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 seven-week investigation, I uncovered and proved a silent, platform-level crash in AWS Lambda — affecting Node.js functions in a VPC making outbound HTTPS calls. The failure occurred mid-execution, *after* the function had returned a success response. No logs. No errors. No telemetry. No way to catch it.

From day one, I did what AWS claims to value in a partner.

I stripped the function down to minimal reproducible code.
I tested across runtimes, regions, and infrastructure baselines.
I rebuilt on EC2 and proved that the issue vanished entirely.
I shared logs, traces, metrics, and internal observations.

I escalated through every official channel:

- Support dismissed it.
- My Account Executive ignored it.
- Formal complaints were met with silence.
- Internal re-escalations led nowhere.
- AWS Activate — the startup programme — refused to engage.
- And executive outreach yielded nothing but a two-line response weeks later.

At every stage, I remained professional. I kept the tone restrained. I offered AWS every opportunity to engage constructively.

Instead, they claimed the bug was in my code — despite the function crashing *after* returning a 201.
They claimed I had forgotten a reject() — despite the error occurring deep inside https.request(), and *their own reproduction* missing the handler.
They suggested I move to EC2 — by then, I already had.
I asked for Lambda engineering — they gave me sales. Then silence.

Even AWS Activate, whose sole purpose is to support startups like ours, refused to take part. Their response wasn't technical — it was procedural. A polite copy-paste directing us back to the same failing support system we were already trapped in.

This wasn't just a Lambda bug. It was a **platform-level failure, misdiagnosed through a broken support process**, and left to rot in plain sight.

AWS's internal systems failed. Their support model failed. Their startup engagement model failed. And above all, their cultural commitment to ownership — the thing they claim defines them — was nowhere to be found.

So, we left.

We migrated everything.
Lambda was decommissioned.
Critical services were refactored for Azure.
And our engineering culture now lives on a platform that still understands trust has to be earned — not assumed.

If you're building on Lambda: know this.

It may fail silently.
And if it does — AWS may blame you, even after they reproduce the failure themselves.

Contents

Executive Summary

Over a seven-week investigation, I — as CTO and principal engineer for a healthcare-focused AWS Activate startup — diagnosed and proved a fatal runtime flaw in AWS Lambda that:

- Affected **Node.js functions in a VPC**
- Caused **silent crashes during outbound HTTPS calls**
- Produced **no logs, no exceptions, and no catchable errors**
- Was **fully reproducible** using minimal test harnesses

To isolate the issue, I:

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

Despite this, AWS Support repeatedly insisted — with **no evidence** — that our code was to blame.

Across **four internal escalations**, **three formal complaints**, and an **attempted executive escalation**, AWS provided:

- No telemetry
- No runtime diagnostics
- No access to Lambda engineering
- No proof that the platform was behaving as designed

Instead of escalating to technical owners, AWS routed the issue through sales and support. A non-technical meeting was arranged with no one present from Lambda engineering. No investigation into the platform itself was ever initiated — until I directly challenged AWS's flawed internal reproduction.

When their own test code reproduced the same silent crash — the one they had blamed us for — the case was quietly escalated to Lambda.

The same team AWS insisted never needed to see it.

No accountability followed.

Not from Support.
Not from the Account Executive.
Not from AWS Activate.
Not from anyone.

We rebuilt our infrastructure.
We stabilised production without them.
And we left the platform.

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 failure in AWS Lambda — and a support escalation process so fundamentally unfit for purpose that it compounded the impact at every layer.

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.

What Happened After AWS Escalated to the Lambda Team

When AWS Support finally confirmed internal escalation to the Lambda service team — the same team I had been asking for since day one — I paused escalation. I gave them room. I expected a shift in posture.

Instead, three days later — **35 days after the original ticket was raised** — I received a message asking for:

- A deployment zip file with proper reject(err) handling
- Logs showing the crash with the new implementation
- Confirmation of whether the function still fails inside a VPC
- A rerun of the exact test they had already internally reproduced

In short: they asked me to start again.

This wasn't a request for clarification. It was a reset — a request for artefacts that have already been delivered, multiple times, across prior case updates, call notes, and debug summaries. The very same package AWS previously reviewed. The same logs they had already quoted. The same failure they had already confirmed.

It is now unclear — and deeply concerning — whether the Lambda team ever received the context they were allegedly escalated with.
If they did, they should not be asking for this.
If they didn't, it means the escalation was nominal — not technical.

And if the behaviour observed — silent function termination with no logs, no rejection, and no error propagation — is truly the intended outcome, then AWS needs to say so, clearly and publicly.

We are not asking for reassurance. We are asking for confirmation:

- Is this silent termination a documented and supported behaviour of VPC-attached Lambda functions?
- Does the Lambda runtime team consider this an acceptable and expected result?
- If not — why has it taken five weeks to engage meaningfully?

If this is, in fact, by design, then it must be documented — not discovered by startups debugging in the dark.

A function that:

- Returns a 201
- Emits no errors
- Logs nothing
- And silently terminates before completing its work

...should not be treated as healthy.

If this is the designed behaviour of VPC-attached Lambda under Node.js — then the documentation is missing. The architecture is incomplete. And the observability contract is broken by design.

Because in a platform that promises "no infrastructure to manage," this isn't just confusing. It's dangerous.

And 35 days after calling it out as a platform failure — after:

- Providing a minimal reproduction in a standalone Lambda
- Stripping all framework overhead and native modules
- Rebuilding our CI/CD pipeline on Amazon Linux 2023 for runtime parity
- Confirming the exact same bundle runs flawlessly on EC2
- Instrumenting every layer with logs, debug flags, and rejection handlers
- Watching AWS reproduce the crash — then blame their own omission on us
- And watching them quietly escalate it to the Lambda team anyway...

…AWS Support has returned.
Not with logs.
Not with telemetry.
Not with confirmation that Lambda engineering ever reviewed the failure.
But with a request to start again — as if none of it happened.

And this time, they added something new:

"The fact that the code works on EC2 does not mean that there is certainty that it will on Lambda."

That sentence says everything.
It divorces platform behaviour from platform accountability.
It rewrites the premise: Lambda isn't just a compute abstraction — it's a runtime lottery.

Your code can pass in EC2, pass in staging, pass in every controlled test — and still fail silently in production Lambda with no way to explain why.

And apparently, that's fine.
That's not a bug. That's not alarming. That's just how it is.

After five weeks, four escalations, internal reproduction, and a complete diagnostic teardown, AWS Support has returned — not to close the loop, but to reopen the doubt.
To pretend the history never happened.
To reset the burden of proof back on the customer — again.

This isn't support. It's erasure.
And the longer it continues, the clearer it becomes:

The real failure here isn't the runtime.
It's the refusal to acknowledge what it did.

Why We Left AWS

The decision to leave AWS wasn't technical — it was cultural.

By the time AWS support escalated our case to the Lambda team, we had already rebuilt our deployment pipeline, restored production stability on EC2, and proven — conclusively — that the crash was rooted in AWS's platform, not our code.

But AWS's behaviour after escalation told us everything we needed to know:

- They asked for artefacts they already had.
- They reset the conversation as if the past five weeks hadn't happened.
- They implied — with no evidence — that parity between EC2 and Lambda meant nothing.
- And they returned the burden of proof to us, despite reproducing the failure themselves.

This wasn't a misunderstanding. It was systemic minimisation.

We gave AWS:

- Logs,
- Reproductions,
- CI/CD rebuilds,
- Runtime-parity isolation,
- And instrumentation at every layer.

They gave us:

- Deflection,
- Delay,
- A misread reproduction,
- A denial built on their own mistake,
- And finally, silence — dressed as follow-up.

At some point, platform trust isn't just about features — it's about whether your provider will stand behind them when they fail.

AWS didn't.

They left a startup debugging in the dark, then blamed the flashlight.

And that's what cuts deeper than the crash.

- We are an Activate startup.
- We followed the Well-Architected Framework.
  We used their tooling, invested in their ecosystem, and built our stack around their recommended services.
  We did everything AWS says startup partners should do.
- And when the platform failed us — they vanished.
  - Our Account Executive stopped responding.
  - AWS Activate gave us a copy-paste reply and refused to engage.
  - No one from Lambda engineering ever appeared.
  - And the only admission of failure came by accident — through AWS's own flawed test code.

This wasn't just about technical support.
It was a loud, clear message to every startup:
If something goes wrong in the platform — even if it's provable, reproducible, and acknowledged internally — **you will be on your own**.
So, we made the call.

- Lambda was decommissioned.
- Core infrastructure was rebuilt on EC2, then refactored for portability.
- Kinesis, SES, and other tied services were deprecated.
- Azure is now our primary provider.
- And our engineering culture now rests on a platform that still believes customer trust is earned — not assumed.

This wasn't a cost decision.
It wasn't emotional.
It was the inevitable outcome of a support structure that could not — or would not — own its platform.
The crash may have started in the microVM.
But the failure was much larger — and entirely human.

And when we turned to the startup program?
AWS Activate — the very initiative designed to support early-stage companies — declined to assist, citing billing limitations and redirecting us to the same support paths that had already failed us. If the startup program can't intervene during a critical service fault affecting a production workload, one has to ask: *what is its purpose?* Certainly not to support startups when it matters most.

AWS Final Response – A Point-by-Point Rebuttal

On June 5, 2025, our AWS Account Executive issued a formal email response summarising AWS's position on Case #174369491300086. While the email was structured as a closure summary, its contents reflect a mix of misrepresentation, contradiction, and carefully worded legal positioning — with clear evidence of internal inconsistency and technical avoidance.

This section breaks down key claims made in that email, contrasted against the verifiable diagnostic record.

❝ "AWS investigated the reported issue thoroughly and concluded that the issue was most likely caused by your usage of anti-patterns..." ❞

This opening assertion sets the tone for the rest of the letter: blame shifted to the customer, based on a vague and unsupported claim.

Response: AWS provided **no definition** of what anti-pattern was allegedly in use, nor did they specify which line(s) of code or architectural decisions were responsible. The only link supplied was to a general AWS Lambda best practices page. This claim ignores:

- Our use of a Promise wrapper with explicit reject() calls.
- The fact that their own reproduction omitted these and **still crashed**.
- The success of the same codebase in EC2 under identical runtime conditions.

❝ "AWS team successfully reproduced the exact issue across Node.js versions 18, 20, and 22." ❞

This admission is significant: AWS replicated the crash **across three major production runtimes**. The impact of this line cannot be overstated — it is **internal confirmation** that the issue is not confined to our environment or codebase.

Response: This is a direct contradiction. You cannot simultaneously reproduce the issue and deny it is a Lambda failure. Either the crash is real, or it isn't. AWS chose to admit it occurred, then claimed it was expected — without documentation, telemetry, or any runtime-level justification.

❝ "The problem was confirmed to be related to how these Node.js runtimes interact with AWS Lambda." ❞

This is an implicit admission that **the failure is platform-level**. It is not a function of our application logic — it is a consequence of the runtime and microVM interaction.

Response: This sentence *should* have been the start of a platform-level root cause analysis. Instead, it was used to support the idea that such crashes are "normal," and that the onus was on us to work around them.

But this behaviour — silent termination mid-execution, with no errors, no catch blocks, and no logs — is **not documented**. It is not disclosed as a hazard in the Lambda docs. It is not presented in any lifecycle diagram. AWS is admitting to undefined platform behaviour — and then blaming the customer for encountering it.

❝ "AWS provided support above and beyond the Customer's subscribed business support tier…" ❞

This line is defensive. It attempts to reframe AWS's underperformance as a favour.

Response: In reality:

- No Lambda engineering contact was ever granted.
- No platform diagnostics were shared.
- No access to runtime traces, microVM state, or kernel logs was offered.
- Only after public escalation did AWS even admit their own test reproduced the crash.

Support "above scope" does not mean **correct** or **complete**. It means the problem was beyond Business Support's capability — and instead of escalating it to the right team, AWS stalled and misdirected.

❝ "To check that the solution recommended by AWS successfully resolved your issue as implemented, AWS asked you to provide updated code or technical logs… you have not provided." ❞

This is **demonstrably false**.

Response: We submitted:

- Multiple code bundles (.zip) with and without framework overhead.
- Logs with NODE_DEBUG=net,tls,https,stream.
- Consistent request IDs, payloads, and test events.
- Side-by-side EC2 vs. Lambda comparisons.
- Annotated test code with reject() handlers already included.

AWS themselves quoted our log output in the ticket — including the line after req.end() that was **never reached**. To claim we did not provide technical artefacts is either a lie or an indication that internal handoff between teams failed catastrophically.

❝ "AWS does not share internal diagnostics data for AWS Services, this is highly sensitive and confidential information." ❞

This refers to our repeated request for:

- MicroVM crash traces
- Runtime signal exit reasons
- Internal service logs tied to our RequestId

Response: This is the **core problem**. AWS **blamed the customer** while withholding the only telemetry that could confirm or refute that claim. They possess the crash data. We do not. Their refusal to share it breaks the trust contract implicit in any serverless model.

If the platform can terminate your function silently and AWS refuses to provide the reason — **then no amount of instrumentation will protect you.**

❝ "The meeting was conducted with… Senior Serverless Solutions Architects; Technical Account Manager; Lambda Subject Matter Experts..." ❞

This is an **inaccurate characterisation** of the call held on May 3.

Response: The call did **not** include any Lambda engineering presence. No participant was able to:

- Explain the platform's internal behaviour
- Discuss microVM lifecycle states
- Interpret the crash pattern technically
- Provide actionable diagnostic evidence

We were told it was "difficult for Business Support customers to get to this level," even after weeks of escalation, internal reproduction, and verified crash logs.

Furthermore, the **meeting was recorded** — a fact AWS later confirmed. But when we requested access to that recording, they refused to share it, citing "confidentiality."

So AWS:

- **Recorded the meeting**
- **Made incorrect claims about the attendees**
- **Refused to let us review what was said**

This is not transparency. It is narrative control.

❝ "Given that the investigation concluded that there was no issue with AWS Lambda operation… you are not eligible for SLA credits. We are evaluating internally whether we can offer you AWS credits as a measure of good will…" ❞

This line is the most revealing.

Response: If the platform functioned correctly, then **why offer credits?**
You do not offer goodwill compensation in cases where you are confident the platform performed as expected.

The offer itself is a **tacit admission** that AWS knows this was handled poorly — and that the platform misbehaved in ways they can't comfortably explain or defend in writing.

Final Analysis

AWS's final response:

- Confirms that the crash occurred
- Confirms that it spans multiple runtimes
- Suggests the behaviour is "expected," but provides no documentation or telemetry
- Denies access to internal logs or meeting recordings
- Accuses the customer of omission, despite quoting our own logs back to us
- Refuses platform-level accountability, even after internal escalation to Lambda

This was not a technical conclusion. It was a **legal close-out**.

The platform failed. AWS knows it. But admitting it would mean owning responsibility for runtime behaviour that contradicts their own documentation and breaks the contract of Lambda's execution model.

Instead, they:

- Blamed the customer
- Erased the timeline
- Controlled the language
- And offered compensation in place of truth

In the end, AWS didn't solve the issue — they buried it.
But the crash still exists.
The risk is still present.
And this time, we have the evidence.

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.

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.