GitLab

初心者向けガイド

- 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の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の実行環境を設計、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 Infrastructure-as-code (IaC) 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eBookでは GitOps によるインフラストラクチャのオートメーションプロセスとGitOpsがのインフラストラクチャの設計 、変更及びデプロイにどんな効果があるか、そのエンドツーエンドソリューション となる仕組みについて説明します。 このeBookでは以下を学習できます。

- 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 文化を持つ組織は本番環境にコードを1日何百回もデプロイできます。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は未だに大部分が手動プロセ スです。デプロイの頻度に IT チームが追いつくことが難しいという問 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要な場合、インフラストラクチャのオートメ ーションは事実上不可能でした。しかし、仮想化により、少し作業が しやすくなり、パブリッククラウドの登場により、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可能です。

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、今ではこういったタスクを簡単に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良い方 法でした。今ではこのような手順が自動化され、新しいサーバーの設定も大幅に楽になりまし た。しかし、新しい VM のプロビジョニングは依然として実行できず、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 といった第二世代の 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。例えば、10 の EC2 インスタンスを作成でき、

各インスタンスそれぞれに必要なソフトウェアをすべてインストールできます。 このような 構成管理ツール の重大な欠点の一つはプロビジョニングや設定の対象がサーバーと VM だけで、クラウドネイティブサービスに対するソリューションにはならなかった点です。

Amazon CloudFormation は第二世代の 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 の関係と同様で、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、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/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 では実証済みの DevOps プロセスを、インフラストラクチャを定義したコードに適用します。名前が示す通り、Git とオペレーション、またはリソース管理を組み合わせたも のです。Git はオープンソースのバージョン管理システムで、コードの変更を管理しま す。DevOps と同様、GitOps の目標は CI/CD を使用して自動的にリソースをデプロイすること で、これにはお使いの 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、必要に応じて前のバージョンへロールバックするこ とができます。

また、Git を使用することで、インフラストラクチャを定義したコードについてコードレビューが可能になります。 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、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てしてしまい、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 リポジトリとはプロジェクト内の .gitフ ォルダで、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し ます。Infrastructure as code (IaC) とは、コードとして保存されたインフラス トラクチャの構成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～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。

監査が簡単に。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 を利用すれば、環境に対する変更すべてを git ログに保存できるため、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。

エラーが起こりにくい。インフラストラクチャの定義がコード化され、繰り返し利用できるため、人為的ミスが起こりにく 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はクラウドリソースを管理しやすくなり、クラウドの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）。

コンプライアンスを守りつつ共同作業。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ケーションは自己完結型であるため、各アプリケーション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 ツールを使用して新しい VM を構成 します。

Git code repository	git		
Git management tool	GitLab	GitHub	Bitbucket
Continuous Integration tool	GitLab	Jenkins	circleci
Continuous Delivery tool	GitLab	Spinnaker	flux
Container Registry	GitLab	docker hub	
Configuration manager	ANSIBLE	CHEF	puppet
Infrastructure provisioning	Terraform	pulumi	AWS CloudFormation
Container orchestration	kubernetes		

GitOps の利用を始めるための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む上で役立ちます。

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 構成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良いでしょう。

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。デフォルト設定を使用し、設定に 1 分しかかからないサービス 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。このサービスを手動でデプロイしなければならないことに気づく人もほとんどいない可能性もありますが、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、簡単にCloudFormation 設定ファイルを既存のリソースから作成できます。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

常に全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、サードパーティープロバイダー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 レ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。始めるに当たって、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、丁度良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。

何度も言いますが、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良いでしょう。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一つに、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。

リポジトリ戦略を用意する

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レポ ジトリに保存することも考えられます。これは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、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が、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てください。

小さな変化に留める

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、一つ一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、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** がこれに当たります。

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、気が気で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 の概要

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