**Scenario 1 - development set-up**:

1. Discuss & justify potential deployment options in terms of infrastructure for development environment of above application

2. Demonstrate example infrastructure set-up and application deployment model (CD)

Based on the information provided, I'll discuss potential deployment options for the development environment and demonstrate an example infrastructure setup and application deployment model (CD) for this application on AWS EKS.

1. Potential deployment options for development environment:

a) Single EKS cluster:

* Pros: Simplifies management, reduces costs
* Cons: Less isolation between components

b) Multiple EKS clusters (one per component):

* Pros: Better isolation, allows independent scaling
* Cons: Higher complexity and cost

c) Hybrid approach (EKS for core services, EC2 for databases):

* Pros: Flexibility, cost-effective for stateful services
* Cons: Increased management overhead

Based on the requirements and characteristics of the voting app, I believe option (a) - a single EKS cluster - is the best choice for deploying this application in a development environment.

**Here's why:**

1. **Simplicity**: The voting app has multiple interconnected components (Python frontend, Redis, .NET worker, Postgres database, and Node.js results app). Deploying all of these in a single EKS cluster simplifies the overall architecture and makes it easier to manage and troubleshoot during development.

2. **Cost-effectiveness**: For a development environment, using a single cluster is more cost-effective than running multiple clusters or a hybrid setup. This is particularly important when resources may not need to be as highly available or scalable as in production.

3. **Inter-service communication**: The components of this app need to communicate with each other frequently (e.g., the worker consuming votes from Redis and storing them in Postgres). A single cluster makes this communication asyn more straightforward and potentially faster.

4. **Resource utilization**: In a development environment, I don't need the level of isolation or independent scaling that multiple clusters would provide. A single cluster allows for better resource utilization across all components.

5. **Easier testing and debugging**: Having all components in one cluster makes it easier to test the entire application flow and debug issues that may span multiple services.

6. **Stateful services**: While Postgres is typically run as a stateful service, for a development environment, running it within the EKS cluster (perhaps using a StatefulSet) can be acceptable and simplifies the overall setup compared to managing a separate EC2 instance.

7. **One-vote-per-client requirement**: This functionality is likely implemented at the application level (in the Python frontend or .NET worker) and doesn't significantly impact the deployment architecture choice.

**Deployment model for this option:**

I understand you're looking for a comprehensive deployment model that leverages all the tools on the Job Description, you've mentioned: Kubernetes (EKS), ArgoCD, GitHub & GitHub Actions, Terraform, Helm Charts, and Ansible, with a preference for AWS and more... I will design a deployment model that incorporates all these technologies for the voting app. I'll explain my thought process and approach to this engineering problem.

Deployment Model:

1. Infrastructure Provisioning with Terraform:

* I used Terraform to provision the AWS infrastructure, including the EKS cluster, VPC, subnets, security groups, and any additional resources like RDS for PostgreSQL and ElastiCache for Redis.
* Terraform allows us to define our infrastructure as code, making it versioned, repeatable, and easily manageable.

1. Application Containerization:

* I will dockerize each component of the voting app (Python frontend, .NET worker, Node.js results app).
* Store Docker images in Amazon ECR or Docker Hub.

1. Kubernetes Manifests with Helm Charts:

* I created a Helm chart for each component of the application.
* The Helm charts allow for templating and easier management of Kubernetes manifests.

1. GitOps with ArgoCD:

* Set up ArgoCD in the EKS cluster to manage deployments.Define ArgoCD applications that point to the Helm charts in your GitHub repository.

1. CI/CD with GitHub Actions:

* Set up GitHub Actions workflows for CI/CD.
* On push to main branch:

a. Build and test the application

b. Build and push Docker images

c. Update Helm chart versions

d. Commit changes to the GitOps repo

1. Ansible for Configuration Management:

* Use Ansible for any additional configuration management tasks, such as:

a. Initial EKS node setup

b. Installing monitoring agents

c. Applying security patches

**Approach and Thinking Process:**

1. **Infrastructure as Code:** Using Terraform for infrastructure provisioning ensures repeatability and version control of our AWS resources.
2. **Containerization**: Dockerizing applications provides consistency across environments and simplifies deployments.
3. **Kubernetes for Orchestration**: EKS provides a managed Kubernetes platform, allowing for scalable and resilient application deployment.
4. **Helm for Package Management**: Helm charts simplify the management of complex Kubernetes applications and allow for easy versioning.
5. **GitOps with ArgoCD**: Implementing GitOps principles ensures that the desired state of our application is always reflected in our Git repository, with ArgoCD automatically syncing changes.
6. **CI/CD Automation**: GitHub Actions automates our build, test, and deployment processes, ensuring quick and consistent updates.
7. **Configuration Management**: Ansible fills the gap for any additional configuration needs that aren't covered by container orchestration.

This approach creates a fully automated, scalable, and manageable deployment pipeline. It leverages the strengths of each tool:

* Terraform for infrastructure
* Docker for containerization
* Kubernetes (EKS) for orchestration
* Helm for Kubernetes package management
* ArgoCD for GitOps-based deployments
* GitHub Actions for CI/CD
* Ansible for additional configuration management

This model allows for easy scaling, quick rollbacks, and consistent deployments across different environments. It also provides a clear separation of concerns between infrastructure provisioning, application deployment, and configuration management.