z0-9_]+\})+$"

  ids-are-strings:
    description: Los identificadores se serializan como string, nunca como entero.
    severity: error
    given: $..properties[?(@property == 'id')].type
    then:
      function: pattern
      functionOptions:
        notMatch: "^(integer|number)$"

  errors-use-problem-json:
    description: Toda respuesta 4xx/5xx documenta application/problem+json.
    severity: error
    given: $.paths[*][*].responses[?(@property.match(/^[45]/))].content
    then:
      field: application/problem+json
      function: truthy

  collections-are-paginated:
    description: Las colecciones aceptan limit y cursor.
    severity: warn
    given: $.paths[*].get.parameters
    then:
      function: schema
      functionOptions:
        schema:
          type: array
          contains:
```

```
            type: object
            properties:
              name:
                const: limit
```

En CI va antes que oasdiff, por una razón práctica: una especificación que no pasa el lint no merece que le hagas un diff.

```
- name: Lint OpenAPI
  run: npx --yes @stoplight/spectral-cli lint openapi.json --ruleset .spectral.yaml
    --fail-severity=error
```

El error clásico al adoptarlo es activar cuarenta reglas en `error` el primer día sobre una API con seis años de historia. El resultado es un pipeline rojo que todo el mundo aprende a ignorar. La forma que funciona es un trinquete: todas las reglas entran como `warn`, el pipeline sólo falla con `error`, y cada semana se arregla una categoría y se promueve esa regla a `error`. Las rutas existentes se quedan como están —renombrarlas sería un cambio incompatible por motivos estéticos, que es justo lo que no queremos—; la regla se aplica a lo nuevo, y lo viejo se marca como excepción explícita en el ruleset, con un comentario que diga por qué.

Merece la pena añadir también un ruleset de seguridad basado en el OWASP API Security Top 10. No sustituye a una revisión de seguridad, pero coge gratis los descuidos habituales: un endpoint sin `security`, una respuesta 200 documentada donde debería haber un 401, un esquema de array sin `maxItems`.

gRPC y Protobuf: compatibilidad de cable frente a compatibilidad de código

Si parte de tu tráfico interno va por gRPC, tienes un segundo contrato con reglas propias, y la trampa está en que Protobuf parece más indulgente de lo que es. La clave es que Protobuf no serializa los nombres de los campos: serializa el *número* de campo y su tipo de cable. De ahí salen dos nociones distintas de compatibilidad que conviene no mezclar.

- **Compatibilidad de cable.** ¿Puede un binario viejo deserializar un mensaje nuevo? Renombrar un campo manteniendo su número es compatible de cable: los bytes son idénticos. Cambiar el número, o pasar un `int32` a `string`, no lo es, y el efecto no es una excepción clara sino un campo que llega con basura o vacío.
- **Compatibilidad de código generado.** ¿Sigue compilando el cliente cuando regeneras? Ese mismo renombrado que era inofensivo en el cable rompe todo el código que llamaba al accesor viejo.

Y hay un tercer nivel que pilla a mucha gente: la codificación JSON de Protobuf —la que usan Connect, grpc-gateway y la transcodificación JSON de gRPC— *sí* serializa los nombres. En cuanto expones el servicio por JSON, renombrar un campo pasa de ser gratis a ser un cambio incompatible para la mitad de tus consumidores.

`buf breaking` hace explícita esa distinción con categorías, de la más estricta a la más laxa: `FILE` (por defecto, unas 46 reglas, detecta roturas del código generado fichero a fichero), `PACKAGE` (lo mismo pero a nivel de paquete, tolera mover definiciones entre ficheros del mismo paquete), `WIRE_JSON` (rotura del cable binario o del JSON) y `WIRE` (sólo el cable binario, 15 reglas).

La elección no es de estilo: `WIRE_JSON` es la mínima defendible si algo de tu tráfico va por JSON, y `FILE` es lo correcto si publicas los `.proto` a equipos que compilan contra ellos.

```
# buf.yaml
version: v2
modules:
  - path: proto
lint:
  use:
    - STANDARD
breaking:
  use:
    - WIRE_JSON
```

La comparación en CI se hace contra la rama principal, no contra el commit anterior, para que una rama larga no vaya acumulando roturas invisibles:

```
# .github/workflows/proto-contract.yml
name: Proto contract
on: pull_request

jobs:
  breaking:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - uses: bufbuild/buf-action@v1
        with:
          breaking_against: ".git#branch=origin/main,subdir=proto"
```

Y la higiene que evita la mitad de los incidentes: cuando quites un campo, reserva su número y su nombre. Es una línea, y es lo que impide que dentro de dos años alguien reutilice el número 4 para un tipo distinto y se encuentre con mensajes antiguos en la cola que se deserializan como algo que no son.

```
syntax = "proto3";

package orders.v1;

message Order {
  // Retirados en 2026-05. No reutilizar.
  reserved 4, 7 to 9;
  reserved "legacy_shipping_code";

  string id = 1;
  string customer_id = 2;
  Money total = 3;
  OrderStatus status = 5;
  string shipping_country_code = 6;
}

enum OrderStatus {
  ORDER_STATUS_UNSPECIFIED = 0;
  ORDER_STATUS_PENDING = 1;
  ORDER_STATUS_PAID = 2;
  ORDER_STATUS_SHIPPED = 3;
}
```

Ese `ORDER_STATUS_UNSPECIFIED = 0` no es una formalidad. Los enums de proto3 son abiertos: un valor desconocido no falla al deserializar, llega como un entero que tu código no reconoce. Es exactamente el mismo problema del enum ampliado que veíamos en JSON, con la misma solución —una rama por defecto

que no asuma nada— y con una diferencia a favor: aquí la biblioteca ya te lo entrega en lugar de reventar. Lo que no te salva es de un `match` exhaustivo sin caso por defecto en el lenguaje que uses.

GraphQL: el contrato que no se puede versionar

GraphQL no tiene `/v2`, y no es un descuido de diseño: la premisa es que el cliente pide exactamente los campos que necesita, así que añadir campos nunca rompe a nadie y no hace falta duplicar el API. El precio de esa premisa es que **toda la evolución tiene que ocurrir en el mismo esquema**, sin la válvula de escape de publicar una versión nueva al lado.

Las reglas de compatibilidad también se invierten respecto a REST, y conviene tenerlas claras:

- **Añadir un campo o un tipo: seguro.** Nadie lo recibe si no lo pide.
- **Quitar o renombrar un campo: rompe** a cualquier operación que lo mencione, incluso si el cliente no usaba el valor.
- **Pasar una salida de `String!` a `String`: rompe.** El cliente generó tipos no nulos. Al revés (de `String` a `String!`) es seguro para los que leen.
- **En los argumentos de entrada las reglas se invierten**, igual que en el cuerpo de una petición REST: añadir un argumento opcional es seguro, añadir uno obligatorio rompe, y pasar un argumento de obligatorio a opcional es seguro.
- **Añadir un valor a un enum es peligroso** sin ser estrictamente incompatible: el esquema lo permite, pero el `switch` del cliente no lo contempla. El mismo tolerant reader de siempre.
- **Cambiar un campo de un tipo a una unión o a una interfaz: rompe** todas las selecciones que no usan fragmentos.

Como no hay versiones, la deprecación es el único mecanismo de retirada, y por suerte está en el propio lenguaje. La directiva `@deprecated` saca el campo de la introspección por defecto, aparece en el autocompletado de cualquier IDE con un aviso, y da un sitio natural donde poner la fecha:

```
type Order {
  id: ID!
  total: Money!
  shippingCountryCode: String

  shippingCountry: String
    @deprecated(reason: "Usa shippingCountryCode (ISO 3166-1 alpha-2). Retirada prevista:
      2026-12-01.")
}
```

La verificación en CI se hace con un diff de esquemas. GraphQL Inspector compara el esquema de la rama con el de `main` y clasifica cada cambio en seguro, peligroso o incompatible:

```
# .github/workflows/graphql-contract.yml
name: GraphQL contract
on: pull_request

jobs:
  schema-diff:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
```

```
      - run: |
          npx --yes @graphql-inspector/cli diff \
            "git:origin/main:schema.graphql" "schema.graphql" \
            --rule suppressRemovalOfDeprecatedField
```

Esa regla `suppressRemovalOfDeprecatedField` merece un párrafo, porque es el equivalente de `can-i-deploy` en este mundo. Por defecto, quitar un campo deprecado sigue contando como incompatible —y con razón: que esté deprecado no significa que nadie lo use—. La regla lo degrada a «peligroso» para que no bloquee el pipeline. Activarla sin medir el uso real es exactamente el mismo error que poner una fecha de sunset a ojo. Actívala sólo cuando tengas el otro lado resuelto: un registro de operaciones o consultas persistidas que te diga qué cliente ha pedido ese campo en los últimos treinta días.

Ese registro, por cierto, es la gran ventaja de GraphQL en esta materia. En REST sabes que alguien llamó a `/v1/orders`; averiguar si leía `shipping_country` requiere adivinar o preguntar. En GraphQL la petición *enumera* los campos, así que el uso a nivel de campo es un dato que ya tienes. Es la única tecnología de las que hemos visto donde la pregunta «¿puedo borrar esto?» tiene una respuesta exacta en lugar de una estimación.

SDKs generados: el cliente también tiene contrato

Si publicas una biblioteca cliente, has firmado un segundo contrato con su propia superficie y su propio versionado, y los dos no coinciden. La API puede seguir en `v1` mientras el SDK va por la `4.2.0`, y está bien: versionan cosas distintas.

La forma sana de gestionarlo es generar el SDK desde `openapi.json` en CI, nunca a mano, con tres reglas:

- **Fija la versión del generador.** Una actualización del generador puede cambiar nombres de métodos o la forma de los modelos sin que tu especificación se haya movido un milímetro. Si el generador flota, un lunes por la mañana publicas un major sin haberlo decidido.
- **Nunca edites el código generado.** Lo que haga falta encima va en una capa fina escrita a mano que envuelve al cliente generado. En cuanto alguien parchea el fichero generado, la siguiente regeneración lo pisa y el arreglo desaparece.
- **Deriva el salto de versión del diff, no de la sensación.** Ya tienes la señal: si oasdiff marca un cambio incompatible, el SDK sube de major. Si sólo hay adiciones, minor. Si sólo cambian descripciones o ejemplos, patch.

```
- name: Generar SDK de Python
  run: |
    npx --yes @openapitools/openapi-generator-cli@2.24.0 generate \
      -i openapi.json \
      -g python \
      -o sdk/python \
      --additional-properties=packageName=acme_orders,packageVersion=4.2.0
    git diff --stat sdk/python
```

Ese `git diff --stat` del final es más útil de lo que parece: convierte el efecto de un cambio de esquema en algo que se ve en la revisión del pull request. Un ajuste de una línea en el `.proto` o en el modelo Pydantic que toca ochenta ficheros del SDK es una señal de alarma que ningún linter te iba a dar.

Y documenta en el README la correspondencia entre versiones del SDK y versiones de la API —qué rango de majors del cliente habla con `/v1`, cuál con `/v2`—, porque es la primera pregunta que hace cualquiera que

llega a integrar y la que peor se contesta buscando en el historial de commits.

Ocho errores recurrentes

1. **Creer que añadir a la respuesta siempre es seguro.** Ampliar un enum devuelto rompe a cualquier cliente con deserialización estricta, y suele descubrirse en producción porque el entorno de staging no tiene datos con el valor nuevo. Introduce los valores de enum nuevos sólo detrás de un flag y avisa antes.
2. **El** `openapi.json` **que no se regenera.** El check de breaking changes compara dos ficheros; si el fichero no refleja el código, compara dos ficciones. El paso `git diff --exit-code` es obligatorio, no opcional.
3. **Usar Pact como test de snapshot.** Copiar la respuesta entera al contrato, con valores literales incluidos, produce un contrato que falla cada vez que el proveedor añade un campo inofensivo. El equipo aprende a ignorar el fallo y el contrato deja de aportar señal. El contrato debe contener el mínimo que consumes, con matchers de tipo.
4. **Estados del proveedor con fixtures globales.** Un único «seed de datos de test» compartido por todos los `given(...)` hace que los contratos se acoplen entre sí y que un cambio en el seed rompa verificaciones sin relación. Cada estado debe crear lo suyo y deshacerlo.
5. `can-i-deploy` **sin** `record-deployment`**.** El gate responde siempre que sí, porque el broker cree que no hay nada desplegado. Es peor que no tener gate: da una falsa sensación de seguridad.
6. **Deprecar sin medir.** Sin métrica por consumidor, la fecha de sunset se fija a ojo, llega, nadie se atreve a apagar, y la versión vieja sobrevive tres años. La métrica es lo que convierte la decisión en técnica en lugar de política.
7. **Versionar toda la API por un endpoint.** Obliga a migrar lo que no ha cambiado, con lo cual nadie migra y acabas manteniendo dos superficies completas.
8. **Cambios de comportamiento sin cambio de esquema.** Bajar el límite de paginación, cambiar el código de error de 409 a 422, dejar de ordenar por fecha, o triplicar la latencia p99. Ninguna herramienta de diff los ve. Contra esto sólo hay dos defensas: contratos de consumidor que codifiquen esas expectativas cuando importan, y la disciplina de tratar los códigos de error y los límites como parte pública de la API.

Checklist de producción

- La especificación OpenAPI se genera desde el código y CI falla si la versión committeada no coincide.
- Cada pull request ejecuta un diff de compatibilidad y bloquea los cambios incompatibles no aprobados.
- Existe un mecanismo explícito y auditable para aprobar un cambio incompatible cuando de verdad hace falta.
- Los consumidores críticos publican contratos; el proveedor los verifica en CI contra las versiones desplegadas.
- Pending y WIP pacts están activados para que un contrato nuevo no rompa el build del proveedor.
- `can-i-deploy` es un paso obligatorio antes de desplegar, y `record-deployment` se ejecuta después de cada despliegue en todos los entornos.
- Los clientes son lectores tolerantes: ignoran campos desconocidos y degradan los valores de enum no previstos en lugar de fallar.
- Los endpoints deprecados emiten `Deprecation`, `Sunset` y `Link`, y existe una página de migración real detrás de ese enlace.
- Hay una métrica de uso de endpoints deprecados etiquetada por consumidor, y una alerta cuando

queda uso a menos de treinta días del sunset.

- Antes del apagado definitivo hay al menos un brownout anunciado.
- Los esquemas de eventos tienen modo de compatibilidad definido y se comprueban en CI contra el registro de producción.

Preguntas frecuentes

¿Contract testing sustituye a los tests de integración? No. Sustituye a la mayoría de los tests de integración *entre servicios*, que son los lentos, frágiles y caros de mantener. Sigues necesitando tests de integración dentro de cada servicio, contra su base de datos y su cola, y algún test de extremo a extremo de los flujos críticos de negocio.

¿Cuántos contratos por par de servicios? Uno por interacción que el consumidor necesite, no uno por endpoint del proveedor. Si checkout usa tres endpoints de orders y de uno de ellos sólo dos campos, eso son tres interacciones con exactamente esos campos.

¿Merece la pena con dos servicios? Con dos servicios y un solo equipo, probablemente no: la coordinación directa es más barata. El punto de inflexión suele estar cuando quien cambia el proveedor ya no conoce de memoria a todos sus consumidores, y eso llega mucho antes de lo que parece.

¿Cuánto tiempo debe durar una deprecación? Depende del ciclo de despliegue del consumidor más lento, no del tuyo. Para consumidores internos, entre uno y tres meses suele bastar. Para clientes externos o aplicaciones móviles —donde hay versiones instaladas que nunca se actualizan— piensa en seis o doce meses, y asume que necesitarás un corte por versión mínima de cliente además del sunset.

¿Hace falta un broker o basta con ficheros en el repositorio? Se puede empezar con ficheros, pero pierdes lo único que hace seguro el despliegue continuo: saber qué versiones están vivas en cada entorno. Sin broker no hay `can-i-deploy`, y sin `can-i-deploy` el contract testing sólo te dice que dos commits concretos encajaban en algún momento.

¿Cómo se versiona un webhook saliente? Igual que una API, con la incomodidad de que el consumidor no puede elegir cuándo actualizarse: el que llama eres tú. Lo que funciona es que cada endpoint registrado guarde la versión de payload que espera, fijada el día que se dio de alta, y que el emisor serialice según esa versión. Así una suscripción antigua sigue recibiendo el formato de siempre mientras las nuevas nacen con el formato actual, y la retirada consiste en migrar suscripciones una a una en vez de romperlas todas a la vez. Incluye siempre el número de versión dentro del propio cuerpo del evento, no sólo en una cabecera: los cuerpos acaban en colas, en logs y en reprocesos donde las cabeceras ya no existen.

¿Y si un consumidor no contesta nunca a los avisos de deprecación? Es lo normal, y por eso el correo y el mensaje de Slack no son el mecanismo: son la cortesía. El mecanismo es la métrica por consumidor, que te da un nombre concreto con el que hablar, y el *brownout* programado —devolver 410 durante ventanas cortas y anunciadas antes del apagado definitivo—. Un brownout de quince minutos consigue más respuestas que tres meses de avisos, porque convierte un problema futuro y abstracto en una alerta en el canal de guardia de otro equipo. Anúncialo con fecha y hora, hazlo en horario laboral y no lo repitas más de dos o tres veces.

¿Las cabeceras y los códigos de estado forman parte del contrato? Sí, y son un punto ciego clásico. Pasar de devolver 200 con una lista vacía a devolver 404 rompe a cualquiera que ramifique por código de estado, y no aparece en ningún diff de esquema si no documentaste ese 404. Lo mismo con `Location` en un 201, con `ETag` si alguien hace peticiones condicionales, o con `Retry-After` en un 429. Si está documentado, oasdiff lo vigila; si no, es folklore.

¿Los límites de tasa y los tiempos de espera también cuentan? Formalmente no son parte del esquema,

pero son la parte del contrato que más rápido se nota cuando cambia. Bajar el límite de 1.000 a 100 peticiones por minuto rompe integraciones exactamente igual que quitar un campo, sólo que a las tres de la mañana y de forma intermitente. Trátalo con el mismo procedimiento: documentado, versionado, avisado con antelación y con una métrica que te diga a quién vas a afectar antes de aplicarlo.

¿Cuántas versiones conviene mantener vivas a la vez? Dos. Una en producción y una en retirada. En cuanto hay tres, la matriz de pruebas se multiplica, cada corrección de errores hay que hacerla por triplicado y, lo que es peor, los consumidores aprenden que nunca apagas nada y dejan de migrar. Si te ves con tres versiones en pie, el problema casi nunca es que publiques demasiado rápido: es que no has apagado ninguna.

Glosario

- **Cambio incompatible (breaking change)**: modificación que hace fallar a un cliente existente que no cambia su código.
- **CDC (consumer-driven contract)**: contrato escrito por el consumidor, con lo que necesita, y verificado por el proveedor.
- **Broker**: servicio que almacena contratos, resultados de verificación y qué versión está desplegada en cada entorno.
- **Pending pact**: contrato aún no verificado con éxito por el proveedor; se reporta pero no bloquea.
- **Provider state**: precondición de datos que el proveedor prepara antes de verificar una interacción.
- **Tolerant reader**: cliente que lee sólo lo que usa e ignora lo demás, incluidos los valores que no conoce.
- **Brownout**: corte breve, anunciado y programado de un endpoint deprecado, para localizar consumidores antes del apagado.
- **Sunset**: fecha a partir de la cual un recurso deja de estar disponible, anunciada con la cabecera del mismo nombre.

Si te llevas una sola idea: el objetivo no es impedir los cambios incompatibles, sino hacer que sean **visibles, deliberados y con fecha de salida**. Un equipo que puede romper su API a propósito, sabiendo exactamente a quién afecta y con cuánto margen, se mueve mucho más rápido que uno que no se atreve a tocar nada.

ENGLISH VERSION

API Versioning and Contract Testing: Safe Changes, OpenAPI in CI, Pact, and Retiring Endpoints with Sunset

The problem: coordinating deploys does not scale

Two services, one change. The *orders-api* team renames a field, announces it on Slack, and the *checkout-web* team ships the same Tuesday at six. It works. With five services you already need a spreadsheet. With twenty, coordination becomes the bottleneck: every small change waits for a shared window, and when something breaks there is no cheap way to find out who broke whom.

Contract testing flips the setup. Instead of discovering the incompatibility in an integration environment —or in production, at three in the morning— you turn it into a CI failure on the commit that introduced it. API versioning solves the other half: what you do when the breaking change is unavoidable. Not how to prevent it, but how to give your consumers a way out with a date attached.

This guide covers the three pieces that are rarely put in place together: knowing which changes actually break things, catching them automatically before merge, and retiring the old surface with a procedure rather than a hopeful email.

What a breaking change really is

The usual intuition —"adding is safe, removing is not"— is right less than half the time. The real rule depends on the direction of the data: **what the server receives** and **what the server returns** follow opposite rules.

In the request, the server is the consumer

Loosening is safe; tightening breaks.

- Adding an **optional** field: safe.
- Adding a **required** field: breaks. Every existing client starts getting 400s at once.
- Making a previously required field optional: safe.
- Widening the set of accepted enum values: safe. Narrowing it: breaks.
- Relaxing a validation (raising a `maxLength`, widening a range): safe. Tightening it: breaks, and it breaks silently for exactly the subset of clients that were already sending values outside the new limit.

In the response, the rules invert

- Removing or renaming a field: breaks.
- Adding a field: *almost* always safe. It breaks if a client validates against a strict schema or deserializes with `additionalProperties: false`.
- Widening a returned enum: **breaks**. This is the one that slips through most often. If you return `status` with three values and add a fourth, any client with an exhaustive `switch`, a `match` without a default arm, or a typed enum falls over the first time it sees the new value.
- Making a field `nullable` when it never was: breaks.
- Changing the type, even between "equivalent" representations such as a numeric id that becomes a

string: breaks.

And then there is the layer no schema captures: changing the *meaning* of a field without changing its shape, returning different error codes for the same failure, dropping the pagination maximum from 1000 to 100, or turning a 200 ms endpoint into a 4 s one that trips timeouts upstream. These are real incompatibilities that no diffing tool detects. We come back to them in the common mistakes section.

The tolerant reader

Much of the above stops hurting if clients follow the *tolerant reader* pattern: read only the fields you use, ignore unknown ones, and do not blow up on an unexpected enum value.

```
from enum import Enum
from pydantic import BaseModel, ConfigDict, field_validator


class OrderStatus(str, Enum):
    PENDING = "pending"
    SHIPPED = "shipped"
    DELIVERED = "delivered"
    UNKNOWN = "unknown"  # escape hatch for statuses we do not know about yet


class Order(BaseModel):
    # Ignoring unknown fields is already Pydantic's default, but making it explicit
    # stops someone from flipping it to "forbid" without measuring the blast radius.
    model_config = ConfigDict(extra="ignore")

    id: str
    status: OrderStatus
    total_cents: int

    @field_validator("status", mode="before")
    @classmethod
    def tolerate_unknown_statuses(cls, value: str) -> str:
        known = {member.value for member in OrderStatus}
        return value if value in known else OrderStatus.UNKNOWN.value
```

That `UNKNOWN` is the whole difference between "the provider added a status" and "checkout returned 500s for forty minutes". A caveat: tolerating is not guessing. If business logic needs to distinguish the new status, staying up only buys you time. But time is exactly what you need: time to ship real support without a rush and without an incident.

It also pays to emit a metric whenever the validator falls into the `UNKNOWN` branch. A counter labelled by field and unknown value turns a silent failure into a soft alert: "the provider is returning something we do not model".

OpenAPI as an executable contract

An OpenAPI file written by hand and updated "when there is time" is not a contract: it is stale documentation in YAML. For it to be worth anything it has to meet two conditions: it must be generated from the code (or generate the code), and it must be compared against the previous version on every pull request.

With FastAPI the first condition is free. What is missing is exporting the schema as a versioned artifact in the repository:

```
# scripts/export_openapi.py
import json
import sys

from app.main import app


def main() -> int:
    spec = app.openapi()
    # sort_keys keeps the git diff readable: without it, any internal reordering
    # by the framework produces a 400-line diff that nobody reviews.
    with open("openapi.json", "w", encoding="utf-8") as fh:
        json.dump(spec, fh, indent=2, sort_keys=True, ensure_ascii=False)
        fh.write("\n")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

With `openapi.json` committed, every API change shows up in the pull request diff, where a human can see it before a client does.

Catching breaking changes in CI

oasdiff compares two specifications and classifies every difference. It supports OpenAPI 3.0, 3.1 and 3.2, and covers hundreds of compatibility rules, including the counterintuitive ones from the previous section —yes, it catches a widened response enum—.

```
# .github/workflows/api-contract.yml
name: API contract
on: pull_request

jobs:
  breaking-changes:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Base branch specification
        run: git show origin/${{ github.base_ref }}:openapi.json > /tmp/base.json

      - name: Regenerate the specification from the code
        run: |
          pip install -e .
          python scripts/export_openapi.py

      # Without this step everything else is theatre: if the committed file does not
      # reflect the code, the diff comes out empty and the check passes green and lying.
      - name: Committed specification is up to date
        run: git diff --exit-code openapi.json

      - name: Look for breaking changes
        uses: oasdiff/oasdiff-action/breaking@main
        with:
          base: /tmp/base.json
          revision: openapi.json
          fail-on: ERR
```

Three details separate a decorative check from a useful one:

- `git diff --exit-code openapi.json` stops the contract from drifting. It is the step everyone omits and the one that makes the rest work.
- `fail-on: ERR` blocks only what is incompatible. Minor changes pass and are recorded as warnings, which is what you want: the tool must not become something people ignore because it is noisy.
- When the breaking change is intentional and already negotiated with consumers, approve it explicitly: a PR label that skips the gate, or `--exclude-elements` scoped to that one path. The goal is not to make breaking impossible, but to make it a visible decision someone signs off on.

Where OpenAPI stops: consumer-driven contract testing

OpenAPI describes what the API *can* return. It does not describe what each consumer *uses*. That gap cuts both ways.

If your response has forty fields and no client reads `legacy_shipping_code`, removing it is a breaking change according to the schema and a harmless cleanup in reality. The reverse happens too: a consumer relies on `items` arriving sorted by creation date, something the schema expresses nowhere and that you can break without touching a single line of OpenAPI.

Consumer-driven contract testing (CDC, and in practice Pact) fixes this by inverting who writes the contract. Each consumer declares, as an executable test, exactly what it needs. The union of those expectations is the contract the provider must honour.

Consumer side

```
# checkout/tests/contract/test_orders_api.py
import pytest
from pact.v3 import Pact, match

from checkout.clients.orders import OrdersClient


@pytest.fixture(scope="module")
def pact() -> Pact:
    return Pact("checkout-web", "orders-api").with_specification("V4")


def test_get_order_returns_the_fields_checkout_needs(pact: Pact) -> None:
    # Only the fields checkout actually reads. The other 36 in the real response
    # stay out of the contract: the provider must be free to change them.
    expected = {
        "id": match.str("ord_42"),
        "status": match.regex("shipped", regex=r"pending|shipped|delivered"),
        "total_cents": match.int(1999),
        "currency": match.str("EUR"),
    }

    (
        pact.upon_receiving("a request for order ord_42")
        .given("order ord_42 exists and has been shipped")
        .with_request("GET", "/v1/orders/ord_42")
        .will_respond_with(200)
        .with_body(expected, content_type="application/json")
    )

    with pact.serve() as srv:
        order = OrdersClient(base_url=str(srv.url)).get_order("ord_42")

    assert order.total_cents == 1999
    assert order.status == "shipped"
```

Two things make this test worth having:

- The `match.*` helpers assert **type and shape**, not literal values. The provider can return any valid identifier without breaking anything. A contract written with fixed values degenerates into a snapshot test and fails every week for reasons that are not incompatibilities.
- `given(...)` declares the state the provider will have to set up before verifying. It is not decorative: it is where most implementations go wrong, because it ends up as a provider endpoint that inserts rows by hand. It is worth designing well from day one, with small parameterised states instead of one giant shared fixture.

Provider side

The provider pulls every consumer's contract from the broker and replays it against a real instance of the service:

```
# orders/tests/contract/test_provider.py
import os

from pact.v3 import Verifier


def test_verify_consumer_contracts(running_app_url: str) -> None:
    verifier = (
        Verifier("orders-api")
        .add_transport(url=running_app_url)
        # Internal endpoint that sets up the state requested by given(...) and tears
        # it down afterwards. Registered in test environments only, never in production.
        .set_state(f"{running_app_url}/_pact/provider-state", teardown=True)
    )

    (
        verifier.broker_source(
            os.environ["PACT_BROKER_URL"],
            token=os.environ["PACT_BROKER_TOKEN"],
            selector=True,
        )
        .consumer_versions(deployed_or_released=True)
        .build()
    )

    verifier.verify()
```

Version selection is the part almost everyone skips and later misses. `deployed_or_released=True` means "verify against the consumer versions that are currently deployed or released in some environment". Without that filter you get two bad options: verify against the latest published contract, and let a consumer still experimenting on a branch block your release; or verify against the entire history, and let a version nobody has run in eight months block it instead.

The piece that makes it safe: can-i-deploy

A green verification says *those two specific versions* are compatible. It does not say whether the version you are about to deploy is compatible with what is running in production right now. Only the broker knows that:

```
# Before deploying the provider
pact-broker can-i-deploy \
  --pacticipant orders-api \
```

```
  --version "$GIT_SHA" \
  --to-environment production \
  --retry-while-unknown 12 \
  --retry-interval 10

# And once the deploy finishes, record the fact
pact-broker record-deployment \
  --pacticipant orders-api \
  --version "$GIT_SHA" \
  --environment production
```

`record-deployment` looks like paperwork and is the opposite. Without that record the broker does not know which version runs in which environment, and `can-i-deploy` cannot answer anything useful. A deploy that goes unrecorded turns the whole contract setup into expensive theatre.

`--retry-while-unknown` covers a very common race: the provider pipeline reaches the check before the consumer pipeline has finished publishing its verification result. Without retries that race shows up as flaky failures, which people learn to fix by re-running the job —the worst possible way to treat a safety gate—.

Rolling it out without stopping the factory

Turn everything on at once and the first consumer to add a new expectation will turn a provider's pipeline red for something the provider did not do. That is what two mechanisms exist for, and they should be on from day one:

- **Pending pacts.** A contract the provider has never verified successfully is reported in the build output but does not fail it. As soon as the provider verifies it once, it stops being pending and becomes binding. This is what lets the consumer write the expectation first and the provider implement it after.
- **WIP pacts.** The provider additionally picks up contracts from consumer feature branches, again without blocking. It lets the consumer find out, before even opening the pull request, whether what it needs already works.

The adoption order that works in practice: start with the pair of services that produces the most integration incidents, leave contracts in pending for a couple of sprints so the team sees the value without suffering the gate, and only then make `can-i-deploy` a required pipeline step.

What if the provider is a third party?

CDC requires the provider to run verification, so it does not work for APIs you do not control. The alternative there is **bi-directional contracts**: the provider publishes its OpenAPI, the consumer publishes its pact, and the broker checks that the second is contained in the first. It is a weaker guarantee —it compares schemas, not real behaviour— but it works with zero cooperation from the other side, and it beats nothing by a wide margin.

Versioning strategies

When the breaking change is unavoidable, you have to serve both versions for a while. How the client picks one is the versioning decision, and there are three real options:

- **In the path** (`/v1/orders`, `/v2/orders`). Visible in logs, traces, dashboards and CDN; trivial to route at the gateway; trivial to test with curl. The price is that it violates the idea that a URI identifies a resource: the same order has two addresses. In practice it is what nearly everyone uses, and for good reason.
- **In a custom header** (`X-API-Version: 2026-09-14`). Keeps URIs stable and lets you version by date,

which scales far better than integers once you have dozens of small changes. The cost is operational: the version disappears from access logs unless you add it yourself, and caches need `Vary` or they will serve the wrong representation.

- **In the content type** (`Accept: application/vnd.acme.order+json;version=2`). The most correct option per HTTP and the most awkward to use. Reserve it for APIs with sophisticated consumers and a genuine need to negotiate representations.

Any of the three works. What does not work is picking none of them and trusting that a breaking change will never come.

Version as little as possible

The expensive mistake is usually not picking the wrong mechanism but the wrong **granularity**. Bumping the entire API to `/v2` because one endpoint changed forces every consumer to migrate everything, including the parts that did not change. The predictable result: nobody migrates, you maintain two complete surfaces for years, and every new feature has to be built twice.

The boring, correct alternative is to evolve in place and version only the affected resource. Add the new field alongside the old one and populate both during the transition. If you truly need two incompatible shapes, expose `/v2/orders` and leave the rest of the API on `/v1`. A consumer that only uses `/v1/invoices` should not even notice a migration happened.

Retiring a version: Deprecation, Sunset and data

Publishing `/v2` is the easy part. The hard part is switching `/v1` off, and almost nobody manages it because they try with an email instead of a procedure.

HTTP has two standard headers for this, and they mean different things:

- `Deprecation` (RFC 9745): "this resource is deprecated". Its value is a structured field of type Date, that is, an at sign followed by Unix seconds. It can be in the past (already deprecated) or in the future (it will be). Deprecating does not change the behaviour of the resource: it keeps working exactly as before.
- `Sunset` (RFC 8594): "from this date onwards the resource is likely to stop responding". Its value is an HTTP date in IMF-fixdate format. RFC 9745 is explicit about it: the `Sunset` date must never be earlier than the `Deprecation` one.

On top of that, a `Link` with `rel="deprecation"` points at the migration documentation, and `rel="successor-version"` at what to use instead. Between the three headers, a client can detect and diagnose the situation without anyone reading a changelog.

```
from datetime import datetime, timezone
from email.utils import format_datetime

from fastapi import FastAPI, Request

app = FastAPI()

DEPRECATED_AT = datetime(2026, 9, 14, tzinfo=timezone.utc)
SUNSET_AT = datetime(2027, 3, 14, tzinfo=timezone.utc)
DEPRECATED_PREFIXES = ("/v1/orders",)

DOCS = "https://api.acme.com/docs/migrations/orders-v2"
SUCCESSOR = "https://api.acme.com/v2/orders"
```

```
@app.middleware("http")
async def deprecation_headers(request: Request, call_next):
    response = await call_next(request)
    if request.url.path.startswith(DEPRECATED_PREFIXES):
        # RFC 9745: structured field Date = "@" + unix seconds
        response.headers["Deprecation"] = f"@{int(DEPRECATED_AT.timestamp())}"
        # RFC 8594: HTTP-date in IMF-fixdate (usegmt yields "GMT", not "+0000")
        response.headers["Sunset"] = format_datetime(SUNSET_AT, usegmt=True)
        response.headers["Link"] = (
            f'<{DOCS}>; rel="deprecation"; type="text/html", '
            f'<{SUCCESSOR}>; rel="successor-version"'
        )
    return response
```

A deprecated response then looks like this:

```
HTTP/1.1 200 OK
Content-Type: application/json
Deprecation: @1789344000
Sunset: Sun, 14 Mar 2027 00:00:00 GMT
Link: <https://api.acme.com/docs/migrations/orders-v2>; rel="deprecation"; type="text/html",
      <https://api.acme.com/v2/orders>; rel="successor-version"
```

The part that decides whether you can actually switch it off: measuring

A sunset date picked by gut feeling always slips. What lets you switch things off for real is knowing, by name, who is still calling:

```
from prometheus_client import Counter

DEPRECATED_CALLS = Counter(
    "deprecated_api_calls_total",
    "Calls to deprecated endpoints, by consumer",
    ["consumer", "route", "api_version"],
)


@app.middleware("http")
async def track_deprecated_usage(request: Request, call_next):
    response = await call_next(request)
    if request.url.path.startswith(DEPRECATED_PREFIXES):
        DEPRECATED_CALLS.labels(
            # The consumer comes from the token, not the User-Agent: the User-Agent
            # is "python-requests/2.32" 80% of the time and identifies nobody.
            consumer=getattr(request.state, "client_id", "unknown"),
            # The route template, not the concrete URL: /v1/orders/{order_id}
            # and not /v1/orders/ord_42, or metric cardinality explodes.
            route=request.scope.get("route").path if request.scope.get("route") else "unmatched",
            api_version="v1",
        ).inc()
    return response
```

And then the query that answers the only question that matters:

```
# Who has called /v1/orders in the last 7 days
sum by (consumer) (
  increase(deprecated_api_calls_total{api_version="v1", route=~"/v1/orders.*"}[7d])
```

```
) > 0
```

With that list you can do the only thing that works: talk to each team by name, instead of announcing it in a general channel and hoping. And once the list drops to zero or only stragglers remain, add **brownouts** before the final shutdown: scheduled, announced windows in which the endpoint returns `410 Gone` for an hour. A brownout finds in a day the consumers no metric attributed correctly, and it finds them during office hours rather than at three in the morning on shutdown day.

Contracts are not only HTTP

If you publish events, the event schema is an equally binding contract, with one important difference: you cannot deploy producers and consumers at the same instant, and old messages stay in the topic after you change the schema. That is why schema registries offer compatibility modes, and picking the right one is a decision about *who upgrades first*:

- **BACKWARD**: a consumer on the new schema can read data written with the old one. You upgrade consumers first. It is the default and what you want almost always.
- **FORWARD**: a consumer on the old schema can read data written with the new one. You upgrade producers first. Useful when you do not control every consumer.
- **FULL**: both. It only allows very conservative changes (adding or removing optional fields with defaults), and it is what you need when deploy order is not guaranteed.

The equivalent of `can-i-deploy` here is running the registry's compatibility check in CI, against the schema registered in production, before merging. Same principle: make the failure show up in the pull request, not in the consumer.

Errors are part of the contract too: RFC 9457

Almost everything written about versioning covers the happy path: the schema of the 200 response. Then, in production, the change that breaks a client is a different one. Someone rewrites the error handler and what used to be an object with an `error` key becomes a list under an `errors` key. The 200 has not moved a comma, oasdiff says nothing because the errors were never documented, and the consumer that decided whether to retry by reading `error` starts retrying requests that will never succeed.

Errors are part of the contract exactly like successes are. They deserve the same treatment: a stable format, documented in OpenAPI and watched in CI. RFC 9457, *Problem Details for HTTP APIs* (July 2023, obsoleting RFC 7807), defines that format so every team does not have to invent its own. The body is served with `Content-Type: application/problem+json` and has five standard fields:

- `type`: a URI identifying the *class* of problem. It is the only field meant to be read by a machine, and therefore the one that really belongs to the contract.
- `title`: a short human-readable summary of that class of problem. You can rewrite or translate it without breaking anything.
- `status`: the HTTP code, duplicated in the body so it survives proxies and logs that only keep the payload.
- `detail`: an explanation of *this* specific occurrence. Also for humans.
- `instance`: a URI for the occurrence. Useful for correlating with traces if you put the request id in it.

On top of that you can add your own fields —the *extensions*— and that is where the format earns its keep:

`balance` and `required` on an insufficient-funds error, `retry_after_seconds` on a 429, `field` and `rule` on a validation failure. Extensions follow the same additive rule as the rest of the response: adding one is safe, removing one breaks.

A minimal FastAPI implementation that keeps the format in one place:

```
# app/errors.py
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

PROBLEM_BASE = "https://api.example.com/problems/"


class ProblemError(Exception):
    """A domain error that already knows how to serialize itself."""

    def __init__(self, slug: str, title: str, status: int, detail: str = "", **extensions):
        self.slug = slug
        self.title = title
        self.status = status
        self.detail = detail
        self.extensions = extensions


def install(app: FastAPI) -> None:
    @app.exception_handler(ProblemError)
    async def handle(request: Request, exc: ProblemError):
        body = {
            "type": PROBLEM_BASE + exc.slug,
            "title": exc.title,
            "status": exc.status,
            "detail": exc.detail,
            "instance": request.url.path,
        }
        body.update(exc.extensions)
        return JSONResponse(
            status_code=exc.status,
            content=body,
            media_type="application/problem+json",
        )
```

Raising it from the domain stays reasonably clean:

```
raise ProblemError(
    slug="insufficient-funds",
    title="Insufficient funds",
    status=422,
    detail="The account does not cover the order total.",
    balance="12.40",
    required="49.90",
)
```

And the client gets something it can act on without guessing:

```
HTTP/1.1 422 Unprocessable Content
Content-Type: application/problem+json

{
  "type": "https://api.example.com/problems/insufficient-funds",
  "title": "Insufficient funds",
```

```
  "status": 422,
  "detail": "The account does not cover the order total.",
  "instance": "/v1/orders",
  "balance": "12.40",
  "required": "49.90"
}
```

Three rules turn this into a contract rather than a nice-looking format:

- **The `type` URI is immutable.** Changing `insufficient-funds` to `payment-declined` is as real a breaking change as renaming a response field, because clients branch on that value. If you need a new class, add it; the old one gets deprecated and retired through the same procedure as an endpoint.
- **Never put anything in `detail` that a client needs to parse.** The moment somebody extracts a number from it with a regular expression, `detail` stops being human text and becomes API. If the data is needed, it is an extension with a field of its own.
- **Document errors in OpenAPI.** If they are not in the schema, oasdiff cannot see that you changed one, and all the work from the previous section stops halfway.

```
from pydantic import BaseModel


class Problem(BaseModel):
    type: str
    title: str
    status: int
    detail: str = ""
    instance: str = ""


PROBLEM_RESPONSES = {
    409: {"model": Problem, "content": {"application/problem+json": {}}},
    422: {"model": Problem, "content": {"application/problem+json": {}}},
}


@app.post("/v1/orders", responses=PROBLEM_RESPONSES)
def create_order(...):
    ...
```

One detail that saves grief: keep a table in the repository listing every `type` URI the service publishes, plus a test that walks the code looking for `ProblemError` and fails when it finds a `slug` that is not in that table. It is five minutes of work, and it is what stops you from having forty problem classes in six months, eight of them duplicates and none documented.

Pagination, filtering and sorting: the invisible contract

Pagination is where the most contracts get broken without anybody calling it breaking a contract. The response schema does not change: there is still a list of orders and a total. What changes is the *behaviour*, and no OpenAPI file captures that.

Start with the things almost nobody counts as breaking changes, which are:

- **Lowering the maximum `limit`.** If you accepted `limit=500` and now return a 400 above 100, you broke everyone paging 500 at a time. That you did it to protect the database is an excellent reason for the change and no reason at all to skip the deprecation procedure.

- **Changing the default page size.** A client that calls without `limit` and assumes 50 starts receiving 20. And if it stopped paging once it got fewer items than expected, it now processes 40% of the data and does not fail: it loses the rest silently, which is worse.
- **Changing the default ordering.** If you never documented the order, technically you are free. In practice there is a client showing "latest orders" that trusts the first element is the most recent, and that client breaks without a single error in the logs.
- **Tightening a filter that used to be lenient.** Rejecting an unknown query parameter with a 400 when you used to ignore it is tightening request validation: the same case as adding a required field.

Practical rule: pagination and ordering defaults are part of the contract from day one, documented or not. Write them into the schema, including `default` and `maximum`, and treat them like any other field.

From offset to cursor without breaking anyone

The other problem with `offset`/`limit` is not about contracts but about correctness: with concurrent writes, page 2 overlaps page 1 or skips rows, because the offset is computed over a set that moves between requests. And the cost grows linearly with the offset, since the database has to read and discard every preceding row. The alternative is *keyset* pagination: instead of saying "skip 10,000", you say "give me what comes after this exact point".

```
import base64
import json
from datetime import datetime


def encode_cursor(created_at: datetime, order_id: str) -> str:
    raw = json.dumps({"t": created_at.isoformat(), "id": order_id}, separators=(",", ":"))
    return base64.urlsafe_b64encode(raw.encode()).decode().rstrip("=")


def decode_cursor(cursor: str) -> tuple[datetime, str]:
    padded = cursor + "=" * (-len(cursor) % 4)
    data = json.loads(base64.urlsafe_b64decode(padded))
    return datetime.fromisoformat(data["t"]), data["id"]
```

The query uses the full tuple as a tie-breaker, which is the detail that makes it work with repeated timestamps:

```
-- The index has to match the ORDER BY exactly
CREATE INDEX CONCURRENTLY idx_orders_customer_created
  ON orders (customer_id, created_at DESC, id DESC);

SELECT id, created_at, total
FROM orders
WHERE customer_id = $1
  AND (created_at, id) < ($2, $3)
ORDER BY created_at DESC, id DESC
LIMIT $4;
```

The migration works like any other compatible evolution: the endpoint accepts both shapes for a while and the old one is marked deprecated.

```
@app.get("/v1/orders")
def list_orders(
    customer_id: str,
```

```
    limit: int = Query(default=25, ge=1, le=100),
    cursor: str | None = None,
    offset: int | None = Query(default=None, deprecated=True),
):
    if cursor is not None and offset is not None:
        raise ProblemError(
            slug="conflicting-pagination",
            title="Conflicting pagination parameters",
            status=400,
            detail="Use cursor or offset, not both.",
        )
    if offset is not None:
        DEPRECATED_PARAM.labels(param="offset", consumer=consumer_of(request)).inc()
        rows = query_by_offset(customer_id, limit, offset)
    else:
        rows = query_by_cursor(customer_id, limit, cursor)

    next_cursor = encode_cursor(rows[-1].created_at, rows[-1].id) if len(rows) == limit else None
    return {"items": [serialize(r) for r in rows], "next_cursor": next_cursor}
```

Note the `DEPRECATED_PARAM` counter: it is the same trick from the sunset section applied to a parameter instead of a whole route. Without it, the date on which you can delete the `offset` branch is a guess.

And a warning about the cursor: document explicitly that it is **opaque**, and honour that yourself. The moment a client discovers it is base64 of a JSON object, it will decode it, mint its own cursors, and tie you to that internal format forever. If that worries you, sign it with HMAC or encrypt it; a string nobody can construct from outside is a string you can change whenever you want.

Expand / contract: renaming a field without stopping the factory

So far we have talked about detecting breaking changes and announcing them. What is missing is the mechanical part: how you actually execute a change that touches the database, the code and the public contract at once, with no maintenance window and without a rollback leaving you with data the previous code cannot read.

The pattern is called *expand / contract* (or *parallel change*) and the idea is simple: between the old state and the new one there is a period where both coexist, and that period lasts as long as it needs to. What makes it safe is not the pattern itself but a rule worth carving in stone: **every deploy must be revertible without reverting the schema**. If rolling back a release requires undoing a migration, you do not have a rollback, you have a hope.

Take the concrete case of replacing `shipping_country` (free text: "Spain", "ES", "españa") with `shipping_country_code` (ISO 3166-1 alpha-2). Six phases, each in its own deploy.

Phase 1 — Expand the schema

A strictly additive migration. New column, nullable, no default, no constraints. The old code keeps working because it does not know it exists.

```
ALTER TABLE orders ADD COLUMN shipping_country_code char(2);

-- CONCURRENTLY does not block writes; in exchange it cannot run
-- inside a transaction, so it goes in its own migration.
CREATE INDEX CONCURRENTLY idx_orders_shipping_country_code
  ON orders (shipping_country_code);
```

No `NOT NULL` yet, and no `DEFAULT` with a volatile function: on large tables, depending on the engine version, that can rewrite the whole table and take the service down exactly when you thought you were making a harmless change.

Phase 2 — Dual write

The service writes both fields on every operation. It still reads the old one. This deploy is fully reversible: revert it and the new column simply stops being updated.

```
COUNTRY_ALIASES = {"spain": "ES", "españa": "ES", "es": "ES", "france": "FR", "fr": "FR"}


def normalize_country(raw: str | None) -> str | None:
    if not raw:
        return None
    return COUNTRY_ALIASES.get(raw.strip().lower())


def save_order(conn, order) -> None:
    conn.execute(
        "UPDATE orders SET shipping_country = %s, shipping_country_code = %s WHERE id = %s",
        (order.shipping_country, normalize_country(order.shipping_country), order.id),
    )
```

Phase 3 — Backfill

Historical rows still have the new field at `NULL`. Fill them in batches, off the request path, with pauses so replication can keep up.

```
import time

BATCH = 5_000


def backfill(conn) -> None:
    while True:
        rows = conn.execute(
            "SELECT id, shipping_country FROM orders "
            "WHERE shipping_country_code IS NULL AND shipping_country IS NOT NULL "
            "ORDER BY id LIMIT %s FOR UPDATE SKIP LOCKED",
            (BATCH,),
        ).fetchall()
        if not rows:
            return
        conn.executemany(
            "UPDATE orders SET shipping_country_code = %s WHERE id = %s",
            [(normalize_country(country), row_id) for row_id, country in rows],
        )
        conn.commit()
        time.sleep(0.2)
```

`FOR UPDATE SKIP LOCKED` lets several backfill workers make progress at once without stepping on each other and without blocking normal writes. It is also worth logging how many rows could not be normalized: that number tells you whether the mapping is complete or whether there are twenty spellings of "United Kingdom" nobody had seen.

Phase 4 — Read the new one, with a safety net

The service switches to reading the new field, falling back to the old one when the new one is empty. That fallback branch carries its own metric: once the counter has been at zero for days, the backfill is genuinely complete.

```
code = row.shipping_country_code
if code is None:
    FALLBACK_READS.inc()
    code = normalize_country(row.shipping_country)
```

Phase 5 — Publish both and deprecate the old one

Now, and only now, the change reaches the public contract. The response carries both fields, the old one is marked deprecated in OpenAPI, and calls that use it are counted the way we saw in the sunset section. Adding a field is compatible, so this deploy breaks nobody.

```
class OrderOut(BaseModel):
    id: str
    shipping_country_code: str | None = None
    shipping_country: str | None = Field(
        default=None,
        deprecated=True,
        description="Deprecated. Use shipping_country_code (ISO 3166-1 alpha-2). Sunset: 2026-12-01.",
    )
```

Phase 6 — Contract

When no consumer reads the old field —and that is something the Pact contracts and the usage counter tell you, not an intuition— it gets retired: first from the API, then from the dual write, and last from the database. The three steps are separate deploys, in that order. The `DROP COLUMN` is the last one and the only irreversible one, so it happens weeks after the rest, when everybody has forgotten about it.

```
-- Weeks after the API stops exposing it and the code stops writing it
ALTER TABLE orders ALTER COLUMN shipping_country_code SET NOT NULL;
ALTER TABLE orders DROP COLUMN shipping_country;
```

The nice part about fitting this together with everything above is that the phases do not depend on anyone remembering. Phase 5 passes the oasdiff gate because it only adds. Phase 6 fails oasdiff —removing a response field breaks— and that is where `can-i-deploy` comes in: if any consumer still expects `shipping_country` in its pact, the removal never reaches production. The pattern tells you what to do; the pipeline stops you from getting ahead of yourself.

Governance with Spectral: making the contract coherent too

oasdiff answers "did I break something?". It does not answer "does this look like the rest of our API?". And that second question matters more than it seems, because inconsistency is paid for later in the form of breaking changes: the endpoint that returned the identifier as an integer because nobody reviewed it will

migrate to a string the day ids stop fitting in an integer, and that migration will break someone.

Spectral is a JSON/YAML linter maintained by Stoplight that validates OpenAPI and AsyncAPI specifications against rules you define. It ships a base set (`spectral:oas`) with the structural checks, and you layer your house conventions on top.

```
# .spectral.yaml
extends: ["spectral:oas"]

rules:
  operation-operationId: error
  operation-description: warn
  operation-tags: error

  paths-kebab-case:
    description: Paths use kebab-case and plural nouns.
    severity: error
    given: $.paths[*]~
    then:
      function: pattern
      functionOptions:
        match: "^(/[a-z0-9-]+|/\{[a-zA-Z0-9_]+\})+$"

  ids-are-strings:
    description: Identifiers serialize as strings, never as integers.
    severity: error
    given: $..properties[?(@property == 'id')].type
    then:
      function: pattern
      functionOptions:
        notMatch: "^(integer|number)$"

  errors-use-problem-json:
    description: Every 4xx/5xx response documents application/problem+json.
    severity: error
    given: $.paths[*][*].responses[?(@property.match(/^[45]/))].content
    then:
      field: application/problem+json
      function: truthy

  collections-are-paginated:
    description: Collections accept limit and cursor.
    severity: warn
    given: $.paths[*].get.parameters
    then:
      function: schema
      functionOptions:
        schema:
          type: array
          contains:
            type: object
            properties:
              name:
                const: limit
```

In CI it runs before oasdiff, for a practical reason: a specification that does not pass the lint is not worth diffing.

```
- name: Lint OpenAPI
  run: npx --yes @stoplight/spectral-cli lint openapi.json --ruleset .spectral.yaml
    --fail-severity=error
```

The classic adoption mistake is switching on forty rules at `error` on day one against an API with six years of

history. The result is a red pipeline everyone learns to ignore. What works is a ratchet: every rule lands as `warn`, the pipeline only fails on `error`, and each week you fix one category and promote that rule to `error`. Existing paths stay as they are —renaming them would be a breaking change for cosmetic reasons, which is exactly what we are trying to avoid—; the rule applies to what is new, and the old stuff is listed as an explicit exception in the ruleset, with a comment saying why.

It is also worth adding a security ruleset based on the OWASP API Security Top 10. It does not replace a security review, but it catches the usual slips for free: an endpoint with no `security`, a documented 200 where there should be a 401, an array schema with no `maxItems`.

gRPC and Protobuf: wire compatibility versus source compatibility

If part of your internal traffic runs over gRPC, you have a second contract with rules of its own, and the trap is that Protobuf looks more forgiving than it is. The key point is that Protobuf does not serialize field names: it serializes the field *number* and its wire type. Two distinct notions of compatibility follow from that, and they are worth keeping apart.

- **Wire compatibility.** Can an old binary deserialize a new message? Renaming a field while keeping its number is wire compatible: the bytes are identical. Changing the number, or moving an `int32` to a `string`, is not, and the effect is not a clean exception but a field that arrives empty or full of garbage.
- **Generated-code compatibility.** Does the client still compile after you regenerate? That same rename which was harmless on the wire breaks every line of code that called the old accessor.

And there is a third level that catches a lot of people: Protobuf's JSON encoding —the one used by Connect, grpc-gateway and gRPC JSON transcoding— *does* serialize names. The moment you expose the service over JSON, renaming a field goes from free to breaking for half your consumers.

`buf breaking` makes that distinction explicit through categories, from strictest to most lenient: `FILE` (the default, around 46 rules, detects generated-source breakage file by file), `PACKAGE` (the same at package level, tolerating definitions moving between files in the same package), `WIRE_JSON` (breakage of the binary wire format or the JSON one) and `WIRE` (binary wire only, 15 rules).

The choice is not a matter of taste: `WIRE_JSON` is the minimum defensible setting if any of your traffic goes over JSON, and `FILE` is the right one if you publish the `.proto` files to teams that compile against them.

```
# buf.yaml
version: v2
modules:
  - path: proto
lint:
  use:
    - STANDARD
breaking:
  use:
    - WIRE_JSON
```

The CI comparison runs against the main branch, not against the previous commit, so a long-lived branch cannot accumulate invisible breakage:

```
# .github/workflows/proto-contract.yml
name: Proto contract
on: pull_request
```

```
jobs:
  breaking:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - uses: bufbuild/buf-action@v1
        with:
          breaking_against: ".git#branch=origin/main,subdir=proto"
```

And the piece of hygiene that prevents half the incidents: when you remove a field, reserve its number and its name. It is one line, and it is what stops someone two years from now from reusing number 4 for a different type and finding old messages in a queue that deserialize into something they are not.

```
syntax = "proto3";

package orders.v1;

message Order {
  // Removed 2026-05. Do not reuse.
  reserved 4, 7 to 9;
  reserved "legacy_shipping_code";

  string id = 1;
  string customer_id = 2;
  Money total = 3;
  OrderStatus status = 5;
  string shipping_country_code = 6;
}

enum OrderStatus {
  ORDER_STATUS_UNSPECIFIED = 0;
  ORDER_STATUS_PENDING = 1;
  ORDER_STATUS_PAID = 2;
  ORDER_STATUS_SHIPPED = 3;
}
```

That `ORDER_STATUS_UNSPECIFIED = 0` is not a formality. proto3 enums are open: an unknown value does not fail to deserialize, it arrives as an integer your code does not recognise. It is exactly the widened-enum problem we saw in JSON, with the same fix —a default branch that assumes nothing— and one advantage: here the library hands it to you instead of blowing up. What it does not save you from is an exhaustive `match` with no default arm in whichever language you use.

GraphQL: the contract you cannot version

GraphQL has no `/v2`, and that is not an oversight: the premise is that the client asks for exactly the fields it needs, so adding fields never breaks anyone and there is no reason to duplicate the API. The price of that premise is that **all evolution has to happen in the same schema**, without the escape hatch of publishing a new version alongside the old one.

The compatibility rules also invert relative to REST, and they are worth having straight:

- **Adding a field or a type: safe.** Nobody receives it unless they ask for it.
- **Removing or renaming a field: breaks** any operation that mentions it, even if the client never used the value.

- **Moving an output from `String!` to `String`: breaks.** The client generated non-nullable types. The other direction (`String` to `String!`) is safe for readers.
- **On input arguments the rules invert**, just like the body of a REST request: adding an optional argument is safe, adding a required one breaks, and moving an argument from required to optional is safe.
- **Adding an enum value is dangerous** without being strictly breaking: the schema allows it, but the client's `switch` does not handle it. The same tolerant reader as always.
- **Turning a field's type into a union or an interface: breaks** every selection that does not use fragments.

Since there are no versions, deprecation is the only retirement mechanism, and happily it is built into the language. The `@deprecated` directive hides the field from introspection by default, shows up with a warning in any IDE's autocomplete, and gives you a natural place to put the date:

```
type Order {
  id: ID!
  total: Money!
  shippingCountryCode: String

  shippingCountry: String
    @deprecated(reason: "Use shippingCountryCode (ISO 3166-1 alpha-2). Planned removal: 2026-12-01.")
}
```

CI verification is a schema diff. GraphQL Inspector compares the branch schema against `main` and classifies every change as safe, dangerous or breaking:

```
# .github/workflows/graphql-contract.yml
name: GraphQL contract
on: pull_request

jobs:
  schema-diff:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - run: |
          npx --yes @graphql-inspector/cli diff \
            "git:origin/main:schema.graphql" "schema.graphql" \
            --rule suppressRemovalOfDeprecatedField
```

That `suppressRemovalOfDeprecatedField` rule deserves a paragraph, because it is the `can-i-deploy` of this world. By default, removing a deprecated field still counts as breaking —and rightly so: deprecated does not mean unused. The rule downgrades it to "dangerous" so it stops blocking the pipeline. Enabling it without measuring real usage is exactly the same mistake as picking a sunset date by gut feeling. Turn it on only once you have solved the other half: an operation registry or persisted queries telling you which client asked for that field in the last thirty days.

That registry, incidentally, is GraphQL's great advantage here. In REST you know somebody called `/v1/orders`; working out whether they read `shipping_country` takes guesswork or a conversation. In GraphQL the request *enumerates* the fields, so field-level usage is data you already have. It is the only technology we have covered where "can I delete this?" has an exact answer instead of an estimate.

Generated SDKs: the client has a contract too

If you publish a client library, you have signed a second contract with its own surface and its own versioning, and the two do not line up. The API can stay on `v1` while the SDK is on `4.2.0`, and that is fine: they version different things.

The healthy way to manage it is to generate the SDK from `openapi.json` in CI, never by hand, with three rules:

- **Pin the generator version.** A generator upgrade can change method names or model shapes without your specification moving an inch. If the generator floats, one Monday morning you publish a major you never decided to publish.
- **Never edit generated code.** Whatever needs to sit on top goes in a thin hand-written layer wrapping the generated client. The moment someone patches the generated file, the next regeneration overwrites it and the fix disappears.
- **Derive the version bump from the diff, not from a feeling.** You already have the signal: if oasdiff flags a breaking change, the SDK goes up a major. If there are only additions, minor. If only descriptions or examples changed, patch.

```
- name: Generate Python SDK
  run: |
    npx --yes @openapitools/openapi-generator-cli@2.24.0 generate \
      -i openapi.json \
      -g python \
      -o sdk/python \
      --additional-properties=packageName=acme_orders,packageVersion=4.2.0
    git diff --stat sdk/python
```

That trailing `git diff --stat` is more useful than it looks: it turns the effect of a schema change into something visible in the pull request review. A one-line tweak to the `.proto` or the Pydantic model that touches eighty SDK files is an alarm no linter was going to raise for you.

And document in the README how SDK versions map to API versions —which range of client majors talks to `/v1`, which to `/v2`— because it is the first question anyone integrating asks and the worst one to answer by digging through the commit history.

Eight recurring mistakes

1. **Believing that adding to a response is always safe.** Widening a returned enum breaks any client with strict deserialization, and it is usually discovered in production because staging has no data carrying the new value. Introduce new enum values behind a flag and warn first.
2. **The `openapi.json` that never gets regenerated.** The breaking-change check compares two files; if the file does not reflect the code, it compares two works of fiction. The `git diff --exit-code` step is mandatory, not optional.
3. **Using Pact as a snapshot test.** Copying the whole response into the contract, literal values included, produces a contract that fails every time the provider adds a harmless field. The team learns to ignore the failure and the contract stops carrying signal. A contract should hold the minimum you consume, with type matchers.
4. **Provider states backed by global fixtures.** A single shared "test data seed" behind every `given(...)` couples the contracts to each other, so a change to the seed breaks unrelated verifications. Each state

should create what it needs and tear it down.

5. `can-i-deploy` **without** `record-deployment`**.** The gate always says yes, because the broker thinks nothing is deployed. That is worse than no gate: it manufactures false confidence.
6. **Deprecating without measuring.** With no per-consumer metric the sunset date is set by gut feeling, it arrives, nobody dares to flip the switch, and the old version survives three more years. The metric is what turns the decision from political into technical.
7. **Versioning the whole API for one endpoint.** It forces migration of things that did not change, so nobody migrates and you end up maintaining two complete surfaces.
8. **Behavioural changes with no schema change.** Lowering the pagination limit, switching an error from 409 to 422, dropping the implicit sort by date, or tripling p99 latency. No diffing tool sees any of these. There are only two defences: consumer contracts that encode those expectations where they matter, and the discipline of treating error codes and limits as part of the public API.

Production checklist

- The OpenAPI specification is generated from the code and CI fails when the committed version does not match.
- Every pull request runs a compatibility diff and blocks unapproved breaking changes.
- There is an explicit, auditable way to approve a breaking change when one is genuinely needed.
- Critical consumers publish contracts; the provider verifies them in CI against deployed versions.
- Pending and WIP pacts are enabled so a new contract cannot break the provider's build.
- `can-i-deploy` is a required step before deploying, and `record-deployment` runs after every deploy in every environment.
- Clients are tolerant readers: they ignore unknown fields and degrade unexpected enum values instead of failing.
- Deprecated endpoints emit `Deprecation`, `Sunset` and `Link`, and there is a real migration page behind that link.
- A usage metric for deprecated endpoints exists, labelled by consumer, with an alert when usage remains within thirty days of sunset.
- At least one announced brownout happens before the final shutdown.
- Event schemas have a defined compatibility mode, checked in CI against the production registry.

Frequently asked questions

Does contract testing replace integration tests? No. It replaces most integration tests *between services*, which are the slow, flaky, expensive ones. You still need integration tests inside each service, against its database and its queue, plus a handful of end-to-end tests for the critical business flows.

How many contracts per pair of services? One per interaction the consumer needs, not one per provider endpoint. If checkout uses three orders endpoints and reads only two fields from one of them, that is three interactions with exactly those fields.

Is it worth it with only two services? With two services and a single team, probably not: direct coordination is cheaper. The tipping point tends to be when whoever changes the provider no longer knows all of its consumers by heart, and that arrives much sooner than people expect.

How long should a deprecation last? It depends on the slowest consumer's deploy cycle, not yours. For internal consumers, one to three months is usually enough. For external clients or mobile apps —where

installed versions never update— think six to twelve months, and assume you will need a minimum-client-version cutoff on top of the sunset.

Do I need a broker, or are files in the repo enough? You can start with files, but you lose the one thing that makes continuous deployment safe: knowing which versions are alive in each environment. No broker means no `can-i-deploy`, and without `can-i-deploy` contract testing only tells you that two particular commits once fit together.

How do you version an outgoing webhook? Like an API, with the inconvenience that the consumer cannot choose when to upgrade: you are the caller. What works is for every registered endpoint to store the payload version it expects, fixed on the day it was created, and for the sender to serialize according to that version. An old subscription keeps receiving the format it always got while new ones are born on the current format, and retirement becomes migrating subscriptions one at a time instead of breaking them all at once. Always include the version number inside the event body, not only in a header: bodies end up in queues, in logs and in replays where the headers no longer exist.

What if a consumer never responds to deprecation notices? That is the normal case, which is why the email and the Slack message are not the mechanism: they are the courtesy. The mechanism is the per-consumer metric, which gives you a specific name to talk to, and the scheduled *brownout* —returning 410 during short, announced windows before the final shutdown. A fifteen-minute brownout gets more replies than three months of notices, because it turns an abstract future problem into a page in another team's on-call channel. Announce it with a date and a time, do it during business hours, and do not repeat it more than two or three times.

Are headers and status codes part of the contract? Yes, and they are a classic blind spot. Going from a 200 with an empty list to a 404 breaks anyone branching on status code, and it shows up in no schema diff if you never documented that 404. Same with `Location` on a 201, with `ETag` if anyone makes conditional requests, or with `Retry-After` on a 429. If it is documented, oasdiff watches it; if not, it is folklore.

Do rate limits and timeouts count too? Formally they are not part of the schema, but they are the part of the contract whose change gets noticed fastest. Dropping the limit from 1,000 to 100 requests per minute breaks integrations exactly like removing a field does, only at three in the morning and intermittently. Handle it with the same procedure: documented, versioned, announced in advance, and with a metric telling you who you are about to affect before you apply it.

How many versions should you keep alive at once? Two. One in production and one being retired. Once there are three, the test matrix multiplies, every bug fix has to be made three times and, worse, consumers learn that you never switch anything off and stop migrating. If you find yourself with three versions standing, the problem is almost never that you ship too fast: it is that you have not retired anything.

Glossary

- **Breaking change**: a modification that makes an existing client fail without that client changing its code.
- **CDC (consumer-driven contract)**: a contract written by the consumer, stating what it needs, and verified by the provider.
- **Broker**: the service that stores contracts, verification results, and which version is deployed in each environment.
- **Pending pact**: a contract the provider has not yet verified successfully; reported but non-blocking.
- **Provider state**: the data precondition the provider sets up before verifying an interaction.
- **Tolerant reader**: a client that reads only what it uses and ignores the rest, including values it does not recognise.

- **Brownout**: a brief, announced, scheduled outage of a deprecated endpoint, used to flush out consumers before shutdown.
- **Sunset**: the date from which a resource stops being available, announced with the header of the same name.

If you take away one idea: the goal is not to prevent breaking changes, but to make them **visible, deliberate and dated**. A team that can break its API on purpose, knowing exactly who it affects and how much runway they have, moves far faster than one that is afraid to touch anything.