CI/CD: Description of a pipeline, where every commit of each developer goes through phases (stages) finishing on a development environment and promoting to production environment.

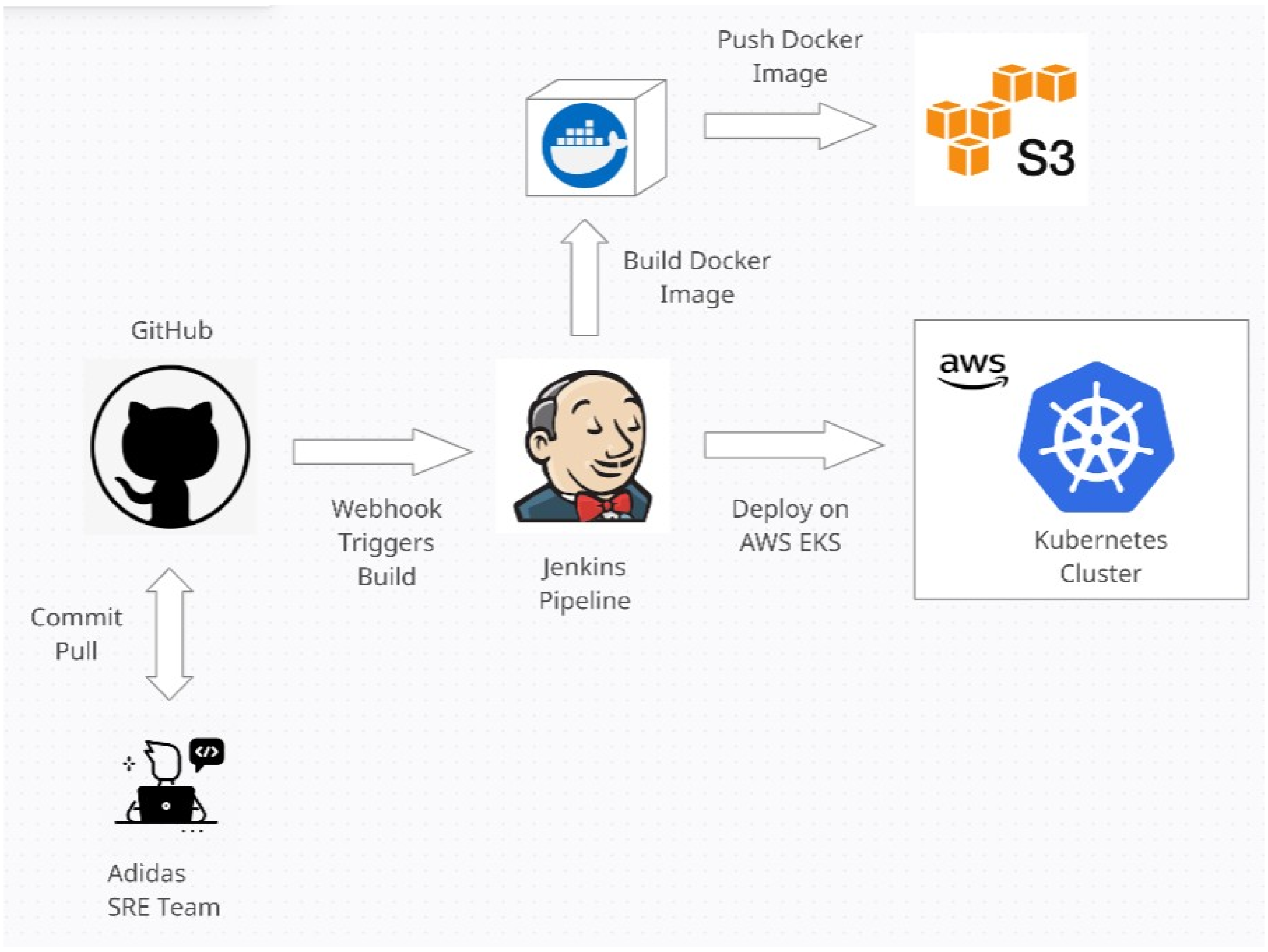
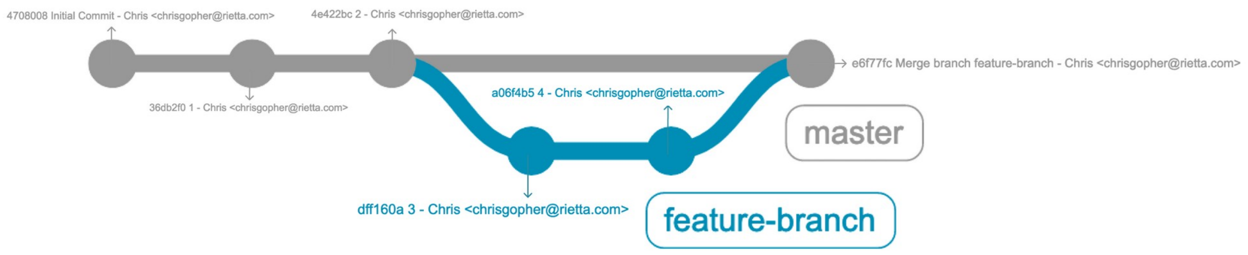


Diagram created using creately.com

CI adopting a GitOps approach to implement DevOps best practices, using GitHub as version control and merge request tool.

Dev Team members collaborate in the project 'Product Service' by merging their code changes to the different feature branches they are working on by creating a pull request:


4708008 Initial Commit - Chris <chrisgopher@rietta.com>
4e422bc 2 - Chris <chrisgopher@rietta.com>
36db2f0 1 - Chris <chrisgopher@rietta.com>
e6f77fc Merge branch feature-branch - Chris <chrisgopher@rietta.com>
a06f4b5 4 - Chris <chrisgopher@rietta.com>
master
dff160a 3 - Chris <chrisgopher@rietta.com>
feature-branch

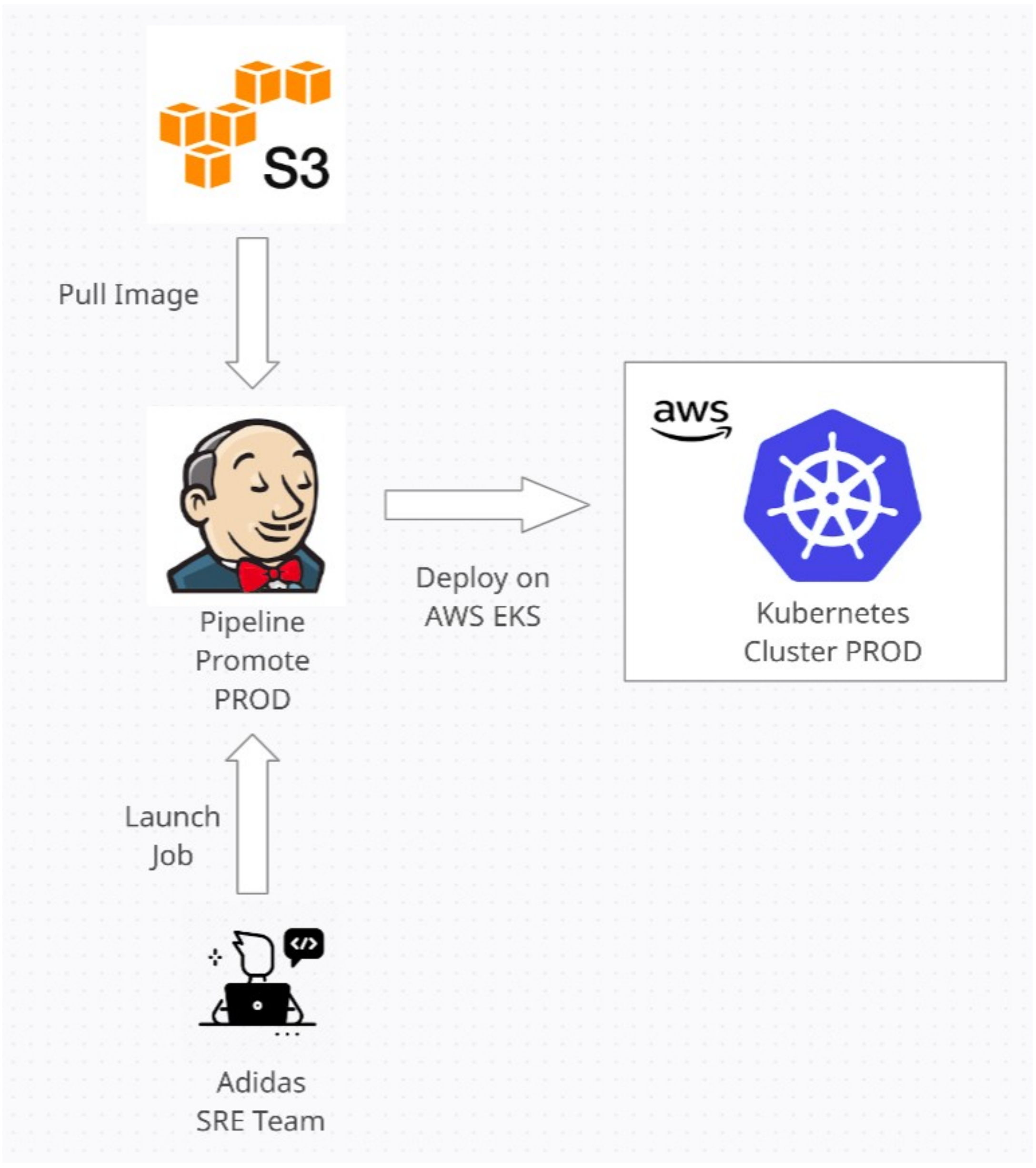
Automation: A Webhook needs to be configured in order to trigger the Jenkins job to build the application each time there is a change in the code. An access token from GitHub is required to give Jenkins access to the repository.

The job will have three main stages as defined in the Jenkinsfile: Build, Test Code, Deploy.

My solution recommends to enable a required status check on each pull request so that it must run successfully to enable automatic merging. Moreover, using pull requests acts as an audit trail.

The built image is pushed to an S3 bucket, Docker Hub, or Nexus repository if preferred, and the application is deployed in the Kubernetes Cluster for non-production running in AWS. Testing Team are now happy to start their work.

CD: Another Jenkins Pipeline can be configured with the stages to promote the image from non-production to production environment, below a diagram where I assume S3 stores the image:

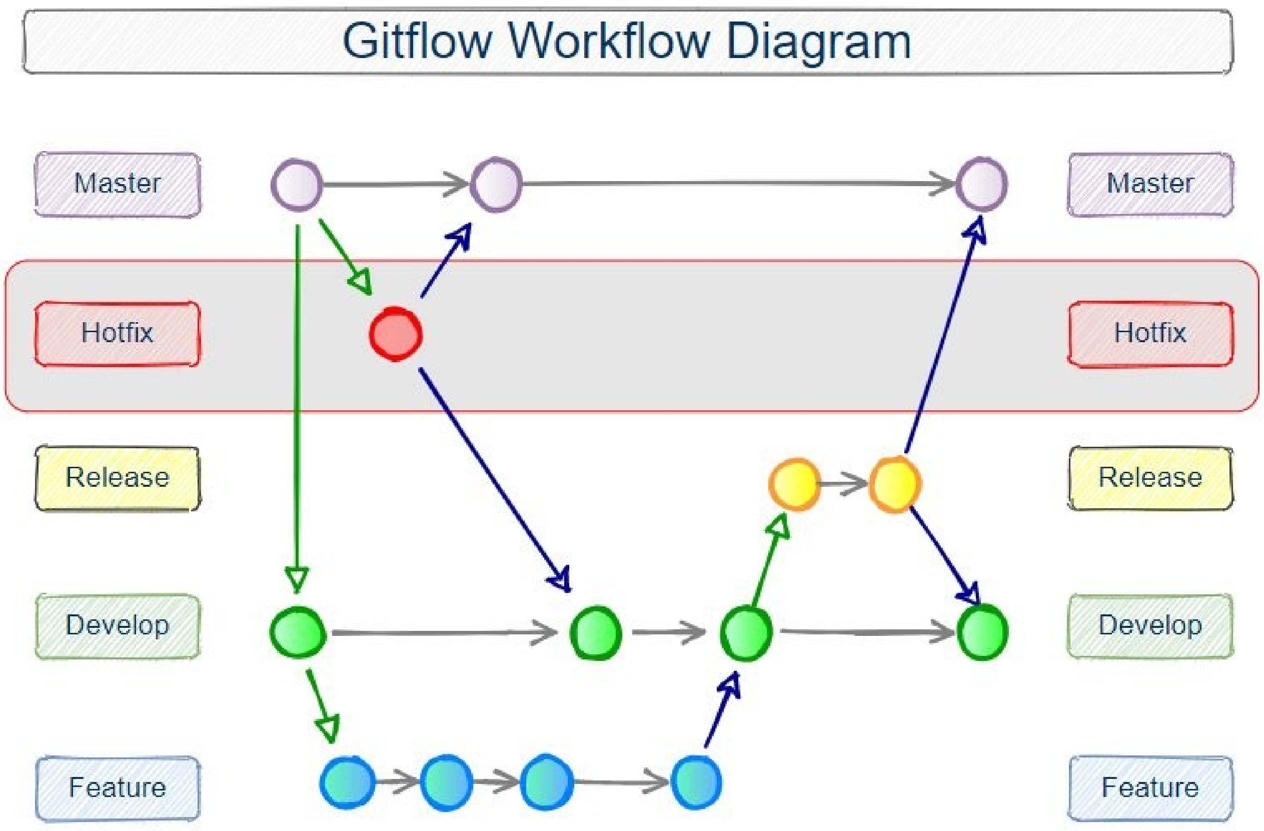


The only non-automated stage before promoting the application to production environment is a manual approval to confirm the release, implementing Continuous Delivery instead of Deployment.

All these stages can also be implemented, alternatively to Jenkins, using AWS services for CI/CD: CodeCommit, CodeBuild, CodeDeploy, CodePipeline (Defining stages, including the manual approval).

The proposal for deploying hotfixes is based on using branches, same as for working on new features.

The following Gitflow workflow diagram displays the rational:



An important point to take into account is the flexibility that this approach provides. Several scenarios are covered:

- During the development of a major release, several features are going to be included.
- In case a hotfix needs to be deployed ASAP, the developers can pull the current master branch code, work in the issue(s) and push the code to be deployed in production.
- Or the team can merge the code to fix the issue(s) to develop stream, and then merge the release features, so that hotfix+features will be included in the next release stream.

Considerations: In case the hotfix needs to be tested in parallel with the major release, the hotfix should be first tested in another environment before deploying to production. The idea is to avoid interruptions during the testing phase of the major release, and fulfil the testing requirements for the hotfix version.

Observability: Description of what would you implement to improve application observability.

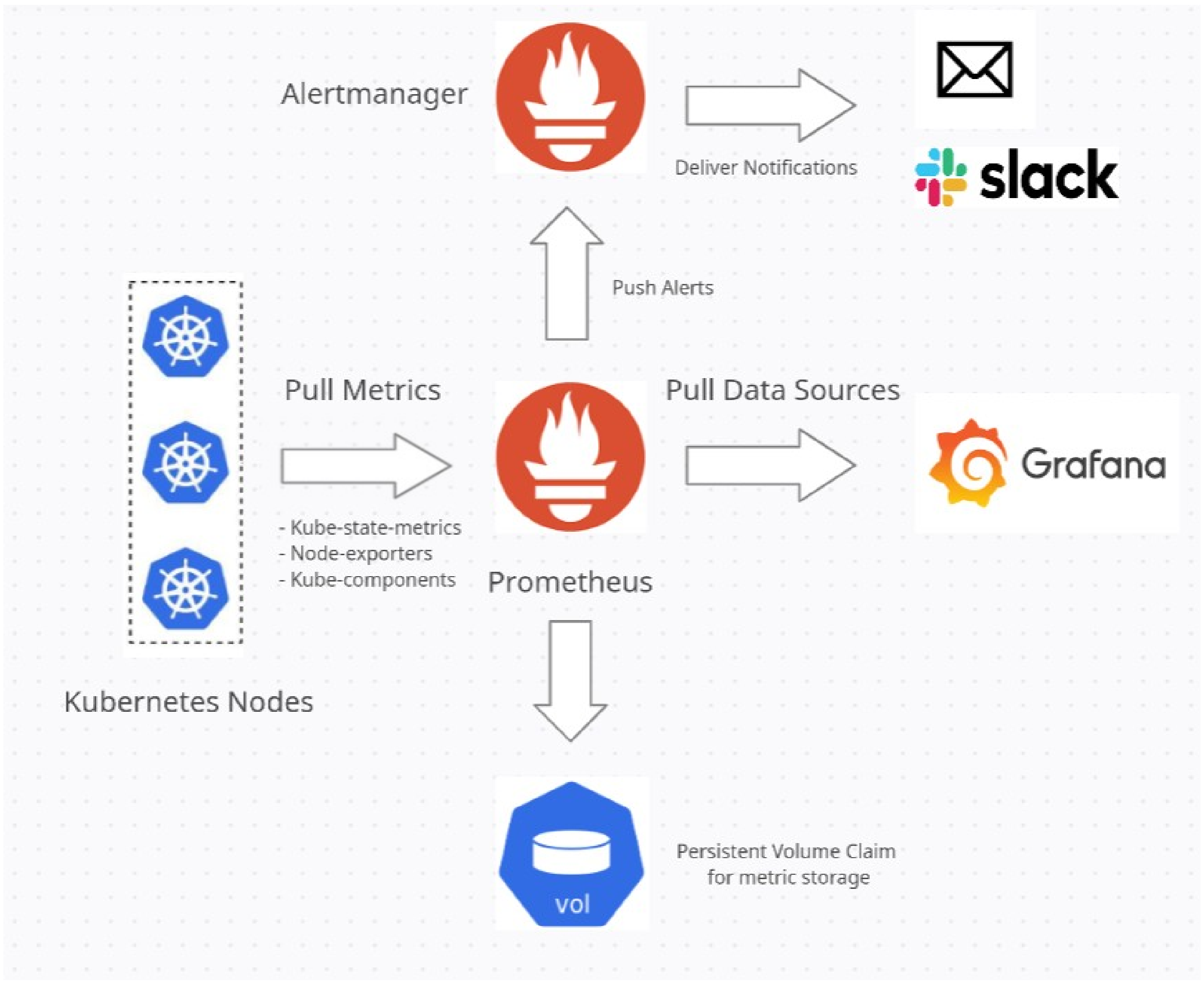
The proposed solution uses a monitoring and observability platform which may include a simple Dashboard with customisable panels, in this case: Grafana.

The metrics will be pulled and collected with Prometheus, which requires to be running on the same network as the monitored instances. In our example, target definition includes the Kubernetes nodes where the Product Service application is installed. Prometheus will be used as well to configure alerts with Alertmanager. Note that a requirement for event logging instead of metrics, may require a more suitable tool like InfluxDB.

The kube-state-metrics will provide the following information:

- Kubernetes nodes sysadmin metrics: CPU, load, memory, disk, etc.
- Orchestration level metrics: Scheduling, controller manager, deployment state, requests…
- kube-system internal components: scheduler, controller manager, dns, etc.
- Pushgateway: In case some custom scripts are required out of the box (e.g.: MQ queue depth)

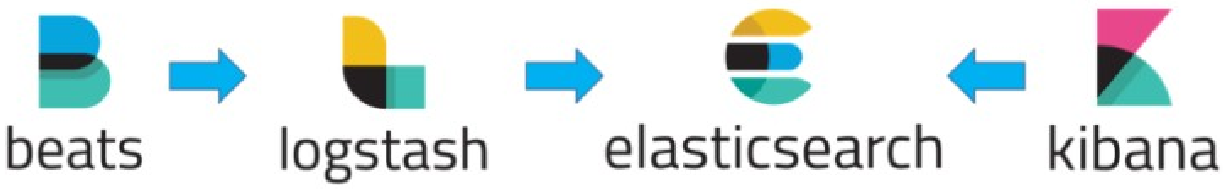
The following diagram displays the architecture:



What would you do to enable quick and easy debug of the application?

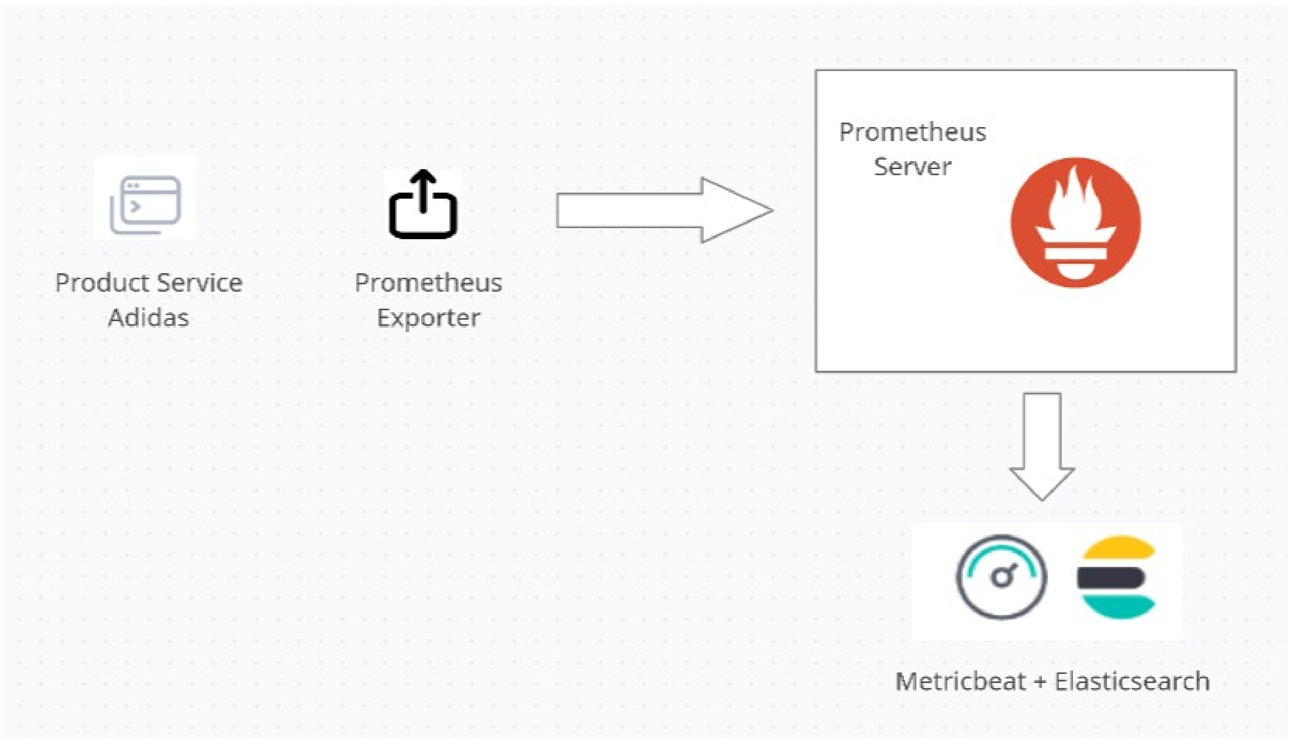
Proposed solution using the ELK stack logging capabilities.

The solution proposes:
- Beats: For data collection
- Logstash: For data aggregation and processing
- Elasticsearch: Indexing and storage
- Kibana: A dashboard to analyze and visualize data



ELK can process logs from different sources, including Kubernetes nodes, logs may include:
- Docker log and metrics
- K8 logs and metrics
- Response time & DNS metrics
- Apache logs & metrics

Elasticsearch can easily be integrated with Prometheus by connecting to it and broadcast indicators, or the metrics already collected can be pull through the Prometheus Federation endpont /metrics or API.



SLOs: Our stakeholders expect to receive product data 99.5% of the requests that they made to product-service within 200ms. Describe how would you measure and alert to ensure SLOs are accomplished.

I had to do some research here, although the initial answer that popped into my mind was to rely on Elasticsearch to aggregate the log information concerning the response time for product data requests and display the information on Kibana, as well as send alerts when needed.

One technical implementation may involve using the Elapsed Filter Plugin, to calculate the elapsed time between specific start/end tag values in the log messages, applied to the particular Product Service operation by using its unique id.

To ensure SLO, it is possible to use Kibana UI and configure a threshold alert indicating the index, time field, and the time interval to trigger the watch. As mentioned, aggregation is required by type, field, and target. We may also define conditions and actions.

Deployment: Description of a strategy for deploying the application containers in a high availability environment without downtime. How would you improve the resilience of the application?

Deploying without downtime can be achieved by using one of the following deployment models:

- Canary: Shift traffic between older version and new version in two increments spaced by a specific interval, for instance send 10 percent of traffic to the new version, and 30 minutes later it will send the rest of the traffic. These 30 minutes allow enough time to perform some tests.
- Linear: Using equal increments spaced by an equal number of minutes. For instance, shift 10 percent of traffic every 10 minutes, until 100% of the traffic gets sent to the new version of the application.
- Blue/Green: Create a new environment for the new version, and shift the traffic gradually. The advantage of this approach is that the rollback is faster in case of issues.

At AWS level it can be implemented using CodeDeploy under section: DeploymentPreference.

For the Jenkins Pipelines + Kubernetes implementation, it is required to define the strategy within the Kubernetes Deployment configuration, including the ReplicaSet to be applied during rollout of a new version. Moreover, keeping the previous ReplicaSet from the previous version around is an optimal mechanism to roll back in case of issues.

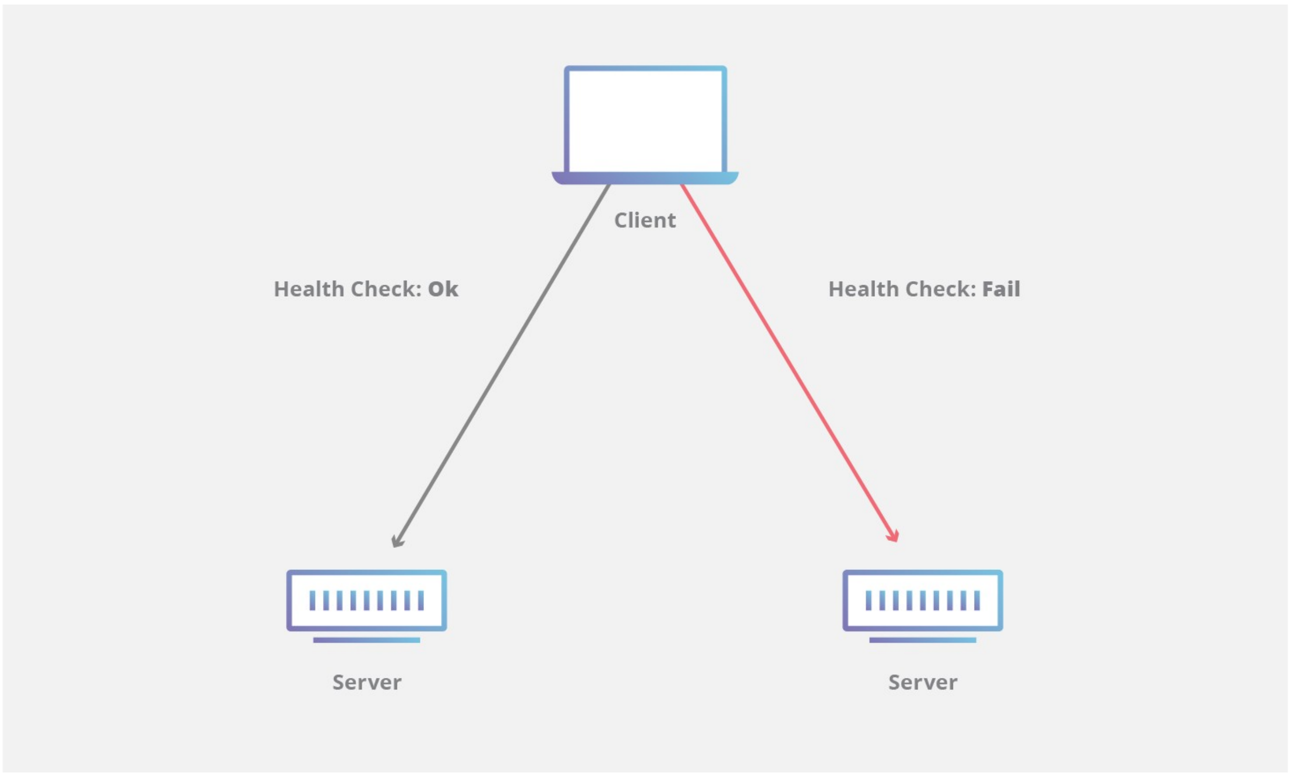
To avoid and/or fix errors after commiting new code in the repositories, the use of branches and pull requests is encouraged. If needed, it is possible to use the revert functionality in Git, or even bisect between last known code revision and the current bad revision.

Resiliency: Currently, if product reviews service is down, we won’t serve any product data at all. Describe how would you improve product-service in a way that can continue fulfilling (at least partly) the expectations in the event that product reviews service is unavailable in a region or globally.

One solution is to use a CDN (Content Delivery Network) and use its caching capabilities to improve website load time, reduce bandwith costs, increase content redundancy and availability, and improve security (e.g.: DDOS attacks).

The CDN features that minimize downtime:

- Load balancing to distribute trafffic across several servers, useful for scalability in case there is a rapid boost in traffic.
- Interruption in one server is minimized by redistributing traffic to another available server.
- Anycast routing: To switch traffic from one entire data center with issues to another one which is available.



From your solution design topics (CI/CD, Observability, SLOs, Deployment, Resiliency) we'd like you to pick just one and implement the solution proposed. You can choose your favorite one, even if it's one of the high-level overview.

Please, create a private fork repo under your own username for your implementation. You'll be able to show and explain your implementation details in the posterior challenge review session.

The chosen topic to implement is Deployment using a Jenkins Pipeline from SCM.

The deliverable consists of the Jenkinsfile which defines the different stages:

- Checkout code from GitHub
- Maven Package
- Docker Build Image
- Starting application using Compose
- Push image to Docker Hub

Plus a modification of the docker-compose.yml file so that includes:

- Zero downtime feature by using parallelism
- Rollback in case of failure

Git Repo:
https://github.com/israelpg/product-manage-challenge