

# 서버리스 인스턴스 동결과 Postgres 풀러 클라이언트 슬롯 고갈: Next.js·Supabase 운영 사례와 로컬 재현

임재호  
NextgenAI  
khan@nextgenai.kr

## 초록

서버리스 함수는 응답 뒤 실행 환경이 얼어붙는다. 이때 앱이 트랜잭션 풀러에 연 커넥션이 어떻게 남는지는 잘 드러나지 않는다. 이 논문은 Next.js·Vercel·Supabase로 만든 사내 도구 zenith에서 2026년 7-9월에 반복된 커넥션 장애를 장애 기록 11건과 커밋 16건으로 재구성하고, 원인을 네 기제로 나눈다. 핵심 장애는 풀러 클라이언트 슬롯 200개의 고갈이었고, 풀 상한을 10에서 4로 낮춘 뒤에도 재발했다. 동결된 인스턴스에서는 유틸 타이머가 실행되지 않으므로, 점유가 요청을 한 번이라도 처리한 인스턴스 수  $W$ 에 비례한다는 모형  $W \cdot \min(q, m)$ 을 세웠다. 프로세스 정지 신호로 동결을, TCP 프록시로 슬롯 상한 40을 흉내 낸 재현 실험에서 타이머에만 맡긴 두 구성의 최대 점유는 모든 조건에서  $\min(W \cdot \min(q, m), 40)$ 과 같았고, 실패 인스턴스는 상한 4에서  $W = 16$ 일 때 6개였다. 풀 전체가 조용해진 뒤 소켓을 닫는 반납 구성은 인스턴스를 8개에서 16개로 늘려도 최대 점유가 32-36개로 상한 40 아래였고 실패가 없었다. 그 대가로 성공 요청의 지연 중앙값이 2.6-8.7ms 늘었다. 동결이 유틸 타이머보다 짧고 그동안 소켓이 끊긴 경우에는 세 방어 구성 중 쿼리 전 풀 교체만이 첫 요청과 재시도를 모두 성공시켰다.

**주제어:** 서버리스; 커넥션 풀; PostgreSQL; 트랜잭션 풀러; 장애 분석

## 1. 서론

서버리스(serverless) 함수는 요청이 올 때만 실행되고, 요청 사이에는 실행 환경이 얼어붙는다(freeze) [1]. 이 실행 모형은 데이터베이스처럼 오래 사는 상태를 가진 시스템과 잘 맞지 않는다는 지적이 있다 [2]. PostgreSQL을 쓰는 서버리스 앱은 보통 트랜잭션 풀러(transaction pooler)를 앞에 둔다. Supabase 문서는 짧게 사는 커넥션을 많이 여는 서버리스·엣지 함수에 트랜잭션 모드 풀러를 쓰라고 권한다 [3]. 풀러가 데이터베이스 백엔드 수를 줄여 주는 대신, 앱이 풀러에 연 클라이언트 커넥션의 수에는 별도 상한이 있다.

이 논문은 사내 업무 도구 zenith(Next.js·Vercel·Supabase)에서 2026년 7-9월에 반복된 커넥션 장애를 분석한다. 가장 큰 장애는 2026-09-13의 (EMAXCONN) max client connections reached, limit: 200이었다. 당시 Postgres의 활성 커넥션은 한가했고, 고갈된 것은 풀러의 클라이언트 슬롯이었다. 개발자는 먼저 인스턴스당 풀 상한(max)을 10에서 4로 낮췄지만 장애는 그대로 재발했다. 이 경험이 이 논문의 연구 질문을 만든다.

**연구 질문.** 응답 직후 얼어붙는 서버리스 인스턴스에서, 타이머 기반 커넥션 정리(idle\_timeout)와 풀 상한은 풀러 클라이언트 슬롯의 점유를 제한하는가? 제한하지 못한다면 어떤 반납 방식이 점유를 제한하며 그 비용은 얼마인가?

이 논문의 기여는 다음과 같다.

- 운영 기록 11건과 커밋 16건으로 장애 계보를 재구성한다. 원인을 네 가지 기제(얼어붙은 소켓의 슬롯 점유, 죽은 소켓 hang, 동시 요청 간 반납 간섭, 파이프라인 문장 유실)로 나눈다(절 3).
- 인스턴스 동결 아래의 슬롯 점유 모형을 세운다. 타이머 기반 정리에서는 점유가 활성 인스턴스가 아니라 한 번이라도 요청을 처리한 인스턴스 수에 비례한다는 예측을 낸다(절 4).

- 프로세스 정지 신호(SIGSTOP)로 동결을 흉내 내는 재현 환경을 만든다. 그 환경에서 예측과 세 가지 반납 구성의 차이를 측정한다(절 6).

## 2. 배경

### 2.1 서버리스 실행 환경의 동결

AWS Lambda 문서는 런타임과 확장이 모두 끝나고 대기 이벤트가 없으면 실행 환경을 열린다고 설명한다 [1]. 다음 호출이 오면 환경을 녹여 재사용한다. 이때 핸들러 밖에서 만든 데이터베이스 커넥션은 그대로 남아 다음 호출에서 다시 쓰인다 [1]. Next.js의 `after`는 응답 뒤에 실행할 작업을 예약한다. 서버리스 환경에서는 `waitUntil`이 그 작업이 끝날 때까지 호출의 수명을 늘린다 [4]. Vercel의 Fluid compute는 한 인스턴스가 여러 호출을 동시에 처리하게 한다 [5]. 따라서 한 인스턴스의 풀은 동시 요청이 함께 쓰는 공유 자원이다.

서버리스 플랫폼의 내부 동작은 측정 연구로 드러났다. Wang 등은 세 플랫폼에서 5만 개 이상의 함수 인스턴스를 띄워 격리 방식과 콜드 스타트를 측정했다 [6]. Shahrade 등은 Azure Functions의 실제 작업 부하에서 대부분의 함수가 드물게 호출된다고 보고했다 [7]. 이 연구들은 플랫폼의 자원 관리를 다룬다. 이 논문은 그 위에서 앱이 연 외부 커넥션이 동결과 어떻게 상호작용하는지를 다룬다.

### 2.2 postgres.js 풀과 트랜잭션 풀러

zenith는 Drizzle ORM 아래 postgres.js 3.4.9를 쓴다. postgres.js는 쿼리가 만들어질 때 커넥션을 지연 생성한다 [8]. 풀 상한 `max`의 기본값은 10이다. 유휴 커넥션은 `idle_timeout`초 뒤 닫힌다 [8]. 이 정리는 이벤트 루프의 타이머가 실행해야 일어난다.

Supabase의 트랜잭션 모드 풀러(Supavisor, 포트 6543)는 준비된 문장(prepared statement)을 지원하지 않는다 [3]. postgres.js 이슈 #970에는 트랜잭션 풀러를 거칠 때 풀이 멈추는 보고가 모여 있다 [9]. 댓글 작성자들은 원인을 한 TCP 커넥션에 여러 쿼리를 이어 보내는 파이프라이닝으로 지목했다. 우회책 `max_pipeline: 0`은 `sql.begin`을 깨뜨린다는 보고도 있다. 이 내용은 사용자 보고이며 관리자가 확인하지 않았다 [9]. Postgres 자체의 커넥션 한도 초과는 `SQLSTATE 53300 (too_many_connections)`으로 보고된다 [10].

## 3. 사례: zenith의 커넥션 장애 계보

### 3.1 시스템과 자료

zenith는 Vercel에 배포한 Next.js 16 앱이다. 데이터는 Supabase Postgres에 있고 앱은 Supavisor 트랜잭션 모드로 접속한다(그림 1). 분석 자료는 저장소에 남은 장애 기록(ai-docs/lessons) 11건, DB 계층 파일의 커밋 16건, 현재 코드 `src/data/db.ts`다. 장애 기록에 적힌 운영 수치(Supavisor 로그 집계, `pg_stat_activity`)는 저자가 장애 당시 직접 조회해 기록한 값이다. 이 논문을 쓰며 원본 로그를 다시 조회하지는 않았다.

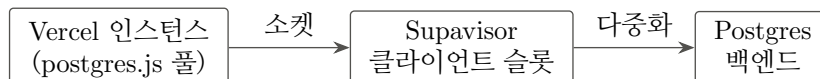


그림 1. zenith의 데이터베이스 경로. 고갈된 자원은 가운데 칸의 클라이언트 슬롯이다. 오른쪽 Postgres 백엔드 수는 풀러가 다중화하므로 앱의 소켓 수와 다르다.

### 3.2 장애와 대응의 순서

표 1은 장애 기록과 커밋을 시간순으로 정리한다. 증상은 네 기제로 묶인다.

**기제 1: 얼어붙은 소켓의 슬롯 점유.** 09-13 기록에 따르면 상류 Postgres 커넥션은 49개였고 모두 유휴 상태였다. Supabase 대시보드에는 마지막 쿼리가 같은 `insert`이고 유휴 시간이 같은(2분 45초)

표 1. zenith 커넥션 장애 계보(2026년, 장애 기록과 커밋에서 정리)

| 날짜       | 증상(기록)                                | 대응(커밋)                                |
|----------|---------------------------------------|---------------------------------------|
| 07-01    | 로컬 53300. 강제 종료된 개발 서버가 남긴 유휴 커넥션 95개 | 수동 정리(개발 환경)                          |
| 07-14-15 | 무한 로딩. 죽은 소켓에서 쿼리가 응답 없이 대기           | 쿼리 JS 타임아웃 12s(37922c0)               |
| 07-19    | 타임아웃 뒤에도 무한 로딩. hang 소켓이 풀 슬롯을 점유     | 세션 쿠키 캐시, 타임아웃 시 풀 교체, 8s(5f040d8)    |
| 08-06    | 쿼리 한 건만 8s 대기. 동결 중 idle_timeout 미발화  | 20s 이상 조용하면 쿼리 전 풀 교체 (8ad5c5d)       |
| 09-13    | EMAXCONN, 한도 200                      | max 10→4, idle_timeout 20→5s(657d192) |
| 09-13    | 같은 날 재발                               | 응답 뒤 반납(ced817e)                      |
| 09-13    | 동시 요청의 쿼리가 끊김                         | 풀이 조용해진 뒤 반납(bd54483)                 |
| 09-14    | 큐 포화를 죽은 소켓으로 오판                      | 성공 기록 지점 확대(eaef1ce)                  |
| 09-15    | 늦게 출발한 쿼리 한 건 8s 대기                   | 파이프라이닝 끄 + 드라이버 패치(6816311)           |

커넥션 수집 개가 있었다. 개발자는 26분 창 의 Supervisor 로그에서 `Connection authenticated 1,483건`과 `Terminate received from client 957건`을 켜다. 차이 526건은 분당 약 20개의 소켓이 닫히지 않고 남았다는 뜻이다. 200 슬롯은 약 10분이면 찬다. 로그에는 6-7분 주기의 톱니 모양이 있었다. max를 4로 낮춘 뒤에도 같은 장애가 났다.

**기제 2: 죽은 소켓 hang.** 07-15와 08-06 기록은 동결 중 풀러가 끊은 소켓을 깨어난 풀이 재 사용해 쿼리가 응답 없이 기다린 사례다. `connect_timeout`은 이미 열린 소켓에 작동하지 않는다. `statement_timeout`은 서버가 쿼리를 받아야 작동한다. 그래서 대응은 앱 수준 타임아웃, 풀 교체, 쿼리 전 교체 순으로 진화했다.

**기제 3: 동시 요청 간 반납 간섭.** 응답 뒤 풀을 닫는 첫 반납 구현(ced817e)은 누수를 멈췄다. 기록상 06:10-06:40 사이 인증 1,095건, 종료 1,226건, EMAXCONN 0건이었다. 그러나 같은 인스턴스의 다른 요청이 쓰던 쿼리가 끊겨 /dashboard가 실패했다. 반납 시점을 “진행 중 쿼리 0, 마지막 활동 뒤 300ms”로 바꾼 구현이 bd54483이다.

**기제 4: 파이프라인 문장 유실.** 풀의 커넥션 4개가 모두 바쁘면 postgres.js는 다음 쿼리를 응답 대기 중인 소켓에 이어 쓴다. 09-15 기록은 트랜잭션 풀러를 거치며 이 문장의 응답이 오지 않은 사례다. 이 현상은 이슈 #970의 보고와 일치한다 [9]. 개발자는 TCP 프록시로 `pg_sleep(0.2)` 4건의 서버 도착 시각을 켜다. 기본값에서는 0, 0, 0, 0ms(한 번에 전송), `max_pipeline: 0`에서는 0, 203, 405, 608ms(응답 뒤 차례로 전송)였다.

#### 4. 인스턴스 동결 아래의 슬롯 점유 모형

기제 1을 정량화한다. 인스턴스  $i$ 가 요청 하나에서 동시에  $q$ 개의 읽기를 실행하고 풀 상한이  $m$ 이면, 지연 생성 풀은  $c = \min(q, m)$ 개의 소켓을 연다 [8].

**가정 1.** 인스턴스는 응답과 `after` 작업이 끝나면 곧바로 얼어붙는다. 얼어 있는 동안 타이머는 실행되지 않는다. 동결까지 걸리는 시간은 `idle_timeout`보다 짧다.

**가정 2.** 풀러는 앱이 연 클라이언트 소켓마다 슬롯 하나를 쓰며, 동시에 열린 슬롯 수가  $L$ 을 넘는 접속을 거절한다. 풀러는 실행 시간 동안 유휴 클라이언트를 회수하지 않는다.

타이머에만 맡기면, 가정 1에 따라 응답 직후 얼어붙은 인스턴스의 소켓은 닫히지 않는다. 그 인스턴스가 다시 깨어나도 풀은 같은 소켓을 재사용하므로 소켓 수는 늘지 않는다. 따라서 관측 구간에서 요청을 한 번이라도 처리한 인스턴스 수를  $W$ 라 하면 점유는 다음과 같다.

$$O_{\text{timer}} = W \cdot \min(q, m) \quad (1)$$

$O_{\text{timer}} > L$ 이면 가정 2에 따라 새 소켓이 필요한 요청이 거절된다. 식 (1)에는 요청률이 없다. 점유는 동시에 일하는 인스턴스가 아니라 일한 적이 있는 인스턴스의 수로 정해진다.  $m$ 을 낮추면 기울기만 줄어든다. 09-13 기록의 “max는 채워지는 속도만 바꾼다”는 관찰이 이 식의 정성적 형태다.

반납 방식에서는 얼기 전에 소켓을 모두 닫는다. 얼어 있는 인스턴스의 점유는 0이고, 점유는 요청 처리 중이거나 반납을 기다리는 인스턴스 수  $A$ 에만 묶인다.

$$O_{\text{release}} \leq A \cdot \min(q, m) \quad (2)$$

$A$ 는 요청률과 요청 하나의 체류 시간(처리 시간 + 반납 대기)으로 정해진다. 두 식은 추론이며, 절 6에서 측정으로 확인한다.

## 5. 재현 실험 방법

### 5.1 환경

실험은 Apple M3 Pro(12코어, 36 GB), macOS 26.6.2 한 대에서 실행했다. 런타임은 Bun 1.4.2, 드라이버는 zenith와 같은 postgres.js 3.4.9다. 데이터베이스는 실험 전용 PostgreSQL 17.10 클러스터(max\_connections=300)다. 코드와 원자료는 bench/ 폴더에 있다.

### 5.2 동결과 풀러의 모사

그림 2는 실험 구성이다. 각 인스턴스는 별도 Bun 프로세스이며 postgres.js 풀 하나를 가진다. 컨트롤러는 요청을 보내기 직전에 SIGCONT로 인스턴스를 깨우고, 응답과 응답 뒤 작업이 끝나면 곧바로 SIGSTOP으로 멈춘다(가정 1). 멈춘 프로세스의 소켓은 커널이 열어 둔 채로 유지한다.

인스턴스와 Postgres 사이에는 직접 만든 TCP 프록시를 둔다. 프록시는 열린 클라이언트 소켓 수가  $L = 40$ 에 이르면 새 접속에 SQLSTATE 53300과 (EMAXCONN) max client connections reached 메시지를 보내고 닫는다. 프록시는 접속 수락 수와 종료 수를 세어 Supervisor 로그의 인증·종료 집계와 같은 지표를 낸다. E2에서는 열린 소켓을 FIN 없이 버리는 동작(blackhole)도 흉내 낸다. 바이트를 더 전달하지 않고 소켓은 열어 둔다.

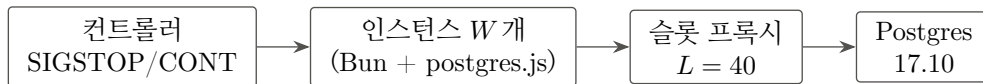


**그림 2.** 재현 실험 구성. 프록시는 Supervisor의 클라이언트 슬롯 상한과 인증·종료 집계만 본뜨며, 백엔드 다중화는 하지 않는다.

### 5.3 실험 설계

**E1(슬롯 점유).** 요청 하나는 pg\_sleep(0.02) 읽기  $q = 6$ 건을 동시에 실행한다. 20 ms는 장애 기록에 적힌 앱 쿼리 평균(10–30 ms)의 중간값이다. 컨트롤러는 50 ms마다 쉬고 있는 인스턴스 하나를 균등 무작위로 골라 요청을 보낸다. 쉬는 인스턴스가 없으면 그 차례를 건너뛴다. 인스턴스 수  $W \in \{4, 8, 12, 16\}$ , 각 조건 30 s, 시드 3개로 반복한다. 비교 구성은 zenith의 시기별 설정 셋이다.

- **A:** max=10, idle\_timeout=20 s, 타이머에만 맡김(09-13 이전).
- **B:** max=4, idle\_timeout=5 s, 타이머에만 맡김(657d192).

- **C**: B에 반납을 더함. 진행 중 쿼리가 0이고 마지막 활동 뒤 300 ms가 지나면 폴을 닫고 소켓이 없는 새 폴로 바꾼다(bd54483). 동결은 반납이 끝난 뒤에 한다.

지표는 프록시의 최대 열린 소켓 수, 수락·종료·거절 수, 실패 요청 비율, 한 요청도 성공하지 못한 인스턴스 수, 성공 요청의 지연(요청 안 쿼리 지연의 최댓값)이다.

**E2(죽은 소켓)**. 인스턴스 하나( $q = 4$ ,  $\max=4$ ,  $\text{idle\_timeout}=5\text{s}$ )가 요청을 처리해 소켓 4개를 연 뒤 멈춘다. 프록시가 그 소켓을 blackhole 처리한다. 간격  $G \in \{2, 8\}\text{s}$  뒤 인스턴스를 깨워 같은 요청을 보내고, 곧바로 재시도 요청을 보낸다. 응답이 20s 안에 없으면 hang으로 기록한다. 방어 구성은 셋이다. M0은 방어 없음, M1은 쿼리마다 8s JS 타임아웃(37922c0), M2는 M1에 더해 쿼리 전 폴 교체(8ad5c5d)다. M2의 교체 임계는 원본 20s 대신 2s로 줄였다. 조건마다 5회 반복한다.

**E3(반납 간섭)**. zenith 저장소의 검증 스크립트 `verify:db-release`를 실험 클러스터에 대고 3회 실행한다. 이 스크립트는 응답 즉시 반납(ced817e)과 조용해진 뒤 반납(bd54483)을 같은 동시 부하에서 비교한다.

## 6. 결과

### 6.1 E1: 타이머 기반 정리에서 점유는 인스턴스 수를 따른다

표 2은 E1의 결과다. A와 B에서 최대 열린 소켓 수는 모든 조건에서  $\min(W \cdot \min(q, m), L)$ 과 같았다. 예를 들어 A는  $W = 4$ 에서 24개, B는  $W = 8$ 에서 32개를 열었다. 예측값이  $L = 40$ 을 넘는 조건(A의  $W \geq 8$ , B의  $W \geq 12$ )에서는 점유가 40에서 멈추고 새 접속이 거절됐다. A에서는 30s 동안 동결된 인스턴스의 소켓이 하나도 닫히지 않았다(종료 0).

**표 2.** E1 결과(구성·인스턴스 수별 시드 3회, 각 30s). 예측은 식 (1)의  $W \cdot \min(q, m)$ . 범위는 시드 3회의 최솟값-최댓값이며 하나만 적은 칸은 세 값이 같다. 실패 인스턴스는 요청을 받았으나 한 번도 성공하지 못한 인스턴스 수다.

| 구성 | $W$ | 예측 | 최대 열린 소켓 | 종료 소켓     | 실패 인스턴스 | 실패 요청(%) | 성공 p50(ms) |
|----|-----|----|----------|-----------|---------|----------|------------|
| A  | 4   | 24 | 24       | 0         | 0       | 0.0      | 23.3–23.4  |
| A  | 8   | 48 | 40       | 0         | 2       | 2.7–2.9  | 23.3–23.4  |
| A  | 12  | 72 | 40       | 0         | 6       | 6.1–6.2  | 23.4       |
| A  | 16  | 96 | 40       | 0         | 10      | 9.6      | 23.4–23.6  |
| B  | 4   | 16 | 16       | 0         | 0       | 0.0      | 46.5–47.4  |
| B  | 8   | 32 | 32       | 0         | 0       | 0.0      | 47.5       |
| B  | 12  | 48 | 40       | 0–14      | 2       | 1.0      | 47.5–47.7  |
| B  | 16  | 64 | 40       | 0–47      | 6       | 3.1–3.2  | 47.4–47.6  |
| C  | 4   | –  | 16       | 1164–1168 | 0       | 0.0      | 62.7–63.3  |
| C  | 8   | –  | 32       | 2360–2376 | 0       | 0.0      | 50.1–51.4  |
| C  | 12  | –  | 32       | 2364–2372 | 0       | 0.0      | 50.6–51.2  |
| C  | 16  | –  | 32       | 2336–2356 | 0       | 0.0      | 51.1–56.1  |

실패는 소켓을 얻지 못한 인스턴스에 몰렸다. 실패 인스턴스가 생긴 모든 조건에서 그 수는  $W - \lfloor L / \min(q, m) \rfloor$ 와 같았다. A는 소켓 6개를 다 얻은 인스턴스 6개만 성공했고(A,  $W = 16$ 에서 실패 10개), B는 4개를 다 얻은 인스턴스 10개만 성공했다(B,  $W = 16$ 에서 실패 6개). 남은 슬롯 4개를 얻은 A의 인스턴스 하나도 6개가 필요해 계속 실패했다. 실패 요청 비율(최대 9.6%)은 실패 인스턴스 비율보다 낮다. 실패한 요청의 일부가 오류 대신 응답 없이 멈춰, 컨트롤러가 그 인스턴스에 요청을 덜 보냈기 때문이다.  $W = 16$ 에서 15s 안에 응답이 없던 요청은 A가 9–13건, B가 9–12건이었다. B의 실패 요청은 중앙값이 15s 상한이었다. 이 대기의 원인은 이 실험에서 밝히지 않았다.

그림 3은 표 2의 최대 열린 소켓과 실패 인스턴스를  $W$ 에 대해 그린다. A와 B의 점유는  $W$ 를 따라 올라가  $L$ 에 닿는다. C는  $W \geq 8$ 에서 32에 머문다. 같은 설계로 먼저 돌린 1차 스왑(지표 추가

전)에서는 C,  $W = 16$ 의 한 실행이 36이었다. C의 점유는 반납 대기 중인 인스턴스 수로 정해진다. 요청 하나가 약 0.35s(쿼리와 300 ms 무활동 대기) 동안 인스턴스를 붙잡고 요청이 50ms마다 오므로, 동시에 붙잡힌 인스턴스는 약 7-8개로 추정된다(직접 세지 않았다). 여기에  $\min(q, m) = 4$ 를 곱하면 식 (2)의 상한 32와 맞는다. C의  $W = 4$ 는 인스턴스가 모자라 요청의 약 절반(평균 292건)만 보냈으므로 지연 비교에서 제외한다.

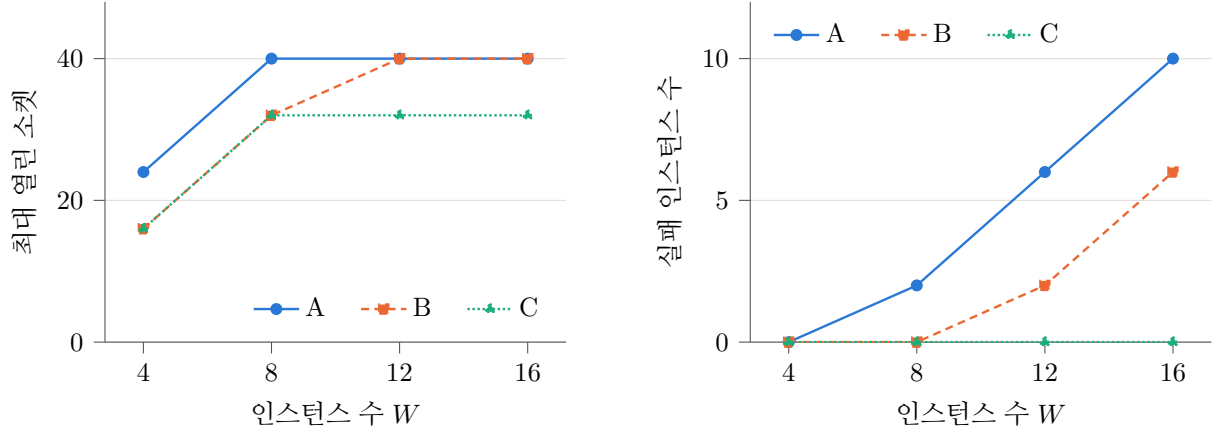


**그림 3.** E1의 최대 열린 소켓(왼쪽)과 실패 인스턴스 수(오른쪽). 슬롯 상한  $L = 40$ . 시드 3회 모두 같은 값이다(표 2).

B에서는  $W \geq 12$ 일 때 소켓 일부가 닫혔다(종료 0-47). 원인을 가르려고 B에서 `idle_timeout` 만 600s로 늘린 구성을  $W = 16$ , 같은 시드 3개로 다시 돌렸다. 종료 소켓은 세 번 모두 0이었고 실패 인스턴스는 6개, 실패 요청은 3.0-3.2%였다. 따라서 B의 종료는 5s보다 오래 멈췄던 인스턴스가 깨어날 때 밀린 `idle_timeout` 타이머가 발화한 결과다. 이 회수는 그 인스턴스가 다시 요청을 받을 때만 일어난다.

반납(C)은 점유 대신 연결 수립 비용을 치렀다. C는  $W \geq 8$ 에서 30s마다 소켓 2,336-2,376 개를 열고 닫았다(요청당 약 4개). 성공 요청의 p50은 B보다 2.6-8.7 ms 길었다(B 47.4-47.7 ms 대 C 50.1-56.1 ms,  $W = 8-16$ , 같은  $W$ 끼리 비교). 이 차이는 같은 기기 안의 루프백 연결 비용이며, 원격 풀러에서는 왕복 시간과 TLS 수립만큼 커진다.

40s 시계열 실행( $W = 16$ )에서 A는 시작 502 ms 만에, B는 751 ms 만에 슬롯 40개를 채웠다. C는 실행 내내 최대 32개였고 소켓 3,128개를 열어 3,124개를 닫았다. 실험은 50ms마다 요청을 보내 모든 인스턴스를 1s 안에 한 번씩 깨운다. 그래서 운영 기록의 “약 10분”보다 훨씬 빨리 찬다.

## 6.2 E2: 죽은 소켓 hang과 방어 구성

표 3는 E2의 결과다. 동결이 `idle_timeout`보다 짧을 때( $G = 2s$ ) 방어가 없는 M0은 5회 모두 첫 요청과 재시도가 20s 상한까지 응답하지 않았다. JS 타임아웃만 둔 M1은 5회 모두 첫 요청과 재시도가 8.0s에 실패했다. 재시도도 같은 죽은 소켓을 다시 잡았기 때문이다. 이는 07-19 기록의 “다시 시도도 같은 죽은 풀을 재히트”를 재현한다. 쿼리 전에 풀을 바꾸는 M2는 5회 모두 첫 요청이 39-45 ms에 성공했다.

동결이 `idle_timeout`보다 길 때( $G = 8s$ )는 세 구성 모두 성공했다. 깨어난 직후 밀린 `idle_timeout` 타이머가 죽은 소켓을 닫았고, 첫 요청이 새 소켓을 열었다. 첫 요청 지연(35-45 ms)이 재시도(22-24 ms)보다 긴 이유는 이 연결 수립이다.

## 6.3 E3: 요청 경계 반납과 조용해진 뒤 반납

zenith의 `verify:db-release`는 3회 모두 전 항목을 통과했다. 세 실행에서 같은 결과가 나왔다. 응답 즉시 반납(옛 방식)에서는 다른 요청의 대기 쿼리 8건 중 4건이 `CONNECTION_DESTROYED`로, 유예를 넘는 트랜잭션이 `CONNECTION_CLOSED`로, 교체 전에 만든 빌더가 `CONNECTION_ENDED`로

**표 3.** E2 결과(조건마다 5회). 성공은 요청 안 쿼리 4건이 모두 성공한 횟수, 지연은 요청 안 쿼리 지연 최댓값의 범위다. hang은 20s 상한까지 응답이 없었음을 뜻한다.

| 구성         | G (s) | 첫 요청             | 재시도              |
|------------|-------|------------------|------------------|
| M0 방어 없음   | 2     | 0/5 성공, hang     | 0/5 성공, hang     |
| M1 타임아웃 8s | 2     | 0/5 성공, 8.0s 실패  | 0/5 성공, 8.0s 실패  |
| M2 쿼리 전 교체 | 2     | 5/5 성공, 39–45 ms | 5/5 성공, 22–23 ms |
| M0 방어 없음   | 8     | 5/5 성공, 35–43 ms | 5/5 성공, 23–24 ms |
| M1 타임아웃 8s | 8     | 5/5 성공, 36–44 ms | 5/5 성공, 23–24 ms |
| M2 쿼리 전 교체 | 8     | 5/5 성공, 39–44 ms | 5/5 성공, 24 ms    |

실패했다. 조용해진 뒤 반납(현재 방식)에서는 같은 부하의 쿼리 8건과 트랜잭션이 모두 성공했다. 반납은 8.39–8.40s 뒤에 일어났고 반납 뒤 남은 앱 소켓은 0개였다.

처음 실행은 zenith 저장소에 의존성이 설치되지 않은 상태였다. Bun이 패치되지 않은 postgres.js 3.4.9를 자동으로 받아 실행했고, 트랜잭션 항목이 UNSAFE\_TRANSACTION으로 실패했다. 패치를 적용해 설치한 사본에서 다시 실행한 결과가 위의 3회다. 이 실패는 09-15 기록의 “max\_pipeline: 0은 드라이버 패치와 한 몸”이라는 경고를 그대로 보여 준다.

## 7. 논의

### 7.1 해석

E1은 식 (1)의 예측과 일치했다. 타이머 기반 정리에서 슬롯 점유는 동시에 일하는 인스턴스가 아니라 일한 적이 있는 인스턴스의 수를 따른다. 따라서 max를 10에서 4로 낮춘 조치(A→B)는 슬롯 40개를 채우는 데 필요한 인스턴스 수를 7개에서 10개로 늘렸을 뿐이다. 운영에서는 인스턴스가 계속 새로 생기므로 이 차이는 시간 차이로 나타난다. 09-13 기록에서 max를 낮춘 뒤에도 같은 날 장애가 재발한 것과 맞다. 반면 반납(C)은 점유를 요청률과 체류 시간의 함수로 바꿨다. 인스턴스가 늘어도 점유는 늘지 않았다.

E1과 E2는 idle\_timeout이 “동결된 인스턴스가 다시 깨어날 때”만 일한다는 점도 보여 준다. 깨어날 기회가 없는 인스턴스의 소켓은 남는다. 이는 09-13 기록의 2분 45초째 유휴 상태인 커넥션 무리와 같은 모양이다. 깨어난 인스턴스에서는 동결 시간이 idle\_timeout보다 짧을 때만 죽은 소켓 hang이 났다(표 3).

### 7.2 운영 진단에 주는 함의

세 가지를 제안한다. 첫째, 슬롯 고갈이 의심되면 풀러 로그의 인증 수와 종료 수의 차이를 먼저 센다. 그 차이가 0보다 크면 누수이고 상한 조정으로는 끝나지 않는다. E1의 A와 B에서 종료(0–47개)는 수락(16–55개)보다 적었고, C에서는 수락과 종료의 차이가 4개였다 (마지막 인스턴스의 반납 진행분으로 추정). 둘째, pg\_stat\_activity는 풀러 뒤의 백엔드만 보여 주므로 이 문제를 드러내지 못한다. 09-13 기록에서도 백엔드 49개가 모두 유휴였다. 셋째, 반납 장치는 공유 풀의 활동으로 시점을 정해야 한다. E3에서 응답 즉시 반납은 같은 인스턴스의 다른 요청 쿼리 절반을 끊었다.

### 7.3 비용과 트레이드오프

반납은 공짜가 아니다. E1에서 요청마다 소켓을 다시 열어 성공 지연 p50이 2.6–8.7ms 늘었다. E3에서 조용해지기를 기다린 시간은 8.4s였다. 플랫폼은 after 작업이 끝날 때까지 호출의 수명을 늘린다 [4]. 그래서 이 대기 시간은 인스턴스 활성화 시간으로 청구될 수 있다. zenith의 09-23 기록은 반납 대기가 함수 비용을 키운 경로로 지목된 사례다. 이 논문은 비용을 측정하지 않았다. [실험 결과 미제공]

## 7.4 한계

- 동결을 SIGSTOP으로 흉내 냈다. 프로세스 정지 중에도 벽시계는 흐르므로 깨어난 직후 밀린 타이머가 발화한다. 마이크로VM 단위로 멈추는 실제 플랫폼에서 시계와 타이머가 어떻게 동작하는지는 확인하지 않았다. [검증 필요]
- 프록시는 클라이언트 슬롯 상한만 본뜨고 Supervisor의 백엔드 다중화와 유틸 클라이언트 회수는 본뜨지 않는다. 기록의 6-7분 주기 톱니는 재현 범위 밖이다.
- 인스턴스 수  $W$ 를 고정했다. 운영에서는 인스턴스가 생겼다 사라지므로 실패 인스턴스의 비율과 고갈까지 걸리는 시간은 다를 수 있다.
- 기제 4(파이프라인 문장 유실)는 Supervisor 없이 재현할 수 없어 운영 기록만으로 서술했다.
- 슬롯 고갈 상태에서 일부 실패 요청이 오류 대신 15s 이상 멈춘 원인은 규명하지 않았다.
- 모든 측정은 한 기기의 루프백에서 했다. 지연 값은 원격 폴러의 값이 아니다.
- 운영 수치(로그 집계, 커넥션 수)는 장애 당시 기록을 인용했으며 원본 로그를 다시 조회하지 않았다.

## 8. 결론

연구 질문에 대한 답은 다음과 같다. 응답 직후 얼어붙는 서버리스 인스턴스에서 타이머 기반 정리와 풀 상한은 폴러 클라이언트 슬롯의 점유를 제한하지 못했다. 재현 실험에서 점유는  $\min(W \cdot \min(q, m), L)$ 을 따랐고, 상한을 낮추면 고갈 시점만 늦춰졌다(표 2). 풀 전체가 조용해진 뒤 소켓을 닫는 반납은 인스턴스를 8개에서 16개로 늘려도 최대 점유를 32-36개로 유지했다. 그 대가는 요청당 연결 수립(루프백에서 p50 2.6-8.7ms 증가)과 반납 대기 시간이다. 죽은 소켓 hang에는 쿼리 전 풀 교체가 첫 요청과 재시도를 모두 살렸다(표 3). 후속 과제는 실제 플랫폼에서 동결 중 타이머 동작을 확인하는 일과 반납 대기의 비용을 측정하는 일이다.

## 참고문헌

- [1] Amazon Web Services. Understanding the Lambda execution environment lifecycle. <https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html>. Accessed 2026-10-11.
- [2] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *Conference on Innovative Data Systems Research (CIDR)*, 2019. URL <https://arxiv.org/abs/1812.03651>.
- [3] Supabase. Connect to your database. <https://supabase.com/docs/guides/database/connecting-to-postgres>. Accessed 2026-10-11.
- [4] Vercel. after — Next.js API reference. <https://nextjs.org/docs/app/api-reference/functions/after>, 2026. Accessed 2026-10-11 (docs version 16.4.0).
- [5] Vercel. Fluid compute. <https://vercel.com/docs/fluid-compute>, 2026. Accessed 2026-10-11 (page last updated 2026-08-24).
- [6] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133-146, 2018. ISBN 978-1-939133-01-4. URL <https://www.usenix.org/conference/atc18/presentation/wang-liang>.

- [7] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020. ISBN 978-1-939133-14-4. URL <https://www.usenix.org/conference/atc20/presentation/shahrad>.
- [8] porsager (GitHub). postgres.js: PostgreSQL client for Node.js and Deno (README, The Connection Pool). <https://github.com/porsager/postgres>. Accessed 2026-10-11.
- [9] nikhilro. Deadlocked/hung pool when connected to Postgres through Supavisor (porsager/postgres issue #970). <https://github.com/porsager/postgres/issues/970>, 2024. Opened 2024-10-27. Accessed 2026-10-11.
- [10] The PostgreSQL Global Development Group. PostgreSQL 17 documentation, appendix A. PostgreSQL error codes. <https://www.postgresql.org/docs/17/errcodes-appendix.html>. Accessed 2026-10-11.