Adesola Gabriel Adeola

Security Engineer · Columbus, Ohio

aadeola20@outlook.com · linkedin.com/in/adesola-adeola-5255841b7 · github.com/Godson90

This document is the offline edition of my portfolio. It mirrors the website at gabrieladeola.dev: an about page, a how-I-work statement, and write-ups of every project listed under selected work.

————————————————————————————

About

Adesola Gabriel Adeola

Security engineer. Columbus, Ohio.

Five-plus years of hands-on work in enterprise security across cloud, on-prem, and corporate IT. The day job is threat hunting, incident response, IAM and PAM administration, third-party risk reviews, and audit-evidence collection. Around that, building internal apps and workflow scripts that close the gap between alert and action. None of it is a demo. It runs.

Path here

Bachelor of Technology in Chemical Engineering from Ladoke Akintola University of Technology in Nigeria, then IT technical support at Globacom Telecommunication in Lagos. Moved to Ohio, worked five years as a Direct Support Professional while studying cybersecurity. Associate's at Columbus State Community College, B.S. Cybersecurity and Information Assurance from Western Governors University. The non-linear part is the most honest part.

Currently

Security Engineer at STRS Ohio since December 2021. Shipping COMPASS, a bi-weekly CIS assessment agent that turns thousands of CrowdStrike Falcon findings into one operator's minutes of review.

Cloud security

A large share of the security work I do has a cloud edge. On Azure, that means securing App Service web apps end to end: managed identities instead of stored credentials, Key Vault references for secrets, private endpoints and access restrictions on the network side, App Service authentication wired in front, RBAC scoped to least privilege, and Defender for App Service for runtime signal. On AWS, that means IAM that does not over-grant, S3 hardening

(public-access block, default encryption, access points where they help), KMS-managed keys for data at rest, Security Hub and GuardDuty for cross-account visibility, and CloudTrail piped into the SIEM so audits do not have to be reconstructed after the fact. Detection telemetry from both clouds feeds the same threat-hunting workflow.

Agentic AI and LLM security

Increasingly the tooling is agentic. I build with Claude Code, OpenAI Codex, and Microsoft Copilot Studio on top of LangGraph and CrewAI for orchestration, RAG over pgvector for grounding, and a verifier pass for anything an operator will act on. The security side is not optional: every LLM surface gets OWASP LLM Top Ten guardrails before it ships, including prompt-injection prevention, data-leakage controls, sensitive-information disclosure checks, and secure-output enforcement. Agents propose, operators confirm. The model never closes the loop alone.

Stack

- Languages: Python, PowerShell, Go, TypeScript, SQL
- Frameworks: FastAPI, Next.js, LangGraph, CrewAI, Flask
- Storage and infra: Postgres + pgvector, Docker, Celery + Redis, Alembic
- Agentic AI / LLM dev: Claude Code, OpenAI Codex, Microsoft Copilot Studio, LangGraph, CrewAI, OpenAI SDK, RAG with pgvector
- AI security: OWASP LLM Top Ten, prompt-injection prevention, data-leakage controls, secure-output enforcement, verifier-gated narratives
- Security: EDR, DLP, SIEM, CyberArk PAM, MITRE ATT&CK, OWASP Top 10
- Cloud: AWS (IAM, S3, KMS, CloudTrail, GuardDuty, Security Hub, Secrets Manager), Azure (App Service, Key Vault, Managed Identities, Defender, NSGs, Private Endpoints, Entra ID), Microsoft 365
- GRC: ISO 27001, NIST, GDPR, OneTrust
- Certs: SSCP, CEH (EC-Council), CompTIA Network+, Project+, A+, Secure Infrastructure Specialist

How I work

Securing the infrastructure starts where the threat actually lives. Continuous threat hunting across EDR, DLP, identity, network, and cloud telemetry. Detection rules tuned to what the business actually does, not what the catalog says it might. IAM and PAM administration that keeps the access map tight — quarterly access reviews, privileged sessions logged and re-checkable, irregular authentication investigated, not just alerted on. Cloud posture work on Azure App Service and AWS that closes the gap between what the platform allows and what the business should permit.

Proactive work runs alongside the day-to-day, not as an annual event. Breach-and-attack simulations validate that the detections in place still catch what they were designed to catch, and the findings, alongside vulnerability assessments, turn into remediation tickets with owners

and dates, not PDFs that get filed. Third-party risk reviews carry the same weight: every vendor either passes the bar or gets a documented exception with an expiry. ISO 27001, NIST, and GDPR evidence is a byproduct of how the controls run, not a scramble before an audit window.

Security meets the business where it is. Engineering ships faster when they are not blocked by the security team, so guardrails live in CI/CD, OWASP-aligned (including the LLM top ten), and visible before the merge button. Audit and legal get evidence on demand because the controls were instrumented to produce it. Risk decisions are made by the business with the security context they need, not made by the security team alone.

Strategically, the work bends toward fewer hours per investigation, fewer manual touches per audit, and a smaller blast radius per incident. Every automation, every agent, every design doc is graded against those three. I won't skip the process to move faster.

———————————————————————————

Selected Work

COMPASS — Configuration Assessment Agent

#01 · 2026 · Solo engineer

Multi-agent LLM enrichment of CrowdStrike CIS findings against the official benchmark PDF, with human-in-loop approval and automated ServiceNow change tickets.

Problem

An auditor walks in with a CIS benchmark — two hundred and fifty-seven controls across Microsoft Edge and Google Chrome. CrowdStrike Falcon hands the team a pile of failed checks across every endpoint in the fleet. The scanner tells you what failed. It doesn't tell you what to do about it. The benchmark — three hundred and fifty pages of PDF — tells you what good looks like. Nobody reads three hundred and fifty pages of PDF.

That gap is the whole job.

Approach

COMPASS is a bi-weekly assessment agent that closes it. The pipeline has six stages — Falcon, Triage, Research, Narrative, Verifier, Dashboard, ServiceNow — and each one was rebuilt at least once after something broke in production. The dashboard puts a human in the loop: every narrative the model writes gets a single click of approve or disapprove before it leaves the system, and approved findings become ServiceNow change tickets automatically.

Hard problems

Grounding the LLM in CIS PDFs

An LLM that's allowed to invent its own facts is a liability. Auditors won't accept training-data sourcing. My first attempt was a tier-one web allowlist — cisecurity.org, learn.microsoft.com, chromeenterprise.google, NIST — with a Serper fallback restricted to the same domains for niche controls. It worked for the common ones and got patchy for vendor-specific Group Policy settings the open web doesn't index well.

The real fix was the obvious one. The CIS benchmark PDF itself is the source of truth. So I ingested it. CIS Microsoft Edge 4.0.0: 139 controls, 1110 chunks. CIS Google Chrome 3.0.0: 118 controls, 931 chunks. The parser splits each control into its subsections — description, rationale, audit, remediation — embeds them with text-embedding-3-small, stores them in pgvector with an HNSW cosine index.

Retrieval is a hierarchical cascade. For each claim the narrative agent wants to make, the retriever asks three questions in order. First — strong match in this control's own chunks? Cosine above 0.75, stop. Second — strong match anywhere in the same browser's chunks? Above 0.65, stop. Third — anything globally? Take the best we can find. Three nested concentric circles. Start tight. Widen only when the tight one fails.

Making the LLM accountable

Good sources are necessary. Not sufficient. The model still paraphrases wrong, drops hedges, invents confidence. So I added a verifier — a second LLM pass that rates each narrative claim against the same sources the narrative agent saw. Supported, or unsupported. Binary. The verifier originally rated low/medium/high confidence; I cut that because operators couldn't act on the middle.

The verifier also has to emit a verbatim supporting quote from one of the cited sources. The orchestrator validates that quote is a literal substring after case-insensitive whitespace normalization. If the verifier hallucinates the quote to make its job easier, the orchestrator catches it, coerces the verdict to unsupported, and the rewrite loop continues. The model cannot lie its way past the verifier by inventing quotes.

The operator has three distinct primitives: override the flag with a mandatory note, mark verified to vouch for the narrative, or cast a correct/wrong feedback vote that feeds a per-operator aggregate used to drive prompt tuning. Each one is its own audit table. The system is accountable in three directions — against its sources, against the operator who reviews it, and against the operator who reviews the reviewer.

Crash survival and idempotency

Bi-weekly runs touch thousands of LLM calls over hours. Things crash. So the run is resumable. Every Run row carries a heartbeat timestamp updated at every commit boundary. If the heartbeat is older than five minutes, the next CLI invocation declares the run interrupted and

resumes it if it's within the configured window. Phase A (Falcon fetch) re-runs cheaply because Falcon is idempotent. Phase B (triage) re-runs cheaply because triage hits its own LLM cache. Phase C (per-finding enrichment) only processes groups that haven't been persisted.

When the same control comes up two weeks later with the same claims, the prior decision carries forward. The check is exact — claim-set equality on the verification evidence, normalized. If the sets match, the prior decision is inherited and the operator sees an "auto-decided" badge. Operators never re-click Approve on a finding whose narrative didn't change.

The same discipline applies on the ServiceNow side. Every change ticket has a correlation ID; approvers can create a new CHG or add to an existing one. Disapproving the last finding on an open CHG cancels the ticket. The system never duplicates a ticket by accident.

Honest operator UX

This part is internal tooling. It still has to feel good.

Inline row expansion, j/k keyboard navigation, an e-to-approve/d-to-disapprove shortcut on the detail page, a bulk-decide toolbar for batch approvals. Saved views per user. A filter sidebar that carves the index five ways. SLA aging badges computed from severity at the moment of approval. Source-freshness icons next to every citation, re-checked at the end of each run — dead link, auth-wall, unreachable. The CHG number is a deep link straight to ServiceNow.

Honesty here is non-negotiable. Auto-decided findings wear a badge. Every prompt change stamps a new version on the enrichment row, so an old narrative can never falsely claim to come from the current prompt. Manual verification, operator override, and verifier flag are three distinct states that stack newest-on-top without overwriting — the operator can always see what the model said, what the verifier said, and what every prior operator did, each in its own card.

Stack

- Language: Python 3.11+
- Pipeline: bespoke async orchestrator; OpenAI SDK against an OpenAI-wire-compatible gateway, with direct-OpenAI failover for budget exhaustion
- Storage: Postgres 16 with pgvector; SQLAlchemy + Alembic; migrated from SQLite via a custom ID-preserving tool that ships in the image
- Embeddings: text-embedding-3-small, HNSW index
- Dashboard: FastAPI + Jinja, JWT cookies (15-min access, 7-day refresh), three roles, log-out-everywhere via token-version
- Observability: structlog with run/request/user correlation IDs; Prometheus /metrics; OpenTelemetry tracing opt-in via OTELEXPORTEROTLP_ENDPOINT
- Deployment: Docker Compose stack — app + Postgres + cron sidecar; alembic upgrade head on container start; schedule version-controlled in deploy/cron/crontab

- Integrations: CrowdStrike Falcon, ServiceNow Change Management, Slack/Teams webhooks, SMTP notifications

Outcomes

COMPASS runs every two weeks against every endpoint in the fleet, across two browser benchmarks and roughly two hundred and fifty controls. Days of analyst toil per cycle is now a few minutes of operator review.

What I want from this is not the feature list. It's the discipline. Every feature in this system started as a design document that named the problem before it named the solution. Thirty-nine specs in the repo. Each one paired with an implementation plan. Each one paired with the commit that closed it out.

That's how I work.

———————————————————————————————

Risk Register

#02 · 2026 · Solo engineer

Enterprise risk management with LangGraph agents — intake, scoring, monitoring, mitigation tracking.

Problem

Enterprise risk management is dominated by spreadsheet sprawl and once-a-year audit theatre. I wanted a system that captures risks as they're surfaced, scores them consistently, and tracks mitigation through to closure — without becoming a CMS.

Approach

The model started as a state machine, not as a screen. A risk is a row with a lifecycle — DRAFT → OPEN → UNDER_REVIEW → MITIGATING → CLOSED, with side-doors for ACCEPTED and reopen — and everything the system does has to be expressible as a transition on that machine. The API enforces the transitions; the agents propose them; the operator confirms them. There is no path that lets an agent write a closed risk back to open without going through the same audit trail a human would.

On top of that state machine sit four domain agents — Intake, Assessment, Monitoring, Mitigation — each implemented as a LangGraph StateGraph with named nodes for the discrete moves it knows how to make. A master Orchestrator routes intents into one of the four; an event bus moves signals between them so scoring changes can fan out to monitoring without modules reaching across the dependency graph. Celery Beat is the heartbeat: cadence checks every six hours, KRI polling hourly, integration polling every fifteen minutes, milestone slip detection daily, dashboard refresh every fifteen.

Hard problems

LangGraph as a router, not a chatbot

The temptation with agent frameworks is to let the model improvise — open-ended ReAct loops, tool-using agents that figure it out as they go. That's fine for prototypes and terrible for an audit-bearing system. The Orchestrator in this app is a router: it classifies the inbound intent into one of four domains and hands off to a StateGraph whose nodes are the only moves that exist. The Intake graph has exactly three nodes — SubmissionBot, AutoDetector, DuplicateFinder. The Assessment graph has three — RiskScorer, ControlLinker, TrendTracker. If the model wants to do something outside the graph, it can't. The shape of the workflow is enforced by the framework, not by the prompt.

This decision paid for itself the first time the model tried to "helpfully" close a duplicate during intake. The graph had no edge from DuplicateFinder to status transitions, so the LLM's output was ignored and the duplicate surfaced as a link instead. Behavior I'd otherwise have had to discover in production was structurally impossible.

Duplicate detection that doesn't false-positive

A risk register loses operators the moment it tells them their submission is a duplicate of something it isn't. Fuzzy string matching on titles fails on real intake — two genuinely distinct risks can share half their words, and two phrasings of the same risk can share almost none. So intake embeds the submission with text-embedding-3-small, runs a pgvector cosine search against open risks scoped to the same category, and only flags candidates above a calibrated threshold. Above the threshold the operator sees a side-by-side comparison and decides; below it, the risk goes through as new.

The embedding dimension and similarity threshold are config values, not constants — both were tuned against a hand-labeled set of known duplicates and known near-misses before the feature shipped.

Cross-domain triggers without a god-object

Risk scoring changes have to propagate. A risk dropping from CRITICAL to HIGH should trigger a reassessment of the review cadence; a milestone slipping should re-open the assessment to check whether the residual score still holds. The naive shape is one big service that knows everything; the result is a circular dependency between modules and tests that mock half the codebase.

The fix was an in-process event bus. Each domain emits events — RiskScoreChanged, MilestoneSlipped, ThresholdBreached — and subscribes to the events it cares about. No module imports another module's service to call into it. Tests assert that domain X emits the right event for input Y; integration tests assert that the bus actually wires them up. Cross-domain behavior is composable instead of tangled.

Testing async, Postgres-specific code without standing up Postgres

The whole API is async SQLAlchemy 2.0 against Postgres 16 + pgvector. The schema uses JSONB columns and a Vector(1536) column the embedding service depends on. Standing up a real Postgres for every test run is slow and flaky in CI; mocking the database hides exactly the bugs the tests are supposed to catch.

The test harness uses SQLite in-memory with two compiler hooks: one that compiles JSONB to TEXT for storage and re-hydrates it as a dict on read, and one that compiles Vector(n) to a JSON-encoded array so similarity tests run against a pure-Python cosine. Result: the same models, the same services, the same factories — running in milliseconds against an in-process database. The integration tests that exercise pgvector's HNSW index do run against real Postgres in Docker, but those are a thin layer and the rest of the suite — agents, scoring, state transitions, event-bus wiring — runs without leaving the process.

Stack

- API: FastAPI on Python 3.12, async SQLAlchemy 2.0, Pydantic 2
- Persistence: Postgres 16 + pgvector; Alembic migrations
- Agents: LangGraph StateGraph per domain, OpenAI gpt-4o for reasoning nodes, text-embedding-3-small for the duplicate-finder
- Scheduling: Celery 5 + Redis 7 — worker for one-off agent runs, Beat for cadence/KRI/integration/milestone/dashboard jobs
- Frontend: Next.js 16 + React 19, TanStack Query, React Hook Form + Zod, Tailwind 4, shadcn/ui, Recharts
- Integrations: Qualys, Tenable, Jira, ServiceNow — HMAC-verified inbound webhooks plus periodic polling
- Infrastructure: Docker Compose, structlog with request/agent correlation IDs, optional LangSmith tracing

Outcomes

The app runs end-to-end against a fleet's worth of synthetic risks: intake from an analyst form, intake from a scanner webhook, duplicate detection, scoring, control mapping, review-cadence enforcement, KRI threshold alerts, mitigation milestone tracking, and an executive dashboard that refreshes every fifteen minutes. Status transitions are state-machine-enforced from the API down; agents propose and operators confirm, never the other way around.

What I want from this project is the same thing I want from every project on this site: a system whose hard parts are visible. The state machine is one decision. LangGraph as a router instead of a chatbot is another. The event bus over a god-object is a third. Testing async pgvector code against SQLite is a fourth. Each one started as a design document that named the problem before naming the solution — and each one is the answer to a question I could not have answered by writing the code first.

———————————————————————————————

Mail IOC Scanner

#03 · 2025 · Solo engineer

Microsoft Graph harvesting + VirusTotal / urlscan.io / AbuseIPDB enrichment for shared mailbox triage.

Problem

SOC teams triage shared phishing-report mailboxes by hand. The same indicators get re-checked across three vendor APIs every shift. I wanted a deterministic tool that walks a date-windowed mailbox, extracts indicators from message metadata, and batches them through enrichment APIs — no LLM, no flakiness.

Approach

The tool runs as a CLI against a shared mailbox the SOC monitors for phishing reports. Inputs are simple: a mailbox address, a date window, and a path to write the report to. Outputs are an HTML triage report and a CSV that mirrors it, both produced from the same in-memory artifact.

The pipeline has four stages. First, harvest — Microsoft Graph fetches messages in the window using delegated app permissions scoped to the shared mailbox; pagination, throttling, and the $batch endpoint are handled by a thin Graph client. Second, extract — each message is parsed for indicators: sender address, sender IP from Authentication-Results and Received headers, URLs in the HTML body (defanged forms re-fanged, redirectors unwrapped where the source is known), attachment SHA-256 hashes, and the domains/IPs the URLs resolve to. Third, dedupe — indicators are normalized and de-duplicated across the entire window so each one is enriched once, not once per message that contained it. Fourth, enrich — each indicator class is routed to the right API: file hashes to VirusTotal, URLs to VirusTotal and urlscan.io, IPs to VirusTotal and AbuseIPDB, domains to VirusTotal. Results are merged into a single record per indicator, then re-joined back to the messages that contained them.

No LLM is in the path. No agent decides anything. The deterministic shape is the feature.

Hard problems

Three APIs, three rate-limit models, one report

VirusTotal, urlscan.io, and AbuseIPDB each rate-limit differently — VirusTotal as a per-minute and per-day quota with separate buckets for paid and community keys, urlscan.io as a per-minute submission cap, AbuseIPDB as a per-day check limit and a separate per-day report quota. A naive loop that fires requests as it finds indicators trips one limit, gets throttled, and silently corrupts the report.

The fix is a per-API client wrapper that owns the budget. Each wrapper exposes a submit method that queues the request, respects the cooldown, surfaces 429s as backoff signals rather than failures, and emits a structured log line for every consumed unit. The orchestrator queries the budget before scheduling, prioritizes the indicators most likely to be malicious (matched by

a static heuristic on URL/IP class), and falls back to "unenriched" with a tag rather than failing the run.

IOC extraction from real email is not regex

Phishing emails defang URLs (hxxps://, [.]), wrap them in tracking redirectors, and sometimes hide them in image hyperlinks or in the href of a misleading anchor. Naive regex pulls the visible text and misses the actual destination. The extractor uses a parsed-DOM walker for HTML bodies, normalizes defanged forms before extraction, and follows known-safe redirectors (one HEAD hop, no recursion, time-boxed) to recover the real destination. Indicators recovered this way are tagged with their extraction path so the operator can see what the tool did and did not trust.

Determinism without losing context

Operators rerun the same window after new IOCs surface; the report has to give the same answer or explain why it did not. Every Graph fetch, every API call, and every extraction step writes to a content-addressed cache keyed by the input. A rerun against the same window pulls from cache and is byte-for-byte identical unless the operator passes --refresh-enrichment. New indicators discovered in the window produce a new report; previously-seen indicators surface their cached verdict with a freshness timestamp so nothing in the report looks newer than it is.

Stack

- Microsoft Graph for mailbox access (delegated permissions, shared mailbox scope)
- VirusTotal v3, urlscan.io, AbuseIPDB clients
- httpx with retry + rate-limit handling
- No LLM, no agent — fully deterministic

Outcomes

Walks a 24-hour window over a high-volume phishing-report mailbox in a few minutes, deduplicates several hundred raw indicators to a few dozen unique ones, and produces an HTML report the on-call analyst can sort, filter, and pivot from. The same in-memory artifact backs a CSV for tabulation and a JSON for downstream ingestion into the SIEM enrichment store.

The reason this tool is deterministic and not agentic is the same reason it earns its place in the shift handoff: the operator needs to know that the report tomorrow will agree with the report today on the indicators that did not change. An LLM-mediated tool cannot promise that. This one can.

———————————————————————————————

Event Planner — CrewAI

#04 · 2025 · Solo engineer

Multi-agent event-planning web app on CrewAI, demonstrating agent collaboration patterns outside the COMPASS pipeline.

Problem

Most agent-framework demos are command-line scripts. I wanted a small but real web-served multi-agent app to internalize CrewAI's collaboration patterns and contrast them with the bespoke orchestration in COMPASS.

Approach

The crew has four agents — Venue Coordinator, Logistics Manager, Marketing & Communication, and Vendor Outreach — each defined in YAML with a role, a goal, and a backstory that pins the agent to the variables it's allowed to use. The crew runs with CrewAI's shared memory turned on so an agent's output is available to every agent that runs after it. Input is a single dictionary — topic, description, city, date, participants, budget, venue type — and output is a Markdown event plan rendered to HTML for the web, or compiled to DOCX and PDF for download.

There are two surfaces over the same core. A Rich-prompted CLI was the first interface and is still the simplest way to run a plan locally; a Flask web layer wraps the same initializecrew and runevent_crew functions and adds login, async execution, and a thirty-day event history. Both paths produce the same Markdown artifact from the same agents.

Hard problems

Constraint-anchored prompts to keep agents from improvising

The default failure mode for an agent system is the model "helpfully" inventing details — a venue that doesn't exist, a caterer whose website is fabricated, a contact phone number that resolves nowhere. CrewAI gives the model enough freedom that this happens easily.

The fix lives in the YAML. Every agent's backstory ends with an explicit constraint clause: operate as a CrewAI agent under strict collaboration protocols — all suggestions must be data-driven, verified, and cross-checked with the given variables — do not hallucinate or assume details not in the provided inputs. The Vendor Outreach agent's constraint is stricter still — only contact details verifiable via web search, never fabricated phone numbers or emails. This is not a prompt-engineering flourish; it's the structural difference between a demo that looks impressive and an output an operator can act on. Pairing it with the Serper-backed search tool gives the agent a source of truth it can cite instead of paraphrase.

Long-running agent runs over a request/response transport

A four-agent crew with verbose=True and shared memory takes one to three minutes to produce a plan. Flask's default request lifecycle is not that. The naive shape — submit form, wait synchronously, render — yields proxy timeouts in production and an unresponsive UI in dev.

So the web layer puts the crew run on a background thread the moment the POST lands, returns a task ID immediately, and exposes a /api/status/<taskid> endpoint the frontend polls. The browser-side UI shows a progress indicator until the task finishes; the result, once written, is served by /api/download/<taskid>. The same task ID is the key used by the thirty-day history store, so a plan run today is recoverable by URL tomorrow. There's no Celery and no Redis here — the workload is bounded and personal, so threading plus an in-process task table is exactly enough.

Markdown as the agent contract

The crew emits Markdown. The web UI needs HTML. Some users want a DOCX they can edit; others want a PDF they can email. The temptation is to ask the agents for whichever format the user picked — and immediately discover that agent outputs in HTML drift toward verbose tag soup, and outputs targeted at DOCX or PDF lose structure entirely.

The crew always returns Markdown. Conversion happens after. A small markdowntohtml helper renders the web view; document_generator compiles DOCX and PDF from the same source. Markdown is the only contract the agents need to honor, and rendering is a downstream concern the agents never see. The cost is one extra conversion step per output; the benefit is that agent prompts never have to talk about formatting at all.

One core, two surfaces

The CLI came first, with Rich-prompted input and Markdown panels in the terminal. The web app wraps that core — initializecrew() returns the same Crew object, runeventcrew() runs the same flow, validateevent_details() enforces the same required-field set. The web layer adds session auth, async dispatch, history, and three output formats; it does not re-implement the planning. Adding an interface — a desktop wrapper, a Slack bot, a scheduled job — would mean wrapping the same two functions again.

Stack

- Framework: CrewAI with memory=True and verbose=True; OpenAI as the underlying model (configurable, defaults to gpt-4o-mini)
- Agents: Venue Coordinator, Logistics Manager, Marketing & Communication, Vendor Outreach — declared in config/agents.yaml, instantiated in Python
- Tools: Serper-backed web search wrapper, custom sentiment helper
- Web: Flask with session auth, bcrypt-hashed credentials, threaded task dispatch, polling status endpoint
- CLI: Rich for prompts, panels, and Markdown rendering
- Output: Markdown source; markdowntohtml for the web view; DOCX and PDF via document_generator
- Storage: JSON file store for thirty-day event history with auto-cleanup

Outcomes

The app plans end-to-end against the inputs it was built for — a city, a date, a participant count, a budget, a venue type. The crew returns a Markdown plan in one to three minutes; the operator gets a venue recommendation, a logistics outline, a marketing brief, and a vendor list with contact details the agent claims it verified. The constraint clauses in the YAML are the difference between that output being acted on and being rewritten by hand.

The point of this project on the site is not the event plans. It's the contrast with COMPASS. COMPASS is bespoke orchestration — every transition between stages is code I wrote and own. Event Planner is CrewAI orchestration — every transition is the framework's. Both ship. The choice between them is not about which is better; it's about which one earns the discipline a given problem demands. Building this one is how I know.

———————————————————————————————

CyberArk Python SDK

#05 · 2024 · Solo engineer

REST API wrapper for CyberArk privileged-access workflows, authored from scratch.

Problem

CyberArk's REST API surface is broad, sparsely documented in the open, and not consistently shaped. I wanted a Pythonic wrapper that abstracts authentication, retries, and pagination, and exposes high-level workflows (safe management, account lifecycle, user provisioning) instead of raw HTTP calls.

Approach

The SDK presents a single class, CyberArkAccount, that exposes the operations operators actually perform: retrieve a password, change it, verify it against the target, reconcile it, rotate SSH keys, generate a new password from the platform's complexity policy, and update account properties — address, port, database name, platform ID. Everything underneath that surface — which endpoint to call, which token to mint, which fallback to try when the primary path fails — is the SDK's problem, not the caller's.

The transport layer is three concentric paths. First the Central Credential Provider (CCP) on the main site, an app-context HTTP service that returns a password against an AppID + Safe + UserName query with no user login required. If that fails, the DR-site CCP — same shape, different host, kept on hot standby. If both CCPs are unreachable, the SDK falls back to PVWA REST: an RSA-decrypted bootstrap secret on disk is used to log in directly against /PasswordVault/api/auth/Cyberark/Logon, and the resulting token authenticates the management calls. Most callers never know which path resolved their request. The audit log knows.

Hard problems

Three different APIs hiding behind one product

CyberArk's REST surface is not one tree. CCP is one service, with one auth model (an AppID string passed as a query parameter) and one response shape. PVWA REST is another service, with bearer tokens, different status codes for the same conditions, and pagination that's predictable in the documentation and inconsistent in practice. AIM Web Service is a third, vended as a CCP variant but with its own error envelope.

The SDK's job is to hide that. Every public method on CyberArkAccount returns a single shape, even when its implementation routed through two services to get there. A getpassword call hits CCP first and PVWA second; a changepassword call hits PVWA's accounts API and emits a CCP cache-invalidation hint; an updateplatformid call walks PVWA's search endpoint to resolve the internal numeric ID before it can hit the update endpoint at all. The caller writes the same line either way.

Two-and-a-half-step failover that has to be invisible

Enterprise CyberArk deployments run main and DR sites in active-passive. The naive failover — "try main, on exception try DR" — is also wrong, because some failures (a 429 rate-limit, an expired AppID) should propagate; some (a connection timeout, a 502 from the load balancer) should fall through.

The pvwa() helper in dr_testing.py codifies the rule: CCP-main first, CCP-DR second, PVWA REST third — and only specific exception classes trigger fall-through. The SDK never silently swallows authentication errors, never retries on a 4xx that means the request was wrong, and never falls back if the operator explicitly forced a target. This is the difference between a wrapper that hides failure and one that hides infrastructure.

Mutations require search-then-act

CyberArk's update endpoints take internal account IDs — opaque integers operators never see. Operators know their accounts by username, safe, and address. So every update method in the SDK runs a search against /api/Accounts first, filters the result set down to one match using a custom postFilter class, validates uniqueness, and only then issues the PATCH.

The filter is not an optimization. The accounts API paginates inconsistently and supports server-side filtering only on a subset of fields; client-side filtering by (username, safe, address, database) is the only way to guarantee uniqueness across all account types. The SDK refuses to mutate if zero matches or more than one match come back — better to fail loud than rotate the wrong credential.

Encryption in transit and at rest, even in memory

The corporate constraint was that secrets retrieved from the vault must not exist as plaintext Python strings any longer than the call that consumed them. So every CCP response is Fernet-encrypted the moment it leaves the HTTPS socket — a fresh key is minted per request, the

ciphertext is held in the SDK, the plaintext is decrypted only inside the method that returns it to the caller. The bootstrap PVWA secret on disk is itself RSA-encrypted (rsadecryptfrom_files()), so even a developer with read access to the SDK's working directory cannot recover the master credential without the private key.

This is paranoid. It is also the rule for any tool that touches the privileged-access vault in production. The SDK ships with the discipline pre-applied so callers don't have to remember it.

Stack

- Language: Python 3.9+
- Transport: requests against PVWA REST + AIM Web Service (CCP); urllib3 warnings suppressed because corporate-CA-signed endpoints are validated out-of-band
- Auth: CyberArk Cyberark Logon flow with concurrentSession=True; token reuse within a method, no shared mutable state between calls
- Failover: CCP main → CCP DR → PVWA REST, with exception-class gating
- Crypto: cryptography library (Fernet for in-memory secrets, RSA for the on-disk bootstrap credential)
- Validation: per-argument regex + type checks in input_validate.py and inline at every public method boundary
- Filtering: custom postFilter and databasepostfilter for client-side narrowing of paginated PVWA results
- Audit: every credential retrieval injects a Reason string into the CCP request; every PVWA mutation logs the resolved account ID and the requesting cloud ID

Outcomes

The SDK is the foundation for every in-house CyberArk automation that doesn't go through the GUI — bulk reconciliation runs, scheduled password rotations on managed service accounts, address updates after a server migration, platform-ID corrections after a Safe restructure, SSH key retrieval for jump-host workflows. Operations that used to be hand-run through the PVWA web console one account at a time are now a Python script over a CSV.

What I want from this on the site is the part the demo can't show. A REST wrapper is not interesting; a REST wrapper that survives a DR failover during a Friday rotation while the operator is at lunch is interesting. The SDK is small. The engineering it took isn't.

———————————————————————————————

Azure Web App Secure Design

#06 · 2025 · Solo architect

Defense-in-depth secure architecture and remediation plan for six critical risk domains on Azure App Service.

Problem

A web tier on Azure has six places it can fail at the same time: the edge, the secrets, the uploads, the network capacity, the access map, and the detection seam. Each one has a vendor doc, an ARM template, and a Microsoft Learn article. None of them are a security design. A design is the shape that says where each control belongs, what it protects against, what it leaves uncovered, and how the gaps get closed.

This is that design — captured as an 18-slide deck I prepared and walked stakeholders through end to end.

Approach

The architecture is edge-to-data defense in depth. Identity gates the top, secrets feed the app from the side, and monitoring, DevSecOps, and governance are always-on across the stack. User traffic enters through Azure Front Door (WAF, DDoS, bot protection), hits a regional Application Gateway WAF v2 (OWASP CRS, TLS termination), reaches the App Service web tier (Managed Identity, Private Endpoints, NSGs), and resolves against a data tier of Azure SQL with TDE + Always Encrypted and Defender-watched Storage. Secrets sit out of band in Key Vault and Managed HSM with soft-delete + purge protection and auto-rotation; the app never carries one.

The remediation plan is organized as six risk domains. Each domain names its risks, its Azure controls, and the implementation steps. The deck closes with two deep-dives — programmatic secret retrieval via Managed Identity (no creds in code or config), and a data-at-rest encryption model that pairs service-managed TDE with the compensating controls TDE alone cannot provide — plus a DevSecOps pipeline that wires SAST, SCA, IaC scan, container image scan, and DAST into Azure DevOps as gates instead of suggestions.

The six risk domains

1. Application and network attacks

OWASP Top 10, SQLi, XSS, RCE, Layer-7 floods, malicious bots. Controlled by Azure Front Door or Cloudflare at the edge (WAF, bot protection, HTTPS-only, TLS 1.2+, rate limiting), Application Gateway WAF v2 with the OWASP CRS plus custom rules, Network Security Groups, Azure Firewall for egress, and Private Endpoints with public access disabled. Content Security Policy restricts allowed content sources so uploaded files cannot execute.

2. Exposure of API keys, certificates, connection strings, and passwords

Hardcoded secrets, credential leakage, compromised repos, leaked-key privilege escalation, long-lived static credentials. Controlled by Azure Key Vault and Managed HSM, Managed Identities replacing every embedded credential, Azure RBAC scoped to least privilege, Just-in-Time access to vaults, secret scanning + push protection in CI/CD via GitHub Advanced Security and Defender for DevOps, and automatic rotation policies for certificates and credentials.

The deep-dive: the developer pattern. The app requests a token at the Managed Identity endpoint (no secrets in code), Entra ID issues an OAuth bearer token scoped to

https://vault.azure.net, the app calls Key Vault's REST API with the bearer token, and the vault enforces RBAC and returns the secret with the access logged. Surface forms include the raw REST call, the SecretClient + DefaultAzureCredential SDK, and the zero-code App Service Key Vault reference (@Microsoft.KeyVault(SecretUri=...)).

3. Insecure file uploads

Malware, web shells, ransomware staging, ZIP bombs, MIME and extension spoofing. Controlled by file type, MIME, and signature validation, antivirus and sandbox scanning, Content Disarm and Reconstruction, private Blob containers with no anonymous access, short-lived SAS tokens, immutable storage where it fits, and Microsoft Defender for Storage doing malware scanning at the storage tier. Allow list (PDF, DOCX, XLSX, JPG, PNG); block EXE, JS, BAT, DLL, PS1. Uploads stored outside the web root with executable permissions removed.

4. Denial-of-service and DDoS

Service outages, resource exhaustion, API abuse, Layer-7 floods bypassing L3/L4 defenses, burst traffic against unscaled tiers. Controlled by Azure Front Door for global edge filtering and L3/L4 mitigation, Azure DDoS Protection Standard on VNets and public endpoints, WAF rate limiting per IP and per API (for example 100 req/min per IP with API-specific thresholds), CDN caching for static and public content, App Service autoscaling tied to CPU, memory, and request thresholds, and multi-region failover.

5. Excessive developer and admin privileges

Insider threats, accidental misuse, privilege escalation via standing admin roles, unauthorized production changes. Controlled by Microsoft Entra ID as central identity, Privileged Identity Management with approval workflows and time-bound elevation, Conditional Access policies including MFA, compliant devices, and geographic restrictions, least-privilege RBAC separating developers, admins, security, and DevOps, removal of all standing admin access in favor of JIT + monitored break-glass accounts, and continuous audit of role assignments, failed logins, and privileged operations.

6. Insufficient detection and incident response

Delayed breach detection, undetected lateral movement, no incident visibility across services, compliance and audit failure, slow manual response. Controlled by Microsoft Sentinel as SIEM + SOAR, Defender for Cloud (CSPM and workload), Defender for App Service / Storage / SQL, Azure Monitor + Log Analytics, diagnostic settings on every sensitive service, and Logic Apps automating response playbooks (block IP, disable account, quarantine upload). Detection rules cover impossible travel, failed-login bursts, suspicious API calls, WAF anomalies, and Key Vault access anomalies.

Data-at-rest encryption — design decision

Baseline: service-managed Transparent Data Encryption on Azure SQL — AES-256, on by default, Microsoft-managed protector with automatic rotation, covering data files, transaction log, and backups, inherited by geo-replicas and failover groups, transparent to the application. TDE alone

does not cover authenticated SQL access with compromised credentials, application-tier compromise reading through legitimate connections, in-memory data once loaded into the buffer pool, or column-level confidentiality for PII, PCI, or credential data. Compensating controls: Microsoft Entra authentication for SQL, least-privilege SQL RBAC, Private Endpoints (no public network access), Defender for SQL threat detection, auditing streamed to Log Analytics and Sentinel, Always Encrypted for sensitive columns, Dynamic Data Masking on read paths, and Managed Identity for app-to-SQL access.

DevSecOps gates

Deployments flow through Azure DevOps pipelines with security wired into every stage, SAST shifted left into the PR build and DAST anchoring the staging deploy. Stages: (1) commit — secret scan with push protection; (2) PR build — SAST, SCA, IaC scan with inline PR decoration; (3) build and test — unit tests and container image scan; (4) stage deploy — DAST scan and API security test; (5) prod deploy — runtime protection by Defender for App Service. SAST runs SonarCloud and Microsoft Security DevOps, with branch policy blocking merge on Critical / High findings. DAST runs OWASP ZAP via native Azure DevOps task plus Burp Suite Enterprise for richer coverage, triggered after stage deploy and blocking promotion to production on Critical findings.

Framework alignment

The design satisfies the controls expected by every major web application security and cloud governance framework: OWASP Top 10 (WAF rules, input validation, secure auth, logging), Zero Trust Architecture (verify explicitly, least privilege, assume breach), CIS Azure Foundations Benchmark (identity, network, logging, data baselines), NIST Cybersecurity Framework (Identify, Protect, Detect, Respond, Recover), ISO/IEC 27001 (ISMS controls for access, ops, and incident management), and the Microsoft Cloud Security Benchmark as Microsoft's prescriptive Azure baseline.

Stack

- Edge: Azure Front Door Premium / Cloudflare — WAF, DDoS, bot protection, rate limiting
- Regional WAF: Application Gateway WAF v2 with OWASP CRS
- Web tier: Azure App Service with Managed Identity, Private Endpoints, NSGs
- Data tier: Azure SQL (TDE + Always Encrypted), Azure Storage with Defender for Storage
- Identity: Microsoft Entra ID, Conditional Access, PIM, Managed Identity
- Secrets: Azure Key Vault + Managed HSM, soft-delete and purge protection, auto-rotation
- Monitoring: Microsoft Sentinel, Defender for Cloud, Defender for SQL / Storage / App Service, Log Analytics, Logic Apps for SOAR
- DevSecOps: Azure DevOps, Microsoft Security DevOps, SonarCloud, OWASP ZAP, Burp Suite Enterprise, GitHub Advanced Security
- Governance: Azure Policy, Landing Zones, Microsoft Cloud Security Benchmark, Defender CSPM

Outcomes

What good looks like, in five lines: defense in depth — Front Door, WAF, NSGs, Private Endpoints, Managed Identities, no single failure point. Secrets in Key Vault — Managed Identities replace embedded credentials across the stack. Detection and response — Sentinel + Defender for Cloud with Logic Apps automating containment. Least privilege, JIT — Entra ID, PIM, Conditional Access, MFA, no standing admin. DevSecOps gates — SAST and DAST in every Azure DevOps build, vulnerabilities caught before they ship.

The deck itself is the artifact: a single document a stakeholder can read and a sequence an engineer can implement against, mapped to controls every auditor recognizes. Download the presentation.

———————————————————————————