AWS Lambda Silent Crash – A Platform Failure, Not an Application Bug

What happens when a production-ready startup proves a runtime failure beyond doubt – and is still told it's their fault?

Over a four-week investigation, I uncovered and proved a silent, platform-level crash in AWS Lambda (Node.js, VPC-attached) that terminates functions mid-execution during outbound HTTPS calls — with no logs, no exceptions, and no catchable errors. I provided AWS with stripped-down repros, deep forensic traces, and a fully working EC2 comparison. In return, AWS denied the issue, blamed my code, and withheld internal telemetry — until their own reproduction accidentally proved I was right.

They claimed I missed a reject() — but their test was the one missing it.
They said the crash was app-level — but it happened **after** my function returned.
They suggested EC2 — I was already there.
And when I asked for engineering — they gave me sales.

This isn't just a Lambda bug. It's a platform failure, misdiagnosed through a broken support process.
If you're running production on Lambda: **it may fail silently — and AWS might blame you for it.**

Contents

Executive Summary

Over the course of four weeks, I — as the principal engineer and CTO for a healthcare-based AWS Activate startup on AWS Business Support — diagnosed a fatal AWS Lambda runtime failure that:

- Occurred exclusively in VPC-attached Lambda functions (Node.js)
- Caused silent termination during outbound HTTPS calls — with **no logs, no exceptions, no metrics**
- Was fully reproducible across minimal test cases and stripped-down runtime environments

To isolate the issue, I:

- Provided AWS with **minimal reproducible examples**
- **Rebuilt our entire CI/CD pipeline** using Amazon Linux 2023 containers
- **Removed all native binaries**
- Verified the same code ran flawlessly on EC2 — under **identical IAM, VPC, and network conditions**

Despite this, AWS Support repeatedly insisted — without evidence — that our code was at fault.

Across four internal escalations, three formal complaints, and an attempted executive escalation (which was never delivered), AWS provided:

- No runtime telemetry
- No crash diagnostics
- No access to Lambda engineering
- No proof that the platform was functioning as expected

I requested an executive escalation in writing. The AWS account executive — responsible for routing that request — failed to deliver it, and later deflected responsibility by claiming no executive was copied. But ensuring it reached one was *his* job.

Instead, they arranged a non-technical meeting with account management and support — without involving anyone capable of addressing the runtime itself.

At no point did AWS proactively investigate the platform.
I proved it wasn't us.
They blamed us anyway.

Then — when I challenged their internal reproduction —
they quietly escalated it to the Lambda team.

The very team they insisted never needed to see it.

It Started with a Simple Question

I just wanted to know why our transactional email wasn't being sent.

A single HTTPS call inside our AWS Lambda function — meant to trigger a welcome email — appeared to succeed.

- It returned a 201 Created.
- But no email was sent.
- No logs were produced.
- Our email service was never contacted.

So, I started debugging.

What I found wasn't an app-layer bug — it was something far worse:

A fatal, undocumented platform-level failure in AWS Lambda.

The First Symptom: A Success That Wasn't

The endpoint I used to trigger email sending was simple:

```
@Post('/debug-test-email')
async sendTestEmail() {
  this.eventEmitter.emit(events.USER_REGISTERED, {
    name: Joe Bloggs,
    email: 'email@foo.com, // legitimate email was used for testing
    token: 'dummy-token-123',
  });
  return { message: 'Manual test triggered' };
}
```

It emits an event, then immediately returns a response — meaning it always reports success (201), regardless of whether the downstream email handler succeeds or fails.

But here's what happened:

- I received the HTTP response
- No email arrived
- No logs appeared in CloudWatch
- No errors fired
- And the USER_REGISTERED event handler was never called

The Lambda simply **stopped executing** — silently, mid-flight.

The 201 response was intentional — and critical. It allowed the controller to return before downstream failures occurred, revealing that Lambda wasn't completing execution even after responding successfully.

A response was returned, but the function **never completed its actual work**.

That’s what led me deeper — to runtime tracing, stripped-down builds, and eventually to the discovery of a **silent HTTPS-triggered crash inside the Lambda microVM itself.**

The Failure

Our production Lambda workload sends transactional emails over HTTPS. In VPC-attached Lambdas running Node.js 20.x, **any HTTPS call caused the function to terminate instantly** – mid-stream, without error, without logging.

Examples of affected code paths:

- https.request → .end() → crash
- axios.post(...) → crash
- new SESClient().send(...) → crash

These failures happened:

- With **no exception thrown**
- With **no logs** after the call
- Even with **global error handlers and process.on('uncaughtException') hooks**

Our Diagnostic Effort

To isolate the issue, I conducted a full-stack diagnostic teardown:

- Rebuilt the CI/CD pipeline from scratch using Amazon Linux 2023 Docker containers to ensure runtime parity with Lambda
- Verified build artifacts using unzip -t, ls -lR, and du -sh to ensure file integrity, module presence, and size compliance
- Used npm ci --omit=dev to ensure production-only installs, eliminating extraneous dependencies
- Stripped all native .node binaries and rebuilt packages to avoid Linux/Windows incompatibilities
- Revalidated the Node.js runtime behaviour across 18.x, 20.x, and EC2-based equivalents
- Instrumented builds with NODE_DEBUG=net,tls,stream,https to trace stream lifecycles and pinpoint termination
- Implemented layered logging at every service level: controller, event emitter, service method, and internal HTTPS request
- Verified full outbound connectivity and DNS resolution from EC2 — using the same VPC, IAM role, and runtime configuration — confirming the issue was isolated to Lambda's execution environment
- Deployed minimal plain Node.js functions — outside NestJS — to remove framework overhead and ensure reproducibility
- Isolated Lambda invocation patterns to rule out concurrency, memory exhaustion, or ephemeral state errors

- Swapped email destinations between SES, mocked endpoints, and API Gateway to test DNS vs. TLS stack behaviour
- Confirmed full application success on EC2 — using the *exact* IAM role, VPC configuration, environment variables, and deployment bundle — proving the code itself was sound and the issue was isolated to AWS Lambda's runtime environment
- Created full crash signature traces showing termination *after* req.end() but *before* any error or timeout event, with zero log continuation
- Maintained forensic parity by preserving request IDs, event payloads, and service response comparisons between environments
- Validated Lambda handler structure with try/catch, .on('error'), .on('timeout'), and process.on('uncaughtException') to confirm platform-level bypass

Every diagnostic step confirmed: **this was not an app-layer crash.**

Here's where our logs stop:

```
>>> [Native HTTPS Test] Finalizing request (req.end())...
HTTP: outgoing message end.
TLS: client initRead
STREAM: reading, ended or constructing: false
--- FUNCTION TERMINATES HERE ---
```

AWS Support's Response

Despite our evidence, AWS Support:

- Provided no Lambda-side logs, termination reasons, or microVM data
- Claimed (without proof) the issue was "code-related"
- Suggested switching to EC2 or Fargate (which I had already implemented)
- Offered meetings with sales staff and TAMs, not runtime engineers

On April 30, AWS Account Management wrote:

"I exclude that issue is based on our Lambda Service and would like to share our ideas about how to adjust your code to make it run with Lambda."

No ideas were shared. No logs were offered. No engagement occurred with the Lambda team.

This deflection directly contradicted:

- AWS's own prior admission that Lambda microVM diagnostics were required
- Our logs, NODE_DEBUG output, and validated reproductions

What This Means

This was a **platform-level crash**, with all evidence pointing to the intersection of:

- Node.js runtime
- HTTPS stream termination
- Lambda’s VPC microVM networking stack

AWS failed to acknowledge or investigate it. Instead, they:

- **Assumed without proving** it was our code
- **Ignored multiple formal complaints and escalation deadlines**
- **Offered sales calls instead of diagnostics**

When the Escalation Path Fails — and the People Responsible Let It Stall

I didn’t skip steps.
I didn’t go public immediately.
I followed every channel AWS prescribes.

I filed four internal escalations.
I submitted three formal complaints.
I initiated an executive escalation in writing — with clear context, detailed reproduction steps, a final deadline, and direct requests for Lambda runtime engineering engagement.

The escalation email was sent to multiple AWS leadership addresses, including Activate program managers and senior AWS stakeholders.

It outlined the failure in detail.

It was calm, structured, and technically specific.

A representative from AWS Account Management acknowledged the message and proposed a meeting — but without Lambda engineering, without platform diagnostics, and without addressing the core conditions I explicitly outlined.
When I reiterated our need for technical ownership and escalation, the request stalled.

Later, I was told the escalation “wasn’t delivered to executives.”
But the recipients were clear.
The intent was explicit.
And facilitating internal routing was the responsibility of the AWS team copied on the thread.

I followed the process.
They let it stall.
And then suggested I hadn’t escalated at all.

This wasn’t a communication gap.
It was a breakdown in **ownership, escalation integrity, and customer advocacy** — at the exact point AWS claims to take these issues most seriously.

Missing Logs, Missing Accountability

Throughout our investigation, we repeatedly asked AWS Support for any internal logs that could explain the sudden termination — including:

- Lambda microVM lifecycle logs
- Internal runtime errors
- Process exit signals
- Crash diagnostics tied to the RequestId

We received none.

When directly asked why no such logs had been shared, the AWS Support Engineer responded:

"We don't provide those logs to the customer."

This admission matters. AWS effectively claimed the crash was our fault, while simultaneously:

- **Denying access to the only telemetry that could prove it**
- **Avoiding involvement from the Lambda runtime team who could validate it**

In other words:

AWS insisted we were wrong — but refused to show the evidence they claimed to have.

They had internal visibility.
We had runtime blindness.
And despite our willingness to collaborate, investigate, and rebuild — they stonewalled.

The Internal Reproduction That Proved Us Right – "Smoking Gun"

After four weeks of denials, AWS internal team reviewed our code to identify the root cause of the silent crashes we had documented extensively.

Here's what AWS Support reported back:

"The internal team reviewed the code and came back with the following findings. They noted that the provided code is not production-ready and should be considered only a best-effort guide toward a solution."

They then cited our existing implementation from the sendEmailViaApi() method — specifically the log line just before req.end() — as the last observable log before the function terminated. In their words:

```
"Final log before crash: >>> [Native HTTPS Test] Finalizing request (req.end())...
Next expected line (never reached): >>> [Native HTTPS Test] Promise resolved with
data..."
```

They acknowledged they could **reproduce the crash** — and even supplied their own test code which mirrored ours, minus one crucial detail: **they forgot to call reject() in their Promise error handlers.** Their own support update explicitly confirms this omission:

“We noticed that in our initial reproduction, the reject() calls were missing in the error and timeout handlers.”

Here is the code AWS used to reproduce the crash:

```
const https = require('https');

exports.handler = async (event) => {
  try {
    const endpoint = 'https://nonexistent12345.fakedomain.test';

    await new Promise((resolve, reject) => {
      const req = https.request(endpoint, (res) => {
        let data = '';
        res.on('data', (chunk) => data += chunk);
        res.on('end', () => resolve(data));
      });

      req.on('error', (e) => {
        console.log('if this is the last line, then "crash" occurred.');
        // MISSING: reject(e);
      });

      req.on('timeout', () => {
        console.log('Request timed out.');
        // MISSING: reject(new Error('timeout'));
      });

      console.log('Sending request...');
      req.end();
    });

    console.log('This log is never reached if request crashes');
    return { statusCode: 200, body: 'Success' };
  } catch (err) {
    console.error('Caught error:', err);
    return { statusCode: 500, body: 'Error occurred' };
  }
};
```

What Did the Logs Show?

Exactly what we’d been reporting all along. The function terminated silently after the HTTPS request — with no errors, no exception, and no return path triggered. This was the full log output from AWS’s test:

```
2025-04-30T13:27:00.173Z  INFO Sending request...
2025-04-30T13:27:00.254Z  INFO if this is the last line, then "crash" occurred.
END RequestId
```

No catch. No error. Just silence. Identical to what we observed in production.

The Follow-Up "Fix" That Didn't Fix It

AWS then submitted a revised version of their code that added the missing reject() statements. This time, instead of a silent crash, the function logged an error and returned as expected — but only because the domain (bogusdomain.test) caused a DNS resolution failure.

They concluded:

“Based on these findings, the internal team concluded that the issue is not related to the Lambda service itself.”

But this conclusion simply doesn’t hold.

Here’s Why That “Fix” Doesn’t Apply

1. **Our code already had those reject() handlers.**
   Their critique didn’t match our implementation — it matched theirs.
2. **Their test used a DNS failure.**
   In our production scenario (using valid HTTPS endpoints like AWS SES), the crash occurred *before* any handler was triggered. Adding reject() did nothing — because the process exited *before* it mattered.
3. **They never tested our real use case.**
   Their logs confirmed the exact crash signature — but their conclusion ignored it.

What This Really Shows

- AWS did reproduce the crash.
- AWS did confirm the silent termination.
- AWS did forget reject() in their test.
- And AWS did falsely attribute that mistake to us.

Then — after correcting their own bug — they declared the issue solved. But what they “solved” was a DNS hang in broken test code, not the actual platform-level failure affecting real outbound HTTPS connections in VPC Lambda functions.

So, let’s call this what it is:

The only evidence AWS has ever provided to argue we were wrong… is a test that actually proved we were right.

The Meeting That Proved the Point

Following four weeks of unanswered diagnostics, missed escalation deadlines, and unresolved support tickets, AWS invited us to a meeting to “move forward.”

I accepted — with clearly written conditions:

- Presence of Lambda engineering or runtime/platform ownership
- Access to internal telemetry or microVM diagnostics
- A commitment to technical accountability, not sales guidance

- A record of the meeting, and a clear technical summary afterward

Only one of our stated conditions was met: the session was recorded. As of this writing, no copy of that recording has been provided.

Who Attended

The call consisted of:

- Support personnel
- Account management
- Solutions Architects
- No senior support manager
- No Lambda runtime engineer
- No diagnostic authority

The AWS attendees could not provide technical logs, identify crash signals, explain Lambda's behaviour, or speak for the platform team.

What Was Said

Much of the meeting consisted of evasive language and narrative management. At one point, AWS support staff stated:

"It's difficult for Business Support customers to get to this level."

That single line exposed what I had already experienced:
The **support model isn't structured to handle platform-level failures** — especially not for startups.

I raised this directly during the call:

- Why, after a month of reproducible evidence, was no engineering presence on the call?
- Why had no Lambda team member reviewed the issue, despite a formal executive escalation?
- Why was I expected to accept sales representatives and SAs in place of root cause analysis?

When pressed, support staff attempted to answer for AWS Account Management — until I explicitly stated the question was directed to the commercial team, not support. The resulting tension was visible.

When I requested that a **Support Manager** be present on the call — as is standard escalation protocol — I was asked by support:

"Why do you need a Support Manager?"

I responded:

"Because this is your escalation process, and I expect technical accountability and a clear next step — not another deferral."

The request was dismissed. No support management attended.

I also highlighted the most damning technical fact:

“Our API is never hit. There are no logs. That’s not a code bug — that’s Lambda terminating without cause.”

There was no rebuttal.

AWS claimed their simplified function — using https.request() — reproduced the crash *only* because it lacked reject() calls in the error and timeout handlers.

I was told, confidently:

“Your code has the same issue — it’s missing reject()s.”

But this claim was **false**.

The version of the code I submitted to AWS **included proper reject() handlers** — as confirmed in our logs and source artifacts.
The statement that our code “had the same issue” was inaccurate — and reflected a lack of attention to detail at a critical stage of escalation.

They claimed to have read our code. But they missed the very thing they were using to blame us.

I asked what runtimes were used. I was told Node.js 20 and 18 — but the screenshots AWS referenced showed Node.js 22.x, not the same version I was using.

I asked for the logs from the other runs. They were not shared.

Lack of Preparation

During at least two separate calls leading up to this meeting, I was asked by our AWS Account contact to provide an update on the support case — despite the full case history, logs, and technical documentation already being available in the ticket.

This individual had not read the support thread.

During the final call, I asked a direct question to the group:

“How many people on this call have read the full ticket history and reviewed the technical evidence?”

Only one person responded.

In a meeting positioned as a resolution, with four weeks of context, logs, reproductions, and escalation history — **only one AWS participant came prepared.**

AWS asked us to explain the issue again — because they hadn’t reviewed their own support record.

This wasn’t just a lapse in communication.
It was a clear sign that the meeting had not been taken seriously as a technical escalation — even after executive-level requests.

What It Revealed

- **The support structure failed.**
  Business Support has no path to runtime diagnostics, regardless of evidence.
- **The internal coordination failed.**
  AWS teams were unaware of escalation history, timelines, and conditions previously agreed to in writing.
- **The technical response failed.**
  The call ended with a vague insistence that their code proved the issue was ours — even though it reproduced the crash exactly.

And yet… AWS Quietly Escalated It Anyway

Within hours of the meeting, I received a ticket update confirming the case had been **escalated to the Lambda team.**
This was the very team I had asked for since Day One — the same team AWS previously insisted didn't need to be involved.

The meeting was designed to manage us.
Instead, it proved us right — and triggered the internal escalation AWS had resisted for four weeks.

Why I'm Publishing This

I believe other developers, CTOs, and cloud architects should know:

- AWS Lambda can fail silently at the system level
- Standard debugging tools (logs, traces, X-Ray) offer no visibility
- Support may deflect or delay — even when all app-layer issues are disproven

I have now:

- Initiated migration planning to alternative providers
- Briefed our stakeholders on the risks of serverless in production
- Published this report to raise visibility and accountability

I don't publish this lightly. I built everything on AWS.
I did the work. I escalated through every internal channel.
I proved our case — and I was dismissed without cause.

Despite four internal escalations, three formal complaints, a written executive-level escalation, and a meeting AWS themselves proposed (then failed to staff), the only technical insight AWS ultimately provided was a **"smoking gun"** — code that inadvertently validated our claim of a fatal, uncatchable failure inside Lambda.

No root cause was acknowledged.
No telemetry was shared.
No platform ownership was taken.

This wasn't just an investigation — it became a masterclass in **applied debugging under fire.**

As a solo engineer, I outpaced AWS's own diagnostics, rebuilt our deployment stack from scratch, eliminated every confounding factor, and **forced AWS to inadvertently reproduce a failure they still refused to acknowledge.**

This was not a guess or a hunch.
It was disciplined. Technical. Forensic.

The evidence didn't just point to Lambda.
It cornered it.

And when AWS told me to move to EC2?

I had already done it — and I still kept digging.

Because production didn't need a workaround.
It needed an answer.

I did not go public prematurely.

I published only after AWS had internally escalated the issue to the Lambda service team — the very team I had asked for since day one.

At that point, every responsible channel had been exhausted.
And AWS had acknowledged — through action, if not admission — that the issue warranted platform-level review.

This report is not a reaction.
It is a record.

We are happy to update this post if AWS provides:

- A full technical root cause
- A public acknowledgment of the issue
- Or credible evidence that disproves our findings

Until then:

Lambda is silently failing — and AWS has now proven it themselves.
They just won't admit it.
And worse — the support process designed to detect and resolve these failures is broken.
If you're running production workloads on Lambda, the platform may fail silently — and the support chain meant to protect you may delay, deflect, or outright misdiagnose the issue.
That's not just a runtime bug. That's a reliability risk baked into the ecosystem.

Postscript: What Happened When I Tried to Share This

Following the completion of this report, I attempted to share a link to the findings on [r/AWS] — a forum dedicated to technical discussion within the Amazon Web Services ecosystem.

The post was submitted via a newly created account. Within seconds, before the submission had even cleared moderation or become visible to other users, the account was **permanently suspended**.

This was not a content removal.
It was not a soft moderation action.
It was a full account-level takedown — executed without warning, message, or email notification.

The post never appeared publicly.
The account was rendered inactive almost immediately.
No correspondence followed. The registered email received no alerts or justification.

This response was disproportionate to the activity involved: a first-time technical post, written with precision, submitted to the relevant forum, and not yet visible to any other user. No community feedback had occurred. No moderator message was received. There was simply a silent and complete revocation of access.

Whether this was an automated trigger or a more selective intervention remains unclear. But the outcome is important:

A detailed, fact-based post — never seen by the community — was removed pre-emptively, and the author account terminated without explanation.

In a platform of this scale, rare failures are expected.
But the pattern that emerged here suggests more than chance.

It reflects the same systemic pattern that defined this entire investigation:
No logs. No response. No ownership.

Not a single mechanism — support, telemetry, or public discourse — allowed the issue to surface.
Given the consistency of this suppression, it is difficult to believe this was coincidental.

Disclaimer

This document reflects the outcome of a four-week independent technical investigation, conducted in good faith and supported by reproducible test cases, platform comparisons, and recorded AWS support interactions. All findings are based on direct observation, verifiable diagnostics, and contextual interpretation of AWS behaviour.
While every effort has been made to ensure factual accuracy, conclusions regarding AWS's support process and moderation handling are informed by firsthand experience.
This document is not legal, commercial, or professional advice. It is a technical record compiled by an engineer responsible for mission-critical production systems.