TECHNICAL PORTFOLIO

Sharif Sule Ndlovu

Software Developer · DevOps Engineer · DevBridge

Proof-of-work documentation across three production systems.

Government & public-sector clients · Pretoria, South Africa

shariffndlovu@gmail.com · knowledgebase.devbridge.co.za

CASE STUDY 01

The Workplace Basics

Online Educational & Compliance Training Platform

PROJECT OVERVIEW

The Workplace Basics was a Django-based e-commerce and content delivery platform commissioned by a private client, designed to sell access to workplace compliance training modules — covering public accountability, policy, finance, and employment topics. The platform featured recurring subscription billing via Paystack, a session-based shopping cart, Google OAuth2 login, content gating by subscription status, and async email and task processing via Celery and Redis.

The project was ultimately cancelled before going live — the client pivoted to a book and blog model. However, Sharif used the engagement as an opportunity to go significantly deeper than the requirements demanded, implementing and understanding concepts that became the technical foundation for every government system that followed. The time and effort invested beyond the brief — particularly in raw WebSockets, async Django, and low-level payment integration — proved to be the most valuable professional development of the early career.

Django 5 **Celery** **Redis** **PostgreSQL** **Paystack** **AWS S3** **Docker Compose** **Google OAuth2** **Django Channels**

KEY CHALLENGES & RESOLUTIONS

Paystack Webhook Integration — Payment Lifecycle from First Principles

Coming from a .NET background where payment libraries abstract away the flow, this project required building a ground-level understanding of how webhooks work: the payment provider calls your server, not the other way around. Sharif implemented and reasoned through every event in the subscription lifecycle — charge.success, invoice.payment_failed, subscription.disable, and subscription.expiring_cards — and built the correct state transitions (active → attention → non-renewing → complete) for each. Card expiry notifications required sending proactive emails before the next billing date. Failed payment handling required distinguishing a soft failure (card declined, retry possible) from a hard cancellation. Every inbound webhook was validated against a Paystack signature to prevent spoofed events. This was the most challenging aspect of the project and the most instructive.

Docker Container Networking — Inter-Service Communication and Host Modes

One of the first runtime failures was services that could not reach each other inside Docker Compose. The root issue was misunderstanding Docker's network model: containers on the same Compose network reach each other by service name — not localhost. A Django app

connecting to localhost:5432 would fail because PostgreSQL runs in a separate container. Sharif learned the difference between bridge mode (isolated network, DNS-resolved service names) and host mode (container shares the host's network stack), a distinction that became foundational for all subsequent containerised projects.

Django Migration Surgery — Changing Integer PKs to UUIDs Mid-Project

After the initial data model was live, a requirement emerged to switch primary keys from Django's default auto-increment integers to UUIDs for privacy and API safety — sequential integer IDs expose record counts and allow enumeration. This was the first encounter with raw SQL migrations. Changing a primary key referenced by foreign keys across multiple tables requires a precise sequence: add the UUID column, backfill existing rows, drop the old PK constraint, promote the UUID column, and update all foreign key references. Having not written SQL since high school, this required learning PostgreSQL constraint syntax, transaction safety, and RunSQL operations inside Django migration files. The lesson: the ORM abstracts SQL, but production schema changes often require dropping down and understanding exactly what the database is doing.

Django Channels, Redis, and Async — Closing the Gap Through Structured Learning

Implementing realtime notification features required Django Channels, which introduces ASGI and Python's async/await model alongside the standard synchronous Django request cycle. The challenge was conceptual: understanding when code runs synchronously versus asynchronously, and why mixing the two incorrectly causes deadlocks or dropped connections. Redis served two distinct roles simultaneously — as the Channels layer for message passing between WebSocket consumers, and as the cache backend — which required careful configuration to keep the two uses isolated. The gap was closed through systematic study of Django by Example (editions 4 and 5), which provided the mental model for async Django, Channels consumers, and Redis integration. The time spent on raw WebSockets in particular built an intuition for concurrency that proved directly valuable on the OTP project.

WSGI vs ASGI and First AWS Infrastructure — S3, Web Servers, and Cloud Foundations

This project was the first real engagement with AWS infrastructure: static and media files were moved off the local filesystem into S3, requiring an understanding of IAM roles, bucket policies, presigned URL generation, and how Django's storage backends wire into boto3. Alongside this, the difference between WSGI (synchronous, one request per worker) and ASGI (asynchronous, supports long-lived connections) became clear — as did why Django's development runserver is not suitable for production and what Gunicorn provides. These two foundations — cloud storage and production server selection — became core competencies carried forward into e-PGLUM and OTP.

IMPACT / OUTCOME

The platform was not deployed to production. The client cancelled the project and pivoted to a publishing model. The professional outcome, however, was significant: every major concept underpinning the later government systems — async task queues, containerisation, cloud storage, payment webhooks, OAuth, raw database migrations, and async Django — was first encountered, broken, and understood here. This was the project that turned framework familiarity into genuine engineering depth.

CASE STUDY 02

e-PGLUM

Electronic Provincial Government Land Use Management Platform

---

PROJECT OVERVIEW

e-PGLUM is a nationwide electronic land use management platform serving South African municipalities, live in production at epglum.co.za. The platform digitises a previously manual, paper-based submission and assessment process for land use applications, building plans, outdoor advertising, objections, and appeals. It is deployed across three environments (QA, UAT, Production) on AWS ECS Fargate, backed by RDS PostgreSQL with PostGIS for geospatial queries, ElastiCache Redis, and S3 for static and document storage.

Sharif was the AWS infrastructure owner for the full project, responsible for designing and managing the cloud environment across all three deployment tiers, while also contributing to the Django backend alongside a team of three developers, one frontend developer, testers, and a project manager. This was the largest and most technically demanding project undertaken to that point.

Django 5 **AWS ECS Fargate** **RDS PostgreSQL** **PostGIS** **ElastiCache Redis** **CloudFormation** **AWS WAF** **S3** **EFS** **Celery** **WeasyPrint** **Matplotlib**

KEY CHALLENGES & RESOLUTIONS

96 Government Submission Forms — Python Mixins as a Survival Strategy

The platform required 96 distinct form classes (verified from codebase) covering land use applications, building plans, outdoor advertising, objections, appeals, rezoning, township establishment, subdivision, consolidation, and collaboration workflows. Each form shared significant overlap: applicant identity fields, property location data, municipality references, document attachment handlers, and validation rules that varied only slightly by submission type. Managing this without duplication required a deep understanding of Python's class system — specifically mixins as a tool for composing shared behaviour across unrelated class hierarchies. A new form type could be assembled from existing mixins in minutes rather than hours of copy-paste. The same pattern extended to Django models and class-based views.

Report Generation — Building Visual Analytics at Scale

The platform required visual reports: graphs, summary tables, and statistics scoped to selected models — municipality, submission type, date range, and status. This was the first production use of Python data visualisation packages: Matplotlib for charts, openpyxl and xlsxwriter for Excel exports, and WeasyPrint for PDF rendering. The challenge was making

output dynamic and efficient — a report filtered by one municipality must produce different charts than one filtered by a province, pulling from the same queryset. Initially, report generation ran synchronously inside the HTTP request cycle. As the platform accumulated real data, reports began timing out. The generation pipeline was migrated to Celery workers: the view triggers a task, returns immediately with a job ID, and the frontend polls when the report is ready. This reinforced the architectural principle that anything taking longer than a few seconds does not belong in the request/response cycle.

WAF / Web ACL Production Blocking — The Cost of Environment Mismatch

When e-PGLUM went live, AWS WAF managed rules blocked legitimate requests that had passed without issue in lower environments. Large form submissions triggered SizeRestrictions_BODY rules; certain field content patterns triggered XSS rules. These had never surfaced in dev or staging because WAF was configured leniently there. The investigation required reading CloudFront and WAF logs, identifying which managed rule group was matching which request, and writing byte match exclusion rules scoped to specific URI paths. Application code also required tightening in places where payloads were larger or structured in ways the WAF interpreted as suspicious. The root lesson: production security controls must be present in all environments from the start. This mistake directly shaped how OTP was built.

Large File Uploads — Tuning Timeouts Across the Full Request Path

Government document uploads — building plans, title deeds, site development plans — regularly exceeded 20MB. At this size, default timeout configurations across the stack became a problem: CloudFront has a default origin response timeout, ALB has its own idle timeout, and Gunicorn has request timeout settings. Files exceeding any threshold failed silently. Resolving this required understanding the full request path — browser → CloudFront → ALB → ECS task — and tuning timeouts at each layer consistently. It also introduced the question of presigned POST URLs for direct-to-S3 uploads, bypassing the application server entirely, which became a pattern evaluated for future projects.

First Serious AWS Infrastructure — Full Environment Ownership from Scratch

e-PGLUM was the first project where a complete production AWS environment was designed and owned end-to-end: VPCs with public and private subnets, security groups with least-privilege rules, RDS PostgreSQL in a private subnet, ElastiCache Redis, ECS Fargate services for web and Celery workers, ALB with health checks, S3 for static and media, Route 53 for domain management, and ACM certificates. Every component had to be wired together correctly — a misconfigured security group meant the app could not reach the database; a wrong subnet placement meant ECS tasks had no internet access for pulling images. This project built the mental model of AWS networking that all subsequent infrastructure work has been built on. CloudFormation was adopted as infrastructure-as-code to manage the three-environment setup repeatably.

Nightly Statistics Pipeline — Post-Deployment Integration with the Middleman Server

Approximately two months after go-live, a new requirement emerged: e-PGLUM needed to publish nightly statistics to the same RabbitMQ middleman server used by the OTP mobile integration. A scheduled Celery Beat task was built to aggregate platform activity nightly — application counts by submission type, new user registrations, active versus static users, stalled applications, daily and weekly totals, and additional operational metrics — and publish a structured payload to the broker. The challenge was building queryset aggregations across a live production database efficiently: at scale, poorly written aggregation queries lock rows or cause slow table scans that affect daytime performance. The task had to run within a narrow nightly window and produce output matching the broker's expected schema exactly.

Copilot CLI Deprecation — Migrating Live Infrastructure to Standalone CloudFormation

During active development, AWS announced that Copilot CLI would reach end-of-life in June 2026. Both e-PGLUM and OTP were built on Copilot. Rather than wait, the infrastructure for both systems was migrated to standalone CloudFormation templates managed via DRY bash scripts. The migration required reverse-engineering the stacks Copilot had generated, extracting reusable patterns, and rebuilding them as maintainable templates with parameter files per environment. The result is a deployment system with discrete scripts for each concern: push-env.sh, push-image.sh, deploy-env-addons.sh, deploy-all.sh, and release.sh. This work was completed across two codebases simultaneously while both systems remained in active development.

IMPACT / OUTCOME

e-PGLUM is live in production at epglum.co.za, serving South African municipalities. It replaced a manual, paper-based land use application process with a digital submission and assessment workflow. Sharif established himself as the infrastructure owner on government-scale Django deployments — responsible for the full AWS environment across three deployment tiers, while contributing meaningfully to a 96-form Django codebase. The WAF and Copilot lessons from this project directly improved how the next system, OTP, was designed and deployed.

CASE STUDY 03

OTP — Livi Lemphakatsi

Multi-Tier Government Case Management System

PROJECT OVERVIEW

OTP (branded Livi / Livi Lemphakatsi) is a multi-tier government case management system serving the Office of the Premier, Mpumalanga, across 10 provincial departments. Citizens submit cases — service delivery complaints, missing persons, vacancies, crime hotspots, community projects — via a mobile application. The Django portal handles intake, routing, assessment, inter-departmental collaboration, and resolution, with case workers operating under SLA enforcement. The system is live in production at livilemphakatsi.co.za.

Architecture is queue-driven: the mobile app and the OTP portal communicate through a RabbitMQ middleman broker rather than directly, providing loose coupling and a clear contract boundary. Each of the 10 departments operates on its own branded subdomain with its own colour scheme, making one application feel like 10 distinct systems. POPIA compliance is enforced by deploying production in the af-south-1 (Cape Town) AWS region.

Django 5.2 **AWS ECS Fargate** **RDS PostgreSQL + PostGIS** **RabbitMQ (Amazon MQ)** **Redis** **Celery** **SSE + ASGI** **CloudFormation** **AWS WAF** **AWS Rekognition** **LangChain + pgvector** **django-tenants**

KEY CHALLENGES & RESOLUTIONS

Architecture Evaluation — Choosing Simplicity Over Workflow Engines

Before a line of application code was written, the team evaluated a range of tools: Camunda (BPMN workflow engine), Salesforce-adjacent tooling, django-viewflow (BPMN handling inside Django), n8n (low-code automation), and multi-tenant frameworks including django-tenants. After formal evaluation, all external workflow engines were scrapped. The complexity they introduced outweighed their value for a system where the "workflow" is simply a case moving through a sequence of statuses. The status-as-state-machine pattern from e-PGLUM was adopted and extended — NEW → IN_PROGRESS → RESOLVED / DUPLICATE / CANCELLED — with SLA enforcement and escalation layered on top via Celery Beat. The lesson: evaluate tools properly before building on them, and default to the simplest model that satisfies the actual requirements.

10 Departments, One Application, 10 Distinct Identities

OTP serves 10 government departments: Agriculture, Safety & Security Liaison, Economic Development & Tourism, Co-operative Governance & Human Settlements, Culture Sport & Recreation, Health, Office of the Premier, Public Works, Social Development, and Social Services. Each operates on its own subdomain and has its own branded theme — primary

colour, secondary colour, accent, and custom CSS — applied dynamically via DepartmentThemeMiddleware reading the Host header on every request. Architecturally it is one Django application and one database; visually it must feel like 10 different systems. Every view, template, and email had to be theme-aware.

Per-Department Domain Management — 30 Subdomains, No Wildcards Where It Counted

Each of the 10 departments requires three subdomains — one per environment (health-qa, health-uat, health) — totalling 30 subdomains. A wildcard certificate handles TLS, but subdomains could not be wildcarded in all contexts: WAF rules, CSRF trusted origins, and session cookie domains all required explicit entries because the DepartmentThemeMiddleware resolves department identity from the exact subdomain string. Any mismatch between a registered subdomain and the middleware's expectation silently fails to resolve a department. Each subdomain had to be set precisely in the database and kept in sync with DNS across all environments.

Queue Architecture — From AWS SQS to RabbitMQ on the Middleman Server

The queue integration went through two phases. Initially, AWS SQS was implemented as the messaging layer. As the mobile integration matured, it became clear that the middleman ran RabbitMQ and expected both OTP and the mobile app to publish and consume from queues on that broker. OTP publishes outbound payloads — case updates, status changes, responses — to queues on the middleman, and consumes inbound queues (q_livi_case_activities, q_livi_case_chat, q_livi_missing_persons, q_livi_vacancies, and others) to pull data submitted by the mobile app. The migration from SQS to RabbitMQ required understanding the protocol differences — SQS is HTTP-polled; RabbitMQ uses AMQP with long-lived TCP connections via pika — and re-implementing consumer and publisher logic to match the middleman's payload schema exactly. Every message is recorded in RxQueueLog and TxQueueLog audit tables for traceability and replay.

WAF Parity — Production-Level Firewall in QA from Day One

The most costly infrastructure mistake on e-PGLUM was running light WAF rules in lower environments, then discovering production blockages only after go-live. On OTP, the QA environment was configured with the same AWS Web ACL managed rule groups as production from the start. When team members used QA during testing, any request that would have been blocked in production was blocked immediately. Blocking issues were identified and resolved during internal testing — not by end users after deployment. Turnaround time dropped from days (production incident under pressure) to hours (caught in QA while the team was present).

Realtime Chat — SSE, Redis Pub/Sub, and the ASGI Transition

Case workers and citizens communicate through a threaded chat on each case. Implementing live message delivery required Server-Sent Events over async Django views

(Daphne/ASGI), with Redis pub/sub decoupling message persistence from streaming: a message is saved to the database, a signal publishes it to a Redis channel, and the SSE view streams it to the browser. A Last-Event-ID recovery scheme using an ISO8601#UUID format handles reconnects without duplicate delivery. The feature was deployed behind a CASE_CHAT_REALTIME_ENABLED flag, allowing it to go live and be toggled without a redeployment. The ASGI and async knowledge built on Workplace Basics was put to direct production use here.

Health Checks — 5 Checks Per Group as a Diagnostic Tool

OTP's health check configuration runs 5 checks per group: database connectivity, Redis cache, S3 storage, migrations applied, and Celery worker availability. Getting all 5 to pass consistently required careful ECS task startup ordering — the task must have network access to RDS, Redis, and S3 before the health endpoint responds 200. A misconfigured security group or missing IAM permission on any one service causes the task to fail its health check, be marked unhealthy, and be replaced in a continuous loop. This made health checks a useful diagnostic tool: a cycling ECS task is a signal that one specific dependency is unreachable, and the endpoint identifies which one.

IMPACT / OUTCOME

OTP is live in production serving the Office of the Premier, Mpumalanga, across 10 government departments. The decision to scrap workflow engines in favour of a status-driven case model resulted in a system case workers can operate without specialised training. The infrastructure migration from Copilot to standalone CloudFormation, completed during active development, leaves both OTP and e-PGLUM on a maintainable foundation beyond the Copilot EOL date. The WAF parity practice established on this project is now standard across all DevBridge deployments.

PROGRAM

Community Web Modernization

A DevBridge Social Initiative — Modern Websites for South African Communities

---

WHAT IT IS

The Community Web Modernization Program is a DevBridge initiative that provides modern, secure, and sustainable static websites to community organisations across South Africa — at low or no cost for qualifying recipients. The program exists because many community institutions (estates, schools, churches, NGOs, sports clubs, local initiatives) have either no web presence or maintain outdated sites that are slow, insecure, and no longer representative of the communities they serve.

HOW IT WORKS

Every site is built as a modern static website — no server, no database, no CMS to patch. Sites are deployed to Cloudflare's edge network with automatic HTTPS, DDoS protection, and sub-second global load times. The architecture is intentional: by eliminating the traditional server stack, operating costs drop to near zero and the site remains secure and functional indefinitely with minimal maintenance. Organisations are given full ownership of the website files and can choose between managed content updates (send changes via email) or a self-service CMS integration for frequent editors. Delivery follows a four-step process: initial conversation → design → build and deploy → handover with full file ownership.

LIVE CLIENTS

The following organisations are live through the programme: Reliance Pro, Makgotoso Motloung, Rehoboth Boerdery, Sknn, Dream Engineering, and I-Ntuthuko Enterprise. Clients span residential estates, agricultural businesses, engineering firms, and community enterprises across South Africa.

Static HTML/CSS/JS **Cloudflare CDN** **Cloudflare DNS** **HTTPS by default** **Git deploy pipeline** **Barlow Condensed + Jost** **GSAP** **Lenis Scroll**

WHY IT MATTERS FOR THE PORTFOLIO

This program sits outside the government contract work and demonstrates a different dimension of the work: product thinking (designing a repeatable delivery model), client acquisition and management across a diverse range of organisations, frontend design and build skills, and a deliberate choice to apply production standards — the same infrastructure discipline used on e-PGLUM and OTP — to community-scale problems. It is an ongoing initiative, not a completed project.

CERTIFICATIONS

Professional Credentials

Certifications were pursued in parallel with live project work — not as prerequisites but as structured consolidation of concepts being applied in production. Each certification maps directly to a phase of the project timeline.

AWS Certified Cloud Practitioner (CCP)

Earned during The Workplace Basics project. Formalised the foundational AWS knowledge being built through the first S3 and IAM implementations.

AWS Certified Solutions Architect – Associate (SAA)

Earned during The Workplace Basics project. Provided the architectural vocabulary for VPC design, multi-AZ deployments, and service selection — directly applied on e-PGLUM.

AWS Certified Developer – Associate

Earned during The Workplace Basics project. Deepened understanding of ECS, Lambda, SQS, and deployment patterns used throughout the government systems.

CompTIA Network+

Earned just before beginning e-PGLUM. Provided the networking fundamentals — subnets, routing, DNS, TLS — that underpinned the full AWS VPC and domain work on e-PGLUM.

CompTIA Project+

Earned just before beginning e-PGLUM. Provided project management fundamentals relevant to operating as infrastructure owner in a structured 7-person team.

Microsoft Azure Fundamentals (AZ-900)

Passed and credential linked via Microsoft Learn. Earned during the OTP development period as part of broadening cloud platform knowledge beyond AWS.

Microsoft Azure Developer Associate (AZ-204)

In progress — exam scheduled approximately week of 2026-06-30.