

Code Review Efectivo: PRs Pequeños, Comentarios Claros y Automatización que Respeta el Tiempo de Todos

Versión en español - PuigFlows

Una revisión de código bien hecha es probablemente la herramienta barata más potente que tiene un equipo para mantener la calidad, repartir conocimiento y evitar incidentes. Mal hecha, es un cuello de botella que enfada a todos: PRs que esperan días, comentarios sobre comas mientras se cuele un fallo de diseño, y autores que aprenden a defenderse en lugar de a mejorar. Esta guía reúne lo que de verdad funciona: qué mirar y en qué orden, cómo mantener los cambios pequeños, cómo escribir comentarios que se puedan aceptar sin discutir y qué automatizar para que las personas se dediquen a lo que sólo ellas pueden juzgar.

El criterio que lo ordena todo: mejorar la salud del código

Las guías públicas de ingeniería de Google formulan el estándar de forma muy clara: un revisor debe aprobar un cambio en cuanto éste mejora de manera definitiva la salud global del código, aunque no sea perfecto. No existe el código perfecto, sólo código mejor. Esta idea desactiva dos patologías opuestas. La primera es el perfeccionismo del revisor, que bloquea un PR por preferencias personales. La segunda es la laxitud, que aprueba cualquier cosa «porque funciona» y deja que el sistema se degrade un PR razonable a la vez.

Una regla práctica que se deriva de ahí: las preferencias de estilo que no están en la guía del equipo son opiniones, no defectos, y se marcan como tales. Los principios de diseño, en cambio, no son una cuestión de gustos: si el cambio introduce complejidad innecesaria o acopla dos módulos que deberían ser independientes, eso sí se discute, aunque el autor haya terminado ya.

Qué revisar, y en qué orden

El orden importa porque el esfuerzo de un revisor es limitado y los problemas más caros de arreglar son los que se descubren más tarde. Una buena secuencia, de lo más estructural a lo más superficial:

1. Diseño. ¿Tiene sentido el cambio? ¿Vive en el lugar correcto? ¿Encaja con el resto del sistema o abre un camino paralelo que habrá que mantener? Si el diseño es incorrecto, nada de lo demás merece comentarios todavía.
2. Funcionalidad. ¿Hace lo que dice? Piensa en los casos límite: listas vacías, valores nulos, concurrencia, reintentos, zonas horarias, entradas hostiles.
3. Complejidad. ¿Podría entenderlo alguien que llega por primera vez dentro de seis meses? Funciones largas, abstracciones prematuras y flags booleanos en cascada son señales.
4. Pruebas. ¿Fallarían si el código se rompiera? Un test que pasa también con el bug no protege nada. Comprueba que cubren el caso nuevo y no sólo el camino feliz.
5. Nombres y comentarios. Los nombres deben comunicar intención; los comentarios deben explicar el porqué, no repetir el qué.
6. Estilo y formato. Aquí no debería haber trabajo humano: lo hacen las herramientas, como veremos más abajo.

Un buen hábito es leer primero la descripción y los tests, y sólo después el código de producción. Los tests dicen qué cree el autor que ha construido; si no coinciden con la descripción, ya tienes la primera pregunta.

El tamaño del cambio decide la calidad de la revisión

Hay una verdad incómoda en esto: la atención de una persona se degrada rápido con el volumen. Un PR de 50 líneas recibe comentarios útiles; uno de 1.500 recibe un «LGTM» cansado. Los estudios de revisión

clásicos de SmartBear/Cisco ya apuntaban en esa dirección: la efectividad en la detección de defectos cae cuando se revisan demasiadas líneas de una vez o a demasiada velocidad. No hace falta citar cifras exactas para actuar: si no puedes explicar el cambio en un par de frases, probablemente son dos cambios.

Técnicas que ayudan a mantener los PRs pequeños:

- Separa refactor y comportamiento. Primero un PR que mueve o renombra sin cambiar lógica (fácil de revisar porque no debería alterar ningún test), después el que añade la funcionalidad.
- Despliega detrás de un feature flag para poder integrar trabajo incompleto en main sin exponerlo.
- Apila cambios (stacked PRs) cuando uno depende de otro, en lugar de acumular todo en una rama de tres semanas.
- Las migraciones van aparte de la lógica que las usa.

Para que el tamaño sea visible y no una opinión, mídelo. Este script cuenta las líneas cambiadas respecto a la rama base e ignora ficheros generados que sólo añaden ruido:

```
#!/usr/bin/env bash
# pr-size.sh - mide el tamaño real de un cambio, ignorando archivos generados
set -euo pipefail
BASE="${1:-origin/main}"

git diff --numstat "$BASE"...HEAD -- . \
  ':(exclude)*.lock' ':(exclude)package-lock.json' ':(exclude)*.snap' ':(exclude)*.min.js' |
awk '{ add += $1; del += $2; files++ }
     END { printf "files=%d added=%d deleted=%d total=%d\n", files, add, del, add + del }'
```

Y esta acción de GitHub avisa en el propio PR cuando se pasa de un umbral. Fíjate en que emite un aviso y no un error: el objetivo es abrir una conversación, no bloquear. Hay cambios legítimamente grandes (una migración mecánica, una dependencia actualizada) y el equipo debe poder justificarlos.

```
name: pr-size
on:
  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  size:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - name: Medir tamaño del PR
        env:
          BASE: origin/${{ github.base_ref }}
        run: |
          total=$(git diff --numstat "$BASE"...HEAD -- . ':(exclude)*.lock' ':(exclude)package-lock.json' \
            | awk '{ s += $1 + $2 } END { print s + 0 }')
          echo "changed lines: $total"
          if [ "$total" -gt 800 ]; then
            echo "::warning::Este PR supera las 800 líneas. Considera dividirlo en cambios más pequeños."
          fi
```

La descripción es la mitad del PR

Un revisor que abre un PR sin contexto tiene que reconstruir la intención a partir del diff, y eso es caro y propenso a errores. Una buena descripción responde al porqué, no al qué (el qué ya lo muestra el diff), señala dónde está el riesgo e indica cómo probar el cambio. Una plantilla en `.github/pull_request_template.md` consigue que se rellene siempre:

```
## Qué cambia y por qué
<!-- El problema que resuelve, en dos o tres frases. Enlaza el ticket. -->

## Cómo está hecho
<!-- Decisiones de diseño y alternativas descartadas. -->
```

```

## Riesgo y despliegue
- [ ] Es seguro hacer rollback
- [ ] Hay migración de base de datos (adjunta el plan)
- [ ] Está detrás de un feature flag

## Cómo probarlo
<!-- Comandos, capturas o la consulta exacta que valida el cambio. -->

## Dónde quiero tu atención
<!-- Las partes con dudas. Lo demás es mecánico. -->

```

La última sección es la más valiosa. Decir «aquí tengo dudas, aquí no» dirige la atención del revisor a donde más la necesitas y hace que la revisión sea una colaboración, no un examen.

Automatiza lo mecánico para que las personas juzguen lo importante

Cada comentario sobre indentación, imports sin ordenar o una variable sin usar es tiempo humano malgastado y, además, genera fricción personal innecesaria: una herramienta puede ser inflexible sin resultar antipática. Todo lo que pueda decidir una máquina debería decidirlo una máquina, preferiblemente antes de que el PR exista.

En local: pre-commit

```

# .pre-commit-config.yaml
repos:
- repo: https://github.com/astral-sh/ruff-pre-commit
  rev: v0.8.0
  hooks:
    - id: ruff
      args: [--fix]
    - id: ruff-format
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v5.0.0
  hooks:
    - id: check-merge-conflict
    - id: detect-private-key
    - id: end-of-file-fixer

```

Los hooks se instalan una vez con pre-commit install y evitan que el PR llegue con ruido de formato. Repite las mismas comprobaciones en CI (con pre-commit run --all-files), porque un hook local siempre se puede saltar.

En el repositorio: CODEOWNERS y protección de ramas

El fichero CODEOWNERS hace que GitHub solicite automáticamente la revisión de las personas o equipos responsables de cada ruta. Combinado con una regla de protección de rama que exija la aprobación de los propietarios del código, garantiza que los cambios en zonas sensibles (pagos, infraestructura, migraciones) los vea quien conoce el área:

```

# .github/CODEOWNERS
# La última regla que coincide gana. Pon lo general arriba y lo específico abajo.
*                               @acme/engineering
/infra/                         @acme/platform
/db/migrations/                 @acme/data @acme/platform
/src/payments/                  @acme/payments
*.sql                           @acme/data

```

Cuidado con el exceso: si todo exige a cinco propietarios, la revisión se convierte en un trámite y el tiempo de espera se dispara. Reserva las aprobaciones obligatorias para lo que de verdad tiene riesgo y confía en el resto.

En CI: pruebas, análisis estático y seguridad

Que los tests, el linter, el comprobador de tipos y un análisis de dependencias se ejecuten antes de que nadie mire el PR cambia la dinámica: el revisor sabe que lo que llega ya compila, pasa las pruebas y cumple el

formato. Marca estas comprobaciones como obligatorias en la protección de la rama main.

Cómo escribir comentarios que se puedan aceptar

El mismo hallazgo puede provocar una mejora o una pelea según cómo se formule. Algunas pautas que funcionan:

- Comenta el código, no a la persona. «Esta función hace dos cosas» en lugar de «haces demasiadas cosas aquí».
- Explica el porqué. «Esto concatena la consulta con la entrada del usuario, lo que permite inyección SQL» enseña; «cámbialo» no.
- Pregunta cuando no estés seguro. «¿Qué pasa si el cupón ha caducado?» deja abierta la posibilidad de que el autor sepa algo que tú no.
- Señala la severidad. Un prefijo explícito evita que el autor tenga que adivinar si un comentario bloquea la aprobación.
- Reconoce lo bueno. Un «esta separación quedó muy limpia» cuesta diez segundos y cambia el tono de toda la revisión.

Una convención sencilla de prefijos que el equipo puede adoptar:

- blocking: hay que resolverlo antes de fusionar (fallo, riesgo de seguridad, rotura de un contrato).
- suggestion: mejora recomendada; el autor decide.
- question: necesito entender algo; no implica cambio.
- nit: detalle menor, opcional. Si hay muchos, probablemente falta una regla en el linter.

Un ejemplo completo

Veamos una función real y cómo se revisaría. Éste es el código que llega en el PR:

```
def apply_discount(order, code):
    coupon = db.query("SELECT * FROM coupons WHERE code = '%s'" % code).first()
    if coupon.percent > 0:
        order.total = order.total - order.total * coupon.percent / 100
    order.save()
    return order
```

Un buen revisor no empieza por el estilo. Empieza por lo que puede costar dinero o seguridad:

- blocking: la consulta se construye con interpolación de cadenas, lo que permite inyección SQL si code viene del usuario. Usa parámetros o la capa de acceso a datos del proyecto.
- blocking: si el cupón no existe, .first() devuelve None y la línea siguiente lanza AttributeError, que el cliente verá como un error 500 en lugar de «cupón inválido».
- question: ¿se comprueba en algún sitio que el cupón no haya caducado ni agotado sus usos?
- suggestion: ¿el total es un tipo flotante o decimal? Si es flotante, la aritmética de porcentajes acumulará errores de redondeo; para dinero usa Decimal.
- suggestion: el descuento puede dejar el total en negativo si el porcentaje supera 100; conviene limitarlo.
- nit: save(update_fields=[...]) evita sobrescribir campos modificados por otra petición concurrente.

Y una versión que resuelve los puntos bloqueantes y las sugerencias:

```
def apply_discount(order: Order, code: str) -> Order:
    coupon = coupons.get_active(code) # el repositorio filtra caducados y usa parámetros, nunca interpolación
    if coupon is None:
        raise InvalidCoupon(code)

    discount = (order.total * coupon.percent / 100).quantize(Decimal("0.01"))
    order.total = max(order.total - discount, Decimal("0"))
    order.save(update_fields=["total"])
    return order
```

Observa que en toda la revisión sólo dos comentarios bloquean. El resto son mejoras razonadas que el autor puede aceptar o discutir. Ese equilibrio es lo que mantiene la revisión productiva.

La velocidad también es calidad

Un PR que espera dos días obliga al autor a cambiar de contexto, aumenta los conflictos de fusión y desincentiva los cambios pequeños, que es justo lo contrario de lo que queremos. Las guías de Google proponen que, salvo casos excepcionales, la respuesta a una petición de revisión llegue en un día laborable como máximo; muchos equipos fijan un objetivo más ambicioso, de pocas horas. Lo importante es acordar un compromiso explícito y medirlo.

Este script consulta la API de GitHub y calcula la mediana y el percentil 90 del tiempo hasta la primera revisión de los últimos PRs fusionados. Es un punto de partida: en un repositorio grande tendrás que paginar y manejar el límite de peticiones.

```
import os
import statistics
from datetime import datetime

import requests

REPO = os.environ["REPO"] # "acme/api"
HEADERS = {
    "Authorization": f"Bearer {os.environ['GITHUB_TOKEN']}",
    "Accept": "application/vnd.github+json",
}
API = f"https://api.github.com/repos/{REPO}"

def ts(value: str) -> datetime:
    return datetime.fromisoformat(value.replace("Z", "+00:00"))

def first_review_hours(pr: dict) -> float | None:
    reviews = requests.get(f"{API}/pulls/{pr['number']}/reviews", headers=HEADERS, timeout=30).json()
    submitted = [ts(r["submitted_at"]) for r in reviews
                  if r.get("submitted_at") and r["user"]["login"] != pr["user"]["login"]]
    if not submitted:
        return None
    return (min(submitted) - ts(pr["created_at"])).total_seconds() / 3600

prs = requests.get(f"{API}/pulls", headers=HEADERS, timeout=30,
                  params={"state": "closed", "per_page": 50, "sort": "updated", "direction": "desc"}).json()
merged = [p for p in prs if p.get("merged_at")]
waits = [h for p in merged if (h := first_review_hours(p)) is not None]

print(f"PRs: {len(waits)}")
print(f"Mediana hasta la primera revisión: {statistics.median(waits):.1f} h")
print(f"p90: {statistics.quantiles(waits, n=10)[-1]:.1f} h")
```

Mira la mediana para ver el funcionamiento normal y el percentil 90 para detectar los PRs que se quedan olvidados. Si ese segundo número es mucho mayor, hay un problema de asignación (siempre se pide al mismo experto) o de visibilidad. No uses la métrica para evaluar a individuos: medirás lo que se vea bien en lugar de lo que sea bueno.

Errores frecuentes

- Revisar sólo el diff. Algunos defectos se ven al abrir el archivo completo o al buscar quién más llama a la función modificada.
- Aprobar sin leer por confianza en el autor. La revisión no es una auditoría de personas; es una red de seguridad.
- Pedir reescrituras por gusto personal. Si no hay un principio detrás, es un nit.
- Acumular comentarios en rondas sucesivas. Haz una revisión completa a la primera, para que el autor no

reciba nuevas objeciones cada vez que actualiza.

- Debates largos en los comentarios. Si dos intercambios no resuelven la discusión, llamad o hablad; después dejad el resumen escrito en el PR.
- Un único revisor para todo. Eso crea un cuello de botella y concentra el conocimiento. Rotad.
- Ignorar el contexto emocional. Tras la pantalla hay una persona que ha dedicado horas a su trabajo. La claridad y la amabilidad no son incompatibles.

Revisar cambios generados con IA

Los asistentes de código producen hoy una parte creciente de los cambios, y eso no relaja la revisión, la endurece. El código generado suele ser plausible y bien formateado, lo que lo hace más difícil de cuestionar. Algunas precauciones: exige al autor que entienda y pueda explicar cada línea que envía; verifica que las dependencias importadas existen y son las que se pretendían; desconfía de tests que parecen cubrir mucho pero no afirman nada; y presta atención extra a la gestión de errores, que es donde el código generado suele quedarse en lo genérico. La responsabilidad del cambio es de quien abre el PR, venga el código de donde venga.

Checklist para el revisor

- He leído la descripción y entiendo el objetivo del cambio.
- El diseño es adecuado y vive en el lugar correcto.
- He pensado en los casos límite y en los fallos (nulos, vacíos, concurrencia, reintentos).
- Hay pruebas que fallarían si el código se rompiera.
- No hay secretos, datos personales en logs ni consultas construidas con entrada del usuario.
- Los nombres comunican intención y los comentarios explican el porqué.
- Mis comentarios están marcados por severidad y argumentados.
- He respondido dentro del plazo acordado.

Checklist para el autor

- El cambio hace una sola cosa y puedo explicarlo en dos frases.
- La descripción explica el porqué, el riesgo y cómo probarlo.
- Me he revisado a mí mismo el diff antes de pedir revisión.
- Los tests y el linter pasan en local y en CI.
- He señalado dónde quiero atención específica.
- Contesto a cada comentario: aplicado, discutido o aplazado con un ticket.

Revisar en tres pasadas: un método para no perderse

Quien revisa un PR leyendo el diff de arriba abajo, archivo por archivo, en el orden alfabético que impone la interfaz, hace la peor versión posible del trabajo: llega al archivo importante con la atención gastada y juzga los detalles antes de haber entendido la intención. Un método más fiable consiste en separar la revisión en tres pasadas con objetivos distintos, cada una con un coste de atención diferente.

Primera pasada: orientación (dos o tres minutos). No se mira el código todavía. Se lee el título, la descripción, el ticket enlazado y la lista de archivos tocados. La pregunta es: ¿qué problema resuelve esto y qué forma esperaría yo que tuviera la solución? Si el PR toca quince archivos y la descripción habla de un cambio de texto en un botón, ya hay una discrepancia que merece una pregunta antes de leer una sola línea. Esta pasada también decide si el PR es revisable: si no lo es (demasiado grande, sin descripción, mezcla tres cambios), lo más barato para todos es devolverlo con una petición concreta de división, no revisarlo a medias.

Segunda pasada: estructura (cinco a diez minutos). Se leen primero los tests y después los archivos de mayor responsabilidad: interfaces públicas, esquemas, migraciones, contratos de API. Se busca si el diseño es el adecuado, si la lógica vive en la capa correcta, si hay duplicación con algo que ya existe y si los casos límite están cubiertos. Aquí se escriben los comentarios de mayor impacto, los que cambian la forma del código. Conviene dejarlos escritos antes de pasar a los detalles, porque si el autor tiene que rehacer la estructura, los comentarios de estilo sobre el código antiguo habrán sido tiempo perdido para los dos.

Tercera pasada: detalle (el tiempo que haga falta, con límite). Nombres, mensajes de error, manejo de nulos, concurrencia, logs, rendimiento en bucles calientes, seguridad. Es la pasada que más se parece a lo que la gente imagina cuando piensa en una revisión, y es la que más se beneficia de la automatización previa: si el linter ya resolvió el formato, esta pasada puede dedicarse a lo que sólo un humano ve.

Para que el orden de lectura no dependa de la interfaz, este pequeño script de Python ordena los archivos de un PR por prioridad de revisión: primero tests, luego contratos y migraciones, luego el resto por cantidad de cambios. Sirve como ayuda de lectura, no como regla.

```
#!/usr/bin/env python3
"""review_order.py - lista los archivos de un PR en el orden en que conviene leerlos.

Uso: gh pr diff 1234 --name-only | python3 review_order.py
"""
import re
import subprocess
import sys

# Cuanto menor el número, antes se lee.
PRIORIDADES = [
    (0, re.compile(r"^(^/)(tests?|__tests__)/|_test\.|\.\test\.|\.\spec\.")),
    (1, re.compile(r"migrations?/|schema|\.\proto$|openapi|\.\sql$")),
    (2, re.compile(r"^(^/)(api|routes?|handlers?|contracts?)/")),
    (3, re.compile(r"Dockerfile|\.ya?ml$|terraform/|\.\tf$")),
]

def prioridad(ruta: str) -> int:
    for nivel, patron in PRIORIDADES:
        if patron.search(ruta):
            return nivel
    return 4

def cambios(ruta: str, base: str) -> int:
    salida = subprocess.run(
        ["git", "diff", "--numstat", base, "--", ruta],
        capture_output=True, text=True, check=False,
    ).stdout.split()
    try:
        return int(salida[0]) + int(salida[1])
    except (IndexError, ValueError):
        return 0

def main() -> None:
    base = sys.argv[1] if len(sys.argv) > 1 else "origin/main"
    rutas = [l.strip() for l in sys.stdin if l.strip()]
    ordenadas = sorted(rutas, key=lambda r: (prioridad(r), -cambios(r, base)))
    for r in ordenadas:
        print(f"[{prioridad(r)}] {r}")

if __name__ == "__main__":
    main()
```

El valor del script no está en el código, sino en la decisión que fija: los tests se leen primero porque son la especificación ejecutable del cambio. Si los tests no cuentan la misma historia que la descripción, ya sabes dónde preguntar.

Revisión de seguridad y fiabilidad: tres bugs que se cuelan por costumbre

Los bugs que más se escapan en una revisión no son los exóticos. Son los que tienen buen aspecto en el diff

porque el código está bien escrito y es plausible. Estos tres aparecen una y otra vez, y vale la pena tener el ojo entrenado para ellos.

1. La autorización que falta, no la que sobra

Un diff muestra lo que se añade, y la ausencia de una comprobación no aparece en ningún lado. Fíjate en este endpoint, que parece correcto:

```
@router.get("/facturas/{factura_id}")
async def obtener_factura(factura_id: int, usuario: Usuario = Depends(usuario_actual)):
    factura = await repo.obtener(factura_id)
    if factura is None:
        raise HTTPException(404)
    return factura
```

Autentica al usuario pero no comprueba que la factura sea suya: cualquiera con sesión puede leer las facturas de otro cambiando el número. La pregunta de revisión que lo detecta es siempre la misma: ¿qué impide que el usuario A acceda al recurso del usuario B? La versión corregida filtra por propietario en la propia consulta, de modo que el recurso ajeno es indistinguible de uno inexistente:

```
factura = await repo.obtener(factura_id, propietario_id=usuario.id)
if factura is None:
    raise HTTPException(404) # mismo error para "no existe" y "no es tuya"
```

2. La condición de carrera disfrazada de lógica correcta

Leer, decidir y escribir en pasos separados funciona en el test y falla bajo carga:

```
saldo = await cuentas.saldo(cuenta_id)
if saldo >= importe:
    await cuentas.actualizar_saldo(cuenta_id, saldo - importe)
```

Dos peticiones simultáneas leen el mismo saldo y las dos descuentan. En revisión, la señal es un patrón de lectura seguida de escritura sobre el mismo dato sin bloqueo ni condición atómica. La corrección mueve la condición a la base de datos:

```
UPDATE cuentas
SET saldo = saldo - :importe
WHERE id = :cuenta_id AND saldo >= :importe
RETURNING saldo;
-- 0 filas devueltas = saldo insuficiente, sin ventana de carrera
```

3. El error tragado

```
try:
    await enviar_confirmacion(pedido)
except Exception:
    pass
```

Un except amplio con pass convierte un fallo en un silencio. En revisión, cada captura de excepción debería responder a tres preguntas: qué fallo concreto se espera, qué se hace en consecuencia y dónde queda registrado. Si la respuesta es «ninguna», probablemente sobra la captura o falta el log. Una versión defendible captura sólo lo esperado, registra con contexto y decide explícitamente si el fallo debe propagarse:

```
try:
    await enviar_confirmacion(pedido)
except ErrorTemporalDeCorreo as exc:
    logger.warning("confirmación diferida", extra={"pedido": pedido.id, "error": str(exc)})
    await cola_reintentos.encolar(pedido.id)
```

Estos tres ejemplos comparten una lección: el revisor aporta más valor cuando pregunta por lo que no está en el diff (la comprobación ausente, la atomicidad, el camino de fallo) que cuando comenta lo que sí está.

Las reglas de la rama como código: rulesets, aprobaciones obsoletas y cola de fusión

Una política de revisión que vive sólo en la cabeza del equipo dura hasta el primer viernes con prisa. Las

reglas importantes conviene escribirlas donde la plataforma las haga cumplir. En GitHub, los rulesets permiten definir esas reglas por rama o por patrón de ramas, versionarlas y aplicarlas a varios repositorios. Estas son las reglas que más rendimiento dan, y por qué:

- Exigir pull request antes de fusionar, con al menos una aprobación. Es la base: sin esto, todo lo demás es opcional.
- Descartar aprobaciones obsoletas al subir nuevos commits. Sin esta regla, alguien aprueba un PR de veinte líneas y el autor añade después trescientas sin que nadie las mire. La aprobación debe referirse al código que se fusiona, no a una versión anterior.
- Exigir aprobación del último push revisable por alguien distinto de quien lo hizo. Evita que el autor empuje un cambio de última hora y se autoapruebe con la aprobación previa de otra persona.
- Exigir revisión de los propietarios del código (CODEOWNERS) para las rutas sensibles: autenticación, pagos, infraestructura, migraciones.
- Exigir que se resuelvan todas las conversaciones antes de fusionar, para que un comentario bloqueante no quede enterrado.
- Exigir comprobaciones de estado (tests, linter, análisis de seguridad) y que la rama esté actualizada con la base, o usar la cola de fusión (merge queue), que prueba cada PR contra el estado real de la rama principal antes de integrarlo.

Estas reglas se pueden declarar como código y aplicarse con la CLI de GitHub. Este ejemplo crea un ruleset para la rama principal; ajusta los nombres de las comprobaciones a los de tu CI:

```
#!/usr/bin/env bash
# crear-ruleset.sh - protege la rama por defecto con reglas de revisión
set -euo pipefail
REPO="$1" # uso: crear-ruleset.sh owner/repo

gh api --method POST "repos/$REPO/rulesets" --input - <<'JSON'
{
  "name": "revision-obligatoria",
  "target": "branch",
  "enforcement": "active",
  "conditions": { "ref_name": { "include": ["~DEFAULT_BRANCH"], "exclude": [] } },
  "rules": [
    { "type": "pull_request",
      "parameters": {
        "required_approving_review_count": 1,
        "dismiss_stale_reviews_on_push": true,
        "require_code_owner_review": true,
        "require_last_push_approval": true,
        "required_review_thread_resolution": true
      } },
    { "type": "required_status_checks",
      "parameters": {
        "strict_required_status_checks_policy": true,
        "required_status_checks": [ { "context": "ci / test" }, { "context": "ci / lint" } ]
      } },
    { "type": "non_fast_forward" }
  ]
}
JSON
```

Un detalle práctico: empieza con "enforcement": "evaluate" (modo de evaluación) durante una semana. Registra qué PRs habrían sido bloqueados sin bloquear a nadie, y te deja ajustar las reglas antes de que causen fricción real. Pasar a active después es una línea de diff.

Una regla que suele olvidarse: el bypass

Cada regla necesita una vía de escape para emergencias, pero esa vía debe ser estrecha y auditable. Limita el bypass a un rol concreto (por ejemplo, el equipo de guardia), y exige que su uso deje rastro: un PR de seguimiento, una nota en el canal de incidentes. Si la mitad del equipo puede saltarse la revisión «sólo esta vez», la regla es decorativa.

CODEOWNERS bien diseñado: pocos dueños, rutas claras

El archivo CODEOWNERS asigna revisores automáticos por ruta. El fallo habitual es el exceso: si todo el repositorio tiene tres dueños, cada PR los notifica a todos, y la notificación deja de significar algo. Un diseño sano tiene tres rasgos. Primero, la regla general asigna a un equipo, no a personas, para que las vacaciones de alguien no bloqueen nada. Segundo, las rutas sensibles (seguridad, pagos, infraestructura) tienen dueños específicos con conocimiento real. Tercero, el archivo se revisa como cualquier otro: una ruta sin dueño o con un dueño que ya dejó la empresa es una deuda silenciosa.

```
# .github/CODEOWNERS
# Regla general: cualquier PR necesita al menos al equipo de plataforma
* @acme/plataforma

# Rutas sensibles: dueños con conocimiento específico
/src/auth/ @acme/seguridad
/src/pagos/ @acme/pagos @acme/seguridad
/db/migrations/ @acme/datos
/infra/ @acme/sre
/.github/workflows/ @acme/sre @acme/seguridad

# El propio CODEOWNERS lo revisa quien lo gobierna
/.github/CODEOWNERS @acme/plataforma
```

En CODEOWNERS la última regla que coincide gana, de modo que las más específicas van al final. Un error frecuente es poner la regla general * al final y anular sin querer todas las anteriores.

Medir la revisión con datos reales: tiempo hasta la primera respuesta

Una métrica útil para una revisión sana es el tiempo hasta la primera respuesta humana, porque es lo que el autor siente como «bloqueo». El siguiente script consulta la API GraphQL de GitHub y calcula la mediana y el percentil 90 de los últimos PRs fusionados. Excluye a los bots para no falsear el resultado.

```
#!/usr/bin/env python3
"""tiempo_primera_revision.py - mediana y p90 de espera hasta la primera revisión humana."""
import json
import statistics
import subprocess
import sys
from datetime import datetime

QUERY = """
query($owner:String!, $repo:String!) {
  repository(owner:$owner, name:$repo) {
    pullRequests(last: 100, states: MERGED) {
      nodes {
        createdAt
        author { login }
        reviews(first: 20) { nodes { submittedAt author { __typename login } } }
      }
    }
  }
}
"""

def parse(ts: str) -> datetime:
    return datetime.fromisoformat(ts.replace("Z", "+00:00"))

def main(owner: str, repo: str) -> None:
    salida = subprocess.run(
        ["gh", "api", "graphql", "-f", f"query={QUERY}", "-F", f"owner={owner}", "-F", f"repo={repo}"],
        capture_output=True, text=True, check=True,
    ).stdout
    prs = json.loads(salida)["data"]["repository"]["pullRequests"]["nodes"]
    esperas = []
    for pr in prs:
        autor = (pr["author"] or {}).get("login")
        humanas = [
            parse(r["submittedAt"]) for r in pr["reviews"]["nodes"]
            if r["author"] and r["author"].get("login") != autor
        ]
        if humanas:
            esperas.append(max(humanas))
```

```

        if r["author"] and r["author"]["__typename"] == "User" and r["author"]["login"] != autor
    ]
    if humanas:
        esperas.append((min(humanas) - parse(pr["createdAt"])).total_seconds() / 3600)
if len(esperas) < 10:
    sys.exit("Muestra demasiado pequeña para sacar conclusiones")
esperas.sort()
p90 = esperas[int(len(esperas) * 0.9) - 1]
print(f"PRs analizados: {len(esperas)}")
print(f"Mediana: {statistics.median(esperas):.1f} h | p90: {p90:.1f} h")

if __name__ == "__main__":
    main(sys.argv[1], sys.argv[2])

```

Mira el p90 antes que la mediana. La mediana suele verse bien incluso en equipos donde uno de cada diez PRs espera tres días, y esos son los que generan frustración y ramas desactualizadas. Si el p90 es varias veces la mediana, el problema es de reparto de la carga o de PRs que nadie quiere revisar, y eso se arregla con guardias de revisión, no con exigir más rapidez.

Caso práctico: cómo partir un PR de 1.200 líneas en cuatro

Imagina un PR titulado «Soporte de descuentos por volumen». Toca 31 archivos y suma 1.200 líneas: una migración que añade la tabla de tramos, un refactor del cálculo de precios que mueve funciones de sitio, la lógica nueva, el endpoint, y cambios de formato que el editor aplicó de paso. Nadie lo va a revisar bien. La pregunta es cómo partirlo sin perder coherencia, y la respuesta es pensar en qué se puede fusionar y desplegar sin cambiar el comportamiento observable.

1. PR 1, refactor puro (~300 líneas). Mueve el cálculo de precios a su propio módulo sin cambiar nada. Los tests existentes deben pasar sin modificarse; esa es la prueba de que es un refactor. Se revisa en diez minutos porque el revisor sólo comprueba que nada cambió.
2. PR 2, migración aditiva (~80 líneas). Crea la tabla de tramos, vacía. No hay código que la use todavía. Se despliega con seguridad porque añadir una tabla no rompe nada.
3. PR 3, lógica detrás de un feature flag (~500 líneas). Implementa el cálculo con descuentos y lo activa sólo si el flag descuentos_volumen está encendido. En producción no cambia nada hasta que se encienda, así que el riesgo de fusionar es mínimo y el revisor puede concentrarse en la corrección de la lógica y de sus tests.
4. PR 4, exposición (~200 líneas). Endpoint y documentación. Con el resto ya en la rama principal, este PR es pequeño y legible.

Observa que cada PR tiene una única razón para existir y puede revertirse de forma independiente. El coste es algo más de coordinación; el beneficio es que las cuatro revisiones, sumadas, son más rápidas y encuentran más problemas que la revisión única de 1.200 líneas.

PRs apilados: cuando los cambios dependen unos de otros

Si el PR 3 depende del 2 y no quieres esperar a que se fusione, puedes apilarlos: cada rama parte de la anterior y cada PR apunta a la rama de su antecesor. El flujo con git puro es sencillo, pero tiene un punto delicado: al fusionar el primero, los demás deben reubicarse sobre la rama principal.

```

# Pila: main <- refactor-precios <- tramos-migracion <- descuentos-logica
git switch -c refactor-precios main
# ...commits...
git switch -c tramos-migracion refactor-precios
# ...commits...
git switch -c descuentos-logica tramos-migracion
# ...commits...

# Tras fusionar refactor-precios en main (con squash), reubica el resto de la pila:
git fetch origin
git rebase --onto origin/main refactor-precios tramos-migracion
git rebase --onto tramos-migracion refactor-precios descuentos-logica # si hace falta ajustar
git push --force-with-lease origin tramos-migracion descuentos-logica

```

El uso de `--onto` es lo que evita arrastrar los commits ya fusionados. Y `--force-with-lease`, en vez de `--force`, falla si alguien más empujó a esa rama, lo que protege el trabajo ajeno. Herramientas de pila como Graphite, `git-branchless` o `gh stack` automatizan estos pasos, pero es útil entender qué hacen por debajo antes de delegarlo.

Revisar migraciones de base de datos: el cambio que no se puede deshacer con un revert

Un `revert` de código revierte código. No revierte los datos que una migración ya transformó ni el bloqueo que dejó una tabla inaccesible durante diez minutos. Por eso las migraciones merecen una revisión específica con estas preguntas:

- ¿Es compatible con la versión anterior del código? Durante un despliegue gradual conviven dos versiones del servicio contra la misma base de datos. Si la migración elimina una columna que el código antiguo todavía lee, el despliegue rompe a mitad de camino.
- ¿Qué bloqueo toma y durante cuánto? Añadir una columna con valor por defecto, crear un índice o añadir una restricción puede bloquear escrituras en una tabla grande. Debe usarse la variante concurrente o por fases.
- ¿Tiene una ruta de vuelta? No hace falta una migración inversa para todo, pero sí saber qué se haría ante un fallo.
- ¿Se ha probado con un volumen realista? Una migración que tarda dos segundos con mil filas puede tardar una hora con cien millones.

El patrón que permite cambios seguros se llama expandir y contraer (`expand/contract`). Para renombrar una columna sin parar el servicio, se reparte en varios despliegues:

```
-- Despliegue 1 (expandir): añadir la columna nueva, sin tocar la antigua
ALTER TABLE clientes ADD COLUMN nombre_completo text;

-- Despliegue 2: el código escribe en ambas columnas y lee de la antigua.
-- Relleno en lotes, fuera de la migración, para no mantener un bloqueo largo:
UPDATE clientes
  SET nombre_completo = nombre
 WHERE id BETWEEN 1 AND 50000 AND nombre_completo IS NULL;
-- ...repetir por lotes hasta terminar

-- Despliegue 3: el código lee de la columna nueva (la antigua se sigue escribiendo)

-- Despliegue 4 (contraer): cuando ninguna versión desplegada usa la antigua
ALTER TABLE clientes DROP COLUMN nombre;

-- Índices sobre tablas grandes: sin bloquear escrituras
CREATE INDEX CONCURRENTLY idx_clientes_nombre_completo ON clientes (nombre_completo);
```

Un revisor que conoce este patrón reconoce al instante el PR que hace `RENAME COLUMN` en un solo paso y sabe qué pedir: dividirlo en fases. Es uno de los comentarios con más retorno que puede dejar.

Revisar tests: que prueben comportamiento, no que den verde

Un test que pasa no demuestra que el código sea correcto; demuestra que el código hace lo que el test comprueba. Al revisar tests, la pregunta útil es qué fallaría si el código estuviera roto. Estas son las señales de un test débil:

- Aserciones que no pueden fallar, como comprobar que el resultado «no es nulo» cuando la función nunca devuelve nulo.
- Mocks tan extensos que el test verifica el mock y no el código.
- Un único caso feliz, sin casos límite: lista vacía, valor negativo, entrada con caracteres Unicode, zona horaria.
- Tests que replican la implementación (calculan el valor esperado con la misma fórmula del código), de

modo que un error en la fórmula pasa desapercibido.

- Datos aleatorios sin semilla fija, que producen fallos intermitentes imposibles de reproducir.

Un truco de revisión poco usado es hacer una mutación mental: elige una línea de la lógica, imagina que la cambias (un `>=` por un `>`) y pregúntate qué test lo detectaría. Si ninguno, hay un hueco. Herramientas de pruebas de mutación como `mutmut` en Python o `Stryker` en JavaScript automatizan esta idea, y aunque no conviene ejecutarlas en cada PR por su coste, un análisis periódico de los módulos críticos revela dónde la cobertura es sólo aparente.

```
def aplicar_descuento(total: Decimal, unidades: int) -> Decimal:
    if unidades >= 100:
        return total * Decimal("0.85")
    if unidades >= 10:
        return total * Decimal("0.95")
    return total

# Test débil: sólo comprueba un punto intermedio de cada tramo.
def test_descuento_debil():
    assert aplicar_descuento(Decimal("100"), 50) == Decimal("95")

# Test fuerte: ataca los límites, que es donde viven los errores de un solo paso.
@pytest.mark.parametrize("unidades,esperado", [
    (9, Decimal("100")), # justo antes del primer tramo
    (10, Decimal("95")), # primer valor del tramo
    (99, Decimal("95")), # último valor del tramo
    (100, Decimal("85")), # primer valor del tramo superior
    (0, Decimal("100")), # caso degenerado
])
def test_descuento_en_los_limites(unidades, esperado):
    assert aplicar_descuento(Decimal("100"), unidades) == esperado
```

Si cambiaras `>= 10` por `> 10`, el test débil seguiría en verde y el fuerte fallaría en el caso de diez unidades. Esa diferencia es lo que un revisor atento puede señalar con una sola frase: «¿qué pasa exactamente en 10 y en 100?».

Un hilo de revisión completo, comentado

La teoría se asienta mejor con un caso. Este es un fragmento de revisión sobre un PR que añade reintentos a una llamada a una pasarela de pagos. El código original:

```
def cobrar(pedido):
    for intento in range(5):
        try:
            return pasarela.cobrar(pedido.importe, pedido.tarjeta)
        except Exception:
            time.sleep(1)
    return None
```

Una revisión de poca calidad diría «este bucle es feo» o «usa una librería de reintentos». Una buena revisión localiza los problemas reales, los ordena por gravedad y propone una salida concreta:

bloqueante: Si la pasarela procesó el cobro pero la respuesta se perdió (un `timeout`), el reintento cobrará al cliente dos veces. Necesitamos una clave de idempotencia estable por pedido; la mayoría de las pasarelas la aceptan en una cabecera. Sin ella, estos reintentos no son seguros.

bloqueante: `except Exception` reintentará también ante errores permanentes (tarjeta rechazada, importe inválido). Cinco intentos contra un rechazo definitivo sólo retrasan el error y pueden activar los sistemas antifraude del banco. Reintentaríamos sólo ante errores de red y respuestas 5xx.

importante: Devolver `None` al agotar los intentos hace que el llamador tenga que adivinar qué pasó. Mejor lanzar una excepción propia con el último error como causa.

sugerencia: Una espera fija de un segundo bajo carga sincroniza a todos los clientes que reintentan. Retroceso exponencial con jitter lo evitaría. No bloquea este PR.

pregunta: ¿Hay un test que simule un `timeout` tras cobro exitoso? Es el escenario más peligroso y el que

menos se prueba.

La versión resultante tras la revisión:

```
import random
import time

class CobroFallido(Exception):
    """Se agotaron los reintentos o el error no era reintentable."""

ERRORES_REINTENTABLES = (TimeoutError, ConnectionError, ErrorServidor5xx)

def cobrar(pedido, max_intentos: int = 4):
    clave = f"cobro-{pedido.id}" # estable entre reintentos: la pasarela deduplica
    ultimo_error = None
    for intento in range(max_intentos):
        try:
            return pasarela.cobrar(
                pedido.importe, pedido.tarjeta, idempotency_key=clave,
            )
        except ERRORES_REINTENTABLES as exc:
            ultimo_error = exc
            espera = min(8, 2 ** intento) * random.uniform(0.5, 1.0)
            logger.warning("cobro reintentable", extra={"pedido": pedido.id, "intento": intento + 1})
            time.sleep(espera)
        except TarjetaRechazada:
            raise # permanente: reintentar no ayuda
    raise CobroFallido(f"pedido {pedido.id}") from ultimo_error
```

Observa lo que hizo funcionar esta revisión: cada comentario explica la consecuencia, no sólo la regla; los dos bloqueantes se distinguen claramente de la sugerencia; y la pregunta final pide un test, no una opinión. El autor sabe qué debe cambiar para fusionar y qué puede dejar para después.

Revisar rendimiento sin caer en la optimización prematura

«Esto podría ser más rápido» es un comentario barato y casi siempre inútil. Para que un comentario de rendimiento merezca la pena debe cumplir una de dos condiciones: señalar una complejidad que escala mal con datos reales (una consulta dentro de un bucle, una búsqueda lineal sobre una lista que crece), o apoyarse en una medición. Las señales que justifican un comentario son pocas y reconocibles: consultas a base de datos o llamadas de red dentro de bucles, carga de colecciones enteras en memoria cuando bastaría un iterador, ausencia de paginación en endpoints que devuelven listas, e índices que faltan para las nuevas consultas. En el resto de los casos, la regla es pedir datos antes que reescrituras: «¿has medido cuánto tarda con diez mil filas?» es una pregunta que orienta; «yo lo haría con un diccionario» es una opinión.

Revisar dependencias, configuración e infraestructura como código

Hay cambios que casi no tienen líneas pero mucho riesgo. Una actualización de dependencia son dos líneas en un archivo de bloqueo; un cambio en un workflow de CI es un puñado de YAML. La revisión de estos cambios exige otra mirada.

Dependencias. Antes de aprobar un PR que añade o actualiza una librería, comprueba tres cosas: que es necesaria (¿podrían resolverlo diez líneas propias?), que está mantenida (fecha del último lanzamiento, issues abiertas, número de mantenedores) y que su licencia es compatible con tu uso. En las actualizaciones, lee las notas de la versión buscando cambios incompatibles. Los PRs automáticos de Dependabot o Renovate merecen la misma atención que uno humano: que los abra un bot no significa que su contenido sea inocuo.

Workflows de CI. Un workflow con permisos amplios es una puerta de entrada. Al revisar, comprueba que el bloque permissions concede lo mínimo, que las acciones de terceros están fijadas a un hash de commit y no a una etiqueta mutable, y que ningún paso interpola datos controlados por un atacante (como el título de un PR) dentro de un script. Este último fallo permite ejecutar código arbitrario con los secretos del repositorio:

```
# Peligroso: el título del PR se interpola directamente en el script de shell
- name: Saludar
  run: echo "Nuevo PR: ${github.event.pull_request.title}"
```

```
# Seguro: el dato pasa por una variable de entorno y no se interpreta como código
- name: Saludar
  env:
    TITULO: ${github.event.pull_request.title}
  run: echo "Nuevo PR: $TITULO"
```

Infraestructura como código. En Terraform u OpenTofu, el diff del código no basta: lo que importa es el plan. Un cambio de una línea puede destruir y recrear un recurso con estado, como una base de datos. Haz que el plan se publique como comentario en el PR y revísalo buscando las palabras destroy y replace. Aprobar un cambio de infraestructura sin leer el plan es aprobar a ciegas.

Cuando no estáis de acuerdo: cómo resolver un desacuerdo técnico

Tarde o temprano, autor y revisor discrepan y ninguno cede. Los hilos de comentarios que se alargan a diez mensajes rara vez se resuelven por escrito. Estas reglas evitan que un desacuerdo se convierta en un conflicto:

1. Distingue la preferencia del problema. Si el revisor no puede explicar qué se rompe, qué se vuelve más difícil de mantener o qué riesgo se introduce, probablemente es una preferencia, y debería marcarse como sugerencia no bloqueante o retirarse.
2. Cita el criterio, no la autoridad. «Prefiero esto» y «el senior lo hace así» no son argumentos. «Esto crea una dependencia circular entre dos módulos» sí lo es.
3. Pasa a una conversación en directo a la tercera ronda. Cinco minutos de llamada resuelven lo que veinte comentarios no. Después, deja constancia por escrito de lo acordado en el PR, para quien lo lea más tarde.
4. Escala con un tercero, no con jerarquía. Pide a una tercera persona con criterio técnico que desempate; si el desacuerdo es de fondo, probablemente deba convertirse en un ADR o una discusión de diseño, no resolverse dentro de un PR.
5. Cuando la decisión sea cercana, gana el autor. Si ambas opciones son razonables y no hay un riesgo claro, la regla de las guías de Google es dejar que el autor decida. Evita el bloqueo por gusto.

Y una nota sobre la dinámica de poder: los comentarios de alguien con más antigüedad pesan más de lo que su autor cree. Un «¿no sería mejor...?» de un lead puede leerse como una orden. Quien tiene autoridad debería ser especialmente explícito sobre si un comentario es bloqueante o no.

Revisión entre zonas horarias: cómo no perder un día por comentario

En equipos distribuidos, cada ida y vuelta de un PR puede costar un día. Si el autor está en Madrid y el revisor en Ciudad de México, una conversación de cuatro rondas consume una semana. Hay prácticas que lo reducen. Escribe comentarios completos: en lugar de «¿esto por qué?», explica qué te preocupa y propón una alternativa, para que el autor pueda actuar sin preguntar de nuevo. Aprueba condicionalmente cuando los cambios pedidos sean menores («Aprobado, siempre que se resuelvan los dos comentarios marcados como bloqueantes»), de modo que el autor pueda fusionar al terminar sin esperar otra ronda. Y fija una ventana de solapamiento para revisiones que requieran conversación, en vez de dejar que cada PR dependa de la coincidencia fortuita de horarios.

Revisión asistida por IA: una configuración razonable

Los revisores automáticos con modelos de lenguaje son útiles para una primera pasada: detectan errores evidentes, inconsistencias entre comentario y código, validaciones que faltan. Tienen límites claros: no conocen la historia de tu producto, ni por qué se tomó una decisión hace dos años, ni qué compromisos has hecho con un cliente. Una configuración sensata los trata como un revisor junior muy rápido: sus comentarios se evalúan, no se obedecen, y su aprobación nunca sustituye a la humana.

Para que sean útiles y no ruidosos, dales instrucciones específicas del repositorio. La mayoría de las

herramientas aceptan un archivo de instrucciones; este es un ejemplo del tipo de contenido que funciona:

```
# Instrucciones para el revisor automático

## Prioridades (en este orden)
1. Errores de corrección y condiciones de carrera.
2. Seguridad: autorización ausente, inyección, secretos en el código.
3. Manejo de errores: excepciones tragadas, fallos sin registrar.
4. Tests: casos límite sin cubrir.

## Ignorar
- Formato, orden de imports y estilo (los resuelve el linter).
- Sugerencias de renombrar variables salvo que induzcan a error.

## Convenciones del proyecto
- Los importes monetarios usan Decimal, nunca float.
- Todo endpoint nuevo debe pasar por el decorador de autorización.
- Las migraciones deben ser compatibles con la versión anterior del código.

## Formato
- Un comentario por problema, con el archivo y la línea.
- Indica la severidad: bloqueante, importante o sugerencia.
- Si no estás seguro, di que no lo estás en lugar de afirmar.
```

Dos riesgos merecen vigilancia. El primero es el ruido: si el bot deja veinte comentarios triviales por PR, el equipo aprende a ignorarlo y se pierden los dos que importaban. Ajusta las instrucciones hasta que la mayoría de sus comentarios sean accionables. El segundo es la complacencia: cuando una herramienta ya ha mirado el código, es tentador revisar con menos cuidado. La regla práctica es que el revisor humano forme su opinión antes de leer los comentarios del bot, no después.

Plan de adopción en 30 días

Cambiar la cultura de revisión de un equipo no se hace con un anuncio. Funciona mejor de forma gradual y medible:

1. Semana 1: medir. Ejecuta los scripts de tamaño de PR y tiempo hasta la primera revisión sobre los últimos 100 PRs. Sin una línea base, cualquier mejora será una impresión.
2. Semana 2: automatizar lo mecánico. Instala pre-commit y un linter en CI. Cada discusión de estilo que desaparece libera tiempo de revisión.
3. Semana 3: acordar la política. En una reunión de treinta minutos, el equipo fija el objetivo de tiempo de primera respuesta, el tamaño máximo recomendado, y el uso de prefijos en los comentarios. Escribe el resultado en una página del repositorio.
4. Semana 4: activar las reglas y revisar. Pasa el ruleset de modo evaluación a activo y repite la medición. Comparte los números con el equipo, sin señalar a nadie.

Preguntas frecuentes

¿Cuántas personas deben aprobar un PR? Para la mayoría de los cambios, una aprobación basta. Exigir dos suele diluir la responsabilidad («ya lo mirará el otro») y duplicar la espera. Reserva la doble aprobación para rutas críticas: seguridad, pagos, infraestructura.

¿Debo revisar mis propios cambios antes de pedir revisión? Sí, siempre. Leer tu propio diff en la interfaz del PR, como si fueras otra persona, encuentra una cantidad sorprendente de restos de depuración, comentarios obsoletos y cambios accidentales. Es la revisión más barata que existe.

¿Es mejor fusionar con squash, merge o rebase? Depende del equipo, pero el squash da un historial lineal con un commit por PR, lo que facilita los reverts. Asegúrate de que el mensaje del commit final conserve el porqué del cambio.

¿Qué hago con un PR que lleva días sin revisión? Primero, comprueba si es revisable: ¿tamaño, descripción, CI en verde? Después, pide la revisión a una persona concreta, no al canal general. Y si el patrón se repite, mira el p90 de espera: probablemente es un problema de reparto de carga.

¿La revisión de código sustituye a los tests? No. La revisión detecta problemas de diseño y de intención que los tests no ven; los tests detectan regresiones que ningún humano puede comprobar en cada cambio. Se complementan.

Glosario

CODEOWNERS: archivo que asigna revisores automáticos según la ruta modificada. Ruleset: conjunto de reglas de GitHub aplicadas a ramas o etiquetas. Aprobación obsoleta: aprobación dada a una versión anterior del PR, descartada al subir nuevos commits. Merge queue: cola que prueba cada PR contra el estado real de la rama principal antes de integrarlo. PR apilado: serie de PRs dependientes, donde cada uno parte de la rama del anterior. Expandir y contraer: patrón de migración en fases que mantiene la compatibilidad entre versiones. Flag de funcionalidad: interruptor que permite fusionar código sin activarlo. Percentil 90: valor por debajo del cual cae el 90 % de las observaciones; muestra la cola de casos lentos que la mediana oculta.

Conclusión

La revisión de código funciona cuando se trata como una conversación sobre el producto y no como una inspección del autor. Pequeña en tamaño, clara en contexto, automatizada en lo mecánico, explícita en la severidad y rápida en la respuesta: con esos cinco ingredientes, el tiempo de revisión deja de ser un coste y pasa a ser la inversión con mejor retorno del ciclo de desarrollo. Empieza por lo más barato: añade la plantilla de PR y los prefijos de comentario esta semana, mide el tiempo hasta la primera revisión durante un mes y ajusta a partir de los datos.

Referencias: The Standard of Code Review y Speed of Code Reviews, en las Google Engineering Practices (google.github.io/eng-practices); documentación de GitHub sobre CODEOWNERS y reglas de protección de ramas; documentación de pre-commit y de ruff.

Effective Code Review: Small PRs, Clear Comments and Automation That Respects Everyone's Time

English version - PuigFlows

A well-run code review is probably the most powerful inexpensive tool a team has for keeping quality high, spreading knowledge and preventing incidents. Done badly, it becomes a bottleneck that frustrates everyone: PRs that sit for days, comments about commas while a design flaw slips through, and authors who learn to defend themselves instead of improving. This guide collects what actually works: what to look at and in what order, how to keep changes small, how to write comments people can accept without a fight, and what to automate so humans can spend their attention on what only humans can judge.

The principle that orders everything: improve code health

Google's public engineering practices state the standard very clearly: a reviewer should approve a change as soon as it definitely improves the overall health of the codebase, even if it is not perfect. There is no such thing as perfect code, only better code. That idea defuses two opposite pathologies. The first is reviewer perfectionism, where a PR is blocked over personal preference. The second is laxity, where anything gets approved because "it works" and the system degrades one reasonable PR at a time.

A practical rule follows from it: style preferences that are not in the team guide are opinions, not defects, and should be labeled as such. Design principles, on the other hand, are not a matter of taste: if the change adds needless complexity or couples two modules that should stay independent, that is worth discussing even if the author considers the work finished.

What to review, and in what order

Order matters because a reviewer's effort is limited and the most expensive problems are the ones found last. A good sequence, from the most structural to the most superficial:

1. Design. Does the change make sense? Does it live in the right place? Does it fit the rest of the system, or does it open a parallel path someone will have to maintain? If the design is wrong, nothing else deserves comments yet.
2. Functionality. Does it do what it claims? Think about edge cases: empty lists, nulls, concurrency, retries, time zones, hostile input.
3. Complexity. Could someone arriving for the first time in six months understand it? Long functions, premature abstractions and cascading boolean flags are warning signs.
4. Tests. Would they fail if the code broke? A test that passes with the bug present protects nothing. Check that they cover the new case and not just the happy path.
5. Names and comments. Names should communicate intent; comments should explain why, not repeat what.
6. Style and formatting. There should be no human work here: tools handle it, as we will see below.

A good habit is to read the description and the tests first, and the production code afterwards. The tests tell you what the author believes they built; if they do not match the description, you already have your first question.

Change size decides review quality

There is an uncomfortable truth here: a person's attention degrades quickly with volume. A 50-line PR gets useful comments; a 1,500-line one gets a tired "LGTM". The classic SmartBear/Cisco review studies pointed the same way: defect detection drops when too many lines are reviewed at once or too quickly. You do not need exact figures to act on this: if you cannot explain the change in a couple of sentences, it is probably two changes.

Techniques that keep PRs small:

- Separate refactoring from behavior. First a PR that moves or renames without changing logic (easy to review because no test should change), then the one that adds the feature.
- Ship behind a feature flag so unfinished work can land on main without being exposed.
- Stack changes (stacked PRs) when one depends on another, instead of piling everything onto a three-week branch.
- Keep migrations separate from the logic that uses them.

To make size visible rather than a matter of opinion, measure it. This script counts changed lines against the base branch and ignores generated files that only add noise:

```
#!/usr/bin/env bash
# pr-size.sh - measures the real size of a change, ignoring generated files
set -euo pipefail
BASE="${1:-origin/main}"

git diff --numstat "$BASE"...HEAD -- . \
':(exclude)*.lock' ':(exclude)package-lock.json' ':(exclude)*.snap' ':(exclude)*.min.js' |
awk '{ add += $1; del += $2; files++ }
END { printf "files=%d added=%d deleted=%d total=%d\n", files, add, del, add + del }'
```

And this GitHub Action warns on the PR itself when it crosses a threshold. Note that it emits a warning rather than an error: the goal is to open a conversation, not to block. Some large changes are legitimate (a mechanical migration, a dependency bump) and the team should be able to justify them.

```
name: pr-size
on:
  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  size:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - name: Measure PR size
        env:
          BASE: origin/${{ github.base_ref }}
        run: |
          total=$(git diff --numstat "$BASE"...HEAD -- . ':(exclude)*.lock' ':(exclude)package-lock.json' \
          | awk '{ s += $1 + $2 } END { print s + 0 }')
          echo "changed lines: $total"
          if [ "$total" -gt 800 ]; then
            echo "::warning::This PR is over 800 lines. Consider splitting it into smaller changes."
          fi
```

The description is half the PR

A reviewer who opens a PR without context has to reconstruct the intent from the diff, which is expensive and error-prone. A good description answers why, not what (the diff already shows what), points to where the risk is, and says how to test the change. A template at [.github/pull_request_template.md](#) makes sure it always gets filled in:

```
## What changes and why
<!-- The problem this solves, in two or three sentences. Link the ticket. -->

## How it is done
<!-- Design decisions and alternatives you discarded. -->

## Risk and rollout
- [ ] Safe to roll back
- [ ] Includes a database migration (attach the plan)
- [ ] Behind a feature flag

## How to test it
<!-- Commands, screenshots, or the exact query that validates the change. -->
```

```
## Where I want your attention
<!-- The parts I am unsure about. The rest is mechanical. -->
```

The last section is the most valuable. Saying "I am unsure here, not here" directs the reviewer's attention to where you need it most and turns the review into a collaboration rather than an exam.

Automate the mechanical so people can judge what matters

Every comment about indentation, unsorted imports or an unused variable is wasted human time and creates needless personal friction: a tool can be inflexible without being unfriendly. Anything a machine can decide should be decided by a machine, preferably before the PR even exists.

Locally: pre-commit

```
# .pre-commit-config.yaml
repos:
- repo: https://github.com/astral-sh/ruff-pre-commit
  rev: v0.8.0
  hooks:
    - id: ruff
      args: [--fix]
    - id: ruff-format
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v5.0.0
  hooks:
    - id: check-merge-conflict
    - id: detect-private-key
    - id: end-of-file-fixer
```

Hooks are installed once with pre-commit install and keep formatting noise out of the PR. Repeat the same checks in CI (with pre-commit run --all-files), because a local hook can always be skipped.

In the repository: CODEOWNERS and branch protection

The CODEOWNERS file makes GitHub automatically request review from the people or teams responsible for each path. Combined with a branch protection rule requiring code owner approval, it guarantees that changes in sensitive areas (payments, infrastructure, migrations) are seen by someone who knows the area:

```
# .github/CODEOWNERS
# The last matching rule wins. Put general rules first and specific ones below.
*                                @acme/engineering
/infra/                          @acme/platform
/db/migrations/                  @acme/data @acme/platform
/src/payments/                   @acme/payments
*.sql                            @acme/data
```

Beware of overdoing it: if everything requires five owners, review turns into a formality and waiting time explodes. Reserve mandatory approvals for what is genuinely risky and trust the rest.

In CI: tests, static analysis and security

Having tests, the linter, the type checker and a dependency scan run before anyone looks at the PR changes the dynamic: the reviewer knows what arrives already compiles, passes the tests and meets the formatting rules. Mark these checks as required in the protection rules for main.

How to write comments people can accept

The same finding can lead to an improvement or a fight depending on how it is phrased. Some guidelines that work:

- Comment on the code, not the person. "This function does two things" instead of "you are doing too much here".
- Explain why. "This concatenates the query with user input, which allows SQL injection" teaches; "change it" does not.

- Ask when you are unsure. "What happens if the coupon has expired?" leaves room for the author to know something you do not.
- State the severity. An explicit prefix saves the author from guessing whether a comment blocks approval.
- Acknowledge what is good. "This split came out really clean" costs ten seconds and changes the tone of the whole review.

A simple prefix convention a team can adopt:

- blocking: must be resolved before merging (a bug, a security risk, a broken contract).
- suggestion: a recommended improvement; the author decides.
- question: I need to understand something; no change implied.
- nit: minor and optional. If there are many, a linter rule is probably missing.

A complete example

Let us look at a realistic function and how it would be reviewed. This is the code that arrives in the PR:

```
def apply_discount(order, code):
    coupon = db.query("SELECT * FROM coupons WHERE code = '%s'" % code).first()
    if coupon.percent > 0:
        order.total = order.total - order.total * coupon.percent / 100
    order.save()
    return order
```

A good reviewer does not start with style. They start with what can cost money or security:

- blocking: the query is built with string interpolation, which allows SQL injection if code comes from the user. Use parameters or the project's data access layer.
- blocking: if the coupon does not exist, `.first()` returns `None` and the next line raises `AttributeError`, which the client will see as a 500 instead of "invalid coupon".
- question: is there a check anywhere that the coupon has not expired or run out of uses?
- suggestion: is the total a float or a decimal? If it is a float, percentage arithmetic will accumulate rounding errors; for money use `Decimal`.
- suggestion: the discount can push the total negative if the percentage exceeds 100; it should be capped.
- nit: `save(update_fields=[...])` avoids overwriting fields changed by a concurrent request.

And a version that resolves the blocking points and the suggestions:

```
def apply_discount(order: Order, code: str) -> Order:
    coupon = coupons.get_active(code) # the repository filters expired coupons and uses parameters, never interpolation
    if coupon is None:
        raise InvalidCoupon(code)

    discount = (order.total * coupon.percent / 100).quantize(Decimal("0.01"))
    order.total = max(order.total - discount, Decimal("0"))
    order.save(update_fields=["total"])
    return order
```

Notice that only two comments in the whole review are blocking. The rest are reasoned improvements the author can accept or discuss. That balance is what keeps review productive.

Speed is quality too

A PR that waits two days forces the author to switch context, increases merge conflicts and discourages small changes, which is exactly the opposite of what we want. Google's guidelines suggest that, barring exceptional cases, a response to a review request should arrive within one business day at most; many teams set a more ambitious target of a few hours. What matters is agreeing on an explicit commitment and measuring it.

This script queries the GitHub API and computes the median and 90th percentile time to first review for

recently merged PRs. It is a starting point: on a large repository you will need to paginate and handle rate limits.

```
import os
import statistics
from datetime import datetime

import requests

REPO = os.environ["REPO"] # "acme/api"
HEADERS = {
    "Authorization": f"Bearer {os.environ['GITHUB_TOKEN']}",
    "Accept": "application/vnd.github+json",
}
API = f"https://api.github.com/repos/{REPO}"

def ts(value: str) -> datetime:
    return datetime.fromisoformat(value.replace("Z", "+00:00"))

def first_review_hours(pr: dict) -> float | None:
    reviews = requests.get(f"{API}/pulls/{pr['number']}/reviews", headers=HEADERS, timeout=30).json()
    submitted = [ts(r["submitted_at"]) for r in reviews
                  if r.get("submitted_at") and r["user"]["login"] != pr["user"]["login"]]
    if not submitted:
        return None
    return (min(submitted) - ts(pr["created_at"])).total_seconds() / 3600

prs = requests.get(f"{API}/pulls", headers=HEADERS, timeout=30,
                  params={"state": "closed", "per_page": 50, "sort": "updated", "direction": "desc"}).js
    on()
merged = [p for p in prs if p.get("merged_at")]
waits = [h for p in merged if (h := first_review_hours(p)) is not None]

print(f"PRs: {len(waits)}")
print(f"Median time to first review: {statistics.median(waits):.1f} h")
print(f"p90: {statistics.quantiles(waits, n=10)[-1]:.1f} h")
```

Look at the median for normal behavior and the 90th percentile to catch PRs that get forgotten. If the second number is much larger, there is an assignment problem (the same expert is always requested) or a visibility problem. Do not use the metric to evaluate individuals: you will measure what looks good instead of what is good.

Common mistakes

- Reviewing only the diff. Some defects show up when you open the whole file or search for who else calls the modified function.
- Approving without reading out of trust in the author. Review is not an audit of people; it is a safety net.
- Requesting rewrites out of personal taste. If there is no principle behind it, it is a nit.
- Trickling comments across successive rounds. Do a complete review the first time, so the author does not get new objections every time they push.
- Long debates in comment threads. If two exchanges do not settle it, call or talk; then leave a written summary on the PR.
- A single reviewer for everything. That creates a bottleneck and concentrates knowledge. Rotate.
- Ignoring the human side. Behind the screen is a person who spent hours on their work. Clarity and kindness are not incompatible.

Reviewing AI-generated changes

Code assistants now produce a growing share of changes, and that does not relax review, it tightens it. Generated code tends to be plausible and neatly formatted, which makes it harder to question. A few

precautions: require the author to understand and be able to explain every line they submit; verify that imported dependencies exist and are the intended ones; distrust tests that look like they cover a lot but assert nothing; and pay extra attention to error handling, which is where generated code tends to stay generic. Responsibility for the change belongs to whoever opens the PR, wherever the code came from.

Reviewer checklist

- I read the description and understand the goal of the change.
- The design is appropriate and lives in the right place.
- I thought about edge cases and failures (nulls, empties, concurrency, retries).
- There are tests that would fail if the code broke.
- No secrets, no personal data in logs, no queries built from user input.
- Names communicate intent and comments explain why.
- My comments are labeled by severity and justified.
- I responded within the agreed time.

Author checklist

- The change does one thing and I can explain it in two sentences.
- The description explains why, the risk and how to test it.
- I reviewed my own diff before requesting review.
- Tests and linter pass locally and in CI.
- I flagged where I want specific attention.
- I answer every comment: applied, discussed, or deferred with a ticket.

Reviewing in three passes: a method to avoid getting lost

Anyone who reviews a PR by reading the diff top to bottom, file by file, in the alphabetical order the interface imposes, does the worst possible version of the job: they reach the important file with their attention spent and judge details before understanding the intent. A more reliable method splits the review into three passes with different goals, each with a different attention cost.

First pass: orientation (two or three minutes). You do not look at the code yet. You read the title, the description, the linked ticket and the list of touched files. The question is: what problem does this solve, and what shape would I expect the solution to have? If the PR touches fifteen files and the description talks about a text change on a button, there is already a discrepancy worth a question before you read a single line. This pass also decides whether the PR is reviewable: if it is not (too big, no description, three changes mixed together), the cheapest thing for everyone is to send it back with a concrete request to split it, not to review it halfway.

Second pass: structure (five to ten minutes). Read the tests first, then the files with the most responsibility: public interfaces, schemas, migrations, API contracts. Look for whether the design is right, whether the logic lives in the correct layer, whether it duplicates something that already exists and whether the edge cases are covered. This is where you write the highest-impact comments, the ones that change the shape of the code. It pays to write them before moving on to details, because if the author has to rework the structure, style comments on the old code will have been wasted time for both.

Third pass: detail (as long as it takes, with a limit). Names, error messages, null handling, concurrency, logs, performance in hot loops, security. It is the pass that most resembles what people imagine when they think of a review, and the one that benefits most from prior automation: if the linter already settled formatting, this pass can be spent on what only a human sees.

So that reading order does not depend on the interface, this small Python script sorts the files of a PR by

review priority: tests first, then contracts and migrations, then the rest by amount of change. It is a reading aid, not a rule.

```
#!/usr/bin/env python3
"""review_order.py - list a PR's files in the order they are best read.

Usage: gh pr diff 1234 --name-only | python3 review_order.py
"""
import re
import subprocess
import sys

# The lower the number, the earlier it is read.
PRIORITIES = [
    (0, re.compile(r"^(^|/)(tests?|__tests__)/|_test\.|\..test\.|\..spec\.")),
    (1, re.compile(r"migrations?/|schema|\..proto$|openapi|\..sql$")),
    (2, re.compile(r"^(^|/)(api|routes?|handlers?|contracts?)/")),
    (3, re.compile(r"Dockerfile|\..ya?ml$|terraform/|\..tf$")),
]

def priority(path: str) -> int:
    for level, pattern in PRIORITIES:
        if pattern.search(path):
            return level
    return 4

def changes(path: str, base: str) -> int:
    out = subprocess.run(
        ["git", "diff", "--numstat", base, "--", path],
        capture_output=True, text=True, check=False,
    ).stdout.split()
    try:
        return int(out[0]) + int(out[1])
    except (IndexError, ValueError):
        return 0

def main() -> None:
    base = sys.argv[1] if len(sys.argv) > 1 else "origin/main"
    paths = [l.strip() for l in sys.stdin if l.strip()]
    ordered = sorted(paths, key=lambda p: (priority(p), -changes(p, base)))
    for p in ordered:
        print(f"[{priority(p)}] {p}")

if __name__ == "__main__":
    main()
```

The value of the script is not in the code but in the decision it fixes: tests are read first because they are the executable specification of the change. If the tests do not tell the same story as the description, you already know where to ask.

Security and reliability review: three bugs that slip in out of habit

The bugs that most often escape a review are not the exotic ones. They are the ones that look fine in the diff because the code is well written and plausible. These three keep showing up, and it is worth training your eye for them.

1. The authorization that is missing, not the one that is extra

A diff shows what is added, and the absence of a check appears nowhere. Look at this endpoint, which seems correct:

```
@router.get("/invoices/{invoice_id}")
async def get_invoice(invoice_id: int, user: User = Depends(current_user)):
    invoice = await repo.get(invoice_id)
    if invoice is None:
        raise HTTPException(404)
    return invoice
```

It authenticates the user but does not check that the invoice belongs to them: anyone with a session can read someone else's invoices by changing the number. The review question that catches it is always the same:

what stops user A from accessing user B's resource? The corrected version filters by owner in the query itself, so that another user's resource is indistinguishable from a nonexistent one:

```
invoice = await repo.get(invoice_id, owner_id=user.id)
if invoice is None:
    raise HTTPException(404) # same error for "does not exist" and "is not yours"
```

2. The race condition disguised as correct logic

Reading, deciding and writing in separate steps works in the test and fails under load:

```
balance = await accounts.balance(account_id)
if balance >= amount:
    await accounts.set_balance(account_id, balance - amount)
```

Two simultaneous requests read the same balance and both deduct. In review, the signal is a read followed by a write on the same data with no lock or atomic condition. The fix moves the condition into the database:

```
UPDATE accounts
  SET balance = balance - :amount
 WHERE id = :account_id AND balance >= :amount
RETURNING balance;
-- 0 rows returned = insufficient funds, with no race window
```

3. The swallowed error

```
try:
    await send_confirmation(order)
except Exception:
    pass
```

A broad except with pass turns a failure into silence. In review, every exception catch should answer three questions: which specific failure is expected, what is done about it, and where it is recorded. If the answer is "none", the catch is probably unnecessary or the log is missing. A defensible version catches only what is expected, logs with context and explicitly decides whether the failure should propagate:

```
try:
    await send_confirmation(order)
except TemporaryMailError as exc:
    logger.warning("confirmation deferred", extra={"order": order.id, "error": str(exc)})
    await retry_queue.enqueue(order.id)
```

These three examples share a lesson: the reviewer adds more value by asking about what is not in the diff (the missing check, atomicity, the failure path) than by commenting on what is.

Branch rules as code: rulesets, stale approvals and the merge queue

A review policy that lives only in the team's head lasts until the first rushed Friday. The important rules are best written where the platform enforces them. On GitHub, rulesets let you define those rules per branch or branch pattern, version them and apply them across several repositories. These are the rules that pay off most, and why:

- Require a pull request before merging, with at least one approval. It is the foundation: without it, everything else is optional.
- Dismiss stale approvals when new commits are pushed. Without this rule, someone approves a twenty-line PR and the author later adds three hundred that nobody looks at. The approval must refer to the code being merged, not to an earlier version.
- Require approval of the most recent reviewable push by someone other than the person who made it. It prevents the author from pushing a last-minute change and self-approving on the strength of someone else's earlier approval.
- Require review from code owners (CODEOWNERS) on sensitive paths: authentication, payments, infrastructure, migrations.

- Require all conversations to be resolved before merging, so a blocking comment does not get buried.
- Require status checks (tests, linter, security analysis) and an up-to-date branch, or use the merge queue, which tests each PR against the real state of the main branch before integrating it.

These rules can be declared as code and applied with the GitHub CLI. This example creates a ruleset for the main branch; adjust the check names to those of your CI:

```
#!/usr/bin/env bash
# create-ruleset.sh - protect the default branch with review rules
set -euo pipefail
REPO="$1" # usage: create-ruleset.sh owner/repo

gh api --method POST "repos/$REPO/rulesets" --input - <<'JSON'
{
  "name": "mandatory-review",
  "target": "branch",
  "enforcement": "active",
  "conditions": { "ref_name": { "include": ["~DEFAULT_BRANCH"], "exclude": [] } },
  "rules": [
    { "type": "pull_request",
      "parameters": {
        "required_approving_review_count": 1,
        "dismiss_stale_reviews_on_push": true,
        "require_code_owner_review": true,
        "require_last_push_approval": true,
        "required_review_thread_resolution": true
      } },
    { "type": "required_status_checks",
      "parameters": {
        "strict_required_status_checks_policy": true,
        "required_status_checks": [ { "context": "ci / test" }, { "context": "ci / lint" } ]
      } },
    { "type": "non_fast_forward" }
  ]
}
JSON
```

A practical detail: start with "enforcement": "evaluate" (evaluate mode) for a week. It records which PRs would have been blocked without blocking anyone, and lets you tune the rules before they cause real friction. Switching to active afterwards is a one-line diff.

A rule that is often forgotten: the bypass

Every rule needs an escape hatch for emergencies, but that hatch must be narrow and auditable. Limit the bypass to a specific role (for example, the on-call team) and require that using it leaves a trace: a follow-up PR, a note in the incident channel. If half the team can skip review "just this once", the rule is decorative.

A well-designed CODEOWNERS: few owners, clear paths

The CODEOWNERS file assigns automatic reviewers by path. The usual failure is excess: if the whole repository has three owners, every PR notifies all of them, and the notification stops meaning anything. A healthy design has three traits. First, the general rule assigns a team, not individuals, so someone's vacation blocks nothing. Second, sensitive paths (security, payments, infrastructure) have specific owners with real knowledge. Third, the file is reviewed like any other: a path with no owner, or with an owner who left the company, is silent debt.

```
# .github/CODEOWNERS
# General rule: any PR needs at least the platform team
*                               @acme/platform

# Sensitive paths: owners with specific knowledge
/src/auth/                     @acme/security
/src/payments/                 @acme/payments @acme/security
/db/migrations/                @acme/data
/infra/                        @acme/sre
/.github/workflows/            @acme/sre @acme/security

# CODEOWNERS itself is reviewed by whoever governs it
```

In CODEOWNERS the last matching rule wins, so the most specific ones go at the end. A common mistake is putting the general `*` rule last and unintentionally overriding all the previous ones.

Measuring review with real data: time to first response

A useful metric for a healthy review process is the time to first human response, because it is what the author experiences as "blocked". The following script queries GitHub's GraphQL API and computes the median and 90th percentile for the latest merged PRs. It excludes bots so as not to skew the result.

```
#!/usr/bin/env python3
"""first_review_time.py - median and p90 wait until the first human review."""
import json
import statistics
import subprocess
import sys
from datetime import datetime

QUERY = """
query($owner:String!, $repo:String!) {
  repository(owner:$owner, name:$repo) {
    pullRequests(last: 100, states: MERGED) {
      nodes {
        createdAt
        author { login }
        reviews(first: 20) { nodes { submittedAt author { __typename login } } }
      }
    }
  }
}
"""

def parse(ts: str) -> datetime:
    return datetime.fromisoformat(ts.replace("Z", "+00:00"))

def main(owner: str, repo: str) -> None:
    out = subprocess.run(
        ["gh", "api", "graphql", "-f", f"query={QUERY}", "-F", f"owner={owner}", "-F", f"repo={repo}"],
        capture_output=True, text=True, check=True,
    ).stdout
    prs = json.loads(out)["data"]["repository"]["pullRequests"]["nodes"]
    waits = []
    for pr in prs:
        author = (pr["author"] or {}).get("login")
        human = [
            parse(r["submittedAt"]) for r in pr["reviews"]["nodes"]
            if r["author"] and r["author"]["__typename"] == "User" and r["author"]["login"] != author
        ]
        if human:
            waits.append((min(human) - parse(pr["createdAt"])).total_seconds() / 3600)
    if len(waits) < 10:
        sys.exit("Sample too small to draw conclusions")
    waits.sort()
    p90 = waits[int(len(waits) * 0.9) - 1]
    print(f"PRs analyzed: {len(waits)}")
    print(f"Median: {statistics.median(waits):.1f} h | p90: {p90:.1f} h")

if __name__ == "__main__":
    main(sys.argv[1], sys.argv[2])
```

Look at the p90 before the median. The median often looks fine even on teams where one in ten PRs waits three days, and those are the ones that cause frustration and stale branches. If the p90 is several times the median, the problem is load distribution or PRs nobody wants to review, and that is fixed with review rotations, not by demanding more speed.

Case study: splitting a 1,200-line PR into four

Imagine a PR titled "Volume discount support". It touches 31 files and adds up to 1,200 lines: a migration that

adds the tier table, a refactor of price calculation that moves functions around, the new logic, the endpoint, and formatting changes the editor applied along the way. Nobody is going to review it well. The question is how to split it without losing coherence, and the answer is to think about what can be merged and deployed without changing observable behavior.

1. PR 1, pure refactor (~300 lines). Moves price calculation into its own module without changing anything. Existing tests must pass unmodified; that is the proof it is a refactor. It is reviewed in ten minutes because the reviewer only checks that nothing changed.
2. PR 2, additive migration (~80 lines). Creates the tier table, empty. No code uses it yet. It deploys safely because adding a table breaks nothing.
3. PR 3, logic behind a feature flag (~500 lines). Implements discount calculation and enables it only if the `volume_discounts` flag is on. Nothing changes in production until it is switched on, so the merge risk is minimal and the reviewer can focus on the correctness of the logic and its tests.
4. PR 4, exposure (~200 lines). Endpoint and documentation. With the rest already in the main branch, this PR is small and readable.

Notice that each PR has a single reason to exist and can be reverted independently. The cost is a bit more coordination; the benefit is that the four reviews, added together, are faster and find more problems than the single 1,200-line review.

Stacked PRs: when changes depend on one another

If PR 3 depends on PR 2 and you do not want to wait for it to merge, you can stack them: each branch starts from the previous one and each PR targets its predecessor's branch. The flow with plain git is simple but has a delicate point: when the first one merges, the others must be re-based onto the main branch.

```
# Stack: main <- refactor-pricing <- tiers-migration <- discount-logic
git switch -c refactor-pricing main
# ...commits...
git switch -c tiers-migration refactor-pricing
# ...commits...
git switch -c discount-logic tiers-migration
# ...commits...

# After merging refactor-pricing into main (squash), re-base the rest of the stack:
git fetch origin
git rebase --onto origin/main refactor-pricing tiers-migration
git rebase --onto tiers-migration refactor-pricing discount-logic # if adjustment is needed
git push --force-with-lease origin tiers-migration discount-logic
```

Using `--onto` is what avoids dragging along the commits that were already merged. And `--force-with-lease`, instead of `--force`, fails if someone else pushed to that branch, which protects other people's work. Stack tools such as Graphite, git-branchless or gh stack automate these steps, but it is worth understanding what they do underneath before delegating it.

Reviewing database migrations: the change a revert cannot undo

A code revert reverts code. It does not revert the data a migration already transformed, nor the lock that left a table inaccessible for ten minutes. That is why migrations deserve a specific review with these questions:

- Is it compatible with the previous version of the code? During a rolling deployment two versions of the service coexist against the same database. If the migration drops a column the old code still reads, the deployment breaks halfway.
- What lock does it take, and for how long? Adding a column with a default, creating an index or adding a constraint can block writes on a large table. The concurrent or phased variant must be used.
- Is there a way back? You do not need a reverse migration for everything, but you do need to know what you would do on failure.
- Was it tested at a realistic volume? A migration that takes two seconds with a thousand rows can take an hour with a hundred million.

The pattern that enables safe changes is called expand/contract. To rename a column without stopping the service, spread it over several deployments:

```
-- Deployment 1 (expand): add the new column without touching the old one
ALTER TABLE customers ADD COLUMN full_name text;

-- Deployment 2: code writes to both columns and reads from the old one.
-- Backfill in batches, outside the migration, to avoid holding a long lock:
UPDATE customers
  SET full_name = name
 WHERE id BETWEEN 1 AND 50000 AND full_name IS NULL;
-- ...repeat in batches until done

-- Deployment 3: code reads from the new column (the old one is still written)

-- Deployment 4 (contract): once no deployed version uses the old one
ALTER TABLE customers DROP COLUMN name;

-- Indexes on large tables: without blocking writes
CREATE INDEX CONCURRENTLY idx_customers_full_name ON customers (full_name);
```

A reviewer who knows this pattern instantly recognizes the PR that does RENAME COLUMN in a single step and knows what to ask for: splitting it into phases. It is one of the highest-return comments they can leave.

Reviewing tests: they should prove behavior, not just turn green

A passing test does not prove the code is correct; it proves the code does what the test checks. When reviewing tests, the useful question is what would fail if the code were broken. These are the signs of a weak test:

- Assertions that cannot fail, such as checking that the result "is not null" when the function never returns null.
- Mocks so extensive that the test verifies the mock and not the code.
- A single happy path, with no edge cases: empty list, negative value, input with Unicode characters, time zone.
- Tests that replicate the implementation (they compute the expected value with the same formula as the code), so a formula error goes unnoticed.
- Random data without a fixed seed, which produces intermittent failures that cannot be reproduced.

A little-used review trick is mental mutation: pick a line of logic, imagine changing it (a $\geq$ to a $>$) and ask which test would catch it. If none, there is a gap. Mutation testing tools such as mutmut in Python or Stryker in JavaScript automate this idea, and although it is not worth running them on every PR because of their cost, a periodic analysis of critical modules reveals where coverage is only apparent.

```
def apply_discount(total: Decimal, units: int) -> Decimal:
    if units >= 100:
        return total * Decimal("0.85")
    if units >= 10:
        return total * Decimal("0.95")
    return total

# Weak test: only checks one middle point of each tier.
def test_discount_weak():
    assert apply_discount(Decimal("100"), 50) == Decimal("95")

# Strong test: attacks the boundaries, where off-by-one errors live.
@pytest.mark.parametrize("units,expected", [
    (9, Decimal("100")), # just before the first tier
    (10, Decimal("95")), # first value of the tier
    (99, Decimal("95")), # last value of the tier
    (100, Decimal("85")), # first value of the upper tier
    (0, Decimal("100")), # degenerate case
])
def test_discount_at_boundaries(units, expected):
    assert apply_discount(Decimal("100"), units) == expected
```

If you changed `>= 10` to `> 10`, the weak test would stay green and the strong one would fail on the ten-unit case. That difference is what an attentive reviewer can point out in one sentence: "what happens exactly at 10 and at 100?".

A complete, annotated review thread

Theory settles better with a case. This is an excerpt of a review on a PR that adds retries to a payment gateway call. The original code:

```
def charge(order):
    for attempt in range(5):
        try:
            return gateway.charge(order.amount, order.card)
        except Exception:
            time.sleep(1)
    return None
```

A low-quality review would say "this loop is ugly" or "use a retry library". A good review locates the real problems, orders them by severity and proposes a concrete way out:

blocking: If the gateway processed the charge but the response was lost (a timeout), the retry will charge the customer twice. We need a stable idempotency key per order; most gateways accept one in a header. Without it, these retries are not safe.

blocking: except Exception also retries permanent errors (declined card, invalid amount). Five attempts against a definitive rejection only delay the error and can trigger the bank's fraud systems. We should retry only on network errors and 5xx responses.

important: Returning None when attempts run out forces the caller to guess what happened. Better to raise a dedicated exception with the last error as its cause.

suggestion: A fixed one-second wait under load synchronizes all the clients that retry. Exponential backoff with jitter would avoid that. It does not block this PR.

question: Is there a test that simulates a timeout after a successful charge? It is the most dangerous scenario and the least tested.

The resulting version after review:

```
import random
import time

class ChargeFailed(Exception):
    """Retries were exhausted or the error was not retryable."""

RETRYABLE_ERRORS = (TimeoutError, ConnectionError, Server5xxError)

def charge(order, max_attempts: int = 4):
    key = f"charge-{order.id}" # stable across retries: the gateway deduplicates
    last_error = None
    for attempt in range(max_attempts):
        try:
            return gateway.charge(
                order.amount, order.card, idempotency_key=key,
            )
        except RETRYABLE_ERRORS as exc:
            last_error = exc
            wait = min(8, 2 ** attempt) * random.uniform(0.5, 1.0)
            logger.warning("retryable charge", extra={"order": order.id, "attempt": attempt + 1})
            time.sleep(wait)
        except CardDeclined:
            raise # permanent: retrying does not help
    raise ChargeFailed(f"order {order.id}") from last_error
```

Notice what made this review work: each comment explains the consequence, not just the rule; the two blocking comments are clearly distinguished from the suggestion; and the final question asks for a test, not an opinion. The author knows what must change to merge and what can wait.

Reviewing performance without falling into premature optimization

"This could be faster" is a cheap and almost always useless comment. For a performance comment to be worthwhile it must meet one of two conditions: point to a complexity that scales badly with real data (a query inside a loop, a linear search over a list that keeps growing), or rest on a measurement. The signs that justify a comment are few and recognizable: database queries or network calls inside loops, loading entire collections into memory when an iterator would do, missing pagination on endpoints that return lists, and missing indexes for the new queries. In every other case, the rule is to ask for data before rewrites: "have you measured how long it takes with ten thousand rows?" is a question that guides; "I would do it with a dictionary" is an opinion.

Reviewing dependencies, configuration and infrastructure as code

Some changes have almost no lines but a lot of risk. A dependency update is two lines in a lock file; a CI workflow change is a handful of YAML. Reviewing these changes calls for a different eye.

Dependencies. Before approving a PR that adds or updates a library, check three things: that it is necessary (could ten lines of your own solve it?), that it is maintained (date of the last release, open issues, number of maintainers) and that its license is compatible with your use. For updates, read the release notes looking for breaking changes. Automatic PRs from Dependabot or Renovate deserve the same attention as a human one: the fact that a bot opened it does not mean its content is harmless.

CI workflows. A workflow with broad permissions is an entry point. When reviewing, check that the permissions block grants the minimum, that third-party actions are pinned to a commit hash rather than a mutable tag, and that no step interpolates attacker-controlled data (such as a PR title) inside a script. That last flaw allows arbitrary code execution with the repository's secrets:

```
# Dangerous: the PR title is interpolated directly into the shell script
- name: Greet
  run: echo "New PR: ${ github.event.pull_request.title }"

# Safe: the value goes through an environment variable and is not interpreted as code
- name: Greet
  env:
    TITLE: ${ github.event.pull_request.title }
  run: echo "New PR: $TITLE"
```

Infrastructure as code. In Terraform or OpenTofu, the code diff is not enough: what matters is the plan. A one-line change can destroy and recreate a stateful resource, such as a database. Have the plan posted as a comment on the PR and review it looking for the words destroy and replace. Approving an infrastructure change without reading the plan is approving blind.

When you disagree: resolving a technical disagreement

Sooner or later author and reviewer disagree and neither gives way. Comment threads that stretch to ten messages are rarely resolved in writing. These rules keep a disagreement from becoming a conflict:

1. Distinguish preference from problem. If the reviewer cannot explain what breaks, what becomes harder to maintain or what risk is introduced, it is probably a preference, and should be marked as a non-blocking suggestion or withdrawn.
2. Cite the criterion, not the authority. "I prefer this" and "the senior does it this way" are not arguments. "This creates a circular dependency between two modules" is.
3. Move to a live conversation by the third round. Five minutes on a call resolves what twenty comments cannot. Afterwards, record what was agreed in the PR, for whoever reads it later.
4. Escalate to a third party, not up the hierarchy. Ask a third person with technical judgment to break the tie; if the disagreement is fundamental, it probably should become an ADR or a design discussion, not be settled inside a PR.
5. When the call is close, the author wins. If both options are reasonable and there is no clear risk, the rule in Google's guidelines is to let the author decide. Avoid blocking out of taste.

And a note on power dynamics: comments from someone more senior weigh more than their author believes. A "wouldn't it be better to..." from a lead can read as an order. Whoever holds authority should be especially explicit about whether a comment is blocking.

Review across time zones: not losing a day per comment

On distributed teams, each round trip on a PR can cost a day. If the author is in Madrid and the reviewer in Mexico City, a four-round conversation consumes a week. Some practices reduce it. Write complete comments: instead of "why this?", explain what worries you and propose an alternative, so the author can act without asking again. Approve conditionally when the requested changes are minor ("Approved, provided the two comments marked as blocking are resolved"), so the author can merge when done without waiting for another round. And set an overlap window for reviews that need conversation, instead of letting every PR depend on the chance coincidence of schedules.

AI-assisted review: a reasonable setup

Automated reviewers based on language models are useful for a first pass: they catch obvious errors, inconsistencies between comment and code, missing validations. They have clear limits: they do not know your product's history, why a decision was made two years ago, or what commitments you have made to a customer. A sensible setup treats them as a very fast junior reviewer: their comments are evaluated, not obeyed, and their approval never replaces a human one.

To make them useful rather than noisy, give them repository-specific instructions. Most tools accept an instructions file; this is an example of the kind of content that works:

```
# Instructions for the automated reviewer

## Priorities (in this order)
1. Correctness bugs and race conditions.
2. Security: missing authorization, injection, secrets in code.
3. Error handling: swallowed exceptions, unlogged failures.
4. Tests: uncovered edge cases.

## Ignore
- Formatting, import order and style (the linter handles those).
- Suggestions to rename variables unless they are misleading.

## Project conventions
- Monetary amounts use Decimal, never float.
- Every new endpoint must go through the authorization decorator.
- Migrations must be compatible with the previous version of the code.

## Format
- One comment per problem, with file and line.
- State the severity: blocking, important or suggestion.
- If you are unsure, say so instead of asserting.
```

Two risks deserve watching. The first is noise: if the bot leaves twenty trivial comments per PR, the team learns to ignore it and the two that mattered get lost. Tune the instructions until most of its comments are actionable. The second is complacency: once a tool has looked at the code, it is tempting to review with less care. The practical rule is for the human reviewer to form their opinion before reading the bot's comments, not after.

A 30-day adoption plan

Changing a team's review culture is not done with an announcement. It works best gradually and measurably:

1. Week 1: measure. Run the PR size and time-to-first-review scripts over the last 100 PRs. Without a baseline, any improvement is just an impression.
2. Week 2: automate the mechanical. Install pre-commit and a linter in CI. Every style discussion that disappears frees up review time.

3. Week 3: agree on the policy. In a thirty-minute meeting, the team sets the first-response time target, the recommended maximum size, and the use of comment prefixes. Write the result on a page in the repository.
4. Week 4: enable the rules and review. Move the ruleset from evaluate mode to active and repeat the measurement. Share the numbers with the team without singling anyone out.

Frequently asked questions

How many people should approve a PR? For most changes, one approval is enough. Requiring two tends to dilute responsibility ("the other one will look at it") and double the wait. Reserve double approval for critical paths: security, payments, infrastructure.

Should I review my own changes before asking for review? Yes, always. Reading your own diff in the PR interface, as if you were someone else, finds a surprising amount of debugging leftovers, stale comments and accidental changes. It is the cheapest review there is.

Is it better to merge with squash, merge commit or rebase? It depends on the team, but squash gives a linear history with one commit per PR, which makes reverts easier. Make sure the final commit message keeps the reason for the change.

What do I do with a PR that has gone days without review? First, check whether it is reviewable: size, description, CI green? Then ask a specific person for the review, not the general channel. And if the pattern repeats, look at the wait p90: it is probably a load distribution problem.

Does code review replace tests? No. Review catches design and intent problems that tests do not see; tests catch regressions that no human can check on every change. They complement each other.

Glossary

CODEOWNERS: file that assigns automatic reviewers based on the modified path. Ruleset: set of GitHub rules applied to branches or tags. Stale approval: approval given to an earlier version of the PR, dismissed when new commits are pushed. Merge queue: queue that tests each PR against the real state of the main branch before integrating it. Stacked PR: series of dependent PRs, where each starts from the previous one's branch. Expand/contract: phased migration pattern that preserves compatibility between versions. Feature flag: switch that lets you merge code without enabling it. 90th percentile: value below which 90% of observations fall; it shows the tail of slow cases that the median hides.

Conclusion

Code review works when it is treated as a conversation about the product rather than an inspection of the author. Small in size, clear in context, automated where mechanical, explicit about severity and quick to respond: with those five ingredients, review time stops being a cost and becomes the best-returning investment in the development cycle. Start with the cheapest steps: add the PR template and the comment prefixes this week, measure time to first review for a month, and adjust based on the data.

References: The Standard of Code Review and Speed of Code Reviews in Google Engineering Practices (google.github.io/eng-practices); GitHub documentation on CODEOWNERS and branch protection rules; pre-commit and ruff documentation.