> **SAMPLE - fictional client.** BrightLedger is not a real company. This memo shows the structure, depth, and tone of a Triage Pro deliverable. Findings, evidence patterns, configurations, and recommendations are invented. No real client data is referenced.

Production diagnostic - BrightLedger

Tier: Triage Pro - **Date:** 2026-05-11 - **Reviewer:** Olha Shevchenko

---

Summary

This review is based on the fictional intake packet provided for the sample: production symptoms, CloudWatch screenshots, ECS service settings, GitHub Actions deploy workflow, and selected Laravel dashboard code. I did not assume shell access or direct production access.

BrightLedger is stable at low load and degrades under sustained traffic. The highest-confidence causes are:

1. Dashboard N+1 query on high-employee accounts
2. Burstable RDS instance running out of headroom
3. Deploy pipeline with weak health checks and no automated rollback

Severity: Moderate overall. No active outage, no data integrity issue, and manual recovery exists. But the failures are load-correlated and will worsen as traffic grows. Finding #3 is urgent because it already turned one bad deploy into ~3 hours of degraded production.

Recommended next step: fix deploy safety this sprint, then address the dashboard query and RDS sizing in the next remediation pass.

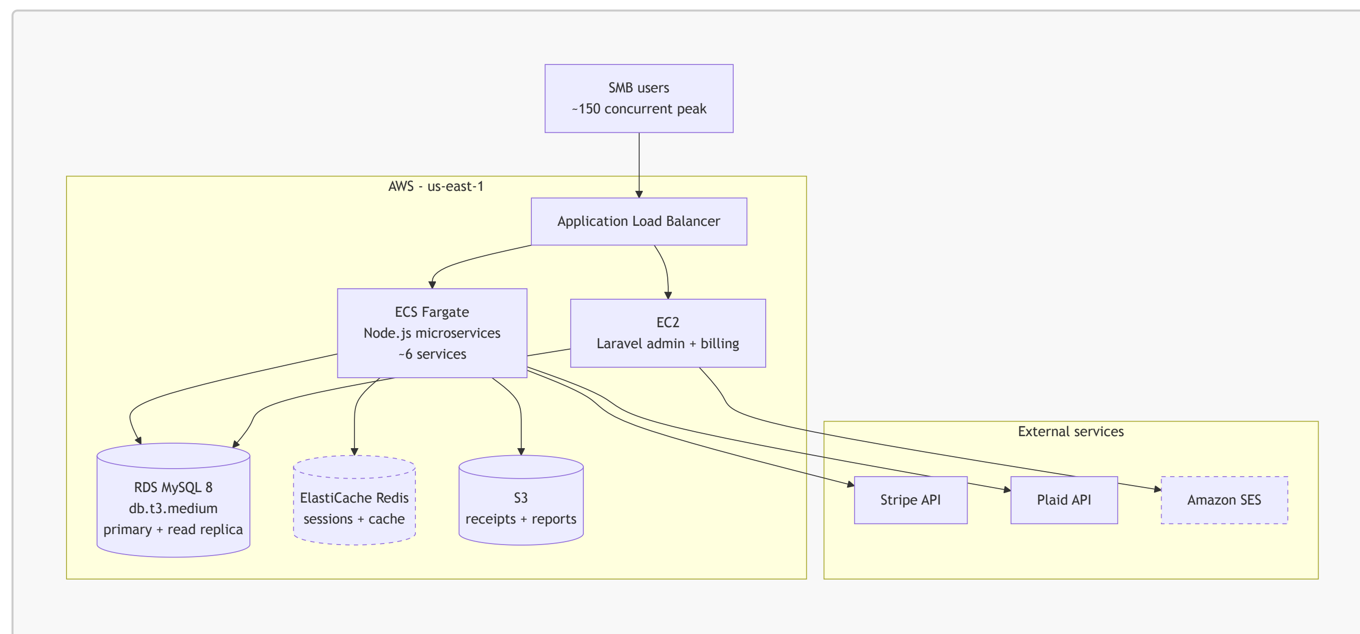
Assessment basis

Input reviewed	Signal
Intake symptoms	Slow dashboards for large accounts; peak-hour 502s; 3-hour degraded deploy
CloudWatch screenshots	RDS pressure during peak windows; no obvious app-wide outage
ECS service settings	Rolling deploy without a health check that proves dependency readiness
GitHub Actions workflow	Deploy success is not gated by post-deploy smoke tests
Laravel dashboard excerpt	Expense rows loaded per employee instead of as a bounded relation

I would not rate this urgent overall because there is no active incident, data loss, or security exposure. I would not rate it low because all three findings worsen with traffic and at least two would surface during investor technical diligence.

Architecture snapshot

Reconstructed from intake answers and shared artifacts. Solid boxes are supported by the sample packet; dotted boxes are assumed.



The main pressure point is **RDS**. Five Node services plus the Laravel admin read/write through the same MySQL primary. The read replica appears underused because application code is not routing reads to it. When primary IOPS or CPU credits run out, dashboards slow together.

Secondary risk: the **Laravel admin** runs on one EC2 instance with no autoscaling and no meaningful ALB health feedback. If it fails during deploy, admin access stays down until manual recovery.

Findings

1. Dashboard endpoint has an N+1 query pattern - Moderate

Observed symptom: dashboard loads in 8-15 seconds for larger customers, but ~200 ms for small accounts.

Evidence: the dashboard code iterates employees and accesses expense rows inside the loop. That shape can turn an 80-employee account into 81 queries instead of 2.

Verify:

1. Enable Laravel query logging for one representative dashboard request.
2. Look for repeated `SELECT * FROM expenses WHERE employee_id = ?`.
3. Confirm query count grows with employee count.

Fix: eager-load the relation with `Employee::with('recentExpenses')`, constrain it with `latest()->limit(...)`, and add a regression test asserting bounded query count for a large fixture org.

Effort: 4-8 hours.

---

2. RDS is undersized for sustained 800 RPM - Moderate

Observed symptom: sporadic 502s and dashboard slowness during peak hours.

Evidence: the RDS class is `db.t3.medium`, which is burstable. It performs well while CPU credits exist, then drops to baseline once credits are exhausted. With the N+1 query pattern above, sustained 800 RPM can burn credits quickly.

Storage may compound this. If the volume is small and still on `gp2`, baseline IOPS may be around 300 for a 100 GB disk, which is easy to saturate with expense lookups and joins.

Verify in CloudWatch:

1. `CPUCreditBalance`
2. `BurstBalance`
3. `DatabaseConnections`
4. Read/write latency around peak traffic

Credit depletion during peak windows confirms the sizing issue. High connections without credit depletion would point instead to pool exhaustion.

Fix: migrate to a non-burstable class such as `db.m5.large` and move storage to `gp3` so IOPS is explicit. Test in staging and apply during a maintenance window.

Effort: 1-2 days.

---

3. Deploy pipeline lacks reliable health checks and rollback - Urgent

Observed symptom: Friday deploy left production degraded for ~3 hours before manual revert.

Evidence: ECS is marking deploys healthy once tasks reach `RUNNING`, but the current health check does not prove the app can serve real traffic. Friday's incident is the proof: a deploy completed cleanly while the new tasks could not actually serve. A static `/` check will pass even if DB, Redis, or critical config is broken.

Verify:

1. Review ECS service health check and `HealthCheckGracePeriodSeconds`.
2. Review ALB target group health path.
3. Check GitHub Actions for post-deploy smoke tests.
4. Reconstruct Friday's detection and rollback timeline.

Fix:

- Add `/health` that validates DB, Redis, and one critical downstream dependency.
- Configure ECS health-check grace period and realistic timeouts.
- Add CI smoke tests that must pass before deploy success is declared.
- Consider CodeDeploy linear/canary promotion for safer rollout.

Effort: 2-3 days. This is the highest-impact fix.

Recommended remediation

Track A - internal sprint fix

Assign Finding #3 to the engineer who owns CI/CD. Use the verification and fix list above. Expected time: 2-3 days. No external cost.

Track B - focused remediation engagement

I can remediate all three findings and add the minimum observability needed to prevent recurrence.

Item	Scope
Effort	5 working days
Deliverables	All three fixes, RDS dashboard, deploy alerts, dashboard query-count assertion in CI
Price	~$3,500 fixed-price
Delivery model	1-2 days read-only investigation, then PR-by-PR remediation with your approval

A narrower Finding #3 engagement would be 2-3 days, roughly $1,500.

Decision point: if your team has capacity, do Track A immediately. If investor-DD prep is consuming engineering time, Track B buys a week of senior execution without interrupting the roadmap.

Not worth prioritizing now

- **Node-to-Go rewrite:** the bottleneck is database behavior under load (Findings #1 and #2), not runtime. A rewrite carries the same pattern into a different language.
- **MySQL-to-PostgreSQL migration:** Findings #1 and #2 are sizing and query shape, not engine choice. Both fail identically on PostgreSQL.
- **Stripe webhook reliability:** the handler excerpt in the intake packet is idempotent and verifies signatures. Stripe's at-least-once delivery is well-tested at scale.
- **Multi-region:** premature until the current single-region setup is healthy and the read replica is actually used.

None of these would materially improve the symptoms described.

Follow-up questions

Triage Pro includes **5 rounds** of follow-up questions. Send clarifying questions in Upwork chat, for example: "expand on cause #2" or "what CloudWatch query should I use for BurstBalance?"

Remediation review window

You have **30 days** from today, until **2026-06-10**, to share your remediation plan or fix attempts. I will return one round of written feedback within 3 business days.

To use it, post your plan, PR links, or deploy post-mortem in Upwork chat with the label "remediation review."

Olha Shevchenko - Senior Backend Engineer - Node.js + AWS - 2026-05-11

Delivered as a single PDF. Follow-up Q&A is for clarifying these findings, not rewriting the memo.