© 2026 Yiguedan Philippe Andy Yaro



Secure Data Hub: Client Data, One Role Away

Yiguedan Philippe Andy Yaro

View GitHub Repository: github.com/andyyaro/Building-A-Secure-Data-Hub-in-the-cloud-AWS-

Let's connect! Email: yyaro@gmu.edu

LinkedIn: linkedin.com/in/andy-yaro

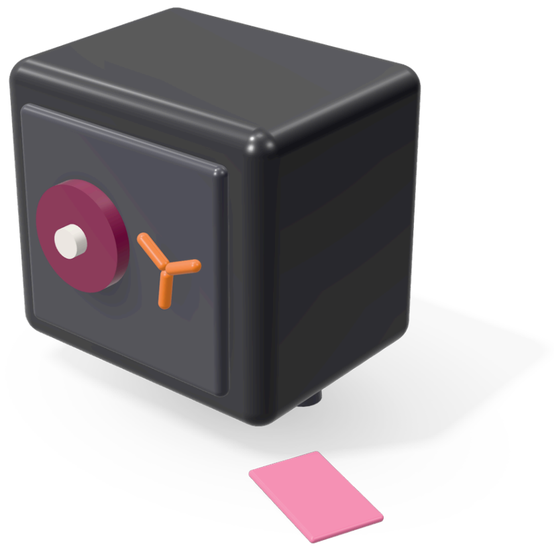


Secure Data Hub is my design for keeping clients' personal data on Amazon Web Services (AWS) so that the app serving it cannot simply read it: the data sits in a separate account, and each request must borrow a narrow role there.

I built the working core alone in November 2025, mostly in the AWS console: two Python functions, their permission policies, two web APIs and two tables, one encrypted. In August 2026 I added tests, Terraform (cloud set-up as code) and CI checks; in November 2026, a short film, committed with Claude as co-author.

On 3 November 2026 this deck checks the repository at its latest commit: it runs the tests and Terraform's checks, reads every policy, and says where my README, the project's front page, claims more than the code. I am Yiguedan Philippe Andy Yaro.

A. Why It Exists



Small teams that hold sensitive client data, in healthcare, law or finance, rarely have a security team. A break-in to their web app can mean a break-in to their data.

The idea: put the data in a separate AWS account (Amazon Web Services) that the app can reach only by borrowing one narrow role, a set of permissions, one request at a time.

And take the client's identity from a verified sign-in token, never from what the caller sends.

1. What Was Built, and What Was Only Designed

Built, in the Repo

Two Python functions on AWS Lambda: 196 and 90 lines.

Four exported permission policies.

Two exported API Gateway definitions and a 31-line template.

The encryption library, packed as a Lambda layer.

A small admin web page.

Evidence It Ran

A console test of the sign-in check, returning 200, on 15 Nov 2025.

A verified sign-in token from the same day.

A test event carrying a real API Gateway request time.

My written record of checking the data account's two roles on 19 Aug 2026 (no output committed).

Added in August 2026

52 tests that need no AWS account.

CI: the tests, Terraform's checks and a secrets scan.

Terraform for both accounts, never shown applied.

Written steps to fix one wrong policy file.

Designed Only

A nine-account AWS setup with account-wide limits and single sign-on.

Org-wide logging, alerts and threat detection.

Writing encrypted records; editing them as admin.

The client web page.

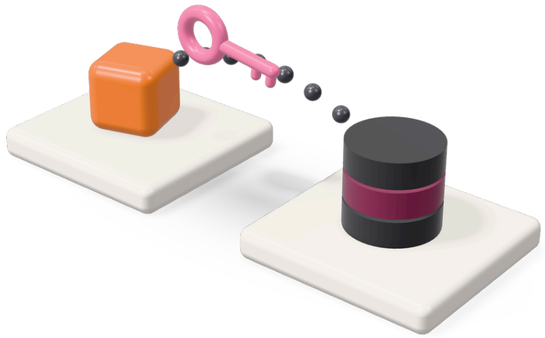
Commits, 142 in all, all under my name (the film commit credits Claude as co-author)



95 of the 142 change only slides, the README (the project's front page), docs or media. Only 2 of the 32 slide images in the repo are screenshots of the running system; the rest are designed slides.

Lambda = AWS's run-a-function service
API = an interface other programs call
API Gateway = AWS's front door for APIs
Terraform = cloud resources written as code
CI = checks that run on every push
layer = extra libraries for a function

B. Architecture

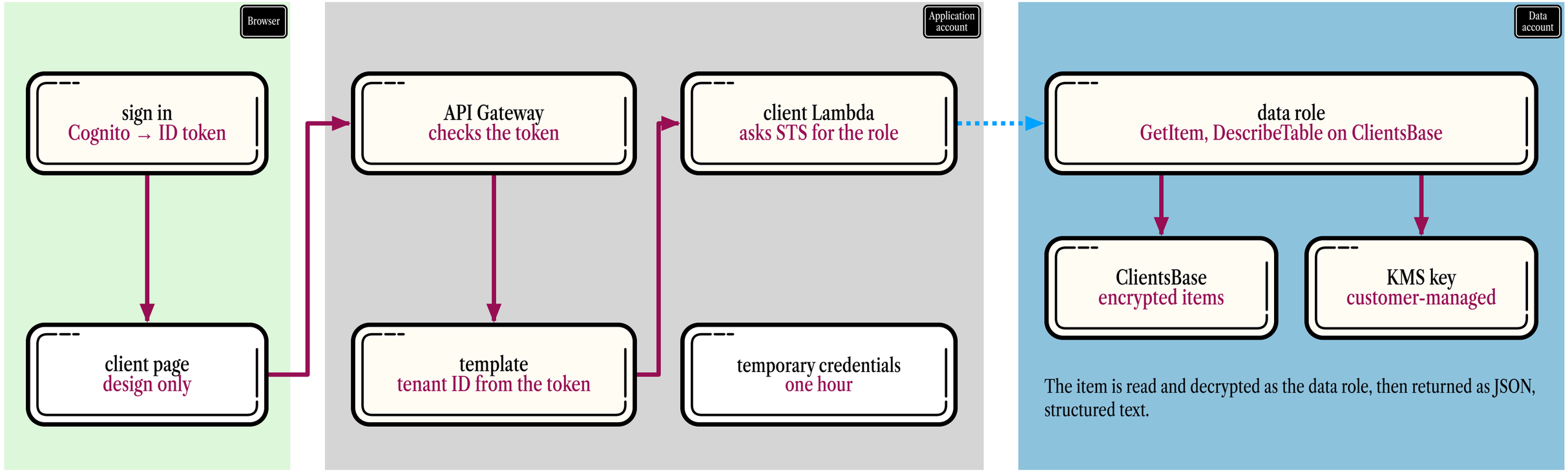


Two Amazon Web Services (AWS) accounts: an application account that runs the code, and a data account that holds the tables and the encryption key.

Every request crosses the line the same way: the function asks AWS's Security Token Service (STS) for a role, a set of permissions, in the data account, and works with credentials that expire within an hour.

No long-lived keys anywhere in the code. The client function's own role has no data permission; one committed admin file wrongly gives its role direct read access (next slides).

1. The Client Request, End to End



In the design deck, sign-in lives in a separate Security account and the pages sit behind CloudFront, AWS's content delivery network. The repo's Terraform puts sign-in in the application account and builds only the client API.

Cognito = AWS's sign-in service
ID token = a signed record of who signed in
STS = Security Token Service
KMS = Key Management Service
GetItem = read one item by its key
tenant = one client whose records stay separate
Lambda = AWS's run-a-function service

: Browser : Application account : Data account : Request : Assumes the role

1. The Four Policies, Read

```
$ python ../checks/policies.py lambda/IAM_policies
app account, client Lambda role
  logs:CreateLogGroup                          on every log group  <- wildcard   1
  logs:CreateLogStream, logs:PutLogEvents      on one log group
  sts:AssumeRole                               on ROLE_ARN_IN_DATA_ACCOUNT      2
data account, client read role
  dynamodb:GetItem, dynamodb:DescribeTable     on table/ClientsBase             3
  logs:CreateLogGroup                          on every log group  <- wildcard
  logs:CreateLogStream, logs:PutLogEvents      on one log group
app account, admin Lambda role
  dynamodb:Scan                                on table/<SUMMARY_TABLE_NAME>
  logs:CreateLogGroup                          on every log group  <- wildcard
  logs:CreateLogStream, logs:PutLogEvents      on one log group
  no sts:AssumeRole, yet the handler calls it: the export defect                4
data account, admin summary role
  dynamodb:Scan                                on table/<SUMMARY_TABLE_NAME>
  logs:CreateLogGroup                          on every log group  <- wildcard
  logs:CreateLogStream, logs:PutLogEvents      on one log group
```

Run on 3 November 2026 with a script written for this deck that reads the four JSON files in lambda/IAM_policies. The placeholders are in the files. I checked the data account's two roles live on 19 August 2026; the app account's live policy is still unverified.

1. Every policy lets its role create any log group in its account and region: a wildcard my README says is not there.

2. The client function's app-side role has one data-related right: to assume one role in the data account.

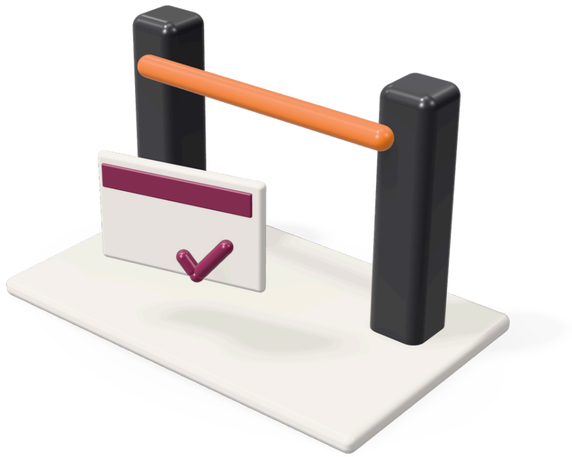
3. That data role may read items from, and describe, one table. No Scan or Query, so it cannot list the table.

4. The admin file grants Scan directly and has no AssumeRole: a wrong copy of the policy, kept visible by a test that is expected to fail.

i

Scan = read a whole table
DescribeTable = read a table's settings
wildcard = a * that matches everything
AssumeRole = the call that borrows a role
log group = a named set of logs

C. Tenant Identity



Several clients share one table, so every request must be tied to one client, the tenant, and return only that tenant's record.

The tenant ID comes from a claim in the verified sign-in token, copied by API Gateway before the function runs. No ID the caller puts in the query, path, headers or body is read.

That holds for requests the template handles. The edge cases, and the limits of the data role, follow.

1. From Token to Record

```
$ sed -n 15,31p apigw/ClientApp/mapping_template.vtl                    1
#set($client_id = $context.authorizer.claims['custom:client_id'])

#if(!$client_id)                                                        2
  #set($context.responseOverride.status = 400
  {
    "message": "Missing Client Id."
  }
#else
  {
    "client_id": "$client_id",
    "request_id": "$context.requestId",
    "source": "client-portal",
    "http_method": "$context.httpMethod",
    "path": "$context.resourcePath",
    "request_time": "$context.requestTimeEpoch"
  }
#end
```

1 The tenant ID is read from the token's claims, which the gateway has already checked.

2 No claim: the template sets a 400 status. AWS documents this override for responses, so whether the caller gets that 400, and whether the function is skipped as my README says, is unverified. Either way no record is read: the function fails on the missing field before it borrows the role.

Can a caller choose another tenant?

A caller tries to	Result
Edit the token	No: its signature is checked
Put an ID in the query, path or a header	No: the template reads none of them
Leave the claim out	No record is read
Send a body with a content type the template does not cover	Possibly: the exported console API is set to pass it through unchanged, and the function trusts the field. Untested; Terraform's setting would refuse it
Pick the claim at sign-up	Open: depends on sign-in settings that are not in the repo and not set in Terraform

claim = one field inside the token
template = rewrites the request first
400 = the web (HTTP) code for a bad request
passthrough = sending the body on unchanged

2. Where the Boundary Holds

What the Design Blocks

The client function's own role touching the data directly.

Listing the encrypted table: the client data role has no Scan or Query.

A caller choosing another tenant through the template.

KMS calls made as the wrong role (section D).

What It Does Not Block

Whoever controls the app listing every ID through the admin role, then reading each record, decrypted, through the client role.

Unmatched request bodies on the console-built API, if they reach it.

Tenant IDs in logs: the function prints the whole event and the tenant ID.

Errors that can come back with a success code: neither function handles one.

The honest version of my README's claim: a compromised app cannot read the tables directly or list the encrypted table, but it can list every ID through the admin role and read each record through the client role. Getting the data credentials from the user's verified token, with a DynamoDB key condition on the tenant, would close that: a compromised function could then read only the records of users whose tokens pass through it.



DynamoDB = AWS's key-value database
Scan, Query = reads of many items
KMS = Key Management Service
key condition = a rule on which keys a role may read

D. Encryption

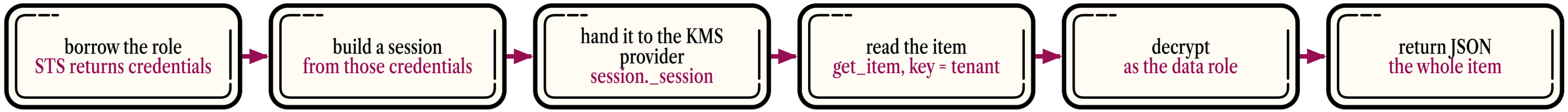


Each client record is encrypted field by field before DynamoDB, AWS's database, stores it, with a key the owner controls in AWS's Key Management Service (KMS).

The subtle part: the encryption library must be handed the borrowed role's session, its set of credentials, or it silently picks the function's own role.

The repo only reads records. Writing encrypted records exists in the design slides.

1. How a Record Is Read



What the Library Does by Default

Every field except the key is encrypted and signed.

The key, id, stays readable so DynamoDB can find the item, but it is signed.

Two extra fields per item hold the encryption details and the signature.

Each item's data key is tied to its table and its id, so it cannot be moved to another record.

Why the Session Matters

Inside Lambda, a library left to find its own credentials uses the function's role.

Then the KMS call would be made as the app account's role. The library gives no warning: the call fails if that role has no key access, as in my committed policy, and quietly crosses the line if it is ever granted access.

I pass the borrowed session on purpose, explain it in the code, and a test checks it.

The library is the DynamoDB Encryption Client for Python 3.3.0, which AWS now lists as a legacy client; its successor is the AWS Database Encryption SDK (software development kit). My career notes named the successor by mistake.

data key = a one-time key, itself encrypted by KMS
signed = any change is detected
session = a bundle of credentials
provider = the part that fetches keys
JSON = structured text

: Order of steps

1. The Silent Default

```
$ python ../checks/sdk_fallback.py
dynamodb-encryption-sdk 3.3.0, material_providers/aws_kms.py
  176:         if botocore_session is None:                          1
  177:             botocore_session = botocore.session.Session()
$ grep -n "botocore_session=" lambda/LambdaFunctionForBaseTable_Client.py | tr -s ' '
104: botocore_session=session._session,                              2
$ pytest tests/test_client_path.py -k kms_provider -q --no-header -rA | tail -2   3
PASSED tests/test_client_path.py::test_kms_provider_receives_the_assumed_role_session_not_the_default
1 passed, 9 deselected in 0.19s
```

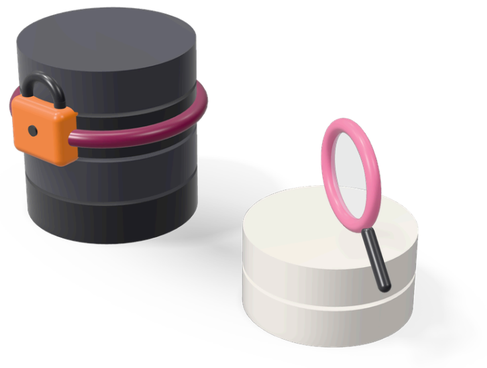
1. In the library, a provider made without a session builds a fresh default one: inside Lambda, the function's own role.
2. My handler passes the borrowed role's session instead, so KMS calls run as the data account's role.
3. A test fails if the provider ever receives anything but that session.

Run on 3 November 2026 in a clean Python environment with dynamodb-encryption-sdk 3.3.0, the version in the function's layer. sdk_fallback.py, written for this deck, prints those two lines of the installed library.

i

SDK = software development kit
botocore = the layer under boto3, AWS's Python SDK
handler = the function's entry point
pytest = Python's test runner

E. The Admin Path



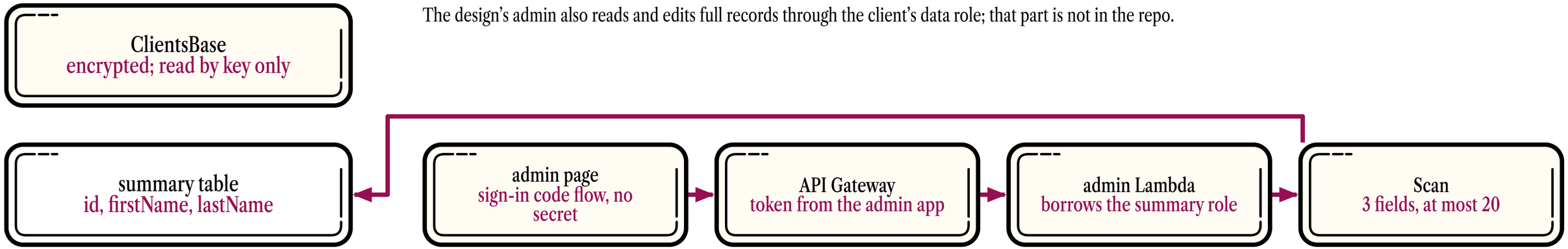
Staff need to see who their clients are, which means listing records: a Scan, the broadest read DynamoDB, AWS's database, offers.

So the list reads only an ID and a name, from a second table, never the encrypted record.

That holds for the list in the repo. The rest of the admin design is not built, and one policy file is wrong.

1. Two Tables, One Narrow List

The design's admin also reads and edits full records through the client's data role; that part is not in the repo.



What Holds

The admin role's data-side Scan is limited to the summary table; I checked it live on 19 Aug 2026.

The API accepts calls from one web origin only (its CORS setting), with two methods and two headers.

Sign-out clears stored tokens before Cognito's logout.

What Does Not

The committed admin role file grants Scan directly and lacks AssumeRole, the call that borrows a role. The written fix is not applied yet.

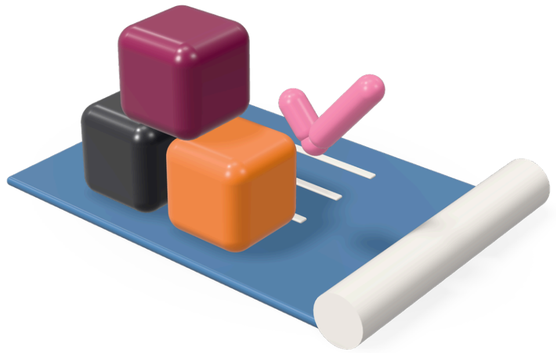
Nothing checks that the caller is an admin, only which sign-in app issued the token.

Names are personal data, though my README calls them non-sensitive.

The list stops at 20 and never asks for more. Nothing fills or syncs the summary table.

: Request

F. Tests, Terraform and CI



Nine months after the build I added what it lacked: 52 tests, Terraform (cloud set-up as code) for both accounts and CI checks that run on every push, in seven commits within one hour on 19 August 2026.

The tests need no AWS account: every AWS call is replaced with a stand-in, so they check the code and the committed settings, not the live system.

Terraform passes its own checks, but nothing shows it was ever applied.

1. Tests and Terraform, Run Today

```
$ pytest -q
.....................................................x......                [100%]
51 passed, 1 xfailed in 0.28s
$ pytest -q --collect-only | tail -1
52 tests collected in 0.06s
$ terraform version | head -1
Terraform v1.13.3
$ terraform fmt -check -recursive terraform/ && echo "formatting: no changes needed"
formatting: no changes needed
$ terraform -chdir=terraform validate -no-color
Success! The configuration is valid.
```

1. 51 pass. The one expected failure is the admin policy file, kept visible on purpose.
2. 52 test cases from 36 test functions. My film says “fifty-one tests”; 51 passing is exact.
3. Terraform 1.13.3, the CI version: formatting clean and the configuration valid.

The CI Pipeline: 3 Jobs

Job	What it runs
test	pytest on Python 3.11 and 3.12
terraform	format check, set-up without state, validate
secrets	gitleaks over the current files only

Not in CI: Python linting, static security analysis, Terraform scanning, a dependency audit, a deploy job.

Run on 3 November 2026 on a clean copy at commit e1df855: Python 3.13, pytest 9.2.0; Terraform 1.13.3 with AWS provider 5.100.0 after set-up without saved state; no AWS credentials.

i

pytest = Python’s test runner
gitleaks = a scanner for committed secrets
linting = automatic style and error checks
state = Terraform’s record of what exists

1. What the Tests and Terraform Cover

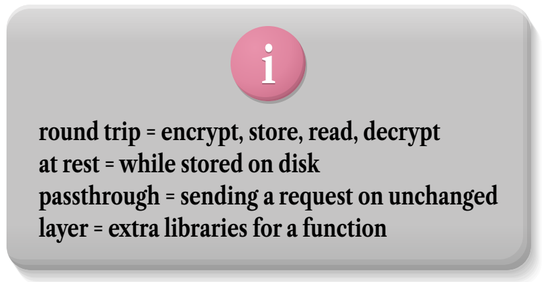
What the tests check

File (cases)	Checks
Policy files (24)	no Action "*", no table wildcard, no writes, one role to assume
API settings (11)	tenant read from the token, eight caller sources absent, the 400 guard present, each sign-in check tied to one app
Client function (10)	borrows the role, hands that session to the KMS provider, keys by the event's ID, never scans
Admin function (7)	scans the summary table only, three fields, at most 20

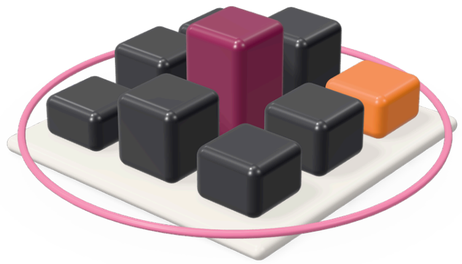
The console build against Terraform

Setting	Console build or design	Terraform
Unmatched bodies	passed through	refused (NEVER)
KMS access	the key's own policy	only through DynamoDB: likely blocks the library
Writing records	not built	no key permission to write
Base-table encryption at rest	an AWS-owned key	the same customer key
Admin API	present	missing
Sign-in	Security account	application account
Python version	layer built for 3.14	functions set to 3.12

Not checked: the passthrough setting, error responses, numeric fields, Terraform, the web page, or a real encryption round trip. Several tests search the source text rather than run it.



G. The Design Beyond the Code

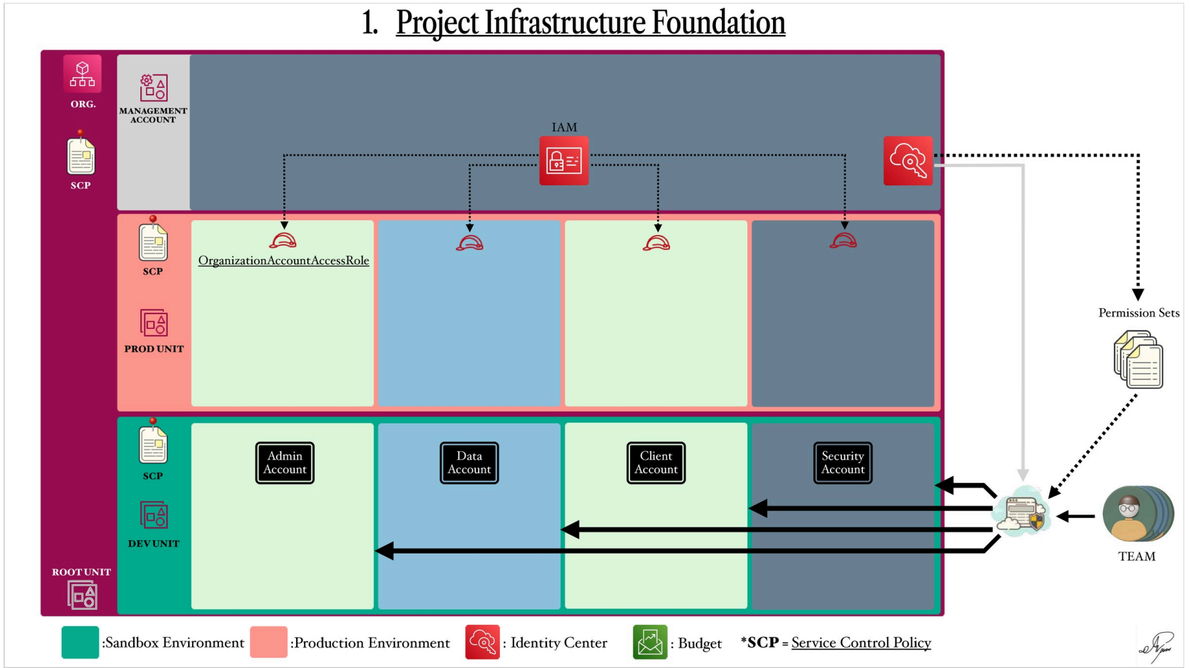


About a third of my 31-slide design deck (10 slides) describes what a small company would need around this core: nine AWS accounts, account-wide limits, single sign-on, budgets, logging and threat detection.

None of it is in the repo as code or settings. It is design work: my README says so, but my design deck's opening slide says I built it with "full logging", which claims too much.

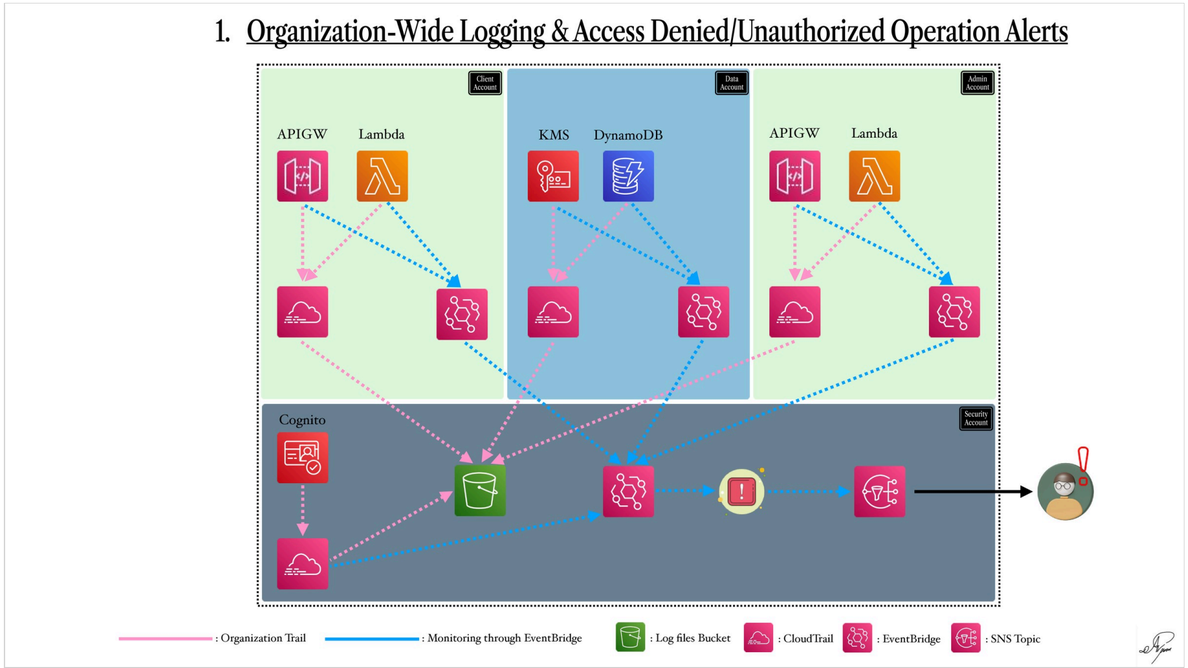
One README line says these "were built by hand in the console"; that line contradicts the rest and should go.

1. Nine Accounts and Org-Wide Alerts, as Designed



Nine accounts: Management, plus Prod and Dev groups each holding Admin, Data, Client and Security accounts.

Service Control Policies cap every account; IAM Identity Center, AWS's sign-in service for staff, gives people single sign-on.



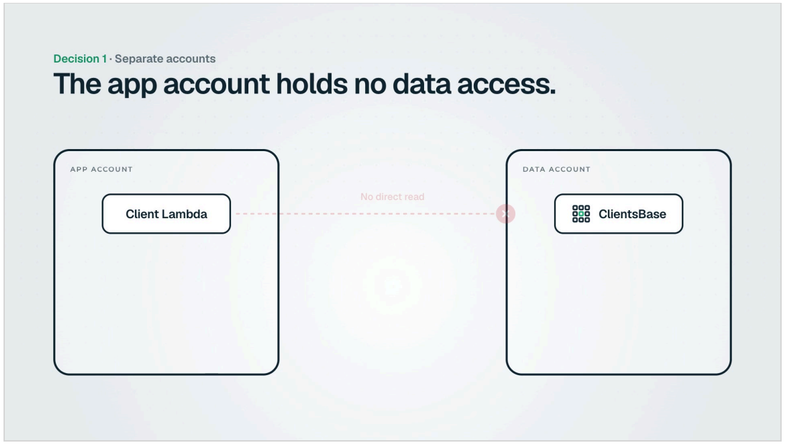
A CloudTrail trail for the whole organization, with tamper checks, into a Security-account bucket.

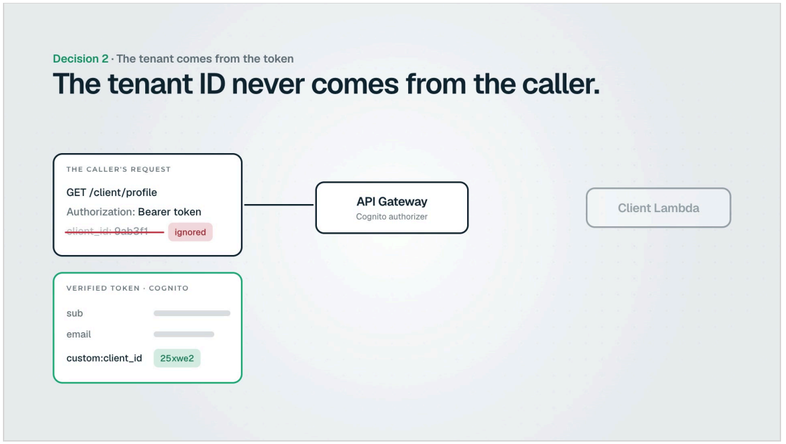
EventBridge sends access-denied events to the Security account and SNS; GuardDuty reads network traffic, domain-name lookups and AWS call records.

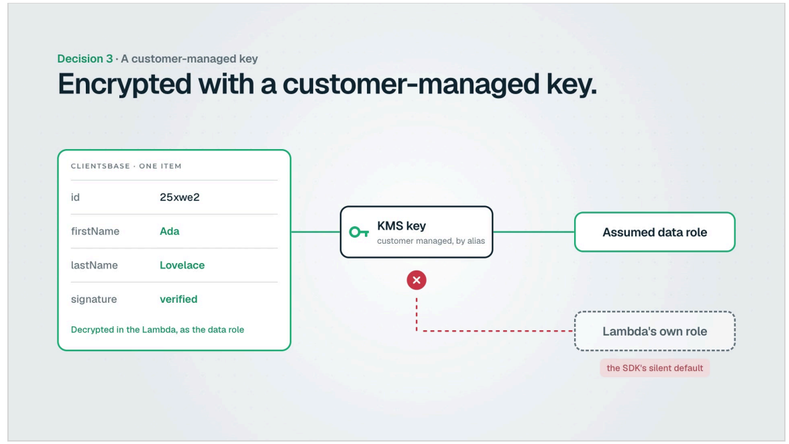
SCP = Service Control Policy, a cap on an account
CloudTrail = AWS's record of API calls
EventBridge = routes events to targets
SNS = AWS's notification service
GuardDuty = AWS's threat detection

Slides 4 and 27 of my 31-slide design deck, from the repo's readme_images; the tamper checks and GuardDuty come from slides 26 and 29. Nothing on this slide was found deployed.

2. The 63-Second Film







The Film

62.5 seconds, 1920 × 1080 at 30 frames a second, 9.0 MB.

Six scenes drawn as web pages (HTML and SVG), animated with HyperFrames and GSAP, two web animation tools; no screen recordings.

Narration voiced with Kokoro-82M, an open text-to-speech model; 16 captions; on GitHub Pages since 2 November 2026, in a commit that credits Claude as co-author.

A claims register lists the source of every on-screen claim.

Lines to Soften

“Fifty-one tests”: 52 tests, 51 passing.

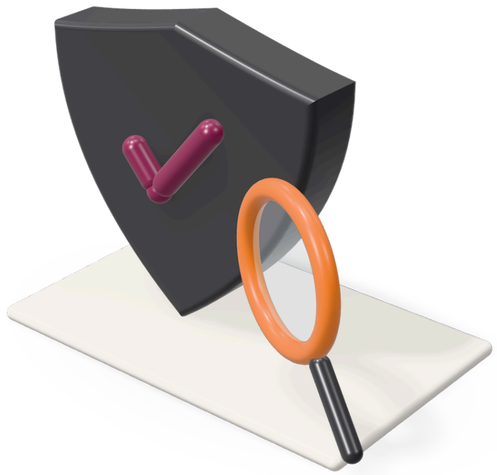
“The app account holds no data access”: it holds permission to borrow the data role.

“The tenant ID never comes from the caller”: true for requests the template handles.

“Breaking into the app doesn’t expose the data”: it can still list IDs and read records (slide C2).

On screen, a token without the claim gets a 400 and the Lambda is “never invoked”: unverified.

H. Proof and Limits



Checked on 3 November 2026: the tests and Terraform's checks run on a clean copy, every policy statement read, the library's default traced, and the history read.

The core ideas hold up in the code; several sentences in my README and notes do not.

The last slides show four problems in the code, set my claims against the repo, list fixes, and say what the project shows.

2. Four Problems, Shown

```
$ grep -n '"passthroughBehavior"' apigw/*/RESTAPIConfiguration_REPLACE_PLACEHOLDERS.json | tr -s ' '
apigw/AdminApp/RESTAPIConfiguration_REPLACE_PLACEHOLDERS.json:45: "passthroughBehavior": "when_no_match",
apigw/AdminApp/RESTAPIConfiguration_REPLACE_PLACEHOLDERS.json:95: "passthroughBehavior": "when_no_match"
apigw/ClientApp/RESTAPIConfiguration_REPLACE_PLACEHOLDERS.json:38: "passthroughBehavior": "when_no_match",
$ grep -n 'passthrough_behavior' terraform/api.tf | tr -s ' '
121: passthrough_behavior = "NEVER" # no template, no request
$ grep -n -A1 'kms:ViaService' terraform/data_account.tf | tr -s ' '
144: variable = "kms:ViaService"
145- values = ["dynamodb.${var.region}.amazonaws.com"]
$ grep -c 'try:' lambda/*.py
lambda/LambdaFunctionForBaseTable_Client.py:0
lambda/LambdaFunctionForSummaryTable_Admin.py:0
$ python -c "import json; from decimal import Decimal; json.dumps({'age': Decimal('16')})" 2>&1 | \
    tail -1
TypeError: Object of type Decimal is not JSON serializable
```

1 The exported client API is set to pass unmatched bodies through, and the function trusts the field. Terraform's NEVER would refuse them.

2 Terraform lets the data role decrypt only through DynamoDB, but the library calls KMS directly.

3 Neither function handles an error, so failures can come back with a success code.

4 Any numeric field, such as an age in my design's sample records, would crash the response.

Run on 3 November 2026 on a clean copy at commit e1df855. Problems 1 and 2 are my reading of the code and AWS's documentation, not tested against AWS. For 3 and 4 the cause is shown directly (no try statements; a Decimal cannot be turned into JSON); the success code in 3 is AWS's documented default for this kind of integration.

i

NEVER = the setting that refuses unmatched requests
kms:ViaService = only through a named AWS service
Decimal = Python's exact-number type

1. My Claims Against the Repo

My notes said	The repo shows	So I now say
Solo, 140 commits	Solo author on all 142; the film commit credits Claude as co-author. 133 through Nov 2025, the LinkedIn figure; 140 from 19 Aug to 31 Oct 2026	142 commits under my name, one co-authored with Claude; about a third touch code
Repo created Nov 7, 2025	Built 6–26 Nov 2025; tests, Terraform and docs Aug 2026; film Nov 2026	Built Nov 2025, extended Aug and Nov 2026
Each role scoped to one DynamoDB action on one table	The client data role has two actions on one table; every file has a wildcard log rule	Data access limited to one table per role
Encrypted with the AWS Database Encryption SDK	The code uses the DynamoDB Encryption Client 3.3.0, now a legacy client	Field-level encryption with the DynamoDB Encryption Client
Crypto bound to the assumed role	KMS calls run with the borrowed role's credentials; there is no cryptographic link	KMS calls run as the borrowed data role
Tenant ID never from client input	True for requests the template handles; the exported API is set to pass unmatched bodies through (untested)	Tenant ID from the verified token, for templated requests
Admin view's Scan never reaches protected fields	True for the list; names are personal data; the admin role file is a wrong copy	The admin list reads only ID and name
Targets "the application tier holding standing access to the data"	It holds standing permission to assume the data role	No direct data permission; one role to borrow
Landing zone, logging and GuardDuty: design, not deployed	Correct; one README line says they were built in the console	Design only, and fix that README line

From my career-hub notes, checked against the repo on 3 November 2026. KMS = Key Management Service; GuardDuty = AWS's threat detection; SDK = software development kit; DynamoDB = AWS's key-value database; landing zone = a ready-made set of accounts and limits.

2. What to Fix Next

1. **Scope each session to one tenant**
   Get the data credentials from the user's verified token, for example a Cognito identity pool that tags the tenant, and limit DynamoDB keys to that tag.

2. **Close the passthrough**
   Set the console-built client API to refuse unmatched bodies, as Terraform already does, and test it.

3. **Fix the admin role file**
   Read the live policy, re-export it, and turn the expected failure into a pass.

4. **Fix Terraform's key grant**
   Allow the library's direct KMS calls and writing, then apply it once and record the result.

5. **Handle errors**
   Return not-found and forbidden codes (404, 403), map them in API Gateway, and convert numbers.

6. **Stop logging tenant data**
   Log the request ID, not the whole event, and stop printing the function's runtime details.

7. **Clean up what is public**
   Redact the two token screenshots; decide whether to purge the old account ID and sign-in and API identifiers from history.

8. **Make the README match**
   Drop the lines saying there is no Terraform or tests, that the design was built in the console, and "It does not raise"; say 51 passing.

In order of what each protects: the data first, then the API, then correctness, then what is said about it. A tag set inside the function would not do for fix 1: a compromised function picks its own tags.

3. What It Shows, and What It Does Not

It Shows

A clean account line: each function reaches data only by borrowing one narrow role (the admin role file needs re-exporting).

The tenant taken from a verified token, for requests the template handles.

A subtle library pitfall found, explained and checked by a test.

52 offline tests, valid Terraform and CI, added after the build.

It Does Not Show

That it runs today: the evidence of a running system is from November 2025, plus my August 2026 check of two roles.

Isolation that survives a compromised app: the admin role lists every ID, and the client role reads any record by its ID.

Writing records, the client page or the monitoring design, in code.

That the Terraform works when applied.

The core is small and careful. The next step is making the boundary hold even when the function itself is compromised, and making my README say exactly what the code does.