**Automated CI/CD Pipeline Setup on AWS with Terraform, CodeDeploy, CodeBuild, and CodePipeline**

**Introduction**

This guide provides a comprehensive walkthrough for establishing a fully automated Continuous Integration and Continuous Deployment (CI/CD) pipeline on Amazon Web Services (AWS). This solution leverages a powerful combination of AWS services and third-party tools to create a robust and efficient software release process. We will utilize Terraform for declarative infrastructure provisioning, AWS CodeDeploy for controlled application deployments, AWS CodeBuild for consistent build and test execution, and AWS CodePipeline to orchestrate the entire workflow.

The primary objective is to create a seamless, automated pipeline that initiates upon code changes in a GitHub repository, thereby minimizing manual intervention and accelerating the delivery of new features and fixes.

**Key Components:**

* **Terraform:** An industry-standard Infrastructure as Code (IaC) tool that enables the definition, provisioning, and management of cloud infrastructure using a declarative configuration language. It ensures that infrastructure is version-controlled, repeatable, and consistently deployed.
* **AWS CodeDeploy:** A fully managed deployment service that automates the deployment of applications to compute services such as Amazon EC2, AWS Fargate, AWS Lambda, and on-premises servers. It supports sophisticated deployment strategies like in-place and blue/green to minimize downtime and risk.
* **AWS CodeBuild:** A fully managed continuous integration service that compiles source code, runs unit and integration tests, and produces software packages that are ready for deployment. It operates in ephemeral, containerized environments, ensuring clean and consistent builds every time.
* **AWS CodePipeline:** A continuous delivery service that acts as a workflow orchestrator. It models, visualizes, and automates the steps required to release software, seamlessly integrating with source control, build, and deployment services.

By following this guide, you will establish a scalable and efficient deployment process, enabling true continuous deployment where every validated change pushed to your repository is automatically deployed to your production or staging environment.

**Step 1: Provision EC2 Instance Using Terraform**

**Objective:** Automate the provisioning of an EC2 instance with Terraform, ensuring the AWS CodeDeploy agent is installed and configured during the bootstrap process.

**Detailed Steps:**

1. **Install and Configure Prerequisites:**
   * Install the latest version of the Terraform CLI from the official HashiCorp website.
   * Install and configure the AWS Command Line Interface (CLI) with the necessary credentials and default region to allow Terraform to authenticate with your AWS account.
2. **Develop Terraform Configuration:**
   * Create a dedicated directory to house your Terraform configuration files.
   * Within this directory, create a main.tf file. This file will serve as the primary configuration for defining your AWS resources.
3. **Define AWS Provider:**
   * Inside main.tf, declare the AWS provider block. This block informs Terraform that you intend to manage AWS resources and specifies the target AWS region for provisioning.
4. **Define the EC2 Instance Resource:**
   * Utilize the aws\_instance resource block within main.tf to define the specifications of your virtual server.
   * Key properties to configure include the Amazon Machine Image (AMI) ID, the instance type (e.g., t2.micro), and the name of the EC2 key pair for SSH access.
   * Crucially, provide a user\_data script. This bootstrap script executes upon the first launch of the instance. The script should be designed to perform initial configuration tasks, such as updating system packages and, most importantly, downloading, installing, and starting the CodeDeploy agent service.
5. **Define the Security Group Resource:**
   * Define an aws\_security\_group resource in main.tf. This acts as a virtual firewall for your EC2 instance.
   * Configure ingress (inbound) rules to allow traffic on essential ports, such as port 22 for SSH access (ideally restricted to your IP address) and port 80 for standard HTTP traffic.
6. **Provision Infrastructure:**
   * Navigate to your Terraform directory in your terminal and run terraform init. This command initializes the backend and downloads the required provider plugins.
   * Run terraform apply. Terraform will present an execution plan and, upon confirmation, will provision the defined EC2 instance and security group in your AWS account.
7. **Verify EC2 Instance and Agent Status:**
   * Once the apply command completes, navigate to the EC2 console in AWS to confirm that the instance is running.
   * Connect to the instance via SSH and execute sudo systemctl status codedeploy-agent to verify that the CodeDeploy agent was successfully installed and is currently in an active (running) state.

**Step 2: Set Up CodeDeploy**

**Objective:** Configure a CodeDeploy Application and Deployment Group to manage and automate deployments to the provisioned EC2 instance.

**Detailed Steps:**

1. **Create a CodeDeploy Application:**
   * In the AWS Management Console, navigate to the CodeDeploy service.
   * Create a new application, which serves as a container for your deployment groups and revisions. Select **EC2/On-Premises** as the compute platform.
2. **Create a Deployment Group:**
   * Within the application, create a new deployment group. This group defines a set of instances where the application will be deployed.
   * Configure the deployment group by specifying the deployment type (e.g., **In-place** to update the current instances, or **Blue/Green** for a more advanced immutable deployment).
   * Use Amazon EC2 tags to associate the deployment group with your provisioned instances. This is a scalable approach that allows new instances with the same tag to be automatically included in future deployments.
   * Assign an IAM service role that grants CodeDeploy the necessary permissions to interact with your EC2 resources.
3. **Define Application Specification:**
   * Ensure your application source code repository includes an appspec.yml file in the root directory. This critical file provides deployment instructions to the CodeDeploy agent, defining the source files to be copied and the lifecycle hook scripts to be executed at various stages of the deployment (e.g., BeforeInstall, AfterInstall, ApplicationStart).
4. **Verify CodeDeploy Configuration:**
   * Confirm that the CodeDeploy agent on the target EC2 instance is running and can communicate with the CodeDeploy service.
   * You can optionally test the setup by manually creating a deployment revision and triggering it from the console to ensure the connection and permissions are correctly configured.

**Step 3: Set Up CodeBuild for Building the Application**

**Objective:** Configure an AWS CodeBuild project to automatically compile source code, run tests, and package the application for deployment.

**Detailed Steps:**

1. **Create a CodeBuild Project:**
   * Navigate to the AWS CodeBuild console and create a new build project.
   * Configure the **Source** section to connect to your GitHub repository and specify the branch to monitor. Authenticate CodeBuild with your GitHub account using OAuth or a personal access token.
   * Configure the **Environment** by selecting a managed image with the appropriate operating system (e.g., Amazon Linux 2) and runtime environment (e.g., Node.js, Java) that aligns with your application's technology stack.
   * Specify the location of the buildspec.yml file within your repository, which is typically at the root.
2. **Create the buildspec.yml File:**
   * In the root of your source repository, create a buildspec.yml file. This YAML file defines the commands CodeBuild will execute during the build process.
   * Structure the file using the standard build phases:
     + **Install:** Define commands for installing dependencies and runtime versions.
     + **Pre-build:** Include commands for setup tasks like logging into a package registry or running linters.
     + **Build:** Specify the core commands for compiling your code, running tests, and packaging the application.
     + **Post-build:** Define commands to be run after the build, such as creating the final build artifact.
   * The primary output of the build or post\_build phase is a build artifact (e.g., a ZIP archive), which CodeBuild will upload to an S3 bucket for CodePipeline to use in the deployment stage.
3. **Test the Build Process:**
   * Manually trigger a build from the CodeBuild console to validate the buildspec.yml configuration. Review the build logs to ensure all phases execute successfully and the artifact is produced as expected.

**Step 4: Set Up CodePipeline**

**Objective:** Automate the end-to-end CI/CD workflow by creating a CodePipeline that orchestrates the source, build, and deploy stages.

**Detailed Steps:**

1. **Create a New Pipeline:**
   * Navigate to the AWS CodePipeline console and initiate the creation of a new pipeline.
   * Provide a unique name for the pipeline and select or create an IAM service role that grants CodePipeline permissions to interact with other AWS services.
2. **Configure the Source Stage:**
   * In the **Source** stage configuration, select GitHub as the source provider.
   * Connect to your GitHub account, then choose the repository and branch that will trigger the pipeline.
   * Select the **GitHub webhooks** detection option. This is the recommended, event-driven method that allows GitHub to automatically notify CodePipeline of new commits, triggering the pipeline almost instantly.
3. **Configure the Build Stage:**
   * Add a **Build** stage to the pipeline.
   * Select AWS CodeBuild as the build provider and choose the project you created in the previous step. CodePipeline will automatically pass the source code artifact from the source stage to CodeBuild.
4. **Configure the Deploy Stage:**
   * Add a **Deploy** stage following the build stage.
   * Select AWS CodeDeploy as the deployment provider.
   * Choose the CodeDeploy application and deployment group created earlier. CodePipeline will pass the build artifact produced by CodeBuild to CodeDeploy for deployment.
5. **Review and Test the Pipeline:**
   * Review the complete pipeline structure and create it.
   * To perform an end-to-end test, push a code change to the configured branch in your GitHub repository.
   * Monitor the pipeline's progress in the CodePipeline console, verifying that it automatically triggers and successfully completes the source, build, and deploy stages.

**Step 5: IAM Roles and Permissions**

Correctly configured IAM roles are fundamental to the security and functionality of the CI/CD pipeline. Each service requires a specific role with the minimum necessary permissions to perform its tasks.

1. **IAM Role for CodePipeline (Service Role):**
   * **Purpose:** Allows CodePipeline to orchestrate actions in other services. It needs permissions to read from S3 (for artifacts), start CodeBuild projects, and initiate CodeDeploy deployments.
   * **Configuration:** When creating the pipeline, you will be prompted to create or select a service role. This role should have policies attached that grant access to CodeBuild, CodeDeploy, and S3.
2. **IAM Role for CodeBuild (Service Role):**
   * **Purpose:** Grants the CodeBuild environment permissions to perform its tasks. This includes permissions to fetch source code, write build logs to CloudWatch Logs, and upload build artifacts to an S3 bucket.
   * **Configuration:** This role is assigned to the CodeBuild project during its creation. It's trusted by the codebuild.amazonaws.com service principal.
3. **IAM Role for EC2 Instance (Instance Profile):**
   * **Purpose:** Allows the EC2 instance to communicate with other AWS services. For this pipeline, it primarily enables the CodeDeploy agent on the instance to poll the CodeDeploy service for new deployments and download deployment artifacts from S3.
   * **Configuration:** This role is created with EC2 as the trusted entity and is attached to the EC2 instance as an "Instance Profile" during its launch via Terraform or afterwards in the EC2 console.
4. **IAM Role for CodeDeploy (Service Role):**
   * **Purpose:** Grants the AWS CodeDeploy service permission to perform actions on your behalf, such as reading EC2 instance tags and managing deployments.
   * **Configuration:** This role is assigned to the CodeDeploy deployment group during its creation. It's trusted by the codedeploy.amazonaws.com service principal and typically includes policies like AWSCodeDeployRole.

**Step 6: Enable Continuous Deployment with GitHub Integration**

**Objective:** Finalize the setup to ensure that any code change pushed to the designated GitHub branch automatically triggers a full build and deployment cycle.

**Detailed Steps:**

1. **Verify GitHub Webhook:**
   * The webhook integration configured in the CodePipeline source stage is the key to full automation.
   * To confirm it is active, navigate to your GitHub repository's settings, go to the "Webhooks" section, and ensure a webhook pointing to AWS CodePipeline exists and is active (indicated by a green checkmark).
2. **Trigger and Monitor the Pipeline:**
   * Make a minor, non-breaking code change in your local repository (e.g., update a text file or a comment).
   * Commit and push this change to the specific branch monitored by CodePipeline.
   * Observe the CodePipeline console. Within moments, the pipeline should automatically transition to an "In Progress" state, indicating that the webhook successfully triggered it.
3. **Monitor End-to-End Execution:**
   * Follow the pipeline's execution through each stage in the console. You can click on the details of each stage to view logs from CodeBuild or track the deployment progress in CodeDeploy.
   * Upon successful completion, verify that your application changes are live on the EC2 instance.

**Final Summary: Full Automation of the Deployment Pipeline**

By completing these steps, you have successfully engineered a robust, end-to-end CI/CD pipeline on AWS. This automated system leverages:

* **Terraform** for repeatable and version-controlled infrastructure provisioning.
* **CodeDeploy** for reliable and controlled application deployments.
* **CodeBuild** for consistent and isolated application builds and tests.
* **CodePipeline** to orchestrate the entire workflow from code commit to deployment.