PuigFlows

Connection Pooling en PostgreSQL: PgBouncer, Modos de Pooling y Tamaño Óptimo del Pool

Connection Pooling in PostgreSQL: PgBouncer, Pooling Modes and Optimal Pool Size

Guía · Intermedio · Proyectos Reales

Autor: Josue Puig · puigflows.com

Edición ampliada — Julio 2026 · Lectura: ~36 min por idioma

Documento bilingüe: Español (parte 1) · English (part 2)

RECURSO PREMIUM

PARTE 1 · ESPAÑOL

Connection Pooling en PostgreSQL: PgBouncer, Modos de Pooling y Tamaño Óptimo del Pool

Por qué las conexiones de PostgreSQL son caras

PostgreSQL usa un modelo de proceso por conexión: cada cliente que se conecta hace que el servidor lance un proceso backend dedicado, con su propia memoria y sus propias estructuras internas. Ese diseño es robusto, pero tiene un precio. Abrir una conexión implica un handshake TCP, autenticación y un fork de proceso; mantenerla abierta consume varios megabytes aunque esté ociosa; y con miles de backends el planificador del sistema operativo y los locks internos de PostgreSQL empiezan a sufrir.

Por eso `max_connections` no es un número que puedas subir alegremente. Un servidor con 8 núcleos rinde mejor con 30 conexiones activas que con 500: más allá de cierto punto, añadir conexiones no añade throughput, solo añade cambios de contexto y contención. El síntoma clásico es una base de datos que se degrada bajo carga aunque la CPU no esté saturada.

El problema real: muchas instancias, cada una con su pool

El pool de conexiones de tu framework (SQLAlchemy, HikariCP, el pool de pg en Node) resuelve el problema dentro de una instancia: reutiliza conexiones en lugar de abrir una por petición. Pero en producción rara vez tienes una sola instancia.

Haz la cuenta: 40 pods en Kubernetes × 20 conexiones por pool = 800 conexiones contra un PostgreSQL que rinde óptimo con 40. Cada deploy, cada autoescalado y cada worker de Celery suma. El pool local no puede coordinarse con los pools de las demás instancias; necesitas un punto central que multiplexe.

Qué hace un pooler externo

PgBouncer es un proxy ligero que se coloca entre tus aplicaciones y PostgreSQL. Las aplicaciones abren miles de conexiones baratas contra PgBouncer, y PgBouncer las atiende con un puñado de conexiones reales al servidor. La clave está en el modo de pooling, que define cuándo una conexión del servidor queda libre para otro cliente.

Los tres modos de pooling

Session (por defecto)

La conexión del servidor se asigna al cliente hasta que este se desconecta. Es transparente (todo funciona igual que sin pooler), pero apenas multiplexa: si tus clientes mantienen conexiones abiertas, ganas poco. Útil como primer paso o cuando dependes de estado de sesión.

Transaction (el que quieres en producción)

La conexión del servidor se presta solo mientras dura una transacción. Entre transacciones, ese backend atiende a otros clientes. Aquí está la multiplexación de verdad: miles de clientes sobre decenas de conexiones. A cambio, todo lo que dependa de estado de sesión deja de ser fiable, porque tu siguiente transacción puede ejecutarse en otro backend:

- `SET` de variables de sesión (usa `SET LOCAL` dentro de la transacción)
- Tablas temporales que esperas reutilizar entre transacciones
- Advisory locks de sesión (usa las variantes `pg_advisory_xact_lock`)
- `LISTEN` / `NOTIFY`
- Cursores `WITH HOLD`

Statement

La conexión se libera tras cada sentencia; las transacciones multi-sentencia están prohibidas. Es un modo para casos muy específicos (sharding agresivo, cargas 100 % autocommit). Si dudas, no es este.

Prepared statements en modo transaction (PgBouncer ≥ 1.21)

Durante años, la gran pega del modo transaction fueron los prepared statements: se preparan en un backend concreto, y si tu siguiente petición cae en otro backend, PostgreSQL responde con el famoso *prepared statement does not exist*. La solución histórica era desactivar la caché de prepared statements del driver, perdiendo rendimiento.

Desde la versión 1.21, PgBouncer rastrea los prepared statements del protocolo extendido y los re-prepara de forma transparente en el backend que toque. Se activa con un solo parámetro:

```
; 0 = desactivado (por defecto). Un valor > 0 activa el soporte
; y define cuántos prepared statements se mantienen en una caché
; LRU por cada conexión al servidor.
max_prepared_statements = 200
```

Ajusta el valor por encima del número de statements que tu aplicación usa de forma habitual. Cuanto más alto, más memoria consume cada conexión (en PgBouncer y en PostgreSQL), así que no pongas 10.000 por si acaso.

Ojo: esto cubre los prepared statements del protocolo (los que usan los drivers); el `PREPARE` de SQL manual sigue sin estar soportado en modo transaction.

Configuración lista para producción

```
[databases]
app = host=10.0.0.5 port=5432 dbname=app

[pgbouncer]
listen_addr = 0.0.0.0
listen_port = 6432
auth_type = scram-sha-256
auth_file = /etc/pgbouncer/userlist.txt

pool_mode = transaction

; Conexiones reales por par usuario+base de datos
default_pool_size = 20
; Conexiones extra si el pool se agota y un cliente
; lleva esperando más de reserve_pool_timeout segundos
reserve_pool_size = 5
reserve_pool_timeout = 3

; Clientes que PgBouncer acepta (baratos, no son backends)
max_client_conn = 2000

; Prepared statements en modo transaction (1.21+)
max_prepared_statements = 200

; Higiene de conexiones al servidor
server_idle_timeout = 300
server_lifetime = 1800
tcp_keepalive = 1
```

Dos parámetros que la gente confunde: `max_client_conn` es cuántos clientes pueden conectarse a PgBouncer (pon un número generoso, son sockets baratos); `default_pool_size` es cuántas conexiones reales abre contra PostgreSQL (pon un número pequeño, son procesos caros). `server_lifetime` recicla conexiones periódicamente, lo que evita procesos backend eternos con memoria fragmentada.

Cuánto debe medir el pool

Más pequeño de lo que crees. Un punto de partida razonable para cargas mixtas es:

```
conexiones ≈ (núcleos × 2) + discos_efectivos
```

Para un servidor de 8 núcleos con SSD, eso son ~20 conexiones activas. Puede parecer poco para "miles de usuarios", pero recuerda: cada conexión solo está ocupada mientras ejecuta una transacción, que suele durar milisegundos. Es preferible que los clientes esperen unos milisegundos en la cola de PgBouncer (`cl_waiting`) a que 300 backends compitan por 8 núcleos.

La cifra correcta se mide, no se calcula: sube o baja `default_pool_size` observando el throughput y la latencia p99 de tu aplicación. Si al aumentar el pool la latencia no mejora, ya te has pasado.

Configurar tu aplicación (Python)

Apunta el driver a PgBouncer (puerto 6432) y reduce el pool local: su trabajo ya no es multiplexar, solo amortizar la creación de sockets.

```
import asyncpg

pool = await asyncpg.create_pool(
    dsn="postgresql://app_user:secret@pgbouncer:6432/app",
    min_size=2,
    max_size=10,
    # Solo si tu PgBouncer es < 1.21 o no has activado
    # max_prepared_statements:
    # statement_cache_size=0,
)
```

```
from sqlalchemy.ext.asyncio import create_async_engine

engine = create_async_engine(
    "postgresql+asyncpg://app_user:secret@pgbouncer:6432/app",
    pool_size=5,
    max_overflow=5,
    # Detecta conexiones muertas tras reinicios del pooler
    pool_pre_ping=True,
)
```

Con `max_prepared_statements` activado en PgBouncer puedes dejar la caché de statements del driver con sus valores por defecto y conservar su rendimiento. Si no puedes actualizar PgBouncer, desactívala (`statement_cache_size=0` en asyncpg) o verás errores intermitentes de prepared statements bajo carga.

Monitorización: SHOW POOLS es tu amigo

PgBouncer expone una consola de administración en una pseudo-base de datos llamada `pgbouncer`:

```
psql -h pgbouncer -p 6432 -U admin_user pgbouncer

SHOW POOLS;
-- database | user | cl_active | cl_waiting | sv_active | sv_idle | maxwait
SHOW STATS;
-- peticiones, bytes y tiempos medios de query y transacción
```

Las columnas que importan: `cl_waiting` (clientes en cola ahora mismo) y `maxwait` (segundos que lleva esperando el cliente más antiguo). Un `cl_waiting` que oscila entre 0 y unos pocos es sano; un `maxwait` que

crece de forma sostenida significa que el pool se queda corto o que hay transacciones largas acaparando conexiones. Alerta sobre `maxwait`, no sobre el número de clientes.

Errores comunes

Usar SET en modo transaction. La variable queda en un backend aleatorio y "se filtra" a otros clientes. Usa `SET LOCAL` o parámetros por consulta.

Sobredimensionar el pool. Duplicar `default_pool_size` ante un pico de latencia suele empeorarlo: más backends compitiendo por los mismos núcleos.

Ignorar que PgBouncer es mono-hilo. Una instancia satura un núcleo con decenas de miles de QPS. La solución estándar es ejecutar varios procesos con `so_reuseport = 1` en el mismo puerto.

Transacciones largas. En modo transaction, una transacción de 30 segundos secuestra una conexión real 30 segundos. Vigila *idle in transaction* y pon `idle_in_transaction_session_timeout` en PostgreSQL.

Health checks que abren transacciones. Un `SELECT 1` en autocommit basta; no envuelvas el health check en una transacción con reintentos.

PgBouncer frente a las alternativas

PgBouncer (escrito en C, en producción desde 2007) sigue siendo la opción por defecto: maduro, con una huella de memoria mínima y soporte de prepared statements desde 1.21. Pero ya no está solo. PgCat, escrito en Rust, es multihilo de serie y añade balanceo entre primario y réplicas y soporte de sharding; si una sola instancia de PgBouncer se te queda corta en CPU, es el candidato natural. Supavisor (Elixir, creado por Supabase) está diseñado para multi-tenancy masivo, con aislamiento de pools por tenant. Y si estás en cloud, valora los gestionados: RDS Proxy en AWS o el PgBouncer integrado de Azure Database for PostgreSQL Flexible Server te ahorran operar el pooler.

Checklist de producción

- `pool_mode = transaction` y código libre de estado de sesión
- `max_prepared_statements` > statements habituales de tu app (PgBouncer ≥ 1.21)
- `default_pool_size` pequeño (empieza por núcleos × 2) y medido con carga real
- `max_client_conn` generoso; pools locales de la app reducidos (5–10 por instancia)
- Alertas sobre `maxwait` y sobre *idle in transaction* en PostgreSQL
- `server_lifetime` configurado para reciclar conexiones
- Varias instancias de PgBouncer (`so_reuseport`) si el tráfico crece

Un pooler bien dimensionado es de las mejoras con mejor relación esfuerzo/beneficio que existen para PostgreSQL: una tarde de trabajo que convierte "la base de datos no aguanta más conexiones" en un problema resuelto de forma permanente.

Autenticación en producción: userlist.txt, auth_query y SCRAM

El ejemplo de configuración anterior usa `auth_file`, y para una base de datos con dos o tres usuarios es suficiente. Pero un `userlist.txt` estático tiene un problema operativo serio: cada vez que alguien rota una contraseña en PostgreSQL, tienes que regenerar el archivo y recargarlo en todas las instancias de PgBouncer. En equipos grandes eso acaba en credenciales desincronizadas y errores de autenticación a las tres de la mañana.

La alternativa es `auth_query`: en lugar de leer las contraseñas de un archivo, PgBouncer las consulta directamente a PostgreSQL cuando un cliente intenta autenticarse. Solo necesitas un usuario técnico (el `auth_user`) con permiso para ejecutar esa consulta:

```
[pgbouncer]
auth_type = scram-sha-256
auth_user = pgbouncer_auth
auth_query = SELECT usename, passwd FROM user_lookup($1)
```

Y en PostgreSQL, una función `SECURITY DEFINER` que limita qué puede ver ese usuario técnico. No le des acceso directo a `pg_shadow`; expón solo lo imprescindible:

```
CREATE OR REPLACE FUNCTION user_lookup(p_user text)
RETURNS TABLE (usename name, passwd text)
LANGUAGE sql SECURITY DEFINER SET search_path = pg_catalog AS $$
  SELECT usename, passwd
  FROM pg_shadow
  WHERE usename = p_user
    AND usename NOT IN ('postgres', 'pgbouncer_auth');
$$;

REVOKE ALL ON FUNCTION user_lookup(text) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION user_lookup(text) TO pgbouncer_auth;
```

Con esto, la rotación de contraseñas es transparente: cambias la contraseña en PostgreSQL con `ALTER ROLE ... PASSWORD` y PgBouncer la recoge en el siguiente intento de login, sin tocar ningún archivo. La función además excluye explícitamente al superusuario y al propio usuario técnico, de modo que nadie puede autenticarse como ellos a través del pooler.

Sobre el método: usa `scram-sha-256` siempre que puedas. El soporte de SCRAM en PgBouncer es maduro, y desde la 1.25 su rendimiento mejoró notablemente (el handshake SCRAM es costoso en CPU, y en picos de reconexión masiva se notaba). Si vienes de una instalación antigua con `md5`, migra: PostgreSQL lo considera obsoleto y las contraseñas md5 desaparecerán en versiones futuras. Desde la 1.25 existe también autenticación LDAP nativa, útil si tu organización centraliza credenciales en Active Directory u OpenLDAP, aunque añade una dependencia externa en el camino crítico de cada login: si el LDAP se cae, nadie nuevo se conecta.

Un detalle que sorprende: en modo `auth_query`, la consulta de autenticación se ejecuta a través de una conexión del pool de la base de datos correspondiente. Si tu pool está saturado, hasta el login puede quedarse en cola. Es un motivo más para dimensionar con margen y vigilar `maxwait`.

TLS: cifrado entre la app, PgBouncer y PostgreSQL

PgBouncer participa en dos tramos de red distintos, y cada uno se cifra por separado: el tramo cliente → PgBouncer (parámetros `client_tls_*`) y el tramo PgBouncer → PostgreSQL (parámetros `server_tls_*`). Es perfectamente posible —y común— cifrar solo uno de los dos, por ejemplo cuando PgBouncer corre en la misma máquina que la aplicación pero la base de datos está en otra red.

```
; Tramo cliente -> PgBouncer
client_tls_sslmode = require
client_tls_cert_file = /etc/pgbouncer/tls/server.crt
client_tls_key_file = /etc/pgbouncer/tls/server.key

; Tramo PgBouncer -> PostgreSQL
server_tls_sslmode = verify-full
server_tls_ca_file = /etc/pgbouncer/tls/root.crt
```

La asimetría es deliberada: hacia el cliente basta con `require` (el cliente ya confía en la infraestructura donde corre PgBouncer), pero hacia el servidor conviene `verify-full`, que valida el certificado y el nombre del host. Sin eso, un atacante con posición en la red podría suplantar a PostgreSQL y PgBouncer le entregaría las credenciales sin protestar.

Dos consideraciones de rendimiento. Primera: el handshake TLS se paga al abrir conexión, no por consulta; como PgBouncer mantiene pocas conexiones al servidor y de larga vida, el coste de `server_tls` es despreciable. El tramo cliente es distinto: si tus aplicaciones abren y cierran conexiones constantemente, cada apertura paga handshake TCP + TLS + autenticación, y eso sí se nota. La respuesta correcta no es quitar TLS, sino mantener un pequeño pool local en la aplicación (como vimos arriba) para amortizar las aperturas. Segunda: desde la 1.25 PgBouncer acepta conexiones TLS "directas" (el cliente empieza el handshake TLS sin la negociación previa del protocolo de PostgreSQL), lo que ahorra un round-trip en cada conexión nueva y simplifica pasar por balanceadores TCP que esperan TLS puro.

Y una advertencia operativa: los certificados caducan. PgBouncer recarga certificados con un simple `RELOAD`, así que integra la renovación (Let's Encrypt, cert-manager, Vault) con un `SHOW VERSION` de verificación y un `RELOAD` posterior. Un certificado caducado en el pooler es un incidente total: nadie se conecta.

PgBouncer en Kubernetes: ¿sidecar o servicio centralizado?

En Kubernetes hay dos formas razonables de desplegar PgBouncer, y elegir mal anula gran parte del beneficio.

Sidecar: un contenedor PgBouncer dentro de cada pod de aplicación. La latencia añadida es mínima (localhost) y la configuración viaja con el pod. Pero cuidado: cada sidecar mantiene su propio pool contra PostgreSQL. Con 40 pods y `default_pool_size = 20` vuelves a tener 800 conexiones reales: exactamente el problema

que querías resolver. El sidecar solo tiene sentido con pools minúsculos (2–5 conexiones) o como capa de amortiguación antes de un pooler central.

Servicio centralizado: un Deployment de 2–3 réplicas de PgBouncer detrás de un Service ClusterIP. Todas las aplicaciones apuntan al Service, y el número total de conexiones reales a PostgreSQL es `réplicas × default_pool_size`, un número que controlas en un solo sitio. Es la topología que multiplexa de verdad, y la que recomiendo por defecto:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pgbouncer
spec:
  replicas: 2
  selector:
    matchLabels: { app: pgbouncer }
  template:
    metadata:
      labels: { app: pgbouncer }
    spec:
      containers:
        - name: pgbouncer
          image: bitnami/pgbouncer:1.25.2
          ports:
            - containerPort: 6432
          resources:
            requests: { cpu: "500m", memory: "128Mi" }
            limits: { memory: "256Mi" }
          readinessProbe:
            tcpSocket: { port: 6432 }
            periodSeconds: 5
          livenessProbe:
            tcpSocket: { port: 6432 }
            initialDelaySeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: pgbouncer
spec:
  selector: { app: pgbouncer }
  ports:
    - port: 6432
      targetPort: 6432
```

Detalles que importan en este manifiesto. Sin límite de CPU (`limits.cpu` ausente): PgBouncer es mono-hilo y sensible a la latencia; el throttling de CFS le sienta fatal y es preferible dejarle usar el núcleo que necesite. Límite de memoria sí, porque su consumo es pequeño y predecible. Probes por TCP y no por consulta SQL: un health check que ejecuta consultas consume conexiones del pool y puede provocar justo la saturación que intenta detectar.

Añade un `PodDisruptionBudget` con `minAvailable: 1` para que los drenajes de nodos no tumben todas las réplicas a la vez, y reparte las réplicas entre zonas con `topologySpreadConstraints`. Recuerda también que

cada réplica tiene su pool independiente: dos réplicas con `default_pool_size = 20` son hasta 40 conexiones reales. Ajusta el tamaño por réplica dividiendo tu presupuesto total de conexiones entre el número de réplicas.

¿Y el operador de turno? Si usas CloudNativePG, Crunchy PGO o Zalando, todos traen integración de PgBouncer (CloudNativePG tiene el CRD `Pooler`, por ejemplo). Úsala: te resuelve certificados, secretos y reinicios coordinados con el ciclo de vida del clúster.

Operación sin cortes: RELOAD, PAUSE/RESUME y rolling restarts

La consola de administración no es solo para mirar estadísticas; es la herramienta de operación. Los tres comandos que usarás:

```
-- Aplica cambios de configuración sin reiniciar
RELOAD;

-- Deja terminar las transacciones en curso y retiene
-- las nuevas consultas en cola (los clientes no ven error)
PAUSE;

-- Reanuda el tráfico retenido
RESUME;
```

`RELOAD` relee el archivo INI y aplica la mayoría de parámetros en caliente: tamaños de pool, timeouts, certificados TLS. Es la razón por la que casi nunca necesitas reiniciar PgBouncer para un cambio de configuración.

El dúo `PAUSE` / `RESUME` es oro para mantenimientos de PostgreSQL. La secuencia para un failover o una actualización menor: `PAUSE` (PgBouncer espera a que acaben las transacciones activas y encola lo nuevo), ejecutas el switchover o reinicias PostgreSQL, `RESUME`. Si la ventana dura pocos segundos, tus aplicaciones solo perciben un pico de latencia, no una lluvia de errores de conexión. Es la diferencia entre "mantenimiento invisible" y una página de incidentes.

Para actualizar PgBouncer en sí, desde la versión 1.23 el comportamiento de las señales está pensado para rolling restarts: `SIGTERM` ya no mata el proceso de inmediato, sino que inicia un apagado "super seguro" que deja de aceptar clientes nuevos y espera a que los existentes se desconecten. El apagado inmediato clásico ahora es `SIGQUIT`. Si tenías un Dockerfile o una unidad de systemd que dependía del viejo comportamiento de SIGTERM, revísalo al actualizar. El procedimiento de rolling restart con varias instancias en `so_reuseport`: arranca la instancia nueva, manda SIGTERM a la vieja, y el kernel va dirigiendo las conexiones nuevas a la instancia superviviente mientras la antigua se vacía.

Queda el failover del propio PostgreSQL. Si el primario cambia de IP detrás de un nombre DNS (típico con Patroni + un DNS interno, o con endpoints gestionados en cloud), PgBouncer re-resuelve el nombre respetando `dns_max_ttl` (15 segundos por defecto). Bájalo si tu failover automatizado es más rápido que eso. Combínalo con `server_fast_close = 1` para que las conexiones al servidor viejo se cierren en cuanto queden libres tras un `RELOAD` o un cambio de destino, en lugar de agotarse lentamente. Y en el lado de la aplicación, `pool_pre_ping` (SQLAlchemy) o el equivalente de tu driver detecta las conexiones muertas que sobrevivieron al cambio.

Qué trae PgBouncer 1.25 (y por qué conviene actualizar)

Este artículo asume PgBouncer moderno, y merece la pena concretar qué ha cambiado recientemente. La serie 1.25 (la actual; la 1.25.2 salió en mayo de 2026) trae cuatro cosas relevantes para producción:

- **Autenticación LDAP** nativa, para organizaciones que centralizan credenciales fuera de PostgreSQL.
- **Conexiones TLS directas** del lado cliente: menos round-trips al conectar y compatibilidad con balanceadores que esperan TLS estándar.
- **SCRAM mucho más rápido**: el coste de CPU del handshake de autenticación bajó de forma notable, lo que se aprecia en tormentas de reconexión (deploys masivos, reinicios de flotas).
- **Clientes ociosos reportados como `idle`** en lugar de `active`, lo que hace que `SHOW CLIENTS` y las métricas derivadas por fin distingan quién está haciendo algo de quién solo mantiene el socket abierto.

Además, la 1.25.2 corrige varias vulnerabilidades relacionadas con autenticación y comandos de administración. Un pooler es un componente de seguridad crítico —ve todas tus credenciales y todo tu tráfico SQL—, así que trátalo como tal: suscríbete a los anuncios de versiones y actualiza con la misma disciplina que aplicarías a OpenSSL o al propio PostgreSQL. Con rolling restarts, actualizar ya no tiene excusa operativa.

Observabilidad seria: pgbouncer_exporter y Prometheus

`SHOW POOLS` a mano está bien para depurar, pero en producción quieres series temporales y alertas. El camino estándar es el `pgbouncer_exporter` de la comunidad Prometheus: se conecta a la pseudo-base `pgbouncer`, ejecuta los comandos `SHOW` y expone el resultado como métricas. Despliégalo como sidecar del pooler (aquí el sidecar sí tiene sentido: es solo lectura) y apunta Prometheus al puerto del exporter.

Las señales que de verdad predicen problemas, en orden de importancia:

- **Clientes en espera** (columna `cl_waiting`, métrica de la familia `pgbouncer_pools_client_waiting_connections`): cuántas peticiones están en cola ahora mismo esperando conexión real.
- **Tiempo máximo de espera** (`maxwait`): cuánto lleva esperando el cliente más antiguo. Es tu métrica de alerta principal.
- **Saturación del pool**: `sv_active / (sv_active + sv_idle)`. Al 100 % sostenido, todo lo demás empeora.
- **Errores de login y conexiones rechazadas**, que delatan problemas de credenciales o un `max_client_conn` corto.

Una alerta razonable en PromQL (ajusta el umbral a tu SLO de latencia):

```
- alert: PgBouncerClientsWaiting
  expr: max_over_time(pgbouncer_pools_client_maxwait_seconds[5m]) > 1
  for: 5m
  labels: { severity: warning }
  annotations:
```

```
    summary: "Clientes esperando >1s por una conexión del pool"
    description: "El pool {{ $labels.database }}/{{ $labels.user }} lleva 5 minutos con
esperas. Revisa transacciones largas antes de ampliar el pool."
```

Fíjate en la anotación: la reacción correcta ante esperas no es subir el pool a ciegas, sino mirar primero si hay transacciones largas secuestrando conexiones. Para eso, correla con PostgreSQL: `pg_stat_activity` te dice qué hace cada backend, y estas dos consultas son las que más veces resuelven el misterio:

```
-- Transacciones abiertas más antiguas (candidatas a secuestrar el pool)
SELECT pid, usename, state,
       now() - xact_start AS xact_age,
       left(query, 80) AS query
FROM pg_stat_activity
WHERE xact_start IS NOT NULL
ORDER BY xact_start
LIMIT 10;

-- ¿Cuántos backends hay por estado?
SELECT state, count(*)
FROM pg_stat_activity
GROUP BY state;
```

Si `maxwait` sube y a la vez ves transacciones con `xact_age` de minutos, el problema es la aplicación (o una migración olvidada en una terminal), no el tamaño del pool. Si `maxwait` sube con transacciones cortas y la CPU de PostgreSQL tiene margen, entonces sí: sube `default_pool_size` un paso y vuelve a medir.

Completa el cuadro con los logs: `log_connections = 0` y `log_disconnections = 0` en producción (a alto QPS generan ruido carísimo), pero deja `log_pooler_errors = 1`, que es donde aparecen los rechazos y los timeouts con su causa.

Mide tu pool con pgbench

Todo lo anterior son heurísticas; la verdad la da la medición. `pgbench`, que viene con PostgreSQL, sirve perfectamente para comparar "directo contra PostgreSQL" frente a "a través de PgBouncer" y para encontrar el tamaño de pool donde tu hardware satura.

Primero, el caso que más favorece al pooler: conexiones nuevas por transacción (flag `-C`), que simula aplicaciones sin pool local, scripts, funciones serverless:

```
# Inicializa datos de prueba (escala 50 ≈ 5M filas)
pgbench -i -s 50 -h db.internal -p 5432 app

# Directo a PostgreSQL, abriendo conexión por transacción
pgbench -C -c 50 -j 4 -T 60 -h db.internal -p 5432 app

# Lo mismo, a través de PgBouncer
pgbench -C -c 50 -j 4 -T 60 -h pgbouncer.internal -p 6432 app
```

Con `-C`, la diferencia suele ser dramática —varias veces más transacciones por segundo vía PgBouncer— porque el coste dominante es abrir conexiones, que es justo lo que el pooler amortiza. Los números concretos dependen de tu hardware y tu red: trata cualquier cifra publicada (incluidas las de este artículo) como orientativa y mide en tu entorno.

Segundo, la curva de saturación. Con conexiones persistentes (`sin -C`), sube la concurrencia por pasos y anota el throughput y la latencia:

```
for c in 10 20 40 80 160 320; do
  pgbench -c $c -j 8 -T 30 -P 10 \
    -h pgbouncer.internal -p 6432 app \
    | grep -E "tps|latency average"
done
```

Verás el patrón clásico: el throughput crece hasta la zona del pool bien dimensionado, se aplana, y a partir de ahí solo crece la latencia. Ese codo es tu número. Repite el barrido con dos o tres valores de `default_pool_size` (por ejemplo 15, 20 y 30 para 8 núcleos) y quédate con el que maximiza throughput con la p99 dentro de tu objetivo. Dos advertencias metodológicas: ejecuta pgbench desde una máquina distinta a la base de datos (si no, compiten por CPU y el resultado miente), y haz cada medición al menos dos veces; la primera calienta cachés.

Más allá de Python: Node, Prisma, Django y Go

El patrón es idéntico en todos los ecosistemas —apuntar al puerto 6432, reducir el pool local, vigilar los prepared statements—, pero cada stack tiene su matiz.

Node.js (node-postgres). El `Pool` de `pg` no usa prepared statements con nombre por defecto (envía consultas parametrizadas por el protocolo extendido sin cachearlas), así que funciona bien en modo transaction sin tocar nada:

```
import pg from "pg";

const pool = new pg.Pool({
  host: "pgbouncer.internal",
  port: 6432,
  database: "app",
  user: "app_user",
  password: process.env.PGPASSWORD,
  max: 10,                    // pool local pequeño
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 5000,
});
```

Evita solo las consultas con `name:` explícito (eso sí crea un prepared statement con nombre) salvo que tu PgBouncer tenga `max_prepared_statements` activado.

Prisma. Prisma usa prepared statements internamente, y lo sabe: añade `?pgbouncer=true` a la URL para que los desactive, y define `directUrl` para que las migraciones vayan directas a PostgreSQL (las migraciones usan advisory locks de sesión y DDL que no quieres pasar por modo transaction):

```
datasource db {
  provider  = "postgresql"
  url       = env("DATABASE_URL")    // ...@pgbouncer:6432/app?pgbouncer=true
  directUrl = env("DIRECT_URL")      // ...@db:5432/app
}
```

Django. Dos ajustes. `CONN_MAX_AGE = 0` (el valor por defecto) hace que Django abra y cierre conexión por petición, que contra PgBouncer es barato y encaja bien con el modo transaction. Y `DISABLE_SERVER_SIDE_CURSORS = True`, porque los cursores de servidor que Django usa en `QuerySet.iterator()` dependen de estado de sesión:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "HOST": "pgbouncer.internal",
        "PORT": "6432",
        "NAME": "app",
        "CONN_MAX_AGE": 0,
        "DISABLE_SERVER_SIDE_CURSORS": True,
    }
}
```

Go (pgx). El pool de `pgxpool` cachea prepared statements por defecto. Con PgBouncer moderno y `max_prepared_statements` activado puedes dejarlo; si no, cambia el modo de ejecución al protocolo simple:

```
cfg, _ := pgxpool.ParseConfig(
    "postgres://app_user@pgbouncer.internal:6432/app")
cfg.MaxConns = 10
// Solo si PgBouncer < 1.21 o sin max_prepared_statements:
// cfg.ConnConfig.DefaultQueryExecMode = pgx.QueryExecModeSimpleProtocol
pool, err := pgxpool.NewWithConfig(ctx, cfg)
```

La regla transversal: si ves errores intermitentes de *prepared statement does not exist* o *bind message* bajo carga, en cualquier lenguaje, la causa es casi siempre la misma —prepared statements con nombre atravesando modo transaction sin soporte activado— y las soluciones son las dos de siempre: activar `max_prepared_statements` en PgBouncer o desactivar la caché del driver.

Casos límite: migraciones, COPY, LISTEN/NOTIFY y trabajos largos

Hay operaciones que no encajan en modo transaction, y la solución profesional no es retorcerlas hasta que pasen, sino darles un camino aparte.

Migraciones de esquema. Herramientas como Alembic, Flyway o las migraciones de Django/Prisma usan advisory locks de sesión para garantizar que solo una instancia migra a la vez. En modo transaction ese lock

puede quedarse en un backend distinto del que ejecuta el DDL, con resultados desconcertantes. Ejecuta las migraciones contra PostgreSQL directo (puerto 5432) o define en PgBouncer una entrada adicional con `pool_mode = session`:

```
[databases]
app = host=10.0.0.5 dbname=app
; Misma base, pool de sesión para tareas administrativas
app_admin = host=10.0.0.5 dbname=app pool_mode=session pool_size=3
```

El override por base de datos es una de las funciones más útiles y menos conocidas de PgBouncer: distintos modos y tamaños de pool contra el mismo servidor, seleccionables con solo cambiar el nombre de base de datos en la cadena de conexión del job.

LISTEN/NOTIFY. No funciona en modo transaction (la suscripción vive en un backend concreto). Si tu aplicación lo usa —invalidación de cachés, colas ligeras—, abre para ello una única conexión dedicada directa a PostgreSQL, fuera del pooler, y mantenla con reconexión automática. Una conexión persistente extra no arruina el presupuesto; cientos sí.

COPY y cargas masivas. `COPY` funciona en modo transaction (es una sentencia), pero una carga de veinte minutos monopoliza una conexión real todo ese tiempo. Igual que las transacciones largas de análisis: mándalas por el pool de sesión administrativo o directas al servidor, y deja el pool de transaction para el tráfico OLTP que es su razón de ser.

Timeouts defensivos. En PostgreSQL, fija `idle_in_transaction_session_timeout` (por ejemplo, 60 segundos) para matar transacciones abandonadas, y un `statement_timeout` global sensato con overrides por rol para los jobs pesados. En PgBouncer, `query_wait_timeout` (120 segundos por defecto) decide cuánto puede esperar un cliente en cola antes de recibir error; bájalo si prefieres fallar rápido y reintentar. Estos límites convierten los incidentes de "todo se ha quedado colgado" en errores localizados con un culpable identificable en los logs.

Caso práctico: dimensionar un SaaS de 60 pods

Aterricemos todo con números. Supón un SaaS con esta forma: 60 pods de API en Kubernetes, 8 workers de Celery para tareas de fondo, un PostgreSQL primario de 16 núcleos con SSD NVMe, picos de 4.000 peticiones por segundo, y transacciones OLTP que duran 2–8 ms.

Paso 1: presupuesto de conexiones reales. Con 16 núcleos y SSD, la fórmula orientativa da `16 × 2 + 1 ≈ 33`. Redondeamos a 40 como techo total de conexiones activas deseadas, y reservamos margen en `max_connections` (por ejemplo 100) para superusuarios, réplicas y emergencias.

Paso 2: ¿da el throughput? Comprobación de coherencia: 40 conexiones × (1000 ms / 5 ms por transacción) = 8.000 transacciones por segundo teóricas, el doble del pico esperado. Bien. Si esta cuenta no saliera, ninguna configuración de pooling la arregla: tocaría optimizar consultas o escalar hardware.

Paso 3: topología. Pooler centralizado, 2 réplicas de PgBouncer (alta disponibilidad y margen de CPU). Presupuesto por réplica: `default_pool_size = 18` (2 × 18 = 36 conexiones, deja hueco para el pool administrativo), `reserve_pool_size = 4` con `reserve_pool_timeout = 2` para picos breves.

Paso 4: separar cargas. Los workers de Celery ejecutan transacciones más largas (50–500 ms). Si comparten pool con la API, un lote de tareas pesadas puede inducir esperas en las peticiones interactivas. Solución: usuarios distintos (`app_api` y `app_worker`), porque PgBouncer crea pools por par usuario+base de datos, con límites propios:

```
[databases]
app = host=10.0.0.5 dbname=app

[users]
app_api = pool_size=14
app_worker = pool_size=4 pool_mode=transaction
```

Así, los workers pueden saturar sus 4 conexiones sin robar ni una a la API. Es el mismo principio de aislamiento que aplicarías con colas separadas en un sistema de mensajería.

Paso 5: lado aplicación. Cada pod de API con pool local de 5 (`pool_size=5, max_overflow=0` en SQLAlchemy): 60 × 5 = 300 conexiones cliente contra PgBouncer, muy por debajo de un `max_client_conn = 2000`. Cada worker con 2. `pool_pre_ping` activado en todos.

Paso 6: validar y vigilar. Barrido de pgbench como el de la sección anterior para confirmar el codo de saturación cerca de 36–40 conexiones; alertas sobre `maxwait` > 1 s y sobre `idle in transaction` > 60 s. Revisión al mes de datos reales: si la p99 de la API va holgada y `cl_waiting` es casi siempre 0, incluso puedes recortar el pool y devolverle RAM a PostgreSQL.

El resultado final: 68 procesos cliente potencialmente ruidosos convertidos en como mucho 40 backends ordenados, con los picos absorbidos por una cola que mide milisegundos. Y cada número de esta receta tiene una justificación que puedes re-derivar cuando tu carga cambie.

PgCat en detalle: cuándo dar el salto

Arriba mencioné las alternativas; profundicemos en la decisión, porque "PgBouncer se me queda corto" significa cosas distintas según el caso.

Señal 1: CPU. PgBouncer es mono-hilo: una instancia satura un núcleo en torno a decenas de miles de QPS (los benchmarks públicos lo sitúan cerca de 40–45k tps en hardware moderno, degradándose con cientos de clientes concurrentes). La primera respuesta es barata: varias instancias con `so_reuseport`. Si aun así la operación de N procesos te pesa, PgCat —multihilo sobre el runtime asíncrono de Rust— rinde más del doble que PgBouncer con cientos de clientes concurrentes en benchmarks independientes, con una sola instancia que aprovecha todos los núcleos.

Señal 2: réplicas. Si quieres repartir lecturas entre réplicas, con PgBouncer necesitas lógica en la aplicación (dos pools, uno de escritura y otro de lectura). PgCat balancea entre servidores y puede excluir réplicas caídas automáticamente. Un ejemplo mínimo de su configuración:

```
# pgcat.toml (simplificado)
[pools.app]
pool_mode = "transaction"
default_role = "any"        # reparte entre primario y réplicas

[pools.app.shards.0]
servers = [
  ["10.0.0.5", 5432, "primary"],
  ["10.0.0.6", 5432, "replica"],
  ["10.0.0.7", 5432, "replica"],
]
database = "app"

[pools.app.users.0]
username = "app_user"
pool_size = 40
```

PgCat incluso sabe enrutar por sentencia (lecturas a réplicas, escrituras al primario) si se lo pides, aunque ese modo exige revisar bien las transacciones de tu aplicación.

Señal 3: multi-tenancy masivo. Si operas miles de bases de datos o tenants (una plataforma, un SaaS por-tenant-una-base), Supavisor —el pooler en Elixir de Supabase— está diseñado para eso: se escala horizontalmente en clúster y aísla pools por tenant. Para un SaaS convencional con una base de datos, es más máquina de la que necesitas.

Señal 4: no quieres operar el pooler. En cloud gestionado, la balanza cambia. RDS Proxy (AWS) se integra con IAM y Secrets Manager y multiplexa bien, con un matiz importante: el *pinning*. Cuando una sesión usa características con estado (ciertos SET, prepared statements según configuración, tablas temporales), el proxy "ancla" esa conexión al cliente y deja de multiplexarla; el síntoma es que el número de conexiones a la base sube hasta parecerse al de clientes, y la métrica de conexiones ancladas te lo confirma. Las mismas disciplinas de código que exige el modo transaction de PgBouncer minimizan el pinning. Azure Database for PostgreSQL Flexible Server trae PgBouncer integrado activable con un interruptor (puerto 6432), y tanto Neon como Supabase exponen endpoints con pooling incorporado —en Neon, añadiendo el sufijo `-pooler` al host—. Para cargas serverless (Lambda, funciones que escalan a cientos de instancias), algún pooler gestionado o incorporado no es opcional: es la única defensa realista contra tormentas de conexiones.

La recomendación práctica no cambia: empieza con PgBouncer (o el pooler que tu plataforma te regala), instrumenta, y migra solo cuando una de estas cuatro señales aparezca en tus métricas, no en un artículo de benchmarks.

Guía de resolución de problemas: los errores que verás y qué significan

Cuando algo falla a través de un pooler, el mensaje de error suele llegar mezclado: a veces lo emite PgBouncer, a veces PostgreSQL, y a veces el driver traduciéndolos a su manera. Esta es la lista de los que aparecen una y otra vez, con su causa real.

`no more connections allowed (max_client_conn)`. PgBouncer rechaza clientes nuevos porque alcanzaste `max_client_conn`. Casi siempre es una fuga de conexiones en la aplicación (pools locales que crecen sin

límite, procesos zombis) más que un límite mal puesto. Antes de subir el número, ejecuta `SHOW CLIENTS` y mira de qué direcciones IP viene la multitud.

`query_wait_timeout`. El cliente esperó en cola más de lo permitido y PgBouncer lo expulsó. Es el error "sano": indica que el sistema prefirió fallar rápido a colgarse. Investiga por qué el pool estuvo ocupado tanto tiempo (transacciones largas, pico de tráfico, pool corto) en lugar de simplemente subir el timeout.

`unsupported startup parameter: options` (o `search_path`, o `application_name` en versiones antiguas). Algunos drivers y ORMs envían parámetros extra en el arranque de la conexión que PgBouncer no reconoce por defecto. La solución es declararlos inofensivos:

```
; Acepta y reenvía parámetros de arranque no críticos
ignore_startup_parameters = extra_float_digits,options
```

Es probablemente la línea de configuración más buscada en los issues de PgBouncer: JDBC envía `extra_float_digits`, algunos frameworks envían `options=-c search_path=...`, y sin esta línea la conexión muere antes de nacer.

`server login has been failing` / **`login failed`**. PgBouncer no consigue autenticarse contra PostgreSQL con las credenciales de `auth_file` o `auth_query`. Suele pasar tras rotar contraseñas con userlist estático, o cuando el `auth_user` perdió permisos sobre la función de lookup. Mientras tanto, los clientes ven timeouts, no errores de contraseña, lo que despista: el fallo real está en el tramo pooler → servidor.

`idle_in_transaction_session_timeout` matando conexiones del pooler. Si activas ese timeout en PostgreSQL (recomendado), recuerda que en modo transaction una conexión del pool puede quedar legítimamente `idle` entre préstamos. El timeout de PostgreSQL que importa aquí es *idle in transaction*, no *idle* a secas; no configures `idle_session_timeout` agresivo en el servidor o matará conexiones sanas del pool y verás reconexiones constantes en los logs de PgBouncer.

Latencia que aparece de la nada con el pool "vacío". Revisa `min_pool_size`: por defecto es 0, así que tras un periodo de calma el pool está frío y la primera ráfaga de tráfico paga la apertura de conexiones reales (más el handshake TLS y la autenticación). Un `min_pool_size` de la mitad del pool mantiene conexiones calientes y elimina ese primer pico:

```
; Mantén al menos 10 conexiones abiertas aunque no haya tráfico
min_pool_size = 10
```

La familia de timeouts, ordenada

PgBouncer tiene más de una docena de timeouts y la documentación los lista alfabéticamente, que es la peor forma de entenderlos. Agrupados por lo que protegen:

Protegen al cliente de esperar para siempre:

- `query_wait_timeout` (120 s por defecto): máximo en cola esperando conexión. El que convierte saturación en errores visibles.

- `client_login_timeout` (60 s): máximo para completar el login. Si el auth_query está lento, este es el que salta.

Protegen al servidor de clientes descuidados:

- `client_idle_timeout` (0 = desactivado): desconecta clientes ociosos. Útil contra fugas de conexiones; ponlo por encima del keepalive de tu aplicación si lo activas.
- `query_timeout` (0 = desactivado): mata consultas que exceden el límite. Mejor déjalo en 0 y usa `statement_timeout` en PostgreSQL, que devuelve un error SQL limpio en lugar de cortar la conexión.

Higiene del pool:

- `server_idle_timeout` (600 s): cierra conexiones al servidor que llevan demasiado ociosas, devolviendo memoria a PostgreSQL.
- `server_lifetime` (3600 s): recicla conexiones aunque estén sanas; evita backends eternos.
- `server_connect_timeout` (15 s) y `server_login_retry` (15 s): cuánto espera al abrir conexión nueva y cuánto tarda en reintentar tras un fallo. En failovers rápidos, bajar `server_login_retry` a 2–5 s acelera la recuperación.

La regla general: los timeouts de PgBouncer gestionan *conexiones*; los de PostgreSQL (`statement_timeout`, `idle_in_transaction_session_timeout`, `lock_timeout`) gestionan *SQL*. Cada problema tiene su capa, y usar la equivocada produce errores crípticos.

Alta disponibilidad fuera de Kubernetes

Si tu infraestructura son máquinas virtuales o bare metal, el patrón equivalente al Deployment con réplicas es: dos o más instancias de PgBouncer en máquinas distintas, y un mecanismo que reparta o conmute el tráfico entre ellas.

La opción más simple y robusta es **keepalived con una IP virtual**: las dos máquinas comparten una VIP que solo una posee en cada momento; si la activa cae, la VIP migra en segundos y los clientes reconectan contra la misma dirección. No reparte carga (una instancia está ociosa), pero para la mayoría de cargas una instancia de PgBouncer sobra, y la operación es trivial.

Si necesitas repartir carga además de conmutar, pon **HAProxy en modo TCP** delante de los poolers:

```
listen pgbouncer
    bind *:6432
    mode tcp
    option tcp-check
    balance leastconn
    server pgb1 10.0.1.10:6432 check inter 2s fall 2
    server pgb2 10.0.1.11:6432 check inter 2s fall 2
```

`mode tcp` es obligatorio (HAProxy no debe interpretar el protocolo de PostgreSQL) y `leastconn` reparte mejor que round-robin cuando las conexiones tienen vidas dispares. El health check TCP básico detecta procesos caídos; si quieres detectar un PgBouncer vivo pero con el backend caído, necesitas checks externos que ejecuten una consulta real y marquen el servidor vía la API de HAProxy.

Sea cual sea el mecanismo, recuerda el efecto multiplicador: cada instancia de PgBouncer abre su propio pool. Dos máquinas activas con `default_pool_size = 20` son hasta 40 conexiones reales; ajusta el presupuesto igual que hicimos en Kubernetes.

Endurecer la consola de administración

La consola de `SHOW POOLS` también ejecuta `PAUSE`, `SHUTDOWN` y `KILL`. En las manos equivocadas es un botón de apagado de toda tu capa de datos, así que trátala como una superficie de ataque:

```
; Quién puede administrar (PAUSE, RELOAD, SHUTDOWN...)
admin_users = pgbouncer_admin
; Quién puede solo mirar (SHOW POOLS, SHOW STATS...)
stats_users = pgbouncer_monitor
; Escucha solo en la red interna, nunca 0.0.0.0 si hay
; segmentación disponible
listen_addr = 10.0.1.10
```

Separa los dos roles: el exporter de Prometheus solo necesita `stats_users`, y tu automatización de failover solo necesita `admin_users`. Ninguno de los dos debería ser un usuario real de la aplicación. Ejecuta el proceso como usuario sin privilegios (los paquetes ya lo hacen), monta la configuración como solo lectura, y si usas contenedores, que el sistema de archivos raíz sea `readOnlyRootFilesystem` con un volumen efímero para los sockets. Y dado que las versiones recientes han corregido CVEs precisamente en autenticación y comandos de administración, mantener el binario actualizado es parte del endurecimiento, no un extra.

Último consejo operativo: guarda la configuración de PgBouncer en el mismo repositorio y pipeline que el resto de tu infraestructura. Un `RELOAD` aplicado a mano en una máquina y olvidado en otra produce los bugs más difíciles de diagnosticar de esta lista: dos poolers idénticos a la vista, comportándose distinto bajo carga.

Lo que el pooling cambia en el propio PostgreSQL

Instalar un pooler no es solo añadir una pieza delante de la base de datos; también cambia cómo conviene configurar PostgreSQL. Tres ajustes se benefician directamente.

`max_connections` puede bajar, y debería. Sin pooler, la gente lo sube a 500 o 1000 "por si acaso", y cada hueco de conexión reserva estructuras internas aunque no se use. Con PgBouncer delante, un `max_connections = 100` cubre el pool (40), el pool administrativo, las réplicas de streaming y un margen de emergencia. Deja siempre `superuser_reserved_connections` (3 por defecto) intacto: es tu puerta de entrada garantizada cuando todo lo demás está saturado, precisamente el momento en que más la necesitas.

`work_mem` puede subir. Este es el beneficio que casi nadie menciona. `work_mem` es memoria por operación de ordenación o hash, y su techo seguro se calcula multiplicando por el número de conexiones potencialmente activas. Con 500 conexiones posibles, un `work_mem` de 64 MB es una bomba de relojería (500 × 64 MB de exposición); con 40 conexiones controladas por el pooler, es un presupuesto razonable que convierte ordenaciones en disco en ordenaciones en memoria. Menos conexiones no solo significa menos cambios de contexto: significa que cada consulta puede disponer de más recursos con seguridad.

Los límites por rol se vuelven útiles. `ALTER ROLE app_worker CONNECTION LIMIT 10` era papel mojado cuando cada pod abría conexiones a voluntad; con el pooler como único cliente real, los límites por rol en PostgreSQL funcionan como una segunda línea de defensa coherente con los `pool_size` por usuario de PgBouncer. Cinturón y tirantes.

Atrapa los bugs de modo transaction en CI, no en producción

Los errores de estado de sesión (el `SET` que se filtra, el prepared statement fantasma, la tabla temporal desaparecida) tienen una propiedad traicionera: no aparecen en desarrollo, porque en local te conectas directo a PostgreSQL. Aparecen en producción, bajo carga, de forma intermitente. La solución es hacer que tu entorno de pruebas atraviese un PgBouncer en modo transaction igual que producción:

```
# docker-compose.test.yml
services:
  postgres:
    image: postgres:17
    environment:
      POSTGRES_PASSWORD: test
      POSTGRES_DB: app_test

  pgbouncer:
    image: bitnami/pgbouncer:1.25.2
    environment:
      POSTGRESQL_HOST: postgres
      POSTGRESQL_PASSWORD: test
      PGBOUNCER_DATABASE: app_test
      PGBOUNCER_POOL_MODE: transaction
      PGBOUNCER_MAX_PREPARED_STATEMENTS: "100"
    ports:
      - "6432:6432"
    depends_on:
      - postgres
```

Y en CI, la suite de tests se conecta al puerto 6432, no al 5432. El efecto es inmediato: cualquier dependencia oculta de estado de sesión que tu código haya adquirido falla en el pipeline con un stack trace legible, en lugar de fallar un martes a las once de la noche con tráfico real. Es una de esas inversiones de una hora que se amortizan con el primer bug capturado.

Un matiz para no pasarse de listo: mantén también una ejecución de tests directa contra PostgreSQL (por ejemplo, la de las migraciones), porque hay operaciones legítimas que nunca pasarán por el pooler en producción y no quieres falsos negativos. El objetivo no es que todo pase por PgBouncer, sino que cada camino de producción tenga su gemelo en CI.

¿Y si mi aplicación es pequeña? Cuándo no necesitas nada de esto

Cerremos con honestidad ingenieril: no toda aplicación necesita un pooler externo. Si tienes una sola instancia de aplicación, o unas pocas, y la suma de sus pools locales queda cómodamente por debajo del óptimo de tu servidor (núcleos × 2), el pool del framework es suficiente y PgBouncer solo añadiría una pieza que operar. Las señales de que ha llegado el momento son concretas: la suma de conexiones potenciales supera con claridad el óptimo del servidor, tienes procesos efímeros o serverless que abren conexiones sin control, los picos de deploy o autoescalado provocan errores de `too many clients`, o la memoria de PostgreSQL se va en backends ociosos. Hasta entonces, la mejor arquitectura es la que tiene menos piezas; a partir de ahí, este artículo te espera.

Preguntas frecuentes

¿Puedo usar modos distintos para bases de datos distintas en el mismo PgBouncer? Sí. `pool_mode` se puede fijar por entrada en `[databases]` (y `pool_size` por usuario en `[users]`). Es la forma idiomática de convivir transaction para OLTP con session para tareas administrativas.

¿PgBouncer en la máquina de la base de datos o en máquinas propias? Para empezar, junto a la base de datos está bien (latencia mínima, una máquina menos). A escala, sepáralo: quieres poder reiniciar, escalar y actualizar el pooler sin tocar la máquina de PostgreSQL, y viceversa.

¿Necesito PgBouncer si ya uso RDS Proxy o el pooler de mi plataforma? No, salvo casos muy concretos. Apilar poolers añade latencia y duplica los puntos donde razonar sobre estado de sesión. Elige una capa y conócela bien.

¿Por qué veo menos conexiones en PostgreSQL que clientes en PgBouncer? ¿Está roto? Al contrario: está funcionando. Esa asimetría (miles de clientes, decenas de backends) es exactamente el multiplexado que buscabas.

¿Cómo detecto que alguien se conecta saltándose el pooler? Restringe `pg_hba.conf` para que solo acepte conexiones desde las IPs de PgBouncer (más las administrativas), y audita `pg_stat_activity.client_addr`: cualquier dirección inesperada es alguien puenteando el pool.

¿Qué pasa con las conexiones cuando PgBouncer se reinicia? Con un reinicio inmediato, los clientes reciben error y deben reconectar (por eso `pool_pre_ping` y reintentos con backoff en la aplicación no son opcionales). Con el apagado seguro de SIGTERM (1.23+) y varias instancias, los reinicios planificados pasan desapercibidos.

¿El pooler afecta a los planes de consulta o a los índices? No. PgBouncer no interpreta SQL (y por eso es tan rápido); PostgreSQL ve las mismas consultas de siempre. Todo tu conocimiento de EXPLAIN, índices y estadísticas sigue aplicando sin cambios.

Con esto, el cuadro completo: por qué las conexiones son caras, cómo multiplexa un pooler, cómo se configura, autentica, cifra, despliega, opera, observa y mide, y cuándo cambiar de herramienta. Pocas piezas de infraestructura devuelven tanto por tan poco: instala el pooler antes de necesitarlo, y esa clase de incidentes simplemente dejará de existir en tu sistema.

PART 2 · ENGLISH

Connection Pooling in PostgreSQL: PgBouncer, Pooling Modes and Optimal Pool Size

Why PostgreSQL connections are expensive

PostgreSQL uses a process-per-connection model: every client that connects makes the server spawn a dedicated backend process with its own memory and internal structures. That design is robust, but it comes at a price. Opening a connection involves a TCP handshake, authentication and a process fork; keeping it open consumes several megabytes even when idle; and with thousands of backends, the OS scheduler and PostgreSQL's internal locks start to suffer.

That's why `max_connections` is not a number you can raise carelessly. A server with 8 cores performs better with 30 active connections than with 500: past a certain point, adding connections adds no throughput — only context switches and contention. The classic symptom is a database that degrades under load even though the CPU isn't saturated.

The real problem: many instances, each with its own pool

Your framework's connection pool (SQLAlchemy, HikariCP, the pg pool in Node) solves the problem within one instance: it reuses connections instead of opening one per request. But in production you rarely run a single instance.

Do the math: 40 Kubernetes pods × 20 connections per pool = 800 connections against a PostgreSQL that performs best with 40. Every deploy, every autoscaling event and every Celery worker adds more. A local pool can't coordinate with the pools in other instances; you need a central point that multiplexes.

What an external pooler does

PgBouncer is a lightweight proxy that sits between your applications and PostgreSQL. Applications open thousands of cheap connections to PgBouncer, and PgBouncer serves them with a handful of real connections to the server. The key is the pooling mode, which defines when a server connection becomes free for another client.

The three pooling modes

Session (the default)

The server connection is assigned to the client until it disconnects. It's transparent (everything works exactly as without a pooler), but it barely multiplexes: if your clients hold connections open, you gain little. Useful as a first step, or when you depend on session state.

Transaction (the one you want in production)

The server connection is only borrowed for the duration of a transaction. Between transactions, that backend serves other clients. This is where real multiplexing happens: thousands of clients over dozens of connections. In exchange, anything that relies on session state becomes unreliable, because your next transaction may run on a different backend:

- `SET` for session variables (use `SET LOCAL` inside the transaction)
- Temporary tables you expect to reuse across transactions
- Session-level advisory locks (use the `pg_advisory_xact_lock` variants)
- `LISTEN` / `NOTIFY`
- `WITH HOLD` cursors

Statement

The connection is released after each statement; multi-statement transactions are forbidden. It's a mode for very specific cases (aggressive sharding, 100% autocommit workloads). If you're unsure, it's not this one.

Prepared statements in transaction mode (PgBouncer ≥ 1.21)

For years, the big catch of transaction mode was prepared statements: they are prepared on one specific backend, and if your next request lands on a different backend, PostgreSQL replies with the infamous *prepared statement does not exist*. The historical workaround was disabling the driver's prepared statement cache, losing performance.

Since version 1.21, PgBouncer tracks extended-protocol prepared statements and transparently re-prepares them on whichever backend handles the request. You enable it with a single parameter:

```
; 0 = disabled (default). A value > 0 turns the support on
; and defines how many prepared statements are kept in an LRU
; cache per server connection.
max_prepared_statements = 200
```

Set the value above the number of statements your application commonly uses. The higher it is, the more memory each connection consumes (in PgBouncer and in PostgreSQL), so don't set 10,000 just in case. Note:

this covers protocol-level prepared statements (the ones drivers use); manual SQL `PREPARE` is still unsupported in transaction mode.

Production-ready configuration

```
[databases]
app = host=10.0.0.5 port=5432 dbname=app

[pgbouncer]
listen_addr = 0.0.0.0
listen_port = 6432
auth_type = scram-sha-256
auth_file = /etc/pgbouncer/userlist.txt

pool_mode = transaction

; Real connections per user+database pair
default_pool_size = 20
; Extra connections if the pool is exhausted and a client
; has been waiting longer than reserve_pool_timeout seconds
reserve_pool_size = 5
reserve_pool_timeout = 3

; Clients PgBouncer accepts (cheap — these are not backends)
max_client_conn = 2000

; Prepared statements in transaction mode (1.21+)
max_prepared_statements = 200

; Server connection hygiene
server_idle_timeout = 300
server_lifetime = 1800
tcp_keepalive = 1
```

Two parameters people confuse: `max_client_conn` is how many clients can connect to PgBouncer (be generous — they're cheap sockets); `default_pool_size` is how many real connections it opens to PostgreSQL (keep it small — they're expensive processes). `server_lifetime` recycles connections periodically, which prevents eternal backend processes with fragmented memory.

How big should the pool be?

Smaller than you think. A reasonable starting point for mixed workloads is:

```
connections ≈ (cores × 2) + effective_spindle_count
```

For an 8-core server on SSDs, that's ~20 active connections. It may sound low for "thousands of users", but remember: each connection is only busy while it executes a transaction, which usually lasts milliseconds. It's better for clients to wait a few milliseconds in PgBouncer's queue (`cl_waiting`) than to have 300 backends fighting over 8 cores.

The right number is measured, not calculated: raise or lower `default_pool_size` while watching your application's throughput and p99 latency. If increasing the pool no longer improves latency, you've already gone too far.

Configuring your application (Python)

Point the driver at PgBouncer (port 6432) and shrink the local pool: its job is no longer multiplexing, just amortizing socket creation.

```
import asyncpg

pool = await asyncpg.create_pool(
    dsn="postgresql://app_user:secret@pgbouncer:6432/app",
    min_size=2,
    max_size=10,
    # Only if your PgBouncer is < 1.21 or you haven't
    # enabled max_prepared_statements:
    # statement_cache_size=0,
)
```

```
from sqlalchemy.ext.asyncio import create_async_engine

engine = create_async_engine(
    "postgresql+asyncpg://app_user:secret@pgbouncer:6432/app",
    pool_size=5,
    max_overflow=5,
    # Detects dead connections after pooler restarts
    pool_pre_ping=True,
)
```

With `max_prepared_statements` enabled in PgBouncer you can leave the driver's statement cache at its defaults and keep its performance. If you can't upgrade PgBouncer, disable it (`statement_cache_size=0` in asyncpg) or you'll see intermittent prepared statement errors under load.

Monitoring: SHOW POOLS is your friend

PgBouncer exposes an admin console on a pseudo-database called `pgbouncer`:

```
psql -h pgbouncer -p 6432 -U admin_user pgbouncer

SHOW POOLS;
-- database | user | cl_active | cl_waiting | sv_active | sv_idle | maxwait
SHOW STATS;
-- requests, bytes and average query/transaction times
```

The columns that matter: `cl_waiting` (clients queued right now) and `maxwait` (how many seconds the oldest client has been waiting). A `cl_waiting` that oscillates between 0 and a few is healthy; a steadily growing

`maxwait` means the pool is too small or long transactions are hogging connections. Alert on `maxwait`, not on the number of clients.

Common mistakes

Using SET in transaction mode. The variable sticks to a random backend and "leaks" to other clients. Use `SET LOCAL` or per-query parameters.

Oversizing the pool. Doubling `default_pool_size` during a latency spike usually makes it worse: more backends competing for the same cores.

Ignoring that PgBouncer is single-threaded. One instance saturates a core at tens of thousands of QPS. The standard fix is running several processes with `so_reuseport = 1` on the same port.

Long transactions. In transaction mode, a 30-second transaction hijacks a real connection for 30 seconds. Watch *idle in transaction* and set `idle_in_transaction_session_timeout` in PostgreSQL.

Health checks that open transactions. A `SELECT 1` in autocommit is enough; don't wrap the health check in a transaction with retries.

PgBouncer versus the alternatives

PgBouncer (written in C, in production since 2007) remains the default choice: mature, with a tiny memory footprint and prepared statement support since 1.21. But it's no longer alone. PgCat, written in Rust, is multi-threaded by default and adds primary/replica load balancing and sharding support; if a single PgBouncer instance runs out of CPU, it's the natural candidate. Supavisor (Elixir, built by Supabase) is designed for massive multi-tenancy, with per-tenant pool isolation. And if you're in the cloud, consider the managed options: RDS Proxy on AWS or the built-in PgBouncer in Azure Database for PostgreSQL Flexible Server save you from operating the pooler yourself.

Production checklist

- `pool_mode = transaction` and code free of session state
- `max_prepared_statements` > your app's commonly used statements (PgBouncer ≥ 1.21)
- Small `default_pool_size` (start at cores × 2), tuned under real load
- Generous `max_client_conn`; shrunken app-side pools (5–10 per instance)
- Alerts on `maxwait` and on *idle in transaction* in PostgreSQL
- `server_lifetime` configured to recycle connections
- Multiple PgBouncer instances (`so_reuseport`) as traffic grows

A well-sized pooler is one of the best effort-to-benefit improvements available for PostgreSQL: one afternoon of work that turns "the database can't take more connections" into a permanently solved problem.

Authentication in production: userlist.txt, auth_query and SCRAM

The configuration example above uses `auth_file`, and for a database with two or three users that's fine. But a static `userlist.txt` has a serious operational problem: every time someone rotates a password in PostgreSQL, you have to regenerate the file and reload it on every PgBouncer instance. In larger teams that ends in out-of-sync credentials and authentication errors at three in the morning.

The alternative is `auth_query`: instead of reading passwords from a file, PgBouncer asks PostgreSQL for them directly whenever a client tries to authenticate. All you need is a technical user (the `auth_user`) with permission to run that query:

```
[pgbouncer]
auth_type = scram-sha-256
auth_user = pgbouncer_auth
auth_query = SELECT usename, passwd FROM user_lookup($1)
```

And in PostgreSQL, a `SECURITY DEFINER` function that limits what that technical user can see. Don't give it direct access to `pg_shadow`; expose only what's strictly necessary:

```
CREATE OR REPLACE FUNCTION user_lookup(p_user text)
RETURNS TABLE (usename name, passwd text)
LANGUAGE sql SECURITY DEFINER SET search_path = pg_catalog AS $$
  SELECT usename, passwd
  FROM pg_shadow
  WHERE usename = p_user
    AND usename NOT IN ('postgres', 'pgbouncer_auth');
$$;

REVOKE ALL ON FUNCTION user_lookup(text) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION user_lookup(text) TO pgbouncer_auth;
```

With this, password rotation becomes transparent: you change the password in PostgreSQL with `ALTER ROLE ... PASSWORD` and PgBouncer picks it up on the next login attempt, without touching any file. The function also explicitly excludes the superuser and the technical user itself, so nobody can authenticate as them through the pooler.

On the method: use `scram-sha-256` whenever you can. SCRAM support in PgBouncer is mature, and since 1.25 its performance improved substantially (the SCRAM handshake is CPU-expensive, and it showed during mass reconnection spikes). If you're coming from an old installation with `md5`, migrate: PostgreSQL considers it deprecated and md5 passwords will disappear in future releases. Since 1.25 there is also native LDAP authentication, useful if your organization centralizes credentials in Active Directory or OpenLDAP, though it adds an external dependency on the critical path of every login: if LDAP goes down, nobody new connects.

One detail that surprises people: in `auth_query` mode, the authentication query runs through a connection from the corresponding database's pool. If your pool is saturated, even logins can queue up. One more reason to size with headroom and watch `maxwait`.

TLS: encryption between the app, PgBouncer and PostgreSQL

PgBouncer sits on two distinct network legs, and each is encrypted separately: the client → PgBouncer leg (`client_tls_*` parameters) and the PgBouncer → PostgreSQL leg (`server_tls_*` parameters). It is perfectly possible — and common — to encrypt only one of the two, for example when PgBouncer runs on the same machine as the application but the database lives on another network.

```
; Client -> PgBouncer leg
client_tls_sslmode = require
client_tls_cert_file = /etc/pgbouncer/tls/server.crt
client_tls_key_file = /etc/pgbouncer/tls/server.key

; PgBouncer -> PostgreSQL leg
server_tls_sslmode = verify-full
server_tls_ca_file = /etc/pgbouncer/tls/root.crt
```

The asymmetry is deliberate: towards the client, `require` is enough (the client already trusts the infrastructure where PgBouncer runs), but towards the server you want `verify-full`, which validates both the certificate and the host name. Without it, an attacker with a network position could impersonate PostgreSQL and PgBouncer would hand over the credentials without complaint.

Two performance considerations. First: the TLS handshake is paid when opening a connection, not per query; since PgBouncer keeps few, long-lived connections to the server, the cost of `server_tls` is negligible. The client leg is different: if your applications constantly open and close connections, every open pays TCP handshake + TLS + authentication, and that does show. The right answer is not to drop TLS but to keep a small local pool in the application (as we saw above) to amortize the opens. Second: since 1.25 PgBouncer accepts "direct" TLS connections (the client starts the TLS handshake without PostgreSQL's protocol-level negotiation first), which saves a round-trip on every new connection and makes it easier to sit behind TCP load balancers that expect plain TLS.

And an operational warning: certificates expire. PgBouncer reloads certificates with a simple `RELOAD`, so integrate renewal (Let's Encrypt, cert-manager, Vault) with a verification `SHOW VERSION` and a subsequent `RELOAD`. An expired certificate on the pooler is a total outage: nobody connects.

PgBouncer on Kubernetes: sidecar or centralized service?

On Kubernetes there are two reasonable ways to deploy PgBouncer, and choosing wrong cancels much of the benefit.

Sidecar: a PgBouncer container inside every application pod. The added latency is minimal (localhost) and the configuration travels with the pod. But beware: each sidecar keeps its own pool against PostgreSQL. With 40 pods and `default_pool_size = 20` you're back to 800 real connections: exactly the problem you wanted to solve. The sidecar only makes sense with tiny pools (2–5 connections) or as a buffering layer in front of a central pooler.

Centralized service: a Deployment of 2–3 PgBouncer replicas behind a ClusterIP Service. All applications point at the Service, and the total number of real connections to PostgreSQL is `replicas × default_pool_size` — a number you control in a single place. It's the topology that truly multiplexes, and the one I recommend by default:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pgbouncer
spec:
  replicas: 2
  selector:
    matchLabels: { app: pgbouncer }
  template:
    metadata:
      labels: { app: pgbouncer }
    spec:
      containers:
        - name: pgbouncer
          image: bitnami/pgbouncer:1.25.2
          ports:
            - containerPort: 6432
          resources:
            requests: { cpu: "500m", memory: "128Mi" }
            limits: { memory: "256Mi" }
          readinessProbe:
            tcpSocket: { port: 6432 }
            periodSeconds: 5
          livenessProbe:
            tcpSocket: { port: 6432 }
            initialDelaySeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: pgbouncer
spec:
  selector: { app: pgbouncer }
  ports:
    - port: 6432
      targetPort: 6432
```

Details that matter in this manifest. No CPU limit (`limits.cpu` absent): PgBouncer is single-threaded and latency-sensitive; CFS throttling hurts it badly, and it's better to let it use whatever core time it needs. A memory limit, yes, because its consumption is small and predictable. TCP probes rather than SQL-query probes: a health check that runs queries consumes pool connections and can cause exactly the saturation it's trying to detect.

Add a `PodDisruptionBudget` with `minAvailable: 1` so node drains can't take down all replicas at once, and spread replicas across zones with `topologySpreadConstraints`. Remember too that each replica has its own independent pool: two replicas with `default_pool_size = 20` mean up to 40 real connections. Set the per-replica size by dividing your total connection budget by the number of replicas.

What about operators? If you use CloudNativePG, Crunchy PGO or Zalando's operator, they all ship PgBouncer integration (CloudNativePG has the `Pooler` CRD, for instance). Use it: it solves certificates, secrets and restarts coordinated with the cluster's lifecycle for you.

Zero-downtime operations: RELOAD, PAUSE/RESUME and rolling restarts

The admin console isn't just for reading statistics; it is the operations tool. The three commands you'll actually use:

```
-- Apply configuration changes without restarting
RELOAD;

-- Let in-flight transactions finish and hold
-- new queries in the queue (clients see no error)
PAUSE;

-- Resume the held traffic
RESUME;
```

`RELOAD` re-reads the INI file and applies most parameters live: pool sizes, timeouts, TLS certificates. It's the reason you almost never need to restart PgBouncer for a configuration change.

The `PAUSE` / `RESUME` pair is gold for PostgreSQL maintenance. The sequence for a failover or a minor upgrade: `PAUSE` (PgBouncer waits for active transactions to finish and queues new work), you run the switchover or restart PostgreSQL, `RESUME`. If the window lasts a few seconds, your applications only perceive a latency blip, not a shower of connection errors. It's the difference between "invisible maintenance" and an incident page.

For upgrading PgBouncer itself, since version 1.23 signal behavior is designed for rolling restarts: `SIGTERM` no longer kills the process immediately; it starts a "super safe" shutdown that stops accepting new clients and waits for existing ones to disconnect. The classic immediate shutdown is now `SIGQUIT`. If you had a Dockerfile or a systemd unit relying on the old SIGTERM behavior, review it when upgrading. The rolling-restart procedure with several instances on `so_reuseport`: start the new instance, send SIGTERM to the old one, and the kernel steers new connections to the surviving instance while the old one drains.

That leaves failover of PostgreSQL itself. If the primary changes IP behind a DNS name (typical with Patroni + internal DNS, or with managed endpoints in the cloud), PgBouncer re-resolves the name honoring `dns_max_ttl` (15 seconds by default). Lower it if your automated failover is faster than that. Combine it with `server_fast_close = 1` so connections to the old server are closed as soon as they're released after a `RELOAD` or a destination change, rather than draining slowly. And on the application side, `pool_pre_ping` (SQLAlchemy) or your driver's equivalent detects the dead connections that survived the switch.

What PgBouncer 1.25 brings (and why you should upgrade)

This article assumes a modern PgBouncer, and it's worth spelling out what changed recently. The 1.25 series (the current one; 1.25.2 shipped in May 2026) brings four things that matter in production:

- **Native LDAP authentication**, for organizations that centralize credentials outside PostgreSQL.
- **Direct TLS connections** on the client side: fewer round-trips when connecting and compatibility with load balancers that expect standard TLS.
- **Much faster SCRAM**: the CPU cost of the authentication handshake dropped substantially, which you notice during reconnection storms (mass deploys, fleet restarts).
- **Idle clients reported as `idle`** instead of `active`, so `SHOW CLIENTS` and derived metrics finally distinguish who is doing work from who is merely holding a socket open.

On top of that, 1.25.2 fixes several vulnerabilities related to authentication and admin commands. A pooler is a critical security component — it sees all your credentials and all your SQL traffic — so treat it as such: subscribe to release announcements and upgrade with the same discipline you'd apply to OpenSSL or PostgreSQL itself. With rolling restarts, upgrading no longer has an operational excuse.

Serious observability: pgbouncer_exporter and Prometheus

Running `SHOW POOLS` by hand is fine for debugging, but in production you want time series and alerts. The standard path is the Prometheus community's `pgbouncer_exporter`: it connects to the `pgbouncer` pseudo-database, runs the `SHOW` commands and exposes the result as metrics. Deploy it as a sidecar of the pooler (here the sidecar does make sense: it's read-only) and point Prometheus at the exporter's port.

The signals that actually predict problems, in order of importance:

- **Waiting clients** (`cl_waiting` column, metric family `pgbouncer_pools_client_waiting_connections`): how many requests are queued right now waiting for a real connection.
- **Maximum wait time** (`maxwait`): how long the oldest client has been waiting. This is your primary alerting metric.
- **Pool saturation**: `sv_active / (sv_active + sv_idle)`. At a sustained 100%, everything else degrades.
- **Login errors and rejected connections**, which reveal credential problems or a short `max_client_conn`.

A reasonable PromQL alert (tune the threshold to your latency SLO):

```
- alert: PgBouncerClientsWaiting
  expr: max_over_time(pgbouncer_pools_client_maxwait_seconds[5m]) > 1
  for: 5m
  labels: { severity: warning }
  annotations:
    summary: "Clients waiting >1s for a pool connection"
```

```
    description: "Pool {{ $labels.database }}/{{ $labels.user }} has had waits for 5
minutes. Check for long transactions before growing the pool."
```

Note the annotation: the correct reaction to waits is not blindly growing the pool, but first checking whether long transactions are hijacking connections. For that, correlate with PostgreSQL: `pg_stat_activity` tells you what each backend is doing, and these two queries solve the mystery more often than any other:

```
-- Oldest open transactions (prime suspects for hogging the pool)
SELECT pid, usename, state,
       now() - xact_start AS xact_age,
       left(query, 80) AS query
FROM pg_stat_activity
WHERE xact_start IS NOT NULL
ORDER BY xact_start
LIMIT 10;

-- How many backends per state?
SELECT state, count(*)
FROM pg_stat_activity
GROUP BY state;
```

If `maxwait` climbs and at the same time you see transactions with an `xact_age` of minutes, the problem is the application (or a migration forgotten in a terminal), not the pool size. If `maxwait` climbs with short transactions and PostgreSQL's CPU has headroom, then yes: raise `default_pool_size` one step and measure again.

Complete the picture with logs: `log_connections = 0` and `log_disconnections = 0` in production (at high QPS they generate expensive noise), but keep `log_pooler_errors = 1`, which is where rejections and timeouts show up with their cause.

Measure your pool with pgbench

Everything above is heuristics; measurement is the truth. `pgbench`, which ships with PostgreSQL, is perfectly suited to compare "direct to PostgreSQL" against "through PgBouncer" and to find the pool size where your hardware saturates.

First, the case that favors the pooler the most: new connections per transaction (the `-C` flag), which simulates applications without a local pool, scripts, serverless functions:

```
# Initialize test data (scale 50 ≈ 5M rows)
pgbench -i -s 50 -h db.internal -p 5432 app

# Direct to PostgreSQL, opening a connection per transaction
pgbench -C -c 50 -j 4 -T 60 -h db.internal -p 5432 app

# Same thing, through PgBouncer
pgbench -C -c 50 -j 4 -T 60 -h pgbouncer.internal -p 6432 app
```

With `-C`, the difference is usually dramatic — several times more transactions per second via PgBouncer — because the dominant cost is opening connections, which is precisely what the pooler amortizes. Exact numbers depend on your hardware and network: treat any published figure (including the ones in this article) as indicative and measure in your own environment.

Second, the saturation curve. With persistent connections (no `-C`), raise concurrency in steps and record throughput and latency:

```
for c in 10 20 40 80 160 320; do
  pgbench -c $c -j 8 -T 30 -P 10 \
    -h pgbouncer.internal -p 6432 app \
    | grep -E "tps|latency average"
done
```

You'll see the classic pattern: throughput grows up to the zone of a well-sized pool, flattens, and beyond that only latency grows. That knee is your number. Repeat the sweep with two or three values of `default_pool_size` (say 15, 20 and 30 for 8 cores) and keep the one that maximizes throughput with p99 within your target. Two methodological warnings: run pgbench from a machine other than the database (otherwise they compete for CPU and the result lies), and run each measurement at least twice; the first run warms the caches.

Beyond Python: Node, Prisma, Django and Go

The pattern is identical across ecosystems — point at port 6432, shrink the local pool, watch out for prepared statements — but each stack has its nuance.

Node.js (node-postgres). The `pg` `Pool` doesn't use named prepared statements by default (it sends parameterized queries over the extended protocol without caching them), so it works well in transaction mode without touching anything:

```
import pg from "pg";

const pool = new pg.Pool({
  host: "pgbouncer.internal",
  port: 6432,
  database: "app",
  user: "app_user",
  password: process.env.PGPASSWORD,
  max: 10,                    // small local pool
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 5000,
});
```

Just avoid queries with an explicit `name:` (that does create a named prepared statement) unless your PgBouncer has `max_prepared_statements` enabled.

Prisma. Prisma uses prepared statements internally, and it knows it: add `?pgbouncer=true` to the URL so it disables them, and define `directUrl` so migrations go straight to PostgreSQL (migrations use session-level advisory locks and DDL you don't want going through transaction mode):

```
datasource db {
  provider  = "postgresql"
  url       = env("DATABASE_URL")    // ...@pgbouncer:6432/app?pgbouncer=true
  directUrl = env("DIRECT_URL")      // ...@db:5432/app
}
```

Django. Two settings. `CONN_MAX_AGE = 0` (the default) makes Django open and close a connection per request, which against PgBouncer is cheap and fits transaction mode well. And `DISABLE_SERVER_SIDE_CURSORS = True`, because the server-side cursors Django uses in `QuerySet.iterator()` depend on session state:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "HOST": "pgbouncer.internal",
        "PORT": "6432",
        "NAME": "app",
        "CONN_MAX_AGE": 0,
        "DISABLE_SERVER_SIDE_CURSORS": True,
    }
}
```

Go (pgx). The `pgxpool` pool caches prepared statements by default. With a modern PgBouncer and `max_prepared_statements` enabled you can leave it as is; otherwise, switch the execution mode to the simple protocol:

```
cfg, _ := pgxpool.ParseConfig(
    "postgres://app_user@pgbouncer.internal:6432/app")
cfg.MaxConns = 10
// Only if PgBouncer < 1.21 or without max_prepared_statements:
// cfg.ConnConfig.DefaultQueryExecMode = pgx.QueryExecModeSimpleProtocol
pool, err := pgxpool.NewWithConfig(ctx, cfg)
```

The cross-cutting rule: if you see intermittent *prepared statement does not exist* or *bind message* errors under load, in any language, the cause is almost always the same — named prepared statements crossing transaction mode without support enabled — and the fixes are the usual two: enable `max_prepared_statements` in PgBouncer or disable the driver's cache.

Edge cases: migrations, COPY, LISTEN/NOTIFY and long-running jobs

Some operations don't fit transaction mode, and the professional solution is not to twist them until they pass, but to give them a separate path.

Schema migrations. Tools like Alembic, Flyway or Django/Prisma migrations use session-level advisory locks to guarantee only one instance migrates at a time. In transaction mode that lock can end up on a different backend than the one running the DDL, with baffling results. Run migrations against PostgreSQL directly (port 5432), or define an extra entry in PgBouncer with `pool_mode = session`:

```
[databases]
app = host=10.0.0.5 dbname=app
; Same database, session pool for administrative tasks
app_admin = host=10.0.0.5 dbname=app pool_mode=session pool_size=3
```

The per-database override is one of PgBouncer's most useful and least known features: different modes and pool sizes against the same server, selectable just by changing the database name in the job's connection string.

LISTEN/NOTIFY. Doesn't work in transaction mode (the subscription lives on one specific backend). If your application uses it — cache invalidation, lightweight queues — open a single dedicated connection straight to PostgreSQL for it, outside the pooler, and keep it alive with automatic reconnection. One extra persistent connection doesn't ruin the budget; hundreds do.

COPY and bulk loads. `COPY` works in transaction mode (it's a statement), but a twenty-minute load monopolizes a real connection the whole time. Same for long analytical transactions: send them through the administrative session pool or straight to the server, and keep the transaction pool for the OLTP traffic it exists for.

Defensive timeouts. In PostgreSQL, set `idle_in_transaction_session_timeout` (say, 60 seconds) to kill abandoned transactions, and a sensible global `statement_timeout` with per-role overrides for heavy jobs. In PgBouncer, `query_wait_timeout` (120 seconds by default) decides how long a client may wait in the queue before getting an error; lower it if you prefer failing fast and retrying. These limits turn "everything is stuck" incidents into localized errors with an identifiable culprit in the logs.

Worked example: sizing a 60-pod SaaS

Let's land all of this with numbers. Suppose a SaaS shaped like this: 60 API pods on Kubernetes, 8 Celery workers for background jobs, a 16-core PostgreSQL primary on NVMe SSDs, peaks of 4,000 requests per second, and OLTP transactions lasting 2–8 ms.

Step 1: real-connection budget. With 16 cores and SSDs, the guideline formula gives `16 × 2 + 1 ≈ 33`. We round to 40 as the total ceiling of desired active connections, and keep headroom in `max_connections` (say 100) for superusers, replicas and emergencies.

Step 2: does the throughput add up? Sanity check: 40 connections × (1000 ms / 5 ms per transaction) = 8,000 theoretical transactions per second, double the expected peak. Good. If this arithmetic didn't work out, no pooling configuration would fix it: you'd need to optimize queries or scale hardware.

Step 3: topology. Centralized pooler, 2 PgBouncer replicas (high availability plus CPU headroom). Per-replica budget: `default_pool_size = 18` (2 × 18 = 36 connections, leaving room for the administrative pool), `reserve_pool_size = 4` with `reserve_pool_timeout = 2` for short spikes.

Step 4: separate workloads. The Celery workers run longer transactions (50–500 ms). If they share a pool with the API, a batch of heavy jobs can induce waits on interactive requests. The solution: different users (`app_api` and `app_worker`), because PgBouncer creates pools per user+database pair, each with its own limits:

```
[databases]
app = host=10.0.0.5 dbname=app

[users]
app_api = pool_size=14
app_worker = pool_size=4 pool_mode=transaction
```

This way the workers can saturate their 4 connections without stealing a single one from the API. It's the same isolation principle you'd apply with separate queues in a messaging system.

Step 5: application side. Each API pod with a local pool of 5 (`pool_size=5, max_overflow=0` in SQLAlchemy): 60 × 5 = 300 client connections against PgBouncer, far below a `max_client_conn = 2000`. Each worker with 2. `pool_pre_ping` enabled everywhere.

Step 6: validate and watch. A pgbench sweep like the one in the previous section to confirm the saturation knee near 36–40 connections; alerts on `maxwait` > 1 s and on `idle in transaction` > 60 s. Review after a month of real data: if the API's p99 is comfortable and `cl_waiting` is almost always 0, you can even trim the pool and give RAM back to PostgreSQL.

The end result: 68 potentially noisy client processes turned into at most 40 orderly backends, with spikes absorbed by a queue measured in milliseconds. And every number in this recipe has a justification you can re-derive when your workload changes.

PgCat in depth: when to make the jump

I mentioned the alternatives above; let's dig into the decision, because "PgBouncer is falling short" means different things in different cases.

Signal 1: CPU. PgBouncer is single-threaded: one instance saturates a core at around tens of thousands of QPS (public benchmarks put it near 40–45k tps on modern hardware, degrading with hundreds of concurrent clients). The first answer is cheap: several instances with `so_reuseport`. If operating N processes still weighs on you, PgCat — multi-threaded on Rust's async runtime — delivers more than twice PgBouncer's throughput with hundreds of concurrent clients in independent benchmarks, with a single instance that uses every core.

Signal 2: replicas. If you want to spread reads across replicas, with PgBouncer you need logic in the application (two pools, one for writes and one for reads). PgCat load-balances across servers and can automatically exclude failed replicas. A minimal example of its configuration:

```
# pgcat.toml (simplified)
[pools.app]
pool_mode = "transaction"
default_role = "any"        # spread across primary and replicas
```

```
[pools.app.shards.0]
servers = [
  ["10.0.0.5", 5432, "primary"],
  ["10.0.0.6", 5432, "replica"],
  ["10.0.0.7", 5432, "replica"],
]
database = "app"

[pools.app.users.0]
username = "app_user"
pool_size = 40
```

PgCat can even route per statement (reads to replicas, writes to the primary) if you ask it to, though that mode demands a careful review of your application's transactions.

Signal 3: massive multi-tenancy. If you operate thousands of databases or tenants (a platform, a one-database-per-tenant SaaS), Supavisor — Supabase's Elixir pooler — is designed for that: it scales horizontally as a cluster and isolates pools per tenant. For a conventional SaaS with one database, it's more machinery than you need.

Signal 4: you don't want to operate the pooler. On managed cloud, the balance shifts. RDS Proxy (AWS) integrates with IAM and Secrets Manager and multiplexes well, with one important nuance: *pinning*. When a session uses stateful features (certain SETs, prepared statements depending on configuration, temporary tables), the proxy "pins" that connection to the client and stops multiplexing it; the symptom is the database connection count climbing until it resembles the client count, and the pinned-connections metric confirms it. The same code discipline that PgBouncer's transaction mode demands also minimizes pinning. Azure Database for PostgreSQL Flexible Server ships built-in PgBouncer you can enable with a switch (port 6432), and both Neon and Supabase expose endpoints with pooling built in — on Neon, by adding the `-pooler` suffix to the host. For serverless workloads (Lambda, functions scaling to hundreds of instances), some managed or built-in pooler isn't optional: it's the only realistic defense against connection storms.

The practical recommendation doesn't change: start with PgBouncer (or the pooler your platform gives you for free), instrument it, and migrate only when one of these four signals shows up in your metrics — not in a benchmark article.

Troubleshooting guide: the errors you'll see and what they mean

When something fails through a pooler, the error message often arrives scrambled: sometimes PgBouncer emits it, sometimes PostgreSQL does, and sometimes the driver translates them its own way. This is the list of the ones that come up again and again, with their real cause.

`no more connections allowed (max_client_conn)`. PgBouncer rejects new clients because you hit `max_client_conn`. It's almost always a connection leak in the application (local pools growing without bounds, zombie processes) rather than a badly chosen limit. Before raising the number, run `SHOW CLIENTS` and check which IP addresses the crowd is coming from.

`query_wait_timeout`. The client waited in the queue longer than allowed and PgBouncer evicted it. It's the "healthy" error: it means the system preferred failing fast over hanging. Investigate why the pool was busy that long (long transactions, traffic spike, short pool) instead of just raising the timeout.

`unsupported startup parameter: options` (or `search_path`, or `application_name` on old versions). Some drivers and ORMs send extra parameters at connection startup that PgBouncer doesn't recognize by default. The fix is declaring them harmless:

```
; Accept and forward non-critical startup parameters
ignore_startup_parameters = extra_float_digits,options
```

It is probably the most-searched configuration line in PgBouncer's issue tracker: JDBC sends `extra_float_digits`, some frameworks send `options=-c search_path=...`, and without this line the connection dies before it's born.

`server login has been failing` / `login failed`. PgBouncer can't authenticate against PostgreSQL with the credentials from `auth_file` or `auth_query`. It usually happens after rotating passwords with a static userlist, or when the `auth_user` lost permissions on the lookup function. Meanwhile, clients see timeouts, not password errors, which is misleading: the real failure is on the pooler → server leg.

`idle_in_transaction_session_timeout` **killing pooler connections**. If you enable that timeout in PostgreSQL (recommended), remember that in transaction mode a pool connection can legitimately sit `idle` between loans. The PostgreSQL timeout that matters here is *idle in transaction*, not plain *idle*; don't configure an aggressive `idle_session_timeout` on the server or it will kill healthy pool connections and you'll see constant reconnections in PgBouncer's logs.

Latency appearing out of nowhere with the pool "empty". Check `min_pool_size`: it defaults to 0, so after a quiet period the pool is cold and the first burst of traffic pays for opening real connections (plus the TLS handshake and authentication). A `min_pool_size` of half the pool keeps connections warm and eliminates that first spike:

```
; Keep at least 10 connections open even with no traffic
min_pool_size = 10
```

The timeout family, organized

PgBouncer has more than a dozen timeouts and the documentation lists them alphabetically, which is the worst way to understand them. Grouped by what they protect:

They protect the client from waiting forever:

- `query_wait_timeout` (120 s by default): maximum time queued waiting for a connection. The one that turns saturation into visible errors.
- `client_login_timeout` (60 s): maximum time to complete login. If the auth_query is slow, this is the one that fires.

They protect the server from careless clients:

- `client_idle_timeout` (0 = disabled): disconnects idle clients. Useful against connection leaks; set it above your application's keepalive if you enable it.
- `query_timeout` (0 = disabled): kills queries that exceed the limit. Better to leave it at 0 and use `statement_timeout` in PostgreSQL, which returns a clean SQL error instead of cutting the connection.

Pool hygiene:

- `server_idle_timeout` (600 s): closes server connections idle for too long, giving memory back to PostgreSQL.
- `server_lifetime` (3600 s): recycles connections even when healthy; prevents eternal backends.
- `server_connect_timeout` (15 s) and `server_login_retry` (15 s): how long to wait when opening a new connection, and how long before retrying after a failure. With fast failovers, lowering `server_login_retry` to 2–5 s speeds up recovery.

The general rule: PgBouncer's timeouts manage *connections*; PostgreSQL's (`statement_timeout`, `idle_in_transaction_session_timeout`, `lock_timeout`) manage *SQL*. Every problem has its layer, and using the wrong one produces cryptic errors.

High availability outside Kubernetes

If your infrastructure is virtual machines or bare metal, the equivalent of the replicated Deployment is: two or more PgBouncer instances on different machines, plus a mechanism that spreads or switches traffic between them.

The simplest and most robust option is **keepalived with a virtual IP**: the two machines share a VIP that only one owns at a time; if the active one dies, the VIP migrates in seconds and clients reconnect against the same address. It doesn't spread load (one instance sits idle), but for most workloads a single PgBouncer instance is plenty, and the operations are trivial.

If you need load spreading as well as failover, put **HAProxy in TCP mode** in front of the poolers:

```
listen pgbouncer
    bind *:6432
    mode tcp
    option tcp-check
    balance leastconn
    server pgb1 10.0.1.10:6432 check inter 2s fall 2
    server pgb2 10.0.1.11:6432 check inter 2s fall 2
```

`mode tcp` is mandatory (HAProxy must not interpret PostgreSQL's protocol) and `leastconn` balances better than round-robin when connections have uneven lifetimes. The basic TCP health check detects dead processes; if you want to detect a PgBouncer that's alive but whose backend is down, you need external checks that run a real query and mark the server via HAProxy's API.

Whatever the mechanism, remember the multiplier effect: each PgBouncer instance opens its own pool. Two active machines with `default_pool_size = 20` mean up to 40 real connections; adjust the budget just as we did on Kubernetes.

Hardening the admin console

The console behind `SHOW POOLS` also runs `PAUSE`, `SHUTDOWN` and `KILL`. In the wrong hands it's a power-off button for your entire data layer, so treat it as an attack surface:

```
; Who can administer (PAUSE, RELOAD, SHUTDOWN...)
admin_users = pgbouncer_admin
; Who can only look (SHOW POOLS, SHOW STATS...)
stats_users = pgbouncer_monitor
; Listen only on the internal network — never 0.0.0.0
; when segmentation is available
listen_addr = 10.0.1.10
```

Separate the two roles: the Prometheus exporter only needs `stats_users`, and your failover automation only needs `admin_users`. Neither should be a real application user. Run the process as an unprivileged user (packages already do), mount the configuration read-only, and if you use containers, make the root filesystem `readOnlyRootFilesystem` with an ephemeral volume for sockets. And since recent releases have fixed CVEs precisely in authentication and admin commands, keeping the binary up to date is part of the hardening, not an extra.

One last operational tip: keep PgBouncer's configuration in the same repository and pipeline as the rest of your infrastructure. A `RELOAD` applied by hand on one machine and forgotten on another produces the hardest-to-diagnose bugs on this list: two poolers identical at first glance, behaving differently under load.

What pooling changes in PostgreSQL itself

Installing a pooler isn't just adding a piece in front of the database; it also changes how PostgreSQL itself should be configured. Three settings benefit directly.

`max_connections` can go down — and it should. Without a pooler, people raise it to 500 or 1000 "just in case", and every connection slot reserves internal structures even when unused. With PgBouncer in front, a `max_connections = 100` covers the pool (40), the administrative pool, streaming replicas and an emergency margin. Always leave `superuser_reserved_connections` (3 by default) intact: it's your guaranteed way in when everything else is saturated — precisely the moment you need it most.

`work_mem` can go up. This is the benefit almost nobody mentions. `work_mem` is memory per sort or hash operation, and its safe ceiling is computed by multiplying by the number of potentially active connections. With 500 possible connections, a `work_mem` of 64 MB is a time bomb (500 × 64 MB of exposure); with 40 connections controlled by the pooler, it's a reasonable budget that turns on-disk sorts into in-memory sorts. Fewer connections doesn't just mean fewer context switches: it means each query can safely be given more resources.

Per-role limits become useful. `ALTER ROLE app_worker CONNECTION LIMIT 10` was a dead letter when every pod opened connections at will; with the pooler as the only real client, per-role limits in PostgreSQL work as a second line of defense consistent with PgBouncer's per-user `pool_size`. Belt and suspenders.

Catch transaction-mode bugs in CI, not in production

Session-state bugs (the leaking `SET`, the phantom prepared statement, the vanished temporary table) have a treacherous property: they don't show up in development, because locally you connect straight to PostgreSQL. They show up in production, under load, intermittently. The solution is making your test environment go through a transaction-mode PgBouncer just like production:

```
# docker-compose.test.yml
services:
  postgres:
    image: postgres:17
    environment:
      POSTGRES_PASSWORD: test
      POSTGRES_DB: app_test

  pgbouncer:
    image: bitnami/pgbouncer:1.25.2
    environment:
      POSTGRESQL_HOST: postgres
      POSTGRESQL_PASSWORD: test
      PGBOUNCER_DATABASE: app_test
      PGBOUNCER_POOL_MODE: transaction
      PGBOUNCER_MAX_PREPARED_STATEMENTS: "100"
    ports:
      - "6432:6432"
    depends_on:
      - postgres
```

And in CI, the test suite connects to port 6432, not 5432. The effect is immediate: any hidden session-state dependency your code has acquired fails in the pipeline with a readable stack trace, instead of failing on a Tuesday at eleven at night with real traffic. It's one of those one-hour investments that pays for itself with the first bug it catches.

One nuance so you don't overdo it: also keep a test run that goes straight against PostgreSQL (the migrations run, for instance), because some legitimate operations will never go through the pooler in production and you don't want false negatives. The goal isn't for everything to go through PgBouncer, but for every production path to have its twin in CI.

What if my application is small? When you need none of this

Let's close with engineering honesty: not every application needs an external pooler. If you run a single application instance, or a handful, and the sum of their local pools sits comfortably below your server's optimum (cores × 2), the framework's pool is enough and PgBouncer would just add a piece to operate. The

signs that the moment has arrived are concrete: the sum of potential connections clearly exceeds the server's optimum, you have ephemeral or serverless processes opening connections without control, deploy or autoscaling spikes cause `too many clients` errors, or PostgreSQL's memory is eaten by idle backends. Until then, the best architecture is the one with fewer pieces; from that point on, this article will be waiting for you.

Frequently asked questions

Can I use different modes for different databases in the same PgBouncer? Yes. `pool_mode` can be set per entry in `[databases]` (and `pool_size` per user in `[users]`). It's the idiomatic way to run transaction mode for OLTP alongside session mode for administrative tasks.

PgBouncer on the database machine or on its own machines? To start, next to the database is fine (minimal latency, one machine fewer). At scale, separate it: you want to restart, scale and upgrade the pooler without touching the PostgreSQL machine, and vice versa.

Do I need PgBouncer if I already use RDS Proxy or my platform's pooler? No, except in very specific cases. Stacking poolers adds latency and doubles the places where you must reason about session state. Pick one layer and know it well.

Why do I see fewer connections in PostgreSQL than clients in PgBouncer? Is it broken? Quite the opposite: it's working. That asymmetry (thousands of clients, dozens of backends) is exactly the multiplexing you were after.

How do I detect someone connecting around the pooler? Restrict `pg_hba.conf` to accept connections only from PgBouncer's IPs (plus administrative ones), and audit `pg_stat_activity.client_addr`: any unexpected address is someone bypassing the pool.

What happens to connections when PgBouncer restarts? With an immediate restart, clients get errors and must reconnect (which is why `pool_pre_ping` and retries with backoff in the application are not optional). With SIGTERM's safe shutdown (1.23+) and several instances, planned restarts go unnoticed.

Does the pooler affect query plans or indexes? No. PgBouncer doesn't interpret SQL (which is why it's so fast); PostgreSQL sees the exact same queries as always. Everything you know about EXPLAIN, indexes and statistics still applies unchanged.

And with that, the full picture: why connections are expensive, how a pooler multiplexes, how to configure, authenticate, encrypt, deploy, operate, observe and measure it, and when to switch tools. Few pieces of infrastructure give back so much for so little: install the pooler before you need it, and that entire class of incidents simply stops existing in your system.