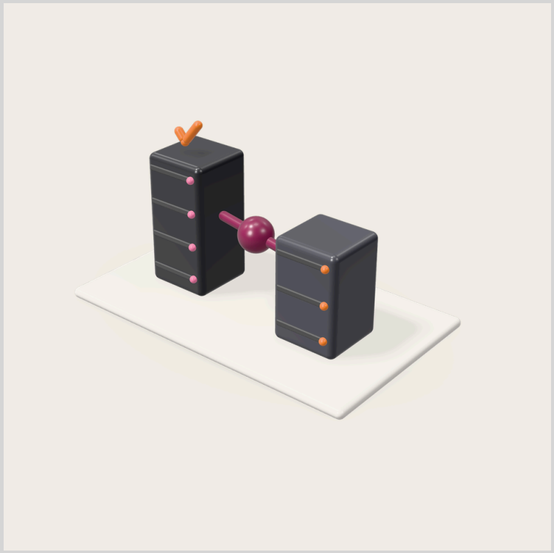
© 2026 Yiguedan Philippe Andy Yaro



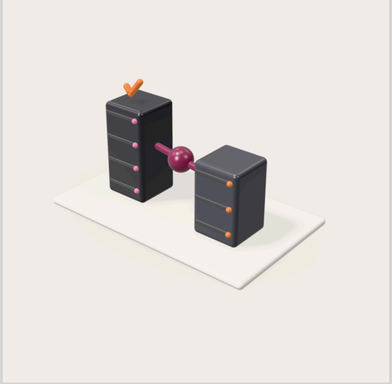
InternPass: RKE2 on EC2, and a Photo Checker

Yiguedan Philippe Andy Yaro

View GitHub Repository: github.com/andyyaro/serverless-profile-photo-checker

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

LinkedIn: linkedin.com/in/andy-yaro



In summer 2025 I was a cloud-computing research intern with InternPass, mentored by Justin Wilson. Following the program's setup steps and the RKE2 quick-start guide, I built a two-node RKE2 cluster, Rancher's version of Kubernetes, the system that runs containers across servers, on two Amazon EC2 virtual servers, and ran a small Python app on it.

On 7 July I committed a serverless profile-photo checker, code AWS runs on demand: Lambda runs it, API Gateway takes the web requests, Rekognition reads the faces, and 14 Terraform resources define it. In August 2026 I copied it from my private fork to a public repo.

On 3 November 2026 this deck reruns the app with a stand-in for its quotes service, and the checker with stand-ins for AWS, and reads the Terraform and history. No AWS account was used. I am Yiguedan Philippe Andy Yaro.

A. The Internship

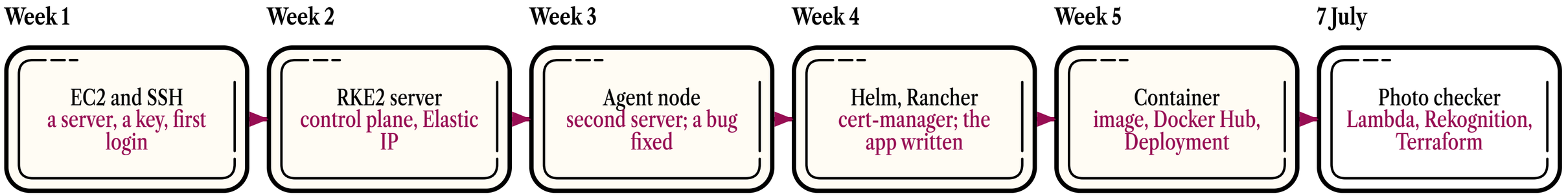


The program's repo opened on 27 May 2025 with a one-line description: deploy a Python application on an RKE2 cluster of AWS EC2 servers.

In the history my fork keeps, three interns each had a folder and sent pull requests, proposed changes, to the mentor's repo. My 2025 commits run from 18 June to 12 July, US Eastern time.

The first five weeks built the cluster and the app. A second project, a serverless photo checker, came after.

1. Five Weeks, Then a Second Project



Who Did What

Mentor, Justin Wilson: created the repo and the security group, set up the DNS record, and merged my first pull request.

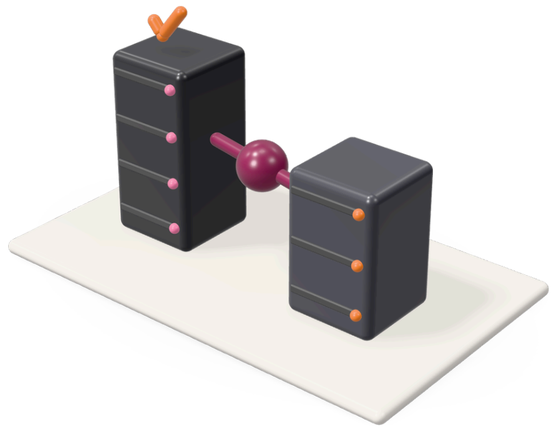
Two other interns: the Week 1 setup steps, a note with SSH tips, an RKE2 agent install script, and their own copies of the starter app.

Me: both servers, the cluster, Helm, cert-manager and Rancher, the container and the Deployment, and the photo checker.

EC2 = Amazon's virtual servers
SSH = a secure remote login
control plane = the programs that manage a cluster
Elastic IP = a fixed public address
security group = an AWS firewall for servers
DNS record = maps a name to an address
Helm = a Kubernetes package installer

Weeks and the cluster work from my June 2025 blog post, copied into my fork; files and pull requests from the repo's history. My August 2026 public README says the cluster was not my work; this deck follows the 2025 blog. The program's official dates are in neither repo.

B. Two Servers, One Cluster

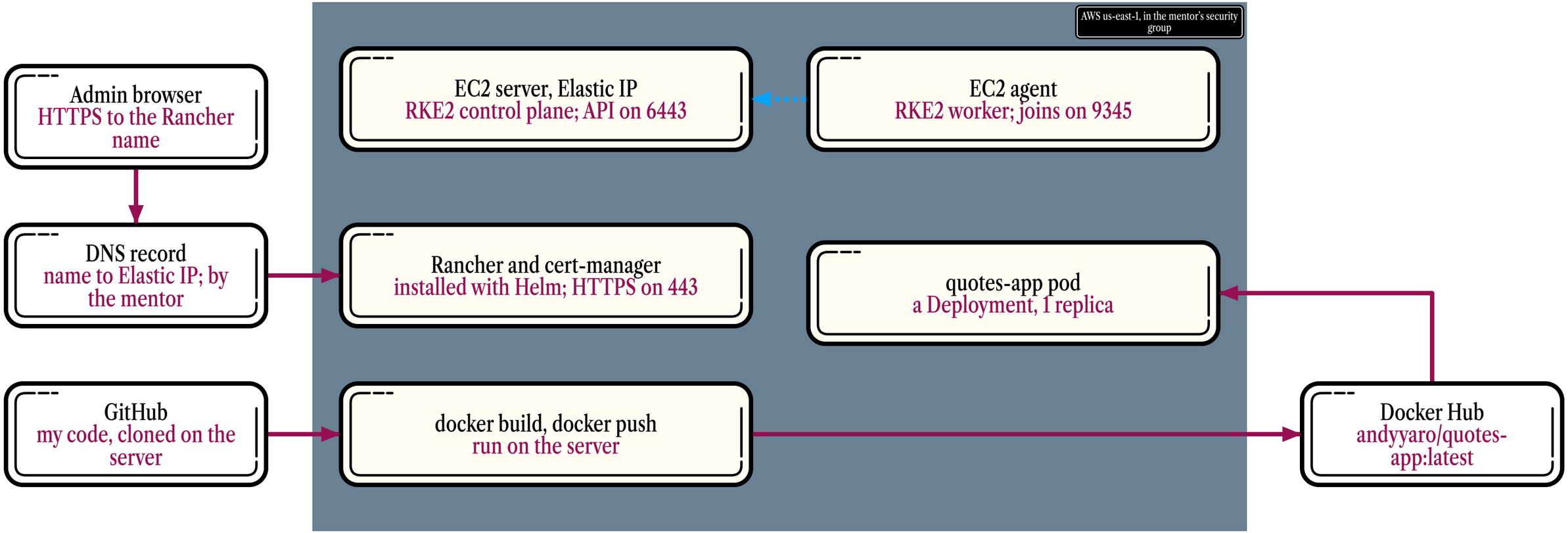


Two t2.xlarge Ubuntu servers, an EC2 size, in us-east-1, AWS's Northern Virginia region: one runs the RKE2 control plane, the other joins it as a worker on port 9345.

The agent's first config pointed at its own address, and the service hung. Pointing it at the server's private address fixed it.

Helm installed cert-manager, which issues TLS certificates, then Rancher, a web console for the cluster. An Elastic IP kept the address fixed.

1. How the Cluster Fit Together

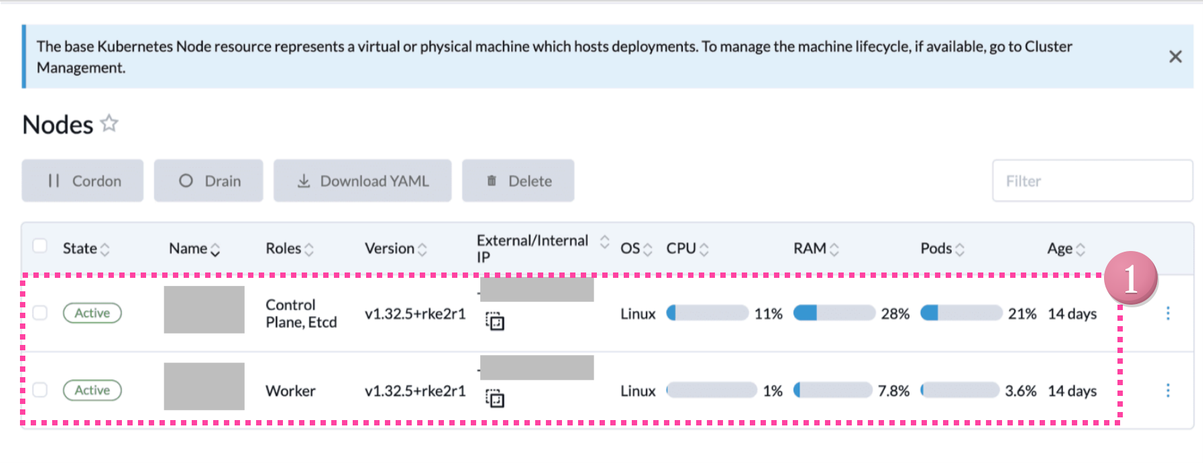


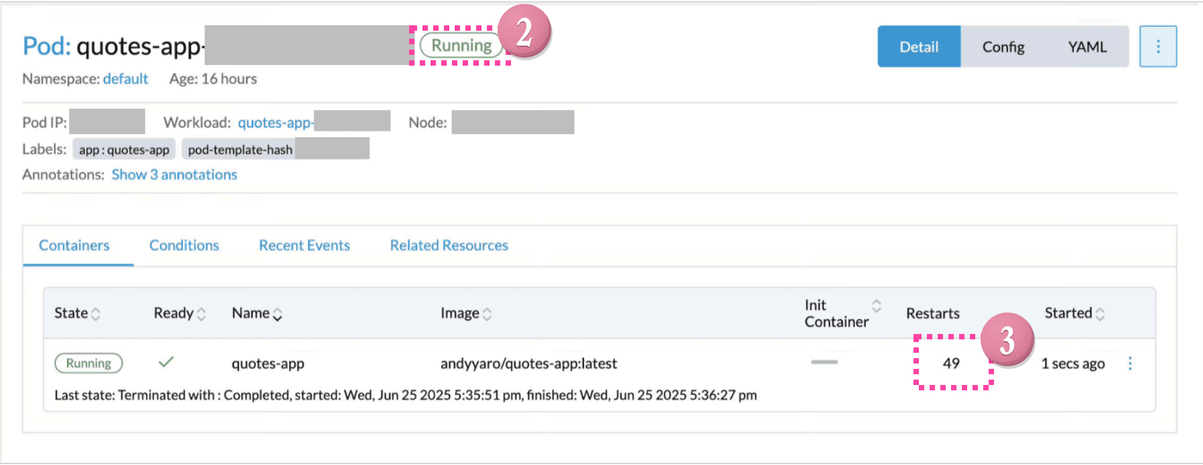
The app listens on no port and has no Service or Ingress: it only prints to its log. My notes and the first film say Route 53, AWS's DNS service, but no 2025 source names the DNS provider.

Service = a stable network address for pods
Ingress = routes outside web traffic in
image = the packaged app a container runs
Deployment = keeps a set number of pods running
TLS = the encryption behind HTTPS

: AWS : Traffic or a build step : Agent joins the server

2. The Cluster, Running in June 2025





1. Both nodes Active on RKE2 v1.32.5: one control plane, one worker.

2. The quotes-app pod Running, from my image on Docker Hub.

3. 49 restarts by the time the pod was 16 hours old: the script ends after five quotes, and Kubernetes restarts the container, as it does for every Deployment's pods.

Screenshots from my June 2025 blog post. Node names, addresses and hashes were only pixelated then; here they are covered with solid boxes. The cluster no longer exists, so none of this could be re-run.

i

Rancher = a web console for Kubernetes clusters
etcd = the database that stores the cluster's state
pod = the smallest unit Kubernetes runs

C. The App and Its Container



The app is a 25-line Python script built from reference code the program provided: it calls ZenQuotes, a public quotes service, five times, seven seconds apart, and prints the first quote from each answer.

My Dockerfile, the recipe for the container image, installs the Python libraries before copying the code, so a code change reuses the cached install.

I built the image on the server, pushed it to Docker Hub, and applied a one-replica Deployment with kubectl.

1. Five Quotes, One Replica

```
$ python3 -I ../../checks/quotes_stub.py
quote 1
quote 2
quote 3
quote 4
quote 5
GET calls: 5 to ['https://zenquotes.io/api/quotes']; sleeps: [7, 7, 7, 7, 7] = 35 s; then it exits
$ python3 -I ../../checks/k8s_docker.py
Deployment quotes-app: replicas 1, image andyyaro/quotes-app:latest, pull Always
not set: resources, livenessProbe, readinessProbe, securityContext, ports
Dockerfile steps: FROM python:3.9-slim | WORKDIR /app | COPY requirements.txt . | RUN pip
  install --no-cache-dir -r  | COPY quotes.py . | CMD ["python", "quotes.py"]
```

1. Five calls to the quotes service, seven seconds apart: 35 seconds, then the script ends.
2. One replica, checked against Docker Hub at every start. No resource limits, health checks or security settings.
3. Requirements go in before the code, so that layer stays cached.

```
                                                  # kubectl logs quotes-app-
To be totally at leisure for one day is to be immortal for one day.
Being wrong opens us up to the possibility of change.
Blessed is he who expects nothing, for he shall never be disappointed.
First learn the meaning of what you say, and then speak.
However difficult life may seem, there is always something you can do and succeed at.
```

Above: run on 3 November 2026 with quotes_stub.py and k8s_docker.py, written for this deck, with requests.get and time.sleep replaced by recorders, so no network and no waiting. Below it: the pod's log from my June 2025 screenshots.

i

replica = one running copy of a pod
layer = one cached step of an image
health check = a probe of whether a container is alive or ready

2. A Script Is Not a Service

What a Deployment Does

A Deployment keeps a set number of pods running, and their containers restart whenever they stop.

The script stops by design after 35 seconds, so Kubernetes starts it again: 49 restarts in 16 hours.

When its output is not a terminal, Python holds it back, so a container run showed nothing until the script ended. My blog records that puzzle.

What Would Fit Better

A Job, which runs a pod once to completion, or a CronJob, which runs it on a schedule.

Unbuffered output, PYTHONUNBUFFERED=1, so lines appear as they print.

A pinned image tag instead of latest, and resource limits.

What Was Mine

The Dockerfile, the manifest, and a .gitignore after a 133 MB image export went over GitHub's 100 MB file limit.

Cutting the wait from 60 to 7 seconds.

The script itself came from reference code: all three interns' copies are nearly the same.

From the manifest, the Dockerfile and quotes.py in my folder, the screenshots, my June 2025 blog post, and the three interns' folders in the repo's history. A manifest describes Kubernetes objects; a tag is an image's version label, latest by default; a .gitignore lists files Git leaves out.

D. A Serverless Photo Checker

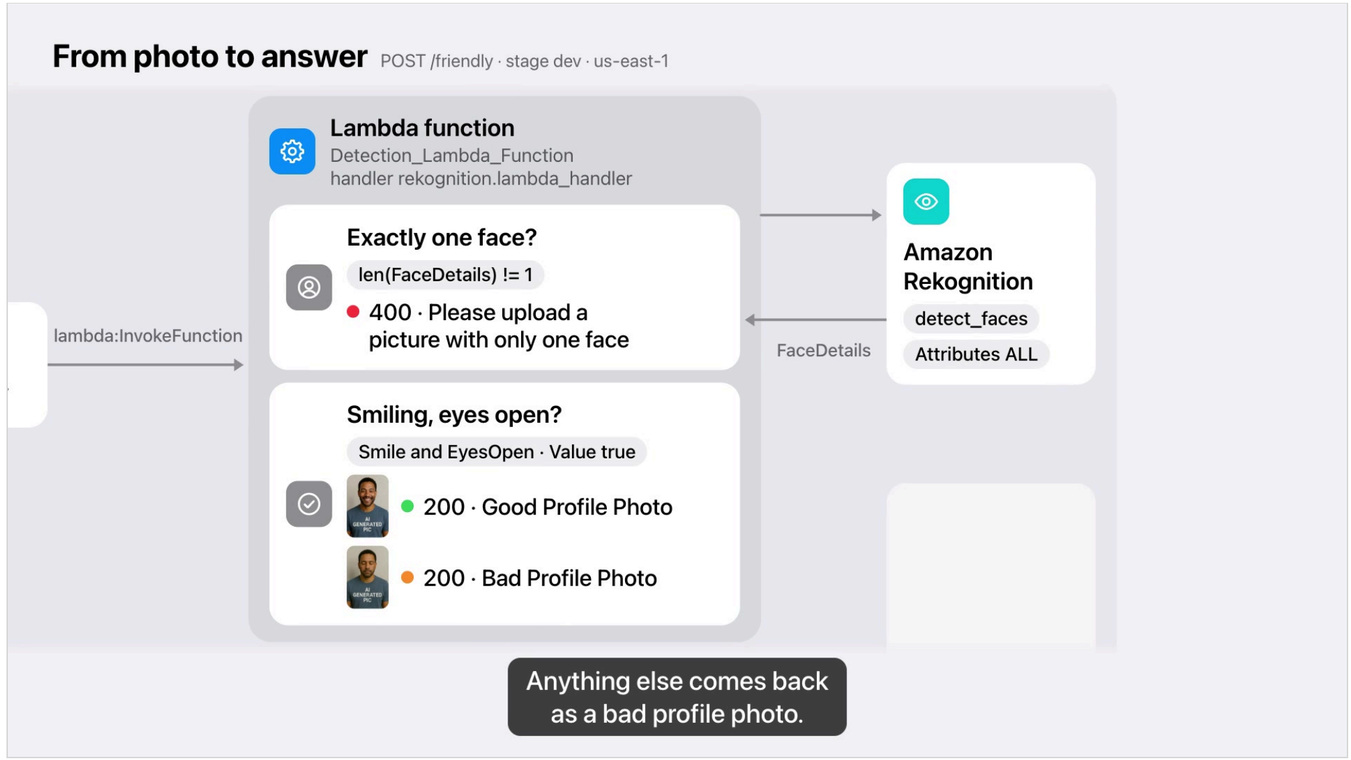


The question it answers: is this photo usable as a profile photo? A small client program sends the image, and AWS answers.

API Gateway, AWS's service for web requests, hands it to a Lambda function, which asks Amazon Rekognition, AWS's image analysis, to describe the faces. Exactly one face, smiling, with eyes open, is a good photo.

It builds on an AWS example; I added the one-face check and the smile-and-eyes rule.

1. From Photo to Answer



A frame from the photo checker's launch film. The faces are AI-generated test images from the repo.

Lambda = AWS runs a function per request
API Gateway = AWS's service that receives web requests
Rekognition = AWS's image analysis service
base64 = binary data written as text
POST, stage = a request that sends data; a published API version
HTTP 200, 400 = the web's codes for success and a bad request

1. interact.py base64-encodes the image and sends it in a POST request to /friendly on the dev stage.
2. API Gateway, with no sign-in, invokes the Lambda function, on Python 3.8.
3. The handler calls Rekognition's DetectFaces with all attributes.
4. Not exactly one face: 400, "Please upload a picture with only one face".
5. Smile and EyesOpen both true: "Good Profile Photo"; otherwise "Bad Profile Photo".
6. When the function answers, API Gateway returns HTTP 200, the only response set up; the real status sits in the body.

2. The Handler and Client, Run

```
$ python3 -I ../checks/handler_stub.py
boto3.client() called at import: ['rekognition'] | detect_faces asks for ['ALL']
1 face, smiling, eyes open      -> 200 "Good Profile Photo"
1 face, no smile, eyes closed   -> 200 "Bad Profile Photo"
1 face, smiling, eyes closed    -> 200 "Bad Profile Photo"
2 faces                         -> 400 ValueError: Please upload a picture with only one face
0 faces                         -> 400 ValueError: Please upload a picture with only one face
1 face, no Smile field          -> UNHANDLED KeyError: 'Smile'
$ python3 -I ../checks/client_loopback.py
run: python interact.py <url> goodphoto.jpeg   (exit 0)
  "Good Profile Photo"
run: python interact.py <url> badphoto.png   (exit 0)
  "Bad Profile Photo"
run: python interact.py --image goodphoto.jpeg --url <url>   (exit 2)
  interact.py: error: unrecognized arguments: --image --url
request bodies the stand-in received: ['POST /dev/friendly, 558,905 bytes', 'POST /dev/friendly, 3,474,621 bytes']
```

1. One face, smiling, with eyes open is good; any other single face is bad.
2. If a face record ever lacked a Smile field, the handler would crash: the code never checks.
3. The real client, against a local stand-in for API Gateway, prints the answer.
4. The README's run command fails: the client takes no --image or --url flags, and there is no api_url output.

Run on 3 November 2026 with handler_stub.py and client_loopback.py, written for this deck. A stub decides what faces Rekognition reports, so this checks the code's logic, not what Rekognition would say about the photos. No AWS call was made; the stand-in listened only on this machine.

i

stub = a fake that returns chosen answers
stand-in = a local server playing API Gateway
KeyError = Python's error for a missing field

3. What Is Mine, and What Is AWS's

The handler, rekognition.py, 81 lines



Lines 37–49 hold the one-face check and the smile-and-eyes rule: 9 lines of code. The rest follows the structure of AWS's example "Analyzing images with an AWS Lambda function", with the call changed to DetectFaces. Git cannot show who wrote which line: the file arrived complete in one upload.

Also From AWS

The client, interact.py: its argument parsing, base64 payload and status branch follow the same example's client, changed to an HTTP request, per my November 2026 film notes.

The README credits AWS for the handler only; it should credit the client too.

The Terraform

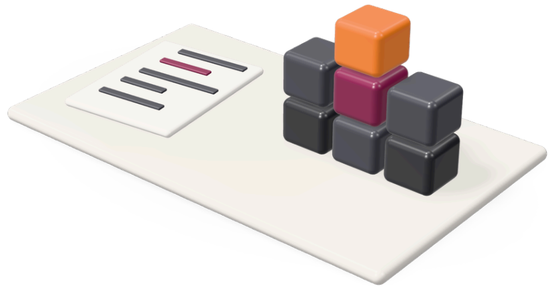
Names such as zip_the_python_code and codedeploy_service_role_policy_attach match public Terraform tutorials; Spacelift's come up in searches, but I have not confirmed the source.

My visible changes, on 7 July 2025: moving the stage into its own resource, the 14th; a trigger that redeploys the API when its parts change; and pointing the output at the stage's URL.

I also moved the workflow file to the repo root, where GitHub Actions looks for it.

Terraform defines cloud resources in text files; GitHub Actions runs automated jobs, called workflows, from a repo.

E. Infrastructure as Code



Terraform, a tool that creates cloud resources from text files, defines the checker's AWS side: 14 resources in two files.

A role lets the function write logs and call Rekognition's read-only actions. The API takes requests from anyone, with no key, sign-in or per-client limit.

A GitHub Actions workflow in my fork was written to apply it on demand, but nothing in either repo shows it ever ran or that the checker was deployed.

1. 14 Resources, Read as Text

```
$ python3 -I ../checks/tf_static.py
blocks: {'resource': 14, 'output': 1, 'provider': 1, 'data': 2}
resources by service: api_gateway 8, iam 4, lambda 2
types: api_gateway_deployment, api_gateway_integration, api_gateway_integration_response,
  api_gateway_method, api_gateway_method_response, api_gateway_resource, api_gateway_rest_api,
  api_gateway_stage, iam_policy, iam_role, iam_role_policy_attachment x2, lambda_function,
  lambda_permission
lambda.tf:39: "Resource": "arn:aws:logs:*:*:*",
apigw.tf:20: authorization = "NONE"
lambda.tf:72: runtime                         = "python3.8"
terraform { } block (version pins, backend): none
source_code_hash: none
API key, usage plan or authorizer: none
```

1. 14 resources: 8 for API Gateway, 4 for roles and policies, 2 for Lambda.
2. The log policy covers every log group in every region, and the API method needs no authorization.
3. No pinned versions or shared state, no code hash to trigger redeploys, and no API key or usage plan.

Run on 3 November 2026 with tf_static.py, written for this deck: it reads the .tf files as text. There was no Terraform program here, and nothing was planned or applied. The Lambda runtime, Python 3.8, is one AWS no longer supports.

i

IAM = AWS's users, roles and permissions
state = Terraform's record of what it created
code hash = a fingerprint that marks code as changed
usage plan = API Gateway's per-key rate and quota limits
provider = Terraform's plug-in for one platform
data source = something Terraform reads, not creates

2. The Deploy Workflow

The Workflow, main.yml

Runs only when started by hand.

Terraform 1.5.7 with long-lived AWS access keys from the repo's secrets.

init, plan (a preview), then apply (the changes) with no confirmation.

What It Lacks

No shared state: each run starts with an empty record, so a second run would try to create everything again and fail on names already taken.

No short-lived credentials through OpenID Connect.

No record in either repo that it ever ran; GitHub's run history was not checked.

Evidence of a Deployment

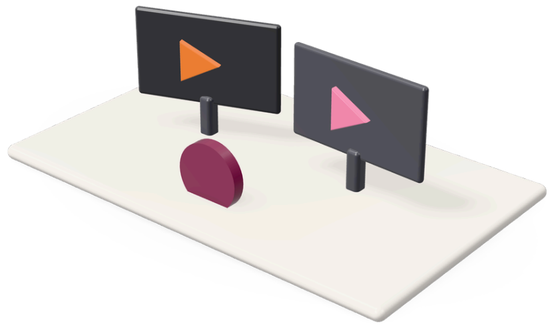
None: no API address, state file, log or screenshot in either repo.

The film claims no deployment; its watch page says nothing was deployed to make it.

So the checker is written and checked offline, not shown running.

From .github/workflows/main.yml in my fork, and a search of both repos and their histories. OpenID Connect lets a workflow assume an AWS role with short-lived credentials instead of storing keys.

F. History and the Films



My fork's history: the mentor, two other interns and me in the team repo from 27 May to 25 June 2025, then my own commits to 12 July.

My 32 commits in 2025 include three made as root, the Linux administrator account, on EC2 servers: two from the server where I built and pushed the image, one adding the checker's client Dockerfile.

In November 2026 I added two short launch films, one for each project.

1. Who Committed What, and When

```
$ git -c mailmap.file=../checks/interns.mailmap log --use-mailmap --date=format:%Y-%m-%d \
    --format='%ad %aN' | sort | uniq -c
      1 2025-05-27 Justin Wilson
      1 2025-06-02 intern B
      3 2025-06-03 intern A
      5 2025-06-17 intern B
      1 2025-06-18 Andy Yaro
      1 2025-06-18 Justin Wilson
      2 2025-06-19 Justin Wilson
      4 2025-06-24 intern A
      3 2025-06-25 Justin Wilson
      1 2025-06-25 andyyaro
      1 2025-06-25 intern A
      2 2025-06-25 root
      5 2025-06-26 andyyaro
     19 2025-07-07 andyyaro
      1 2025-07-07 root
      3 2025-07-12 andyyaro
      1 2026-11-01 Andy Yaro
      1 2026-11-02 Andy Yaro
$ git -c mailmap.file=../checks/interns.mailmap log --use-mailmap --merges --date=short \
    --format='%ad %aN: %s'
2025-06-25 Justin Wilson: Merge pull request #4 from jawhster/jw-dev
2025-06-25 andyyaro: Merge pull request #3 from jawhster/yyaro-dev
2025-06-18 Justin Wilson: Merge pull request #2 from jawhster/yyaro-dev
2025-06-17 intern B: Merge pull request (after moving my file into the folder)
$ git log --author=root --date=short --format='%ad %h %s'
2025-07-07 73a41d3 Add Dockerfile for python image-analyzer app
2025-06-25 7efa78d Updated quotes.app
2025-06-25 548e5cf Deploy Python quotes app to Kubernetes with Docker (exclude tar file)
$ git log --until=2025-12-31 --format=%an | grep -cE '^(andyyaro|root|Andy Yaro)$'
32
```

1. My 2025 commits under three names: Andy Yaro from my laptop, andyyaro from GitHub's website, and root on EC2 servers.

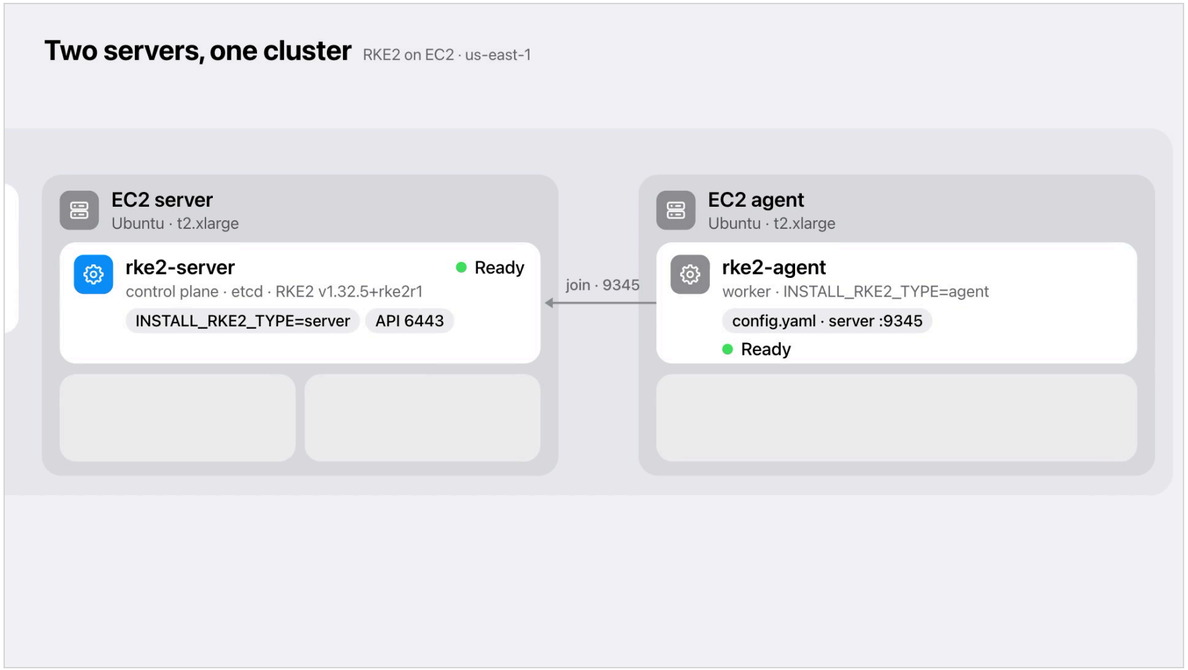
2. Two of my merged pull requests are in this history: #2, merged by the mentor, and #3, which I merged. GitHub also lists #5, which only removed files.

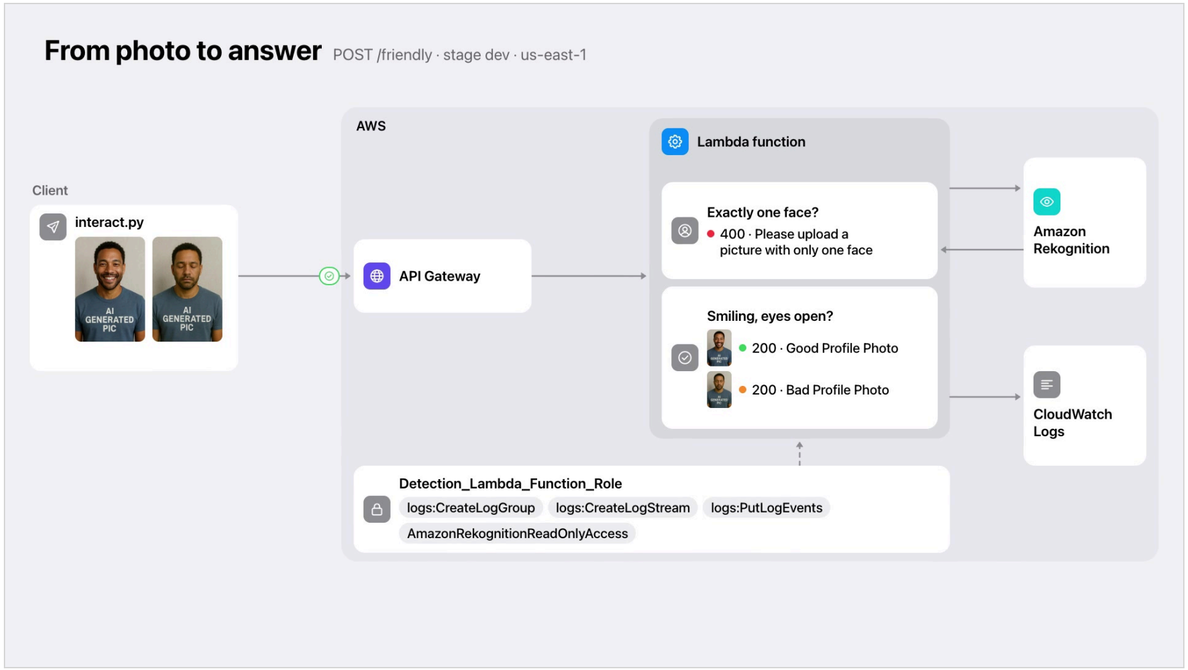
3. The root commits change only my folder. The two from June match my blog: the image built on the server, the large file left out.

4. 32 commits in 2025, over five days: 15 add or change code or configuration; the rest delete files, upload test photos or merge.

Run on 3 November 2026 on a clone of my fork, with the other interns' names replaced by a mailmap file. Dates as recorded: US Eastern, except root's, in the servers' UTC. On 26 June 2025 I removed the other interns' folders and the shared notes from my fork; they remain in its history.

2. Two Launch Films





RKE2 on AWS

68.0 seconds, 8.5 MB; a release file in my private fork.

It repeats Route 53, which no 2025 source supports, and counts three merged pull requests, one of which only removed files.

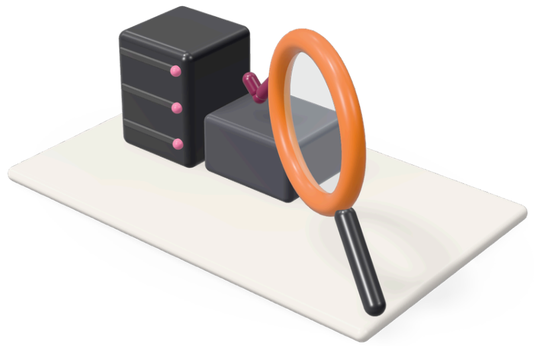
Serverless Profile-Photo Checker

66.3 seconds, 10.9 MB; andyyaro.github.io/serverless-profile-photo-checker/film.html, which loaded on 3 November 2026.

It claims no deployment.

Both are animated boards built with HyperFrames, an open-source tool that renders web pages to video, and GSAP, a JavaScript animation library, narrated by Kokoro-82M, a text-to-speech model, and made in Claude Code sessions; each film commit credits Claude Opus 5.5 as co-author.

G. Proof and Limits



Checked on 3 November 2026: the app with a stand-in for its quotes service, the photo checker's code with stand-ins for AWS, the Terraform as text, and the history.

The cluster is shown by June 2025 screenshots; the photo checker was never shown deployed.

The last slides set my notes against the repos, list risks and fixes, and say what the work shows.

1. My Claims Against the Repos

My notes said	The repos show	So I now say
Approx. Jun to Jul 2025	My commits run 18 June to 12 July 2025; the repo opened 27 May; my blog's five weeks end by 25 June	Late May to July 2025; confirm with official dates
Provisioned an RKE2 cluster on EC2	My 2025 blog: I built both nodes from the RKE2 quick-start; the mentor set up the security group and DNS. My 2026 public README says otherwise	Built a two-node RKE2 cluster
Traffic through Elastic IP and Route 53	Elastic IP confirmed; no 2025 source names Route 53	An Elastic IP and a DNS record
Containerized a Python quotes API	A 25-line script calling the ZenQuotes service, from the program's reference code	Containerized a small Python script
Deployed with Helm and a manifest	Helm installed cert-manager and Rancher; the app went in with kubectl	Helm for Rancher; kubectl for the app
3 merged PRs to the team repo	GitHub lists three merged; #2 and #3 carry my code; #5 only removed files	Two merged pull requests with code
28 commits in my fork	32 commits in 2025 under three names; 28 is the count for one name	32 commits, 15 changing code or configuration
An attribute classifier	A rule on two attributes: smiling and eyes open	A smile-and-open-eyes rule
A least-privilege logging policy	Three log actions on every log group in every region	A logging policy: three actions on all log groups
14 Terraform resources	Correct, plus 2 data sources	As stated

From my career-hub notes, checked against both repos on 3 November 2026. PR = pull request; least privilege = granting only the permissions a task needs.

2. Risks Found in This Review

1. **Real people in the fork's history**
The first test photos, uploaded on 7 July 2025, show apparently real people, source and consent unknown. AI images replaced them on 12 July; history keeps them.

2. **Screenshots were only pixelated**
Node names, addresses and the shell prompt can sometimes be recovered from pixels. This deck uses solid boxes; my live blog post still shows the pixelated ones.

3. **The API is open to anyone**
No key, sign-in or per-client limit: anyone with the address could cause Lambda and Rekognition charges; only AWS's account-wide limit applies.

4. **Errors look like successes**
When the function answers, API Gateway returns HTTP 200, even for a 400 in the body; a missing Smile field would crash the handler.

5. **An out-of-date stack**
Lambda on Python 3.8, which AWS no longer supports; images on Python 3.9, past its October 2025 end of life; requests 2.26.0.

6. **A workflow that cannot run twice**
With no shared state, a second run would fail on names already taken; it relies on long-lived keys in the repo's secrets.

7. **Personal data in commit history**
Personal email addresses, EC2 private host names and Claude session links in commit details, including both public-repo commits.

8. **The public README is out of date**
POST / for /friendly, a failing run command, "least-privilege", a wrong claim about error codes, and "provisioning that cluster was not my work".

Found in this review. The cluster is gone and nothing in the repos shows the checker deployed, so the API risks apply to any future deployment.

3. What to Fix Next

1. **Purge the real photos**
   Rewrite the fork's history to remove them before it is ever made public.

2. **Correct the public README**
   The path, the run command and output name, the logging and error-code wording, and who built the cluster.

3. **Protect the API**
   IAM authorization with signed requests, or an authorizer function, plus a usage plan with throttling.

4. **Return real status codes**
   Map 400s to HTTP 400, and check that each face field exists.

5. **Upgrade the stack**
   A current Lambda Python runtime, newer images and requests, and pinned providers.

6. **Fix the workflow**
   Shared remote state, short-lived OpenID Connect credentials, and a plan to review.

7. **Run the script as a Job**
   A Job or CronJob, unbuffered output, a pinned image tag and resource limits.

8. **Clean up history and notes**
   No-reply email and no session links in both repos, solid boxes in the live blog, and the fork's README and film corrected.

Roughly in order of risk: people's privacy first, then wrong public claims and an open API, then upkeep.

4. What It Shows, and What It Does Not

It Shows

Building a two-node Kubernetes cluster on EC2 by hand, from a guide, and debugging its join.

Packaging and deploying a container, with a cache-friendly Dockerfile.

A serverless API defined in Terraform, with a face rule of my own.

It Does Not Show

That the photo checker was ever deployed.

Safe defaults: an open API, a broad log policy, old runtimes.

Automated tests: there are none.

Network design: the mentor set up the security group and DNS.

A first hands-on cloud project, done step by step and written up in a 2025 blog post. The next step is to deploy the checker safely, with a protected API, a current runtime and tests, and to keep real people out of the history.