Pierre Corazo Cesario

Vimigo DevOps Engineer Assessment

# Question 1

**A fast-growing application requires multiple teams working together with high velocity and less dependency on each other during development. For example, User Profile is a shared resource throughout the application and any changes might break a feature. Propose a one-page solution with plans and technology stack you might use to reduce breaking changes and dependencies. (10 points)**

# Answer 1

To solve these issues, there are three main aspects to be implemented:

1. Source Control Management (SCM)
2. Build and test frameworks
3. Continuous Integration and Continuous Delivery (CI/CD) Pipeline

## Source Control Management

First, an SCM for repository code should be used. With Git's version control and the use of a distributed cloud version control service such as GitHub or GitLab, the origin server, software developers can first commit their changes locally and then push it to the origin repository in the cloud for other developers to pull. Through the use of Git branches, each developer can branch off from the main branch to work on their respective feature or bugfix so as to prevent any conflicts with other developers' changes in code.

### Best Practices

Of course, while Git itself is a great tool for developers to manage and organise the codebase, it's also important to note the best practices on how to use it as well as the proper communication required to prevent merge conflicts and breaking changes.

There are a few practices that all developers should habitualise:

* Frequently commit changes
* Frequently push commits to the origin server
* Frequently pull from the origin server

Such practices can prevent breaking changes since frequent small commits are less likely to have huge sudden changes in the code that may affect each developer's local code.

### Communication

Communication is also vital between developers. For example, picture two developers working on two different branches, Branch B and Branch C, that branch out from the same branch, Branch A. There's a bug in Branch A, so both developers make changes to the same lines of code in their own branches albeit with different approaches. If they both try to merge their changes back into Branch A, there will be a merge conflict. While this is the proper use of branches, the problem here is the lack of communication between developers, something rather vital when it comes to DevOps and Agile based teams. If both developers had known each other's tasks, they wouldn't have had this merge conflict to deal with.

## Build and Test Frameworks

Second, assuming the company works on their products through a monorepo, a build and test framework should be used to manage each package's versions, run tests and build the artefacts. Such build frameworks include Lerna, Turborepo and Nx. In addition to managing the versions of the individual packages in the monorepo, they can be used to run tests and then build the artefacts from the code. To illustrate, Cypress can be used to run End-to-End tests on the application's frontend while Jest can be used to run Unit Tests on the backend (assuming it is JavaScript/TypeScript)

## Continuous Integration/Continuous Deployment Pipeline

Third, a CI/CD server should be incorporated into the build and deployment process. This is because every one of the aforementioned build and test steps can be automated by it to promote repeatability and reliability since there would be no human error due to manual repetition. For example, builds can be triggered either manually or upon new commits being pushed to the main branch on the origin server. Furthermore, it is crucial that the finalised code is first deployed onto a staging server or environment so as to replicate the end user’s experience. Once the build has been validated on the staging environment, then only can it be pushed to production.

## Proposed Technology Stack

The following are the proposed plans and technology stack to be used for this project:

* Version Control System: Git
* Distributed Cloud-based Origin Git Repository: GitHub
* Build Framework: Nx can be used to track and control dependencies between packages in the monorepo and to run builds and tests before artefacts are rolled out to the public
* Unit Test Framework: Jest for JavaScript/TypeScript codebases
* End-to-End Test Framework: Cypress
* CI/CD server: Jenkins can be hosted on an AWS EC2 instance to automate the build, test and deployment process
* Communication: Atlassian’s Jira and Confluence can be used to organise and divide work among teams.

# Question 2

**Significant changes are much easier to develop on a new application than on an existing one. The challenge is to migrate existing data to the new application. The easiest solution is to have downtime, but how would you do it without downtime? Propose a two-page solution with plans and technology stack you would use for the migration with less impact on the user experience. (20 points)**

# Answer 2

Essentially, a zero downtime deployment strategy needs to be implemented to address this issue. The deployment strategy to be employed here is a rolling deployment.

## Rolling Deployment Concept

This can be achieved if the deployment structure of the application employs the use of a load balancer that distributes the load amongst multiple instances. To illustrate, picture three Kubernetes pods, which are all replicas A, B and C, managed by the Kubernetes load balancer. All three pods are currently live and run V1 of the application with the existing database. First, routing to pod A is stopped. While it is stopped, pod A's application is updated to the latest version, V2, which is set to connect to the same database. Having the rollout be done this way allows new incoming requests to still be redirected to pods B and C while pod A gets updated to V2 - no downtime for the end user. Then, a test can be done on pod A to ensure it can successfully receive traffic before it is then added back into the pool of available pods. This process is then repeated for all instances in production until the entire pool runs V2.

This is the concept behind a zero downtime rollout deployment but various parameters may be changed to suit the task at hand. For example, if there were 10 instances in production, the changes can be made to 2-3 instances at a time instead of just 1 to speed up the upgrade process – of course this is dependent on the overall load on the instances at the time of rollout. Additionally, in the case of a Kubernetes cluster deployment, the *maxSurge* and *maxUnavailable* parameters can be set in the deployment pod’s template. *maxSurge* determines how many excess pods can be instantiated in addition to the desired target pods while *maxUnavailable* determines the maximum number of pods that may be taken out from the pool for updates at any one time.

## Additional Measures

Of course, while this is the general idea behind the process, certain steps need to be taken depending on the type of application and feature being rolled out or even rolled back if necessary.

### Preserving State

Any requests or processes on instance A are left to complete running until there is no activity on the instance. This is to ensure user sessions or processes are not interrupted or lost.

### Preserving Compatibility Between Versions

It is important to note that the aforementioned example given for this rolling deployment method will only work assuming V1 and V2 of the application are both fully forward and backward compatible with the database. In other words, data created by V2 instances must be able to be processed by V1 instances and vice versa since two, presumably, radically different versions of the application are going to be live simultaneously.

As such, the rollout process may not be as simple as updating the instances from V1 to V2 and may involve intermediary versions with incrementally different data schemas or architectures to facilitate the transition more smoothly without downtime.

## Proposed Plans and Technology Stack

We will presume that the application is deployed using containers. As such, AWS Elastic Kubernetes Service (EKS) can be used as the cloud service for this purpose.

* First, ensure the deployment pod’s template specifies a RollingUpdate deployment strategy.
* Then, the updated V2 container artefacts are pushed to a repository – either Docker Hub or a privately hosted registry.
* Next, through the use of the *kubectl* commands, the kubernetes cluster is instructed to perform a rolling update on the entire cluster to the new version.

# Question 3

**Deploying a new feature for an application can be a hassle if something unexpected happens. To control the feature rollout, it's best to have multiple stages of deployment that can be rolled back. For example, the first deployment only affects users in Malaysia before the second deployment affects users throughout Southeast Asia. the users won’t feel any changes if the rollback happens. Propose a two-page solution with plans and technology stack to achieve a smooth transition during the deployment rollout and rollback. (20 points)**

# Answer 3

Similar to Question 2, the solution to this problem lies in providing a zero downtime deployment solution, albeit with a different approach to cater to the scenario given. Here, a Canary Deployment strategy is suitable, but employed on a multi-region deployment basis.

## Canary Deployment Concept

The canary deployment strategy revolves around routing a select few users to instances with the new feature, V2, so as to get metrics and data on the new features without exposing it to all users. This mitigates the risk of deploying a broken feature to the entire public since it is only given to a select few ‘beta’ or ‘canary’ testers. If metrics show that the response and user experience on the new version is good, the ratio of V2 to V1 traffic is increased. Otherwise, it signals to the development team that V2 may not be ready for a public release and a rollback is required. Hence, the V2 users are then redirected to the stable V1 instances. This process is repeated until the entire public has been migrated to the V2 instances after which the V1 instances may be kept for some time just in case a sudden rollback is deemed necessary.

As such, this requires 2N instances to be set up since V1 and V2 instances need to be run simultaneously. However, another more cost-efficient approach is to auto-scale the instances running V1 and V2 that are proportional to the traffic being directed to them respectively. This way, the effective number of instances running stays constant. For example, such a strategy can be implemented with the use of AWS’s autoscaling EC2 instances. However, in certain scenarios that require an immediate rollback for all users, a rollback will not be as quick since new V1 instances still have to be spun up.

One important aspect to the canary deployment strategy is that metrics dictate whether more or less traffic is redirected to the V2 instances. Such metrics to be monitored live may include latency to the new instances, dropped connections or even metrics that pertain to the deployed application.

## Additional Measures

Like for the rolling deployment scenario, while this is the general idea behind the process, certain steps need to be taken depending on the type of application and feature being rolled out or even rolled back if necessary. Furthermore, there is an importance placed on an uninterrupted and pleasant user experience during the rollout or even rollback if necessary.

### Preserving State

Any requests or processes on instance A are left to complete running until there is no activity on the instance. This is to ensure user sessions or processes are not interrupted or lost.

### Preserving Compatibility Between Versions

It is important to note that the aforementioned example given for this rolling deployment method will only work assuming V1 and V2 of the application are both fully forward and backward compatible with the database. In other words, data created by V2 instances must be able to be processed by V1 instances and vice versa since two, presumably, radically different versions of the application are going to be live simultaneously.

As such, the rollout process may not be as simple as updating the instances from V1 to V2 and may involve intermediary versions with incrementally different data schemas or architectures to facilitate the transition more smoothly without downtime.

## Region-by-Region Deployment Strategy

Using the canary deployment concept, a region-by-region deployment strategy can be done. The canary deployment concept is done on a per-region basis so that the global user base does not get redirected to V2. To illustrate, release strategy can begin with 10% of users in Malaysia being redirected to V2. Then, metrics will determine whether to increase the load balancer weighting to redirect an additional 10% to V2 or to rollback the changes. This process is repeated until all users in Malaysia have migrated to V2. Next, the same steps are repeated but for other regions in SEA until the entire SEA has been migrated to V2.

## Proposed Technology Stack

* Autoscaling Instances: AWS EC2 Auto Scaling
* Load Balancer: AWS Elastic Load Balancer
* Monitoring System: Prometheus