Cariad Eccleston

Exmouth, Devon, United Kingdom
cariad@cariad.earth
https://linkedin.com/in/cariad
https://github.com/cariad

SUMMARY

Hello! I'm Cariad — pronounced *carry-add*.

I'm a senior software engineer at digiLab in Exeter, United Kingdom, where I build Python backends, deploy infrastructure to AWS, and look after security.

WORK EXPERIENCE

January 2025 - Present

Senior Software Engineer

digiLab, Exeter, United Kingdom

Uncertainty Engine. I contribute to the design, development, and reliability of the *Uncertainty Engine*: a SaaS machine learning platform that models complex simulations and quantifies uncertainty, so safety-critical industries like fusion energy can make decisions from sparse or noisy data.

AWS, Docker, Python

Infrastructure. I designed the platform's architecture for over 50 queue-driven services on ECS Fargate in private subnets, reaching AWS services through VPC endpoints, and for public APIs behind API Gateway with Cognito authorisers.

After prototyping with hand-written CloudFormation to verify the design, I built the AWS CDK application that launched production and now deploys every customer's environment.

I gave every service an IAM role and security group scoped tightly to its work, and I set up CloudWatch alarms that monitor availability, errors, and latency. These alarms have pinpointed several production incidents, like requests outlasting API Gateway's timeout, and helped the team to deploy effective fixes quickly.

I documented the delivery process, and my colleagues now create new customer environments from that documentation alone. I also cut the process from days down to hours, so the team can deliver environments on the day they're requested.

API Gateway, AWS, CDK, CloudFormation, CloudWatch, Cognito,
ECS Fargate, IAM, infrastructure as code, monitoring, SQS, VPC

Security. I implemented technical controls for digiLab's SOC 2 audit: API monitoring with CloudWatch alarms, S3 access logging, VPC flow logs, AWS Backup for customer data, and a disaster recovery runbook to restore those backups. When the audit needed evidence, I deleted my own production data, then followed the runbook to restore it. I monitored the controls and collected their evidence in Sprinto, our compliance platform. digiLab passed the audit.

I made AWS Secrets Manager our standard for storing secrets, and wrote the scripts and runbooks that the team uses to manage them. For new MongoDB databases and Redis clusters, I make sure their network allow-lists accept connections only from the platform. I also triage our Dependabot alerts into work that the team can plan.

Our CISO asked me to research and recommend a third-party penetration tester for the platform. I spoke to several providers, and judged them on their reputation and on what those conversations told me about their abilities. My recommendations were accepted, and the test is pending.

When an independent security reviewer needed access to our AWS account, I defined their IAM user in code, with an expiry date after which their access was revoked automatically.

AWS Backup, CloudWatch, Dependabot, disaster recovery, IAM, MongoDB, penetration testing, Redis, S3, Secrets Manager, SOC 2, Sprinto, VPC

CI/CD. I built several composite actions for GitHub Actions that our workflows share, including one that authenticates Poetry to CodeArtifact so that projects with private dependencies can be linted and tested in CI, another that builds Docker images and pushes them to ECR, and one that starts our deployment pipeline for the change that just passed testing. Over 70 of our repositories use these actions.

I proposed and defined the IAM roles that our GitHub Actions workflows assume through GitHub's OIDC provider, replacing the long-lived access keys that the platform's prototype used. Each kind of job has a specific role with the least-privilege IAM policy it needs.

When GitHub changed the format of its OIDC subject claims, workflows in newly created repositories couldn't assume those roles, so they couldn't lint, test, or publish. I traced the failure to the subject pattern in the roles' trust policies, which didn't match the new claims carrying an immutable ID appended to our organisation's name. I updated the trust policies to explicitly include our immutable ID, rather than widen the pattern's wildcard, which unblocked development on the affected repositories.

CodeArtifact, Docker, ECR, GitHub Actions, IAM, OIDC, Poetry

Authentication. I wrote the proposal for our single sign-on service, then designed and built it. This allows our customers to log in just once to access every application, and saves our engineers the effort of rolling their own authentication every time.

The service follows the backend-for-frontend pattern: refresh tokens are held server-side in Redis, with Lua scripting for atomicity, and never reach the browser. Users log in through OAuth 2.0 and OpenID Connect, with PKCE, against Cognito.

I also wrote our hardening pattern for services, which the single sign-on service adopted: containers run as non-root with a read-only root filesystem and all Linux capabilities dropped, sit behind a WAF with rate limits, and refuse to start if misconfigured.

I designed the service's logging so that no code path can emit a token, cookie, or session ID, and validated redirect origins against a tested list of bypass techniques.

I wrote the scripts and documentation that give engineers local certificates and domain names for testing their applications against the live authentication service.

OAuth 2.0, OIDC, Cognito, Docker, Linux, Redis, SSO, WAF

Release engineering. I built our deployment pipeline with CDK Pipelines on CodePipeline, which deploys every change to a development environment, then promotes it through our seven UAT and production environments after manual approval. The pipeline is self-mutating, so adding a new environment is a configuration change.

I designed each release as a fixed set of immutable images, so production receives only what we tested in UAT, even if commits land mid-promotion. Each release resolves its images just once during synthesis, from Systems Manager Parameter Store where my GitHub Actions record the digest of each service's latest deployable image.

Our AI workers are released with semantic versions. I wrote the proposal for how to deliver that, then updated the pipeline to support side-by-side versions, and wrote the recipe that my colleagues followed to update the workers.

I wrote a runbook for promoting changes directly into higher environments during emergencies, such as when a breaking edge case is discovered at a sensitive time. The fix still enters the pipeline as a new immutable image, and the runbook guides engineers through manually deploying that image where it's needed. My colleagues have followed it to resolve several critical issues, and it's worked every time.

CDK Pipelines, CodePipeline, immutable releases, runbooks, semantic versioning, Systems Manager Parameter Store

Developer experience. Developers building AI workers used to be limited to unit testing their changes before merging them, because running a worker with real datasets, models, or workflows required it to be deployed, which could take up to an hour. I proposed and built infrastructure and a Bash script that starts a local worker then registers it in a cloud environment, so it can be tested with real workflows, datasets, and models. Developers can now test a change in seconds.

I started and host workshops that teach machine learning and forward-deployed engineers how to build and test workers, and running workers locally is now routine.

Our private Python packages used to be installed from GitHub with deploy keys, which discouraged engineers from packaging shared code. I proposed and led our move to a private package index in CodeArtifact, which I defined in infrastructure as code, and wrote the composite action that publishes to it. The team migrated every package, then I retired the deploy keys. We now publish more packages and duplicate less code.

My colleagues start new Python packages from my template repository: a stub package with Poetry and the GitHub Actions workflows that lint, test, and release it.

My Docker build-and-push action also creates its target ECR repository, so a new GitHub repository can publish its first image without an infrastructure deployment.

Bash, CodeArtifact, Docker, ECR, GitHub Actions, IaC, Poetry, Python, workshops

Reliability engineering. I built the end-to-end test suite that's the team's primary signal that an environment is healthy. The tests use only our public APIs and Python SDK, and run real AI workflows as a human would. The suite runs on a schedule against all our environments and notifies the team in Slack when a run fails.

AI workflow output can be non-deterministic, so I wrote a testing framework for asserting that results have the expected shape and values within a tolerance.

I built integration tests for several backend services, which start transient Redis and MongoDB databases and run on every pull request in GitHub Actions.

When the end-to-end tests started failing intermittently, I traced the cause to parallel processes racing on the SDK's token cache, and fixed the SDK to write its cache atomically, which stopped the failures.

After a release doubled the number of AI workers in our development environment, the registry that tracks them intermittently took seconds rather than milliseconds to respond. I suspected the registry's health checks on its workers were blocking its API. I'd developed those checks behind a feature flag, so I disabled them with an environment variable and the slowness stopped. I also realised the checks were unnecessary, so I made the case to remove them. When the problem appeared in UAT, the team agreed, and removing the checks resolved it.

end-to-end testing, GitHub Actions, integration testing, MongoDB, Python, Redis, root-cause analysis

Cost and scaling. Our AWS bill fell by about a quarter in two months, mainly through two changes I made.

Every AI worker takes jobs from an SQS queue, but each also had a load balancer left over from the prototype, where it exposed the worker's debug page. I analysed what still depended on them, proposed removing them, then did so for 200+ workers across every environment with no downtime. Load-balancing spend fell 55% and stayed down.

I designed how our workers scale, and built the autoscaling in our registry rather than with Application Auto Scaling, because the registry already knew every worker's queue and versions. A worker whose queue sits idle scales to zero by default, and when a workflow starts, every worker in its graph scales out, so users usually don't wait. Workers can be held warm for days through our private API, so before a demo the team runs a script instead of disabling scaling by hand and re-enabling it afterwards.

Even with scaling enabled only outside production, container spend fell 20% over those two months.

To enable scaling in production, where several versions of each worker run, I designed and built per-version scaling: superseded versions scale to zero. Production has had no scaling-related issues since.

I applied cost allocation tags to track spend per environment, and bounded storage growth with S3 lifecycle rules and Redis eviction.

AWS, scaling, cost optimisation, ECS Fargate, load balancing, Redis, S3, SQS

July 2019 - January 2025

Contract Software Engineer

Self-employed, Remote, United Kingdom

May 2005 - July 2019

Senior Software Engineer

Thomson Reuters Tax & Accounting, Exmouth, Devon, United Kingdom

---

EDUCATION

Month YYYY - Month YYYY

Qualification

Institution

CERTIFICATIONS

Certification, Issuer, Month YYYY