1.1) In order to dockerize the New Website app, I need to build the corresponding Dockerfile for each service: frontend, backend.

For frontend and backend we are going to have a Dockerfile like this:

Docker image from a registry used as a baseline
FROM <base_docker_image>

#Configure a working directory usually /app
WORKDIR </app>

#Add all application files to working directory
ADD . /app

#Install necessary application dependencies and OS libraries
RUN <command>

For example for python libs

RUN <pip3 install -r requirements.txt>

#Build/Compile app if needed

RUN <build_command>

#run the application

CMD [“<parameter1>”, “<parameter2>”, …”<parameterN>”]

For the database needed, in a non prod environment, it could be run on a container with a volume. But for a production environment I would use Amazon RDS or Amazon Aurora service for a relational database (assuming relational database)

1.2) I’m assuming the existence of a Kubernetes cluster and considering the components frontend and backend (containers) of the NEW WEBSITE application are running on a single pod: the solution is poorly scalable.

I can define a deployment and run these microservices on separated pods, define a set of replicas and scale up and down each service (frontend and backend) depending on demand and also support failover. I also need to define the corresponding services for each frontend and backend service.

For database scalability on prod environment, I would configure a database cluster with master-slave (read replicas) approach.

1.3) In terms of security: REST API calls to Catalog service should be through HTTPS, should have authentication or token. Also needs to take into account if data retrieved from

Catalog is sensitive and should be encrypted. The Backend-Database network should be isolated. Token based communication between frontend and backend

I would handle the secrets using kubernetes secrets and then define the environment variables which contain sensitive data, using those secrets.

2.1)

- Clone: clone repository
- Build: build docker image using Dockerfile and tag the image
- Push: push the docker image to a docker registry (could be ECR)
- Deploy: deploy to Kubernetes cluster, patch the deployment with the new image

Code formatting and linting could also be included.

2.2) Considering the existence of a deployment.yml on the repository pointing to frontend/backend image on the registry:

```
stages {
    stage('Clone repository') {
        steps {
            script{
                checkout scm
            }
        }
    }
    stage('Build') {
        steps {
            script{
                //Build docker image using Dockerfile provided
                app = docker.build(“backend ”) // or "frontend”
            }
        }
    }
    stage('Push') {
        steps {
            script{
                //push docker image to registry and tag using build number and latest tags
                docker.withRegistry(<registry_url>) {
                    app.push("${env.BUILD_NUMBER}")
                    app.push("latest")
                }
            }
        }
    }
    stage('Deploy'){
        steps {
            //Deploy the image to kubernetes
```

```
            sh 'kubectl apply -f deployment.yml'
        }
    }
}
```

2.3) Testing stages I would add:

- Unit tests
- Integration tests
- E2E tests
- Code analysis

```
stages {
    stage('Unit tests') {
        steps {
            script{
                //Run unit tests depends on language/framework
                //For example:
                sh 'pytest tests/unit_tests'
            }
        }
    }
    stage('Integration tests') {
        steps {
            script{
                //Run integration tests depends on language/framework
                //For example:
                sh 'pytest tests/integration_tests'
            }
        }
    }
    stage('E2E tests') {
        steps {
            script{
                //Run Selenium tests
            }
        }
    }
    stage('Code analysis'){
        steps {
            script{
                //Run SonarQube analysis
            }
        }
    }

}
```

3.1) This will be the main.tf files for the infrastructure resource needed (Omitting provider.tf file). NOTE: this terraform file does not include uploading index.html file to the bucket or any other html or js file needed for the static web app.

Another approach is keep in separated files s3.tf cloudfront.tf, locals.tf

```
resource "aws_s3_bucket" "new_website_bucket" {
  bucket = "new-website-bucket"
}

resource "aws_s3_bucket_acl" "new_website_acl" {
  bucket = aws_s3_bucket.new_website_bucket.id
  acl    = "private"
}

resource "aws_s3_bucket_website_configuration" "new_website_config" {
  bucket = aws_s3_bucket.new_website_bucket.bucket

  index_document {
    suffix = "index.html"
  }

  error_document {
    key = "index.html"
  }

}

resource "aws_s3_bucket_policy" "new_website_policty" {
  bucket = aws_s3_bucket.new_website_bucket.id

  policy = jsonencode({
    Version = "2012-10-17"
    Id      = "AllowGetObjects"
    Statement = [
      {
        Sid       = "AllowPublic"
        Effect    = "Allow"
        Principal = "*"
        Action    = "s3:GetObject"
        Resource  = "${aws_s3_bucket.new_website_bucket.arn}/**"
      }
    ]
  })
}
```

```
locals {
  s3_origin_id = "s3-new-website.com"
}

resource "aws_cloudfront_distribution" "new_website_distribution" {

  enabled = true

  origin {
    origin_id                = local.s3_origin_id
    domain_name              =
aws_s3_bucket.new_website_bucket.bucket_regional_domain_name
    custom_origin_config {
      http_port              = 80
      https_port             = 443
      origin_protocol_policy = "http-only"
      origin_ssl_protocols   = ["TLSv1"]
    }
  }

  default_cache_behavior {
    target_origin_id = local.s3_origin_id
    allowed_methods  = ["GET", "HEAD"]
    cached_methods   = ["GET", "HEAD"]
    viewer_protocol_policy = "redirect-to-https"
  }

  restrictions {
    geo_restriction {
      restriction_type = "none"
      locations        = []
    }
  }

  viewer_certificate {
    cloudfront_default_certificate = true
  }
```

3.2) Amazon CloudFront is integrated with Amazon CloudWatch and automatically publishes some metrics with no cost like: Requests, Bytes downloaded, Bytes uploaded ,4xx error rate, 5xx error rate, Total error rate.
All of these metrics are displayed in graphs for each distribution and you can add them to Cloudwatch console by hitting “Add to dashboard”
In addition to the default metrics, you can turn on additional metrics for an additional cost like: cache hit rate, origin latency and Error rate by status code.