Scenario: You're building the infrastructure for our new payment processing microservice called "transaction-validator". This service validates payment transactions before they're sent to payment processors. It needs to be production-ready, secure, and observable.

Assignment Overview

Build a complete, production-ready infrastructure setup that includes:

1. AWS infrastructure (Terraform/CloudFormation)
2. Kubernetes manifests
3. CI/CD pipeline configuration
4. Architecture documentation

Deliverables:

- GitHub repository with all code
- Architecture diagram
- README with setup instructions
- A brief document explaining your design decisions

Part 1: Infrastructure as Code

Requirements:

Create Terraform code (or CloudFormation if you prefer) to provision:

a. **VPC & Networking:**
    i. VPC with public and private subnets across 3 AZs
    ii. NAT Gateway setup for private subnets
    iii. Appropriate security groups for:
        1. EKS cluster
        2. Aurora PostgreSQL database
        3. Redis ElastiCache
        4. Application Load Balancer

b. **Data Layer:**
    i. Aurora PostgreSQL cluster (Multi-AZ)
    ii. Appropriate parameter group for OLTP workload
    iii. Encrypted at rest
    iv. Automated backups with 7-day retention
    v. Connection pooling consideration
    vi. ElastiCache Redis cluster
        1. Cluster mode enabled

        2. Encryption in transit and at rest
        3. Appropriate node size for caching session data

c. **Compute:**
    i. EKS cluster (you can use terraform-aws-modules/eks/aws)
    ii. Node groups with appropriate instance types
    iii. IRSA (IAM Roles for Service Accounts) enabled
    iv. Cluster autoscaler configuration
    v. IAM roles following least privilege principle

d. **Secrets Management:**
    i. AWS Secrets Manager for database credentials
    ii. Appropriate IAM policies for pods to access secrets

e. **Observability:**
    i. CloudWatch Log Groups with appropriate retention
    ii. SNS topic for alerts

Challenge Requirements:

1. **Make it modular** - Structure your Terraform so it can be reused for dev/staging/prod
2. **Cost-conscious** - Use appropriate instance sizes (we're a startup)
3. **Security-first** - No hardcoded credentials, proper encryption, minimal IAM permissions
4. **Document your tfvars** - Show how different environments would be configured

What We're Evaluating:

- Code organization and modularity
- Security best practices
- Understanding of AWS networking
- Ability to balance cost vs. performance
- Documentation quality

Part 2: Kubernetes Manifests

Requirements:

Create Kubernetes manifests for the `transaction-validator` service:

Application Specs:

- Listens on port 8080
- Requires these environment variables:
  - `DATABASE_URL` (from Secrets Manager)
  - `REDIS_ENDPOINT` (from ElastiCache)
  - `LOG_LEVEL`
  - `MAX_TRANSACTIONS_PER_SECOND` (for rate limiting)
- Docker image: `your-registry/transaction-validator:v1.0.0` (mock this)

What to Create:

1. **Deployment manifest:**
   - Minimum 3 replicas for HA
   - Proper resource requests and limits (justify your choices)
   - Health checks (liveness, readiness, startup probes)
   - Rolling update strategy appropriate for financial services
   - Pod Disruption Budget
   - Security context (non-root user, read-only root filesystem)
   - Pod anti-affinity to spread across AZs
2. **Service Account:**
   - With IRSA annotations for accessing Secrets Manager
   - Minimal IAM policy document
3. **ConfigMap and Secret:**
   - ConfigMap for non-sensitive configuration
   - External Secrets Operator manifest (or show how you'd integrate with Secrets Manager)
4. **Service:**
   - ClusterIP service
   - Appropriate port configuration
5. **Ingress/ALB:**
   - AWS Load Balancer Controller annotations
   - SSL/TLS termination
   - Path-based routing to `/api/validate`
   - WAF integration (show annotations/configuration)
6. **HorizontalPodAutoscaler:**
   - Scale based on CPU and custom metrics (if possible)
   - Justify your min/max replicas and target utilization
7. **NetworkPolicy:**
   - Restrict ingress to only ALB and internal services
   - Restrict egress to only database, redis, and external APIs

Bonus (Optional):

- Customize structure for managing dev/staging/prod
- Service Mesh configuration (Istio/Linkerd) if you have experience

What We're Evaluating:

- Understanding of production Kubernetes patterns
- Security consciousness
- Resource management
- High availability design
- Understanding of pod lifecycle

Part 3: CI/CD Pipeline

Requirements:

Create a GitLab CI or GitHub Actions pipeline that:

Pipeline Stages:

1. **Test & Build:**
   - Lint Terraform code (terraform fmt, tflint)
   - Validate Kubernetes manifests (kubeval or similar)
   - Run security scanning (trivy, checkov, or similar)
   - Build Docker image
   - Scan Docker image for vulnerabilities
2. **Infrastructure Deploy (Staging):**
   - Plan Terraform changes
   - Require manual approval
   - Apply Terraform
   - Run smoke tests
3. **Application Deploy (Staging):**
   - Deploy to EKS staging
   - Run integration tests
   - Automated rollback on failure
4. **Production Deploy:**
   - Manual approval required
   - Blue-green or canary deployment strategy
   - Automated health checks
   - Automatic rollback on failure

Additional Requirements:

- Use proper secrets management (no hardcoded credentials)
- Artifact management for Docker images

- Terraform state management (S3 backend configuration)
- Pipeline should be idempotent

What to Include:

- Complete pipeline YAML file(s)
- Scripts for any custom steps
- Documentation on how to set up required CI/CD variables
- Explanation of your deployment strategy choice

What We're Evaluating:

- CI/CD best practices
- Security in pipeline
- Understanding of deployment strategies
- Error handling and rollback procedures
- Documentation clarity

Part 4: Architecture & Documentation

Requirements:

1. Architecture Diagram: Create a comprehensive architecture diagram showing:

- Network topology (VPC, subnets, routing)
- All AWS services and their relationships
- Data flow for a transaction validation request
- Security boundaries
- DR/backup strategy

Use any tool (draw.io, Lucidchart, CloudCraft, Terraform Graph, or even hand-drawn)

2. Design Document (2-3 pages max):

Write a concise document covering:

a) Architecture Decisions:

- Why you chose specific instance types/sizes
- Database connection pooling strategy
- Caching strategy for Redis
- Security measures implemented
- Cost estimates (rough AWS calculator)

b) Trade-offs:

- What you optimized for (cost vs. performance vs. reliability)

- What you'd do differently with unlimited budget
- What you'd change if traffic increased 10x

c) Production Readiness Checklist:

- What's missing for true production deployment?
- What would you add in Phase 2?
- Known limitations of your design

d) Disaster Recovery:

- RTO (Recovery Time Objective) and RPO (Recovery Point Objective)
- Backup strategy
- Failover procedure
- How you'd handle total AZ failure

e) Compliance Considerations:

- PCI-DSS relevant controls you implemented
- Data encryption approach
- Audit logging strategy

3. Runbook Template: Create a runbook template for common operational tasks:

- How to check application health
- How to scale the application
- How to perform database maintenance
- How to investigate high latency issues
- How to perform rollback

What We're Evaluating:

- Systems thinking
- Communication skills
- Consideration of operational concerns
- Understanding of fintech requirements
- Pragmatism (startup realities vs. ideal solutions)

Note:

We value pragmatism. A well-documented, partially complete solution is better than a rushed, complete one without explanations.

Tips for Success

✅ **DO:**

- Start with a working MVP, then iterate
- Document as you go
- Make assumptions explicit
- Show your thinking in commit messages
- Use industry-standard tools and patterns
- Consider operational burden of your choices

❌ **DON'T:**

- Over-engineer (we're a startup, not a Fortune 500)
- Copy-paste without understanding
- Ignore security for "I'll add it later"
- Create brittle, tightly-coupled solutions
- Forget about cost implications
