JobTracker: A Full-Stack Web Application with Production-Grade DevOps Infrastructure on Google Cloud Platform

Adarsh Ajay
adarshajats2003@gmail.com

February 13, 2026

Abstract

This paper presents **JobTracker**, a full-stack web application for managing job applications, accompanied by a comprehensive DevOps infrastructure deployed on Google Cloud Platform (GCP). The application features a secure RESTful API built with Node.js and Express.js, JWT-based authentication, an embedded SQLite database, and a modern frontend using Tailwind CSS. The DevOps pipeline encompasses containerization with Docker, orchestration with Kubernetes (GKE) and Helm, infrastructure-as-code with Terraform, continuous integration/continuous deployment (CI/CD) with Jenkins, declarative deployments via ArgoCD GitOps, and configuration management with Ansible. This work demonstrates the integration of modern web development with production-grade DevOps practices and serves as a reference architecture for deploying Node.js applications at scale on cloud infrastructure.

Keywords: DevOps, Node.js, Kubernetes, Terraform, Jenkins, ArgoCD, GitOps, Google Cloud Platform, CI/CD, Infrastructure as Code

1. Introduction

The modern software development lifecycle demands not only well-architected applications but also robust deployment pipelines that ensure reliability, scalability, and security. While numerous frameworks and tools exist independently, integrating them into a cohesive, production-ready system remains a significant challenge for development teams [1].

This paper addresses this challenge by presenting **JobTracker** — a Job Application Tracker that serves dual purposes: (1) a functional web application solving a real-world problem, and (2) a reference implementation demonstrating the integration of six major DevOps technologies on Google Cloud Platform.

The contributions of this work include:

- A secure, full-stack web application architecture with JWT authentication, input validation, and rate limiting.
- A multi-stage Docker containerization strategy optimized for Node.js production deployments.
- Comprehensive Infrastructure-as-Code (IaC) using Terraform for GCP resource provisioning.
- A complete CI/CD pipeline with Jenkins featuring automated testing, container builds, and staged deployments.
- GitOps-based deployment management using ArgoCD with environment-specific sync policies.
- Server provisioning automation using Ansible roles for Jenkins, Docker, and monitoring stacks.

2. Related Work

DevOps practices have evolved significantly since the term was coined by Patrick Debois

Figure 1: JobTracker application banner showcasing the integrated DevOps technology stack including Docker, Kubernetes, and cloud-native tooling.

in 2009. The foundational principles of continuous integration and continuous delivery were formalized by Humble and Farley [1], establishing the theoretical framework for modern CI/CD pipelines.

Containerization. Docker, introduced in 2013, revolutionized application packaging by providing OS-level virtualization [2]. Multi-stage builds, introduced in Docker 17.05, enabled significant image size reductions while maintaining build flexibility.

Container Orchestration. Kubernetes, open-sourced by Google in 2014, has become the de facto standard for container orchestration [3]. Helm, the package manager for Kubernetes, simplifies deployment management through templated charts.

Infrastructure as Code. Terraform by HashiCorp enables declarative infrastructure provisioning across cloud providers [4]. Its state management and plan-apply workflow provide safety guarantees for infrastructure changes.

GitOps. Weaveworks coined the term GitOps in 2017, proposing Git as the single source of truth for declarative infrastructure [5]. ArgoCD implements this pattern for Kubernetes with automated reconciliation.

Our work distinguishes itself by providing a *complete, integrated implementation* of all these technologies for a single application, rather than treating each in isolation.

3. Application Architecture

3.1. Backend Design

The backend follows a layered architecture pattern built on **Express.js v5.x**, running on **Node.js 20**. The architecture comprises four layers:

1. **Routes Layer** — Defines HTTP endpoints and applies validation middleware.
2. **Middleware Layer** — Implements cross-cutting concerns: JWT authentication, request validation (express-validator), rate limiting, CORS, and security headers (Helmet).
3. **Controller Layer** — Contains business logic for authentication and job CRUD operations.
4. **Data Layer** — Manages SQLite operations via `sql.js`, a pure-JavaScript SQLite implementation requiring zero native compilation.

3.2. Authentication Flow

Authentication uses JSON Web Tokens (JWT) with the following flow:

1. User submits credentials to `POST /api/auth/login`.
2. Server validates credentials against bcrypt-hashed passwords stored in SQLite.
3. Upon success, a signed JWT (HS256) is returned with a configurable expiration (default: 7 days).
4. Subsequent requests include the token in the `Authorization: Bearer <token>` header.
5. The authentication middleware verifies the token signature and extracts the user ID for downstream controllers.

3.3. Database Schema

The application uses two tables:

```sql
CREATE TABLE users (
  id TEXT PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  password TEXT NOT NULL,
  name TEXT NOT NULL,
  created_at DATETIME DEFAULT
    CURRENT_TIMESTAMP
);

CREATE TABLE jobs (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id TEXT NOT NULL,
  company TEXT NOT NULL,
  position TEXT NOT NULL,
  status TEXT DEFAULT 'applied',
  location TEXT,
  salary TEXT,
  url TEXT,
  notes TEXT,
  applied_date DATE,
  created_at DATETIME DEFAULT
    CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT
    CURRENT_TIMESTAMP,
  FOREIGN KEY (user_id)
    REFERENCES users(id)
);
```

Listing 1: Database schema

3.4. API Design

Table 1 summarizes the RESTful API endpoints.

Table 1: API Endpoints

Method	Endpoint	Description
POST	/api/auth/register	Create account
POST	/api/auth/login	Login, receive JWT
POST	/api/auth/logout	Invalidate session
GET	/api/jobs	List applications
POST	/api/jobs	Create application
GET	/api/jobs/:id	Get single app
PUT	/api/jobs/:id	Update application
DELETE	/api/jobs/:id	Delete application
PATCH	/api/jobs/:id/status	Change status
GET	/api/health	Health check

3.5. Security Measures

The application implements defense-in-depth:

- **Input Validation** — All API inputs are validated using express-validator with whitelist sanitization.
- **Password Hashing** — Passwords are hashed using bcrypt with a cost factor of 12.
- **Rate Limiting** — API endpoints are rate-limited to prevent brute-force attacks.
- **Security Headers** — Helmet.js sets HTTP security headers including CSP, HSTS, and X-Frame-Options.
- **CORS** — Cross-Origin Resource Sharing is configured to restrict allowed origins.

3.6. Frontend

The frontend is a single-page application using vanilla HTML, CSS, and JavaScript with **Tailwind CSS** for styling. Key design decisions include:

- No JavaScript framework dependency — reduces bundle size and complexity.
- Tailwind CSS via CDN with custom configuration for brand colors and typography.
- Client-side state management with a simple object store.

- Responsive grid layout for job application cards.
- Modal-based interactions for create/edit/delete operations.

4. DevOps Infrastructure

4.1. Containerization with Docker

The Dockerfile employs a **multi-stage build** pattern to optimize image size and security:

```
# Stage 1: Dependencies
FROM node:20-alpine AS deps
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev

# Stage 2: Runtime
FROM node:20-alpine AS runtime
RUN addgroup -g 1001 -S appgroup && \
    adduser -S appuser -u 1001 -G appgroup
WORKDIR /app
COPY --from=deps /app/node_modules ./
    node_modules
COPY server.js src/ public/ ./
USER appuser
HEALTHCHECK --interval=30s \
  CMD wget --spider http://localhost:3000/api/
      health
CMD ["node", "server.js"]
```

Listing 2: Multi-stage Dockerfile (simplified)

Key security practices include running as a non-root user (`UID 1001`), using Alpine-based images for minimal attack surface, and a `HEALTHCHECK` instruction for container orchestrator integration.

4.2. Infrastructure as Code with Terraform

Terraform provisions the following GCP resources:

Table 2: Terraform-managed GCP Resources

Resource	Configuration
VPC Network	Custom mode, regional routing
Subnet	Primary + secondary ranges for pods/services
Cloud NAT	Outbound internet for private nodes
Firewall	Internal traffic + health check rules
GKE Cluster	Private nodes, Workload Identity, network policy, REGULAR release channel
Node Pool	Autoscaling (1–3), shielded VMs, e2-medium
Artifact Registry	Docker format, 10-version cleanup
IAM	3 service accounts: node, workload identity, CI/CD

The GKE cluster is configured as a **private cluster** with `enable_private_nodes = true`, ensuring worker nodes have no public IP addresses. Outbound connectivity is provided via Cloud NAT, while the API server remains accessible via its public endpoint for operational convenience.

Workload Identity binds Kubernetes service accounts to GCP service accounts, eliminating the need for exported service account keys within pods. This follows Google's recommended security practice for GKE workloads.

4.3. Kubernetes Deployment with Helm

The Helm chart provides templated Kubernetes manifests with environment-specific value overrides:

Table 3: Environment Configuration Comparison

Parameter	Staging	Production
Replicas	1	3+
CPU Request	50m	200m
Memory Request	64Mi	256Mi
Autoscaling	Disabled	3–15 pods
Storage Class	standard	premium-rwo
TLS	No	Yes

The chart includes: Deployment, Service (ClusterIP), Ingress (GCE class), ConfigMap, Secret, HPA (CPU-based), ServiceAccount, and PersistentVolumeClaim for SQLite data persistence.

4.4. CI/CD Pipeline with Jenkins

The Jenkins pipeline implements a **7-stage declarative pipeline**:

1. **Checkout** — Clone repository, compute git short hash for tagging.
2. **Install** — `npm ci` for deterministic dependency resolution.
3. **Lint & Test** — Parallel execution of linting and testing.
4. **Docker Build** — Multi-stage build with metadata labels.
5. **Push to GAR** — Authenticate via service account and push to Artifact Registry.
6. **Deploy Staging** — `helm upgrade -install` to staging namespace with health verification.
7. **Production Approval & Deploy** — Manual approval gate followed by production Helm deployment.

Images are tagged with `${BUILD_NUMBER}-${GIT_COMMIT_SHORT}`, providing traceability from container image back to source code.

4.5. GitOps with ArgoCD

ArgoCD implements the GitOps pattern with environment-specific sync policies:

- **Staging** — Automated sync with `prune: true` and `selfHeal: true`. Any commit to the main branch automatically deploys to staging.
- **Production** — Manual sync requiring explicit approval through the ArgoCD UI or CLI. This provides a safety gate for production changes.

An `AppProject` resource restricts source repositories and destination namespaces, implementing the principle of least privilege for deployment access.

4.6. Configuration Management with Ansible

Ansible automates server provisioning through **four reusable roles**:

1. **Common** — Base packages, deploy user, UFW firewall, sysctl tuning, file descriptor limits.
2. **Docker** — Docker CE installation, daemon configuration (overlay2, log rotation), Google Cloud SDK.
3. **Jenkins** — Java 17, Jenkins server, kubectl, Helm, Docker group membership.
4. **Monitoring** — Prometheus and Grafana deployed as Docker containers with persistent volumes.

Three playbooks compose these roles for specific provisioning tasks: Jenkins server setup, monitoring stack deployment, and direct application deployment (as a Kubernetes bypass for simpler environments).

5. Integration and Workflow

The complete development-to-production workflow integrates all components:

1. **Infrastructure provisioning** — Terraform creates the GCP foundation (VPC, GKE, IAM, Artifact Registry).
2. **Server configuration** — Ansible provisions the Jenkins server with Docker, kubectl, and Helm.
3. **Development** — Developers run the application locally using Docker Compose or `npm run dev`.
4. **Code push** — A `git push` triggers the Jenkins pipeline.
5. **Build and test** — Jenkins runs linting, tests, and builds a Docker image.
6. **Container registry** — The image is pushed to Google Artifact Registry.
7. **Staging deployment** — Jenkins deploys to staging via Helm; ArgoCD monitors and self-heals.
8. **Production promotion** — After manual approval, Jenkins deploys to production; ArgoCD requires manual sync confirmation.

9. **Monitoring** — Prometheus scrapes application health endpoints; Grafana provides dashboards.

6. Discussion

6.1. Design Trade-offs

SQLite vs. managed databases. The choice of SQLite simplifies deployment (no external database service) but limits horizontal scaling since only one pod can write to the PVC simultaneously. For production workloads exceeding single-node capacity, migrating to Cloud SQL (PostgreSQL) with connection pooling is recommended.

Jenkins vs. managed CI/CD. Self-hosted Jenkins on a GCE VM provides maximum flexibility and control but requires operational overhead. Google Cloud Build or GitHub Actions would reduce this burden at the cost of vendor lock-in.

ArgoCD dual strategy. Using automated sync for staging and manual sync for production balances developer velocity with production stability. Teams with mature testing practices may opt for full automation.

6.2. Security Considerations

The architecture implements security at multiple levels:

- **Application level** — JWT auth, bcrypt hashing, input validation, rate limiting.
- **Container level** — Non-root user, Alpine base, no shell in production.
- **Network level** — Private GKE nodes, Cloud NAT, VPC firewall rules.
- **IAM level** — Workload Identity, least-privilege service accounts.
- **Deployment level** — Sealed secrets, manual production approval gates.

6.3. Scalability

The HPA configuration enables automatic horizontal scaling based on CPU utilization (target: 60–70%). The GKE node pool autoscaler adjusts the underlying infrastructure from 1 to 3 nodes per zone. Combined, this provides elasticity for variable workloads.

7. Conclusion and Future Work

This paper presented JobTracker, a full-stack web application with a comprehensive DevOps infrastructure on Google Cloud Platform. The integration of Docker, Helm, Terraform, Jenkins, ArgoCD, and Ansible demonstrates a production-grade deployment pipeline suitable for real-world applications.

Future work includes:

- Migration to PostgreSQL (Cloud SQL) for horizontal read scalability.
- Implementation of a comprehensive test suite (unit, integration, and end-to-end).
- Addition of Prometheus `/metrics` endpoint with custom business metrics.
- Progressive delivery strategies (canary deployments) via Argo Rollouts.
- OAuth2 integration for enterprise single sign-on.
- Cost optimization analysis comparing GKE Autopilot vs. Standard mode.

The complete source code, infrastructure configurations, and deployment manifests are available as an open-source repository, enabling practitioners to adopt and adapt this architecture for their own applications.

References

[1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley, 2010.

[2] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, 2014.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[4] Y. Brikman, *Terraform: Up & Running*, 2nd ed. O'Reilly Media, 2019.

[5] T. A. Limoncelli, "GitOps: A path to more self-service IT," *Communications of the ACM*, vol. 61, no. 9, pp. 38–42, 2018.

[6] OpenJS Foundation, "Node.js Documentation," https://nodejs.org/docs/latest/api/, 2024.

[7] OpenJS Foundation, "Express.js 5.x Documentation," https://expressjs.com/, 2024.

[8] Cloud Native Computing Foundation, "Kubernetes Documentation," https://kubernetes.io/docs/, 2024.

[9] Argo Project, "Argo CD — Declarative GitOps CD for Kubernetes," https://argo-cd.readthedocs.io/, 2024.