GitLab

初心者向けガイド

GitOps

- GitOpsのご紹介
- インフラストラクチャオートメーションのメリット
- GitOpsのベストプラクティス

目次

はじめに

ソフトウェアアプリケーションが洗練されるにつれ、インフラストラクチャに対するニーズも増しています。インフラストラクチャチームは複雑なデプロイを素早く大規模にサポートしなければなりません。多くのアプリケーション開発が自動化される中、インフラストラクチャの大部分は専門チームが必要な手動プロセスで占められていました。手動プロセス以外に再現性のある、安定したソフトウェア実行環境を設計、変更、デプロイする方法はあるのでしょうか。AnsibleやTerraformのようなIaC (Infrastructure-as-code) ツールも取り掛かりとして有用ですが、全体としての問題を解決するわけではありません。チームはIaCを自動的に実行できる規範的なワークフローを作成する必要があります。

この電子書籍では、GitOpsのインフラストラクチャオートメーションプロセスについてご紹介し、そのプロセスがインフラストラクチャの設計や変更、展開向けのエンドツーエンドソリューションとなる仕組みをご説明します。この電子書籍では、以下についてご理解いただけます。

- ☐ すでに利用しているプロセスとGitOpsがアプリケーション開発で連携する仕組み
- ☐ GitOpsの利用を開始する上で必要な3つのコンポーネント
- ☐ GitOpsのベストプラクティスとワークフロー

成熟したDevOpsカルチャーを持つ組織は、1日に数百回の頻度でコードを本番環境にデプロイできます。ソフトウェア開発ライフサイクルが自動化されても、インフラストラクチャのロールアウトの大部分は、依然としてマニュアルプロセスのままです。デプロイ頻度の向上にITチームの作業スピードが追い付かないという問題は、以前から存在していました。

物理的なハードウェアが必須ツールとされていた時代において、インフラストラクチャの自動化という概念は机上の空論でした。しかし、仮想化技術のおかげで、そうしたハードルが少しだけ下がりました。
その発端となったのが、パブリッククラウドの登場です。これにより、大規模なインフラストラクチャを比較的簡単に完全自動化できるようになりました。従来のサーバーや仮想マシン (VM) と違い、クラウドにはハードウェアが必要なく、VMやオペレーティングシステム (OS) をプロビジョニングすることなく、クラウドネイティブサービスを個別に作成・管理できます。

PowerShellやBashなどのスクリプト言語を使用することで、ITチームはさまざまなサービスをクラウドにデプロイできます。サーバーレスサービスなど、オートスケーリング機能を含んだクラウドサービスが多くなっています。オートスケーリング機能がないサービスの場合、即座にインスタンスをデプロイできることが重要になります。

こういったサービスを利用できるからといって、チームが効率よくサービスを利用できるとは限りません。AWSだけでも200以上のサービスを抱えており、多くの企業がその中の多数のサービスを使用しています。こういったサービスには多数の設定がつきものです。AWSポータルを使用してすべてのサービスを手動でデプロイするには時間がかかり、エラーを誘発するため、大規模な組織にとっては現実的でありません。

GitOpsを利用すると、バージョン管理やコードレビュー、CI/CDパイプラインなど、多くの企業がすでに利用しているDevOpsのベストプラクティスを利用して、インフラストラクチャを自動化・管理できます。インフラストラクチャをコードとして記述することにより、同じサービスを何度もデプロイできます。また、パラメータ化により、同じサービスを別の名前や設定を別の環境で使用してデプロイすることもできます。

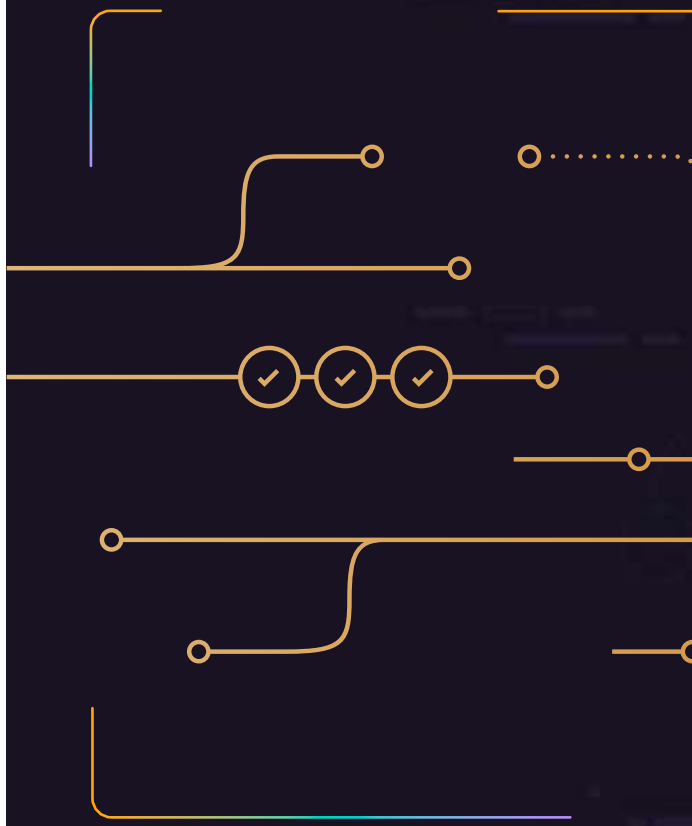
GitOps利用開始への道のり

AWSが2006年に公開される以前から、オンプレミス型インフラストラクチャの管理はITチームにとって困難な作業になりがちでした。複数のアプリケーションやサービスがさまざまなサーバーで実行され、拡張にはITチームによるサーバー全体への手動設定と同じ設定を用いた同じアプリケーションの再インストールが必要でした。幸いなことに、今ではこうしたタスクを簡易化するツールが開発されています。

Puppetや**Chef**など、初期の構成管理 (CM) ツールは既存のサーバーの設定を簡単にしてくれました。ITチームはサーバーやVMを起動し、PuppetまたはChefエージェントをインストールすれば、サーバー上でアプリケーションが動作する為に必要な設定をこれらのツールが実行してくれます。こういったツールはクラウドサーバーだけでなく、オンプレミス型サーバーでも実行できました。

初期の構成管理ツールは新しい本番サーバーを設定する手順すべてを複製する上で効率のよい方法でした。今ではこうした手順が自動化され、新しいサーバーを設定する際の手間も大幅に解消されました。しかし、新しいVMのプロビジョニングには依然として対応しておらず、クラウドネイティブインフラストラクチャでは正常に機能しませんでした。

ここで**Ansible**や**SaltStack**といった第2世代の構成管理ツールが登場します。こういったツールは初期の構成管理ツールと同様にソフトウェアを個別のサーバーにインストールできますが、設定前にVMをプロビジョニングすることもできます。たとえば、10件のEC2インスタンスを作成した上で、必要なすべてのソフトウェアを各インスタンスにインストールできます。



このような構成管理ツールの重大な欠点のひとつは、サーバーとVMをプロビジョニングし設定するだけであるという点です。クラウドネイティブサービスに対するソリューションとしては機能していません。

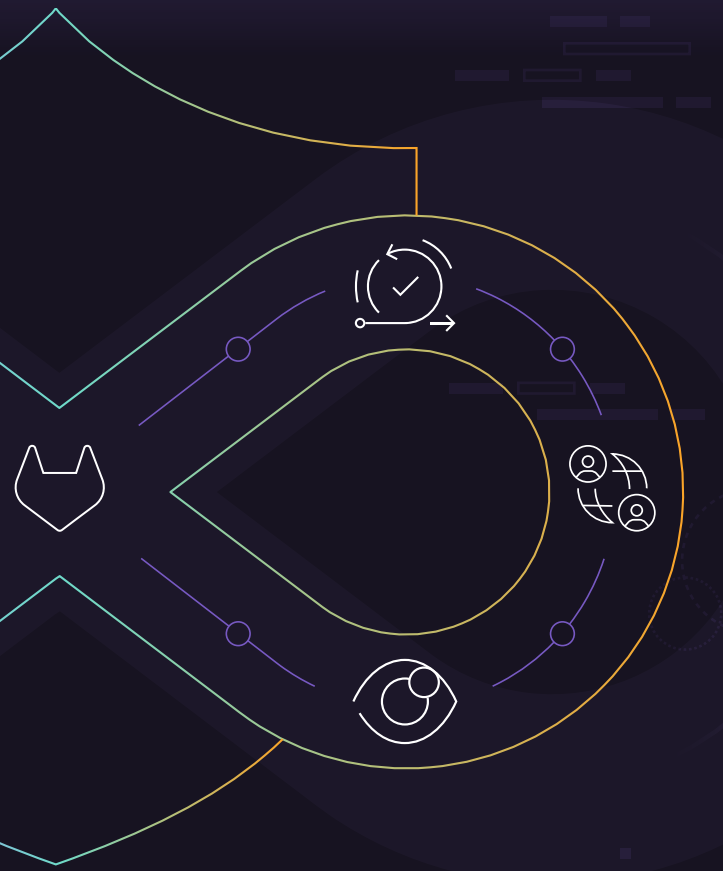
Amazon CloudFormationは第2世代の構成管理ツールと同時期に登場しました。サーバーの設定には対応していませんが、宣言型コードを使用してAWSアプリケーションアーキテクチャ全体をプロビジョニングすることができます。これにより、マネジメントコンソールをクリックしながらリソースを手動で作成する手間がなくなりました。インフラストラクチャをシンプルにJSONまたはYAMLで記述し、**AWS**マネジメントコンソール、コマンドラインインターフェイス (CLI)、またはAWS SDKを使用してデプロイできます。ただし、Amazonのサービスであるため、対応しているのはAWSのみです。

Microsoft AzureもインフラストラクチャをJSONテンプレートで記述できる同様のツール、Azure Resource Manager (ARM) を用意しています。ただし、Amazon CloudFormationとAWSの関係と同様に、ARMが対応するのはAzureサービスのみです。

プライベートクラウドやAzure、**Google Cloud**などのパブリッククラウドが普及し始めると、多くの企業が他のクラウドに切り替えたり、マルチクラウドに移行したりして、単一のクラウドプラットフォームへの依存を回避しようとしました。この新たな要件を満たすため、**Terraform**などのマルチクラウドに対応した構成管理ツールが登場しました。サービスをシンプルに記述して複数のクラウド、プロバイダー、およびクラウドサービスにデプロイできるようになりました。

こういったツールの長所は、インフラストラクチャコードでのバージョン管理やコードレビュー、継続的インテグレーション/継続的デリバリー (CI/CD) を実行できることです。

GitOpsの仕組み



GitOpsは、実証済みのDevOpsプロセスをインフラストラクチャコードに適用します。
その名が示すとおり、Gitとオペレーション (Operation)、またはリソース管理を融合したものです。Gitはオープンソースバージョン管理システムであり、コード管理における変更を追跡します。DevOpsと同様、GitOpsを使用する目的はCI/CDを活用してリソースを自動的にデプロイすることであり、これにはGitリポジトリに保存されているコードを使用します。

GitOpsを使用すると、JSONまたはYAMLで定義され、プロジェクトの.gitフォルダに保存されたインフラストラクチャを定義したコードが信頼できる唯一の情報源 (Single Source of Truth) として機能するGitリポジトリに存在するようになります。Gitの機能を使用することで、組織のインフラストラクチャコードの変更履歴をすべて確認することができ、チームは必要に応じて前のバージョンへロールバックできます。

また、Gitを使用することで、インフラストラクチャ上でのコードレビューも実現できます。コードレビューは、不正なアプリケーションコードを本番環境にリリースしないために重要なDevOpsプラクティスです。これはインフラストラクチャコードにとっても重要です。不正なインフラストラクチャコードは高額なクラウドインフラストラクチャを誤った状態で起動してしまい、1時間あたり数千ドルのコストを生み出す可能性があります。また、不正なスクリプトによりアプリケーションが停止し、サービスの停止につながる場合もあります。コードレビューにより承認前に複数の人の目でチェックでき、このような間違いを防ぐことができます。

GitとIaCを利用する上で最も大きなメリットは、継続的インテグレーションと継続的デプロイが可能になることです。GitLab CI/CDのようなツールを使用することで、インフラストラクチャのデプロイ（更新）を自動化でき、クラウド環境に自動で適用できます。インフラストラクチャコードに追加されたリソースは自動的にプロビジョニングされ、利用可能になります。クラウド環境で変更・更新されたリソースやインフラストラクチャコードから削除されたリソースは自動的に停止・削除されます。これにより、コードを書き、Gitリポジトリにデプロイし、DevOpsプロセスのメリットをインフラストラクチャで最大限に利用することができます。

GitOps = IaC + MR + CI/CD

- **IaC** – GitOpsはGitリポジトリをインフラストラクチャを定義したコードの保管場所として信頼できる唯一の情報源 (Single Source of Truth) として使用します。Gitリポジトリは、プロジェクト内のファイルに対するこれまでの変更すべてを記録する、プロジェクト内の.gitフォルダです。 Infrastructure as code (IaC) とは、コードとして保存されたインフラストラクチャの設定すべてを維持するプラクティスです。実際に期待する状態（レプリカの数、ポッドなど）はコードとして保存される場合とそうでない場合があります。

- **MR** – GitOps**はすべてのインフラストラクチャ更新の変更メカニズムとして**マージリクエスト (MR) を使用します。MRとはチームのメンバーがレビューやコメントを通して共同作業できる場であり、正式な承認を得る場でもあります。マージはマスター（またはトランク）ブランチに送られ、監査やトラブルシューティングで使用できる変更履歴を提供します。

- **CI/CD** – GitOpsは継続的インテグレーションと継続的デリバリー (CI/CD) を使用したGitワークフローによりインフラストラクチャの更新を自動化します。新しいコードがマージされると、CI/CDパイプラインは変更を実行に移します。手動での変更やエラーなど、構成ドリフト (サーバーの構成・設定が時間とともにバラバラになる) はGitOpsにより上書きされるため、環境はGitで定義されている期待通りの状態に落ち着きます。GitLabはCI/CDパイプラインを使用してGitOpsを管理・実施しますが、定義演算子などのオートメーション形式を使用することもできます。

GitOpsのメリット

GitOpsのフレームワークによりインフラストラクチャの自動化が可能になりますが、GitOpsのメリットはそれだけではありません。GitOpsを採用する組織では、以下のようなメリットにより長期にわたる効果が得られます。

インフラストラクチャの変更に関する共同作業。インフラストラクチャの変更における共同作業。どの変更も同じ変更、マージリクエスト、レビュー、承認といったプロセスを経るため、シニアエンジニアは最低限のインフラストラクチャ管理に従事する一方で、他の分野にも注力できます。

市場投入までの時間を短縮。コードを使用して実行するため、手動でカーソルを合わせてクリックするよりも速く作業を進められます。テストケースは自動で繰り返し実行できるので、安定した環境を迅速に提供できます。

監査の簡易化。インフラストラクチャの変更を複数のインターフェイスで手動で行うと、監査が複雑になり、時間がかかる場合があります。監査を実行するために、さまざまな場所からデータを取得し、標準化する必要も出てきます。GitOpsを使用すると、環境に加えられたすべての変更がgitログに保存されるため、監査がシンプルになります。

リスクの低減。インフラストラクチャに対する変更はマージリクエストを通してすべて記録され、変更前の状態にロールバックすることもできます。

エラーリスクの低下。インフラストラクチャの定義がコード化されており、再現性があるためヒューマンエラーが起こりにくくなります。また、コードレビューやマージリクエストでの共同作業により、エラーを特定でき、運用環境にリリースする前に修正できます。

コストとダウンタイムの削減。インフラストラクチャ定義とテストの自動化により、手動タスクが減り、生産性が上がるだけでなく、組み込みのロールバック機能によりダウンタイムを削減できます。また、自動化により、インフラストラクチャチームにとってクラウドリソースの管理が容易になり、クラウドにかかるコストを節約できます。

アクセス権限管理の改善。変更が自動で行われるため、すべてのインフラストラクチャコンポーネントに対して認証情報を提供する必要はありません（アクセス権限が必要なのはCI/CDのみ）。

共同作業時のコンプライアンス遵守。厳重に規制されたコンテキストポリシーでは、多くの場合、本番環境への変更を行える人数をできるだけ少なくするよう規定しています。
GitOpsを利用すれば、マージリクエストにより、ほとんどのユーザーが変更を提案でき、本番ブランチにマージを行える人数を制限してコンプライアンスを守りつつ、共同作業の範囲を大幅に拡大できます。

一般的なGitOpsツール

単一の製品やプラグイン、プラットフォームではないところがGitOpsの特長です。GitOpsはすでにアプリケーション開発で使用しているプロセスを通してチームによるITインフラストラクチャの管理を支援するフレームワークです。Ansible、Terraform、Kubernetesなどが人気ですが、GitOpsのプロセスの多くはテクノロジー（Gitを除く）に依存しません。

GitOpsはさまざまなシナリオに適しており、Kubernetesとの相性が抜群です。Kubernetesは主要なクラウドプラットフォームすべてと連携し、ステートレスで変化しないコンテナを使用します。Kubernetesで実行されるコンテナ化されたアプリは自己完結型であるため、各アプリごとにプロビジョニングやサーバーの設定を行う必要はありません。Kubernetesクラスタやデータベース、ネットワークなどの必要なインフラストラクチャに対するプロビジョニングにはTerraformを使用します。

ステートフルなアプリケーションのデプロイにおいては、Amazon Auroraデータベースインスタンスや Redisキャッシュなどの外部サービスにデータを永続させることを考慮する必要があります。KubernetesはGitOpsのフレームワークにおいて役立ちますが、GitOpsの実行に不可欠なものではなく、VMなど、従来のクラウドインフラストラクチャでも利用できます。この場合、Terraformでプロビジョニングを行い、Ansibleのような構成管理ツールを使用して新しいVMを構成します。

Gitコードリポジトリ	git		
Git管理ツール	GitLab	GitHub	Bitbucket
継続的イングレーションツール	GitLab	Jenkins	circleci
継続的デリバリー	GitLab	Spinnaker	flux
コンテナレジストリ	GitLab	docker hub	
設定マネージャー	ANSIBLE	CHEF	puppet
インフラストラクチャのプロビジョニング	Terraform	pulumi	AWS CloudFormation
コンテナオーケストレーション	kubernetes		

GitOpsの利用開始におけるベストプラクティス

インフラストラクチャに小さな変更を手動で加えることに慣れたチームにとって、GitOpsのようなプロセスを採用することには大きな変化が伴う場合があります。GitOpsはインフラストラクチャチームに古い習慣と新しい習慣を置き換えるよう要求するフレームワークです。チームによっては時間がかかったり、ぎこちなかったりする場合もあるでしょう。いつでも確認できるベストプラクティスを用意しておくと、GitOpsの長期戦略に取り組めるようになります。

すべてのインフラストラクチャを構成ファイルとして定義する

まず、GitOpsを通して管理するインフラストラクチャすべてがIaC設定ファイルに記述されていることを確認します。宣言型コードでファイルを記述することが望ましいでしょう。これにより、何らかの状態に到達するまでの道筋ではなく、期待する最終状態を示します。

たとえば、プロバイダーにサービスを作成する方法を説明するJavaScriptファイルではなく、サービスの設定方法を記述したプロパティを持つJSONファイルを使用します。宣言型構文に対応しているクラウドプロバイダーやCMツールは多数ありますが、宣言型の使用を念頭に設計されたツールを選ぶのが一番よいでしょう。

効果を最大限に引き出すには、**すべての**インフラストラクチャをコードで記述します。デフォルト設定のみを使用し、わずか1分でセットアップが完了するようなサービスはコード化の対象外にしたくなりますが、新しい環境を起動する際に手動の操作は忘れがちになります。このサービスをマニュアルでデプロイしなければならないことに気づかない可能性もあります。このような省略を許してしまうと、小さな技術的負債が重なり、GitOps戦略を妨害することになりかねません。

すでに使用しているIaCをGitOpsを使用して自動化する場合はまず、インフラストラクチャコードをGitOpsで利用するGitリポジトリに追加します。

まだIaCを使用していない場合、設定コードを使用した既存のインフラストラクチャの定義には手間がかかります。AWSを利用すると、CloudFormation設定ファイルを既存のリソースから簡単に作成できます。Terraformではさまざまなプロバイダーから既存のインフラストラクチャをインポートできますが、すべての作業を代行してくれるわけではありません。

目的は、すべてのインフラストラクチャをコードとして記述することですが、これには時間がかかります。既存のインフラストラクチャすべてを一度に自動化しなければならないわけではありません。
少しずつ自動化していきましょう。何度も時間をかけて行えば、GitOpsのワークフローに移行するインフラストラクチャの数も次第に増えていきます。

自動化できないものを文書化

常にすべてのプロセスを自動化できるわけではありません。たとえば、Azureでは一部（通常最新）の設定がまだARMテンプレートに追加されていない場合があります。こういった場合は通常、PowerShellを利用します。

また、サードパーティプロバイダとの協業も自動化できない場合があります。手動でIPアドレスの承認リストを要求しなければならないサプライヤーと協業するとしましょう。承認リストを要求できるのはマネージャーのみです。新しいサービスや環境で行う手動操作にはIPアドレスの検索やマネージャーへの伝聞、サプライヤーへのメール送信があります。こうしたプロセスを分かりやすく、確実に文書化しましょう。

どんな場合でも、手動操作を必要とするレガシー環境はあります。こういった状況にも対応できるよう、分かりやすく文書化しましょう。

コードレビューとマージリクエストプロセスの概要

GitOpsチームにGitやコードレビューについて熟知してもらうことが重要です。一部のチームは設定コードの保存においてすでにGitリポジトリを使用していますが、マージリクエストのような機能はまだ使用していません。始めるにあたって、GitLabオープンソースプロジェクトに関するコードレビューのガイドラインをご覧ください。このプロジェクトでは、コードレビューのガイドラインに追加していく情報の種類について理解することができます。

マージリクエストを承認する前に、最低限の人数のレビュワーを設定し、すべてのコードをチーム内の複数のメンバーがレビューできるようにしましょう。

GitOpsを初めて利用するチームは「必須のブロックレビュー」ではなく「任意レビュー」を設定することもできます。これは新しいプロセスであるため、時間をかけてコードレビューの実行に慣れ、丁度よい頻度を見出しましょう。チームがツールセットやプラクティスを習得したら、必須のレビューを実装し、コードレビューが毎回確実に行われるようにします。

繰り返しになりますが、小さな規模から、シンプルに始めましょう。複雑なガイドラインから始めても、プロセスを採用してもらえません。最高の機能よりも、まずは採用してもらえることを優先しましょう。時間をかけてコードレビューを反復し、堅牢で完全なものに仕上げていけます。

複数の環境を検討する

複数の環境を用意しておくとよいでしょう。たとえば、Development（開発）、Test（テスト）、Acceptance（ユーザー受入）、Production（本番）から成るDTAP環境。開発環境またはテスト環境でコードをロールアウトし、サービスの可用性と正常な動作をテストできます。正常な動作を確認したら、変更を次の環境へロールアウトできます。

コードをお使いの環境へロールアウトしたら、コードと実行しているサービスの同期を維持することが大切です。
システムと設定に違いがあることが分かっているなら、どちらかを修正することも可能です。この問題を解決するには、違いが生じにくい、コンテナなどのイミュータブルなイメージを使用すると良いでしょう。

Chef、**Puppet**、**Ansible**などのツールは、サービスが設定コードと異なる場合に通知してくれる「差分アラート」などの機能を備えています。**Kubediff**や**Terradiff**などのツールを使用することで、同じ機能をKubernetesやTerraformでも利用できます。

CI/CDをリソースへのアクセスポイントにする

GitOpsのワークフローを促進し、クラウドインフラストラクチャへの手動による変更を減らすプラクティスのひとつに、CI/CDツールをクラウドリソースのアクセスポイントにすることが挙げられます。

開発初期の段階でアクセス権限を持っていれば、もちろんチームによるコードの記述に役立ちますし、さまざまな理由により付随的なアクセス権限が必要になる場合もあります。ただし、「〜でなければアクセスする」から「〜だからアクセスする」にマインドを切り替えることが、GitOpsプロセスを採用し、それに従う上で役立ちます。

リポジトリ戦略を立案する

リポジトリをどう設定するかを考えてみましょう。まずは、すべてのインフラストラクチャに対応する単一のリポジトリを使用する場合、複数の共有リソースに単一のリポジトリを使用することは理に適っていますが、サービスに特化したリソースのコードを対象のサービスに保存する必要があります。別のアプローチとしてさまざまなプロジェクトのリソースを別々のリポジトリに保存することも考えられます。これらは組織の構造やサービス、個人の好みによって変わります。

リポジトリを単一にするか複数にするか、戦略を決める際は以下を検討してください。

- [ ] すべてのコードにアクセス権限を持たせることで安全性にリスクが生じる請負業者や個人がプロジェクトに参加することが頻繁にありますか？
- [ ] 考慮すべき依存関係が複数ありますか？
- [ ] リポジトリを信頼できる唯一の情報源 (Single Source of Truth) として使用しますか？

世界最大手のテック企業Googleでは、 すべてのコードに対して単一のリポジトリを使用しています。HashiCorpでは、Terraformコードを含むリポジトリそれぞれをアプリケーション、サービス、その他特定のインフラストラクチャ（一般的なネットワークインフラストラクチャなど）といった管理可能なインフラストラクチャのチャンクにすることを推奨しています。もちろん「管理可能」の定義は会社によって異なります。

複数の人々が同じリポジトリで同時に作業できる機能ブランチ（フィーチャーブランチ）などのGitブランチ戦略も検討してみましょう。

変更を小さく留める

どんな作業であれ、小さな手間で済むように心がけましょう。そうすれば、一つひとつの変更をロールバックするのが簡単な、きめ細やかな変更ログを作れます。GitLabでは、このプロセスをイテレーション（反復）と呼んでおり、素早く変更を行うことができる理由になっています。小さなステップを踏むことで、機能を小さく、シンプルに提供でき、フィードバックを受け取るまでの時間を短縮できます。

GitOpsおよびTerraformを備えたCI/CDパイプライン

Terraformの**validate**コマンドやJSONファイルのLinterを使用することで、最初にインフラストラクチャコードを検証するようにCI/CDを設定しましょう。インフラストラクチャコードはクリーンかつ整合性のとれたプロダクションコードのように処理する必要があります。

誰かが無効なコードを送信すると、ビルドや検証がうまくいかず、直ちにチームに通知が届くように設定しましょう。これにより、チームで送信内容を修正したり、ロールバックしたりして、問題を素早く解決できます。小さな変更を加える場合でも、問題が見つかりやすくなります。

コードが有効である場合、CI/CDは設定コードで定義されたインフラストラクチャのプロビジョニングに必要なコマンドを実行します。たとえば、Terraformの**apply**コマンドやCloudFormationの**AWS update-stack**がこれに該当します。

Terraform、CI/CD、Kubernetesを使用する GitOpsプロジェクトの例は、**GitOps-Demo**グループでご覧いただけます。ここでは、Terraformのセキュリティに関する推奨事項、3つの主要なクラウドプロバイダー向けの設定を表すTerraformコード、このデモを自身のグループで再現するための手順へのリンクを提供しています。

GitOps-Demoグループへ移動する

インフラストラクチャオートメーションの展望

GitOpsは魔法ではなく、すでにご存じのIaCオペレーションツールをDevOpsスタイルのワークフローに落とし込んだものです。これにより、リビジョンの追跡が簡単になり、大きな損失を生むエラーが削減され、インフラストラクチャのデプロイが複数の環境やマルチクラウド設定においても繰り返し実行できるようになり、素早く、自動で行えるようになります。

GitOpsを採用すれば、開発者は、安心できないリリースを完全に自動化でき、コードに集中できるため、業務に対する満足度が上がります。チームは手動の手順を排除または最小化でき、再現性のある安定したデプロイを実現できます。

インフラストラクチャのメンテナンスでは問題が生じることが多く、時間もかかります。このプロセスを完全に自動化することで、インフラストラクチャの柔軟性が高まり、頻繁なアプリケーションのデプロイにも対応できるようになります。

GitOpsはセキュリティや標準化も改善します。GitOpsを利用することで、開発者は手動でクラウドリソースにアクセスする必要がなくなり、追加のセキュリティチェックをCI/CDパイプラインのコードレベルで実施できます。

GitLabはGitOpsのワークフローを開始する上で役立ちます。GitLabでは物理、仮想、およびクラウドネイティブのインフラストラクチャ（Kubernetesやサーバーレステクノロジーを含む）を管理できます。また、GitLabはTerraform、AWS Cloud Formation、Ansible、Chef、Puppetなど、業界をリードするインフラストラクチャオートメーションツールとも緊密に連携します。GitLabはGitリポジトリだけでなく、CI/CD、マージリクエスト、シングルサインオンを提供するため、誰もが簡単に共同作業を行え、プラットフォームからクラウドプロバイダーへデプロイできます。

GitLabについて

GitLabはDevOpsライフサイクルのすべての段階に対応する単一のアプリケーションとしてゼロから構築されたDevOpsプラットフォームであり、製品、開発、QA、セキュリティ、および運用チームが同じプロジェクトで同時に作業できるようにします。

GitLabはDevOpsのライフサイクルを通して単一のデータソース、単一のユーザーインターフェイス、および単一の権限モデルをチームに提供することで、チームの共同作業やプロジェクトでの協業を会話形式で行えるようにし、サイクルにかかる時間を大幅に削減して、素晴らしいソフトウェアを素早く構築することに集中できるようにします。

オープンソースで構築されたGitLabは数千人の開発者と数百万人のユーザーから成るコミュニティの貢献により、新しいDevOpsイノベーションを継続的に提供しています。Ticketmaster、Jaguar Land Rover、NASDAQ、Dish Network、Comcastなど、新興企業からグローバル企業まで、10万社を超える企業がGitLabの優れたソフトウェアをこれまでにないスピードで提供する能力を信頼しています。GitLabは65か国に1,200人以上のチームメンバーを抱える世界最大級のオールリモート企業です。


GitLab