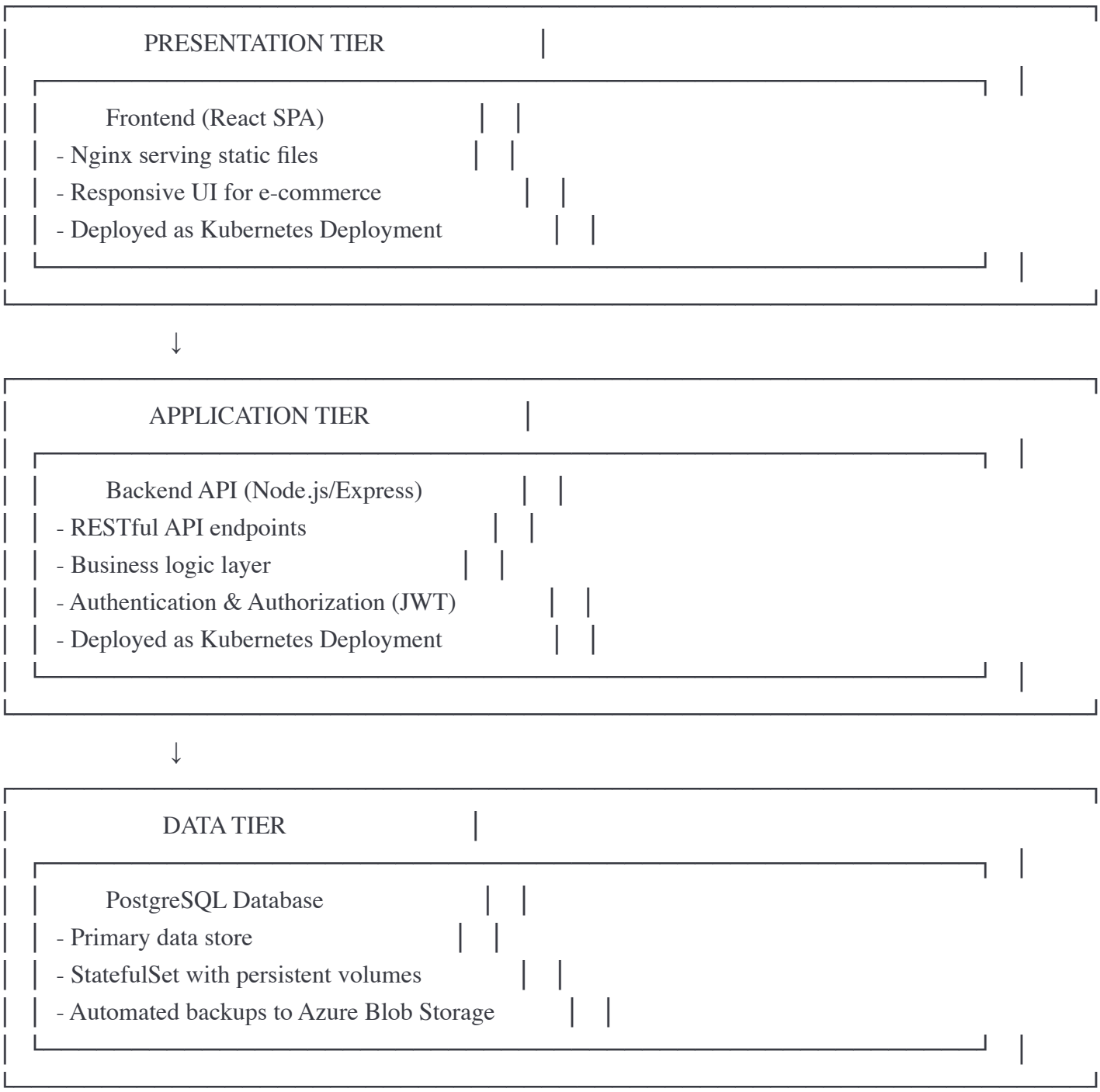
Azure DevSecOps CI/CD Pipeline Documentation

📋 Table of Contents

🏗️ Architecture Overview

3-Tier Application Architecture



Azure Infrastructure Components

- **Azure Kubernetes Service (AKS)**: Container orchestration
- **Azure Container Registry (ACR)**: Private Docker registry
- **Azure Key Vault**: Secrets management
- **Azure Blob Storage**: Database backups
- **Azure Application Insights**: APM & monitoring
- **Azure Log Analytics**: Centralized logging
- **Azure Security Center**: Security posture management

🔄 Pipeline Stages Breakdown

Stage 1: Code Quality & Security Scanning

Duration: ~8-10 minutes
Purpose: Shift-left security - catch vulnerabilities early

What Happens:

1. **Secret Scanning (Gitleaks)**
   - Scans entire repository for hardcoded secrets
   - Checks commit history for exposed credentials
   - Prevents API keys, passwords, tokens from reaching production

   Why Important: 80% of data breaches involve compromised credentials. Catching them in source code prevents exposure.
2. **Static Code Analysis (SonarQube)**
   - Analyzes code for bugs, code smells, vulnerabilities
   - Measures code coverage, complexity, duplication
   - Enforces quality gates (coverage > 80%, no critical issues)

   Why Important: Maintains code quality, reduces technical debt, prevents common vulnerabilities (SQL injection, XSS)
3. **Unit Testing**
   - Runs frontend React tests (Jest)
   - Runs backend Node.js tests (Mocha/Jest)
   - Generates coverage reports

   Why Important: Ensures individual components work correctly before integration
4. **SAST - Static Application Security Testing (Snyk Code)**
   - Deep security analysis of source code
   - Identifies OWASP Top 10 vulnerabilities
   - Provides fix recommendations

   Why Important: Finds security flaws that traditional linters miss

Tools Used:

- **Gitleaks**: Secret detection
- **SonarQube**: Code quality & security
- **Snyk**: Vulnerability scanning
- **Jest/Mocha**: Unit testing

Stage 2: Infrastructure Security & Provisioning

Duration: ~5-7 minutes (scanning), ~15-20 minutes (deployment)
Purpose: Ensure infrastructure is secure before creation

What Happens:

1. **IaC Security Scanning (Checkov)**
   - Scans Terraform files for misconfigurations
   - Checks for:
     - Open security groups
     - Unencrypted storage
     - Missing backup configurations
     - Insecure network settings

   Why Important: Prevents deploying insecure cloud infrastructure
2. **Terraform Security (tfsec)**
   - Additional Terraform-specific security checks

    - Validates Azure best practices
    - Checks for compliance violations

   Why Important: Defense in depth - multiple tools catch different issues

3. **Terraform Deployment**
    - Creates/updates Azure resources:
        - AKS cluster with RBAC
        - ACR with private endpoints
        - Key Vault with access policies
        - Virtual Networks with NSGs
        - Managed identities

   Why Important: Infrastructure as Code ensures consistency, repeatability, version control

Tools Used:

- **Checkov**: IaC security scanning
- **tfsec**: Terraform security
- **Terraform**: Infrastructure provisioning

Infrastructure Created:

hcl

```
# AKS Cluster
- Node pools (system + user)
- Azure CNI networking
- Azure AD integration
- Pod security policies
- Network policies

# Networking
- Virtual Network (10.0.0.0/16)
- Subnets for AKS, databases, services
- Network Security Groups
- Azure Firewall

# Security
- Key Vault for secrets
- Managed identities
- Azure AD integration
- Private endpoints
```

Stage 3: Build & Container Security

Duration: ~10-15 minutes
Purpose: Build secure container images

What Happens:

1. **Docker Image Build**
   - Multi-stage builds for smaller images
   - Frontend: Nginx + React build
   - Backend: Node.js production image
   - Uses specific base image versions (no :latest)

   Why Important: Reproducible builds, smaller attack surface
2. **Container Vulnerability Scanning (Trivy)**
   - Scans OS packages for CVEs
   - Scans application dependencies
   - Checks for misconfigurations
   - Reports HIGH and CRITICAL vulnerabilities

   Why Important: Container images can have thousands of vulnerabilities. Trivy finds them before deployment.
3. **Image Signing (Cosign)**
   - Cryptographically signs images
   - Enables supply chain verification
   - Prevents image tampering

   Why Important: Ensures only verified images run in production
4. **Push to ACR**
   - Uploads images to private registry
   - Tags with build ID and 'latest'

   Why Important: Centralized, secure image storage

Tools Used:

- **Docker**: Container building
- **Trivy**: Vulnerability scanning
- **Cosign**: Image signing
- **Azure Container Registry**: Image storage

Example Dockerfile (Backend):

dockerfile

```
# Build stage
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
RUN npm run build

# Production stage
FROM node:18-alpine
RUN addgroup -g 1001 -S nodejs && \
    adduser -S nodejs -u 1001
WORKDIR /app
COPY --from=builder --chown=nodejs:nodejs /app/dist ./dist
COPY --from=builder --chown=nodejs:nodejs /app/node_modules ./node_modules
USER nodejs
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

Stage 4: Database Migration & Security

Duration: ~5-10 minutes
Purpose: Safely apply database schema changes

What Happens:

1. **Retrieve Secrets from Key Vault**
   - Gets database credentials securely
   - Never exposes secrets in logs

   Why Important: Centralized secret management
2. **Database Backup**
   - Creates pg_dump backup
   - Uploads to Azure Blob Storage
   - Retention: 30 days

   Why Important: Safety net for rollback if migration fails
3. **Run Flyway Migrations**
   - Applies versioned SQL scripts
   - Tracks applied migrations in schema_version table
   - Idempotent - can run multiple times safely

   Why Important: Version-controlled database changes prevent drift
4. **Schema Validation**
   - Verifies migration success
   - Checks schema version

   Why Important: Confirms database is in expected state

Tools Used:

- **Azure Key Vault**: Secret management
- **pg_dump**: PostgreSQL backup
- **Flyway**: Database migrations

- **Azure Blob Storage**: Backup storage

Migration Example:

sql

```sql
-- V1.0__initial_schema.sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL,
    stock_quantity INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    total_amount DECIMAL(10, 2) NOT NULL,
    status VARCHAR(50) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Stage 5: Kubernetes Manifest Security

Duration: ~3-5 minutes
Purpose: Validate K8s configurations before deployment

What Happens:

1. **Kubesec Security Scan**
    - Analyzes Kubernetes manifests
    - Checks for:
        - Missing security contexts
        - Privileged containers
        - Host network/PID namespace usage
        - Missing resource limits
    - Assigns security score (0-10)

Why Important: Kubernetes misconfigurations are common attack vectors

2. **OPA Policy Validation (Conftest)**
    - Enforces organizational policies:
        - All pods must have resource limits
        - No root containers allowed
        - Images must come from approved registries
        - All ingresses must have TLS

    Why Important: Enforces governance and compliance automatically
3. **Helm Template Generation**
    - Generates K8s manifests from Helm charts
    - Injects environment-specific values

    Why Important: Reusable, parameterized deployments

Tools Used:

- **Kubesec**: K8s security analysis
- **Conftest (OPA)**: Policy enforcement
- **Helm**: Package management

Example K8s Security Best Practices:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  template:
    spec:
      securityContext:
        runAsNonRoot: true
        runAsUser: 1001
        fsGroup: 1001
      containers:
      - name: backend
        image: acr.azurecr.io/backend:123
        securityContext:
          allowPrivilegeEscalation: false
          readOnlyRootFilesystem: true
          capabilities:
            drop:
              - ALL
        resources:
          limits:
            cpu: "1"
            memory: "512Mi"
          requests:
            cpu: "500m"
            memory: "256Mi"
        livenessProbe:
          httpGet:
            path: /health
            port: 3000
          initialDelaySeconds: 30
        readinessProbe:
          httpGet:
            path: /ready
            port: 3000
          initialDelaySeconds: 5
```

Stage 6: Staging Deployment

Duration: ~8-12 minutes
Purpose: Deploy to production-like environment for testing

What Happens:

1. **Connect to AKS Staging Cluster**
    - Authenticates with service principal
    - Sets kubectl context
2. **Create Namespace & Secrets**
    - Isolates staging environment
    - Injects secrets from Key Vault as K8s secrets
3. **Deploy Database (PostgreSQL)**
    - StatefulSet with persistent volume
    - Configured for staging workloads
4. **Deploy Backend API**
    - Creates Deployment, Service
    - Auto-scaling enabled (HPA)
    - Health probes configured
5. **Deploy Frontend**
    - Creates Deployment, Service
    - Nginx configuration for SPA routing
6. **Deploy Ingress**
    - Configures NGINX Ingress Controller
    - SSL/TLS termination
    - Routes traffic to services
7. **Wait for Rollout**
    - Monitors deployment progress
    - Fails if pods don't become ready

   Why Important: Validates deployment works before production

Tools Used:

- **kubectl**: Kubernetes CLI
- **Kubernetes Manifests**: Resource definitions
- **NGINX Ingress**: Load balancing

Stage 7: Integration & DAST Testing

Duration: ~15-20 minutes
Purpose: Validate functionality and runtime security

What Happens:

1. **API Integration Tests (Newman/Postman)**
    - Tests all API endpoints
    - Validates business logic
    - Checks authentication flows
    - Tests error handling

   Why Important: Ensures API works end-to-end
2. **E2E Tests (Playwright/Cypress)**
    - Simulates user workflows:
        - User registration
        - Product browsing
        - Add to cart
        - Checkout process
    - Takes screenshots on failure

   Why Important: Validates complete user journeys
3. **DAST - Dynamic Application Security Testing (OWASP ZAP)**
    - Runs active security scans against running app
    - Tests for:
        - SQL injection
        - XSS attacks

        - Authentication bypasses
        - CSRF vulnerabilities
    - Generates detailed report

  Why Important: Finds runtime vulnerabilities that SAST can't detect

Tools Used:

- **Newman**: API testing
- **Playwright**: E2E testing
- **OWASP ZAP**: DAST scanning

Example Integration Test:

javascript

```javascript
// Postman/Newman test
pm.test("User can login successfully", function () {
    pm.response.to.have.status(200);
    const response = pm.response.json();
    pm.expect(response).to.have.property('token');
    pm.environment.set('authToken', response.token);
});

pm.test("Can create order with valid token", function () {
    pm.response.to.have.status(201);
    const response = pm.response.json();
    pm.expect(response.order).to.have.property('id');
});
```

Stage 8: Production Deployment (Blue-Green)

Duration: ~15-20 minutes
Purpose: Zero-downtime production deployment

What Happens:

1. **Manual Approval Gate**
    - Notifies ops team and CTO
    - Requires human approval
    - Includes test results summary

  Why Important: Final human check before production
2. **Deploy to Blue Environment**
    - Deploys new version alongside current (green)
    - Blue runs but doesn't receive traffic
3. **Smoke Tests on Blue**
    - Validates health endpoints
    - Checks database connectivity
    - Confirms core functionality

  Why Important: Verifies deployment before switching traffic

4. **Traffic Switch**
    - Updates service selector to point to blue
    - All new requests go to new version
    - Green remains running
5. **Monitor New Deployment**
    - Watches for 5 minutes
    - Monitors error rates
    - Checks pod health
6. **Automatic Rollback**
    - If error rate > threshold, rolls back
    - Switches traffic back to green
    - Preserves green deployment

   Why Important: Fast rollback if issues detected

Tools Used:

- **kubectl**: Deployment management
- **Kubernetes Services**: Traffic routing
- **Prometheus**: Metrics monitoring

Blue-Green Architecture:


yaml

```yaml
# Blue Deployment (new version)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-blue
spec:
  selector:
    matchLabels:
      app: backend
      version: blue
  template:
    metadata:
      labels:
        app: backend
        version: blue
    spec:
      containers:
      - name: backend
        image: acr.azurecr.io/backend:NEW_VERSION

---
# Service switches between blue and green
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
    version: blue  # Switch to green for rollback
  ports:
  - port: 80
    targetPort: 3000
```

Stage 9: Post-Deployment Validation

Duration: ~5-8 minutes
Purpose: Validate production health and setup monitoring

What Happens:

1. **Production Health Checks**
   - Tests frontend accessibility
   - Tests API health endpoint
   - Validates database connectivity

2. **Performance Baseline (K6)**
    - Runs load test
    - Validates response times < 500ms
    - Ensures throughput meets SLA
    - Checks error rate < 0.1%

   Why Important: Ensures performance doesn't degrade
3. **Configure Application Insights**
    - Sets up alert rules:
        - High error rate (>100/5min)
        - Slow response time (>2s avg)
        - Low availability (<99%)
    - Configures dashboards
4. **Enable Container Insights**
    - Monitors AKS cluster health
    - Tracks pod metrics
    - Alerts on resource exhaustion

   Why Important: Proactive monitoring prevents outages

Tools Used:

- **K6**: Performance testing
- **Azure Application Insights**: APM
- **Azure Monitor**: Infrastructure monitoring

Stage 10: Security Compliance & Reporting

Duration: ~10-15 minutes
Purpose: Generate audit reports and maintain compliance

What Happens:

1. **Azure Security Center Assessment**
    - Retrieves security recommendations
    - Lists compliance violations
2. **CIS Kubernetes Benchmark (kube-bench)**
    - Runs CIS benchmark tests
    - Checks:
        - API server configuration
        - Controller manager settings
        - Scheduler configuration
        - etcd security
        - Node security

   Why Important: Industry standard compliance
3. **Generate SBOM (Syft)**
    - Creates Software Bill of Materials
    - Lists all packages and versions
    - Enables vulnerability tracking

   Why Important: Supply chain security and compliance requirements
4. **Consolidated Report**
    - Aggregates all security scans
    - Pass/fail for each stage
    - Links to