vRealize Automation Code Stream の使用と管理

2022 年 12 月 14 日

vRealize Automation 8.7

vmware®

最新の技術ドキュメントは、 VMware の Web サイト （https://docs.vmware.com/jp/）

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

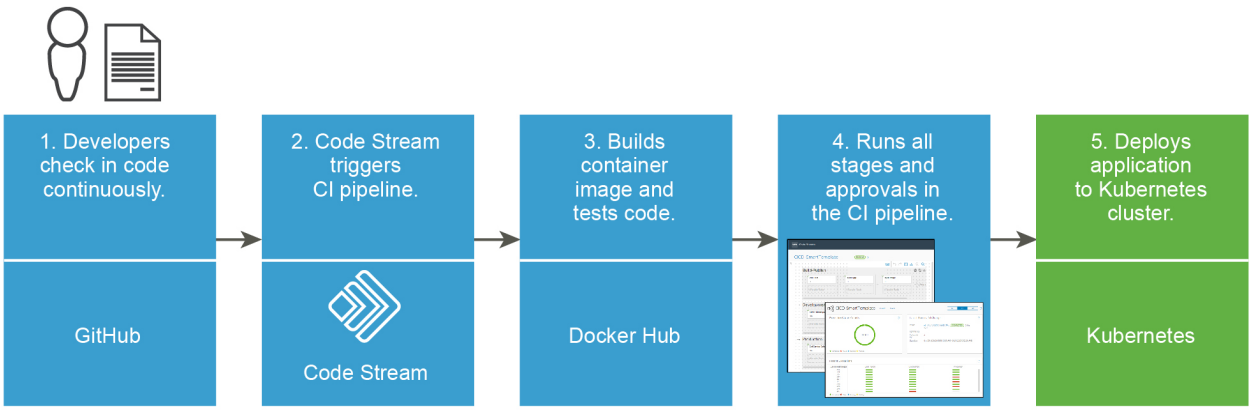
ヴイエムウェア株式会社
〒108-0023 東京都港区芝浦 3-1-1
田町ステーションタワー N 18 階
www.vmware.com/jp

目次

Code Stream の概要と仕組み

1

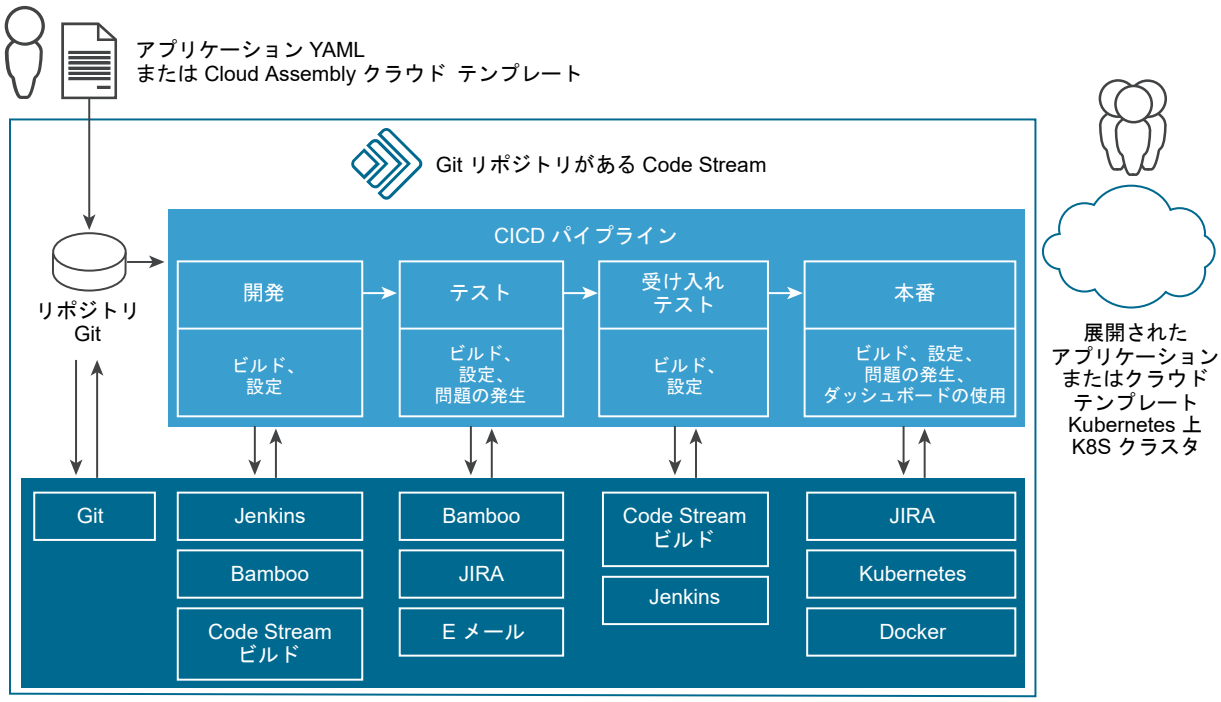
vRealize Automation Code Stream™ は、継続的インテグレーションおよび継続的デリバリ (CICD) ツールです。DevOps ライフサイクルでソフトウェア リリース プロセスをモデル化したパイプラインを作成することにより、ソフトウェアを迅速かつ継続的に配信するコード インフラストラクチャを作成できます。



Code Stream を使用してソフトウェアを配信すると、DevOps ライフサイクルの最も重要な部分のうち、リリース プロセスと開発者ツールの 2 つを統合できます。初期セットアップは Code Stream を既存の開発ツールと統合し、これが終わると、パイプラインは DevOps ライフサイクル全体を自動化します。

vRealize Automation 8.2 以降、ブループリントは VMware Cloud Templates という名称に変更されました。

ソフトウェアをビルド、テスト、およびリリースするパイプラインを作成します。 Code Stream は、このパイプラインを使用して、ソフトウェアをソース コード リポジトリから、テストを経て本番環境に進めます。



4 章 Code Stream でコードをネイティブにビルド、統合、および配信することを計画する で、継続的インテグレーションおよび継続的デリバリ パイプラインの計画の詳細を確認できます。

Code Stream 管理者による Code Stream の使用方法

管理者としてエンドポイントを作成し、開発者が作業インスタンスを使用できるようにします。パイプラインの作成、トリガ、管理などを行うことができます。 Code Stream でユーザー アクセスと承認を管理する方法に記載されているように、`Administrator` ロールが付与されています。

表 1-1. Code Stream 管理者による開発者のサポート方法

開発者へのサポート	実行できる操作
環境を提供および管理します。	開発者がコードをテストして展開するための環境を構築します。 ■ 状態を追跡し、E メール通知を送信します。 ■ 環境が継続的に動作するようにして、開発者の生産性を維持します。 詳細については、 Code Stream 管理者および開発者向けのその他のリソースを参照してください。 また、5 章 Code Stream を使用するためのチュートリアルも参照してください。
エンドポイントを指定します。	エンドポイントのインスタンスが正常に機能していて、開発者が自身のパイプラインに接続できることを確認します。
他のサービスへの統合を提供します。	他のサービスへの統合が機能していることを確認します。 詳細については、VMware Cloud Services ドキュメントを参照してください。
パイプラインを作成します。	リリース プロセスをモデル化するパイプラインを作成します。 詳細については、3 章 Code Stream でのパイプラインの作成と使用を参照してください。

表 1-1. Code Stream 管理者による開発者のサポート方法 （続き）

開発者へのサポート	実行できる操作
パイプラインをトリガします。	イベントが発生したときにパイプラインが実行されていることを確認します。 ■ ビルド アーティファクトの作成または更新時に必ずスタンドアローンの継続的デリバリ (CD) パイプラインをトリガするには、Docker トリガを使用します。 ■ 開発者が変更をコードにコミットしたときにパイプラインをトリガするには、Git トリガを使用します。 ■ 開発者がコードのレビュー、マージなどを行う際にパイプラインをトリガするには、Gerrit トリガを使用します。 ■ ビルド アーティファクトの作成または更新時に必ずスタンドアローンの継続的デリバリ (CD) パイプラインを実行するには、Docker トリガを使用します。 詳細については、7 章 Code Stream でのパイプラインのトリガを参照してください。
パイプラインと承認を管理します。	パイプラインで最新の状態を維持します。 ■ パイプラインのステータスを表示し、パイプラインの実行者を確認します。 ■ パイプライン実行について承認を表示し、アクティブおよび非アクティブなパイプライン実行の承認を管理します。 詳細については、 Code Stream でのユーザー操作と承認についてを参照してください。 また、 Code Stream でカスタム ダッシュボードを使用してパイプラインのキー パフォーマンス インジケータを追跡する方法も参照してください。
開発者環境の監視を行います。	パイプラインのステータス、トレンド、メトリック、および主要インジケータを監視するカスタム ダッシュボードを作成します。カスタム ダッシュボードを使用して、開発環境で合格または失敗したパイプラインを監視します。使用率の低いリソースを特定してレポートし、リソースを解放することもできます。 また、以下を表示することもできます。 ■ パイプラインが正常に完了するまでの実行時間。 ■ パイプラインが承認を待機した時間。パイプラインを承認する必要があるユーザーに通知します。 ■ 最もよく失敗するステージとタスク。 ■ 実行に最も時間がかかるステージとタスク。 ■ 開発チームが進行中のリリース。 ■ 正常に展開およびリリースされたアプリケーション。 詳細については、8 章 Code Stream でのパイプラインの監視を参照してください。
問題のトラブルシューティングを行います。	開発者環境でパイプラインの障害をトラブルシューティングして解決します。 ■ 継続的インテグレーションおよび継続的デリバリ環境 (CICD) の問題を特定して解決します。 ■ 詳細を表示するには、パイプライン ダッシュボードを使用し、カスタム ダッシュボードを作成します。8 章 Code Stream でのパイプラインの監視を参照してください。 また、2 章 リリース プロセスをモデリングするための Code Stream の設定も参照してください。

Code Stream は VMware Cloud Services の一部です。

- Cloud Assembly を使用して、クラウド テンプレートを展開します。
- Service Broker を使用して、カタログからクラウド テンプレートを取得します。

その他に実行できることについては、VMware vRealize Automation ドキュメントを参照してください。

開発者による Code Stream の使用方法

開発者がパイプラインをビルドおよび実行し、ダッシュボードでパイプライン アクティビティを監視するには、Code Stream を使用します。 Code Stream でユーザー アクセスと承認を管理する方法に記載されているように、`User` ロールが付与されています。

パイプラインを実行したら、次のことを確認してください。

- パイプラインのすべての段階でコードが正常に完了したかどうか。確認するには、パイプラインの実行結果を調べます。
- パイプラインが失敗した場合、どうすればよいか、および失敗した原因は何か。確認するには、パイプライン ダッシュボードで上位のエラーを調べます。

表 1-2. Code Stream を使用する開発者

コードを統合してリリースするには	実行する操作
パイプラインをビルドします。	コードをテストして展開します。 パイプラインが失敗したときにコードを更新します。
パイプラインをエンドポイントに接続します。	パイプライン内のタスクを、GitHub リポジトリなどのエンドポイントに接続します。
パイプラインを実行します。	ユーザー操作承認タスクを追加して、別のユーザーが特定の時点でパイプラインを承認できるようにします。
ダッシュボードを表示します。	パイプライン ダッシュボードに結果を表示します。トレンド、履歴、障害などが表示されます。

作業を開始するための詳細については、VMware Code Stream スタート ガイドを参照してください。

製品内サポート パネルで詳細なドキュメントを参照してください

ここで必要な情報が見つからない場合は、製品内にあるヘルプを参照できます。 

- ユーザー インターフェイスの Signpost およびツールチップをクリックして表示し、任意のタイミングと場所でコンテキスト固有の必要な情報を取得します。
- [製品内サポート] パネルを開き、[アクティブなユーザー インターフェイス] 画面に表示されるトピックを参照します。パネルで検索して、質問に対する回答を取得することもできます。

Webhook の詳細

Webhook の構成画面で同じ Git エンドポイントを使用して、ブランチ名に別の値を指定することにより、複数のブランチに対して複数の Webhook を作成できます。同じ Git リポジトリ内の別のブランチに別の Webhook を作成する際に、複数のブランチに対して Git エンドポイントのクローン作成を複数回行う必要はありません。代わりに、Webhook 内でブランチ名を指定すれば、Git エンドポイントを再利用できるようになります。Git Webhook のブランチがエンドポイントのブランチと同じである場合は、Git Webhook 画面でブランチ名を入力する必要はありません。

リリース プロセスをモデリングするための Code Stream の設定

2

リリース プロセスをモデリングするには、ソフトウェアのリリースに通常使用するステージ、タスク、および承認を表すパイプラインを作成します。その後、 Code Stream は、コードのビルド、テスト、承認、および展開を行うプロセスを自動化します。

これで、ソフトウェア リリース プロセスをモデリングするための準備が整いました。次に、 Code Stream でこれを実行する方法を示します。

前提条件

- 使用可能な状態のエンドポイントがあるかどうかを確認します。 Code Stream で、[エンドポイント] をクリックします。
- コードをビルドおよび展開するためのネイティブな方法について説明します。4 章 Code Stream でコードをネイティブにビルド、統合、および配信することを計画する を参照してください。
- パイプラインで使用する一部のリソースを制限付きとしてマークする必要があるかどうかを判断します。Code Stream でユーザー アクセスと承認を管理する方法を参照してください。
- 保持しているロールが管理者ロールではなくユーザー ロールまたはビューア ロールである場合は、 Code Stream インスタンスの管理者を決定します。

手順

1 Code Stream で使用可能なプロジェクトを調べ、適切なプロジェクトを選択します。
   - プロジェクトが表示されない場合は、プロジェクトを作成して自分をプロジェクトのメンバーにするように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。

- リストに表示されているプロジェクトのメンバーでない場合は、自分をプロジェクトのメンバーとして追加するように Code Stream 管理者に依頼します。

2 パイプラインに必要な新しいエンドポイントがあれば追加します。

たとえば、Git、Jenkins、Code Stream ビルド、Kubernetes、および JIRA が必要になる場合があります。

3 パイプライン タスクで値を再利用できるように変数を作成します。

ホスト マシンなど、パイプラインで使用されるリソースを制限するには、制限付き変数を使用します。別のユーザーが明示的に承認するまで、パイプラインが実行を継続することを制限できます。

管理者は、シークレット変数と制限付き変数を作成できます。ユーザーは、シークレット変数を作成できます。

変数は、複数のパイプライン間で何度でも再利用できます。たとえば、ホスト マシンを定義する変数を `HostIPAddress` とします。パイプライン タスクでこの変数を使用するには、`${var.HostIPAddress}` と入力します。

4 管理者であれば、ビジネスに不可欠なエンドポイントと変数を制限付きリソースとしてマークします。

管理者以外のユーザーが、制限されたリソースを含むパイプラインを実行すると、制限されたリソースを使用するタスクでパイプラインが停止します。その場合は、管理者がパイプラインを再開する必要があります。

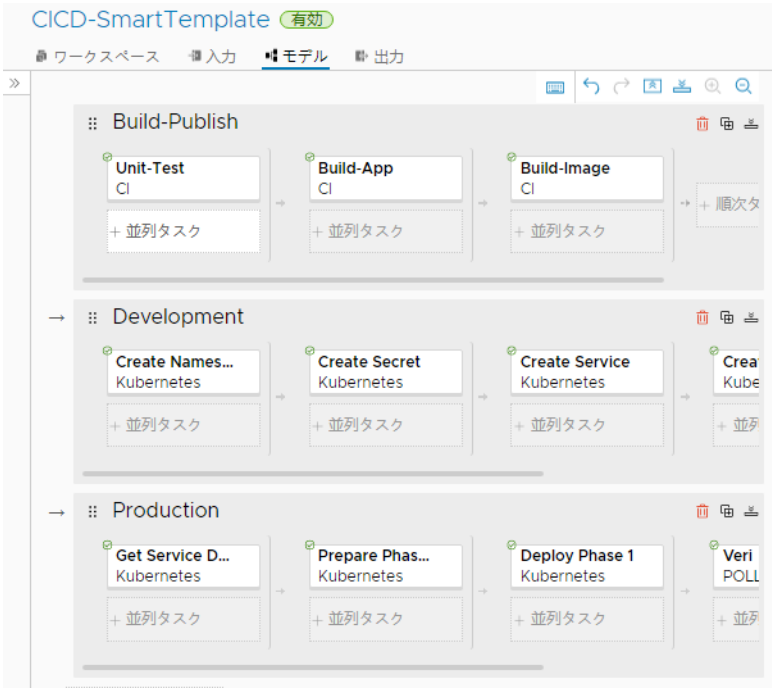
5 ネイティブの CICD、CI、または CD パイプラインのビルド方法を計画します。

コードを連続統合 (CI) および継続展開 (CD) するパイプラインを作成する前に、ビルド方法を計画します。ビルド計画は、コードのネイティブなビルド、統合、テスト、および展開を実行するために、 Code Stream が何を必要とするかを判断するのに役立ちます。

Code Stream ネイティブ ビルドの作成方法	ビルド方法の結果
スマート パイプライン テンプレートの 1 つを使用します。	■ すべてのステージとタスクをビルドします。 ■ ソース リポジトリのクローンを作成します。 ■ コードをビルドおよびテストします。 ■ 展開のコードをコンテナ化します。 ■ 選択内容に基づいて、パイプラインのタスク手順を入力します。
手動でステージとタスクを追加します。	ステージとタスクを追加し、それらに入力する情報を入力します。

6 スマート パイプライン テンプレートを使用するかステージとタスクを手動でパイプラインに追加することにより、パイプラインを作成します。

その後、すべてのリソースを制限付きとしてマークします。必要な承認を追加します。通常、制限付き、またはシークレットのいずれかの変数を適用します。タスク間にバインドを追加します。



7 パイプラインを検証して有効にし、実行します。

8 パイプラインの実行を表示します。

9 ステータスとキー パフォーマンス インジケータ (KPI) を追跡するには、パイプライン ダッシュボードを使用し、カスタム ダッシュボードを作成します。

結果

選択したプロジェクトで使用できるパイプラインを作成しました。

パイプライン YAML をエクスポートし、他のプロジェクトにインポートして再利用することもできます。

次のステップ

環境に適用する可能性のあるユースケースについて説明します。5 章 Code Stream を使用するためのチュートリアルを参照してください。

Code Stream でプロジェクトを追加する方法

プロジェクトを作成し、そこに管理者とメンバーを追加します。プロジェクト メンバーは、パイプラインの作成やエンドポイントの追加などの機能を使用できます。開発チームのプロジェクトを作成、削除、または更新するには、Code Stream 管理者である必要があります。

プロジェクトは、パイプラインを作成する前に設定する必要があります。パイプラインを作成するときは、すべてのパイプライン情報が集約されるプロジェクトを選択します。エンドポイントと変数の定義は、既存のプロジェクトにも依存します。

前提条件

- Code Stream 管理者ロールが割り当てられていることを確認します。Code Stream でのロールとはを参照してください。

  Code Stream 管理者ロールのない Cloud Assembly 管理者である場合は、Cloud Assembly ユーザー インターフェイスを使用してプロジェクトを作成、更新、または削除できます。Cloud Assembly 開発チームのプロジェクトを追加する方法を参照してください。
- Active Directory グループをプロジェクトに追加する場合は、組織で Active Directory グループを設定してあることを確認します。vRealize Automation でプロジェクトに対して Active Directory グループを有効にする方法を参照してください。同期されていないグループはプロジェクトに追加できません。

手順

1. [プロジェクト] を選択し、[新しいプロジェクト] をクリックします。
2. プロジェクト名を入力します。
3. [作成] をクリックします。
4. 新しく作成されたプロジェクトのカードを選択し、[開く] をクリックします。
5. [ユーザー] タブをクリックし、ユーザーを追加してロールを割り当てます。
   - プロジェクト管理者はメンバーを追加できます。
   - サービス ロールを持つプロジェクト メンバーがサービスを使用できます。
   - プロジェクト閲覧者はプロジェクトを表示できますが、作成、更新、削除はできません。

   プロジェクト ロールの詳細については、 Code Stream でユーザー アクセスと承認を管理する方法を参照してください。

6 [保存] をクリックします。

次のステップ

プロジェクトを使用するエンドポイントとパイプラインを追加します。6 章 エンドポイントへの Code Stream の接続および 3 章 Code Stream でのパイプラインの作成と使用を参照してください。

パイプラインを作成すると、すべてのパイプライン情報が集約されるプロジェクトの名前が、パイプライン カードとパイプライン実行カードに表示されます。

Code Stream でユーザー アクセスと承認を管理する方法

Code Stream には、ソフトウェア アプリケーションをリリースするパイプラインを操作するための適切な権限と承諾をユーザーに付与する方法がいくつか用意されています。

チームのメンバーごとにロールが割り当てられています。パイプライン、エンドポイント、ダッシュボード、およびリソースに制限ありとマークする機能に特定の権限を付与するロールです。

ユーザー操作と承認によって、パイプラインをいつ実行し、いつ停止して承認を得る必要があるかを制御できます。ロールによって、パイプラインを再開できるかどうかと、エンドポイントや変数に制限があるパイプラインを実行できるかどうかが決まります。

機密性のある情報を非表示および暗号化するには、シークレット変数を使用します。非表示にして暗号化して実行時に使用を制限する文字列、パスワード、URL については、制限付き変数を使用します。たとえば、パスワードや URL にシークレット変数を使用します。シークレット変数も制限付き変数も、パイプライン内の任意のタイプのタスクで使用できます。

Code Stream でのロールとは

Code Stream のロールによって、実行できるアクションとアクセスできる領域が決まります。たとえば、パイプラインの作成、更新、実行ができるロールがあります。あるいは、パイプラインの表示のみができる権限もあります。

制限付きを除くすべてのアクションは、このロールには、制限付きの変数およびエンドポイントを除くエンティティに対して作成、読み取り、更新、および削除アクションを実行する権限があることを意味します。

表 2-1. Code Stream でのサービス レベルおよびプロジェクト レベルのアクセス権限

	Code Stream ロール				
アクセス レベル	**Code Stream 管理者**	**Code Stream 開発者**	**Code Stream 実行者**	**Code Stream ビューア**	**Code Stream ユーザー**
Code Stream サービス レベルのアクセス	すべてのアクション	制限付きを除くすべてのアクション	実行アクション	読み取り専用	なし
プロジェクト レベルのアクセス : プロジェクト管理者	すべてのアクション	すべてのアクション	すべてのアクション	すべてのアクション	すべてのアクション

表 2-1. Code Stream でのサービス レベルおよびプロジェクト レベルのアクセス権限 （続き）

	Code Stream ロール				
アクセス レベル	**Code Stream 管理者**	**Code Stream 開発者**	**Code Stream 実行者**	**Code Stream ビューア**	**Code Stream ユーザー**
プロジェクト レベルのアクセス：プロジェクト メンバー	すべてのアクション	制限付きを除くすべてのアクション	制限付きを除くすべてのアクション	制限付きを除くすべてのアクション	制限付きを除くすべてのアクション
プロジェクト レベルのアクセス：プロジェクト閲覧者	すべてのアクション	制限付きを除くすべてのアクション	実行アクション	読み取り専用	読み取り専用

プロジェクト管理者ロールを持つユーザーは、自身がプロジェクト管理者になっているプロジェクトに対してすべてのアクションを実行できます。

プロジェクト管理者は、パイプライン、変数、エンドポイント、ダッシュボード、トリガの作成、読み取り、更新、および削除を実行できます。また、これらのリソースが、ユーザーがプロジェクト管理者になっているプロジェクトに含まれている場合は、制限付きのエンドポイントまたは変数が含まれているパイプラインを起動できます。

サービス閲覧者ロールを持つユーザーは、管理者が使用できるすべての情報を表示できます。これらのユーザーは、管理者によってプロジェクト管理者またはプロジェクト メンバーにされない限り、アクションを実行することはできません。プロジェクトに関連しているユーザーは、そのロールに関連する権限を持ちます。プロジェクト閲覧者は、管理者ロールまたはメンバー ロールとは異なり、権限が拡張されることはありません。このロールは、すべてのプロジェクトで読み取り専用です。

プロジェクトに読み取り権限が設定されている場合も、制限付きのリソースは表示されます。

- 制限付きのエンドポイント（エンドポイント カードにロック アイコンが表示される）を表示するには、[構成] - [エンドポイント] の順にクリックします。
- 制限付きの変数とシークレット変数（[タイプ] 列に [制限付き] または [シークレット] と表示される）を表示するには、[構成] - [変数] の順にクリックします。

表 2-2. Code Stream サービス ロールの機能

ユーザー インターフェイスのコンテキスト	**機能**	**Code Stream 管理者ロール**	**Code Stream 開発者ロール**	**Code Stream 実行者ロール**	**Code Stream 閲覧者ロール**	**Code Stream ユーザー ロール**
[パイプライン]						
	パイプラインの表示	はい	はい	はい	はい	
	パイプラインの作成	はい	はい			
	パイプラインの実行	はい	はい	はい		
	エンドポイントや変数に制限があるパイプラインの実行	はい				

表 2-2. Code Stream サービス ロールの機能 （続き）

ユーザー インターフェイスのコンテキスト	機能	Code Stream 管理者ロール	Code Stream 開発者ロール	Code Stream 実行者ロール	Code Stream 閲覧者ロール	Code Stream ユーザー ロール
	パイプラインの更新	はい	はい			
	パイプラインの削除	はい	はい			
[パイプラインの実行]						
	パイプラインの実行状況の表示	はい	はい	はい	はい	
	パイプラインの実行の再開、一時停止、およびキャンセル	はい	はい	はい		
	制限があるリソースの承認のために停止しているパイプラインの再開	はい				
[カスタム統合]						
	カスタム統合の作成	はい	はい			
	カスタム統合の読み取り	はい	はい	はい	はい	
	カスタム統合の更新	はい	はい			
[エンドポイント]						
	実行の表示	はい	はい	はい	はい	
	実行の作成	はい	はい			
	実行の更新	はい	はい			
	実行の削除	はい	はい			
[リソースを制限付きとしてマーク]						
	エンドポイントまたは変数を制限付きとしてマーク	はい				
[ダッシュボード]						
	ダッシュボードの表示	はい	はい	はい	はい	

表 2-2. Code Stream サービス ロールの機能 （続き）

ユーザー インターフェイスのコンテキスト	機能	Code Stream 管理者ロール	Code Stream 開発者ロール	Code Stream 実行者ロール	Code Stream 閲覧者ロール	Code Stream ユーザー ロール
	ダッシュボードの作成	はい	はい			
	ダッシュボードの更新	はい	はい			
	ダッシュボードの削除	はい	はい			

Code Stream のカスタム ロールと権限

Cloud Assembly で、パイプラインを使用するユーザーまで権限を拡張するカスタム ロールを作成できます。Code Stream パイプラインのカスタムロールを作成する場合は、[パイプライン] 権限を 1 つ以上選択します。

このカスタム ロールが割り当てられるユーザーに必要な [パイプライン] 権限の最小数を選択します。

プロジェクトに割り当てられているユーザーに、そのプロジェクト内のロールが付与されていて、さらに 1 つ以上の [パイプライン] 権限を含むカスタム ロールが割り当てられている場合、ユーザーはこの権限で許可されているすべてのアクションを実行できます。たとえば、制限付き変数の作成、制限付きパイプラインの管理、カスタム統合の作成と管理などを行うことができます。

表 2-3. カスタム ロールに割り当てることができるパイプライン権限

パイプライン権限	Code Stream 管理者	Code Stream 開発者	Code Stream 実行者	Code Stream ビューア	Code Stream ユーザー	プロジェクト管理者	プロジェクト メンバー	プロジェクト閲覧者
パイプラインの管理	はい	はい				はい	はい	
制限付きパイプラインの管理	はい					はい		
カスタム統合の管理	はい	はい						
パイプラインの実行	はい	はい	はい			はい	はい	
制限付きパイプラインの実行	はい					はい		
実行の管理	はい					はい		
読み取り。この権限は表示されません。	はい	はい	はい	はい		はい	はい	はい

表 2-4. カスタム ロールでパイプライン権限を使用する方法

権限	実行できる操作
パイプラインの管理	■ パイプラインを作成、更新、削除、クローン作成します。 ■ パイプラインを VMware Service Broker にリリース/リリース解除します。 ■ クラウド テンプレートを作成、更新、削除します。 ■ 通常の変数とシークレット変数を作成、更新、削除します。 ■ Gerrit リスナーを作成、クローン作成、更新、削除します。 ■ Gerrit リスナーを接続および切断します。 ■ Gerrit トリガを作成、クローン作成、更新、削除します。 ■ Git webhook を作成、更新、削除します。 ■ Docker webhook を作成、更新、削除します。 ■ スマート パイプライン テンプレートを使用してパイプラインを作成します。 ■ YAML からパイプラインをインポートし、YAML にエクスポートします。 ■ カスタム ダッシュボードを作成、更新、削除します。 ■ すべてのカスタム統合を読み取ります。 ■ 制限付きのすべてのエンドポイントおよび変数を読み取りますが、それらの値を表示することはできません。
制限付きパイプラインの管理	■ クラウド テンプレートを作成、更新、削除します。 ■ エンドポイントに制限付きとマークして、制限付きエンドポイントを更新し、削除します。 ■ 通常の変数とシークレット変数を作成、更新、削除します。 ■ 制限付き変数を作成、更新、削除します。 ■ パイプラインの管理で使用できるすべての権限。
カスタム統合の管理	■ カスタム統合を作成および更新します。 ■ カスタム統合のバージョン管理とリリースを行います。 ■ カスタム統合バージョンを削除および廃止します。 ■ カスタム統合を削除します。
パイプラインの実行	■ パイプラインを実行します。 ■ パイプラインの実行を一時停止、再開、キャンセルします。 ■ パイプラインの実行を再実行します。 ■ Gerrit トリガ イベントを再開、再実行、手動でトリガします。 ■ ユーザー操作を承認します。ユーザー操作のバッチ承認を実行できます。
制限付きパイプラインの実行	■ パイプラインを実行します。 ■ パイプラインの実行を一時停止、再開、キャンセル、削除します。 ■ パイプラインの実行を再実行します。 ■ 稼動中のパイプラインの実行を同期します。 ■ 稼動中のパイプラインの実行を強制的に削除します。 ■ Gerrit トリガ イベントを再開、再実行、削除、手動でトリガします。 ■ 制限された項目を解決して、パイプラインの実行を続行します。 ■ ユーザー コンテキストを切り替えて、ユーザー操作タスクの承認後にパイプラインの実行を続行します。 ■ パイプラインの実行で使用できるすべての権限。
実行の管理	■ パイプラインを実行します。 ■ パイプラインの実行を一時停止、再開、キャンセル、削除します。 ■ パイプラインの実行を再実行します。 ■ Gerrit トリガ イベントを再開、再実行、削除、手動でトリガします。 ■ パイプラインの実行で使用できるすべての権限。

カスタム ロールには、権限の組み合わせを含めることができます。これらの権限は、リソースが制限されているかどうかに関係なく、パイプラインの管理や実行をユーザーに許可する機能グループ別に編成されています。これらの権限は、各ロールが Code Stream で実行できるすべての機能を表しています。

たとえば、カスタム ロールを作成し、[制限付きパイプラインの管理] 権限を含めると、 Code Stream 開発者ロールを持つユーザーは次のことが可能になります。

- クラウド テンプレートを作成、更新、削除します。
- エンドポイントに制限付きとマークして、制限付きエンドポイントを更新し、削除します。
- 通常の変数とシークレット変数を作成、更新、削除します。
- 制限付き変数を作成、更新、削除します。

表 2-5. カスタム ロールにおけるパイプライン権限の組み合わせの例

カスタム ロールに割り当てられている権限の数	統合された権限の例	この組み合わせの使用方法
単一の権限	[パイプラインの実行]	
2 つの権限	[パイプラインの管理] と [パイプラインの実行]	
3 つの権限	[パイプラインの管理]、[パイプラインの実行]、および[制限付きパイプラインの実行]	
	[パイプラインの管理]、[カスタム統合の管理]、および[制限付きパイプラインの実行]	この組み合わせは Code Stream 開発者ロールに適用されることがありますが、ユーザーがメンバーになっているプロジェクトに限定されます。
	[パイプラインの管理]、[カスタム統合の管理]、および [実行の管理]	この組み合わせは Code Stream 管理者ロールに適用されることがありますが、ユーザーがメンバーになっているプロジェクトに限定されます。
	[パイプラインの管理]、[制限付きパイプラインの管理]、および [カスタム統合の管理]	この組み合わせにより、ユーザーには完全な権限が付与され、 Code Stream のすべての要素を作成および削除できるようになります。

管理者ロールが付与されている場合

管理者は、カスタム統合、エンドポイント、変数、トリガ、パイプライン、およびダッシュボードを作成できます。

プロジェクトを使用すると、パイプラインからインフラストラクチャ リソースにアクセスできます。管理者は、ユーザーがパイプライン、エンドポイント、ダッシュボードをグループ化できるようにプロジェクトを作成します。ユーザーは、パイプラインでプロジェクトを選択します。各プロジェクトには、ロールが割り当てられた管理者とユーザーが含まれます。

管理者ロールがある場合は、エンドポイントおよび変数を制限があるリソースとしてマークし、制限があるリソースを使用しているパイプラインを実行できます。管理者以外のユーザーが制限付きのエンドポイントまたは変数を含むパイプラインを実行すると、制限付きの変数が使用されているタスクでパイプラインが停止し、管理者はパイプラインを再開する必要があります。

管理者は、パイプラインを vRealize Automation Service Broker で公開するように申請することもできます。

開発者ロールが付与されている場合

管理者と同様にパイプラインを操作できますが、制限があるエンドポイントや変数を使用することはできません。

制限付きのエンドポイントまたは変数を使用するパイプラインを実行する場合、パイプラインは、制限されたリソースを使用するタスクまでしか実行されません。その後、パイプラインは停止し、Code Stream 管理者またはプロジェクト管理者が再開する必要があります。

ユーザー ロールが付与されている場合

Code Stream にはアクセスできますが、他のロールによって提供される権限はありません。

閲覧者ロールが付与されている場合

パイプライン、エンドポイント、パイプラインの実行、ダッシュボード、カスタム統合、トリガなど、管理者に表示されるものと同じリソースが表示されますが、それらを作成、更新、または削除することはできません。アクションを実行するには、閲覧者ロールにプロジェクト管理者ロールまたはプロジェクト メンバーロールも付与する必要があります。

閲覧者ロールを持つユーザーは、プロジェクトを表示できます。また、制限付きエンドポイントと制限付き変数も表示できますが、これらの詳細を表示することはできません。

実行者ロールが付与されている場合

パイプラインを実行し、ユーザー操作タスクに対してアクションを実行できます。また、パイプラインの実行の再開、一時停止、キャンセルも可能です。ただし、パイプラインを変更することはできません。

ロールの割り当ておよび更新方法

他のユーザーにロールを割り当てたり更新するには、管理者である必要があります。

1 vRealize Automation で アクティブなユーザーとそのロールを表示するには、右上にある 9 つのドットをクリックします。

2 [ID およびアクセス権の管理] をクリックします。

3 ユーザー名とロールを表示するには、[アクティブなユーザー] をクリックします。

4 ユーザーへのロールの追加、またはロールの変更をするには、ユーザー名の横にあるチェックボックスをクリックし、[ロールの編集] をクリックします。

5 ユーザー ロールを追加または変更するときに、サービスへのアクセス権を追加することもできます。

6 変更を保存するには、[保存] をクリックします。

Code Stream でのユーザー操作と承認について

ユーザー操作領域には、承認が必要な内容を実行するパイプラインが表示されます。必須の承認者は、パイプラインの実行を承認または拒否できます。

パイプラインを作成する場合、次の場合にパイプラインへの承認の追加が必要になる場合があります。

- チーム メンバーはコードを確認する必要があります。
- 別のユーザーはビルド アーティファクトを確認する必要があります。
- すべてのテストが完了していることを確認する必要があります。
- タスクは、管理者が制限付きとしてマークしたリソースを使用しているため、承認が必要です。
- パイプラインがソフトウェアを本番環境にリリースします。

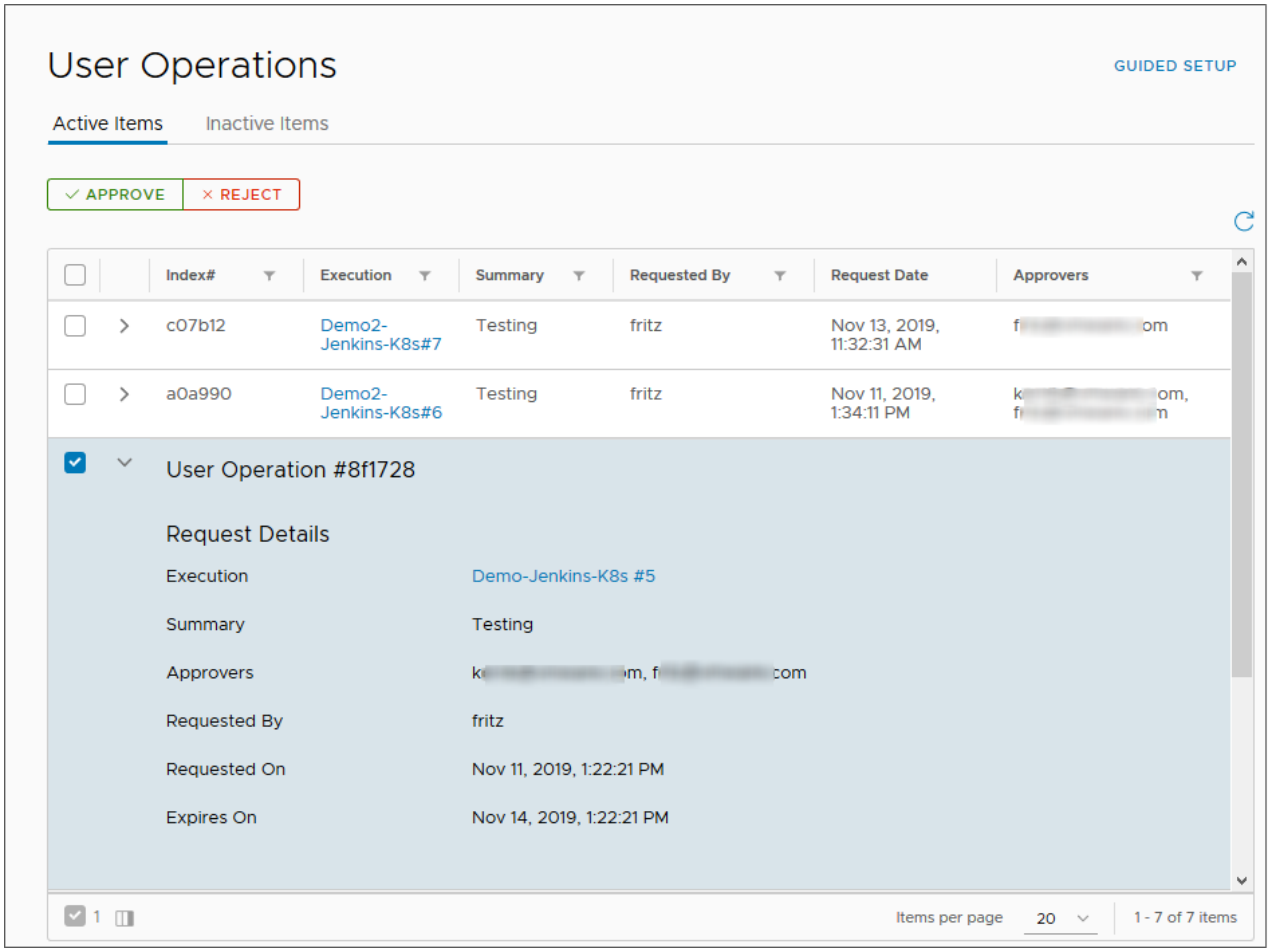
パイプライン タスクを承認するかどうかを決定するには、必須の承認者が権限と専門知識を持っている必要があります。

ユーザー操作タスクを追加する場合は、有効期限のタイムアウトを日、時間、または分単位で設定できます。たとえば、目的のユーザーに、パイプラインを 30 分間で承認するよう依頼しなければならない場合があります。30 分以内に承認されない場合、パイプラインは予期したとおりに失敗します。

E メール通知の送信を有効にした場合、ユーザー操作タスクでは、完全な E メール アドレスを持つ承認者にのみ通知が送信され、E メール形式ではない承認者名には通知が送信されません。

必須のユーザーがタスクを承認すると、次のようになります。

- 保留中のパイプラインの実行を継続できます。
- パイプラインを続行すると、その同じユーザー操作タスクの承認を求める以前の保留中の要求がキャンセルされます。



[ユーザー操作] 領域で、承認または却下するアイテムがアクティブなアイテムまたはアクティブでないアイテムとして表示されます。各アイテムは、パイプライン内のユーザー操作タスクにマッピングします。

- [アクティブなアイテム] では、タスクをレビューする必要がある承認者を待ち、その後タスクを承認または拒否します。承認者リストにあるユーザーは、ユーザー操作行を展開し、[承認] または [拒否] をクリックできます。
- [アクティブでないアイテム] が承認または拒否されました。ユーザーがユーザー操作を拒否した場合、またはタスクの承認がタイムアウトになった場合は、そのタスクは承認できなくなります。

インデックス # は、特定の承認を検索するフィルタとして使用できる一意の 6 文字の英数字文字列です。

パイプラインの承認は、[実行] 領域にも表示されます。

- 承認を待機しているパイプラインは、ステータスが待機中であることを示します。
- その他の状態には、キュー、完了、失敗があります。
- パイプラインが待機状態にある場合は、必要な承認者がパイプライン タスクを承認する必要があります。

Code Stream でのパイプラインの作成と使用

3

vRealize Automation Code Stream を使用して、ビルド、テスト、および展開のプロセスをモデリングすることができます。vRealize Automation Code Stream では、リリース サイクルをサポートするインフラストラクチャを設定し、ソフトウェアのリリース アクティビティをモデル化するパイプラインを作成します。vRealize Automation Code Stream は、開発コードからソフトウェアを提供し、テストを経て本番インスタンスに展開します。

各パイプラインには、ステージとタスクがあります。ステージは開発段階を表し、タスクはステージを介してソフトウェア アプリケーションを配信する必須のアクションを実行します。

vRealize Automation Code Stream のパイプラインについて

パイプラインは、ソフトウェア リリース プロセスの継続的インテグレーションと継続的デリバリのモデルです。ここでソフトウェアはソース コードからテストを経て本番環境にリリースされます。ステージは複数が連続したものとなっており、各ステージにはタスクがあります。タスクとは、ソフトウェア リリース サイクルのアクティビティのことです。ソフトウェア アプリケーションは、パイプラインを介して、あるステージから次のステージへと流れていきます。

パイプラインのタスクがデータ ソース、リポジトリ、または通知システムに接続できるようにエンドポイントを追加します。

パイプラインの作成

パイプラインの作成を空のキャンバスから開始する場合は、スマート パイプライン テンプレートを使用するか、YAML コードをインポートすることができます。

- 空白のキャンバスを使用します。例については、タスクの手動追加を行う前の Code Stream での CICD ネイティブ ビルドの計画を参照してください。
- スマート パイプライン テンプレートを使用します。例については、4 章 Code Stream でコードをネイティブにビルド、統合、および配信することを計画する を参照してください。
- YAML コードをインポートします。[パイプライン] - [インポート] の順にクリックします。[インポート] ダイアログボックスで、YAML ファイルを選択するか YAML コードを入力し、[インポート] をクリックします。

空白のキャンバスを使用してパイプラインを作成する場合は、ステージ、タスク、および承認を追加します。パイプラインは、アプリケーションをビルド、テスト、展開、リリースするプロセスを自動化します。各ステージのタスクは、それぞれのステージでコードをビルド、テスト、およびリリースするアクションを実行します。

表 3-1. パイプライン ステージと使用例

ステージの例	実行できる操作の例
開発	開発ステージでは、マシンのプロビジョニング、アーティファクトの取得が可能です。また、ビルド タスクを追加してコードの継続的インテグレーションに使用する Docker ホストの作成などを行うことができます。 例： ■ vRealize Automation Code Stream のネイティブ ビルド機能を使用してコードを配信する継続的インテグレーション (CI) ビルドを計画して作成するには、スマート パイプライン テンプレートの使用に先立つ Code Stream での継続的インテグレーション ネイティブ ビルドの計画を参照してください。
テスト	テスト ステージでは、ソフトウェア アプリケーションをテストする Jenkins タスクを追加し、JUnit や JaCoCo などの後処理テスト ツールを含めることができます。 例： ■ vRealize Automation Code Stream を Jenkins と統合し、ソース コードをビルドしてテストするパイプラインで Jenkins ジョブを実行します。 Code Stream を Jenkins と統合する方法を参照してください。 ■ 自分のビルド、テスト、展開ツールと統合できるように、vRealize Automation Code Stream の機能を拡張するカスタム スクリプトを作成します。ビルド、テスト、展開用の独自のツールを Code Stream に統合する方法を参照してください。 ■ 継続的インテグレーション (CI) パイプラインの後処理のトレンドを追跡します。 Code Stream でカスタム ダッシュボードを使用してパイプラインのキー パフォーマンス インジケータを追跡する方法を参照してください。
本番	本番ステージでは、Cloud Assembly のクラウド テンプレートと統合して、インフラストラクチャのプロビジョニング、Kubernetes クラスタへのソフトウェアの展開などを行うことができます。 例： ■ 独自のブルーグリーン展開モデルにソフトウェア アプリケーションを展開できる、開発および本番環境用のステージの例を表示するには、 Code Stream のアプリケーションをブルーグリーン展開に展開する方法を参照してください。 ■ クラウド テンプレートをパイプラインに統合するには、Code Stream の YAML クラウド テンプレートから展開するアプリケーションのリリースを自動化する方法を参照してください。また、展開タスクを追加して、アプリケーションを展開するスクリプトを実行することもできます。 ■ Kubernetes クラスタへのソフトウェア アプリケーションの展開を自動化するには、Kubernetes クラスタへの Code Stream のアプリケーションのリリースを自動化する方法を参照してください。 ■ コードをパイプラインに統合し、ビルド イメージを展開するには、コードを my GitHub または GitLab リポジトリから Code Stream の自分のパイプラインに継続的に統合する方法を参照してください。

パイプラインを YAML ファイルとしてエクスポートできます。[パイプライン] をクリックし、パイプライン カードをクリックしてから、[アクション] - [エクスポート] の順にクリックします。

パイプラインの承認

パイプライン内の特定の時点で、別のチーム メンバーから承認を得ることができます。

- パイプラインにユーザー操作タスクを含めることでパイプラインの承認を要求するには、パイプラインを実行して結果を確認する方法を参照してください。このタスクは、承認を行うメンバーに E メール通知を送信します。メンバーが承認するか拒否しないと、パイプラインは実行を継続できません。ユーザー操作タスクの有効期限のタイムアウトが日、時間、または分単位で設定されている場合、タスクの有効期限が切れる前に、目的のユーザーがパイプラインを承認する必要があります。そうしないと、パイプラインは予期したとおり失敗します。

- パイプラインのどのステージでも、タスクまたはステージで障害が発生した場合は、vRealize Automation Code Stream で JIRA チケットを作成できます。パイプライン タスクが失敗したときに Code Stream で JIRA チケットを作成する方法を参照してください。

パイプラインのトリガ

パイプラインは、開発者がコードをリポジトリにチェックインしたり、コードを表示したりするときや、新規または更新されたビルド アーティファクトを識別するときにトリガされる可能性があります。

- vRealize Automation Code Stream を Git ライフサイクルと統合し、開発者がコードを更新するときにパイプラインをトリガするには、Git トリガを使用します。 Code Stream で Git トリガを使用してパイプラインを実行する方法を参照してください。
- vRealize Automation Code Stream を Gerrit コード レビュー ライフサイクルと統合し、コードのレビュー時にパイプラインをトリガするには、Gerrit トリガを使用します。 Code Stream で Gerrit トリガを使用してパイプラインを実行する方法を参照してください。
- Docker ビルド アーティファクトが作成または更新されるときにパイプラインがトリガされるようにするには、Docker トリガを使用します。 Code Stream で Docker トリガを使用して、継続的な配信パイプラインを実行する方法を参照してください。

vRealize Automation Code Stream でサポートされるトリガの詳細については、7 章 Code Stream でのパイプラインのトリガを参照してください。

この章には、次のトピックが含まれています。

- パイプラインを実行して結果を確認する方法
- Code Stream で使用可能なタスクのタイプ
- Code Stream パイプラインで変数のバインドを使用する方法
- 条件タスクで変数バインドを使用して、 Code Stream でパイプラインを実行または停止する方法
- Code Stream でパイプライン タスクをバインドするときに使用できる変数と式
- Code Stream でパイプラインに関する通知を送信する方法
- パイプライン タスクが失敗したときに Code Stream で JIRA チケットを作成する方法
- Code Stream で展開をロールバックする方法

パイプラインを実行して結果を確認する方法

パイプラインは、パイプライン カード、パイプライン編集モード、およびパイプライン実行から実行できます。また、利用可能なトリガーを使用することで、特定のイベントが発生したときに Code Stream がパイプラインを実行するようにすることもできます。

パイプライン内のすべてのステージとタスクが有効であれば、パイプラインはリリース、実行、またはトリガーする準備ができている状態です。

Code Stream を使用してパイプラインを実行またはトリガするには、パイプライン カードから、またはパイプライン内でパイプラインを有効にして実行することができます。次に、パイプラインの実行を表示して、パイプラインがコードをビルド、テスト、および展開したことを確認できます。

パイプラインの実行が進行中の場合は、管理者または管理者以外のユーザーとして実行を削除できます。

- 管理者：実行中のパイプラインの実行を削除するには、[実行] をクリックします。削除する実行に対して、[アクション] - [削除] をクリックします。
- 管理者以外のユーザー：実行中のパイプラインの実行を削除するには、[実行] をクリックして、[Alt Shift d] をクリックします。

パイプラインの実行が進行中になっているにもかかわらず、停止しているように見える場合、管理者は [実行] ページまたは [実行の詳細] ページから実行を更新できます。

- [実行] ページ：[実行] をクリックします。更新する実行に対して、[アクション] - [同期] をクリックします。
- [実行の詳細] ページ：[実行] をクリックし、実行の詳細へのリンクをクリックして、[アクション] - [同期] をクリックします。

特定のイベントが発生したときにパイプラインを実行するには、トリガを使用します。

- Git トリガは、開発者がコードを更新したときにパイプラインを実行できます。
- Gerrit トリガは、コード レビューが発生したときにパイプラインを実行できます。
- Docker トリガは、Docker レジストリでアーティファクトが作成されたときにパイプラインを実行できます。
- `curl` コマンドまたは `wget` コマンドを使用すると、Jenkins のビルドが終了した後に Jenkins にパイプラインを実行させることができます。

トリガの使用の詳細については、7 章 Code Stream でのパイプラインのトリガを参照してください。

次の手順では、パイプラインをパイプライン カードから実行し、実行を表示し、実行の詳細を参照し、アクションを使用する方法について説明します。パイプラインをリリースして、vRealize Automation Service Broker に追加できるようにする方法についても説明します。

前提条件

- 1 つ以上のパイプラインが作成されていることを確認します。5 章 Code Stream を使用するためのチュートリアルの例を参照してください。

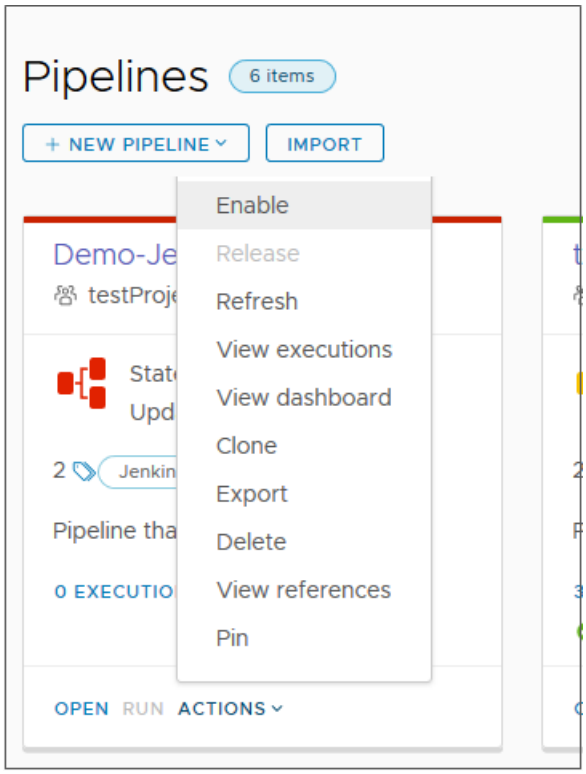
手順

1 パイプラインを有効にします。

パイプラインを実行またはリリースするには、まずパイプラインを有効にする必要があります。

a [パイプライン] をクリックします。

b パイプライン カードで、[アクション] - [有効化] の順にクリックします。



パイプライン内からパイプラインを有効にすることもできます。パイプラインがすでに有効になっている場合、[実行] がアクティブになり、[アクション] メニューには [無効化] が表示されます。

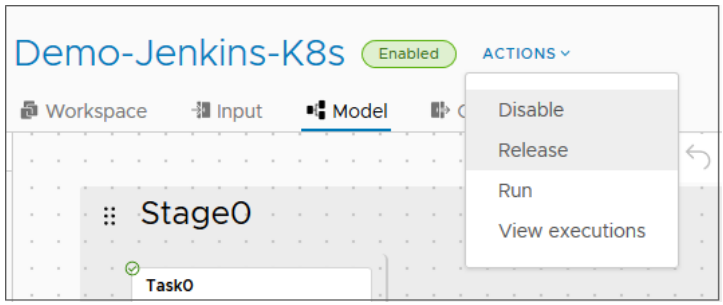
2 （オプション） パイプラインをリリースします。

パイプラインを vRealize Automation Service Broker 内のカタログ アイテムとして使用できるようにする場合は、 Code Stream でリリースする必要があります。

a [パイプライン] をクリックします。

b パイプライン カードで、[アクション] - [リリース] の順にクリックします。

パイプライン内からパイプラインをリリースすることもできます。



パイプラインをリリースした後、Service Broker を開き、パイプラインをカタログ アイテムとして追加して実行します。Service Broker カタログへの Code Stream パイプラインの追加を参照してください。

注： パイプラインの実行に要する時間が 120 分を超える場合は、概算の実行時間を要求のタイムアウト値として指定してください。プロジェクトの要求タイムアウトを設定または確認するには、Service Broker を管理者として開き、[インフラストラクチャ] - [プロジェクト] の順に選択します。次に、プロジェクト名をクリックして、[プロビジョニング] をクリックしてください。

要求タイムアウト値を設定していない場合、120 分を超える時間が必要な実行は失敗として表示され、コールバック タイムアウト要求エラーが示されます。ただし、パイプラインの実行に影響はありません。

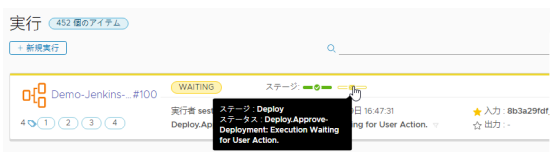
3 パイプライン カードで、[実行] をクリックします。

4 実行中にパイプラインを表示するには、[実行] をクリックします。

パイプラインでは各ステージが順番に実行され、パイプラインの実行により各ステージのステータス アイコンが表示されます。パイプラインにユーザー操作タスクが含まれている場合、ユーザーはそのタスクがパイプラインで引き続き実行されるように承認する必要があります。ユーザー操作タスクが使用されると、パイプラインは実行を停止し、必要なユーザーがタスクを承認するまで待機します。

たとえば、ユーザー操作タスクを使用して、本番環境へのコードの展開を承認することができます。

ユーザー操作タスクの有効期限のタイムアウトが日、時間、または分単位で設定されている場合、タスクの有効期限が切れる前に、目的のユーザーがパイプラインを承認する必要があります。そうしないと、パイプラインは予期したとおり失敗します。



5 ユーザーの承認を待機しているパイプライン ステージを表示するには、ステージのステータス アイコンをクリックします。

6 タスクの詳細を表示するには、タスクをクリックします。

必要なユーザーがタスクを承認した後、適切なロールを持つユーザーがパイプラインを再開する必要があります。必要なロールについては、 Code Stream でユーザー アクセスと承認を管理する方法を参照してください。

実行が失敗した場合は、障害の原因をトリアージして修正する必要があります。次に、実行に移動して、[アクション] - [再実行] をクリックします。

プライマリ パイプラインの実行とネストされた実行を再開できます。

7 パイプラインの実行から、[アクション] をクリックしてパイプラインを表示し、[一時停止]、[キャンセル] などのアクションを選択できます。パイプラインの実行が進行中の場合は、管理者としてパイプラインの実行を削除したり、同期したりできます。管理者以外のユーザーである場合は、実行中のパイプラインを削除できます。

8 実行と実行の間を簡単に移動してタスクの詳細を表示するには、[実行] をクリックし、パイプライン実行をクリックします。次に、上部のタブをクリックし、パイプライン実行を選択します。

結果

完了です。パイプラインを実行し、パイプラインの実行を確認して、パイプラインの実行を継続するために承認が必要なユーザー操作タスクを確認しました。また、必要な変更を行うために、パイプラインの実行で [アクション] メニューを使用してパイプライン モデルに戻りました。

次のステップ

Code Stream を使用してソフトウェアのリリース サイクルを自動化する方法の詳細については、5 章 Code Stream を使用するためのチュートリアルを参照してください。

Code Stream で使用可能なタスクのタイプ

パイプラインを構成するときは、必要なアクションに対してパイプラインが実行する特定タイプのタスクを追加します。各タスク タイプは別のアプリケーションと連携し、ユーザーのアプリケーションをビルド、テスト、配信するときにパイプラインを有効にします。

展開のためにリポジトリからアーティファクトを取得する必要がある場合でも、リモート スクリプトを実行する必要がある場合でも、チーム メンバーからユーザー操作の承認を得る必要がある場合でも、 Code Stream ではパイプライン実行のために、ユーザーに適したタイプのタスク タイプを用意しています。

Code Stream では、さまざまなタイプのタスクでパイプラインの実行をキャンセルできます。パイプラインの実行で [キャンセル] をクリックすると、タスク、ステージ、またはパイプライン全体がキャンセル状態になり、パイプラインの実行がキャンセルされます。

次のタスクを使用しているとき、 Code Stream によってタスク、ステージ、またはパイプライン全体でパイプラインの実行をキャンセルできます。

- Jenkins
- SSH
- PowerShell
- ユーザー操作
- パイプライン
- クラウド テンプレート
- vRO
- ポーリング

Code Stream は、CI、カスタム統合、Kubernetes などのタスクに対するキャンセル動作をサードパーティ システムに伝達しません。 Code Stream はタスクをキャンセル済みとしてマークし、タスクが終了するのを待つことなくステータスの取得をただちに停止します。サードパーティ システムではタスクが完了することも失敗することもありますが、 Code Stream での実行は [キャンセル] をクリックするとただちに停止します。

パイプラインでタスクを使用する前に、対応するエンドポイントが使用可能であることを確認します。

表 3-2. 承認の取得または決定ポイントの設定

タスクのタイプ	機能	例と詳細
[ユーザー操作]	ユーザー操作タスクでは、いつパイプラインを実行し、承認のために停止する必要があるのかを制御する、必須の承認が有効になります。	パイプラインを実行して結果を確認する方法を参照してください。また、 Code Stream でユーザー アクセスと承認を管理する方法も参照してください。
[条件]	決定ポイントを追加します。決定ポイントでは、パイプラインの実行を継続するか、または停止するかが条件式に基づいて決定されます。条件が true の場合、パイプラインは後続のタスクを実行します。false の場合、パイプラインは停止します。	条件タスクで変数バインドを使用して、 Code Stream でパイプラインを実行または停止する方法を参照してください。

表 3-3. 継続的インテグレーションと展開の自動化

タスクのタイプ	機能	例と詳細
[クラウド テンプレート]	GitHub から自動クラウド テンプレートを展開してアプリケーションをプロビジョニングし、クラウド テンプレートの継続的インテグレーションおよび継続的デリバリ (CICD) を展開用に自動化します。	Code Stream の YAML クラウド テンプレートから展開するアプリケーションのリリースを自動化する方法を参照してください。 最初に [作成] または [更新] を選択してから [クラウド テンプレート] と [バージョン] を選択すると、クラウド テンプレートのパラメータが表示されます。次の要素を追加することにより、クラウド テンプレート タスクの入力テキスト領域で変数のバインドを使用できます。 ■ 整数 ■ 列挙文字列 ■ ブール値 ■ 配列変数 入力で変数のバインドを使用する場合は、次の例外に注意してください。列挙の場合は、固定セットから列挙値を選択する必要があります。ブール値の場合は、入力テキスト領域に値を入力する必要があります。 Cloud Assembly 内のクラウド テンプレートに入力変数がある場合、クラウド テンプレートのパラメータがクラウド テンプレート タスクに表示されます。たとえば、クラウド テンプレートに `Integer` の入力タイプが含まれている場合は、整数を直接入力するか、変数バインドを使用して変数として入力できます。
[CI]	CI タスクでは、レジストリ エンドポイントから Docker ビルド イメージを取得し、Kubernetes クラスタに展開することで、コードからパイプラインへの継続的インテグレーションが可能です。 CI タスクでは、ログの 100 行が出力として表示され、ログをダウンロードすると 500 行が表示されます。 CI タスクには、短期ポート 32768 ～ 61000 が必要です。	スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画を参照してください。
[カスタム]	カスタム タスクでは、 Code Stream が自分のビルド、テスト、展開ツールと統合されます。	ビルド、テスト、展開用の独自のツールを Code Stream に統合する方法を参照してください。

表 3-3. 継続的インテグレーションと展開の自動化 （続き）

タスクのタイプ	機能	例と詳細
[Kubernetes]	AWS 上の Kubernetes クラスタへのソフトウェア アプリケーションの展開を自動化します。	Kubernetes クラスタへの Code Stream のアプリケーションのリリースを自動化する方法を参照してください。
[パイプライン]	パイプラインをプライマリ パイプラインにネストします。パイプラインはネストされると、プライマリ パイプラインのタスクとして動作します。 ネストされたパイプラインには、プライマリ パイプラインの [タスク] タブでリンクをクリックして簡単に移動することができます。ネストされたパイプラインは、新しいブラウザ タブで開きます。	[実行] でネストされたパイプラインを検索するには、検索領域に **nested** と入力します。

表 3-4. 開発、テスト、展開アプリケーションの統合

タスク タイプ	機能...	例と詳細
[Bamboo]	展開の準備中にソフトウェアを継続的にビルド、テスト、および統合する Bamboo の継続的インテグレーション (CI) サーバと連携し、開発者が変更をコミットする際にコード ビルドをトリガします。Bamboo ビルドが生成するアーティファクトの場所を公開し、他のタスクでビルドと展開に使用するためにタスクでパラメータを出力できるようにします。	Bamboo サーバ エンドポイントに接続し、パイプラインから Bamboo ビルド プランを開始します。
[Jenkins]	ソース コードのビルドおよびテスト、テスト ケースの実行、カスタム スクリプトの使用が可能な Jenkins ジョブをトリガします。	Code Stream を Jenkins と統合する方法を参照してください。
[TFS]	パイプラインを Team Foundation Server に接続して、コードのビルドとテストを実行する構成済みのジョブを含むビルド プロジェクトの管理と呼び出しが可能になります。	Code Stream でサポートされる Team Foundation Server のバージョンについては、 Code Stream でのエンドポイントとは を参照してください。
[vRO]	vRealize Orchestrator で事前定義またはカスタム ワークフローを実行して、 Code Stream の機能を拡張します。 Code Stream は、vRealize Orchestrator の基本認証とトークンベースの認証をサポートします。 Code Stream は、API トークンを使用して vRealize Orchestrator クラスタの認証と検証を行います。トークンベースの認証では、Code Stream はクラウド拡張性プロキシを使用する vRealize Orchestrator エンドポイントをサポートします。そのため、Code Stream で、クラウド拡張性プロキシを使用する vRealize Orchestrator エンドポイントを使用してワークフローをトリガできます。	Code Stream を vRealize Orchestrator と統合する方法を参照してください。

表 3-5. API を使用した他のアプリケーションの統合

タスク タイプ	機能...	例と詳細
[REST]	Code Stream を REST API を使用する他のアプリケーションと連携して、相互に連携可能なソフトウェア アプリケーションを継続的に開発および提供できるようにします。	REST API を使用して Code Stream を他のアプリケーションと統合する方法を参照してください。
[ポーリング]	REST API を呼び出し、パイプライン タスクが終了基準を満たして完了するまでポーリングします。 Code Stream 管理者は、ポーリング数を最大 10,000 に設定できます。ポーリング間隔は 60 秒以上にする必要があります。 [失敗時に続行] チェック ボックスをオンにすると、回数または間隔がこれらの値を超えた場合でも、ポーリング タスクは引き続き実行されます。 `POLL Iteration Count`：パイプラインの実行に表示され、POLL タスクが URL からの応答を要求した回数が表示されます。たとえば、POLL 入力が 65 で、POLL 要求が実際に実行された回数が 4 の場合、パイプライン実行出力の反復回数は 4（65 件中）と表示されます。	REST API を使用して Code Stream を他のアプリケーションと統合する方法を参照してください。

表 3-6. リモートおよびユーザー定義のスクリプトの実行

タスクのタイプ	機能	例と詳細
[PowerShell]	PowerShell タスクを使用すると、 Code Stream は、リモート ホスト上でスクリプト コマンドを実行できます。たとえば、スクリプトで、テスト タスクを自動化したり、管理タイプのコマンドを実行したりできます。 スクリプトは、リモートとユーザー定義のいずれも可能です。HTTP または HTTPS を介した接続や、TLS の使用も可能です。 Windows ホストでは、`winrm` サービスが構成されている必要があります。また、`winrm` では、`MaxShellsPerUser` と `MaxMemoryPerShellMB` が構成されている必要があります。 PowerShell タスクを実行するには、リモート Windows ホストへのアクティブなセッションが必要です。 [PowerShell コマン ドラインの長さ] base64 PowerShell コマンドを入力する場合は、コマンド全体の長さを計算する必要があることに注意してください。 Code Stream パイプラインは base64 PowerShell コマンドをエンコードして、別のコマンド内にラッピングするため、コマンドの全体の長さが増加します。 PowerShell `winrm` コマンドで許可される最大長は 8192 バイトです。PowerShell タスクがエンコードおよびラッピングされている場合は、コマンド長の上限が低くなります。したがって、PowerShell コマンドを入力する前に、コマンド長を計算する必要があります。 Code Stream PowerShell タスクのコマンド長の上限は、元のコマンドを base64 でエンコードした後の長さによって決まります。コマンド長は次のように計算されます。 `3 * (length of original command / 4)) - (numberOfPaddingCharacters) + 77 (Length of Write-output command)` Code Stream のコマンド長は、上限の 8192 よりも小さくする必要があります。	`MaxShellsPerUser` と `MaxMemoryPerShellMB` を設定する場合： ■ `MaxShellsPerUser` の許容値は、各パイプラインには 5 個の PowerShell タスクがある 50 個の同時実行パイプラインには `500` です。値を設定するには、次を実行します：`winrm set winrm/config/winrs '@{MaxShellsPerUser="500"}'` ■ `MaxMemoryPerShellMB` の許容可能なメモリ値は `2048` です。値を設定するには、次を実行します：`winrm set winrm/config/winrs '@{MaxMemoryPerShellMB="2048"}'` このスクリプトは、別のパイプラインが使用できる応答ファイルに出力を書き込みます。
[SSH]	SSH タスクでは、Bash シェル スクリプト タスクがリモート ホスト上でスクリプト コマンドを実行できます。たとえば、スクリプトで、テスト タスクを自動化したり、管理タイプのコマンドを実行したりできます。 スクリプトは、リモートとユーザー定義のいずれも可能です。HTTP または HTTPS を介した接続が可能で、プライベート キーまたはパスワードが必要です。 Linux ホスト上で SSH サービスが構成されている必要があります。また、`MaxSessions` の SSHD 設定は `50` にする必要があります。 複数の SSH タスクを同時に実行する場合は、SSH ホストの `MaxSessions` と `MaxOpenSessions` を増やします。MaxSessions および MaxOpenSessions の設定を変更する必要がある場合は、vRealize Automation インスタンスを SSH ホストとして使用しないでください。	スクリプトは、リモートとユーザー定義のいずれも可能です。たとえば、スクリプトは次のようになります。 `message="Hello World" echo $message` このスクリプトは、別のパイプラインが使用できる応答ファイルに出力を書き込みます。

Code Stream パイプラインで変数のバインドを使用する方法

パイプライン タスクをバインドすることは、パイプラインの実行時にタスクの依存関係を作成することを意味します。パイプライン タスクのバインドは、いくつかの方法で作成できます。タスクを別のタスクにバインドしたり、変数と式にバインド、または条件にバインドしたりできます。

クラウド テンプレート タスクでドル記号によるバインドをクラウド テンプレート変数に適用する方法

Code Stream パイプライン クラウド テンプレート タスクでは、ドル記号によるバインドをクラウド テンプレート変数に適用できます。 Code Stream で変数を変更する方法は、クラウド テンプレートでの変数プロパティのコーディングによって異なります。

ドル記号によるバインドをクラウド テンプレート タスクで使用する必要があるにもかかわらず、クラウド テンプレート タスクで現在使用しているバージョンのクラウド テンプレートではそれができない場合は、Cloud Assembly クラウド テンプレートを変更し、新しいバージョンを展開します。次に、クラウド テンプレート タスクで新しいバージョンのクラウド テンプレートを使用し、必要に応じてドル記号によるバインドを追加します。

Cloud Assembly クラウド テンプレートで利用できるプロパティ タイプにドル記号によるバインドを適用するには、適切な権限が必要です。

- Cloud Assembly でクラウド テンプレートの展開を作成したユーザーと同じロールを持っていることが必要です。
- パイプラインをモデル化したユーザーとパイプラインを実行したユーザーが別々のユーザーで、ロールも異なる場合があります。
- 開発者が Code Stream の実行者ロールを持ち、パイプラインをモデル化する場合、その開発者は、クラウド テンプレートを展開したユーザーと同じ Cloud Assembly ロールも持っている必要があります。たとえば、Cloud Assembly 管理者のロールが必要な場合があります。
- パイプラインの作成および展開の作成を行えるのは、パイプラインをモデル化したユーザーのみです。これは、必要な権限を持っているためです。

クラウド テンプレート タスクで API トークンを使用するには、次の操作を実行します。

- パイプラインをモデル化したユーザーは、 Code Stream 実行者ロールを持つ別のユーザーに API トークンを提供できます。実行者がパイプラインを実行する場合、API トークンと、API トークンで作成された認証情報が使用されます。
- ユーザーがクラウド テンプレート タスクで API トークンを入力すると、パイプラインで必要な認証情報が作成されます。
- API トークンの値を暗号化するには、[変数の作成] をクリックします。
- API トークンの変数を作成せずに、API トークンをクラウド テンプレート タスクで使用する場合、API トークンの値は、プレーン テキストで表示されます。

クラウド テンプレート タスクでドル記号によるバインドをクラウド テンプレート変数に適用するには、次の手順を実行します。

入力変数プロパティ（integerVar、stringVar、flavorVar、BooleanVar、objectVar、arrayVar など）が定義されているクラウド テンプレートを基礎として使用します。resources セクションで定義されているイメージ プロパティを確認できます。クラウド テンプレート コードのプロパティは、次のようになります。

```
formatVersion: 1
inputs:
  integerVar:
    type: integer
    encrypted: false
    default: 1
  stringVar:
    type: string
    encrypted: false
    default: bkix
  flavorVar:
    type: string
    encrypted: false
    default: medium
  BooleanVar:
    type: boolean
    encrypted: false
    default: true
  objectVar:
    type: object
    encrypted: false
    default:
      bkix2: bkix2
  arrayVar:
    type: array
    encrypted: false
    default:
      - '1'
      - '2'
resources:
  Cloud_Machine_1:
    type: Cloud.Machine
    properties:
      image: ubuntu
      flavor: micro
      count: '${input.integerVar}'
```

image および flavor には、ドル記号の変数 ($) を使用できます。例：

```
resources:
  Cloud_Machine_1:
    type: Cloud.Machine
    properties:
      input: '${input.image}'
      flavor: '${input.flavor}'
```

Code Stream パイプラインでクラウド テンプレートを使用してドル記号によるバインドを追加するには、次の手順を実行します。

1 Code Stream で、[パイプライン] - [空白のキャンバス] の順にクリックします。

2 パイプラインに [クラウド テンプレート] タスクを追加します。

3 クラウド テンプレート タスクで、[クラウド テンプレート ソース] として [Cloud Assembly クラウド テンプレート] を選択し、クラウド テンプレート名を入力して、クラウド テンプレートのバージョンを選択します。

4 パイプラインの認証情報を提供する API トークンを入力できます。クラウド テンプレート タスクで API トークンを暗号化する変数を作成するには、[変数の作成] をクリックします。

5 表示される [パラメータと値] テーブルで、パラメータの値を確認します。`flavor` のデフォルト値は `small`、`image` のデフォルト値は `ubuntu` です。

6 Cloud Assembly のクラウド テンプレートを変更する必要があるとします。その場合は、たとえば、次のような操作を行います。

   a `array` タイプのプロパティを使用するように `flavor` を設定します。Cloud Assembly で `Flavor` に対してコンマ区切り値を使用できるのは、タイプが **`array`** の場合です。

   b [[展開]] をクリックします。

   c [展開タイプ] 画面で、展開名を入力し、クラウド テンプレートのバージョンを選択します。

   d [展開の入力] 画面では、`Flavor` の値を 1 つ以上定義できます。

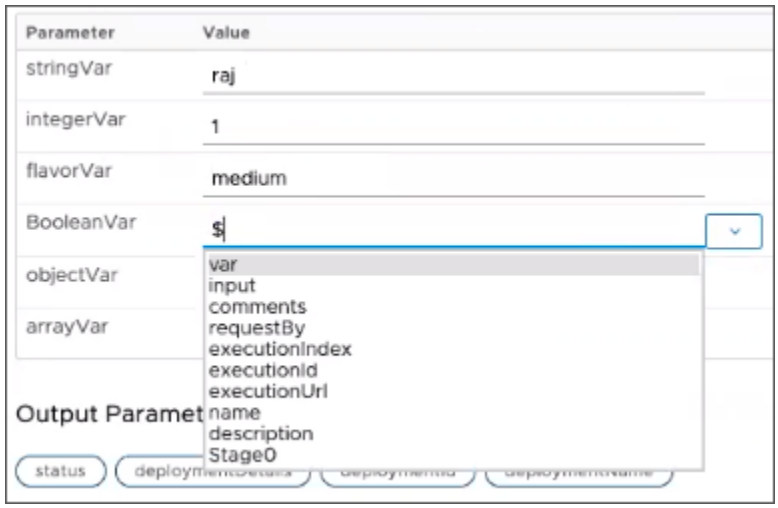
   e [展開の入力] には、クラウド テンプレート コードで定義されているすべての変数が含まれています。[展開の入力] は、クラウド テンプレート コードで定義されているとおりに表示されます。たとえば、`Integer Var`、`String Var`、`Flavor Var`、`Boolean Var`、`Object Var`、`Array Var` などがあります。`String Var` と `Flavor Var` は文字列の値で、`Boolean Var` はチェックボックスです。

   f [[展開]] をクリックします。

7 Code Stream で新しいバージョンのクラウド テンプレートを選択して、[パラメータと値] テーブルに値を入力します。クラウド テンプレートでは、次のタイプのパラメータがサポートされます。これらのパラメータにより、ドル記号変数を使用して Code Stream バインドを有効にできます。 Code Stream クラウド テンプレート タスクのユーザー インターフェイスと Cloud Assembly クラウド テンプレートのユーザー インターフェイスには、若干の違いがあります。Cloud Assembly のクラウド テンプレートのコーディングによっては、Code Stream のクラウド テンプレート タスクで値を入力できない場合があります。

   a [flavorVar] の場合、クラウド テンプレートで定義されているタイプが文字列または配列であるときは、文字列またはコンマ区切り値の配列を入力します。配列は、たとえば **`test, test`** のように入力します。

b [BooleanVar] の場合は、ドロップダウン メニューで [true] または [false] を選択します。変数のバインドを使用するには、**$** を入力して、リストから変数のバインドを選択します。



c [objectVar] の場合は、入力する値を中括弧と引用符で囲みます（**{"bkix":"bkix":}** のような形式になります）。

d クラウド テンプレートに渡された [objectVar] は、そのクラウド テンプレートに基づいたさまざまな方法で使用できます。JSON オブジェクトに対して文字列形式を使用できるため、キーと値のペアをコンマで区切ってキーと値のテーブルに追加できます。JSON オブジェクトには、プレーン テキストを入力することも、JSON の標準文字列形式であるキーと値のペアを入力することもできます。

e [arrayVar] の場合は、コンマ区切り入力値を配列として入力します（**["1","2"]** のような形式になります）。

8 パイプラインで、入力パラメータを配列にバインドできます。

a [入力] タブをクリックします。

b 入力の名前を入力します。たとえば、**arrayInput** のように入力します。

c [パラメータと値] テーブルで [arrayVar] をクリックして、**${input.arrayInput}** と入力します。

d パイプラインを保存して有効にした後、パイプラインを実行するときに配列の入力値を指定する必要があります。たとえば、**["1","2"]** と入力して [実行] をクリックします。

ここまで、 Code Stream パイプライン クラウド テンプレート タスクでドル記号 ($) 変数のバインドをクラウド テンプレート内で使用する方法について説明しました。

パイプラインの実行時にパラメータをパイプラインに渡す方法

パイプラインに入力パラメータを追加すると、Code Stream が入力パラメータをパイプラインに渡すことができます。この場合、ユーザーは、パイプラインの実行時、入力パラメータの値を入力する必要があります。パイプラインに出力パラメータを追加すると、パイプライン タスクは、タスクからの出力値を使用できます。 Code Stream では、独自のパイプライン ニーズに対応したさまざまな方法で各種のパラメータを使用することがサポートされています。

たとえば、REST タスクを含むパイプラインの実行時に Git サーバの URL を入力するためのプロンプトを表示するには、REST タスクを Git サーバの URL にバインドします。

変数のバインドを作成するには、REST タスクに URL バインド変数を追加します。パイプラインが実行され、REST タスクに到達すると、ユーザーは Git サーバの URL の入力を求められます。バインドを作成するには、次の手順を実行します。

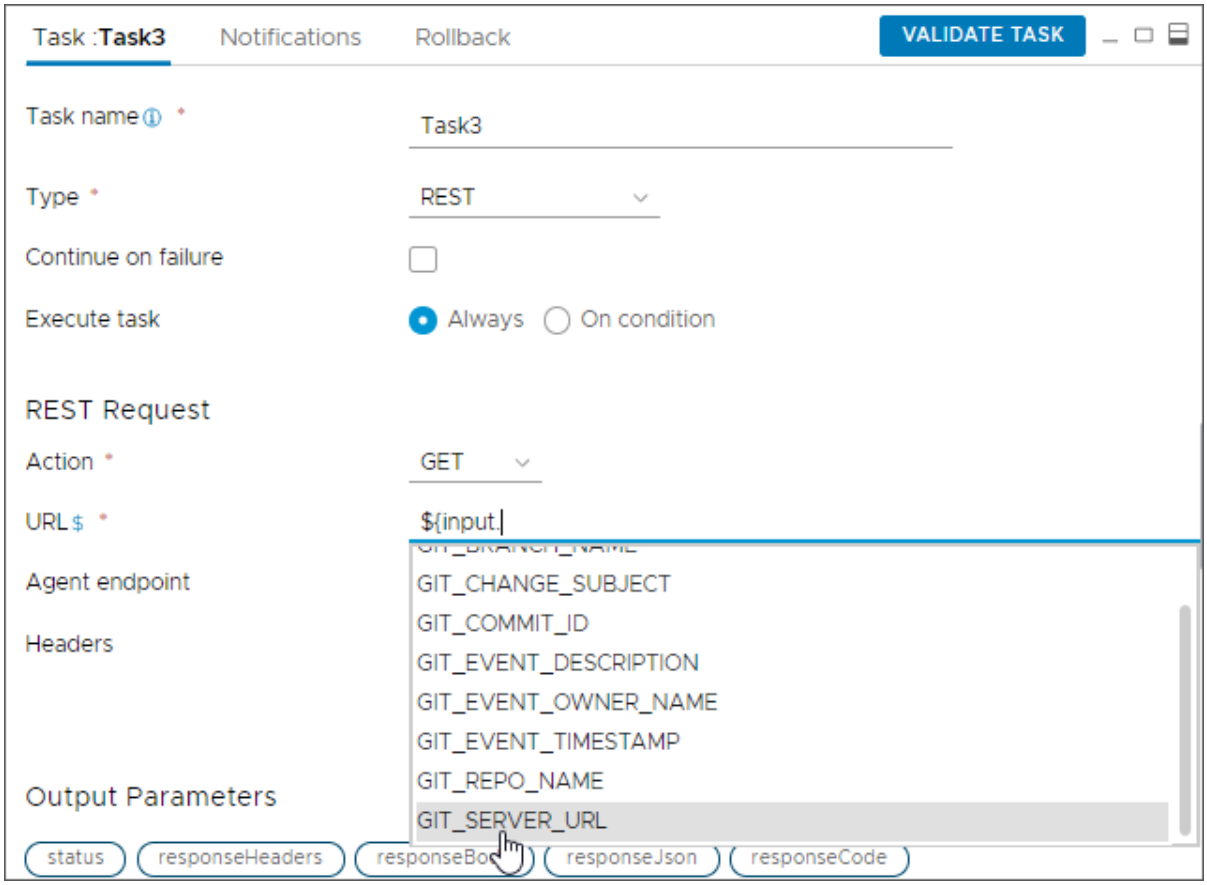
1 パイプラインで、[入力] タブをクリックします。

2 パラメータを設定するには、[パラメータの自動挿入] で [Git] をクリックします。

Git パラメータのリストが表示されます。このリストに [GIT_SERVER_URL] が含まれています。Git サーバの URL にデフォルト値を使用する必要がある場合は、このパラメータを編集します。

3 [モデル] をクリックし、REST タスクをクリックします。

4 [タスク] タブの [URL] 領域に **$** と入力し、[input] を選択してから [GIT_SERVER_URL] を選択します。



エントリは [${input.GIT_SERVER_URL}] のようになります。

5 タスクの変数バインドの整合性を確認するには、[タスクの検証] をクリックします。

Code Stream は、タスクが正常に検証されたことを示します。

6 パイプラインが REST タスクを実行するときに、ユーザーは Git サーバの URL の入力を求められます。そうしないと、タスクの実行が終了しません。

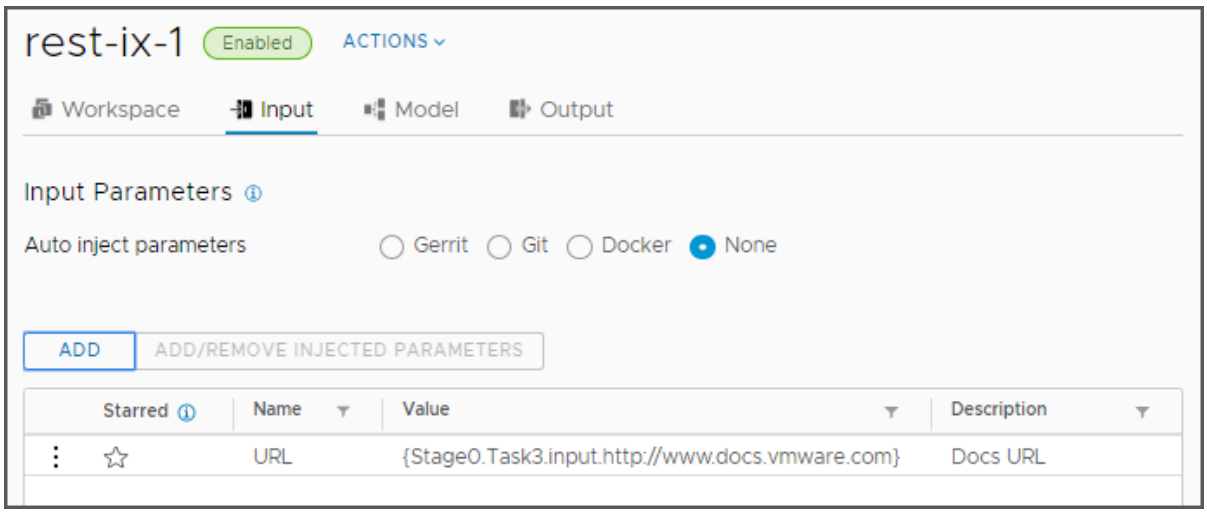
入力および出力パラメータを作成して 2 つのパイプライン タスクをバインドする方法

タスクをまとめてバインドするときは、受信タスクの入力設定にバインド変数を追加します。その後、パイプラインが実行されたら、バインド変数を必要な入力に置き換えます。

パイプライン タスクをまとめてバインドするには、入力パラメータと出力パラメータでドル記号変数 ($) を使用します。その方法を次の例で示します。

パイプラインで REST タスクの URL を呼び出し、応答を出力する必要があるとします。URL を呼び出して応答を出力するには、REST タスクに入力パラメータと出力パラメータの両方を含める必要があります。また、タスクを承認できるユーザーも必要です。ユーザー操作タスクを含めることで、パイプラインの実行時に別のユーザーがタスクを承認できるようにします。この例は、入力パラメータと出力パラメータで式を使用し、パイプラインにタスクの承認を待機させる方法を示しています。

1 パイプラインで、[入力] タブをクリックします。



2 [パラメータの自動挿入] は [なし] のままにします。

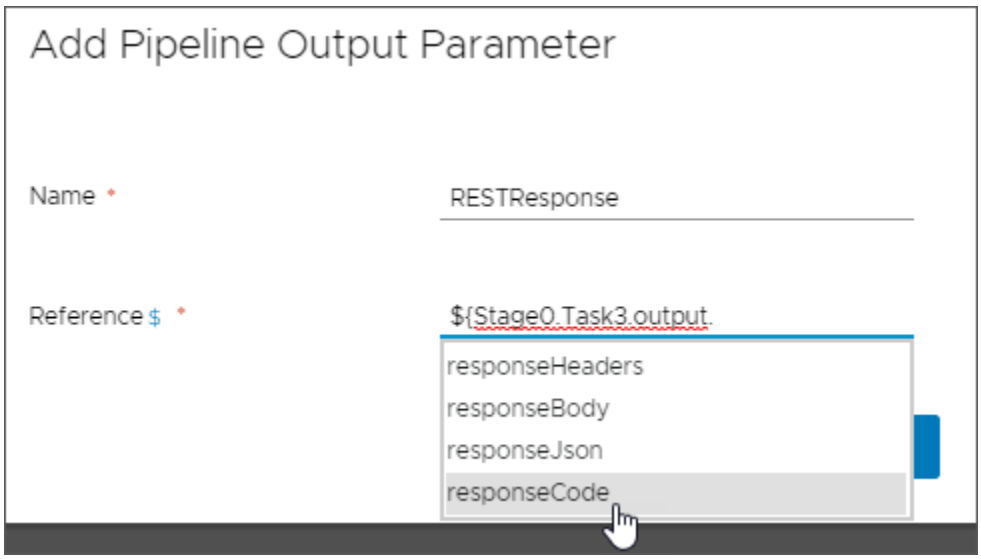
3 [追加] をクリックし、パラメータ名、値、および説明を入力してから、[OK] をクリックします。例：

   a URL 名を入力します。

   b 値として `{Stage0.Task3.input.http://www.docs.vmware.com}` を入力します。

   c 説明を入力します。

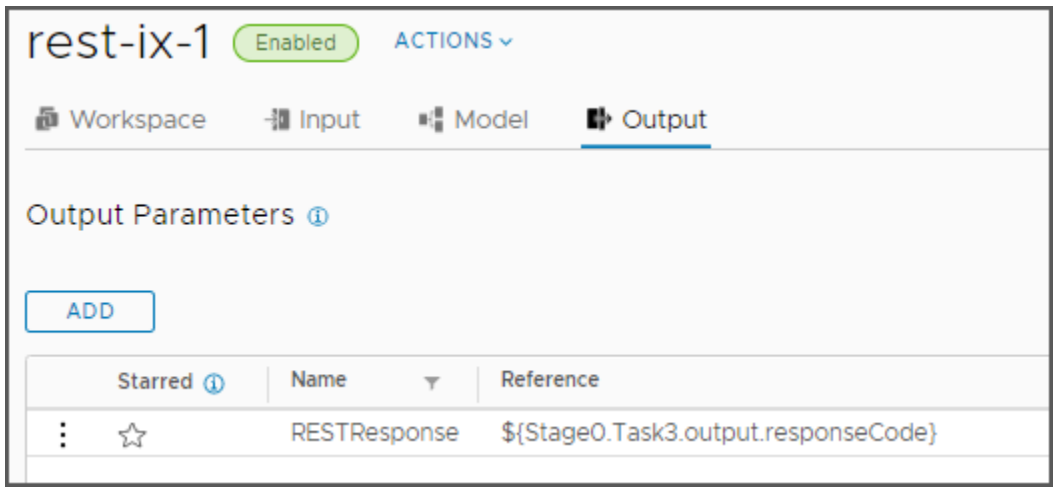
4 [出力] タブをクリックし、[追加] をクリックして、出力パラメータ名とマッピングを入力します。



a 一意の出力パラメータ名を入力します。

b [リファレンス] 領域をクリックし、$ と入力します。

c ポップアップ表示されたオプションを選択して、タスク出力マッピングを入力します。[Stage0]、[Task3]、[output]、[responseCode] の順に選択します。[OK] をクリックします。

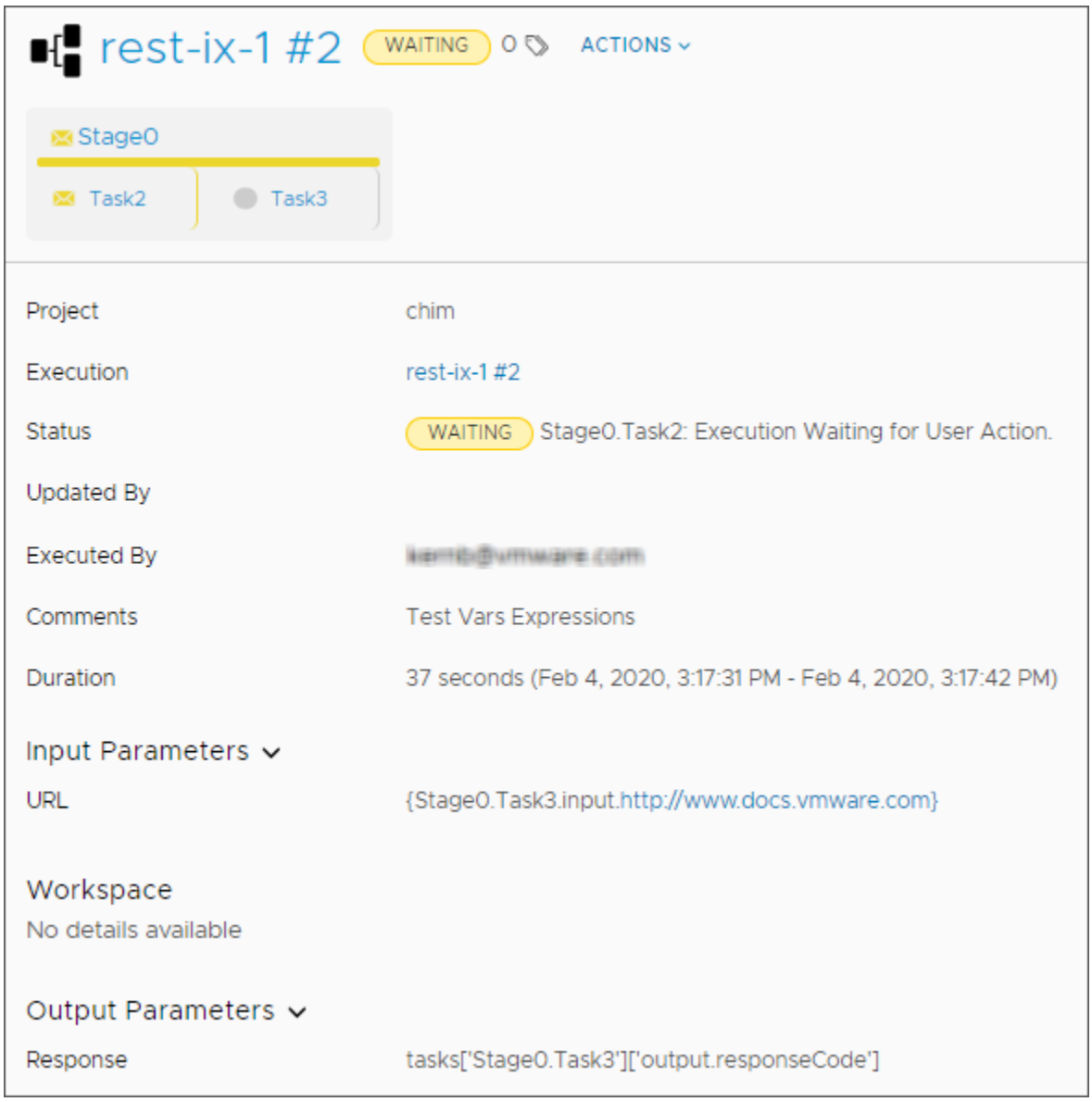


5 パイプラインを保存します。

6 [アクション] メニューから [実行] をクリックします。

7 [アクション] - [実行の表示] の順にクリックします。

8 パイプラインの実行をクリックし、定義した入力パラメータと出力パラメータを確認します。

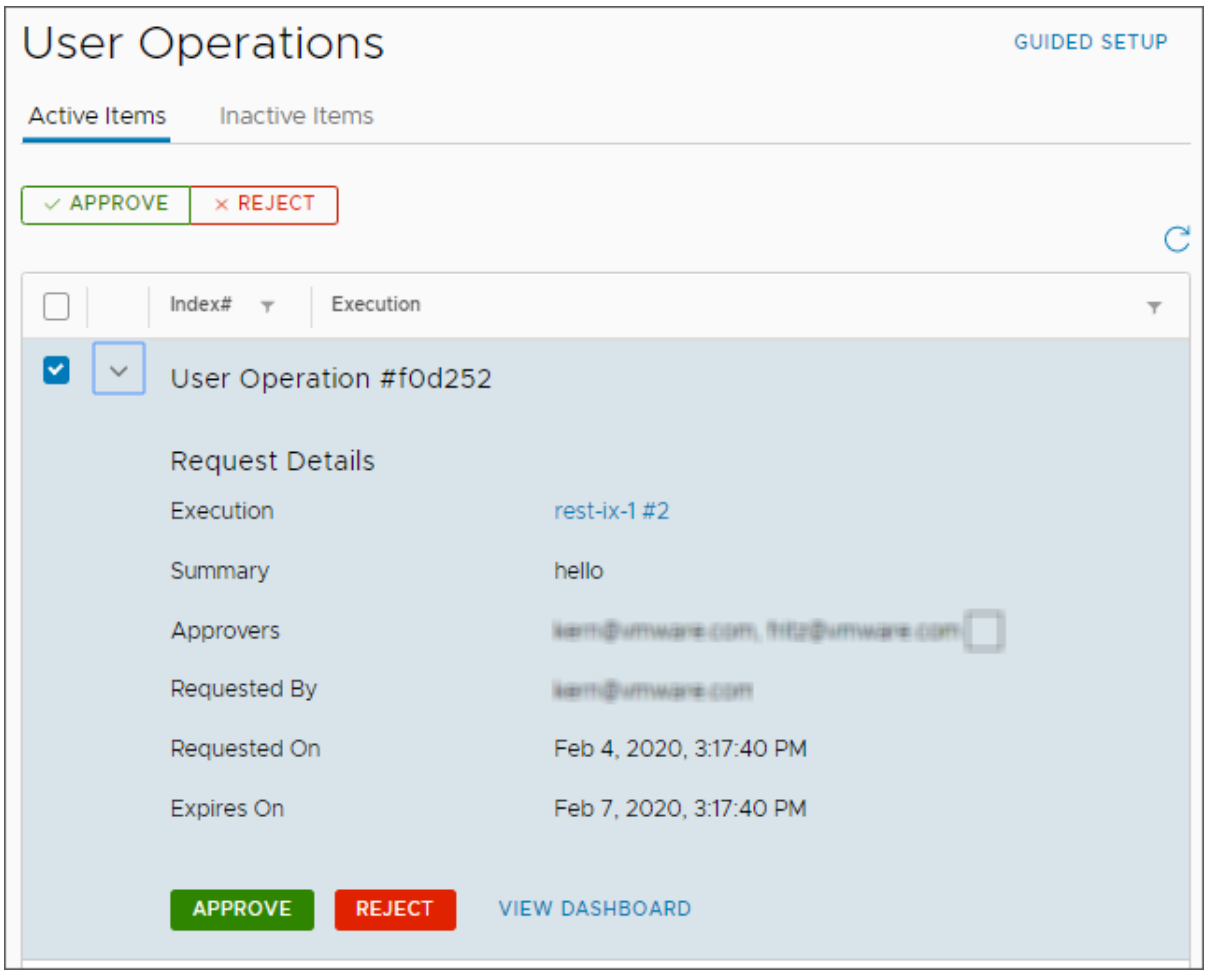


9 パイプラインを承認するには、[ユーザー操作] をクリックし、[アクティブなアイテム] タブで承認のリストを表示します。あるいは、[実行] に留まったままタスクをクリックし、[承認] をクリックします。

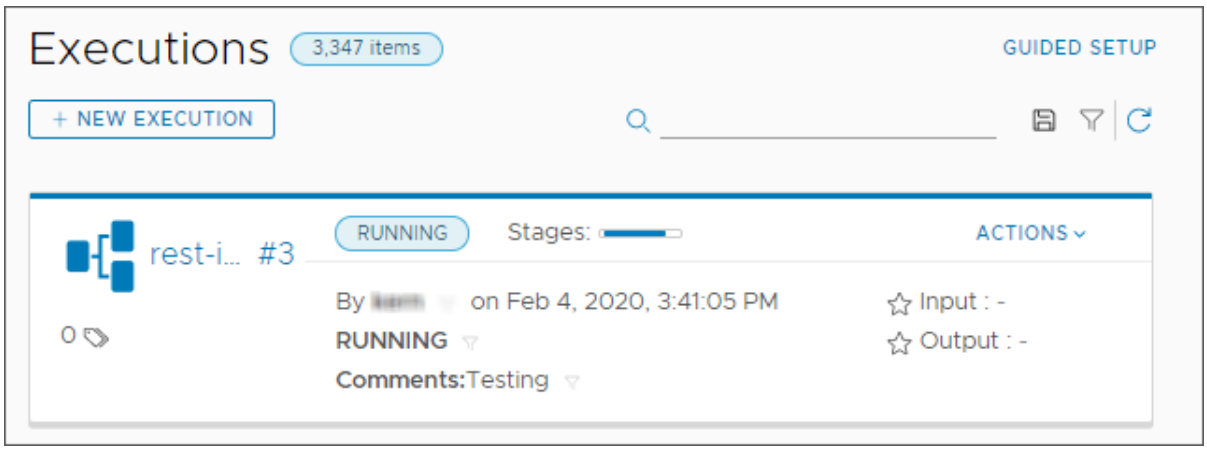
10 [承認] ボタンと [拒否] ボタンを有効にするには、実行の横にあるチェック ボックスをクリックします。

11 詳細を表示するには、ドロップダウンの矢印を展開します。

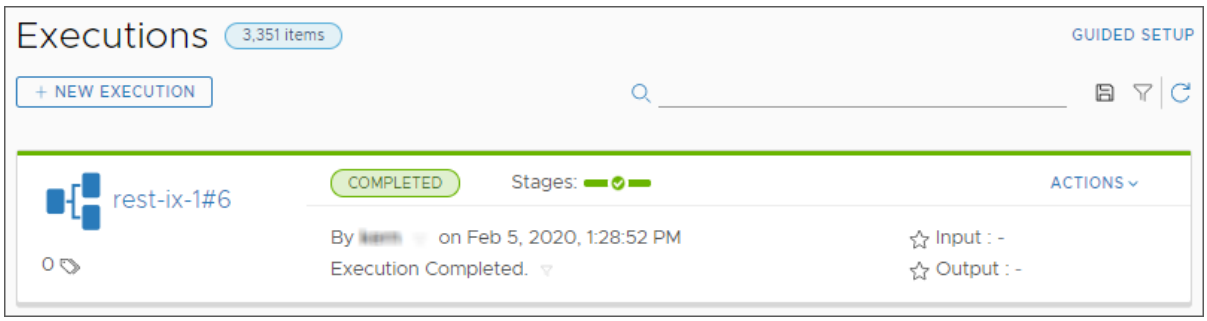
12 タスクを承認するには、[承認] をクリックし、理由を入力して [OK] をクリックします。



13 [実行] をクリックし、パイプラインが続行されることを確認します。



14 パイプラインが失敗した場合は、エラーを修正してからパイプラインを保存し、再度実行します。



変数と式の詳細情報

パイプライン タスクをバインドするときに変数と式を使用する方法の詳細については、 Code Stream でパイプライン タスクをバインドするときに使用できる変数と式を参照してください。

条件変数バインドでパイプライン タスクの出力を使用する方法については、条件タスクで変数バインドを使用して、Code Stream でパイプラインを実行または停止する方法を参照してください。

条件タスクで変数バインドを使用して、 Code Stream でパイプラインを実行または停止する方法

パイプラインのタスクの出力で、指定した条件に基づいてパイプラインの実行または停止を決定できます。タスクの出力に基づいてパイプラインの動作を決めるには、[条件] タスクを使用します。

パイプラインで、[条件] タスクを決定ポイントとして使用します。指定した条件式で [条件] タスクを実行すると、パイプライン、ステージ、およびタスクのプロパティを評価できます。

[条件] タスクの結果によって、パイプラインで次のタスクを実行するかどうかが決まります。

- 条件が True の場合、パイプラインの実行を続行できます。
- 条件が False の場合、パイプラインが停止します。

[条件] タスクとともに複数のタスクをバインドした 1 つのタスクの出力値を、次のタスクの入力として使用する方法については、 Code Stream パイプラインで変数のバインドを使用する方法を参照してください。

表 3-7. [条件] タスクおよびその条件式とパイプラインとの関連

条件タスク	影響を受ける対象	機能
条件タスク	パイプライン	[条件] タスクでは、タスクの出力が True か False かに応じて、パイプラインをその時点で実行または停止するかどうかが決まります。
条件式	[条件] タスクの出力	パイプラインを実行すると、[条件] タスクに含まれる条件式で、True または False の出力ステータスが返されます。たとえば、条件式で、[条件] タスクの出力ステータスを完了にすることや、ビルド番号 74 にする必要があるとします。 条件式は、次のように [条件] タスクの [タスク] タブに表示されます。 

[条件] タスクの機能と動作は、他のタイプのタスクでの [条件に基づく] 設定とは異なります。

他のタイプのタスクの [条件に基づく] では、前提条件式の評価が True であるか False であるかに基づいて、後続のタスクではなく現在のタスクを実行するかどうかが判断されます。[条件に基づく] 設定の条件式は、パイプラインの実行時に現在のタスクの出力ステータスとして True または False を返します。[条件に基づく] 設定は、その条件式とともに [タスク] タブに表示されます。

この例では、[条件] タスクを使用します。

前提条件

- パイプラインが配置されており、ステージとタスクが含まれていることを確認します。

手順

1 パイプラインで、[条件] タスクを表示する必要がある決定ポイントを決めます。

2 ステータスが成功か失敗かに依存するタスクの前に、[条件] タスクを追加します。

3 [条件] タスクに条件式を追加します。

例：
```
"${Stage1.task1.output.status}" == "COMPLETED" || ${input.buildNumber} == 74
```

4 タスクを検証します。

5 パイプラインを保存し、有効にして実行します。

結果

パイプラインの実行を監視し、[条件] タスクでパイプラインの継続実行または停止を確認します。

次のステップ

パイプライン展開をロールバックする場合にも、[条件] タスクを使用できます。たとえば、ロールバック パイプラインでは、[条件] タスクによって、 Code Stream が条件式に基づいてパイプラインの障害をマーク付けし、さまざまな障害タイプに対して 1 つのロールバック フローをトリガできます。

デプロイをロールバックする場合は、 Code Stream で展開をロールバックする方法を参照してください。

Code Stream でパイプライン タスクをバインドするときに使用できる変数と式

変数と式により、パイプライン タスクで入力パラメータと出力パラメータを使用できます。入力するパラメータにより、パイプライン タスクが 1 つ以上の変数、式、または条件にバインドされ、実行時のパイプラインの動作が決定されます。

パイプラインではシンプルなソフトウェア配信ソリューションから複雑なソフトウェア配信ソリューションまで実行可能

パイプライン タスクをバインドするときには、デフォルトの式と複雑な式を含めることができます。このため、パイプラインでは、シンプルなソフトウェア配信ソリューションを実行することも、複雑なソフトウェア配信ソリューションを実行することもできます。

パイプラインにパラメータを作成するには、[入力] または [出力] タブをクリックし、ドル記号 **$** と式を入力して変数を追加します。たとえば、このパラメータは、URL を呼び出すタスクの入力として使用されます：`${Stage0.Task3.input.URL}`。

変数のバインドの形式では、SCOPE（範囲）と KEY（キー）と呼ばれる構文コンポーネントを使用します。`SCOPE` はコンテキストを入力または出力として定義し、`KEY` は詳細を定義します。パラメータの例 `${Stage0.Task3.input.URL}` では、`SCOPE` は `input`、`KEY` は URL です。

いずれのタスクの出力プロパティも、任意のレベル数にネストされた変数バインドとして解決できます。

パイプラインで変数のバインドを使用する方法については、Code Stream パイプラインで変数のバインドを使用する方法を参照してください。

範囲とキーを含むドル記号の式を使用したパイプライン タスクのバインド

ドル記号変数の式を使用してパイプライン タスクをバインドできます。式は、`${SCOPE.KEY.<PATH>}` という形式で入力します。

パイプライン タスクの動作を決定するために、各式の `SCOPE` は Code Stream が使用するコンテキストとなります。この範囲に対して、タスクで実行されるアクションの詳細を定義する `KEY` が検索されます。`KEY` の値がネストされたオブジェクトの場合は、オプションの `PATH` を指定できます。

これらの例では、`SCOPE` と `KEY` について説明し、パイプラインでこれらを使用する方法を示します。

表 3-8. SCOPE と KEY の使用

SCOPE	式の目的と例	KEY	パイプラインで SCOPE と KEY を使用する方法

[input]	パイプラインの入力プロパティ：
	`${input.input1}`
		入力プロパティの名前	タスク内のパイプラインの入力プロパティを参照するには、次の形式を使用します。
			`tasks:`
			`  mytask:`
			`  type: REST`
			`  input:`
			`  url: ${input.url}`
			`  action: get`
			
			`input:`
			`  url: https://www.vmware.com`

[output]	パイプラインの出力プロパティ：
	`${output.output1}`
		出力プロパティの名前	通知を送信するための出力プロパティを参照するには、次の形式を使用します。
			`notifications:`
			`  email:`
			`  - endpoint: MyEmailEndpoint`
			`  subject: "Deployment Successful"`
			`  event: COMPLETED`
			`  to:`
			`  - user@example.org`
			`  body: |`
			`  Pipeline deployed the service successfully. Refer ${output.serviceURL}`

表 3-8. SCOPE と KEY の使用 （続き）

SCOPE	式の目的と例	KEY	パイプラインで SCOPE と KEY を使用する方法

[task input]	タスクへの入力：
	`${MY_STAGE.MY_TASK.input.SOMETHING}`
		通知内にタスクの入力を示します	開始された Jenkins ジョブからは、タスクの入力からトリガーされたジョブの名前を参照できます。この場合、次の形式で通知を送信します。
			`notifications:`
			`  email:`
			`  - endpoint: MyEmailEndpoint`
			`  stage: MY_STAGE`
			`  task: MY_TASK`
			`  subject: "Build Started"`
			`  event: STARTED`
			`  to:`
			`  - user@example.org`
			`  body: |`
			`  Jenkins job ${MY_STAGE.MY_TASK.input.job} started for commit id ${input.COMMITID}.`

[task output]	タスクの出力：
	`${MY_STAGE.MY_TASK.output.SOMETHING}`
		後続のタスクでのタスクの出力を示します	パイプライン タスク 2 からパイプライン タスク 1 の出力を参照するには、次の形式を使用します。
			`taskOrder:`
			`  - task1`
			`  - task2`
			`tasks:`
			`  task1:`
			`  type: REST`
			`  input:`
			`  action: get`
			`  url: https://www.example.org/api/status`
			`  task2:`
			`  type: REST`
			`  input:`
			`  action: post`
			`  url: https://status.internal.example.org/api/activity`
			`  payload: ${MY_STAGE.task1.output.responseBody}`

表 3-8. SCOPE と KEY の使用 （続き）

SCOPE	式の目的と例	KEY	パイプラインで SCOPE と KEY を使用する方法

[var]	変数：
	`${var.myVariable}`
		エンドポイント内の変数の参照	パスワードに関してエンドポイント内のシークレット変数を参照するには、次の形式を使用します。
			`---`
			`project: MyProject`
			`kind: ENDPOINT`
			`name: MyJenkinsServer`
			`type: jenkins`
			`properties:`
			`url: https://jenkins.example.com`
			`username: jenkinsUser`
			`password: ${var.jenkinsPassword}`

[var]	変数：
	`${var.myVariable}`
		パイプライン内の変数の参照	パイプライン URL 内の変数を参照するには、次の形式を使用します。
			`tasks:`
			`task1:`
			`type: REST`
			`input:`
			`action: get`
			`url: ${var.MY_SERVER_URL}`

[task status]	タスクのステータス：
	`${MY_STAGE.MY_TASK.status}`
	`${MY_STAGE.MY_TASK.statusMessage}`		

[stage status]	ステージのステータス：
	`${MY_STAGE.status}`
	`${MY_STAGE.statusMessage}`		

デフォルトの式

パイプラインでは、式に変数を使用できます。このサマリに、使用できるデフォルトの式を示します。

式	説明
${comments}	パイプラインの実行要求時に提供されるコメント。
${duration}	パイプラインの実行に要した時間。
${endTime}	完了した場合は、パイプライン実行の終了時刻 (UTC)。
${executedOn}	開始時刻と同じ。パイプライン実行が開始された時刻 (UTC)。
${executionId}	パイプライン実行の ID。
${executionUrl}	ユーザー インターフェイス内のパイプライン実行に移動する URL。
${name}	パイプラインの名前。
${requestBy}	タスクを申請したユーザーの名前。
${stageName}	ステージの範囲で使用される、現在のステージの名前。
${startTime}	パイプライン実行の開始時刻 (UTC)。
${status}	実行のステータス。
${statusMessage}	パイプライン実行のステータス メッセージ。
${taskName}	タスクの入力または通知で使用される、現在のタスクの名前。

パイプライン タスクでの SCOPE と KEY の使用

式は、サポートされているパイプライン タスクのいずれでも使用できます。次の例では、SCOPE と KEY の定義方法を示し、構文を確認します。コード例では、パイプライン ステージとタスク名として MY_STAGE と MY_TASK を使用しています。

使用可能なタスクの詳細については、 Code Stream で使用可能なタスクのタイプを参照してください。

表 3-9. タスクのゲート

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
[ユーザー操作]			
	Input	summary：ユーザー操作の申請のサマリ description：ユーザー操作の申請の説明 approvers：承認者のメール アドレスのリスト。各エントリは、カンマを使用した変数にすることができます。また、個々のメール アドレスをセミコロンで区切ることができます approverGroups：プラットフォームと ID の承認者グループ アドレスのリスト sendemail：true に設定されている場合、オプションで、申請または応答時に E メール通知を送信します expirationInDays：申請の有効期限を表す日数	${MY_STAGE.MY_TASK.input.summary} ${MY_STAGE.MY_TASK.input.description} ${MY_STAGE.MY_TASK.input.approvers} ${MY_STAGE.MY_TASK.input.approverGroups} ${MY_STAGE.MY_TASK.input.sendemail} ${MY_STAGE.MY_TASK.input.expirationInDays}
	Output	index：申請を表す 6 桁の 16 進数の文字列 respondedBy：ユーザー操作を承認または拒否したユーザーのアカウント名 respondedByEmail：応答者のメール アドレス comments：応答時に返されるコメント	${MY_STAGE.MY_TASK.output.index} ${MY_STAGE.MY_TASK.output.respondedBy} ${MY_STAGE.MY_TASK.output.respondedByEmail} ${MY_STAGE.MY_TASK.output.comments}
[条件]			
	Input	condition：評価する条件。条件が true に評価されるとタスクは完了とマークされ、それ以外の応答ではタスクは失敗します	${MY_STAGE.MY_TASK.input.condition}
	Output	result：評価による結果	${MY_STAGE.MY_TASK.output.response}

表 3-10. パイプライン タスク

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
[パイプライン]			
	Input	`name`：実行するパイプラインの名前 `inputProperties`：ネストされたパイプライン実行に渡す入力プロパティ	`${MY_STAGE.MY_TASK.input.name}` `${MY_STAGE.MY_TASK.input.inputProperties}` # すべてのプロパティを参照 `${MY_STAGE.MY_TASK.input.inputProperties.input1}` # input1 の値を参照

	Output	`executionStatus`：パイプライン実行のステータス
		`executionIndex`：パイプライン実行のインデックス
		`outputProperties`：パイプライン実行の出力プロパティ
			`${MY_STAGE.MY_TASK.output.executionStatus}`
			`${MY_STAGE.MY_TASK.output.executionIndex}`
			`${MY_STAGE.MY_TASK.output.outputProperties}` # すべてのプロパティを参照
			`${MY_STAGE.MY_TASK.output.outputProperties.output1}` # output1 の値を参照

表 3-11. 継続的インテグレーション タスクの自動化

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
[CI]			

	Input	`steps`：実行するコマンドを表す一連の文字列
		`export`：手順の実行後も保持する環境変数
		`artifacts`：共有パスに保持するアーティファクトのパス
		`process`：JUnit、JaCoCo、Checkstyle、FindBugs 処理の構成要素のセット
			`${MY_STAGE.MY_TASK.input.steps}`
			`${MY_STAGE.MY_TASK.input.export}`
			`${MY_STAGE.MY_TASK.input.artifacts}`
			`${MY_STAGE.MY_TASK.input.process}`
			`${MY_STAGE.MY_TASK.input.process[0].path}` # 最初の構成のパスを参照

	Output	`export`：入力 `exports` からエクスポートされる環境変数を表すキーと値のペア
		`artifacts`：正常に保存されたアーティファクトへのパス
		`process`：入力 `processResponse` の処理結果のセット
			`${MY_STAGE.MY_TASK.output.exports}` # すべてのエクスポートを参照
			`${MY_STAGE.MY_TASK.output.exports.myvar}` # **`myvar`** の値を参照
			`${MY_STAGE.MY_TASK.output.artifacts}`
			`${MY_STAGE.MY_TASK.output.processResponse}`
			`${MY_STAGE.MY_TASK.output.processResponse[0].result}` # 最初のプロセス構成の結果

[カスタム]			

表 3-11. 継続的インテグレーション タスクの自動化 （続き）

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法

	Input	`name`：カスタム統合の名前
		`version`：カスタム統合のバージョン、リリース済みまたは廃止済み
		`properties`：カスタム統合に送信するプロパティ
			`${MY_STAGE.MY_TASK.input.name}`
			`${MY_STAGE.MY_TASK.input.version}`
			`${MY_STAGE.MY_TASK.input.properties}` # すべてのプロパティを参照
			`${MY_STAGE.MY_TASK.input.properties.property1}` # property1 の値を参照

	Output	`properties`：カスタム統合の応答からの出力プロパティ	`${MY_STAGE.MY_TASK.output.properties}` # すべてのプロパティを参照 `${MY_STAGE.MY_TASK.output.properties.property1}` # property1 の値を参照

表 3-12. 継続的展開タスクの自動化：クラウド テンプレート

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
[クラウド テンプレート]			
	Input	`action`：[createDeployment]、[updateDeployment]、[deleteDeployment]、[rollbackDeployment] のいずれか `blueprintInputParams`: [展開の作成] アクションおよび [展開の更新] アクションで使用されます。 `allowDestroy`：更新の展開プロセスでマシンを破棄できます。 [CREATE_DEPLOYMENT] ■ `blueprintName`：クラウド テンプレートの名前 ■ `blueprintVersion`：クラウド テンプレートのバージョン または ■ `fileUrl`：Git サーバを選択した後の、リモート クラウド テンプレート YAML の URL。 [UPDATE_DEPLOYMENT] 次のいずれかの組み合わせ： ■ `blueprintName`：クラウド テンプレートの名前 ■ `blueprintVersion`：クラウド テンプレートのバージョン または ■ `fileUrl`：Git サーバを選択した後の、リモート クラウド テンプレート YAML の URL。 ------ ■ `deploymentId`：展開の ID または ■ `deploymentName`：展開の名前 ------ [DELETE_DEPLOYMENT] ■ `deploymentId`：展開の ID または ■ `deploymentName`：展開の名前 [ROLLBACK_DEPLOYMENT] 次のいずれかの組み合わせ： ■ `deploymentId`：展開の ID または ■ `deploymentName`：展開の名前	

表 3-12. 継続的展開タスクの自動化：クラウド テンプレート （続き）

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
		------ ■ blueprintName：クラウド テンプレートの名前 ■ rollbackVersion：ロールバックするバージョン	
	Output		他のタスクまたはパイプラインの出力にバインドできるパラメータ： ■ 展開名は、**${Stage0.Task0.output.deploymentName}** としてアクセスできます。 ■ 展開 ID は、**${Stage0.Task0.output.deploymentId}** としてアクセスできます。 ■ 展開の詳細は複雑なオブジェクトです。内部の詳細には、JSON の結果を使用してアクセスできます。 任意のプロパティにアクセスするには、ドット演算子を使用して JSON 階層を指定します。たとえば、Cloud_Machine_1[0] というリソースのアドレスにアクセスする場合、[$] バインディングは次のとおりです。 **${Stage0.Task0.output.deploymentDetails.resources['Cloud_Machine_1[0]'].address}** 同様に、フレーバーの場合の [$] バインディングは次のとおりです。 **${Stage0.Task0.output.deploymentDetails.resources['Cloud_Machine_1[0]'].flavor}** Code Stream のユーザー インターフェイスでは、任意のプロパティについて [$] バインディングを取得できます。 1 タスクの出力プロパティ領域で、[JSON 出力の表示] をクリックします。 2 [$] バインディングを見つけるために、任意のプロパティを入力します。 3 検索アイコンをクリックすると、対応する [$] バインディングが表示されます。

JSON 出力の例：

展開の詳細オブジェクトの例：

```
{
    "id": "6a031f92-d0fa-42c8-bc9e-3b260ee2f65b",
    "name": "deployment_6a031f92-d0fa-42c8-bc9e-3b260ee2f65b",
    "description": "Pipeline Service triggered operation",
    "orgId": "434f6917-4e34-4537-b6c0-3bf3638a71bc",
    "blueprintId": "8d1dd801-3a32-4f3b-adde-27f8163dfe6f",
    "blueprintVersion": "1",
    "createdAt": "2020-08-27T13:50:24.546215Z",
    "createdBy": "user@vmware.com",
    "lastUpdatedAt": "2020-08-27T13:52:50.674957Z",
    "lastUpdatedBy": "user@vmware.com",
    "inputs": {},
    "simulated": false,
    "projectId": "267f8448-d26f-4b65-b310-9212adb3c455",
    "resources": {
        "Cloud_Machine_1[0]": {
            "id": "/resources/compute/1606fbcd-40e5-4edc-ab85-7b559aa986ad",
            "name": "Cloud_Machine_1[0]",
            "powerState": "ON",
            "address": "10.108.79.33",
            "resourceLink": "/resources/compute/1606fbcd-40e5-4edc-ab85-7b559aa986ad",
            "componentTypeId": "Cloud.vSphere.Machine",
            "endpointType": "vsphere",
            "resourceName": "Cloud_Machine_1-mcm110615-146929827053",
            "resourceId": "1606fbcd-40e5-4edc-ab85-7b559aa986ad",
            "resourceDescLink": "/resources/compute-descriptions/1952d1d3-15f0-4574-
ae42-4fbf8a87d4cc",
            "zone": "Automation / Vms",
            "countIndex": "0",
            "image": "ubuntu",
            "count": "1",
            "flavor": "small",
            "region": "MYBU",
            "_clusterAllocationSize": "1",
            "osType": "LINUX",
            "componentType": "Cloud.vSphere.Machine",
            "account": "bha"
        }
```

```
    },
    "status": "CREATE_SUCCESSFUL",
    "deploymentURI": "https://api.yourenv.com/automation-ui/#/deployment-ui;ash=/deployment/
6a031f92-d0fa-42c8-bc9e-3b260ee2f65b"
}
```

表 3-13. 継続的展開タスクの自動化：Kubernetes

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
[Kubernetes]			

	Input	`action`：[GET]、[CREATE]、[APPLY]、[DELETE]、[ROLLBACK] のいずれか
		■ `timeout`：任意のアクションの全体的なタイムアウト
		■ `filterByLabel`：K8S labelSelector を使用して、アクション **`GET`** に対してフィルタを適用する追加ラベル
		[GET、CREATE、DELETE、APPLY]
		■ `yaml`：処理して Kubernetes に送信するインライン YAML
		■ `parameters`：KEY と VALUE のペア - インライン YAML 入力領域で **`$$KEY`** を **`VALUE`** に置き換える
		■ `filePath`：YAML の取得元となる、SCM Git エンドポイントからの相対パス（指定されている場合）
		■ `scmConstants`：KEY と VALUE のペア - SCM 経由で取得された YAML で **`$${KEY}`** を **`VALUE`** に置き換える
		■ `continueOnConflict`：true に設定すると、リソースがすでにある場合、タスクは続行される
		[ROLLBACK]
		■ `resourceType`：ロールバックするリソース タイプ
		■ `resourceName`：ロールバックするリソース名
		■ `namespace`：ロールバックを実行する必要がある名前空間
		■ `revision`：ロールバックするリビジョン
			`${MY_STAGE.MY_TASK.input.action}` # 実行するアクションを決定します。
			`${MY_STAGE.MY_TASK.input.timeout}`
			`${MY_STAGE.MY_TASK.input.filterByLabel}`
			`${MY_STAGE.MY_TASK.input.yaml}`
			`${MY_STAGE.MY_TASK.input.parameters}`
			`${MY_STAGE.MY_TASK.input.filePath}`
			`${MY_STAGE.MY_TASK.input.scmConstants}`
			`${MY_STAGE.MY_TASK.input.continueOnConflict}`
			`${MY_STAGE.MY_TASK.input.resourceType}`
			`${MY_STAGE.MY_TASK.input.resourceName}`
			`${MY_STAGE.MY_TASK.input.namespace}`
			`${MY_STAGE.MY_TASK.input.revision}`

	Output	`response`：応答全体をキャプチャする `response.<RESOURCE>`：リソースは configMaps、deployments、endpoints、ingresses、jobs、namespaces、pods、replicaSets、replicationControllers、secrets、services、statefulSets、nodes、loadBalancers に対応します。 `response.<RESOURCE>.<KEY>`：キーは apiVersion、kind、metadata、spec のいずれかに対応します。	`${MY_STAGE.MY_TASK.output.response}` `${MY_STAGE.MY_TASK.output.response.}`

表 3-14. 開発、テスト、展開アプリケーションの統合

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
[Bamboo]			
	Input	plan：プランの名前 planKey：プラン キー variables：プランに渡される変数 parameters：プランに渡されるパラメータ	${MY_STAGE.MY_TASK.input.plan} ${MY_STAGE.MY_TASK.input.planKey} ${MY_STAGE.MY_TASK.input.variables} ${MY_STAGE.MY_TASK.input.parameters} # すべてのパラメータを参照 ${MY_STAGE.MY_TASK.input.parameters.param1} # param1 の値を参照
	Output	resultUrl：結果のビルドの URL buildResultKey：結果のビルドのキー buildNumber：ビルド番号 buildTestSummary：実行されるテストのサマリ successfulTestCount：合格したテスト結果 failedTestCount：失敗したテスト結果 skippedTestCount：スキップされたテスト結果 artifacts：ビルドからのアーティファクト	${MY_STAGE.MY_TASK.output.resultUrl} ${MY_STAGE.MY_TASK.output.buildResultKey} ${MY_STAGE.MY_TASK.output.buildNumber} ${MY_STAGE.MY_TASK.output.buildTestSummary} # すべての結果を参照 ${MY_STAGE.MY_TASK.output.successfulTestCount} # 特定のテスト カウントを参照 ${MY_STAGE.MY_TASK.output.buildNumber}
[Jenkins]			
	Input	job：Jenkins ジョブの名前 parameters：ジョブに渡されるパラメータ	${MY_STAGE.MY_TASK.input.job} ${MY_STAGE.MY_TASK.input.parameters} # すべてのパラメータを参照 ${MY_STAGE.MY_TASK.input.parameters.param1} # パラメータの値を参照
	Output	job：Jenkins ジョブの名前 jobId：結果のジョブの ID（1234 など） jobStatus：Jenkins のステータス jobResults：テスト/コード カバレッジ結果のコレクション jobUrl：結果のジョブの実行 URL	${MY_STAGE.MY_TASK.output.job} ${MY_STAGE.MY_TASK.output.jobId} ${MY_STAGE.MY_TASK.output.jobStatus} ${MY_STAGE.MY_TASK.output.jobResults} # すべての結果を参照 ${MY_STAGE.MY_TASK.output.jobResults.junitResponse} # JUnit の結果を参照 ${MY_STAGE.MY_TASK.output.jobResults.jacocoRespose} # JaCoCo の結果を参照 ${MY_STAGE.MY_TASK.output.jobUrl}
[TFS]			

表 3-14. 開発、テスト、展開アプリケーションの統合 （続き）

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
	Input	projectCollection：TFS からのプロジェクト コレクション teamProject：使用可能なコレクションから選択したプロジェクト buildDefinitionId：実行するビルド定義 ID	${MY_STAGE.MY_TASK.input.projectCollection} ${MY_STAGE.MY_TASK.input.teamProject} ${MY_STAGE.MY_TASK.input.buildDefinitionId}
	Output	buildId：結果のビルド ID buildUrl：ビルドのサマリへのアクセス URL logUrl：ログの参照 URL dropLocation：アーティファクトのドロップ場所（ある場合）	${MY_STAGE.MY_TASK.output.buildId} ${MY_STAGE.MY_TASK.output.buildUrl} ${MY_STAGE.MY_TASK.output.logUrl} ${MY_STAGE.MY_TASK.output.dropLocation}
[vRO]			
	Input	workflowId：実行するワークフローの ID parameters：ワークフローに渡されるパラメータ	${MY_STAGE.MY_TASK.input.workflowId} ${MY_STAGE.MY_TASK.input.parameters}
	Output	workflowExecutionId：ワークフロー実行の ID properties：ワークフロー実行からの出力プロパティ	${MY_STAGE.MY_TASK.output.workflowExecutionId} ${MY_STAGE.MY_TASK.output.properties}

表 3-15. API を使用した他のアプリケーションの統合

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
[REST]			
	Input	url：呼び出す URL action：使用する HTTP メソッド headers：渡される HTTP ヘッダー payload：要求ペイロード fingerprint：https の URL と照合するフィンガープリント allowAllCerts：true に設定されている場合、URL が https のすべての証明書を使用できます。	${MY_STAGE.MY_TASK.input.url} ${MY_STAGE.MY_TASK.input.action} ${MY_STAGE.MY_TASK.input.headers} ${MY_STAGE.MY_TASK.input.payload} ${MY_STAGE.MY_TASK.input.fingerprint} ${MY_STAGE.MY_TASK.input.allowAllCerts}

表 3-15. API を使用した他のアプリケーションの統合 （続き）

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法

	Output	`responseCode`：HTTP 応答コード
		`responseHeaders`：HTTP 応答ヘッダー
		`responseBody`：受信する応答の文字列形式
		`responseJson`：コンテンツ タイプが [application/json] の場合のトラバーサブル応答
			`${MY_STAGE.MY_TASK.output.responseCode}`
			`${MY_STAGE.MY_TASK.output.responseHeaders}`
			`${MY_STAGE.MY_TASK.output.responseHeaders.header1}` # 応答ヘッダー header1 を参照
			`${MY_STAGE.MY_TASK.output.responseBody}`
			`${MY_STAGE.MY_TASK.output.responseJson}` # JSON としての応答を参照
			`${MY_STAGE.MY_TASK.output.responseJson.a.b.c}` # 応答に含まれる a.b.c JSON パスの後にネストされたオブジェクトを参照

[ポーリング]			

	Input	`url`：呼び出す URL
		`headers`：渡される HTTP ヘッダー
		`exitCriteria`：タスクが成功または失敗するために満たす必要がある基準。キーと値のペア。「成功」の場合の式、「失敗」の場合の式
		`pollCount`：実行する反復回数。 Code Stream 管理者は、ポーリング数を最大 10,000 に設定できます。
		`pollIntervalSeconds`：各反復処理の間に待機する秒数。ポーリング間隔は 60 秒以上にする必要があります。
		`ignoreFailure`：true に設定すると、途中の応答での失敗を無視
		`fingerprint`：https の URL と照合するフィンガープリント
		`allowAllCerts`：true に設定されている場合、URL が https のすべての証明書を使用できます。
			`${MY_STAGE.MY_TASK.input.url}`
			`${MY_STAGE.MY_TASK.input.headers}`
			`${MY_STAGE.MY_TASK.input.exitCriteria}`
			`${MY_STAGE.MY_TASK.input.pollCount}`
			`${MY_STAGE.MY_TASK.input.pollIntervalSeconds}`
			`${MY_STAGE.MY_TASK.input.ignoreFailure}`
			`${MY_STAGE.MY_TASK.input.fingerprint}`
			`${MY_STAGE.MY_TASK.input.allowAllCerts}`

	Output	`responseCode`：HTTP 応答コード
		`responseBody`：受信する応答の文字列形式
		`responseJson`：コンテンツ タイプが [application/json] の場合のトラバーサブル応答
			`${MY_STAGE.MY_TASK.output.responseCode}`
			`${MY_STAGE.MY_TASK.output.responseBody}`
			`${MY_STAGE.MY_TASK.output.responseJson}` # Refer to response as JSON

表 3-16. リモートおよびユーザー定義のスクリプトの実行

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
[PowerShell] PowerShell タスクを実行するには、次の手順を実行する必要があります。 ■ リモート Windows ホストに対するセッションをアクティブにします。 ■ base64 PowerShell コマンドを入力する場合は、最初にコマンド全体の長さを計算します。詳細については、 Code Stream で使用可能なタスクのタイプを参照してください。			
	入力	host：マシンの IP アドレスまたはホスト名 username：接続に使用するユーザー名 password：接続に使用するパスワード useTLS：https 接続の試行 trustCert：true に設定すると、自己署名証明書を信頼 script：実行するスクリプト workingDirectory：スクリプトを実行する前にディレクトリを切り替える先のパス environmentVariables:設定する環境変数のキーと値のペア arguments：スクリプトに渡す引数	${MY_STAGE.MY_TASK.input.host} ${MY_STAGE.MY_TASK.input.username} ${MY_STAGE.MY_TASK.input.password} ${MY_STAGE.MY_TASK.input.useTLS} ${MY_STAGE.MY_TASK.input.trustCert} ${MY_STAGE.MY_TASK.input.script} ${MY_STAGE.MY_TASK.input.workingDirectory} ${MY_STAGE.MY_TASK.input.environmentVariables} ${MY_STAGE.MY_TASK.input.arguments}
	出力	:ファイル $SCRIPT_RESPONSE_FILE response の内容 responseFilePath： $SCRIPT_RESPONSE_FILE の値 exitCode：プロセス終了コード logFilePath：stdout を含むファイルへのパス errorFilePath：stderr を含むファイルへのパス	${MY_STAGE.MY_TASK.output.response} ${MY_STAGE.MY_TASK.output.responseFilePath} ${MY_STAGE.MY_TASK.output.exitCode} ${MY_STAGE.MY_TASK.output.logFilePath} ${MY_STAGE.MY_TASK.output.errorFilePath}
[SSH]			

表 3-16. リモートおよびユーザー定義のスクリプトの実行 （続き）

タスク	Scope	Key	タスクでの SCOPE と KEY の使用方法
	Input	host：マシンの IP アドレスまたはホスト名 username：接続に使用するユーザー名 password：接続に使用するパスワード（オプションで privateKey を使用可能） privateKey：接続に使用する PrivateKey passphrase：privateKey のロックを解除するためのオプションのパスフレーズ script：実行するスクリプト workingDirectory：スクリプトを実行する前にディレクトリを切り替える先のパス environmentVariables：設定する環境変数のキーと値のペア	${MY_STAGE.MY_TASK.input.host} ${MY_STAGE.MY_TASK.input.username} ${MY_STAGE.MY_TASK.input.password} ${MY_STAGE.MY_TASK.input.privateKey} ${MY_STAGE.MY_TASK.input.passphrase} ${MY_STAGE.MY_TASK.input.script} ${MY_STAGE.MY_TASK.input.workingDirectory} ${MY_STAGE.MY_TASK.input.environmentVariables}
	Output	:ファイル $SCRIPT_RESPONSE_FILE response の内容 responseFilePath：$SCRIPT_RESPONSE_FILE の値 exitCode：プロセス終了コード logFilePath：stdout を含むファイルへのパス errorFilePath：stderr を含むファイルへのパス	${MY_STAGE.MY_TASK.output.response} ${MY_STAGE.MY_TASK.output.responseFilePath} ${MY_STAGE.MY_TASK.output.exitCode} ${MY_STAGE.MY_TASK.output.logFilePath} ${MY_STAGE.MY_TASK.output.errorFilePath}

タスク間で変数のバインドを使用する方法

次の例は、パイプライン タスクで変数のバインドを使用する方法を示しています。

表 3-17. 構文形式のサンプル

例	構文
パイプライン通知およびパイプライン出力プロパティにタスク出力値を使用する	${<Stage Key>.<Task Key>.output.<Task output key>}
現在のタスクの入力として前のタスク出力値を参照する	${<Previous/Current Stage key>.<Previous task key not in current Task group>.output.<task output key>}

詳細

タスクのバインド変数の詳細については、次を参照してください。

- Code Stream パイプラインで変数のバインドを使用する方法
- 条件タスクで変数バインドを使用して、 Code Stream でパイプラインを実行または停止する方法
- Code Stream で使用可能なタスクのタイプ

Code Stream でパイプラインに関する通知を送信する方法

通知は、Code Stream でパイプラインのステータスをチームが把握できるようにするためのチームとの通信方法です。

パイプラインの実行時に通知を送信するには、パイプライン全体、ステージ、またはタスクのステータスに基づいて Code Stream の通知を構成します。

- E メール通知により、次のときに E メールが送信されます。
  - パイプラインの完了、待機中、失敗、キャンセル、または開始。
  - ステージの完了、失敗、または開始。
  - タスクの完了、待機中、失敗、または開始。
- チケット通知により、次のときにチケットが作成され、チーム メンバーに割り当てられます。
  - パイプラインの失敗または完了。
  - ステージの失敗。
  - タスクの失敗。
- Webhook 通知により、次のときに別のアプリケーションに要求が送信されます。
  - パイプラインの失敗、完了、待機中、キャンセル、または開始。
  - ステージの失敗、完了、または開始。
  - タスクの失敗、完了、待機中、または開始。

たとえば、ユーザー操作タスクで E メール通知を設定して、パイプラインの特定の時点で承認を得ることができます。パイプラインが実行されると、このタスクは、タスクを承認する必要があるユーザーに E メールを送信します。ユーザー操作タスクの有効期限のタイムアウトが日、時間、または分単位で設定されている場合、タスクの有効期限が切れる前に、目的のユーザーがパイプラインを承認する必要があります。そうしないと、パイプラインは予期したとおり失敗します。

パイプライン タスクが失敗した場合に JIRA チケットを作成するには、通知を構成します。または、パイプライン イベントに基づいて、パイプラインのステータスに関する要求を Slack チャンネルに送信するには、Webhook 通知を構成します。

すべてのタイプの通知で変数を使用できます。たとえば、Webhook 通知の URL で **`${var}`** を使用できます。

前提条件

- 1 つ以上のパイプラインが作成されていることを確認します。5 章 Code Stream を使用するためのチュートリアルのユースケースを参照してください。
- E メール通知を送信するには、稼動中のメール サーバにアクセスできることを確認します。ヘルプについては、管理者に確認してください。
- JIRA チケットなどのチケットを作成するには、エンドポイントがあることを確認します。 Code Stream でのエンドポイントとは を参照してください。
- 統合に基づいて通知を送信するには、Webhook 通知を作成します。次に、Webhook が追加され、動作していることを確認します。Slack、GitHub、GitLab などのアプリケーションで通知を使用できます。

手順

1 パイプラインを開きます。

2 パイプラインの全体的なステータス、またはステージまたはタスクのステータスに関する通知を作成するには、次の手順を実行します。

作成する通知	実行する操作
パイプライン ステータス	パイプライン キャンバス上の空白領域をクリックします。
ステージのステータス	パイプラインのステージ内の空白領域をクリックします。
タスクのステータス	パイプラインのステージ内のタスクをクリックします。

3 [通知] タブをクリックします。

4 [追加] をクリックし、通知のタイプを選択して、通知の詳細を構成します。

5 パイプラインが成功したときの Slack 通知を作成するには、webhook 通知を作成します。

   a [Webhook] を選択します。

   b Slack 通知を構成するには、情報を入力します。

   c [保存] をクリックします。

   d パイプラインが実行されると、Slack チャンネルがパイプライン ステータスの通知を受け取ります。たとえば、ユーザーの Slack チャンネルには以下が表示される場合があります。

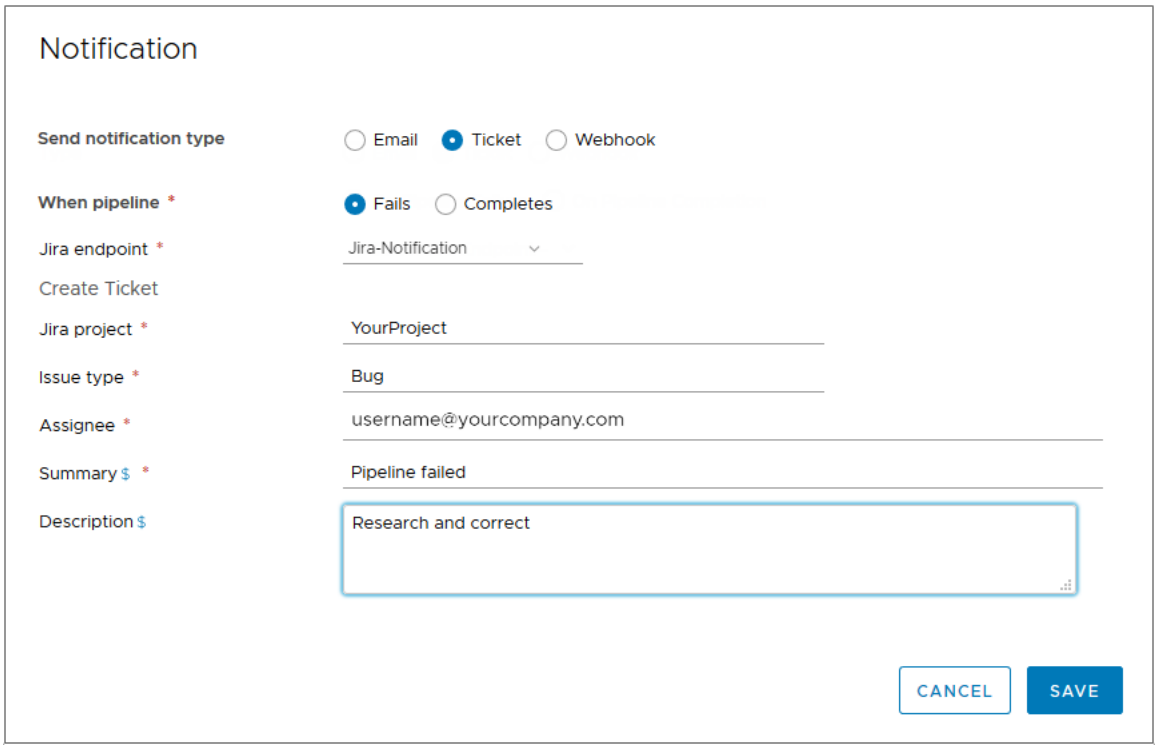
```
Codestream APP [12:01 AM]
Tested by User1 - Staging Pipeline 'User1-Pipeline', Pipeline ID
'e9b5884d809ce2755728177f70f8a' succeeded
```

6 JIRA チケットを作成するには、チケット情報を構成します。

a [チケット] を選択します。

b JIRA 通知を構成するには、情報を入力します。

c [保存] をクリックします。



結果

完了です。 Code Stream では、パイプラインの複数の領域でさまざまなタイプの通知を作成できることを学習しました。

次のステップ

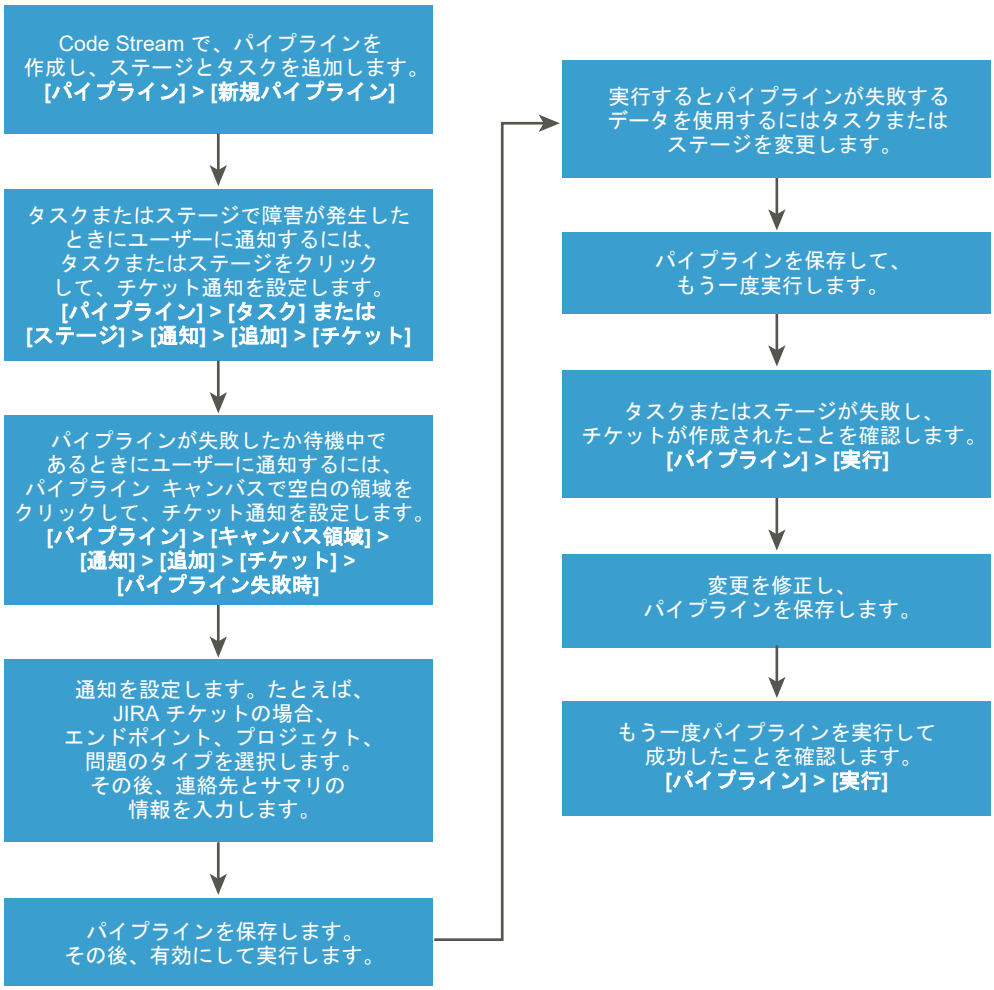
通知を作成する方法の例の詳細については、パイプライン タスクが失敗したときに Code Stream で JIRA チケットを作成する方法を参照してください。

パイプライン タスクが失敗したときに Code Stream で JIRA チケットを作成する方法

パイプライン内のステージまたはタスクで障害が発生した場合は、 Code Stream で Jira チケットを作成できます。この問題を解決する必要があるユーザーにチケットを割り当てることができます。また、パイプラインが待機中または成功したときにチケットを作成することもできます。

タスク、ステージ、またはパイプラインに通知を追加して構成できます。 Code Stream は、通知を追加するタスク、ステージ、またはパイプラインのステータスに基づいてチケットを作成します。たとえば、エンドポイントが使用できない場合、 Code Stream で、エンドポイントに接続できないために失敗したタスクの JIRA チケットを作成することができます。

パイプラインが成功したときに通知を作成することもできます。たとえば、ビルドを確認して別のテスト パイプラインを実行できるように、QA チームに成功したパイプラインについて知らせることができます。または、パイプラインのパフォーマンスの測定と、ステージングまたは本番環境へのアップデートの準備ができるように、パフォーマンス チームに通知することができます。



この例では、パイプライン タスクが失敗したときに JIRA チケットが作成されます。

前提条件

- 有効な JIRA アカウントがあり、JIRA インスタンスにログインできることを確認します。
- JIRA エンドポイントがあり、動作していることを確認します。

手順

1 パイプラインで、タスクをクリックします。

2 タスクの設定領域で、[通知] をクリックします。

3 [追加] をクリックして、チケット情報を構成します。

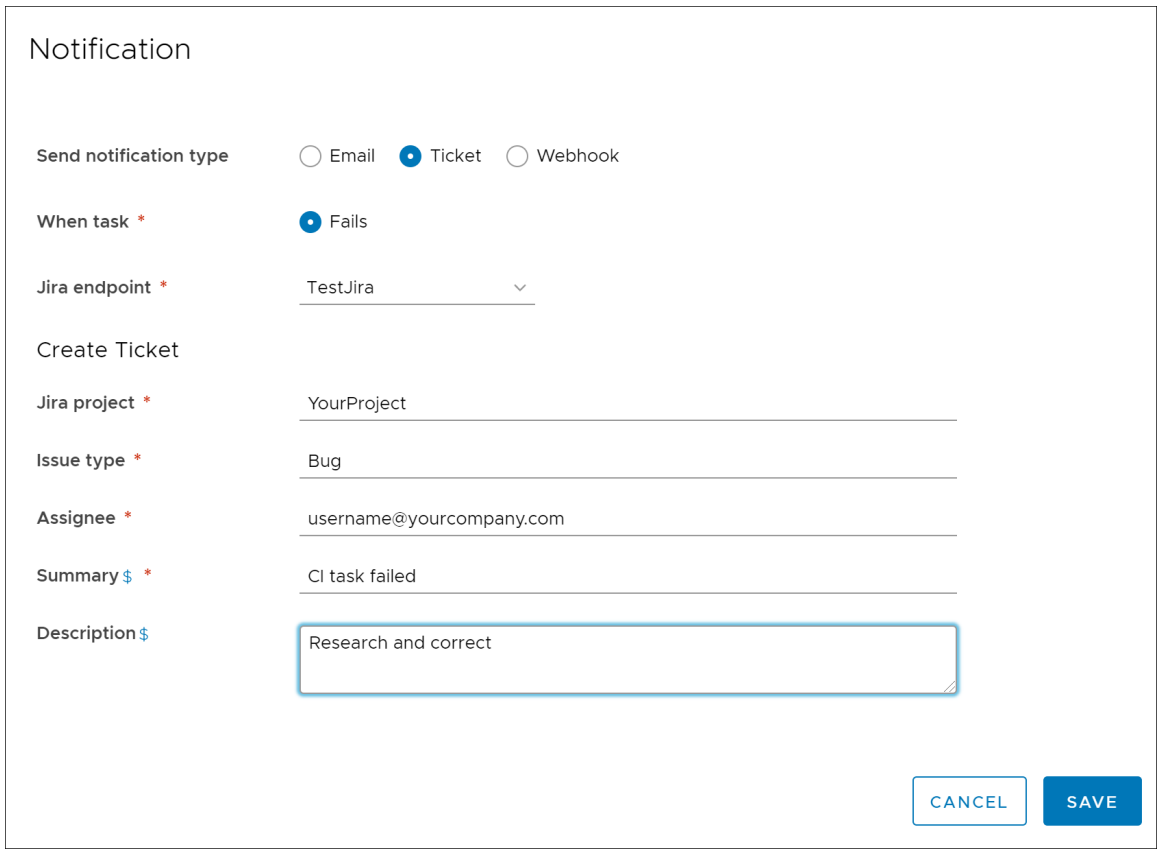
a [チケット] をクリックします。

b JIRA エンドポイントを選択します。

c JIRA プロジェクトと問題のタイプを入力します。

d チケットを受信するユーザーのメール アドレスを入力します。

e チケットの概要と説明を入力し、[保存] をクリックします。



4 パイプラインを保存し、有効にして実行します。

5 チケットをテストします。

a タスクが失敗する原因となるデータを含めるようにタスク情報を変更します。

b パイプラインを保存して、もう一度実行します。

c [実行] をクリックして、パイプラインが失敗したことを確認します。

d 実行中に、 Code Stream がチケットを作成して送信したことを確認します。

e タスク情報を再度変更して修正してから、パイプラインを再度実行し、正常に動作することを確認します。

結果

完了です。パイプライン タスクが失敗したときに、 Code Stream で JIRA チケットを作成し、それを解決する必要があるユーザーに割り当てました。

次のステップ

パイプラインについてチームに警告するための通知を続けて追加します。

Code Stream で展開をロールバックする方法

展開パイプラインで障害が発生した後に、展開を以前の安定した状態に戻すタスクで、ロールバックをパイプラインとして設定します。障害が発生した場合にロールバックするには、ロールバック パイプラインをタスクまたはステージに適用します。

ロールによって、ロールバックする動機が異なる場合があります。

- リリース時に、 Code Stream が処理の成功を検証できれば、リリース エンジニアはリリースを続行するかロールバックするかを把握できるようになります。タスクの障害、UserOps の拒否、メトリックのしきい値の超過などの可能性があります。
- 以前のリリースを再展開できれば、環境の所有者は環境をすぐに正常な状態に戻すことができます。
- ブルーグリーン展開のロールバックをサポートできれば、環境の所有者はリリースに失敗した場合にダウンタイムを最小限に抑えることができます。

ロールバック オプションをクリックした状態でスマート パイプライン テンプレートを使用して CD パイプラインを作成すると、パイプラインのタスクにロールバックが自動的に追加されます。この使用事例では、スマート パイプライン テンプレートで、ローリング アップグレード展開モデルを使用して、Kubernetes クラスタへのアプリケーション展開のロールバックを定義します。スマート パイプライン テンプレートにより、展開パイプラインと 1 つ以上のロールバック パイプラインを作成します。

- 展開パイプラインでは、展開の更新タスクまたは展開の確認タスクが失敗した場合にロールバックが必要です。
- ロールバック パイプラインでは、古いイメージを使用して展開が更新されます。

空のテンプレートを使用してロールバック パイプラインを手動で作成することもできます。ロールバック パイプラインを作成する前に、ロールバック フローを計画すると便利です。ロールバックの背景情報として、Code Stream でのロールバックの計画を参照してください。

前提条件

- Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。
- アプリケーションの展開先となる Kubernetes クラスタをセットアップします。1 つの開発クラスタと 1 つの本番クラスタをセットアップします。
- Docker レジストリがセットアップされていることを確認します。
- パイプライン、エンドポイント、ダッシュボードをはじめすべての作業をグループ化するプロジェクトを特定します。

- たとえば、スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画の CD 部分に記載されている CD スマート テンプレートについて理解しておく必要があります。
  - アプリケーション イメージを Kubernetes クラスタに展開する Kubernetes 開発エンドポイントと本番エンドポイントを作成します。
  - 名前空間、サービス、展開を作成する Kubernetes YAML ファイルを準備します。プライベート所有のリポジトリからイメージをダウンロードする必要がある場合は、Docker 構成の [シークレット] を含むセクションを YAML ファイルに含める必要があります。

手順

1 [パイプライン] - [新しいパイプライン] - [スマート テンプレート] - [継続的デリバリ] の順にクリックします。

2 スマート パイプライン テンプレートに情報を入力します。

a プロジェクトを選択します。

b **`RollingUpgrade-Example`** のようなパイプライン名を入力します。

c アプリケーションの環境を選択します。展開にロールバックを追加するには、[本番] を選択する必要があります。

d [選択] をクリックし、Kubernetes YAML ファイルを選択して、[プロセス] をクリックします。

使用可能なサービスと展開環境がスマート パイプライン テンプレートに表示されます。

e パイプラインによる展開で使用するサービスを選択します。

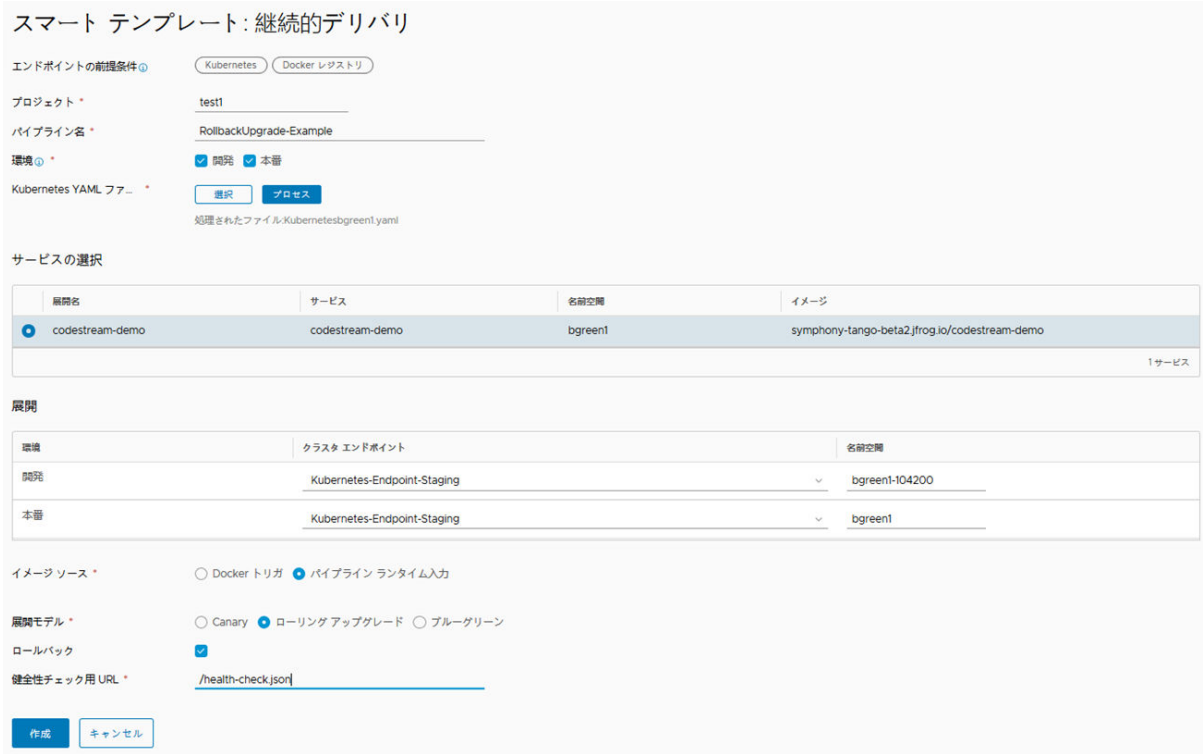
f 開発環境と本番環境のクラスタ エンドポイントを選択します。

g イメージ ソースの場合は、[パイプライン ランタイム入力] を選択します。

h 展開モデルの場合は、[ローリング アップグレード] を選択します。

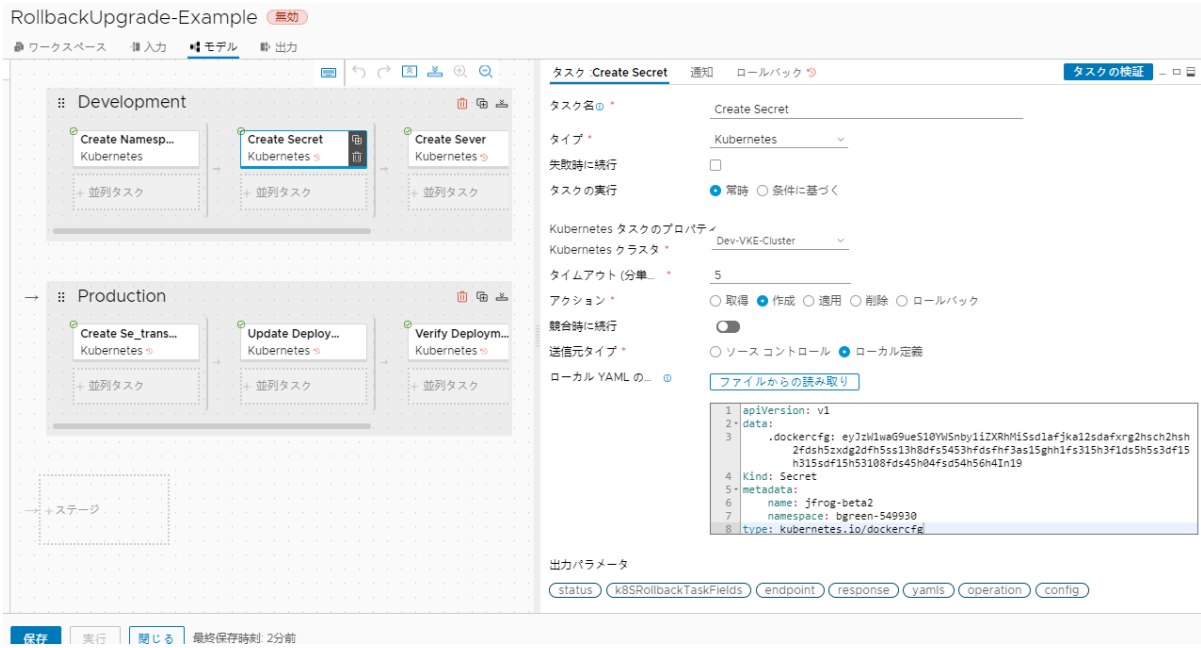
i [ロールバック] をクリックします。

j [健全性チェック用 URL] を指定します。



3 RollbackUpgrade-Example という名前のパイプラインを作成するには、[作成] をクリックします。

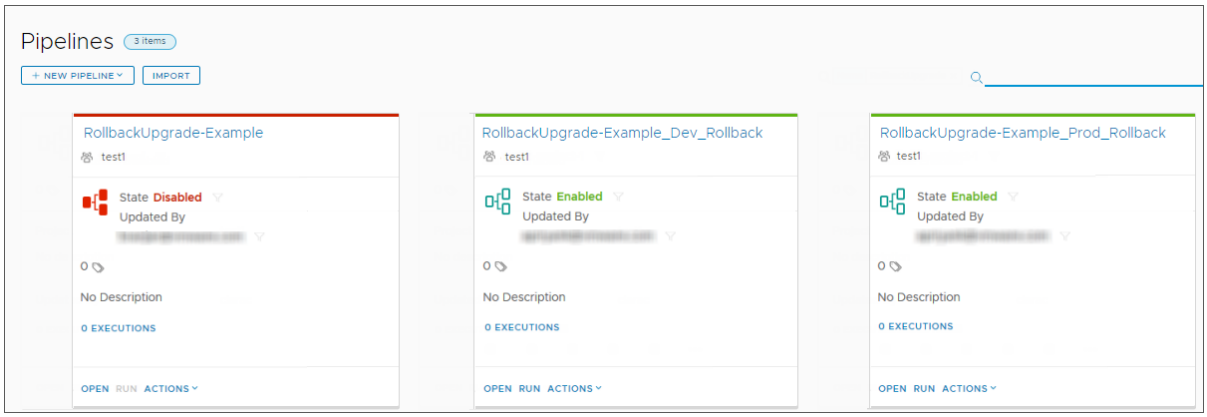
RollbackUpgrade-Example という名前のパイプラインが表示され、開発ステージおよび本番ステージ内のロールバックできるタスクにロールバック アイコンが表示されます。



4 パイプラインを終了します。

[パイプライン] 画面には、作成したパイプラインと、パイプライン内のステージごとに新しいパイプラインが表示されます。

- RollingUpgrade-Example。デフォルトでは、作成したパイプラインは Code Stream によって無効にされています。これは、実行前に必ず確認するためです。
- RollingUpgrade-Example_Dev_Rollback。[サービスの作成]、[シークレットの作成]、[展開の作成]、[展開の検証] など、開発ステージでのタスクが失敗すると、この開発ロールバック パイプラインが呼び出されます。開発タスクが確実にロールバックされるように、Code Stream ではデフォルトで開発ロールバック パイプラインが有効になります。
- RollingUpgrade-Example_Prod_Rollback。[展開フェーズ 1]、[検証フェーズ 1]、[ロールアウト展開フェーズ]、[ロールアウト完了フェーズ]、[ロールアウト検証フェーズ] など、本番ステージでのタスクが失敗すると、この本番ロールバック パイプラインが呼び出されます。本番タスクが確実にロールバックされるように、 Code Stream ではデフォルトで本番ロールバック パイプラインが有効になります。

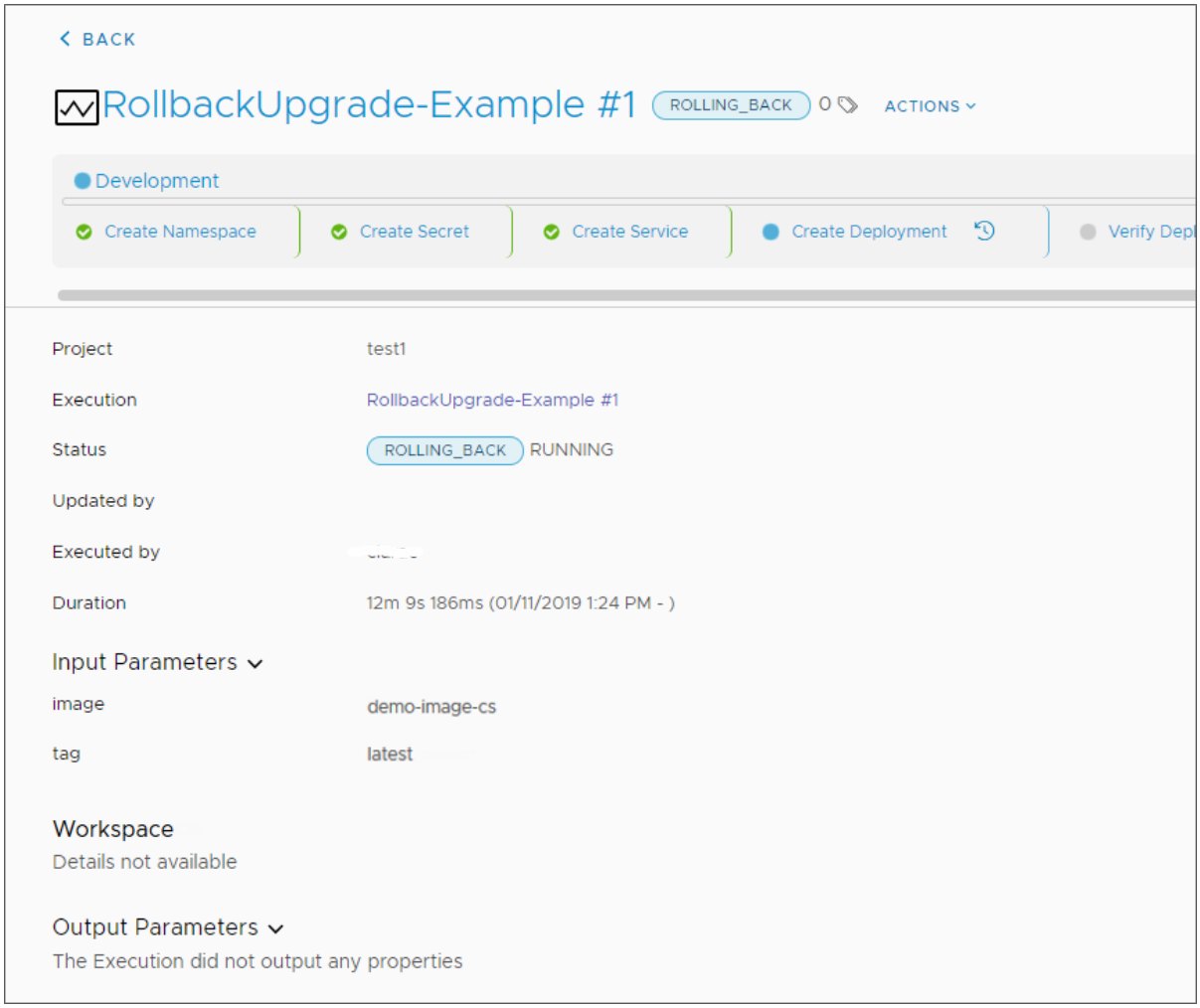


5 作成したパイプラインを有効にして実行します。

実行を開始すると、 Code Stream から入力パラメータの入力を求められます。使用している Docker リポジトリで、エンドポイントのイメージとタグを指定します。

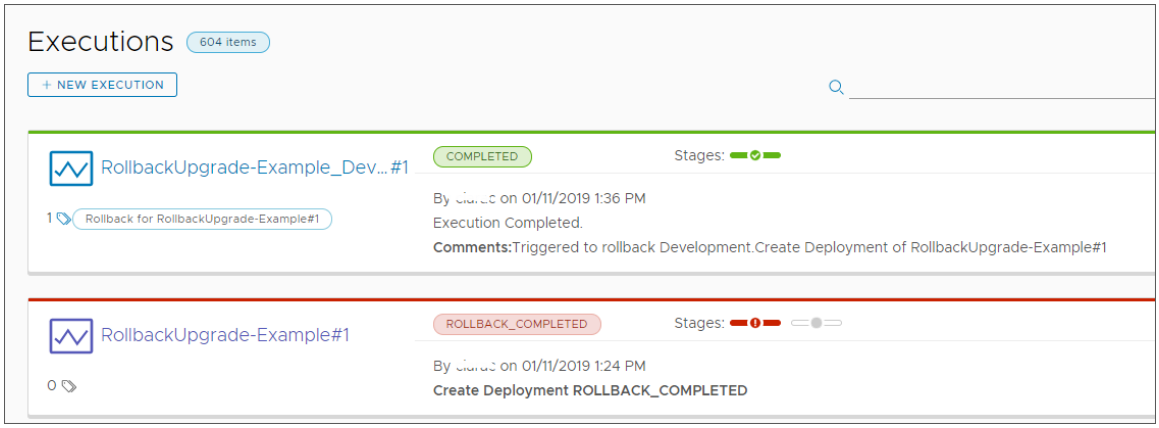
6 [実行] 画面で、[アクション] - [実行の表示] の順に選択し、パイプラインの実行を監視します。

パイプラインは、起動すると [RUNNING] になり、開発ステージの各タスクを移動していきます。開発ステージでパイプラインがタスクを実行できないと、RollingUpgrade-Example_Dev_Rollback という名前のパイプラインがトリガされ、展開がロールバックされて、パイプラインのステータスが [ROLLING_BACK] になります。



ロールバック後、[実行] 画面には RollingUpgrade-Example パイプライン実行が 2 つ表示されます。

- 作成したパイプラインがロールバックされ、[ROLLBACK_COMPLETED] と表示されます。
- ロールバックをトリガおよび実行した開発ロールバック パイプラインにより、[COMPLETED] と表示されます。



結果

完了です。パイプラインにロールバックが正常に定義され、障害が発生した時点でパイプラインがロールバックされたことを Code Stream で確認できます。

Code Stream でコードをネイティブにビルド、統合、および配信することを計画する

4

CICD、CI、または CD パイプラインを作成するネイティブ機能を使用して Code Stream でコードをビルド、統合、および配信する前に、ネイティブ ビルドを計画します。次に、いずれかのスマート パイプライン テンプレートを使用するか、手動でステージおよびタスクを追加して、パイプラインを作成できます。

継続的インテグレーションと継続的デリバリ ビルドの計画作成のために、その方法を示す例をいくつか用意しました。これらの計画では、パイプラインをビルドするときにネイティブのビルド機能を効果的に準備して使用するために役立つ前提条件と概要が示されます。

この章には、次のトピックが含まれています。

- パイプライン ワークスペースの構成
- スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画
- スマート パイプライン テンプレートの使用に先立つ Code Stream での継続的インテグレーション ネイティブ ビルドの計画
- スマート パイプライン テンプレートの使用に先立つ Code Stream での継続的デリバリ ネイティブ ビルドの計画
- タスクの手動追加を行う前の Code Stream での CICD ネイティブ ビルドの計画
- Code Stream でのロールバックの計画

パイプライン ワークスペースの構成

継続的インテグレーション タスクとカスタム タスクを実行するには、 Code Stream パイプラインに対してワークスペースを構成する必要があります。

パイプライン ワークスペースで、[タイプ] に Docker または Kubernetes を選択し、各エンドポイントを指定します。Docker および Kubernetes プラットフォームは、継続的インテグレーション (CI) タスクまたはカスタム タスクを実行するために Code Stream が展開するコンテナのライフサイクル全体を管理します。

- Docker ワークスペースには、Docker ホスト エンドポイント、ビルダー イメージ URL、イメージ レジストリ、作業ディレクトリ、キャッシュ、環境変数、CPU 制限、メモリ制限が必要です。Git リポジトリのクローンを作成することもできます。
- Kubernetes ワークスペースには、Kubernetes API エンドポイント、ビルダー イメージ URL、イメージ レジストリ、名前空間、NodePort、パーシステント ボリュームの要求 (PVC)、作業ディレクトリ、環境変数、CPU 制限、メモリ制限が必要です。Git リポジトリのクローンを作成することもできます。

パイプライン ワークスペースの構成には、次の表に示すように、いくつかの共通パラメータと、ワークスペースのタイプに固有のパラメータがあります。

表 4-1. ワークスペースの領域、詳細、および可用性

選択	説明	詳細および可用性
[タイプ]	ワークスペースのタイプ。	Docker または Kubernetes で使用できます。
[ホスト エンドポイント]	継続的インテグレーション (CI) およびカスタム タスクが実行されるホスト エンドポイント。	Docker ホスト エンドポイントを選択すると、Docker ワークスペースで使用できます。 Kubernetes API エンドポイントを選択すると、Kubernetes ワークスペースで使用できます。
[ビルダー イメージ URL]	ビルダー イメージの名前と場所。このイメージを使用して Docker ホストおよび Kubernetes クラスタ上にコンテナが作成されます。継続的インテグレーション (CI) タスクとカスタム タスクは、このコンテナ内で実行されます。	例：**`fedora:latest`** ビルダー イメージには、`curl` または `wget` が必要です。
[イメージ レジストリ]	ビルダー イメージがレジストリ内にあり、そのレジストリが認証情報を必要とする場合は、イメージ レジストリ エンドポイントを作成し、ここでそれを選択することで、レジストリからイメージを取得できるようにする必要があります。	Docker および Kubernetes ワークスペースで使用できます。
[作業ディレクトリ]	作業ディレクトリは、継続的インテグレーション (CI) タスクの手順が実行されるコンテナ内の場所であり、Git Webhook によってパイプラインの実行がトリガされたときにコードのクローンが作成される場所です。	Docker または Kubernetes で使用できます。
[名前空間]	名前空間を入力しないと、指定した Kubernetes クラスタ内に Code Stream によって一意の名前が作成されます。	Kubernetes ワークスペースに固有。
[プロキシ]	Kubernetes クラスタ内のワークスペース ポッドと通信するために、 Code Stream は、各 Kubernetes クラスタの名前空間 `codestream-proxy` に単一のプロキシ インスタンスを展開します。クラスタ構成に基づいて、[NodePort] または [LoadBalancer] タイプのいずれかを選択できます。 選択するオプションは、展開した Kubernetes クラスタの性質によって決まります。 ■ エンドポイントで指定されている Kubernetes API サーバ URL がプライマリ ノードの 1 台を介して公開されている場合は通常、[NodePort] を選択します。 ■ Amazon EKS (Elastic Kubernetes Service) などで、Kubernetes API サーバの URL がロード バランサによって公開されている場合は、[LoadBalancer] を選択します。	

表 4-1. ワークスペースの領域、詳細、および可用性 （続き）

選択	説明	詳細および可用性
[NodePort]	Code Stream では、Kubernetes クラスタ内で実行されているコンテナとの通信に NodePort が使用されます。 ポートを選択しない場合、 Code Stream は、Kubernetes によって割り当てられる短期ポートを使用します。短期ポートの範囲（30000 ～ 32767）に対する入力を許可するようにファイアウォール ルールが構成されていることを確認する必要があります。 ポートを入力する場合は、クラスタ内の別のサービスがそのポートを使用していないこと、およびポートがファイアウォール ルールで許可されていることを確認する必要があります。	Kubernetes ワークスペースに固有。
[パーシステント ボリュームの要求]	この機能により、Kubernetes ワークスペースのファイルをパイプラインの実行間で保持することができます。パーシステント ボリュームの要求に名前を指定すると、ログ、アーティファクト、およびキャッシュを保存できます。 パーシステント ボリュームの要求の作成の詳細については、`https://kubernetes.io/docs/concepts/storage/persistent-volumes/` にある Kubernetes のドキュメントを参照してください。	Kubernetes ワークスペースに固有。
[環境変数]	ここで渡されるキーと値のペアは、実行されるパイプライン内のすべての継続的インテグレーション (CI) タスクとカスタム タスクで使用できます。	Docker または Kubernetes で使用できます。 ここで変数への参照を渡すことができます。 ワークスペースで提供される環境変数は、パイプライン内のすべての継続的インテグレーション (CI) タスクとカスタム タスクに渡されます。 ここで環境変数が渡されない場合、これらの変数は、パイプライン内のそれぞれの継続的インテグレーション (CI) タスクとカスタム タスクに明示的に渡す必要があります。
[CPU リミット]	継続的インテグレーション (CI) コンテナまたはカスタム タスク コンテナ向けの CPU リソースを制限します。	デフォルトは **1** です。
[メモリ リミット]	継続的インテグレーション (CI) コンテナまたはカスタム タスク コンテナ向けのメモリを制限します。	単位は `MB` です。

表 4-1. ワークスペースの領域、詳細、および可用性 （続き）

選択	説明	詳細および可用性
[Git クローン]	[Git クローン] を選択し、Git Webhook がパイプラインを呼び出すと、コードのクローンがワークスペース（コンテナ）内に作成されます。	[Git クローン] が有効になっていない場合は、パイプライン内で別の明示的な継続的インテグレーション (CI) タスクを構成して、まずコードのクローンを作成してから、ビルドやテストなどの他の手順を実行する必要があります。

[キャッシュ]	Code Stream ワークスペースでは、一連のディレクトリまたはファイルをキャッシュして、その後のパイプライン実行を高速化できます。これらのディレクトリは、たとえば `.m2`、`npm_modules` などです。パイプラインの実行間でデータのキャッシュを必須にしない場合、パーシステント ボリュームの要求が不要になります。
	コンテナ内のファイルやディレクトリなどのアーティファクトは、複数のパイプライン実行で再利用するためにキャッシュされます。たとえば、`node_modules` フォルダや `.m2` フォルダをキャッシュできます。[キャッシュ] には、パスのリストを指定できます。
	例：
	`workspace:`
	`  type: K8S`
	`  endpoint: K8S-Micro`
	`  image: fedora:latest`
	`  registry: Docker Registry`
	`  path: ''`
	`  cache:`
	`  - /path/to/m2`
	`  - /path/to/node_modules`
		ワークスペースのタイプに固有。
		Docker ワークスペースでは、Docker ホスト内にある、キャッシュされたデータ、アーティファクト、およびログを保持するための共有パスを使用することで、[キャッシュ] が実行されます。
		Kubernetes ワークスペースで [キャッシュ] の使用を有効にするには、パーシステント ボリュームの要求を指定する必要があります。それ以外の場合、[キャッシュ] は使用できません。

パイプライン ワークスペースで Kubernetes API エンドポイントを使用する場合、 Code Stream は、継続的インテグレーション (CI) タスクまたはカスタム タスクを実行するために必要な ConfigMap、シークレット、ポッドなどの Kubernetes リソースを作成します。 Code Stream は、NodePort を使用してコンテナと通信します。

パイプラインの実行の間でデータを共有するには、パーシステント ボリュームの要求を指定する必要があります。Code Stream はパーシステント ボリュームの要求をコンテナにマウントしてデータを保存し、以降のパイプラインの実行に使用します。

スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画

Code Stream で継続的インテグレーションと継続的デリバリ (CICD) パイプラインを作成するには、CICD スマート パイプライン テンプレートを使用します。CICD ネイティブ ビルドを計画するには、この例で示されている計画のパイプラインを作成する前にスマート パイプライン テンプレートの情報を収集します。

CICD パイプラインを作成するには、パイプラインの継続的インテグレーション (CI) ステージと継続的デリバリ (CD) ステージの両方を計画する必要があります。

スマート パイプライン テンプレートに情報を入力して保存した後、テンプレートでステージとタスクを含むパイプラインが作成されます。また、開発や導入などの選択した環境タイプに基づいて、イメージの展開先となる場所も示されます。パイプラインは、コンテナ イメージを公開し、実行に必要なアクションを実行します。パイプラインの実行後、その実行のトレンドを監視できます。

パイプラインに Docker Hub のイメージが含まれている場合は、パイプラインを実行する前に、イメージに `cURL` または `wget` が組み込まれていることを確認する必要があります。パイプラインが実行されると、 Code Stream は、`cURL` または `wget` を使用してコマンドを実行するバイナリ ファイルをダウンロードします。

ワークスペースの構成については、パイプライン ワークスペースの構成を参照してください。

継続的インテグレーション (CI) ステージの計画

パイプラインの CI ステージを計画するには、外部および内部の要件を設定し、スマート パイプライン テンプレートの CI 部分で必要とされる情報を決定します。サマリを示します。

この例では、Docker ワークスペースを使用します。

必要となるエンドポイントとリポジトリ：

- 開発者が自分のコードをチェックインする Git ソース コード リポジトリ。 Code Stream は、開発者が変更をコミットするときに、パイプラインに最新のコードを取得します。
- 開発者のソース コードがあるリポジトリの Git エンドポイント。
- コンテナ内でビルド コマンドを実行する Docker ビルド ホストの Docker エンドポイント。
- Code Stream がイメージを Kubernetes クラスタに展開できるようにするための Kubernetes エンドポイント。
- 継続的インテグレーションのテストで実行するコンテナを作成するビルダー イメージ。
- Docker ビルド ホストからビルダー イメージを取得できるようにするためのイメージ レジストリ エンドポイント。

プロジェクトへのアクセス権が必要です。プロジェクトはパイプライン、エンドポイント、ダッシュボードなどすべての作業をグループ化します。 Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。

Git Webhook が必要になります。これにより、開発者がコード変更をコミットするときに Code Stream が Git トリガを使用してパイプラインがトリガされるようにできます。 Code Stream で Git トリガを使用してパイプラインを実行する方法を参照してください。

ビルドのツールセット：

- Maven などのビルド タイプ。
- 使用するすべての後処理ビルド ツール（JUnit、JaCoCo、Checkstyle、FindBugs など）。

公開ツール：

- ビルド コンテナを展開する Docker などのツール。
- イメージ タグ（コミット ID またはビルド番号のいずれか）。

ビルドのワークスペース：

- Docker ビルド ホスト（Docker エンドポイント）。
- イメージ レジストリ。パイプラインの CI 部分によって、選択したレジストリ エンドポイントからイメージが取得されます。コンテナは CI タスクを実行し、イメージを展開します。レジストリに認証情報が必要な場合は、イメージ レジストリ エンドポイントを作成し、ここで選択して、ホストがレジストリからイメージを取得できるようにする必要があります。
- 継続的インテグレーション タスクで実行するコンテナを作成するビルダー イメージの URL。

継続的デリバリ (CD) ステージの計画

パイプラインの CD ステージを計画するには、外部および内部の要件を設定し、スマート パイプライン テンプレートの CD 部分に入力する情報を決定します。

必要なエンドポイント：

- Code Stream がイメージを Kubernetes クラスタに展開できるようにするための Kubernetes エンドポイント。

環境タイプとファイル：

- すべての環境タイプ (Code Stream) は、開発や導入などのアプリケーションを展開します。スマート パイプライン テンプレートは、選択した環境タイプに基づいてパイプライン内のステージとタスクを作成します。

  表 4-2. CICD スマート パイプライン テンプレートで作成されるパイプライン ステージ

パイプライン コンテンツ	機能
ビルド公開ステージ	コードをビルドおよびテストし、ビルダー イメージを作成し、そのイメージを Docker ホストに公開します。
開発ステージ	開発 Amazon Web Services (AWS) クラスタを使用してイメージを作成および展開します。このステージでは、クラスタで名前空間を作成し、プライベート キーを作成できます。
本番ステージ	VMware Tanzu Kubernetes Grid Integrated Edition（旧称 VMware Enterprise PKS）の本番バージョンを使用して、イメージを本番環境の Kubernetes クラスタに展開します。

- CICD スマート パイプライン テンプレートの CD セクションで選択した Kubernetes YAML ファイル。

  Kubernetes YAML ファイルには、名前空間、サービス、展開に関する 3 つの必須のセクションと、シークレットに関する 1 つのオプションのセクションが含まれています。プライベート所有のリポジトリからイメージをダウンロードすることによってパイプラインを作成する場合は、Docker 構成の [シークレット] を含むセクションを含める必要があります。 作成するパイプラインで、パブリックに利用可能なイメージのみを使用する場合はシークレットは不要です。次のサンプル YAML ファイルには、4 つのセクションがあります。

```
apiVersion: v1
kind: Namespace
metadata:
  name: codestream
  namespace: codestream
---
apiVersion: v1
data:
```

```
  .dockerconfigjson:
eyJhdXRocyI6eyJodHRwczovL2luZ1234567890lci5pby92MS8iOnsidXNlcm5hbWUiOiJhdXRvbWF0aW9uYmV0YSI
sInBhc3N3b3JkIjoiVk13YXJlQDEyMyIsImVtYWlsIjoiYXV0b21hdGlvbmJldGF1c2VyQGdtYWlsLmNvbSIsImF1dG
giOiJZWFYwYjIxaGRHbHZibUpsZEdFNlZrMTNZWEpsUURFeU13PT0ifX19
kind: Secret
metadata:
  name: dockerhub-secret
  namespace: codestream
type: kubernetes.io/dockerconfigjson
---
apiVersion: v1
kind: Service
metadata:
  name: codestream-demo
  namespace: codestream
  labels:
    app: codestream-demo
spec:
  ports:
    - port: 80
  selector:
    app: codestream-demo
    tier: frontend
  type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: codestream-demo
  namespace: codestream
  labels:
    app: codestream-demo
spec:
  replicas: 10
  selector:
    matchLabels:
      app: codestream-demo
      tier: frontend
  template:
    metadata:
      labels:
        app: codestream-demo
        tier: frontend
    spec:
      containers:
      - name: codestream-demo
        image: automationbeta/codestream-demo:01
        ports:
```

```
        - containerPort: 80
          name: codestream-demo
      imagePullSecrets:
      - name: dockerhub-secret
```

注： Kubernetes YAML ファイルは、次のユースケースの例のように、CD スマート パイプライン テンプレートでも使用されます。

- Code Stream のアプリケーションをブルーグリーン展開に展開する方法
- Code Stream で展開をロールバックする方法
- Code Stream で Docker トリガを使用して、継続的な配信パイプラインを実行する方法

スマート テンプレートにファイルを適用するには、[選択] をクリックし、Kubernetes YAML ファイルを選択します。次に、[プロセス] をクリックします。使用可能なサービスと展開環境がスマート パイプライン テンプレートに表示されます。サービス、クラスタ エンドポイント、および展開方法を選択します。たとえば、Canary 展開モデルを使用するには、[Canary] を選択し、展開フェーズの割合を入力します。

スマート パイプライン テンプレートを使用してブルーグリーン展開のパイプラインを作成する例については、Code Stream のアプリケーションをブルーグリーン展開に展開する方法を参照してください。

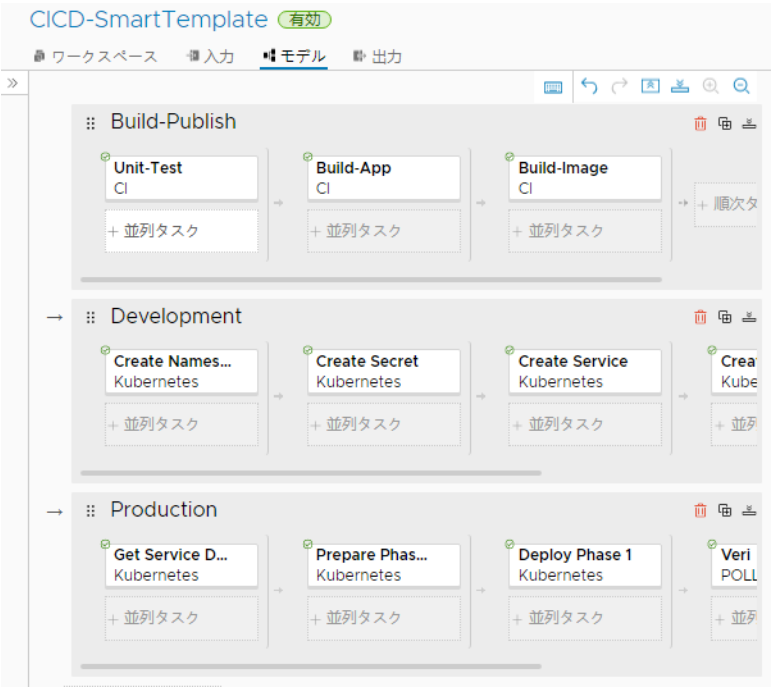
スマート パイプライン テンプレートを使用して CICD パイプラインを作成する方法

すべての情報を収集して必要な設定を行った後、次のように CICD スマート パイプライン テンプレートからパイプラインを作成します。

パイプラインで、[新しいパイプライン] - [スマート テンプレート] の順に選択します。

CICD スマート パイプライン テンプレートを選択します。

テンプレートに入力し、作成されるステージでパイプラインを保存します。最終的な変更を行う必要がある場合は、パイプラインを編集して保存できます。



次に、パイプラインを有効にして実行します。実行後、以下の点を確認できます。

- パイプラインが正常に完了したことを確認します。[実行] をクリックし、パイプラインを検索します。正常に完了しなかった場合は、エラーを修正して再度実行します。
- Git Webhook が正しく動作していることを確認します。Git の [Activity] タブにイベントが表示されます。[トリガ] - [Git] - [アクティビティ] の順にクリックします。

- パイプライン ダッシュボードで、トレンドを確認します。[ダッシュボード] をクリックし、パイプライン ダッシュボードを検索します。カスタム ダッシュボードを作成して、他の KPI もレポートできます。

詳細な例については、コードを my GitHub または GitLab リポジトリから Code Stream の自分のパイプラインに継続的に統合する方法を参照してください。

スマート パイプライン テンプレートの使用に先立つ Code Stream での継続的インテグレーション ネイティブ ビルドの計画

VMware Code Stream で継続的インテグレーション (CI) パイプラインを作成する場合は、継続的インテグレーション スマート テンプレートを使用できます。継続的インテグレーション ネイティブ ビルドを計画するには、この例で示されている計画のパイプラインを作成する前にスマート パイプライン テンプレートの情報を収集します。

スマート パイプライン テンプレートに入力すると、リポジトリに継続的インテグレーション パイプラインが作成され、パイプラインの実行に必要なアクションが実行されます。パイプラインの実行後、その実行のトレンドを監視できます。

継続的インテグレーション スマート パイプライン テンプレートを使用する前にビルドを計画するには、次の手順に従います。

- パイプライン、エンドポイント、ダッシュボードをはじめすべての作業をグループ化するプロジェクトを特定します。
- スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画の継続的デリバリ部分の説明に従って、ビルドの情報を収集します。

  たとえば、 Code Stream がコンテナを展開する Kubernetes エンドポイントを追加します。

次に、継続的インテグレーション スマート パイプライン テンプレートを使用してパイプラインを作成します。

[パイプライン] で、[スマート テンプレート] を選択します。

継続的インテグレーション スマート パイプライン テンプレートを選択します。

パイプラインをそれによって作成されるステージとともに保存するには、テンプレートに入力し、パイプラインの名前を入力します。パイプラインによって作成されたステージとともにパイプラインを保存するには、[作成] をクリックします。

Code Stream パイプライン ワークスペースでは、継続的インテグレーション タスクとカスタム タスクで Docker と Kubernetes がサポートされます。

ワークスペースの構成については、パイプライン ワークスペースの構成を参照してください。

最終的な変更を行うために、パイプラインを編集できます。次に、パイプラインを有効にして実行できます。パイプラインの実行後：

- パイプラインが正常に完了したことを確認します。[実行] をクリックし、パイプラインを検索します。失敗した場合は、エラーを修正し、再度実行します。
- Git の Webhook が正常に動作していることを確認します。Git [アクティビティ] タブに、イベントが表示されます。[トリガ] - [Git] - [アクティビティ] の順にクリックします。
- パイプラインのダッシュボードでトレンドを確認します。[ダッシュボード] をクリックし、パイプラインのダッシュボードを検索します。より多くのキー パフォーマンス インジケータについてレポートを作成するには、カスタム ダッシュボードを作成します。

詳細な例については、コードを my GitHub または GitLab リポジトリから Code Stream の自分のパイプラインに継続的に統合する方法を参照してください。

スマート パイプライン テンプレートの使用に先立つ Code Stream での継続的デリバリ ネイティブ ビルドの計画

Code Stream に継続的デリバリ (CD) パイプラインを作成するには、継続的デリバリ スマート パイプライン テンプレートを使用します。継続的デリバリ ネイティブ ビルドを計画するには、この例で示されている計画のパイプラインを作成する前にスマート パイプライン テンプレートの情報を収集します。

スマート パイプライン テンプレートに入力すると、リポジトリに継続的デリバリ パイプラインが作成され、パイプラインの実行に必要なアクションが実行されます。パイプラインの実行後、その実行のトレンドを監視できます。

継続的デリバリ スマート パイプライン テンプレートを使用する前にビルドを計画するには、次の手順に従います。

- パイプライン、エンドポイント、ダッシュボードをはじめすべての作業をグループ化するプロジェクトを特定します。
- スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画の継続的デリバリ部分の説明に従って、ビルドの情報を収集します。例：
  - Code Stream がコンテナを展開する Kubernetes エンドポイントを追加します。
  - 名前空間、サービス、展開を作成する Kubernetes YAML ファイルを準備します。プライベート所有のリポジトリからイメージをダウンロードするには、Docker 構成の [シークレット] を含むセクションを YAML ファイルに含める必要があります。

次に、継続的デリバリ スマート パイプライン テンプレートを使用してパイプラインを作成します。

[パイプライン] で、[スマート テンプレート] を選択します。

継続的デリバリ スマート パイプライン テンプレートを選択します。

テンプレートに入力し、パイプラインの名前を入力します。パイプラインによって作成されたステージとともにパイプラインを保存するには、[作成] をクリックします。

Code Stream パイプライン ワークスペースでは、継続的インテグレーション タスクとカスタム タスクで Docker と Kubernetes がサポートされます。

ワークスペースの構成については、パイプライン ワークスペースの構成を参照してください。

最終的な変更を行うために、パイプラインを編集できます。次に、パイプラインを有効にして実行できます。パイプラインの実行後：

- パイプラインが正常に完了したことを確認します。[実行] をクリックし、パイプラインを検索します。正常に完了しなかった場合は、エラーを修正して再度実行します。
- Git webhook が正しく動作していることを確認します。Git の [Activity] タブにイベントが表示されます。[トリガ] - [Git] - [アクティビティ] の順にクリックします。
- パイプライン ダッシュボードで、トレンドを確認します。[ダッシュボード] をクリックし、パイプライン ダッシュボードを検索します。より多くのキー パフォーマンス インジケータについてレポートを作成するには、カスタム ダッシュボードを作成します。

詳細な例については、コードを my GitHub または GitLab リポジトリから Code Stream の自分のパイプラインに継続的に統合する方法を参照してください。

タスクの手動追加を行う前の Code Stream での CICD ネイティブ ビルドの計画

Code Stream に継続的インテグレーションおよび継続的デリバリ (CICD) パイプラインを作成するには、手動でステージとタスクを追加します。CICD ネイティブ ビルドを計画するには、必要な情報を収集し、パイプラインを作成して、そのパイプラインにステージとタスクを手動で追加します。

パイプラインの継続的インテグレーション (CI) ステージと継続的デリバリ (CD) ステージの両方を計画する必要があります。パイプラインを作成して実行した後、その実行のトレンドを監視できます。

パイプラインに Docker Hub のイメージが含まれている場合は、パイプラインを実行する前に、イメージに `cURL` または `wget` が組み込まれていることを確認する必要があります。パイプラインが実行されると、 Code Stream は、`cURL` または `wget` を使用してコマンドを実行するバイナリ ファイルをダウンロードします。

Code Stream パイプライン ワークスペースでは、継続的インテグレーション タスクとカスタム タスクで Docker と Kubernetes がサポートされます。

ワークスペースの構成については、パイプライン ワークスペースの構成を参照してください。

外部および内部の要件の計画

パイプラインの CI ステージおよび CD ステージを計画するには、パイプラインを作成する前に実行する必要がある次の要件を満たす必要があります。

この例では、Docker ワークスペースを使用します。

この例の計画からパイプラインを作成するには、Docker ホスト、Git リポジトリ、Maven のほか、後処理ビルド ツールをいくつか使用します。

必要となるエンドポイントとリポジトリ：

- 開発者が自分のコードをチェックインする Git ソース コード リポジトリ。 Code Stream は、開発者が変更をコミットするときに、パイプラインに最新のコードを取得します。
- コンテナ内でビルド コマンドを実行する Docker ビルド ホストの Docker エンドポイント。
- 継続的インテグレーションのテストで実行するコンテナを作成するビルダー イメージ。
- Docker ビルド ホストからビルダー イメージを取得できるようにするためのイメージ レジストリ エンドポイント。

プロジェクトへのアクセス権が必要です。プロジェクトはパイプライン、エンドポイント、ダッシュボードなどすべての作業をグループ化します。 Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。

Git Webhook が必要になります。これにより、開発者がコード変更をコミットするときに Code Stream が Git トリガを使用してパイプラインがトリガされるようにできます。 Code Stream で Git トリガを使用してパイプラインを実行する方法を参照してください。

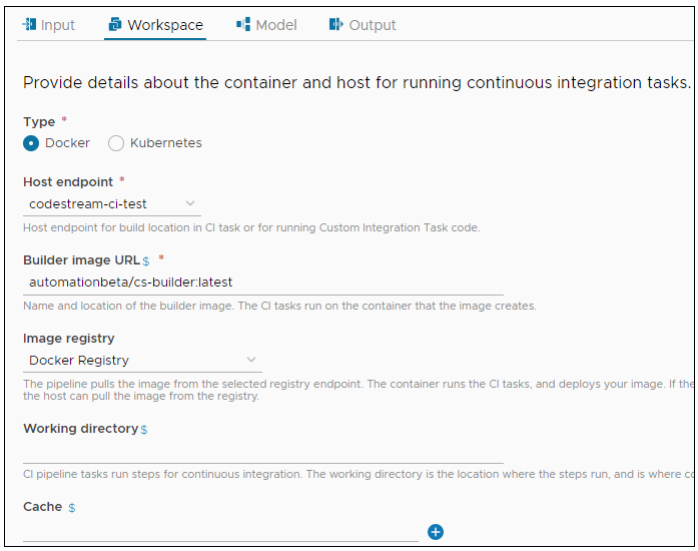
CICD パイプラインを作成してワークスペースを設定する方法

パイプラインを作成してから、ワークスペース、パイプライン入力パラメータ、およびタスクを設定する必要があります。

パイプラインを作成するには、[パイプライン] - [新しいパイプライン] - [空白のキャンバス] の順にクリックします。

[ワークスペース] タブで、継続的インテグレーション情報を入力します。

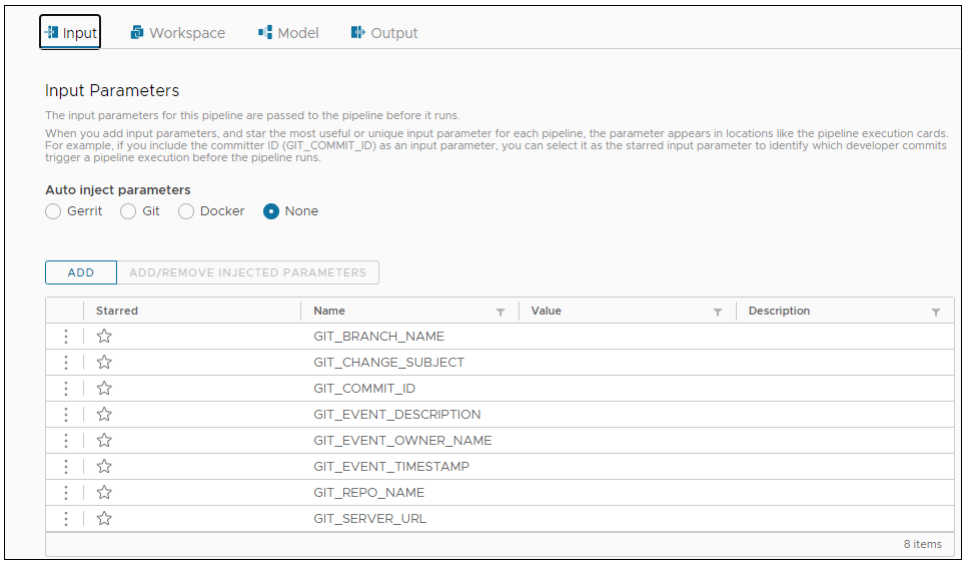
- Docker ビルド ホストを含めます。
- ビルダ イメージの URL を入力します。
- イメージ レジストリ エンドポイントを選択して、そのエンドポイントからパイプラインがイメージをプルできるようにします。コンテナが CI タスクを実行し、イメージを展開します。レジストリに認証情報が必要な場合は、まずイメージ レジストリ エンドポイントを作成する必要があります。次に、ここでそれを選択して、そのレジストリからホストがイメージをプルできるようにします。
- キャッシュする必要があるアーティファクトを追加します。ビルドが正常に完了した場合は、ディレクトリなどのアーティファクトが依存関係としてダウンロードされます。キャッシュは、このようなアーティファクトが配置される場所です。たとえば、依存関係のあるアーティファクトには、Maven の `.m2` ディレクトリや、Node.js の `node_modules` ディレクトリなどがあります。これらのディレクトリは、パイプライン実行をまたがってキャッシュされるため、ビルドの時間が短縮します。



[入力] タブで、パイプラインの入力パラメータを構成します。

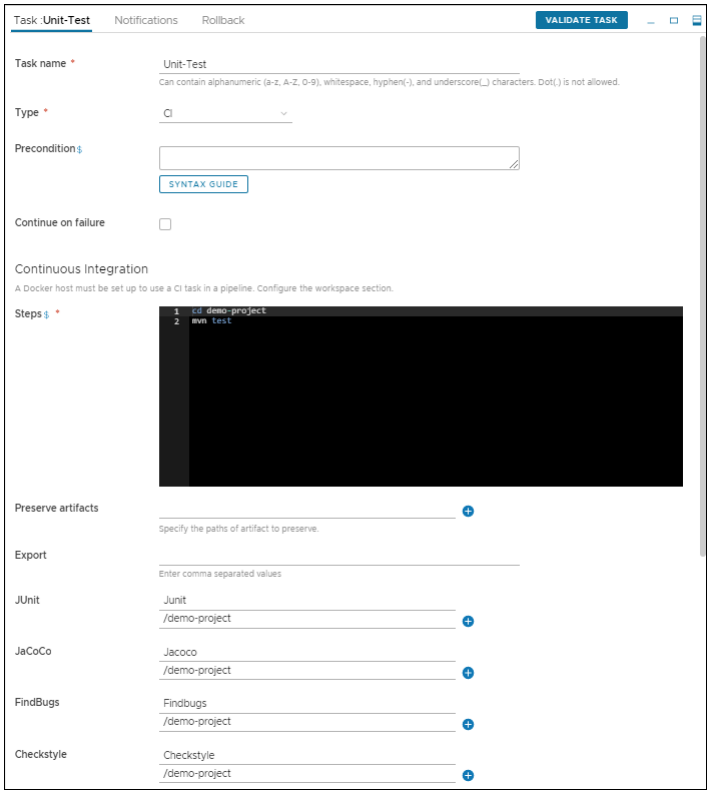
- パイプラインが Git、Gerrit、または Docker トリガ イベントからの入力パラメータを使用する場合は、自動挿入パラメータのトリガ タイプを選択します。イベントには、Gerrit または Git のサブジェクトの変更、または Docker のイベント所有者名を含めることができます。パイプラインでイベントから渡された入力パラメータを使用しない場合は、自動挿入パラメータを [なし] に設定します。
- パイプラインの入力パラメータに値と説明を適用するには、縦に並んだ 3 つのドットをクリックし、[編集] をクリックします。入力した値は、タスク、ステージ、通知への入力値として使用されます。

- パイプラインの入力パラメータを追加するには、[追加] をクリックします。たとえば、各実行でデフォルト値を表示する `approvers` を追加できますが、ランタイムに別の承認者でオーバーライドする可能性があります。
- 挿入されたパラメータを追加または削除するには、[挿入したパラメータの追加/削除] をクリックします。たとえば、使用されていないパラメータを削除して結果ページの不要な情報を減らし、使用されている入力パラメータのみを表示することができます。



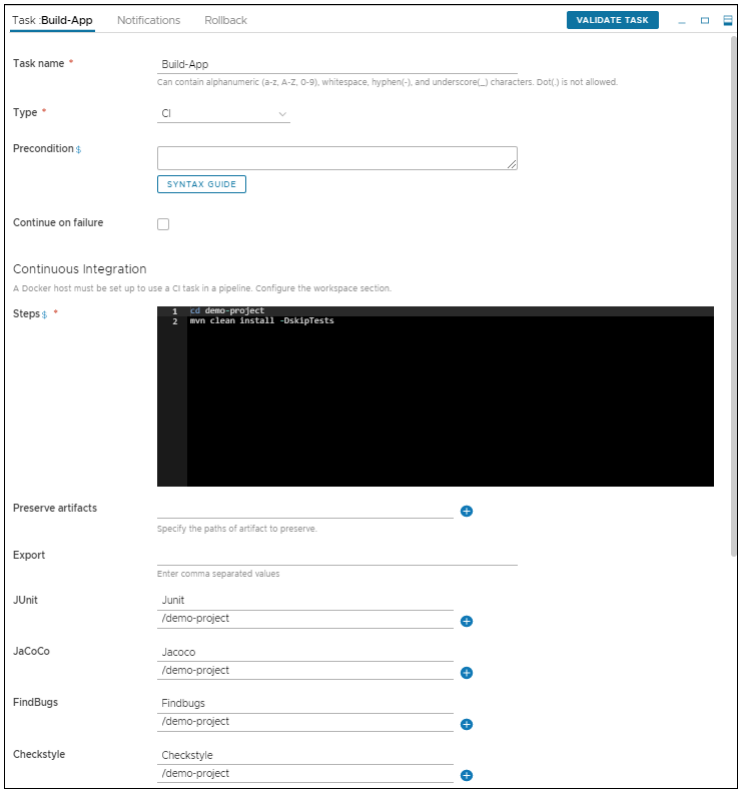
パイプラインを設定してコードをテストします。

- CI タスクを追加して設定します。
- コードに `mvn test` を実行するための手順を含めます。
- タスクの実行後に問題を調べるには、JUnit および JaCoCo、FindBugs、Checkstyle などの後処理ビルドツールを実行します。



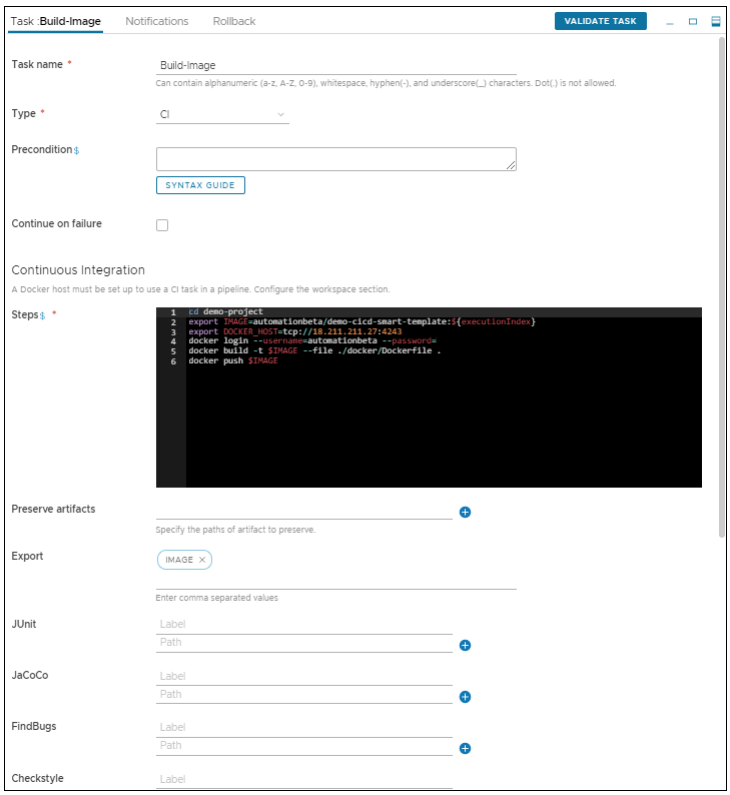
パイプラインを設定してコードをビルドします。

- CI タスクを追加して設定します。
- コードに `mvn clean install` を実行する手順を含めます。
- アーティファクトを保持する JAR の場所とファイル名を含めます。



パイプラインを設定してイメージを Docker ホストに公開します。

- CI タスクを追加して設定します。
- イメージのコミット、エクスポート、ビルド、プッシュを行う手順を追加します。
- 次のタスクで使用するために `IMAGE` のエクスポート キーを追加します。



ワークスペース、入力パラメータ、テスト タスク、およびビルド タスクを設定したら、パイプラインを保存します。

パイプラインを有効にして実行する方法

ステージとタスクを使用してパイプラインを構成したら、パイプラインを保存して有効にすることができます。

次に、パイプラインが実行されて終了するまで待機して、正常に完了したことを確認します。正常に完了しなかった場合は、エラーを修正して再度実行します。

パイプラインが正常に完了したら、次の点を確認します。

- パイプラインの実行状況を確認し、タスクの実行手順の結果を表示します。
- パイプライン実行のワークスペースで、コンテナおよびクローン作成された Git リポジトリに関する詳細を見つけます。
- ワークスペースで、後処理ツールの結果を確認し、エラー、コード カバレッジ、バグ、およびスタイルの問題がないか確認します。
- アーティファクトが保持されていることを確認します。また、イメージがイメージ名と値を使用してエクスポートされたことを確認します。
- Docker リポジトリに移動し、パイプラインがコンテナを公開していることを確認します。

Code Stream がコードを継続的にどのように統合するかを示す詳細な例については、コードを my GitHub または GitLab リポジトリから Code Stream の自分のパイプラインに継続的に統合する方法を参照してください。

Code Stream でのロールバックの計画

パイプラインの実行に失敗した場合は、ロールバックを使用して、環境を以前の安定した状態に戻すことができます。ロールバックを使用するには、ロールバック フローを計画し、その実装方法を把握します。

ロールバック フローは、展開中の障害を元に戻すために必要な手順を規定します。フローは、ロールバック パイプラインの形式を取り、実行および失敗した展開のタイプに応じて異なる 1 つ以上の連続したタスクを含みます。たとえば、従来のアプリケーションの展開とロールバックは、 コンテナ アプリケーションの展開とロールバックとは異なります。

適切なデプロイ状態に戻すために、ロールバック パイプラインには通常、次のタスクが含まれます。

- 状態または環境のクリーンアップ。
- 変更を元に戻すためのユーザー指定スクリプトの実行。
- 以前のバージョンの環境の展開。

既存の展開パイプラインにロールバックを追加するには、展開パイプラインの実行前に、ロールバックする展開パイプライン内のタスクまたはステージにロールバック パイプラインを適用します。

ロールバックの設定

展開環境でロールバックを構成するには、以下を実行する必要があります。

- 展開プロファイルを作成します。
- ロールバック パイプラインを適用できるように、ロールバックをトリガする展開パイプラインの潜在的な障害ポイントを特定します。たとえば、ロールバック パイプラインを、前のタスクが正常に完了したかどうかを確認する展開パイプラインの条件またはポーリング タスク タイプに適用することができます。条件タスクの詳細については、条件タスクで変数バインドを使用して、Code Stream でパイプラインを実行または停止する方法を参照してください。
- タスクやステージの障害など、ロールバック パイプラインをトリガする障害の範囲を特定します。ロールバックをステージに接続することもできます。
- 障害が発生した場合に実行するロールバック タスクを決定します。ロールバック パイプラインは、これらのタスクを使用して作成します。

ロールバック パイプラインは手動で作成できます。また、 Code Stream で作成することもできます。

- 空のキャンバスを使用して、既存の展開パイプラインに並行するフローに基づくロールバック パイプラインを手動で作成できます。次に、ロールバック パイプラインを、 障害発生時にロールバックをトリガする展開パイプライン内の 1 つ以上のタスクに適用します。
- スマート パイプライン テンプレートを使用すると、ロールバック アクションを使用して展開パイプラインを構成できます。その後、Code Stream は、障害発生時に展開をロールバックする事前定義されたタスクを使用して、1 つ以上のデフォルト ロールバック パイプラインを自動的に作成します。

スマート パイプライン テンプレートを使用してロールバックを含む CD パイプラインを構成する方法の詳細な例については、 Code Stream で展開をロールバックする方法を参照してください。

展開パイプラインにロールバックを含む複数のタスクまたはステージがある場合

複数のタスクまたはタスクとステージにロールバックを追加した場合は、ロールバックの順序が変わることに注意してください。

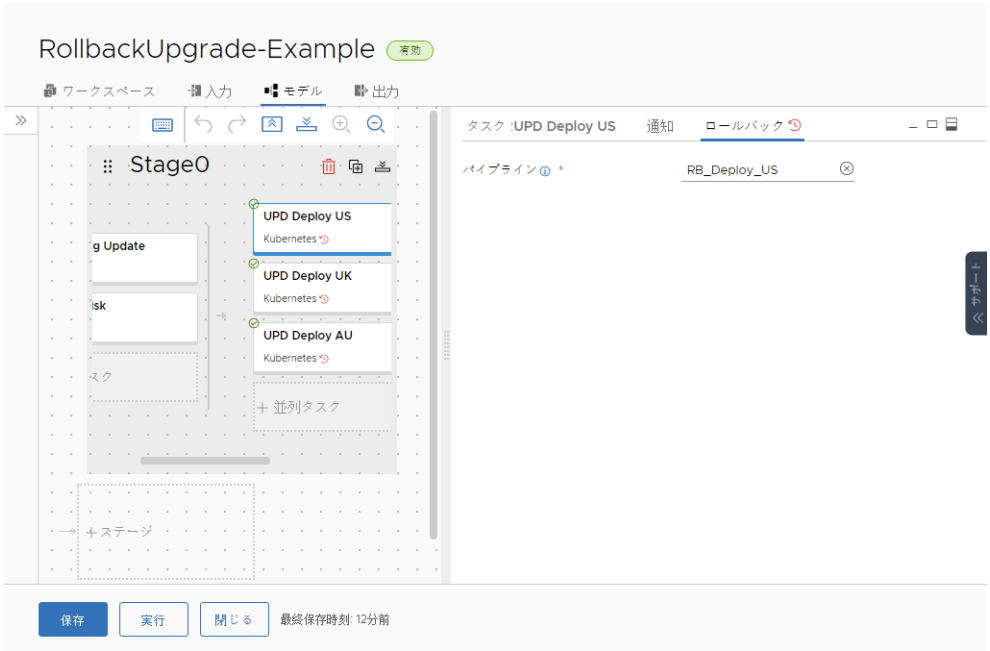
表 4-3. ロールバック シーケンスの決定

ロールバックの追加先	ロールバックのタイミング
並行タスク	並行タスクのいずれかが失敗した場合、すべての並列タスクの完了または失敗後に、タスクのロールバックが実行されます。ロールバックはタスクが失敗した直後には実行されません。
ステージ内のタスクとステージの両方	タスクが失敗した場合、タスクのロールバックが実行されます。タスクが並行タスクのグループに含まれている場合、タスクのロールバックはすべての並列タスクの完了または失敗後に実行されます。ステージのロールバックは、タスクのロールバックが完了するか、または完了に失敗した後に実行されます。

次のようなパイプラインを考えます。

- 本番ステージにロールバックが含まれる。
- 並行タスクのグループで、各タスクに独自のロールバックがある。

[UPD Deploy US] という名前のタスクにロールバック パイプライン [RB_Deploy_US] があります。[UPD Deploy US] が失敗すると、ロールバックは [RB_Deploy_US] パイプラインで定義されたフローに基づいて実行されます。



[UPD Deploy US] が失敗すると、[RB_Deploy_US] パイプラインは、[UPD Deploy UK] と [UPD Deploy AU] の完了または失敗後に実行されます。ロールバックは [UPD Deploy US] が失敗した直後には実行されません。また、本番ステージにもロールバックがあるため、ステージのロールバック パイプラインは [RB_Deploy_US] パイプラインの実行後に実行されます。

Code Stream を使用するためのチュートリアル

5

Code Stream は、DevOps リリース ライフサイクルをモデル化およびサポートし、アプリケーションを継続的にテストして展開環境および本番環境にリリースします。

Code Stream を使用するために必要なすべてのことはすでに設定しています。2 章 リリース プロセスをモデリングするための Code Stream の設定を参照してください。

本番環境にリリースする前に、開発者コードのビルドとテストを自動化するパイプラインを作成できます。 Code Stream によってコンテナベースまたは従来のアプリケーションを展開することができます。

表 5-1. DevOps ライフサイクルにおける Code Stream の使用

機能	実行できる操作の例
Code Stream でネイティブ ビルド機能を使用する。	コードを継続的に統合、コンテナ化、および配信するための継続的インテグレーションおよび継続的デリバリ (CICD) パイプライン、継続的インテグレーション (CI) パイプライン、および継続的デリバリ (CD) パイプラインを作成します。 ■ パイプラインを自動的に作成するスマート パイプライン テンプレートを使用します。 ■ パイプラインにステージとタスクを手動で追加します。
アプリケーションをリリースし、リリースを自動化する。	さまざまな方法でアプリケーションを連携およびリリースします。 ■ GitHub または GitLab リポジトリのコードをパイプラインに継続的に連携します。 ■ ブログ記事「Creating a Docker host for vRealize Automation Code Stream」の手順に沿って、Docker ホストを統合して継続的インテグレーション タスクを実行します。 ■ YAML クラウド テンプレートを使用して、アプリケーションの展開を自動化します。 ■ Kubernetes クラスタへのアプリケーションの展開を自動化します。 ■ アプリケーションをブルーグリーン展開にリリースします。 ■ Code Stream を自分のビルド、テスト、展開ツールと統合します。 ■ 他のアプリケーションと Code Stream を統合する REST API を使用します。
トレンド、メトリック、キー パフォーマンス インジケータ (KPI) を追跡します。	カスタム ダッシュボードを作成し、パイプラインのパフォーマンスを管理します。
問題を解決する。	パイプラインの実行に失敗した場合は、 Code Stream を使用して Jira チケットを作成します。

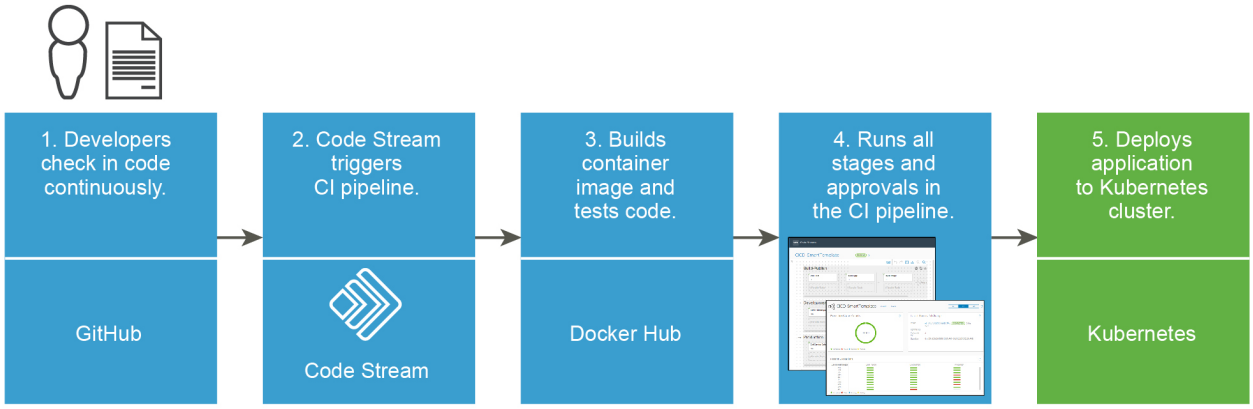
この章には、次のトピックが含まれています。

- コードを my GitHub または GitLab リポジトリから Code Stream の自分のパイプラインに継続的に統合する方法
- Code Stream の YAML クラウド テンプレートから展開するアプリケーションのリリースを自動化する方法

- Kubernetes クラスタへの Code Stream のアプリケーションのリリースを自動化する方法
- Code Stream のアプリケーションをブルーグリーン展開に展開する方法
- ビルド、テスト、展開用の独自のツールを Code Stream に統合する方法
- 次のタスクでクラウド テンプレート タスクのリソース プロパティを使用する方法
- REST API を使用して Code Stream を他のアプリケーションと統合する方法
- パイプラインを Code Stream のコードとして利用する方法

コードを my GitHub または GitLab リポジトリから Code Stream の自分のパイプラインに継続的に統合する方法

開発者は、GitHub リポジトリまたは GitLab Enterprise リポジトリからコードを継続的に統合する必要があります。開発者がコードを更新し、変更をリポジトリにコミットするたびに、 Code Stream はその変更を待機し、パイプラインをトリガできます。



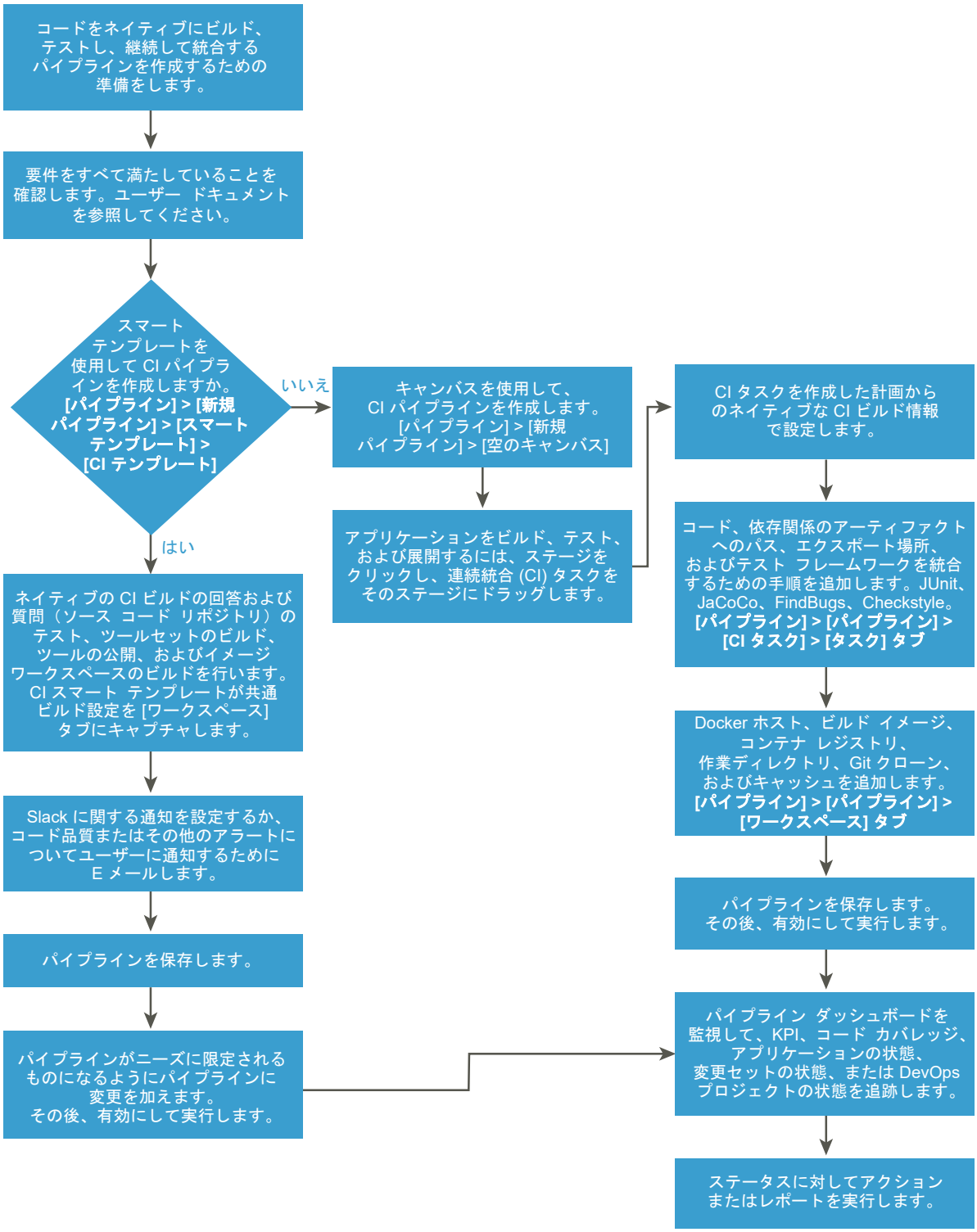
コードが変更されたら Code Stream がパイプラインをトリガするようにするには、Git トリガを使用します。これにより、コードに変更をコミットするたびに Code Stream がパイプラインをトリガします。

Code Stream パイプライン ワークスペースでは、継続的インテグレーション タスクとカスタム タスクで Docker と Kubernetes がサポートされます。

ワークスペースの構成に関する詳細については、パイプライン ワークスペースの構成を参照してください。

次のフローチャートは、スマート パイプライン テンプレートを使用してパイプラインを作成する場合、またはパイプラインを手動でビルドする場合に使用できるワークフローを示しています。

図 5-1. スマート パイプライン テンプレートを使用する、または手動でパイプラインを作成するワークフロー



次の例では、Docker ワークスペースを使用します。

コードをビルドするには、Docker ホストを使用します。テスト フレームワーク ツールとして JUnit と JaCoCo を使用します。単体テストとコード カバレッジを実行するもので、どちらもパイプラインに含めます。

これにより、継続的インテグレーション スマート パイプライン テンプレートを使用して継続的インテグレーション パイプラインを作成できます。このパイプラインは、コードをビルドし、テストしてから、AWS 上にあるプロジェクト チームの Kubernetes クラスタに展開します。コード ビルドの時間を節約するために継続的インテグレーション タスクのコード依存関係アーティファクトを保存するには、キャッシュを使用します。

コードをビルドしてテストするパイプライン タスクには、複数の継続的インテグレーション手順を含めることができます。この継続的インテグレーション手順は、パイプラインがトリガされたときに Code Stream によってソース コードがクローン作成されるのと同じ作業ディレクトリに配置されます。

Kubernetes クラスタにコードを展開するために、パイプラインで Kubernetes タスクを使用できます。その後、パイプラインを有効にして実行します。次に、リポジトリのコードに変更を加え、パイプラインのトリガを監視します。パイプラインの実行後にパイプラインのトレンドを監視およびレポートするには、ダッシュボードを使用します。

次の例では、コードをパイプラインに継続的に統合する継続的インテグレーション パイプラインを作成するために、継続的インテグレーション スマート パイプライン テンプレートを使用します。この例では、Docker ワークスペースを使用します。

必要に応じて、パイプラインを手動で作成し、ステージとタスクを追加できます。継続的インテグレーション ビルドの計画とパイプラインの手動作成に関する詳細については、タスクの手動追加を行う前の Code Stream での CICD ネイティブ ビルドの計画を参照してください。

前提条件

- 継続的インテグレーション ビルドを計画します。スマート パイプライン テンプレートの使用に先立つ Code Stream での継続的インテグレーション ネイティブ ビルドの計画を参照してください。
- GitLab ソース コード リポジトリが配置されていることを確認します。ヘルプについては、 Code Stream 管理者に確認してください。
- Git エンドポイントを追加します。例については、Code Stream で Git トリガを使用してパイプラインを実行する方法を参照してください。
- Code Stream が GitHub リポジトリまたは GitLab リポジトリの変更を待機し、変更が発生したらパイプラインをトリガするようにするには、Webhook を追加します。例については、 Code Stream で Git トリガを使用してパイプラインを実行する方法を参照してください。
- Docker ホスト エンドポイントを追加します。継続的インテグレーション タスク用のコンテナを作成し、それを複数の継続的インテグレーション タスクで使用できるようにするためのものです。エンドポイントの詳細については、 Code Stream でのエンドポイントとは を参照してください。
- イメージ URL、ビルド ホスト、およびビルド イメージの URL を取得します。ヘルプについては、 Code Stream 管理者に確認してください。
- テスト フレームワーク ツールに JUnit と JaCoCo を使用していることを確認します。
- 継続的インテグレーション ビルド用に外部インスタンスを設定します。Jenkins、TFS、または Bamboo です。Kubernetes プラグインによってコードが展開されます。ヘルプについては、 Code Stream 管理者に確認してください。

手順

1 前提条件を満たします。

2 スマート パイプライン テンプレートを使用してパイプラインを作成するために、継続的インテグレーション スマート パイプライン テンプレートを開き、フォームに入力します。

   a [パイプライン] - [新しいパイプライン] - [スマート テンプレート] - [継続的インテグレーション] の順にクリックします。

   b ソース コード リポジトリ、ビルド ツールセット、公開ツール、およびビルド イメージ ワークスペースに関するテンプレートの質問に回答します。

   c チームの Slack 通知または E メール通知を追加します。

   d スマート パイプライン テンプレートでパイプラインを作成するには、[作成] をクリックします。

   e パイプラインをさらに変更するには、[編集] をクリックし、変更を加えてから、[保存] をクリックします。

   f パイプラインを有効にして実行します。

3 パイプラインを手動で作成するには、ステージとタスクをキャンバスに追加し、ネイティブの継続的インテグレーション ビルド情報を継続的インテグレーション タスクに含めます。

   a [パイプライン] - [新しいパイプライン] - [空白のキャンバス] の順にクリックします。

   b ステージをクリックし、ナビゲーション ペインからステージに継続的インテグレーション タスクをいくつかドラッグします。

   c 継続的インテグレーション タスクを設定するには、そのタスクをクリックし、[タスク] タブをクリックします。

   d コードを継続的に統合する手順を追加します。

   e 依存関係アーティファクトへのパスを含めます。

   f エクスポート先を追加します。

   g 使用するテスト フレームワーク ツールを追加します。

   h Docker ホストとビルド イメージを追加します。

   i コンテナ レジストリ、作業ディレクトリ、およびキャッシュを追加します。

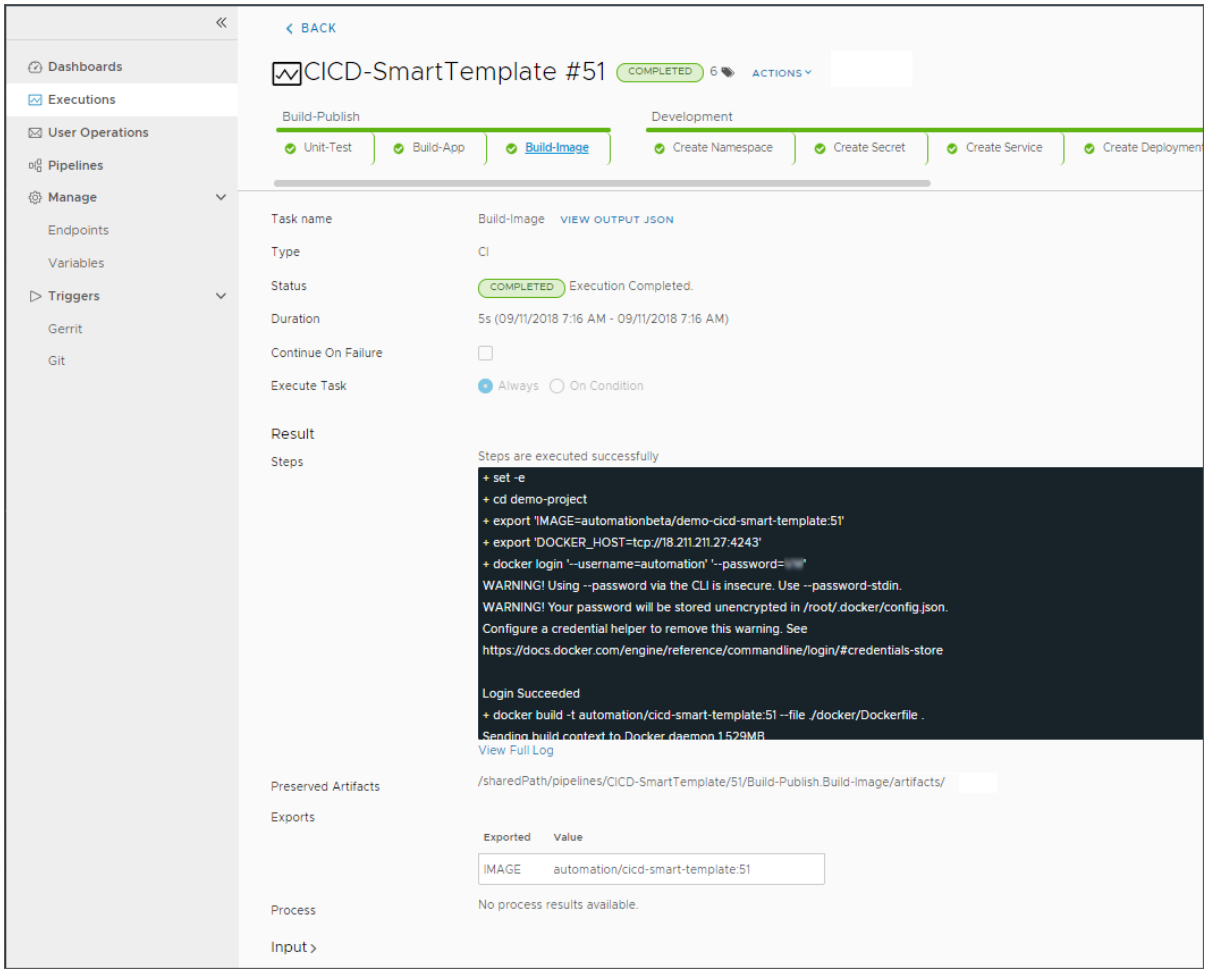
   j パイプラインを保存して有効にします。

4 GitHub リポジトリまたは GitLab リポジトリのコードに変更を加えます。

   Git をトリガしてパイプラインを有効にすると、実行が開始されます。

5 コードの変更によってパイプラインがトリガされたかどうかを確認するには、[トリガ] - [Git] - [アクティビティ] の順にクリックします。

6 パイプラインの実行状況を表示するには、[実行] をクリックします。ビルド イメージが作成されてエクスポートされたことを確認してください。



7 パイプライン ダッシュボードを監視して KPI とトレンドを追跡するには、[ダッシュボード] - [パイプライン ダッシュボード] の順にクリックします。

結果

完了です。コードを GitHub リポジトリまたは GitLab リポジトリからパイプラインに継続的に統合し、ビルド イメージを展開するパイプラインが作成されました。

次のステップ

詳細については、 Code Stream 管理者および開発者向けのその他のリソースを参照してください。

Code Stream の YAML クラウド テンプレートから展開するアプリケーションのリリースを自動化する方法

開発者は、変更をコミットするたびにオンプレミスの GitHub インスタンスから自動化クラウド テンプレートを取得するパイプラインを必要とします。このパイプラインは、WordPress アプリケーションを Amazon Web Services (AWS) EC2 またはデータセンターに展開するために必要です。 Code Stream は、パイプラインからク

ラウド テンプレートを呼び出し、アプリケーションの展開用にそのクラウド テンプレートの継続的インテグレーションおよび継続的デリバリ (CICD) を自動化します。

パイプラインを作成してトリガするには、VMware クラウド テンプレートが必要です。

Code Stream クラウド テンプレート タスクの [クラウド テンプレート ソース] で、次のいずれかを選択します。

- ソース制御としての [Cloud Assembly テンプレート]。この場合、GitLab または GitHub リポジトリは必要ありません。
- ソース制御に GitLab または GitHub を使用している場合、[ソース制御]。この場合は Git Webhook が必要であり、Webhook を介してパイプラインをトリガーする必要があります。

GitHub リポジトリに YAML クラウド テンプレートがあり、パイプラインでそのクラウド テンプレートを使用する場合は、次の操作を実行します。

1 Cloud Assembly で、クラウド テンプレートを GitHub リポジトリにプッシュします。

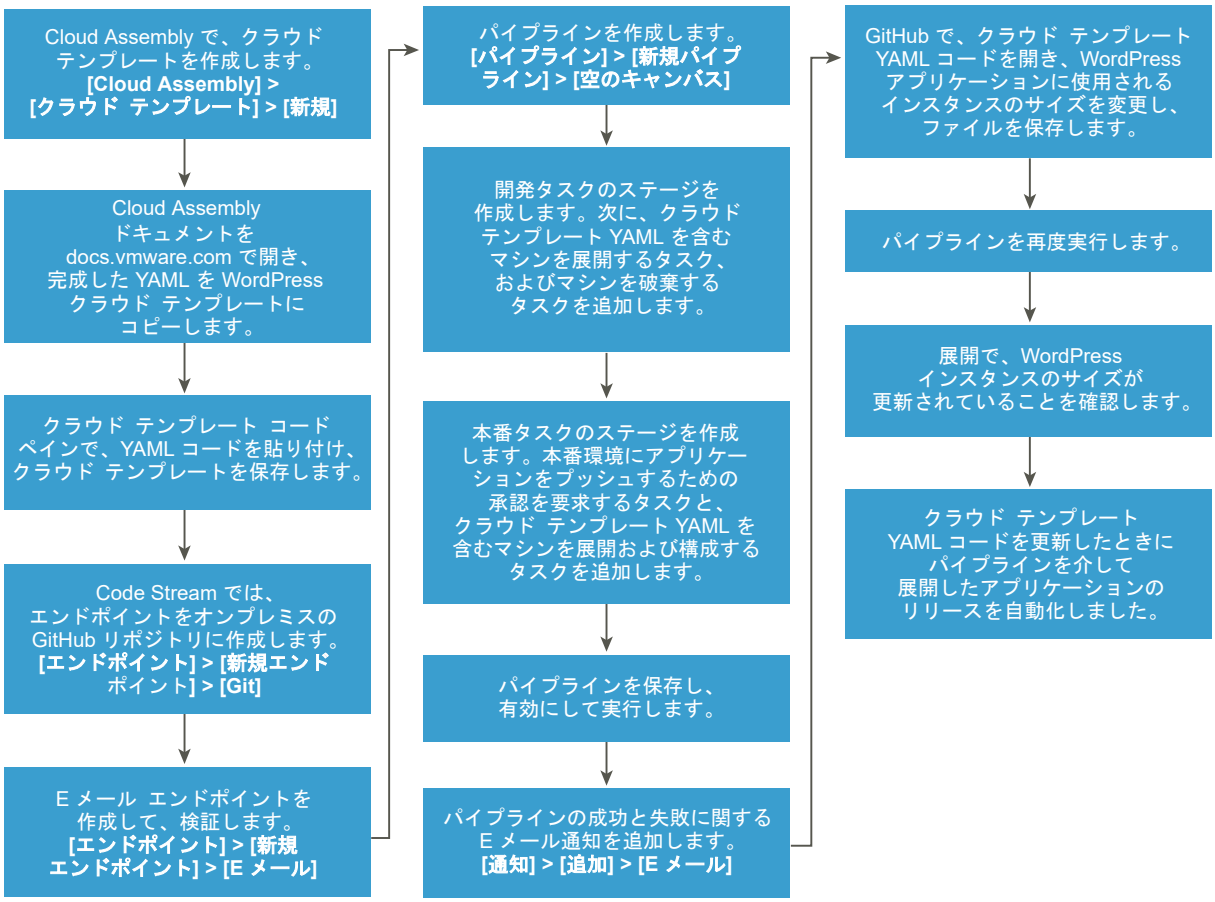
2 Code Stream で、Git エンドポイントを作成します。次に、この Git エンドポイントとパイプラインを使用する Git Webhook を作成します。

3 パイプラインをトリガーするには、GitHub リポジトリ内のファイルを更新し、変更をコミットします。

GitHub リポジトリに YAML クラウド テンプレートがなく、ソース制御のクラウド テンプレートを使用する場合は、この手順を使用してその方法を学習してください。ここでは、WordPress アプリケーション向けにクラウド テンプレートを作成し、オンプレミスの GitHub リポジトリから起動する方法について説明します。YAML クラウド テンプレートに変更を加えると、パイプラインによってアプリケーションのリリースがトリガおよび自動化されます。

- Cloud Assembly で、クラウド アカウントを追加し、クラウド ゾーンを追加して、クラウド テンプレートを作成します。
- Code Stream で、クラウド テンプレートをホストするオンプレミスの GitHub リポジトリにエンドポイントを追加します。次に、クラウド テンプレートをパイプラインに追加します。

このユースケースの例では、オンプレミスの GitHub リポジトリにあるクラウド テンプレートを使用する方法を示します。



前提条件

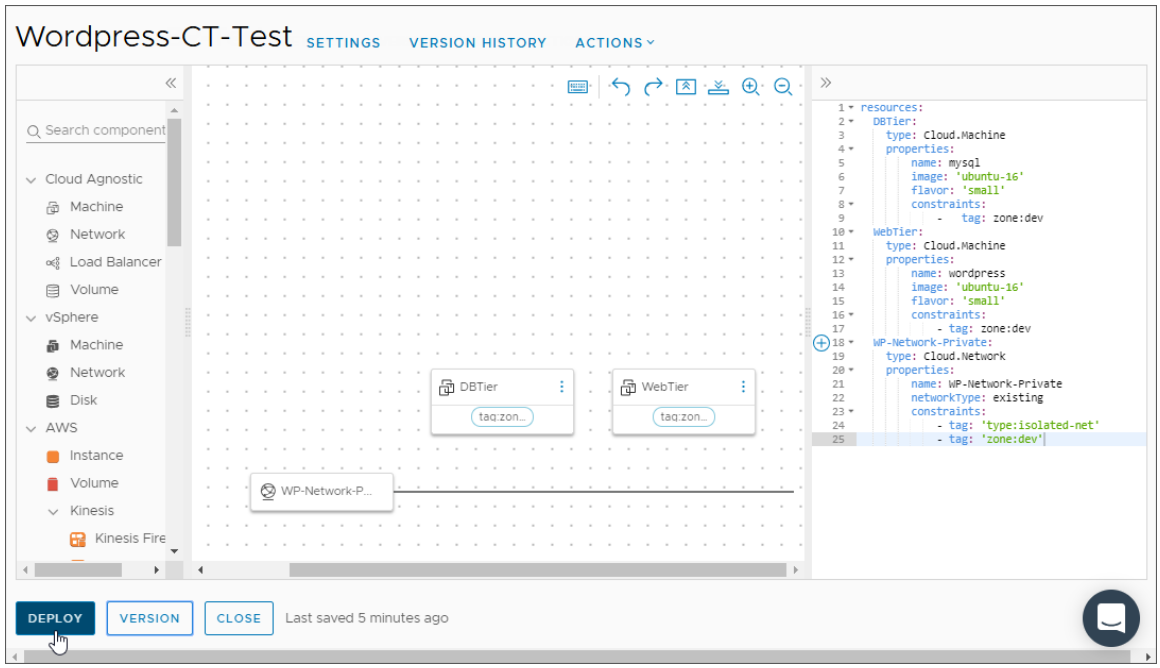
- vRealize Automation Cloud Assembly インフラストラクチャにクラウド アカウントとクラウド ゾーンを追加します。『vRealize Automation Cloud Assembly のドキュメント』を参照してください。
- 次の手順でクラウド テンプレートを作成するには、WordPress YAML コードをクリップボードにコピーします。vRealize Automation Cloud Assembly のドキュメントで WordPress 使用事例のクラウド テンプレート YAML コードを参照してください。
- WordPress アプリケーションの YAML コードを GitHub インスタンスに追加します。
- YAML コードに変更内容をコミットするたびにパイプラインがその YAML コードをプルできるように、Git トリガ用の Webhook を追加します。 Code Stream で、[トリガ] - [Git] - [Git 用 Webhook] の順にクリックします。
- クラウド テンプレート タスクを使用するには、Cloud Assembly のいずれかのロールを持っている必要があります。

手順

1 Cloud Assembly で、次の手順を実行します。

a [VMware Cloud Templates] をクリックし、WordPress アプリケーションのクラウド テンプレートと展開を作成します。

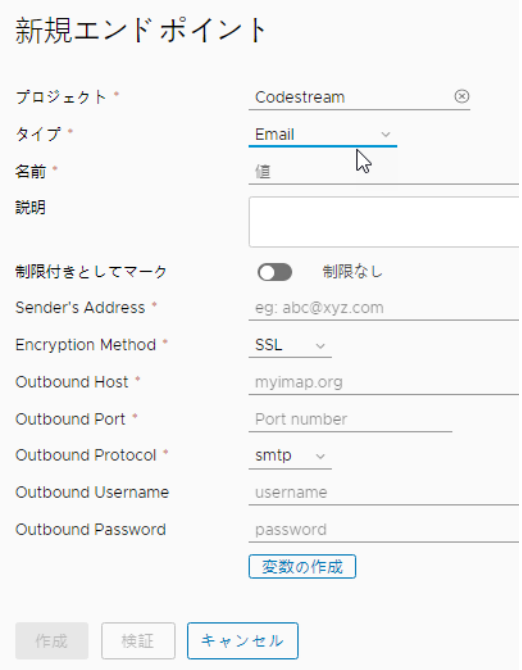
b クリップボードにコピーしておいた WordPress YAML コードをクラウド テンプレートに貼り付け、それを展開します。



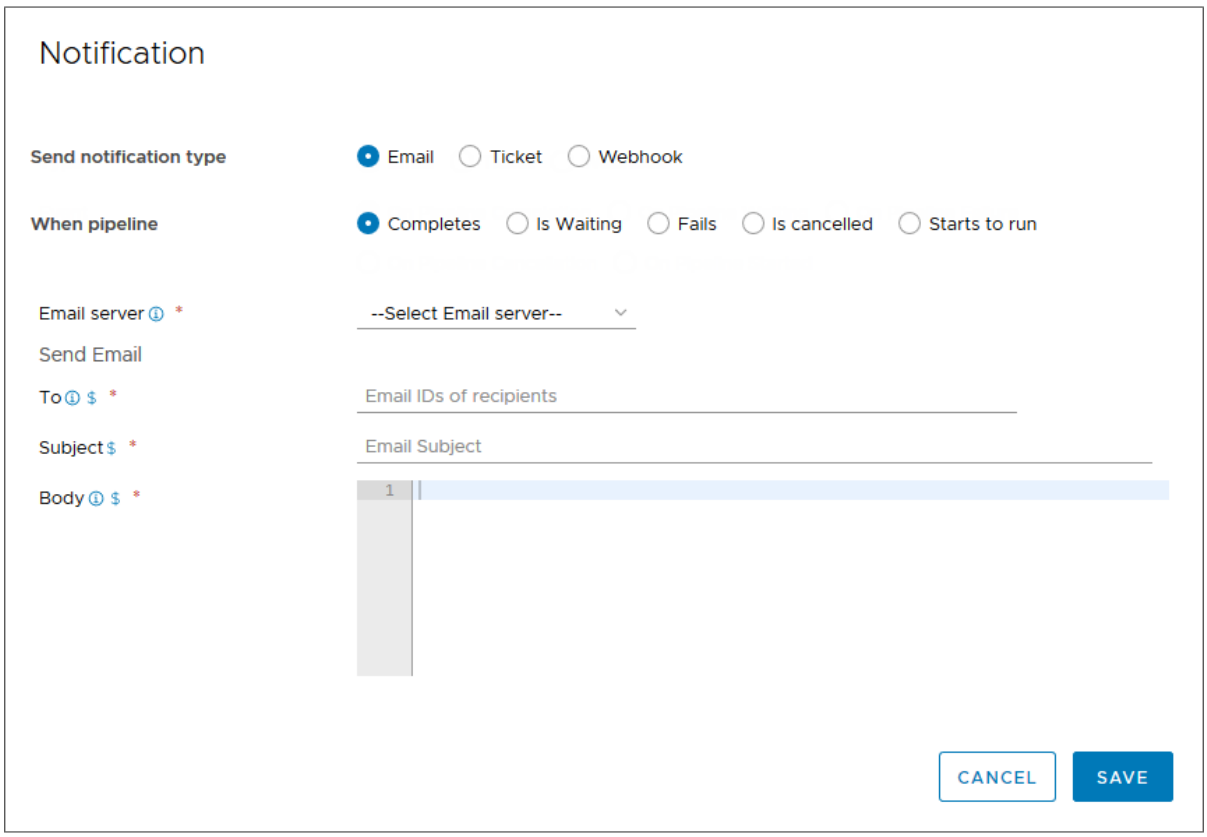
2 Code Stream で、エンドポイントを作成します。

a YAML ファイルが配置されているオンプレミス GitHub リポジトリ用に Git エンドポイントを作成します。

b パイプラインの実行時のステータスをユーザーに通知する E メール エンドポイントを追加します。



3 パイプラインを作成し、パイプラインの成否の通知を追加します。



4 開発のステージを追加し、クラウド テンプレート タスクを追加します。

a マシンを展開するクラウド テンプレート タスクを追加して、WordPress アプリケーションのクラウド テンプレート YAML を使用するように設定します。

```
resources:
  DBTier:
    type: Cloud.Machine
    properties:
        name: mysql
        image: 'ubuntu-16'
        flavor: 'small'
        constraints:
            -   tag: zone:dev
  WebTier:
    type: Cloud.Machine
    properties:
        name: wordpress
        image: 'ubuntu-16'
        flavor: 'small'
        constraints:
            - tag: zone:dev
 WP-Network-Private:
    type: Cloud.Network
    properties:
        name: WP-Network-Private
        networkType: existing
        constraints:
            - tag: 'type:isolated-net'
            - tag: 'zone:dev'
```

b マシンを破棄してリソースを解放するクラウド テンプレート タスクを追加します。

5 本番用のステージを追加し、承認タスクと展開タスクを含めます。

a WordPress アプリケーションを本番にプッシュするための承認を要求するユーザー操作タスクを追加します。

b マシンを展開するためのクラウド テンプレート タスクを追加し、そこに WordPress アプリケーションのクラウド テンプレート YAML を設定します。

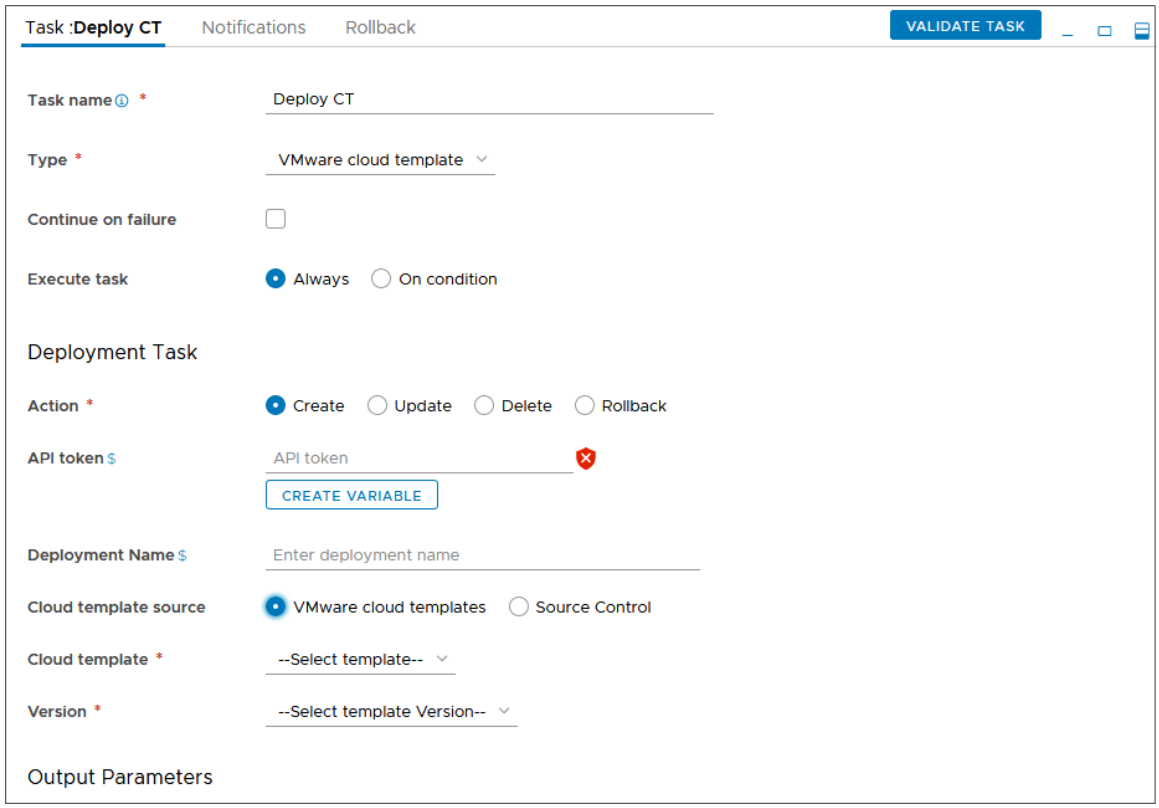
[作成] を選択するときは、展開の名前を一意にする必要があります。名前を空白のままにすると、 Code Stream によって一意のランダムな名前が割り当てられます。

自分のユースケースで [ロールバック] を選択する場合には、次のことを知っておく必要があります。[ロールバック] アクションを選択し、[ロールバック バージョン] を入力する場合、バージョンは **n-X** という形式にする必要があります。たとえば、**n-1**、**n-2**、**n-3** などとします。 Code Stream 以外の場所で展開を作成して更新した場合、ロールバックは許可されます。

Code Stream にログインすると、30 分間有効なユーザー トークンが取得されます。パイプラインの実行時間が長時間になる場合については、クラウド テンプレート タスクの前のタスクの実行に 30 分以上かかるとユーザー トークンの有効期限が切れます。その結果、クラウド テンプレート タスクは失敗します。

パイプラインを 30 分以上実行できるようにするには、オプションの API トークンを入力します。 Code Stream がクラウド テンプレートを呼び出すと API トークンが維持され、クラウド テンプレート タスクは引き続きその API トークンを使用します。

API トークンを変数として使用すると、そのトークンは暗号化されます。それ以外の場合は、プレーン テキストとして使用されます。



6 パイプラインを実行します。

各タスクが正常に完了したことを確認するには、実行中のタスクをクリックし、展開の詳細でステータスを確認して、詳細なリソース情報を調べます。

7 GitHub で、WordPress サーバ インスタンスのフレーバーを `small` から `medium` に変更します。

変更をコミットすると、パイプラインがトリガされます。これにより、更新されたコードが GitHub リポジトリからプルされて、アプリケーションがビルドされます。

```
WebTier:
    type: Cloud.Machine
    properties:
        name: wordpress
        image: 'ubuntu-16'
        flavor: 'medium'
        constraints:
            - tag: zone:dev
```

8 このパイプラインを再度実行します。パイプラインが正常に完了したことと、WordPress インスタンスのフレーバーが `small` から `medium` に変更されたことを確認します。

結果

完了です。YAML クラウド テンプレートから展開したアプリケーションのリリースが自動化されました。

次のステップ

Code Stream の使用方法の詳細については、5 章 Code Stream を使用するためのチュートリアルを参照してください。

この他の参考情報については、 Code Stream 管理者および開発者向けのその他のリソースを参照してください。

Kubernetes クラスタへの Code Stream のアプリケーションのリリースを自動化する方法

Code Stream 管理者または開発者は、 Code Stream および VMware Tanzu Kubernetes Grid Integrated Edition（旧称 VMware Enterprise PKS）を使用して、ソフトウェア プリケーションの Kubernetes クラスタへの展開を自動化できます。この使用事例では、アプリケーションのリリースを自動化するためのその他の方法を紹介します。

この使用事例では、2 つのステージを含むパイプラインを作成し、Jenkins を使用してアプリケーションをビルドして展開します。

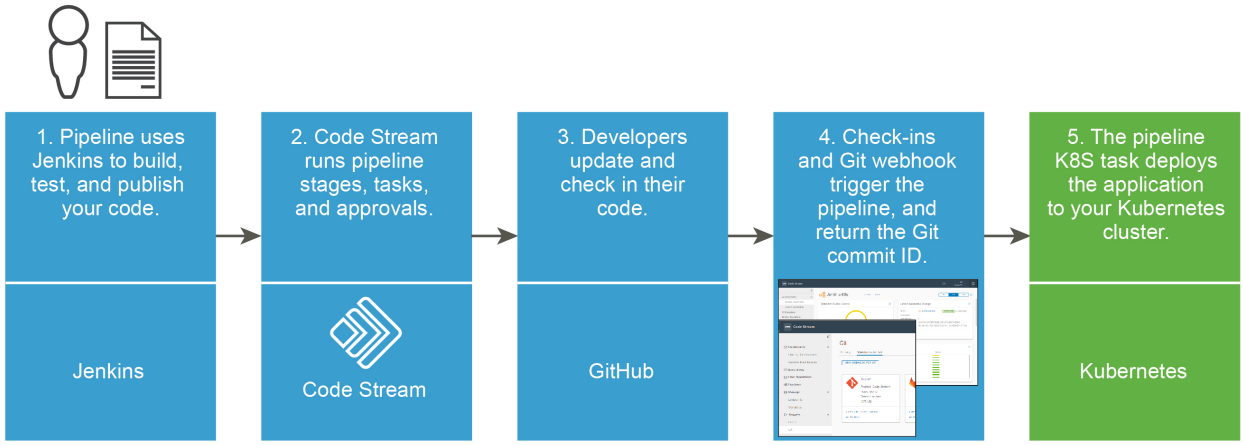
- 最初のステージは開発用です。Jenkins を使用して、GitHub リポジトリ内のブランチからコードを取得し、ビルドおよびテストして、公開します。
- 2 番目のステージは、展開用です。パイプラインがアプリケーションを Kubernetes クラスタに展開する前に、主要なユーザーの承認を必要とするユーザー操作タスクを実行します。

パイプライン ワークスペースで Kubernetes API エンドポイントを使用する場合、 Code Stream は、継続的インテグレーション (CI) タスクまたはカスタム タスクを実行するために必要な ConfigMap、シークレット、ポッドなどの Kubernetes リソースを作成します。 Code Stream は、NodePort を使用してコンテナと通信します。

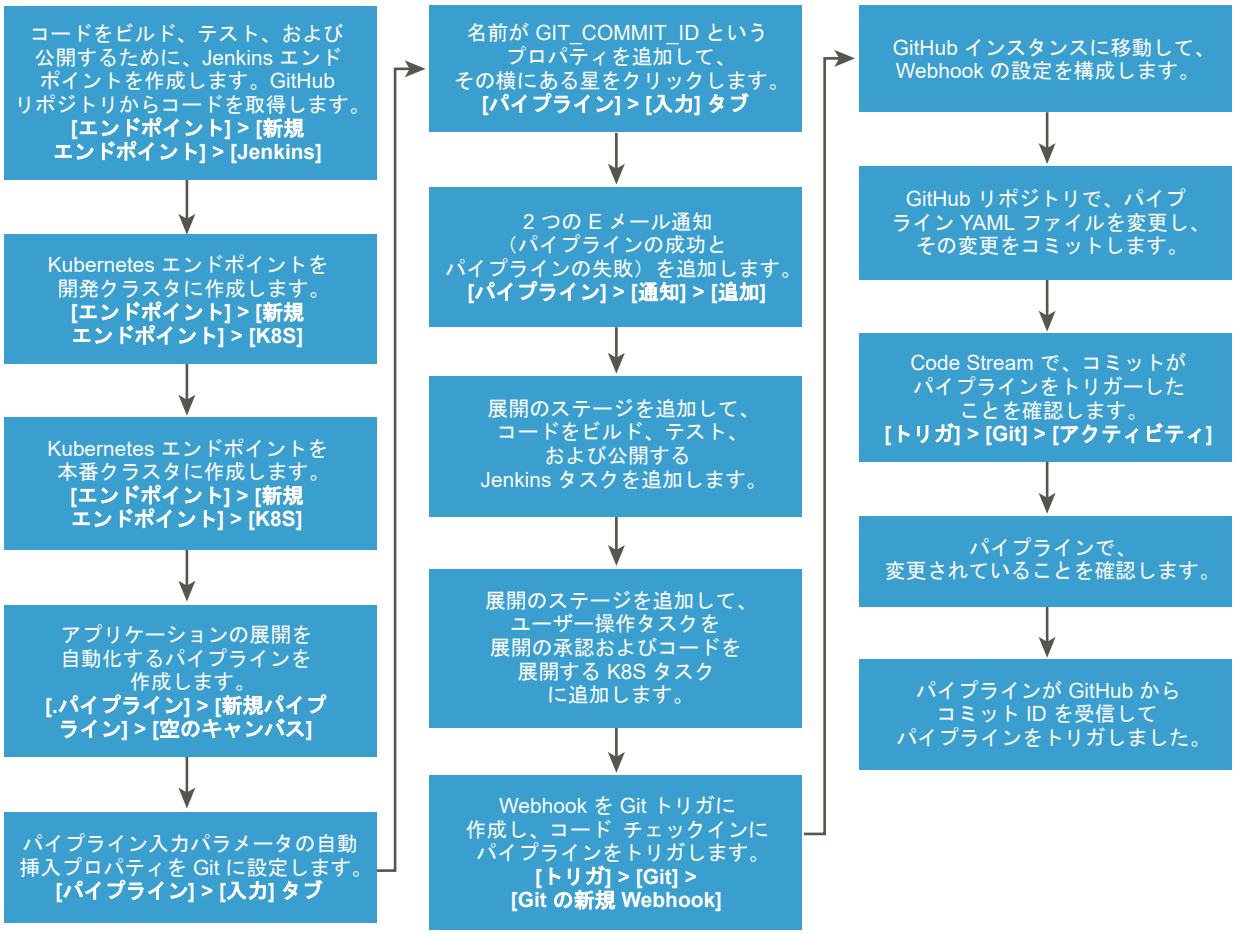
パイプラインの実行の間でデータを共有するには、パーシステント ボリュームの要求を指定する必要があります。Code Stream はパーシステント ボリュームの要求をコンテナにマウントしてデータを保存し、以降のパイプラインの実行に使用します。

Code Stream パイプライン ワークスペースでは、継続的インテグレーション タスクとカスタム タスクで Docker と Kubernetes がサポートされます。

ワークスペースの構成に関する詳細については、パイプライン ワークスペースの構成を参照してください。



パイプラインがアプリケーションをビルド、テスト、公開、および展開できるように、開発ツール、展開インスタンス、およびパイプライン YAML ファイルを使用できる必要があります。パイプラインは、AWS 上の Kubernetes クラスタの開発インスタンスと本番インスタンスにアプリケーションを展開します。



アプリケーションのリリースを自動化するその他の方法は次のとおりです。

- Jenkins を使用してアプリケーションをビルドする代わりに、 Code Stream のネイティブ ビルド機能と Docker ビルド ホストを使用できます。
- アプリケーションを Kubernetes クラスタに展開する代わりに、Amazon Web Services (AWS) クラスタに展開することができます。

Code Stream のネイティブ ビルド機能と Docker ホストの使用方法の詳細については、次を参照してください。

- スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画
- タスクの手動追加を行う前の Code Stream での CICD ネイティブ ビルドの計画

前提条件

- 展開するアプリケーション コードが、運用中の GitHub リポジトリにあることを確認します。
- Jenkins の作業インスタンスがあることを確認します。
- 動作しているメール サーバがあることを確認します。
- Code Stream で、メール サーバに接続するメール エンドポイントを作成します。
- 開発および本番環境用に、Amazon Web Services (AWS) 上に 2 つの Kubernetes クラスタを設定します。これにより、パイプラインでアプリケーションが展開されます。

- GitHub リポジトリにパイプラインの YAML コードが含まれていること、および環境用のメタデータと仕様を定義する YAML ファイルが含まれていることを確認します。

手順

1 Code Stream で、[エンドポイント] - [新規エンドポイント] の順にクリックし、パイプラインで使用する Jenkins エンドポイントを作成して、GitHub リポジトリからコードを取得します。

2 Kubernetes エンドポイントを作成するには、[新規エンドポイント] をクリックします。

   a 開発 Kubernetes クラスタのエンドポイントを作成します。

   b 本番 Kubernetes クラスタのエンドポイントを作成します。

   Kubernetes クラスタの URL では、ポート番号が含まれる場合も含まれない場合もあります。

   例：

   `https://10.111.222.333:6443`

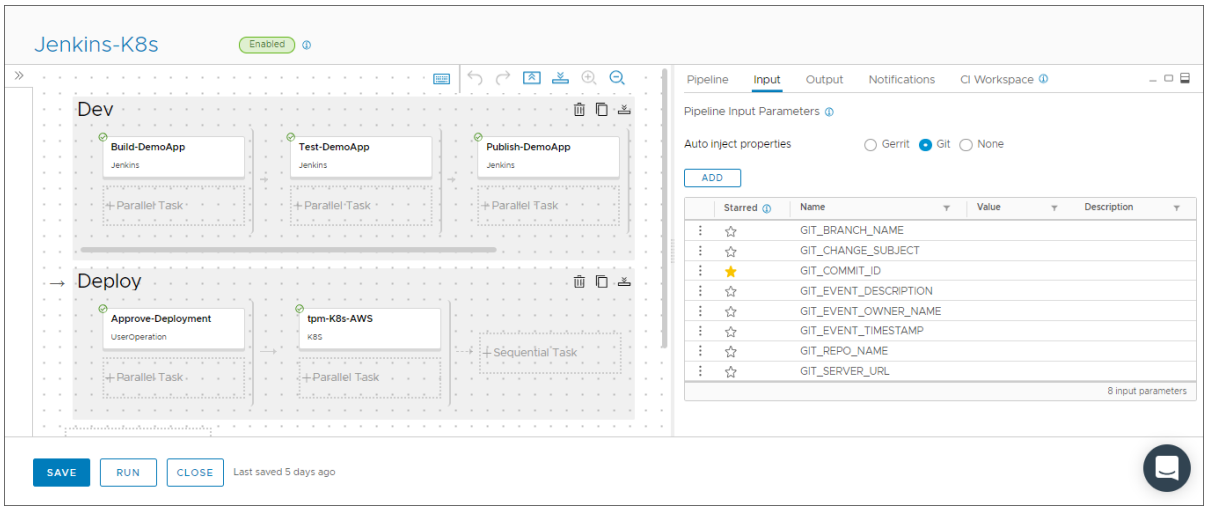
   `https://api.kubernetesserver.fa2c1d78-9f00-4e30-8268-4ab81862080d.k8s-user.com`

3 Wordpress などの、アプリケーションのコンテナを開発 Kubernetes クラスタに展開するパイプラインを作成し、そのパイプラインの入力プロパティを設定します。

   a パイプラインがそれをトリガする GitHub 内のコード コミットを認識できるようにするには、パイプラインで [入力] タブをクリックし、[自動挿入プロパティ] を選択します。

   b [GIT_COMMIT_ID] という名前のプロパティを追加し、隣の星印をクリックします。

   パイプラインが実行されると、パイプラインの実行で、Git トリガが返すコミット ID が表示されます。

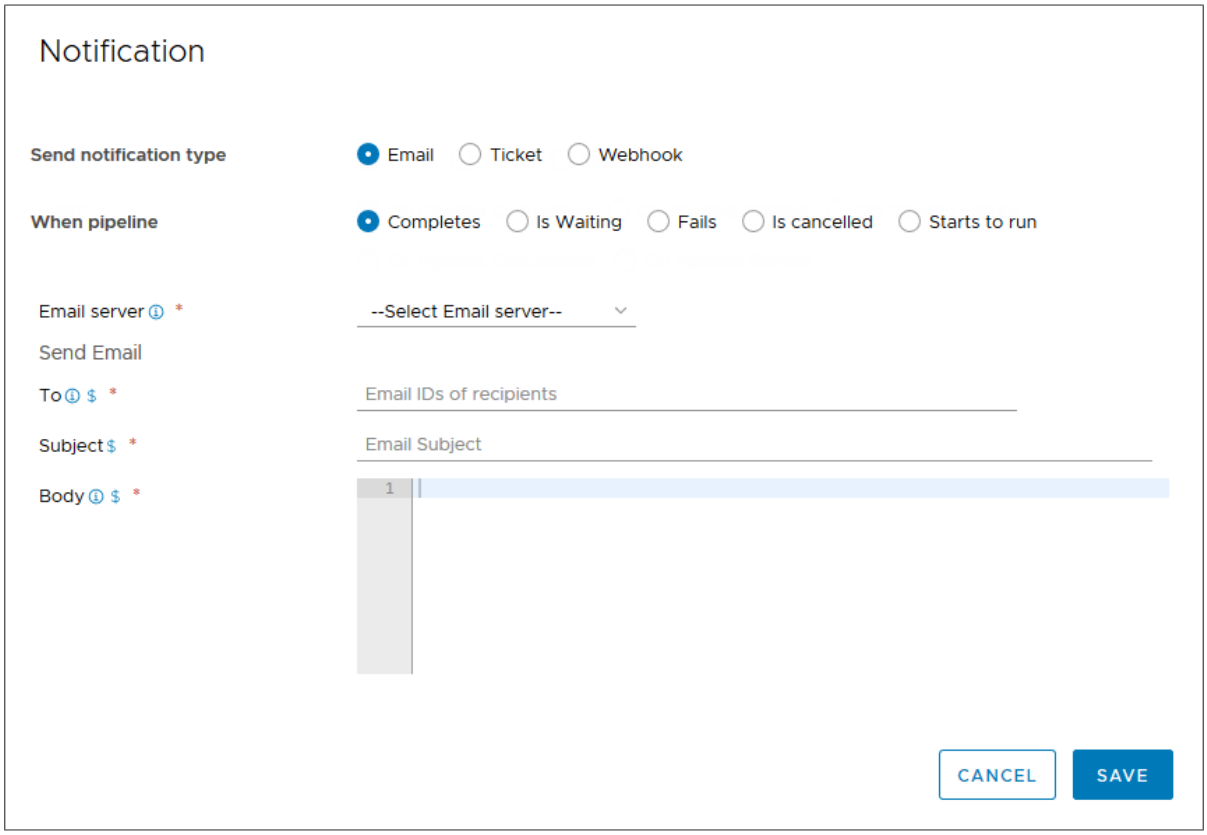


4 パイプラインが成功または失敗したときに E メールを送信するための通知を追加します。

a パイプラインで、[通知] タブをクリックし、[追加] をクリックします。

b パイプラインの実行が完了したときに E メール通知を追加するには、[E メール] を選択し、[の完了時] を選択します。次に、メール サーバを選択し、メール アドレスを入力して、[保存] をクリックします。

c パイプラインの失敗に関して別の E メール通知を追加するには、[の失敗時] を選択して、[保存] をクリックします。

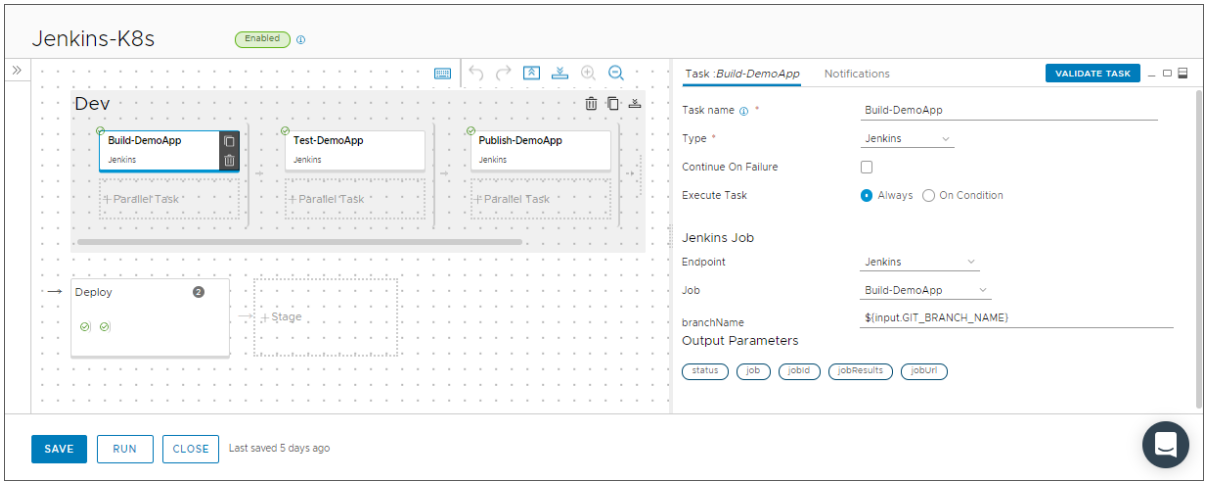


5 開発ステージをパイプラインに追加し、アプリケーションをビルド、テスト、および公開するタスクを追加します。次に、各タスクを検証します。

a アプリケーションをビルドするには、Jenkins エンドポイントを使用する Jenkins タスクを追加し、Jenkins サーバからビルド ジョブを実行します。次に、パイプラインがコードを取得するために、Git ブランチを次の形式で入力します。**`$ {input.GIT_BRANCH_NAME}`**

b アプリケーションをテストするには、同じ Jenkins エンドポイントを使用する Jenkins タスクを追加し、Jenkins サーバからテスト ジョブを実行します。次に、同じ Git ブランチを入力します。

c アプリケーションを公開するには、同じ Jenkins エンドポイントを使用する Jenkins タスクを追加し、Jenkins サーバから公開ジョブを実行します。次に、同じ Git ブランチを入力します。

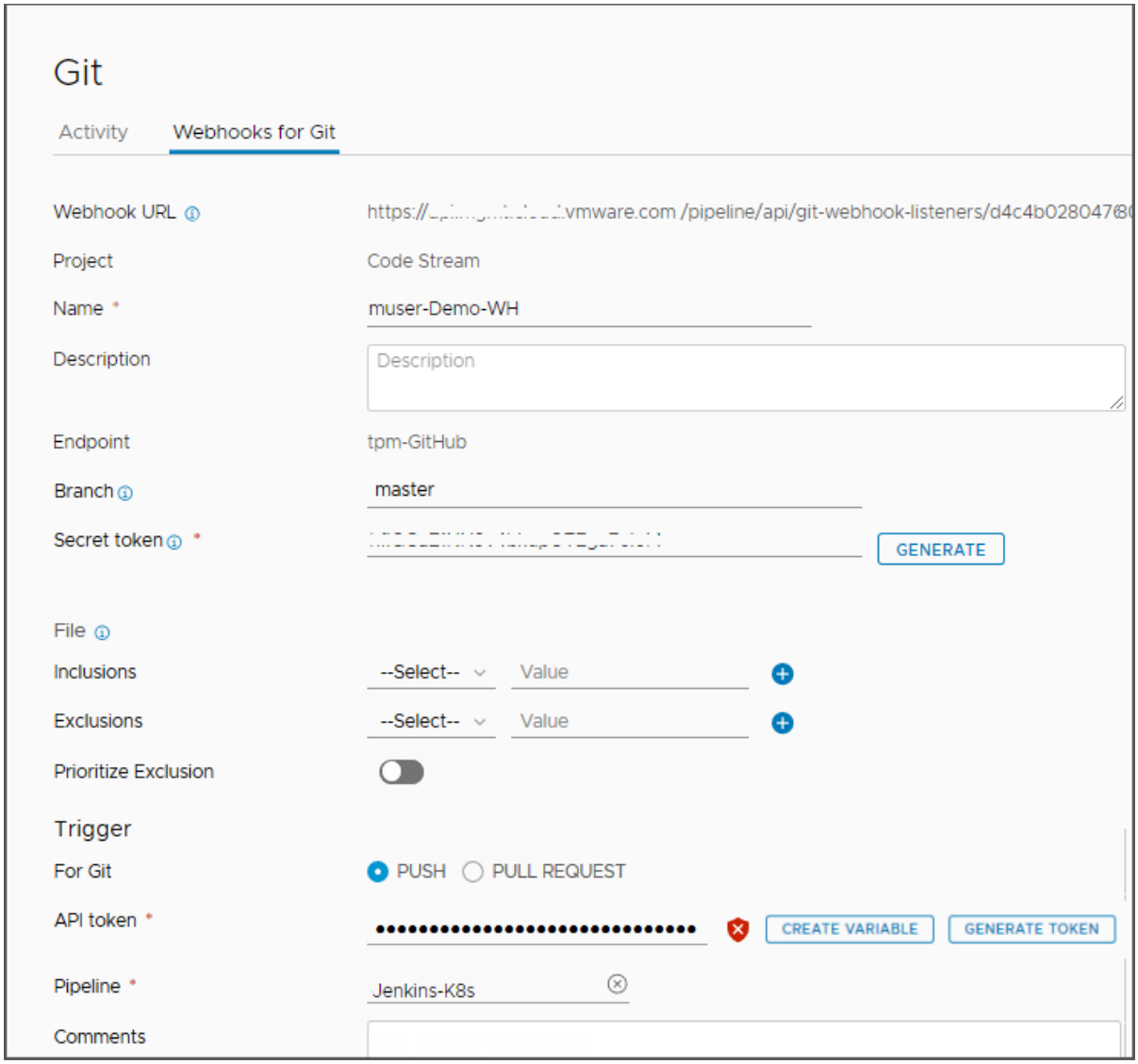


6 パイプラインに展開ステージを追加してから、アプリケーションの展開の承認を必要とするタスクと、アプリケーションを Kubernetes クラスタに展開する別のタスクを追加します。次に、各タスクを検証します。

a アプリケーションの展開で承認を要求するには、ユーザー操作タスクを追加し、承認する必要のあるユーザーのメール アドレスを追加して、メッセージを入力します。次に、[E メールの送信] を有効にします。

b アプリケーションを展開するには、Kubernetes タスクを追加します。次に、Kubernetes タスクのプロパティで、開発 Kubernetes クラスタを選択し、[作成] アクションを選択して、[ローカル定義] ペイロード ソースを選択します。次に、ローカル YAML ファイルを選択します。

7 Code Stream が Git トリガを使用できるようにする Git Webhook を追加します。これにより、開発者がコードをコミットするときにパイプラインがトリガされます。



8 パイプラインをテストするには、GitHub リポジトリに移動し、アプリケーション YAML ファイルを更新して、変更をコミットします。

a Code Stream で、コミットが表示されていることを確認します。

a [トリガ] - [Git] - [アクティビティ] の順にクリックします。

b パイプラインのトリガを探します。

c [ダッシュボード] - [パイプライン ダッシュボード] の順にクリックします。

d パイプライン ダッシュボードで、正常に変更が行われた最新の領域で GIT_COMMIT_ID を見つけます。

9 パイプライン コードを調べて、変更が表示されることを確認します。

結果

完了です。Kubernetes クラスタへのソフトウェア アプリケーションの展開が自動化されました。

例： Kubernetes クラスタにアプリケーションを展開するパイプライン YAML の例

この例で使用されるパイプラインのタイプについては、YAML は次のコードに似ています。

```
apiVersion: v1
kind: Namespace
metadata:
  name: ${input.GIT_BRANCH_NAME}
  namespace: ${input.GIT_BRANCH_NAME}
---
apiVersion: v1
data:
  .dockercfg:
eyJzeW1waG9ueS10YW5nby1iZXRhMi5qZnJvZy5pbyI6eyJ1c2VybmFtZSI6InRhbmdvLWJldGEyIiwicGFzc3dvcmQiOi
JhRGstcmVOLW1UQi1IejciLCJlbWFpbCI6InRhbmdvLWJldGEyQHZtd2FyZS5jb20iLCJhdXRoIjoiZEdGdVoyOHRZbVYw
WVRJNllVUnJMWEpsVGkxdFZFSXRTSG8zIn19
kind: Secret
metadata:
  name: jfrog
  namespace: ${input.GIT_BRANCH_NAME}
type: kubernetes.io/dockercfg
---
apiVersion: v1
kind: Service
metadata:
  name: codestream
  namespace: ${input.GIT_BRANCH_NAME}
  labels:
    app: codestream
spec:
  ports:
    - port: 80
  selector:
    app: codestream
    tier: frontend
  type: LoadBalancer
---
apiVersion: extensions/v1
kind: Deployment
metadata:
  name: codestream
  namespace: ${input.GIT_BRANCH_NAME}
  labels:
    app: codestream
spec:
  selector:
    matchLabels:
      app: codestream
      tier: frontend
  strategy:
```

```
    type: Recreate
  template:
    metadata:
      labels:
        app: codestream
        tier: frontend
    spec:
      containers:
      - name: codestream
        image: cas.jfrog.io/codestream:${input.GIT_BRANCH_NAME}-${Dev.PublishApp.output.jobId}
        ports:
        - containerPort: 80
          name: codestream
      imagePullSecrets:
      - name: jfrog
```

次のステップ

ソフトウェア アプリケーションを本番の Kubernetes クラスタに展開するには、手順を再度実行して、本番クラスタを選択します。

Code Stream と Jenkins の統合の詳細については、 Code Stream を Jenkins と統合する方法を参照してください。

Code Stream のアプリケーションをブルーグリーン展開に展開する方法

ブルーグリーンは、2 台の Docker ホストを使用して両者を同じように Kubernetes クラスタに展開および構成する展開モデルです。ブルーグリーン展開モデルで Code Stream のパイプラインがアプリケーションを展開するようにすると、ご使用の環境のダウンタイムを短縮できます。

展開モデルのブルー インスタンスとグリーン インスタンスは、それぞれ異なる目的を果たします。一度に 1 つのインスタンスのみが、アプリケーションを展開するライブ トラフィックを受け入れます。各インスタンスは、一定の時刻にそのトラフィックを受け入れます。ブルー インスタンスはアプリケーションの最初のバージョンを受け取り、グリーン インスタンスは 2 番目のバージョンを受け取ります。

ブルーグリーン環境のロード バランサによって、アプリケーションを展開するときにライブ トラフィックがどのルートをたどるかが決まります。ブルーグリーン モデルを使用すると、環境は動作し続け、ユーザーはダウンタイムがあっても気づかず、パイプラインはアプリケーションを本番環境に継続的に統合して展開します。

Code Stream でパイプラインを作成すると、ブルーグリーン展開モデルが 2 つのステージで表されます。開発用のステージと本番用のステージです。

Code Stream パイプライン ワークスペースでは、継続的インテグレーション タスクとカスタム タスクで Docker と Kubernetes がサポートされます。

ワークスペースの構成については、パイプライン ワークスペースの構成を参照してください。

表 5-2. ブルーグリーン展開の開発ステージ タスク

タスク タイプ	タスク
Kubernetes	ブルーグリーン展開用の名前空間を作成します。
Kubernetes	Docker Hub のプライベート キーを作成します。
Kubernetes	アプリケーションの展開に使用するサービスを作成します。
Kubernetes	ブルー展開を作成します。
ポーリング	ブルー展開を確認します。
Kubernetes	名前空間を削除します。

表 5-3. ブルーグリーン展開用の本番ステージ タスク

タスク タイプ	タスク
Kubernetes	グリーンは、ブルーからサービスの詳細を取得します。
Kubernetes	グリーン レプリカ セットの詳細を取得します。
Kubernetes	グリーン展開を作成し、プライベート キーを使用してコンテナ イメージをプルします。
Kubernetes	サービスを更新します。
ポーリング	本番 URL で展開が成功したことを確認します。
Kubernetes	ブルー展開を終了します。
Kubernetes	ブルー展開を削除します。

独自のブルーグリーン展開モデルにアプリケーションを展開するには、Code Stream でパイプラインを作成して 2 つのステージを含めます。最初のステージにはブルー インスタンスにアプリケーションを展開するブルー タスクを含め、2 番目のステージにはグリーン インスタンスにアプリケーションを展開するグリーン タスクを含めます。

パイプラインの作成には、CICD スマート パイプライン テンプレートを使用できます。自動的にパイプラインのステージとタスクを作成するもので、展開を選択できます。

パイプラインを手動で作成する場合は、パイプライン ステージを計画する必要があります。例については、タスクの手動追加を行う前の Code Stream での CICD ネイティブ ビルドの計画を参照してください。

この例では、CICD スマート パイプライン テンプレートを使用してブルーグリーン パイプラインを作成します。

前提条件

- AWS 上の動作中の Kubernetes クラスタにアクセスできることを確認します。
- ブルーグリーン展開環境を設定し、ブルー インスタンスとグリーン インスタンスが同一になるように設定していることを確認します。
- AWS 上の Kubernetes クラスタにアプリケーション イメージを展開する Code Stream の Kubernetes エンドポイントを作成します。

- CICD スマート パイプライン テンプレートの使用について理解する必要があります。スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画を参照してください。

手順

1 [パイプライン] - [新しいパイプライン] - [スマート テンプレート] - [CI/CD テンプレート] の順にクリックします。

2 CICD スマート パイプライン テンプレートの CI 部分の情報を入力し、[次へ] をクリックします。

詳細については、スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画を参照してください。

3 スマート パイプライン テンプレートの CD 部分を入力します。

a アプリケーション展開の環境を選択します。たとえば、[開発] と [本番] です。

b パイプラインによる展開で使用するサービスを選択します。

c [展開] 領域で、開発環境と本番環境のクラスタ エンドポイントを選択します。

d 本番展開モデルの場合、[ブルーグリーン] を選択し、[作成] をクリックします。

結果

完了です。スマート パイプライン テンプレートを使用して、AWS 上の Kubernetes 本番クラスタ内のブルーグリーン インスタンスにアプリケーションを展開するパイプラインを作成しました。

例： ブルーグリーン展開タスクのサンプルの YAML コード

ブルーグリーン展開の Kubernetes パイプライン タスクに表示される YAML コードは、名前空間、サービス、展開を作成する次の例のようになります。プライベート所有のリポジトリからイメージをダウンロードする必要がある場合は、Docker 構成の [シークレット] を含むセクションを YAML ファイルに含める必要があります。スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画の CD 部分を参照してください。

スマート パイプライン テンプレートによってパイプラインが作成されたら、それぞれの展開での必要に応じてタスクを変更できます。

サンプルの名前空間を作成する YAML コード：

```
apiVersion: v1
kind: Namespace
metadata:
  name: codestream-82855
  namespace: codestream-82855
```

サンプルのサービスを作成する YAML コード：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: codestream-demo
  name: codestream-demo
  namespace: bluegreen-799584
spec:
  minReadySeconds: 0
  ports:
  - port: 80
  selector:
    app: codestream-demo
    tier: frontend
  type: LoadBalancer
```

サンプルの展開を作成する YAML コード：

```
apiVersion: extensions/v1
kind: Deployment
metadata:
  labels:
    app: codestream-demo
  name: codestream-demo
  namespace: bluegreen-799584
spec:
  minReadySeconds: 0
  replicas: 1
```

```
  selector:
    matchLabels:
      app: codestream-demo
      tier: frontend
  template:
    metadata:
      labels:
        app: codestream-demo
        tier: frontend
    spec:
      containers:
      - image: ${input.image}:${input.tag}
        name: codestream-demo
        ports:
        - containerPort: 80
          name: codestream-demo
      imagePullSecrets:
      - name: jfrog-2
      minReadySeconds: 0
```

次のステップ

Code Stream の使用方法の詳細については、5 章 Code Stream を使用するためのチュートリアルを参照してください。

デプロイをロールバックする場合は、 Code Stream で展開をロールバックする方法を参照してください。

この他の参考情報については、 Code Stream 管理者および開発者向けのその他のリソースを参照してください。

ビルド、テスト、展開用の独自のツールを Code Stream に統合する方法

DevOps 管理者または開発者は、カスタム スクリプトを作成して、 Code Stream の機能を拡張できます。

カスタム スクリプトを使用すると、Code Stream を独自の継続的インテグレーション (CI) および継続的デリバリ (CD) ツールと API に統合して、アプリケーションをビルド、テスト、および展開できます。カスタム スクリプトは、アプリケーション API を公開しない場合に特に便利です。

カスタム スクリプトでは、ビルド、テスト、展開の各ツールを Code Stream と統合するために必要なことをほぼすべて実行できます。たとえば、スクリプトでパイプラインのワークスペースを利用することで、アプリケーションをビルドしてテストする継続的インテグレーション タスクと、アプリケーションを展開する継続的デリバリ タスクをサポートできます。パイプラインの終了時にスラックにメッセージを送信するなど、さまざまなことができます。

Code Stream パイプライン ワークスペースでは、継続的インテグレーション タスクとカスタム タスクで Docker と Kubernetes がサポートされます。

ワークスペースの構成に関する詳細については、パイプライン ワークスペースの構成を参照してください。

カスタム スクリプトを記述するには、サポートされている言語のいずれかを使用します。スクリプトにビジネス ロジックを含めて、入力と出力を定義します。出力タイプには、数値、文字列、テキスト、およびパスワードを含めることができます。異なるビジネス ロジック、入力、および出力を使用して、複数のバージョンのカスタム スクリプトを作成できます。

カスタム タスクで、パイプラインによってスクリプトのバージョンを実行します。作成したスクリプトは、 Code Stream インスタンスに保存されます。

パイプラインでカスタム統合が使用されている場合、カスタム統合を削除しようとすると、エラー メッセージが表示され、削除できないことが示されます。

カスタム統合を削除すると、カスタム スクリプトのすべてのバージョンが削除されます。既存のパイプラインに、そのスクリプトのいずれかのバージョンを使用するカスタム タスクがある場合、パイプラインは失敗します。既存のパイプラインが失敗しないようにするために、不要になったスクリプトのバージョンを廃止して、取り消すことができます。そのバージョンを使用しているパイプラインがない場合は、削除できます。

表 5-4. カスタム スクリプトを記述した後の操作

操作	操作の詳細情報
パイプラインにカスタム タスクを追加します。	カスタム タスク： ■ パイプラインの他の CI タスクと同じコンテナで実行されます。 ■ パイプラインがカスタム タスクを実行する前に、スクリプトでポピュレートする入出力変数を含めます。 ■ 複数のデータ タイプとさまざまなタイプのメタ データをスクリプトに入力および出力として定義できます。
カスタム タスクでスクリプトを選択します。	スクリプトで入力および出力プロパティを宣言します。
パイプラインを保存し、有効にして実行します。	パイプラインを実行すると、カスタム タスクによって指定されたスクリプトのバージョンが呼び出され、そこでビジネス ロジックが実行されます。これにより、ビルド、テスト、展開の各ツールが Code Stream と統合されます。
パイプラインの実行後、実行結果を確認します。	想定どおりの結果であるかを確認します。

カスタム統合バージョンを呼び出すカスタム タスクを使用する場合は、パイプラインの [ワークスペース] タブで、カスタム環境変数を名前と値のペアとして指定できます。CI タスクを実行してイメージを展開するワークスペース コンテナがビルダー イメージによって作成されると、そのコンテナに Code Stream が環境変数を渡します。

たとえば、 Code Stream インスタンスが Web プロキシを必要とする場合に、Docker ホストを使用してカスタム統合のためのコンテナを作成すると、 Code Stream がパイプラインを実行し、Web プロキシ設定変数をそのコンテナに渡します。

表 5-5. 環境変数の名前と値のペアの例

名前	値
HTTPS_PROXY	http://10.0.0.255:1234
https_proxy	http://10.0.0.255:1234
NO_PROXY	10.0.0.32, *.dept.vsphere.local
no_proxy	10.0.0.32, *.dept.vsphere.local
HTTP_PROXY	http://10.0.0.254:1234
http_proxy	http://10.0.0.254:1234
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

このように、名前と値のペアがユーザー インターフェイスに表示されます。

この例では、 Code Stream をスラック インスタンスに接続してメッセージをスラック チャネルにポストするカスタム統合を作成します。

前提条件

- カスタム スクリプトを記述する場合は、Python 2、Python 3、Node.js のいずれかの言語か、Bash、sh、zsh のいずれかのシェル言語を使用できることを確認してください。
- インストールされた Node.js または Python のランタイムを使用して、コンテナ イメージを生成します。

手順

1 カスタム統合を作成します。

a [カスタム統合] - [新規] の順にクリックし、適切な名前を入力します。

b 優先するランタイム環境を選択します。

c [作成] をクリックします。

スクリプトが開き、コードが表示されます。コードには、必要なランタイム環境が含まれています。たとえば、`runtime: "nodejs"` です。スクリプトには、ビルダ イメージが使用するランタイムを含める必要があります。これにより、パイプラインの実行時に、パイプラインに追加したカスタム タスクが正常に実行されます。これを行わない場合、カスタム タスクは失敗します。

カスタム統合 YAML の主な領域は、ランタイム、コード、入力プロパティ、および出力プロパティなどです。この手順では、さまざまなタイプと構文について説明します。

カスタム統合 YAML のキー	説明
runtime	Code Stream がコードを実行するタスク ランタイム環境。次のいずれかの文字列を使用することができ、大文字と小文字は区別しません。 ■ nodejs ■ python2 ■ python3 ■ shell 何も指定しなかった場合は、shell がデフォルトとして使用されます。
code	カスタム タスクの一部として実行するカスタム ビジネス ロジック。
inputProperties	カスタム タスク構成の一部としてキャプチャする入力プロパティの配列。これらのプロパティは、通常はコード内で使用されます。
outputProperties	カスタム タスクからエクスポートしてパイプラインに伝達できる出力プロパティの配列。

2 スクリプトで入力プロパティを宣言するときには、有効なデータ タイプおよびメタ データを使用します。

入力プロパティは、YAML の `code:` セクションでスクリプトにコンテキストとして渡されます。

カスタム タスク YAML の入力キー	説明	必須
type	レンダリングする入力のタイプ： ■ **text** ■ **textarea** ■ **number** ■ **checkbox** ■ **password** ■ **select**	はい
name	カスタム タスクに対する入力の名前または文字列。カスタム統合 YAML コードに取り込まれます。カスタム統合用に定義された入力プロパティごとに一意である必要があります。	はい
title	パイプライン モデル キャンバス上のカスタム タスクに対する入力プロパティのテキスト文字列ラベル。空白のままにすると、デフォルトで **name** が使用されます。	いいえ
required	ユーザーがカスタム タスクを設定するときに入力プロパティを入力する必要があるかどうかを決定します。true または false に設定します。true の場合に、パイプライン キャンバスでカスタム タスクを設定するときにユーザーが値を指定しなければ、タスクの状態は未構成のままになります。	いいえ
placeHolder	値がない場合の入力プロパティ エントリ領域のデフォルト テキスト。html プレースホルダ属性にマッピングされます。特定の入力プロパティ タイプでのみサポートされます。	いいえ
defaultValue	パイプライン モデル画面でカスタム タスクがレンダリングされるときに、入力プロパティ エントリ領域に入力されるデフォルト値。	いいえ
bindable	パイプライン キャンバスでカスタム タスクをモデリングするときに、入力プロパティがドル記号の変数を受け入れるかどうかを決定します。タイトルの横に [$] インジケータを追加します。特定の入力プロパティ タイプでのみサポートされます。	いいえ
labelMessage	ユーザーのヘルプ ツールチップとして機能する文字列。入力タイトルの横にツールチップ アイコン [i] を追加します。	いいえ
enum	特定の入力プロパティ オプションを表示する値の配列を取得します。特定の入力プロパティ タイプでのみサポートされます。 ユーザーがオプションを選択して、カスタム タスク用に保存した場合は、[inputProperty] 値がこの値に対応して、カスタム タスクのモデリング中に表示されます。 たとえば、値として 2015 を使用します。 ■ 2015 ■ 2016 ■ 2017 ■ 2018 ■ 2019 ■ 2020	いいえ

カスタム タスク YAML の入力キー	説明	必須
options	[optionKey] および [optionValue] を使用して、オブジェクトの配列を取得します。 ■ [optionKey]：タスクのコード セクションに伝達される値。 ■ [optionValue]：ユーザー インターフェイスにオプションを表示する文字列。 特定の入力プロパティ タイプでのみサポートされます。 オプションは、次のとおりです。 [optionKey]：key1：カスタム タスク用に選択して保存した場合、この inputProperty の値はコード セクションの [key1] に対応します。 [optionValue]：「1 のラベル」。ユーザー インターフェイスの [key1] の値を表示します。カスタム タスクのほかの場所には表示されません。 [optionKey]：key2 [optionValue]：「2 のラベル」。 [optionKey]：key3 [optionValue]：「3 のラベル」。	いいえ
minimum	この入力プロパティの有効な最小値として機能する値を取得します。数値タイプの入力プロパティでのみサポートされます。	いいえ
maximum	この入力プロパティの有効な最大値として機能する値を取得します。数値タイプの入力プロパティでのみサポートされます。	いいえ

表 5-6. カスタム スクリプトでサポートされるデータ タイプとメタ データ

サポートされるデータ タイプ	入力でサポートされるメタ データ
■ 文字列 ■ テキスト ■ リスト：任意のタイプのリスト ■ マップ： map[string]any ■ セキュア：カスタム タスクを保存するときに暗号化されるパスワード テキスト ボックスとして表示されます。 ■ 数値 ■ ブール値：テキスト ボックスとして表示されます。 ■ URL：文字列と同じですが、検証が追加されています。 ■ 選択肢、ラジオ ボタン	■ type：String | Text ... のいずれか ■ default：デフォルト値 ■ options：オプションのリストまたはマップ。選択肢またはラジオ ボタンで使用されます。 ■ min：最小値または最小サイズ ■ max：最大値または最大サイズ ■ title：テキスト ボックスの詳細な名前 ■ placeHolder：ユーザー インターフェイス プレースホルダ ■ description：ツール チップになります。

例：

```
inputProperties:
        - name: message
          type: text
          title: Message
          placeHolder: Message for Slack Channel
          defaultValue: Hello Slack
          bindable: true
          labelInfo: true
          labelMessage: This message is posted to the Slack channel link provided in the
code
```

3 スクリプトで出力プロパティを宣言します。

出力プロパティはスクリプトのビジネス ロジック `code:` セクションからキャプチャされるため、ここに出力のコンテキストを宣言します。

パイプラインの実行時に、タスク出力の応答コードを入力できます。たとえば、**200** です。

Code Stream が [outputProperty] ごとにサポートされるキー。

key	説明
type	現在は、単一値 **label** が含まれています。
name	カスタム統合 YAML のコード ブロックによって送信されるキー。
title	[outputProperty] を表示するユーザー インターフェイスのラベル。

例：

```
outputProperties:
  - name: statusCode
    type: label
    title: Status Code
```

4 カスタム スクリプトの入力および出力を操作するには、**context** を使用して、入力プロパティを取得するか、出力プロパティを設定します。

入力プロパティの場合：`(context.getInput("key"))`

出力プロパティの場合：`(context.setOutput("key", "value"))`

Node.js の場合：

```
var context = require("./context.js")
var message = context.getInput("message");
//Your Business logic
context.setOutput("statusCode", 200);
```

Python の場合：

```
from context import getInput, setOutput
message = getInput('message')
//Your Business logic
setOutput('statusCode', '200')
```

Shell の場合：

```
# Input, Output properties are environment variables
echo ${message} # Prints the input message
//Your Business logic
export statusCode=200 # Sets output property statusCode
```

5 code: セクションで、カスタム統合のすべてのビジネス ロジックを宣言します。

たとえば、Node.js ランタイム環境では次のようになります。

```
code: |
    var https = require('https');
    var context = require("./context.js")

    //Get the entered message from task config page and assign it to message var
    var message = context.getInput("message");
    var slackPayload = JSON.stringify(
        {
            text: message
        });

    const options = {
        hostname: 'hooks.slack.com',
        port: 443,
        path: '/YOUR_SLACK_WEBHOOK_PATH',
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Content-Length': Buffer.byteLength(slackPayload)
        }
    };

    // Makes a https request and sets the output with statusCode which
    // will be displayed in task result page after execution
    const req = https.request(options, (res) => {
        context.setOutput("statusCode", res.statusCode);
    });

    req.on('error', (e) => {
        console.error(e);
    });
    req.write(slackPayload);
    req.end();
```

6 カスタム統合スクリプトをバージョン化してリリースする前に、Python または Node.js のコンテキスト ファイルをダウンロードし、スクリプトに含まれているビジネス ロジックをテストします。

a スクリプトにポインタを置き、キャンバスの最上部にあるコンテキスト ファイル ボタンをクリックします。たとえば、スクリプトが Python に含まれている場合は、[CONTEXT.PY] をクリックします。

b ファイルを変更して保存します。

c 開発システムで、コンテキスト ファイルを利用してカスタム スクリプトを実行し、テストします。

7 カスタム統合スクリプトにバージョンを適用します。

a [バージョン] をクリックします。

b バージョン情報を入力します。

c [リリース バージョン] をクリックして、カスタム タスクでスクリプトを選択できるようにします。

d バージョンを作成するには、[作成] をクリックします。

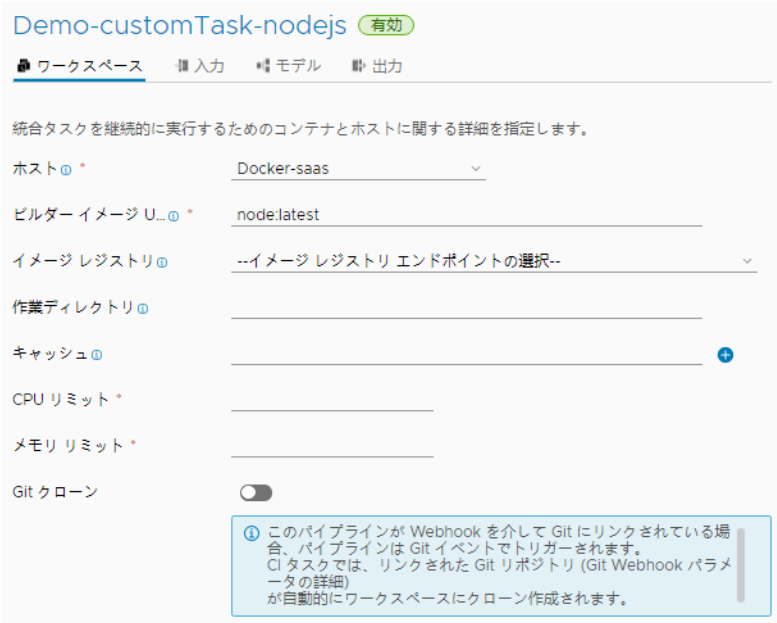
8 スクリプトを保存するには、[保存] をクリックします。

9 パイプラインで、ワークスペースを構成します。

この例では、Docker ワークスペースを使用します。

a [ワークスペース] タブをクリックします。

b Docker ホストおよびビルダ イメージ URL を選択します。



10 パイプラインにカスタム タスクを追加して設定します。

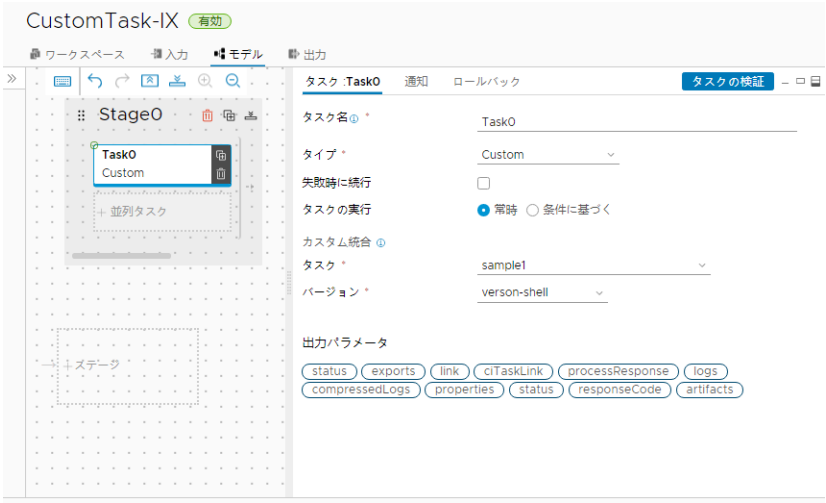
a [モデル] タブをクリックします。

b タスクを追加し、タイプとして [カスタム] を選択し、適切な名前を入力します。

c　カスタム統合スクリプトとバージョンを選択します。

d　スラックにカスタム メッセージを表示するには、メッセージ テキストを入力します。

入力したテキストでカスタム統合スクリプトの `defaultValue` がオーバーライドされます。例：



11　パイプラインを保存して有効にします。

a　[保存] をクリックします。

b　[パイプライン] タブで、[パイプラインを有効にする] をクリックして、円が右側になるようにします。

12　パイプラインを実行します。

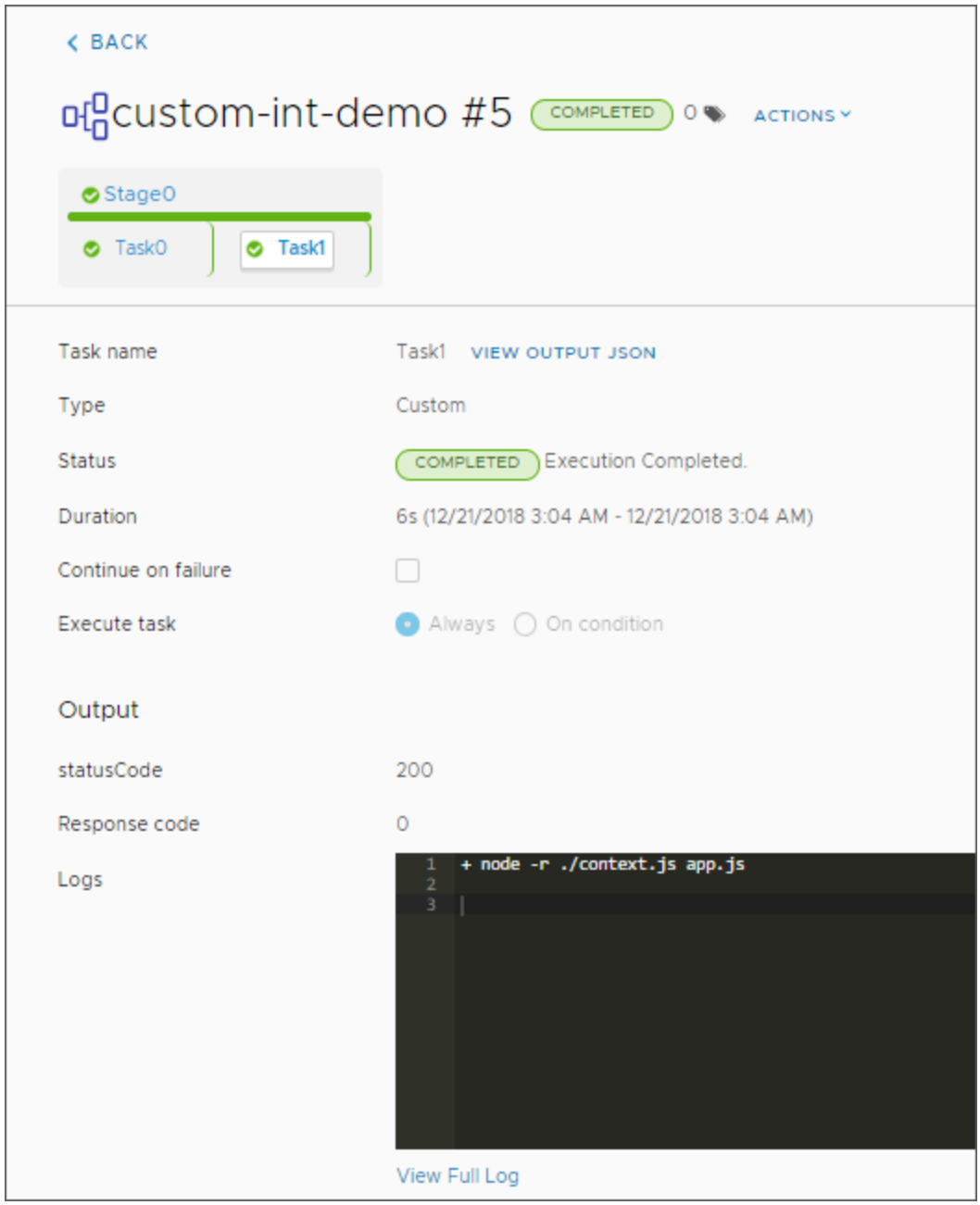
a　[実行] をクリックします。

b　パイプラインの実行状況を確認します。

c ステータス コード、応答コード、ステータス、および宣言出力が想定どおりであることを確認します。

出力プロパティとして [statusCode] を定義しました。たとえば、[statusCode] が 200 であれば、スラックのポストが正常に完了したことになり、[responseCode] が 0 であれば、スクリプトがエラーなしで正常に完了したことになります。

d 実行ログで出力を確認するには、[実行] をクリックし、パイプラインへのリンクをクリックしてから、タスクをクリックし、ログに記録されたデータを確認します。例：



13 エラーが発生した場合は、問題のトラブルシューティングを行い、パイプラインを再度実行します。

たとえば、基本イメージのファイルやモジュールが欠落している場合は、そのファイルが含まれている基本イメージを別途作成する必要があります。次に、Docker ファイルを指定し、パイプラインを介してイメージをプッシュします。

結果

完了です。 Code Stream をスラック インスタンスに接続してメッセージをスラック チャネルにポストするカスタム統合スクリプトを作成しました。

次のステップ

引き続きカスタム統合を作成します。パイプラインでカスタム タスクを使用するためのもので、 Code Stream の機能を拡張してソフトウェア リリース ライフサイクルを自動化できます。

次のタスクでクラウド テンプレート タスクのリソース プロパティを使用する方法

Code Stream でクラウド テンプレート タスクを使用するときに、一般的に問題となるのは、そのタスクの出力をパイプラインの後続のタスクで使用する方法です。クラウド マシンなど、クラウド テンプレート タスクの出力を使用するには、クラウド テンプレート タスクの展開の詳細内でリソース プロパティとクラウド マシンの IP アドレスを検索する方法を把握しておく必要があります。

たとえば、VMware クラウド テンプレートの展開の詳細には、クラウド マシン リソースとその IP アドレスが含まれています。パイプラインでは、クラウド マシンと IP アドレスを変数として使用して、クラウド テンプレート タスクを REST タスクにバインドすることができます。

展開の詳細を使用できるようにするには、VMware クラウド テンプレートの展開が完了している必要があるため、クラウド マシンの IP アドレスを検索する際に使用する方法は一般的ではありません。その後、VMware クラウド テンプレート展開のリソースを使用して、パイプライン タスクをバインドすることができます。

- パイプラインのクラウド テンプレート タスクに表示されるリソース プロパティは、Cloud Assembly の VMware クラウド テンプレートで定義されます。
- このクラウド テンプレートの展開が完了した時期が不明な場合があります。
- 展開が完了した後に Code Stream のクラウド テンプレート タスクで表示できるのは、VMware クラウド テンプレートの出力プロパティのみです。

この例は、アプリケーションを展開し、さまざまな API を呼び出す場合に特に便利です。たとえば、VMware クラウド テンプレートを呼び出すクラウド テンプレート タスクを使用し、呼び出した VMware クラウド テンプレートで REST API を使用して Wordpress アプリケーションを展開するとします。この場合、展開の詳細内で展開されたマシンの IP アドレスを特定し、API を使用してテストすることができます。

クラウド テンプレート タスクでは、オート フィルによって事前入力された詳細を表示することにより、変数のバインドを使用することができます。変数のバインド方法はユーザーが選択できます。

次の例は、この選択方法を示します。

- 実行されて成功したパイプライン内で、クラウド テンプレート タスクの展開の詳細とリソース プロパティを検索します。
- 展開の詳細のリソース セクションで、クラウド マシンの IP アドレスを見つけます。
- パイプラインのクラウド テンプレート タスクの後に REST タスクを追加します。
- REST タスクの URL 内でクラウド マシンの IP アドレスを使用して、クラウド テンプレート タスクを REST タスクにバインドします。
- パイプラインを実行して、クラウド テンプレート タスクから REST タスクまでのバインド作業を監視します。

前提条件

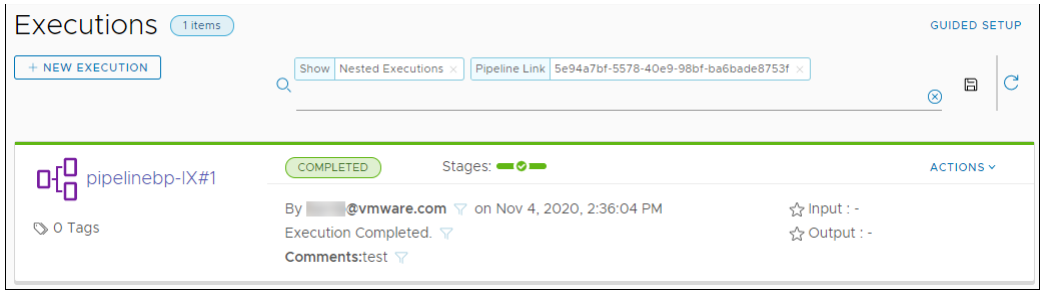
- バージョン管理された稼動中の VMware クラウド テンプレートがあることを確認します。
- Cloud Assembly で VMware クラウド テンプレートの展開に成功したことを確認します。
- VMware クラウド テンプレートを使用するクラウド テンプレート タスクが含まれているパイプラインがあることを確認します。
- パイプラインが実行されて、成功したことを確認します。

手順

1 パイプラインで、クラウド テンプレート タスクの展開の詳細のリソース セクションで、クラウド マシンの IP アドレスを特定します。

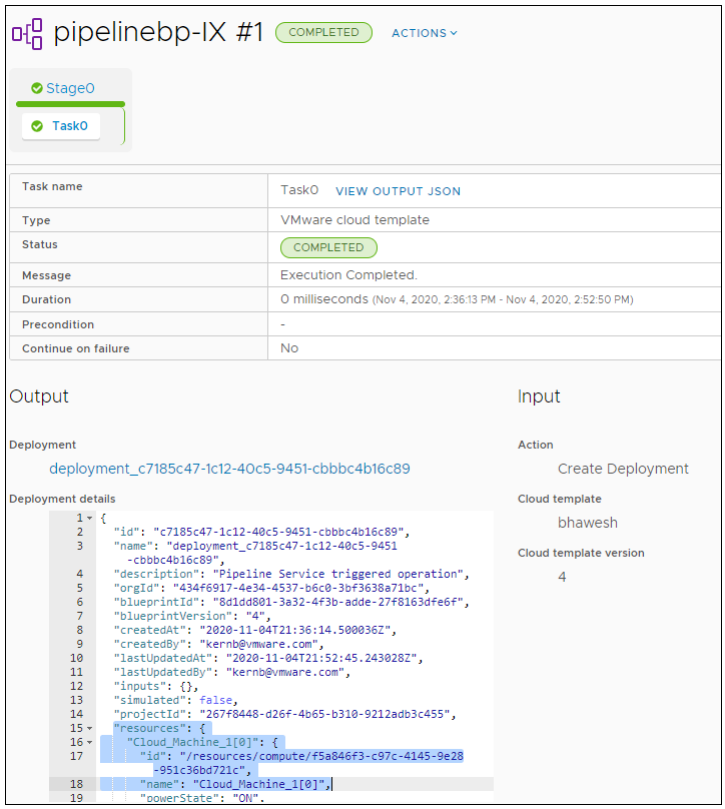
a [アクション] - [実行の表示] の順にクリックします。

b 成功したパイプラインの実行中に、パイプラインの実行へのリンクをクリックします。



c パイプライン名の下にある [タスク] へのリンクをクリックします。

d [出力] 領域で、展開の詳細を特定します。

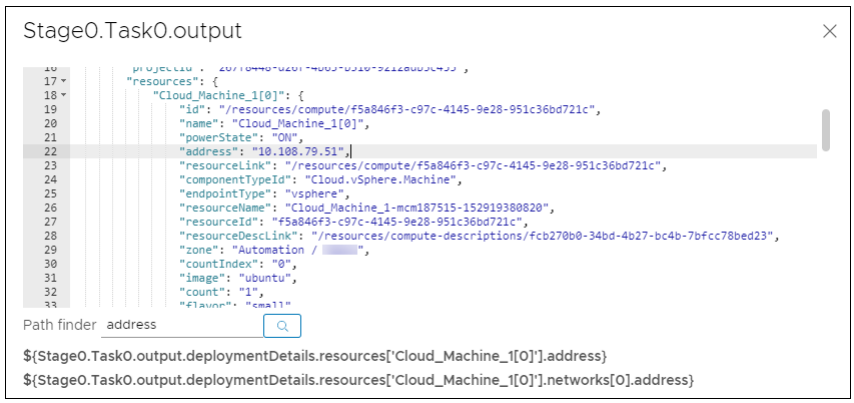


e　展開の詳細のリソース セクションで、クラウド マシンの名前を特定します。

REST タスクの URL に、クラウド マシン名の構文を含めます。

f　クラウド テンプレート タスクの出力プロパティのバインド式を検索するには、[JSON 出力の表示] をクリックし、アドレス プロパティを検索して、クラウド マシンの IP アドレスを特定します。

バインド式は、JSON 出力のプロパティと検索アイコンの下に表示されます。



アドレス リソース プロパティには、クラウド マシンの IP アドレスが表示されます。例：

```
"resources": {
        "Cloud_Machine_1[0]": {
            "name": "Cloud_Machine_1[0]",
            "powerState": "ON",
            "address": "10.108.79.51",
            "resourceName": "Cloud_Machine_1-mcm187515-152919380820"
```

2　パイプライン モデルに戻り、REST タスクに URL を入力します。

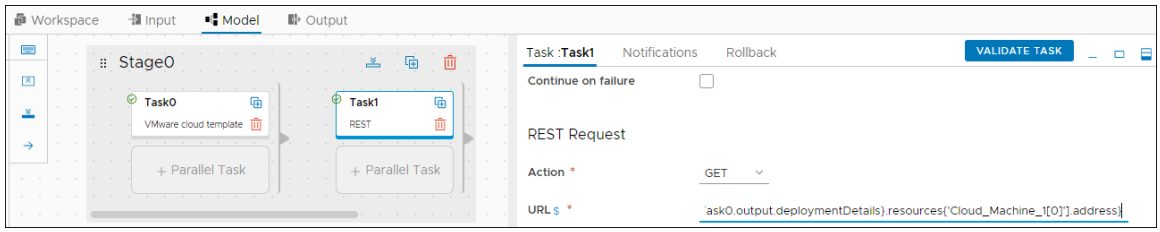
a　[アクション] - [パイプラインの表示] の順にクリックします。

b　REST タスクをクリックします。

c　REST 要求の URL 領域に、**$** と入力し、[Stage]、[Task]、[output]、[deploymentDetails] を選択し、**resources** と入力します。

オート フィルを使用して事前入力する機能は、**resources** を入力する時点まで使用できます。

d　展開の詳細で、**{'Cloud_Machine_1[0]'].address}** のようにクラウド マシン リソースの残りの部分を入力します。



クラウド マシンを入力する場合は、記載されている角括弧の表記を使用する必要があります。

完全な URL の形式は、`${Stage0.Task0.output.deploymentDetails.resources{'Cloud_Machine_1[0]'].address}` です。

3 パイプラインを実行し、REST タスクを監視して、クラウド テンプレート タスクの出力内のクラウド マシンと IP アドレスを、テストする URL として使用します。

結果

完了です。クラウド テンプレート タスクの展開の詳細と JSON 出力にクラウド マシン名と IP アドレスが見つかりました。これらを使用して、クラウド テンプレート タスクの出力をパイプラインの REST タスクの URL 入力にバインドしました。

次のステップ

引き続き、パイプライン内の他のタスクで、クラウド テンプレート タスク内のリソースのバインド変数を使用する方法について確認します。

REST API を使用して Code Stream を他のアプリケーションと統合する方法

Code Stream には、REST プラグインが提供されています。このプラグインを使用すると、REST API を使用する他のアプリケーションと Code Stream を統合できます。これにより、情報をやり取りする必要があるソフトウェア アプリケーションを継続的に開発して配信できます。REST プラグインは、 Code Stream と別のアプリケーションとの間で情報を送受信する API を呼び出します。

REST プラグインでは、次のことが可能です。

- 外部の REST API ベースのシステムと Code Stream パイプラインを統合する。
- Code Stream パイプラインを外部システムのフローの一部として統合する。

REST プラグインは、どの REST API でも動作します。GET、POST、PUT、PATCH、DELETE の各メソッドをサポートしており、 Code Stream と他のアプリケーションとの間で情報を送受信できます。

表 5-7. REST API 経由で通信するためのパイプラインの準備

操作	実行の結果
パイプラインへの REST タスクを追加する。	REST タスクは、アプリケーション間で情報を通信し、パイプライン ステージの連続するタスクに関するステータス情報を提供できます。
REST タスクで REST アクションを選択し、URL を含める。	パイプライン タスクは、パイプラインの実行時に URL を呼び出します。 POST、PUT、PATCH の各アクションには、ペイロードが含まれている必要があります。ペイロードでは、パイプラインの実行時にパイプラインとタスク プロパティをバインドできます。
使用例を検討する。	REST プラグインの使用例： ビルドの Git コミット時にタグを作成する REST タスクを追加します。このタスクでは、リポジトリからチェックイン ID を取得する申請をポストします。タスクは、リポジトリにペイロードを送信し、ビルドのタグを作成できます。リポジトリは、そのタグを使用して応答を返すことができます。

REST プラグインを使用して API を呼び出す場合と同様に、パイプラインにポーリング タスクを含めて REST API を呼び出すことができます。API が完了し、パイプライン タスクが終了条件を満たすまで、API をポーリングできます。

また、REST API を使用してパイプラインをインポートおよびエクスポートし、サンプル スクリプトを使用してパイプラインを実行することもできます。

次の手順では、単純な URL を取得しています。

手順

1 パイプラインを作成するには、[パイプライン] - [新しいパイプライン] - [空白のキャンバス] の順にクリックします。

2 パイプライン ステージで、[+ シーケンシャル タスク] をクリックします。

3 タスク ペインで、次の REST タスクを追加します。

   a タスクの名前を入力します。

   b [タイプ] ドロップダウン メニューで、[REST] を選択します。

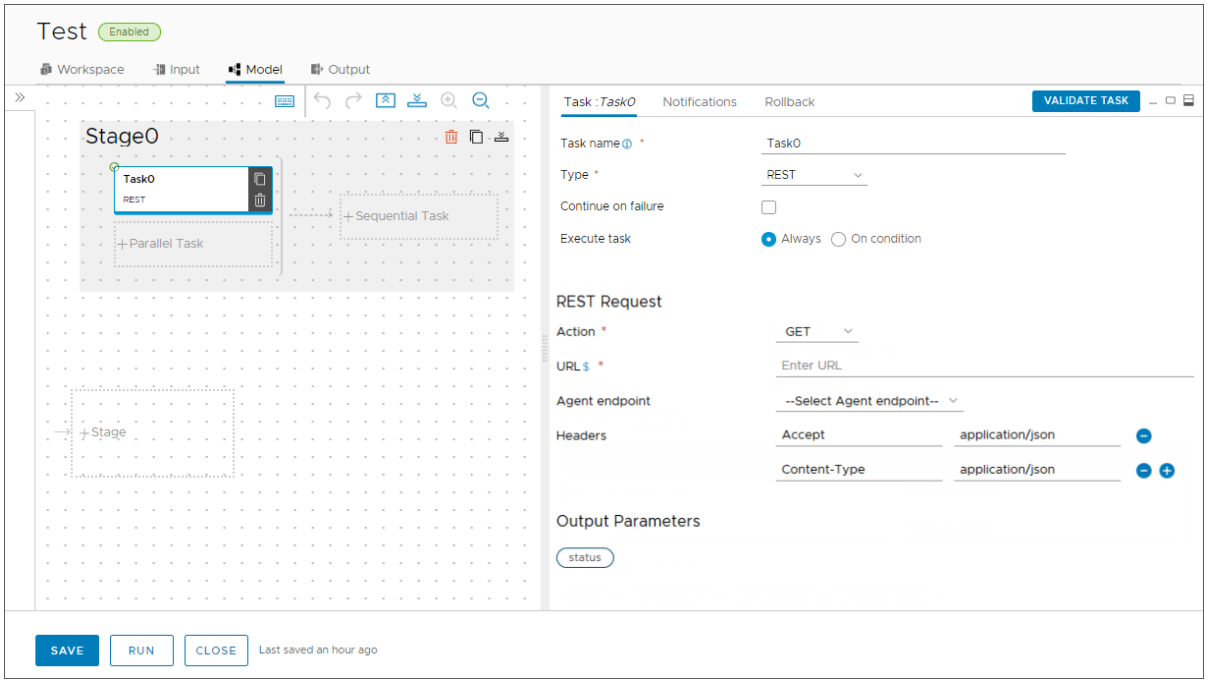
   c [REST リクエスト] 領域で、[GET] を選択します。

      REST タスクが別のアプリケーションにデータを申請するようにするには、GET メソッドを選択します。別のアプリケーションにデータを送信するには、POST メソッドを選択します。

   d REST API エンドポイントを識別する URL を入力します。たとえば、`https://www.google.com` です。

      他のアプリケーションからデータをインポートする REST タスクには、ペイロード変数を含めることができます。たとえば、インポート アクションの場合、`${Stage0.export.responseBody}` と入力できます。応答データのサイズが 5 MB を超えると、REST タスクが失敗することがあります。

   e タスクの認証を行うには、[ヘッダーの追加] をクリックし、ヘッダーのキーと値を入力します。



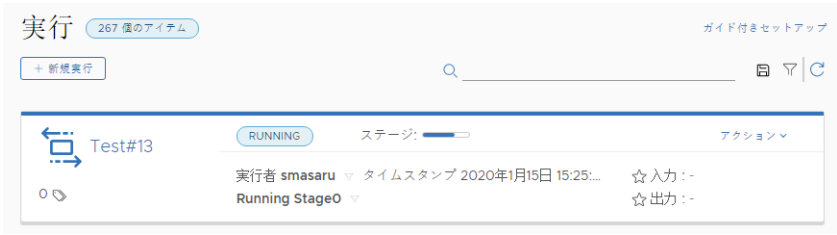
4 パイプラインを保存するには、[保存] をクリックします。

5 [パイプライン] タブで、[パイプラインを有効にする] をクリックします。

6 [保存] をクリックし、[閉じる] をクリックします。

7 [実行] をクリックします。

8 パイプラインの実行を監視するには、[実行] をクリックします。



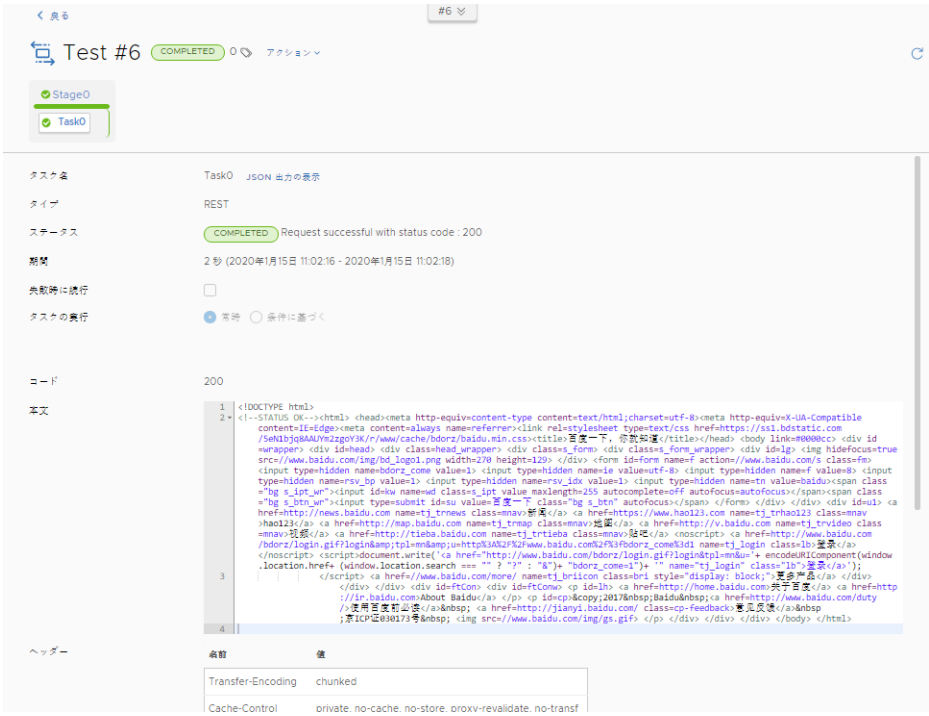
9 REST タスクから想定どおりの情報が返されたことを確認するには、パイプライン実行とタスク結果を調べます。

a パイプラインの完了後、申請したデータが他のアプリケーションから返されたことを確認するには、パイプライン実行へのリンクをクリックします。

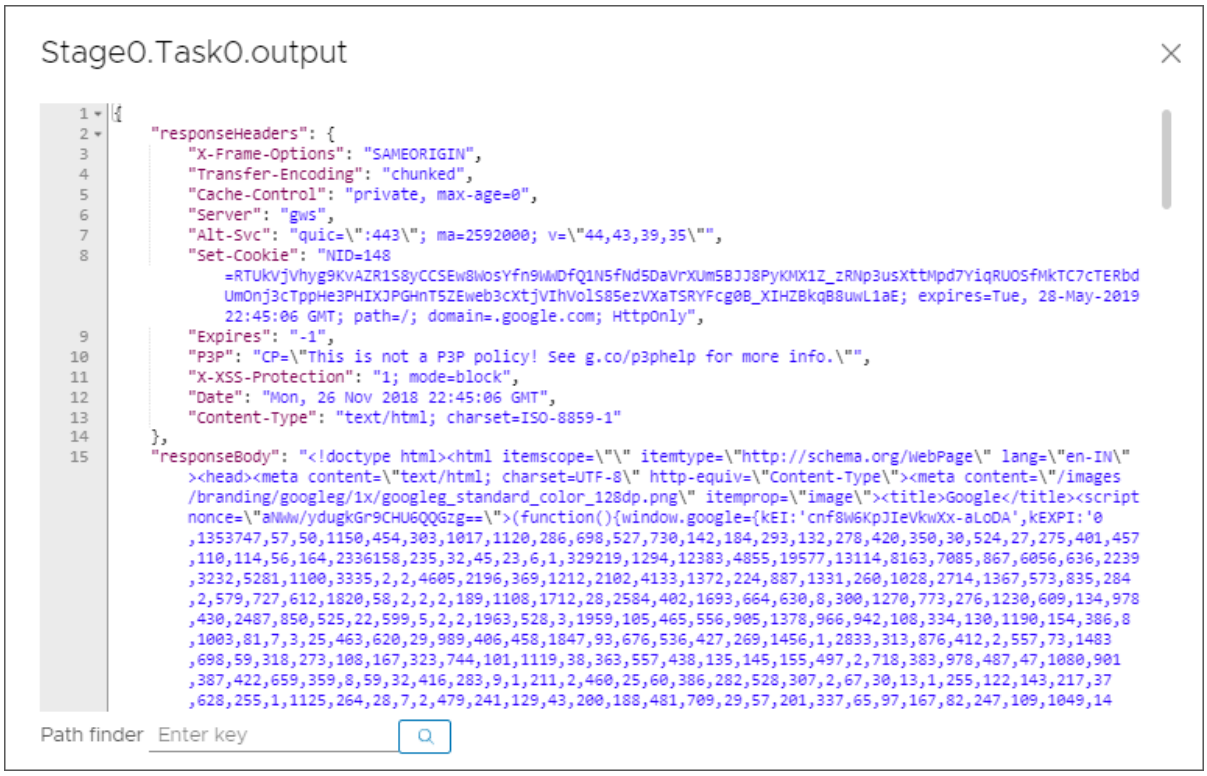
b パイプラインの REST タスクをクリックします。

c パイプライン実行で、タスクをクリックし、タスクの詳細を観察して、REST タスクから想定どおりの結果が返されたことを確認します。

タスクの詳細には、応答コード、本文、ヘッダー キー、および値が表示されます。



10 JSON 出力を表示するには、[JSON 出力の表示] をクリックします。



結果

完了です。REST API を呼び出し、REST プラグインを使用して Code Stream と別のアプリケーション間で情報を送信する REST タスクを設定しました。

次のステップ

引き続きパイプラインで REST タスクを使用してコマンドを実行し、 Code Stream を他のアプリケーションと統合して、ソフトウェア アプリケーションを開発して配信できるようにします。ポーリング タスクを使用すると、API が完了し、パイプライン タスクが終了条件を満たすまで、API をポーリングできます。

パイプラインを Code Stream のコードとして利用する方法

DevOps 管理者または開発者は、ユーザー インターフェイスを使用するのではなく、YAML コードを使用して Code Stream でパイプラインを作成することができます。パイプラインをコードとして作成する場合は、任意のエディタを使用して、パイプライン コードにコメントを挿入できます。

パイプライン コードでは、環境変数やセキュリティ認証情報などの外部構成を参照できます。パイプライン コードで使用する変数を更新する場合、パイプライン コードを更新せずに変数を更新できます。

パイプライン YAML コードをテンプレートとして使用して、他のパイプラインをクローン作成および作成し、テンプレートを他のユーザーと共有できます。

パイプライン コード テンプレートをソース管理リポジトリに保存して、バージョン管理と更新の追跡を行うことができます。ソース管理システムを使用すると、パイプライン コードのバックアップを簡単に作成し、必要に応じてリストアすることができます。

前提条件

- コード エディタがあることを確認します。
- ソース管理リポジトリにパイプライン コードを保存する場合は、作業インスタンスにアクセスできることを確認します。

手順

1 コード エディタで、ファイルを作成します。

2 サンプル パイプライン コードをコピーして貼り付け、特定のパイプラインのニーズを反映するように更新します。

3 エンドポイントをパイプライン コードに含めるには、サンプル エンドポイント コードをコピーして貼り付け、エンドポイントを反映するように更新します。

   パイプライン ワークスペースで Kubernetes API エンドポイントを使用する場合、 Code Stream は、継続的インテグレーション (CI) タスクまたはカスタム タスクを実行するために必要な ConfigMap、シークレット、ポッドなどの Kubernetes リソースを作成します。 Code Stream は、NodePort を使用してコンテナと通信します。

   Code Stream パイプライン ワークスペースでは、継続的インテグレーション タスクとカスタム タスクで Docker と Kubernetes がサポートされます。

   ワークスペースの構成に関する詳細については、パイプライン ワークスペースの構成を参照してください。

4 コードを保存します。

5 パイプライン コードを保存してバージョン管理するには、ソース管理リポジトリにコードをチェックインします。

6 継続的インテグレーションおよび継続的デリバリのパイプラインを作成する場合は、Kubernetes YAML ファイルをインポートする必要があります。

   Kubernetes YAML ファイルをインポートするには、スマート パイプライン テンプレートの [継続的デリバリ] 領域で Kubernetes YAML ファイルを選択し、[プロセス] をクリックします。または、API を使用します。

結果

コードの例を使用して、パイプラインとエンドポイントを表す YAML コードを作成しました。

例： パイプラインとエンドポイントの YAML コードの例

この YAML コードの例には、パイプラインの Code Stream ネイティブ ビルド、ステージ、タスク、通知などのワークスペースを表すセクションが含まれています。

サポートされているプラグインのコードの例については、6 章 エンドポイントへの Code Stream の接続を参照してください。

```
---
kind: PIPELINE
name: myPipelineName
tags:
```

```
  - tag1
  - tag2


# Ready for execution
enabled: false

#Max number of concurrent executions
concurrency: 10

#Input Properties
input:
  input1: '30'
  input2: 'Hello'

#Output Properties
output:
  BuildNo: '${Dev.task1.buildNo}'
  Image: '${Dev.task1.image}'

#Workspace Definition
ciWorkspace:
  image: docker:maven-latest
  path: /var/tmp
  endpoint: my-k8s
  cache:
    - ~/.m2

# Starred Properties
starred:
  input: input1
  output: output1

# Stages in order of execution
stageOrder:
  - Dev
  - QA
  - Prod

# Task Definition Section
stages:
  Dev:
    taskOrder:
      - Task1, Task6
      - Task2 Long, Task Long Long
      - Task5
    tasks:
      Task1:
        type: jenkins
        ignoreFailure: false
        preCondition: ''
        endpoints:
          jenkinsServer: myJenkins
        input:
          job: Add Two Numbers
```

```
          parameters:
            number1: 10
            number2: 20
      Task2:
        type: blah
        # repeats like Task1 above
  QA:
    taskOrder:
      - TaskA
      - TaskB
    tasks:
      TaskA:
        type: ssh
        ignoreFailure: false
        preCondition: ''
        input:
          host: x.y.z.w
          username: abcd
          password: ${var.mypassword}
          script: >
            echo "Hello, remote server"
      TaskB:
        type: blah
        # repeats like TaskA above

# Notificatons Section
notifications:
  email:
    - stage: Dev #optional ; if not found - use pipeline scope
      task: Task1 #optional; if not found use stage scope
      event: SUCCESS
      endpoint: default
      to:
         - user@yourcompany.com
         - abc@yourcompany.com
      subject: 'Pipeline ${name} has completed successfully'
      body: 'Pipeline ${name} has completed successfully'

  jira:
    - stage: QA #optional ; if not found - use pipeline scope
      task: TaskA #optional; if not found use stage scope
      event: FAILURE
      endpoint: myJiraServer
      issuetype: Bug
      project: Test
      assignee: abc
      summary: 'Pipeline ${name} has failed'
      description: |-
        Pipeline ${name} has failed
        Reason - ${resultsText}
  webhook:
    - stage: QA #optional ; if not found - use pipeline scope
      task: TaskB #optional; if not found use stage scope
      event: FAILURE
      agent: my-remote-agent
```

```
      url: 'http://www.abc.com'
      headers: #requestHeaders: '{"build_no":"123","header2":"456"}'
          Content-Type: application/json
          Accept: application/json
      payload: |-
        Pipeline ${name} has failed
        Reason - ${resultsJson}
---
```

この YAML コードは、Jenkins エンドポイントの例を表します。

```
---
name: My-Jenkins
tags:
- My-Jenkins
- Jenkins
kind: ENDPOINT
properties:
  offline: true
  pollInterval: 15.0
  retryWaitSeconds: 60.0
  retryCount: 5.0
  url: http://urlname.yourcompany.com:8080
description: Jenkins test server
type: your.jenkins:JenkinsServer
isLocked: false
---
```

この YAML コードは、Kubernetes エンドポイントの例を表します。

```
---
name: my-k8s
tags: [
  ]
kind: ENDPOINT
properties:
  kubernetesURL: https://urlname.examplelocation.amazonaws.com
  userName: admin
  password: encryptedpassword
description: ''
type: kubernetes:KubernetesServer
isLocked: false
---
```

次のステップ

パイプラインを実行し、必要に応じて調整します。パイプラインを実行して結果を確認する方法を参照してください。

エンドポイントへの Code Stream の接続

6

Code Stream はプラグインを使用して開発ツールと統合されます。サポートされているプラグインには、Jenkins、Bamboo、vRealize Operations、Bugzilla、Team Foundation Server、Git などがあります。

また、 Code Stream を他の開発アプリケーションと統合する独自のプラグインを開発することもできます。

Code Stream には通知タイプとしての Jira チケット作成機能が含まれるため、 Code Stream を Jira に統合するために外部プラグインは必要ではありません。パイプライン ステータスに Jira チケットを作成するには、Jira エンドポイントを追加する必要があります。

この章には、次のトピックが含まれています。

- Code Stream でのエンドポイントとは
- Code Stream を Jenkins と統合する方法
- Code Stream と Git の連携方法
- Code Stream を Gerrit と統合する方法
- Code Stream を vRealize Orchestrator と統合する方法

Code Stream でのエンドポイントとは

エンドポイントは、Code Stream に接続してパイプラインの実行に必要なデータを提供する DevOps アプリケーションのインスタンスです。データ ソース、リポジトリ、通知システムなどがあります。

Code Stream でのロールによって、エンドポイントの使用方法が決まります。

- 管理者および開発者は、エンドポイントを作成、更新、削除、および表示できます。
- 管理者は、エンドポイントを制限ありとしてマークし、制限付きのエンドポイントを使用するパイプラインを実行できます。
- 閲覧者ロールを持つユーザーは、エンドポイントを表示することはできますが、作成、更新、および削除することはできません。

詳細については、 Code Stream でユーザー アクセスと承認を管理する方法を参照してください。

Code Stream をエンドポイントに接続するには、次の手順を実行します。

1 パイプラインにタスクを追加します。

2 エンドポイントと通信するようにタスクを構成します。

3　[検証] をクリックして、 Code Stream がエンドポイントに接続できることを確認します。

4　パイプラインを実行すると、タスクがエンドポイントに接続されて、タスクが実行可能になります。

これらのエンドポイントを使用するタスクのタイプについては、Code Stream で使用可能なタスクのタイプを参照してください。

表 6-1. Code Stream がサポートするエンドポイント

エンドポイント	提供する内容	サポートされるバージョン	要件
Bamboo	ビルド計画を作成します。	6.9.*	
Docker	ネイティブ ビルドでは、展開に Docker ホストを使用できます。		パイプラインに Docker Hub のイメージが含まれている場合は、パイプラインを実行する前に、イメージに `cURL` または `wget` が組み込まれていることを確認する必要があります。パイプラインが実行されると、 Code Stream は、`cURL` または `wget` を使用してコマンドを実行するバイナリ ファイルをダウンロードします。
Docker レジストリ	Docker ビルド ホストがイメージをプルできるようにコンテナ イメージを登録します。	2.7.1	
Gerrit	レビューとトリガーのために Gerrit サーバに接続します。	2.14.*	
Git	開発者がコードを更新してリポジトリにチェックインするときに、パイプラインをトリガします。	Git Hub Enterprise 2.1.8 Git Lab Enterprise 11.9.12-ee	
Jenkins	コード アーティファクトをビルドします。	1.6.* および 2.*	
JIRA	パイプライン タスクが失敗したときに JIRA チケットを作成します。	8.3.*	
Kubernetes	コンテナ化されたアプリケーションを展開、スケール、および管理するための手順を自動化します。	Cloud Assembly 8.4 以降でサポートされているすべてのバージョン Cloud Assembly 8.3 以前では 1.18	パイプライン ワークスペースで Kubernetes API エンドポイントを使用する場合、 Code Stream は、継続的インテグレーション (CI) タスクまたはカスタム タスクを実行するために必要な ConfigMap、シークレット、ポッドなどの Kubernetes リソースを作成します。 Code Stream は、NodePort を使用してコンテナと通信します。 ワークスペースの構成に関する詳細については、パイプライン ワークスペースの構成を参照してください。
PowerShell	Windows または Linux マシンで、PowerShell スクリプトを実行するタスクを作成します。	4 および 5	
SSH	Windows または Linux マシンで、SSH スクリプトを実行するタスクを作成します。	7.0	

表 6-1. Code Stream がサポートするエンドポイント （続き）

エンドポイント	提供する内容	サポートされるバージョン	要件
TFS Team Foundation Server	ソース コード、自動化されたビルド、テスト、および関連するアクティビティを管理します。	2015 および 2017	
vRealize Orchestrator	ビルド プロセスのワークフローを配置して自動化します。	7.* および 8.*	

GitHub エンドポイントの YAML コードの例

この YAML コードの例では、Git タスクで参照できる GitHub エンドポイントを定義しています。

```
---
name: github-k8s
tags: [
  ]
kind: ENDPOINT
properties:
  serverType: GitHub
  repoURL: https://github.com/autouser/testrepok8s
  branch: master
  userName: autouser
  password: encryptedpassword
  privateToken: ''
description: ''
type: scm:git
isLocked: false
---
```

Code Stream を Jenkins と統合する方法

Code Stream には、Jenkins プラグインが搭載されています。Jenkins ジョブをトリガしてソース コードをビルドおよびテストするというものです。この Jenkins プラグインは、テスト ケースを実行し、カスタム スクリプトを使用できます。

パイプラインで Jenkins ジョブを実行するには、Jenkins サーバを使用し、 Code Stream で Jenkins エンドポイントを追加します。次に、パイプラインを作成し、Jenkins タスクを追加します。

Code Stream で Jenkins タスクと Jenkins エンドポイントを使用すると、Jenkins でマルチブランチ ジョブをサポートするパイプラインを作成できます。マルチブランチ ジョブには、Git リポジトリの各ブランチに独立したジョブが含まれます。マルチブランチ ジョブをサポートするパイプラインを Code Stream 内に作成するには、次の処理を行います。

- Jenkins タスクは、Jenkins サーバ上の複数のフォルダにある Jenkins ジョブを実行できます。
- Jenkins タスク設定のフォルダ パスをオーバーライドして、別のフォルダ パスを使用できます。このパスは、Code Stream の Jenkins エンドポイントで定義されているデフォルトのパスをオーバーライドします。

- Code Stream のマルチブランチ パイプラインは、Git リポジトリまたは GitHub リポジトリに含まれる `.groovy` タイプの Jenkins ジョブ ファイルを検出し、リポジトリ内でスキャンした各ブランチでジョブの作成を開始します。
- Jenkins エンドポイントで定義されているデフォルトのパスを、Jenkins タスク設定で指定したパスでオーバーライドして、メインの Jenkins ジョブ内の任意のブランチに関連付けられたジョブとパイプラインを実行できます。

前提条件

- バージョン 1.561 以降を実行している Jenkins サーバをセットアップします。
- Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。
- Jenkins サーバにジョブがあり、パイプライン タスクがそのジョブを実行できるようになっていることを確認します。

手順

1 Jenkins エンドポイントを追加し、検証します。

   a [エンドポイント] - [新規エンドポイント] の順にクリックします。

   b プロジェクトを選択し、エンドポイントのタイプとして [Jenkins] を選択します。次に、名前と説明を入力します。

   c このエンドポイントがインフラストラクチャのビジネス クリティカルなコンポーネントである場合は、[制限付きとしてマーク] を有効にします。

   d Jenkins サーバの URL を入力します。

e Jenkins サーバにログインするためのユーザー名とパスワードを入力します。次に、その他の情報を入力します。

表 6-2. Jenkins エンドポイントのその他の情報

エンドポイントの入力情報	説明
フォルダ パス	ジョブをグループ化するフォルダのパス。Jenkins は、フォルダ内のすべてのジョブを実行できます。サブフォルダを作成できます。例： ■ `folder_1` に `job_1` を含めることができます ■ `folder_1` に `folder_2` を含め、そこに `job_2` を含めることができます `folder_1` のエンドポイントを作成するとフォルダ パスは `job/folder_1` となり、エンドポイントには `job_1` のみが表示されます。 `folder_2` という名前の子フォルダにあるジョブのリストを取得するには、そのフォルダ パスを `/job/folder_1/job/folder_2/` として使用するエンドポイントを別途作成する必要があります。
マルチブランチ Jenkins ジョブのフォルダ パス	マルチブランチ Jenkins ジョブをサポートするには、Jenkins タスクで、Jenkins サーバの URL とジョブの完全なパスを含むフル パスを入力します。Jenkins タスクにフォルダ パスを含めると、そのパスは Jenkins エンドポイントに表示されるパスをオーバーライドします。Jenkins タスクでカスタム フォルダ パスを使用した場合、 Code Stream はそのフォルダ内のジョブだけを表示します。 ■ 例：`https://server.yourcompany.com/job/project` ■ パイプラインでメインの Jenkins ジョブもトリガする必要がある場合は、次のパスを使用します。`https://server.yourcompany.com/job/project/job/main`
URL	Jenkins サーバのホスト URL。URL を `protocol://host:port` の形式で入力します。例：`http://192.10.121.13:8080`
ポーリング間隔	Code Stream が Jenkins サーバをポーリングして更新を確認する間隔。

表 6-2. Jenkins エンドポイントのその他の情報 （続き）

エンドポイントの入力情報	説明
申請の再試行回数	Jenkins サーバのスケジュール設定されたビルド申請を再試行する回数。
再試行の待機時間	Jenkins サーバのビルド申請を再試行するまでの待機時間（秒単位）。

f [検証] をクリックし、エンドポイントが Code Stream に接続されていることを確認します。接続されていない場合は、エラーを修正し、[保存] をクリックします。

2 コードをビルドするには、パイプラインを作成し、Jenkins エンドポイントを使用するタスクを追加します。

a [パイプライン] - [新しいパイプライン] - [空白のキャンバス] の順にクリックします。

b デフォルトのステージをクリックします。

c [タスク] 領域で、タスクの名前を入力します。

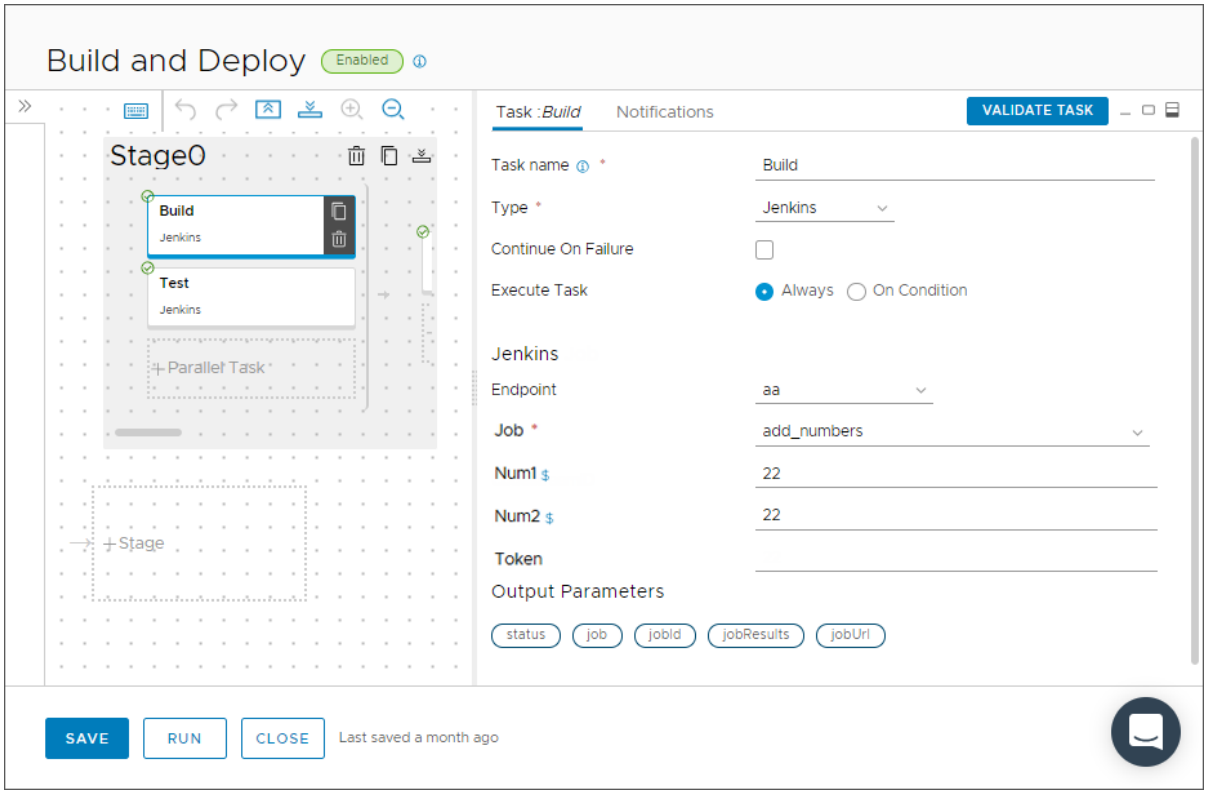
d タスク タイプに [Jenkins] を選択します。

e 作成した Jenkins エンドポイントを選択します。

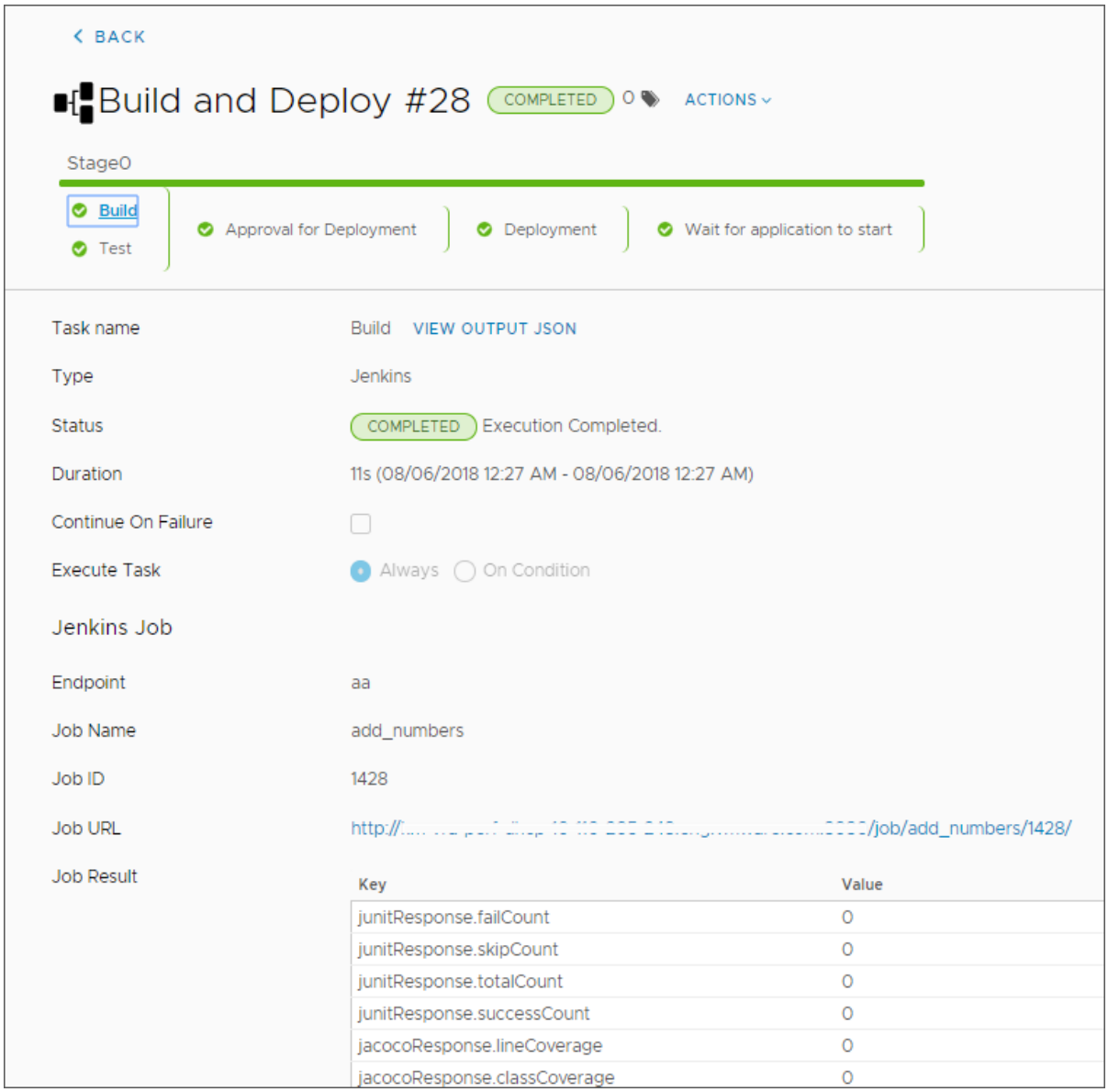
f ドロップダウン メニューから、パイプラインが実行する Jenkins サーバ上のジョブを選択します。

g　ジョブのパラメータを入力します。

h　Jenkins ジョブの認証トークンを入力します。

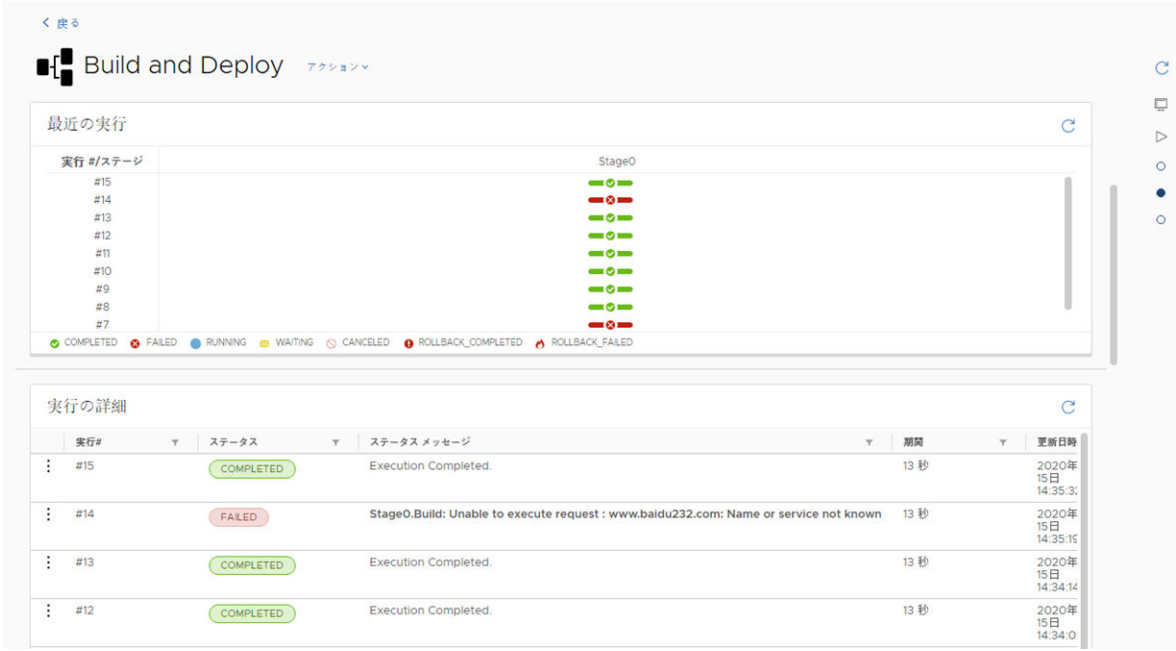


3 パイプラインを有効にして実行し、パイプラインの実行状況を表示します。



4 パイプライン ダッシュボードで実行の詳細とステータスを確認します。

障害があれば特定し、その理由を突き止めます。また、パイプラインの実行期間、完了、および障害に関するトレンドも表示できます。



結果

完了です。エンドポイントを追加し、パイプラインを作成し、コードをビルドする Jenkins タスクを設定して、Code Stream を Jenkins と統合しました。

例： Jenkins ビルド タスクの YAML の例

この例で使用したタイプの Jenkins ビルド タスクの場合、YAML は次のコードに似ています。通知が有効になっています。

```
test:
  type: Jenkins
  endpoints:
    jenkinsServer: jenkins
  input:
    job: Add two numbers
    parameters:
      Num1: '23'
      Num2: '23'
```

次のステップ

詳細については、他のセクションを参照してください。6 章 エンドポイントへの Code Stream の接続を参照してください。

Code Stream と Git の連携方法

Code Stream には、GitHub、GitLab、または Bitbucket リポジトリでコード変更が発生したときにパイプラインをトリガする方法が用意されています。Git トリガは、監視するリポジトリのブランチにある Git エンドポイントを使用します。 Code Stream は、Webhook を介して Git エンドポイントに接続します。

Code Stream で Git エンドポイントを定義するには、プロジェクトを選択し、エンドポイントを配置する Git リポジトリのブランチを入力します。プロジェクトでは、エンドポイントやその他の関連オブジェクトとパイプラインがグループ化されます。Webhook の定義でプロジェクトを選択するときに、トリガするエンドポイントとパイプラインを選択します。

注： エンドポイントを使用して Webhook を定義し、後でエンドポイントを編集する場合、Webhook のエンドポイントの詳細を変更することはできません。エンドポイントの詳細を変更するには、Webhook を削除し、そのエンドポイントを使用して再定義する必要があります。 Code Stream で Git トリガを使用してパイプラインを実行する方法を参照してください。

Webhook の構成画面で同じ Git エンドポイントを使用して、ブランチ名に別の値を指定することにより、複数のブランチに対して複数の Webhook を作成できます。同じ Git リポジトリ内の別のブランチに別の Webhook を作成する際に、複数のブランチに対して Git エンドポイントのクローン作成を複数回行う必要はありません。代わりに、Webhook 内でブランチ名を指定すれば、Git エンドポイントを再利用できるようになります。Git Webhook のブランチがエンドポイントのブランチと同じである場合は、Git Webhook 画面でブランチ名を入力する必要はありません。

前提条件

- 接続する予定の GitHub、GitLab、または Bitbucket リポジトリにアクセスできることを確認します。
- Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。

手順

1 Git エンドポイントを定義します。

   a [エンドポイント] - [新規エンドポイント] の順にクリックします。

   b プロジェクトを選択し、エンドポイント タイプとして [Git] を選択します。次に、名前と説明を入力します。

c このエンドポイントがインフラストラクチャのビジネス クリティカルなコンポーネントである場合は、[制限付きとしてマーク] を有効にします。

パイプラインで制限付きのエンドポイントを使用する場合は、管理者がパイプラインを実行して、パイプラインの実行を承認する必要があります。エンドポイントまたは変数が制限付きとマークされている場合に、管理者以外のユーザーがパイプラインをトリガすると、パイプラインはそのタスクで一時停止し、管理者が再開するまで待機します。

プロジェクト管理者は、制限付きのエンドポイントまたは変数が含まれているパイプラインを起動できます。そのためには、これらのリソースが、ユーザーがプロジェクト管理者になっているプロジェクトに含まれている必要があります。

管理者以外のユーザーが、制限されたリソースを含むパイプラインを実行すると、制限されたリソースを使用するタスクでパイプラインが停止します。その場合は、管理者がパイプラインを再開する必要があります。

制限付きリソース、および [制限付きパイプラインの管理] 権限を含むカスタム ロールの詳細については、以下を参照してください。

- Code Stream でユーザー アクセスと承認を管理する方法
- 2 章 リリース プロセスをモデリングするための Code Stream の設定

d サポートされている Git サーバ タイプの 1 つを選択します。

e パスに含まれるサーバへの API ゲートウェイを持つリポジトリの URL を入力します。例：

GitHub の場合は、**`https://api.github.com/vmware-example/repo-example`** と入力します。

BitBucket の場合は、**`https://api.bitbucket.org/{user}/{repo name}`** または **`http(s)://{bitbucket-enterprise-server}/rest/api/1.0/users/{username}/repos/{repo name}`** と入力します。

f エンドポイントが配置されるリポジトリ内のブランチを入力します。

g 認証タイプを選択し、GitHub、GitLab、または Bitbucket のユーザー名を入力します。次に、そのユーザー名とペアになっているプライベート トークンを入力します。

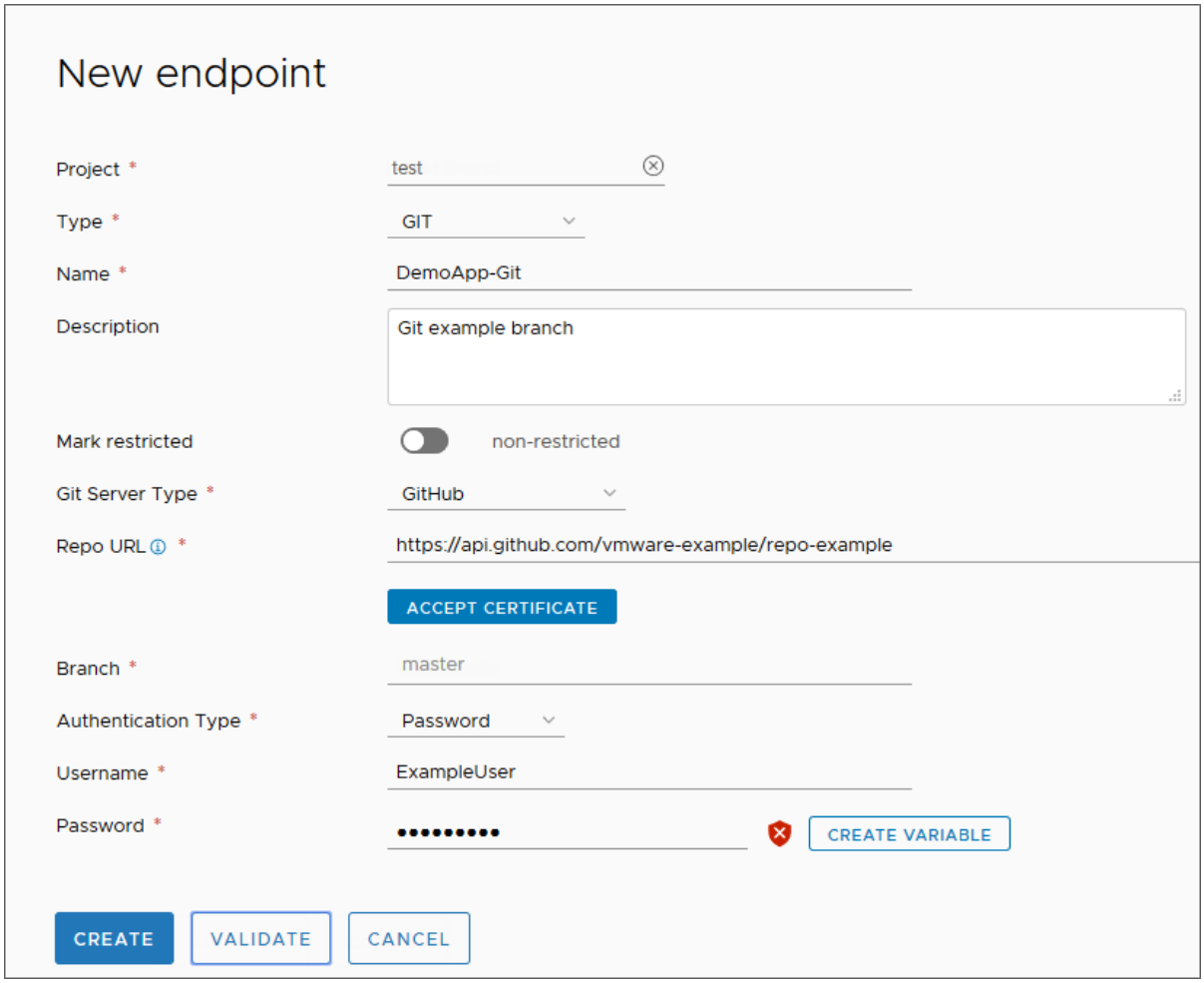
- パスワードの場合：後で Webhook を作成するには、パスワードのプライベート トークンを入力する必要があります。Git 用 Webhook は、ベーシック認証を使用して作成されたエンドポイントをサポートしません。

  機密性のある情報を非表示および暗号化するには、シークレット変数を使用します。非表示にして暗号化して実行時に使用を制限する文字列、パスワード、URL については、制限付き変数を使用します。たとえば、パスワードや URL にシークレット変数を使用します。シークレット変数も制限付き変数も、パイプライン内の任意のタイプのタスクで使用できます。

- プライベート トークンの場合：このトークンは Git 固有であり、特定のアクションへのアクセスを提供します。https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html を参照してください。また、プライベート トークンの変数を作成することもできます。

2 [検証] をクリックし、エンドポイントが Code Stream.に接続することを確認します。

接続しない場合は、エラーを修正してから [作成] をクリックします。



次のステップ

詳細については、他のセクションを参照してください。 Code Stream で Git トリガを使用してパイプラインを実行する方法を参照してください。

Code Stream を Gerrit と統合する方法

Code Stream では、Gerrit プロジェクトでコード レビューが発生したときにパイプラインをトリガできます。Gerrit 用トリガの定義には、イベント タイプに応じて実行する必要がある Gerrit プロジェクトとパイプラインが含まれます。

Gerrit 用トリガは、監視対象の Gerrit サーバにある Gerrit リスナーを使用します。 Code Stream で Gerrit エンドポイントを定義するには、プロジェクトを選択し、Gerrit サーバの URL を入力します。次に、そのサーバに Gerrit リスナーを作成するときに、エンドポイントを指定します。

FIPS が有効になっている vRealize Automation インスタンスで Gerrit サーバを Code Stream エンドポイントとして使用している場合は、Gerrit 構成ファイルに正しいメッセージ認証キーが含まれていることを確認する必要があります。Gerrit サーバの構成ファイルに正しいメッセージ認証キーが含まれていない場合は、サーバが正常に起動せず、「`PrivateKey/PassPhrase is incorrect`」というメッセージが表示されます。

前提条件

- 接続する Gerrit サーバにアクセスできることを確認します。
- Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。

手順

1 Gerrit エンドポイントを定義します。

   a [構成] - [エンドポイント] の順にクリックして、[新規エンドポイント] をクリックします。

   b プロジェクトを選択し、エンドポイントのタイプとして [Gerrit] を選択します。次に、名前と説明を入力します。

   c このエンドポイントがインフラストラクチャのビジネス クリティカルなコンポーネントである場合は、[制限付きとしてマーク] を有効にします。

   d Gerrit サーバの URL を入力します。

   デフォルト ポートを使用するには、URL でポート番号を指定するか、値を空白のままにします。

   e Gerrit サーバのユーザー名とパスワードを入力します。

   パスワードを暗号化する必要がある場合は、[変数の作成] をクリックし、次のいずれかのタイプを選択します。

   - シークレット。パスワードは、何らかのロールを持つユーザーがパイプラインを実行すると解決されます。
   - 制限付き。パスワードは、管理者ロールを持つユーザーがパイプラインを実行すると解決されます。

   値には、Jenkins サーバのパスワードなど、保護する必要があるパスワードを入力します。

   f プライベート キーの場合は、Gerrit サーバへの安全なアクセスに使用する SSH キーを入力します。

   このキーは、`.ssh` ディレクトリにある RSA プライベート キーです。

   g （オプション） プライベート キーにパスフレーズが関連付けられている場合は、そのパスフレーズを入力します。

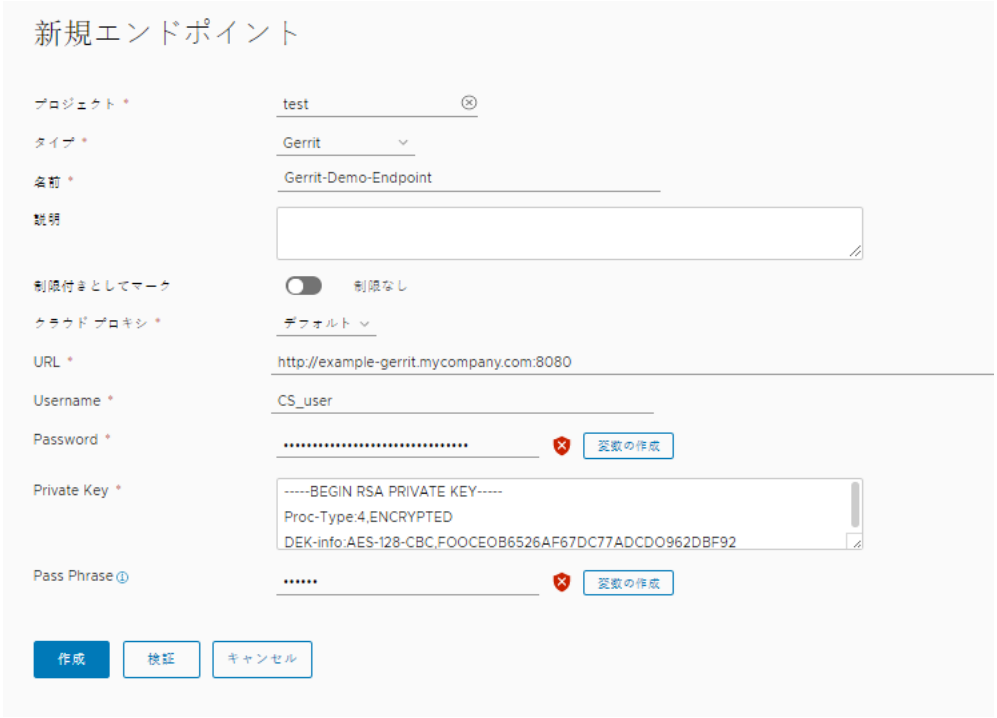
   パスフレーズを暗号化するには、[変数の作成] をクリックし、次のいずれかのタイプを選択します。

   - シークレット。パスワードは、何らかのロールを持つユーザーがパイプラインを実行すると解決されます。
   - 制限付き。パスワードは、管理者ロールを持つユーザーがパイプラインを実行すると解決されます。

   値には、SSH サーバのパスフレーズなど、保護するパスフレーズを入力します。

2 [検証] をクリックし、 Code Stream の Gerrit エンドポイントが Gerrit サーバに接続されていることを確認します。

接続されていない場合は、エラーを修正し、もう一度 [検証] をクリックします。



3 [作成] をクリックします。

4 vRealize Automation 環境で FIPS が有効になっていることを確認します。または、Jenkins ジョブで Jenkins URL を使用して、FIPS が有効な環境を作成します。

a コマンド ラインからコマンドを実行するには、SSH 経由で vRealize Automation 8.x アプライアンスに接続し、root ユーザーとしてログインします。たとえば、`https://cava-1-234-567.yourcompanyFQDN.com` などの完全修飾ドメイン名 URL にポート 22、5480、または 443 で接続します。

b vRealize Automation の FIPS を確認するには、コマンド `vracli security fips` を実行します。

c コマンドが `FIPS mode: strict` を返すことを確認します。

5 Gerrit サーバが、FIPS が有効な vRealize Automation インスタンスのエンドポイントである場合は、Gerrit 構成ファイルに正しいメッセージ認証 (MAC) キーが含まれていることを確認します。

a Gerrit を開き、SSH キー ペアを作成します。

b `'$site_path'/etc/gerrit.config` の Gerrit サーバ構成ファイルを見つけます。

c Gerrit サーバ構成ファイルに、`hmac-MD5` を除くメッセージ認証コード (MAC) キーが 1 つ以上含まれていることを確認します。

注： FIPS モードでは、`hmac-MD5` はサポートされる MAC アルゴリズムではありません。Gerrit サーバが正しく起動するようにするには、Gerrit サーバ構成ファイルでこのアルゴリズムを除外する必要があります。Gerrit サーバが正常に起動しない場合は、「`PrivateKey/PassPhrase is incorrect`」というメッセージが表示されます。

プラス記号 (+) で始まる、サポートされているメッセージ認証コード (MAC) のキー名が有効です。ハイフン (-) で始まる MAC キー名は、デフォルトの MAC のリストから削除されます。デフォルトでは、これらのサポートされる MAC は Gerrit サーバの Code Stream で使用できます。

- `hmac-md5-96`
- `hmac-sha1`
- `hmac-sha1-96`
- `hmac-sha2-256`
- `hmac-sha2-512`

次のステップ

詳細については、他のセクションを参照してください。 Code Stream で Gerrit トリガを使用してパイプラインを実行する方法を参照してください。

Code Stream を vRealize Orchestrator と統合する方法

Code Stream を vRealize Orchestrator (`vRO`) と統合すると、`vRO` ワークフローを実行してその機能を拡張できます。vRealize Orchestrator には、サードパーティ製ツールと統合できる事前定義済みのワークフローが多数含まれています。これらのワークフローは、DevOps プロセスの自動化と管理、一括操作の自動化などに役立ちます。

たとえば、パイプラインで `vRO` タスクのワークフローを使用すると、ユーザーを有効にし、ユーザーを削除し、仮想マシンを移動できます。さらに、テスト フレームワークと統合してパイプラインの実行時にコードをテストするなど、さまざまなことができます。code.vmware.com では、vRealize Orchestrator ワークフローのコード例を参照できます。

パイプラインで vRealize Orchestrator ワークフローを使用すると、アプリケーションをビルド、テスト、および展開するときにアクションを実行できます。定義済みのワークフローをパイプラインに含めることも、カスタム ワークフローを作成して使用することもできます。ワークフローごとに入力、タスク、出力があります。

パイプラインで vRO ワークフローを実行するには、パイプラインに含めた vRO タスクで使用できるワークフローのリストに、そのワークフローが表示されている必要があります。

そのワークフローがパイプラインの vRO タスクに表示されるようにするには、管理者が vRealize Orchestrator で次の手順を実行する必要があります。

1 CODESTREAM タグを vRO ワークフローに適用します。

2 vRO ワークフローをグローバルとしてマークします。

前提条件

- vRealize Orchestrator のオンプレミス インスタンスに管理者としてアクセスできることを確認します。詳細については、管理者に問い合わせるか、vRealize Orchestrator ドキュメントを参照してください。
- Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。
- Code Stream で、パイプラインを作成し、ステージを追加します。

手順

1 管理者として、パイプラインで実行する vRealize Orchestrator ワークフローを準備します。

   a vRealize Orchestrator で、ユーザーを有効にするためのワークフローなど、パイプラインで使用する必要があるワークフローを検索します。

   必要なワークフローがどこにもない場合には作成できます。

   b 検索バーに「**タグ ワークフロー**」と入力し、タグ ワークフロー という名前のワークフローを検索します。

   c タグ ワークフロー という名前のカード上で [実行] をクリックすると、設定領域が表示されます。

   d タグ付けされたワークフロー テキスト領域で、 Code Stream パイプラインで使用するワークフローの名前を入力し、そのワークフローをリストから選択します。

   e タグ および 値 のテキスト領域に CODESTREAM を大文字で入力します。

   f [グローバル タグ] という名前のチェック ボックスをクリックします。

   g [実行] をクリックします。これにより、 Code Stream パイプラインで選択する必要があるワークフローに CODESTREAM という名前のタグが付けられます。

   h ナビゲーション ペインで [ワークフロー] をクリックし、パイプラインで実行されるワークフロー カードに CODESTREAM という名前のタグが表示されることを確認します。

   Code Stream にログインして、パイプラインに vRO タスクを追加すると、タグ付けされたワークフローがワークフロー リストに表示されます。

2 Code Stream で、vRealize Orchestrator インスタンスのエンドポイントを作成します。

   a [エンドポイント] - [新規エンドポイント] の順にクリックします。

   b プロジェクトを選択します。

   c 適切な名前を入力します。

d vRealize Orchestrator エンドポイントの URL を入力します。

次の形式を使用します。**https://vro-appliance.yourdomain.local:8281**

次の形式は使用しないでください。https://vro-appliance.yourdomain.local:8281/vco/api

vRealize Automation アプライアンスに組み込まれている vRealize Orchestrator インスタンスの URL は、アプライアンスの FQDN であり、ポートを含みません。例：**https://vra-appliance.yourdomain.local/vco**

vRealize Automation 8.x 以降の外部 vRealize Orchestrator アプライアンスの場合、アプライアンスの FQDN は次のとおりです。**https://vro-appliance.yourdomain.local**

vRealize Automation 7.x の外部 vRealize Orchestrator アプライアンスの場合、アプライアンスの FQDN は次のとおりです。**https://vro-appliance.yourdomain.local:8281/vco**

エンドポイントの追加時に問題が発生した場合は、必要に応じて、SHA-256 証明書フィンガープリントからコロンを削除して YAML 構成をインポートします。たとえば、**B0:01:A2:72...** は **B001A272...** になります。サンプル YAML コードは次のようになります。

```
```
---
project: Demo
kind: ENDPOINT
name: external-vro
description: ''
type: vro
properties:
  url: https://yourVROhost.yourdomain.local
  username: yourusername
  password: yourpassword
  fingerprint: <your_fingerprint>
```
```

e 入力した URL に証明書が必要な場合は、[証明書を承諾] をクリックします。

f vRealize Orchestrator サーバのユーザー名とパスワードを入力します。

認証に非ローカル ユーザーを使用している場合は、ユーザー名のドメイン部分を省略する必要があります。たとえば、**svc_vro@yourdomain.local** を認証するには、[ユーザー名] テキスト領域に **svc_vro** と入力する必要があります。

3 パイプラインが vRO タスクを実行する準備をします。

a vRO タスクをパイプライン ステージに追加します。

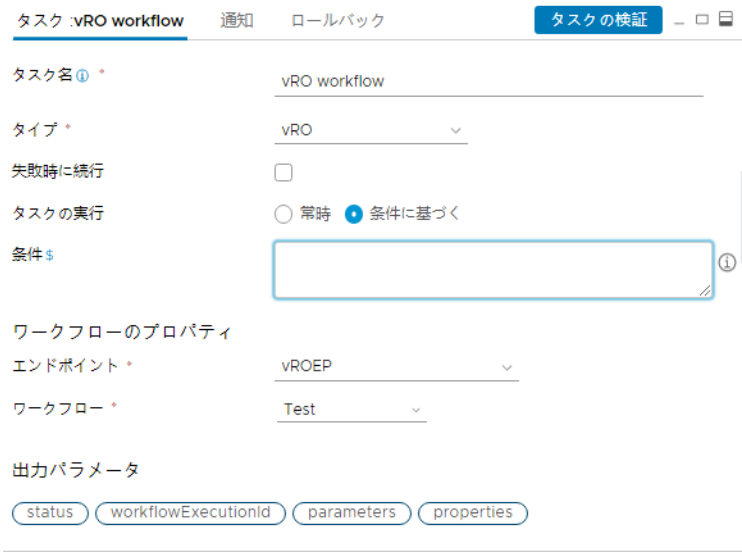
b 適切な名前を入力します。

c [ワークフロー プロパティ] 領域で、vRealize Orchestrator エンドポイントを選択します。

d vRealize Orchestrator で CODESTREAM としてタグ付けしたワークフローを選択します。

作成したカスタム ワークフローを選択した場合は、入力パラメータ値の入力が必要になることがあります。

e [タスクの実行] の [条件に基づく] をクリックします。



f パイプラインの実行時に適用する条件を入力します。

パイプラインを実行するタイミング	条件の選択
条件に基づく	定義した条件が true に評価される場合にのみ、パイプライン タスクを実行します。条件が false の場合、タスクはスキップされます。 vRO タスクには、ブール式を含めることができます。次のオペランドと演算子を使用できます。 ■ `${pipeline.variableName}` などのパイプライン変数。変数を入力するときは、中括弧のみを使用します。 ■ `${Stage1.task1.machines[0].value.hostIp[0]}` などのタスク出力変数。 ■ `${releasePipelineName}` などのデフォルトのパイプライン バインド変数。 ■ `true`、`false`、`'true'`、`'false'` などの大文字と小文字を区別しないブール値。 ■ 引用符を含めない整数値または 10 進数値。 ■ `"test"` や `'test'` など一重引用符または二重引用符で囲んだ文字列値。 ■ `== Equals` や `!= Not Equals` など文字列タイプおよび数値タイプの値。 ■ `>`、`>=`、`<`、`<=` などの関係演算子。 ■ `&&` や `\|\|` などのブール論理。 ■ `+`、`-`、`*`、`/` などの算術演算子。 ■ 丸括弧でネストされた式。 ■ リテラル値 `ABCD` を含む文字列は false に評価され、タスクはスキップされます。 ■ 単項演算子はサポートされていません。 たとえば、`${Stage1.task1.output} == "Passed" || ${pipeline.variableName} == 39` のような条件があります。
常時	[常時] を選択した場合、パイプラインは条件なしでタスクを実行します。

g あいさつのメッセージを入力します。

h [タスクの検証] をクリックし、発生したエラーを修正します。

4 パイプラインを保存し、有効にして実行します。

5 パイプラインの実行後、結果を確認します。

a [実行] をクリックします。

b パイプラインをクリックします。

c タスクをクリックします。

d 結果、入力値、およびプロパティを確認します。

ワークフロー実行 ID、誰がいつタスクに応答したか、また、応答にコメントを含めたかを把握できます。

結果

完了です。 Code Stream で使用する vRealize Orchestrator ワークフローにタグを付け、 Code Stream パイプラインに vRO タスクを追加しました。DevOps 環境でのアクションを自動化するワークフローがパイプラインで実行されるようになります。

例： vRO タスクの出力形式

vRO タスクの出力形式は、この例のようになります。

```
[{
                "name": "result",
                "type": "STRING",
                "description": "Result of workflow run.",
                "value": ""
},
{
                "name": "message",
                "type": "STRING",
                "description": "Message",
                "value": ""
}]
```

次のステップ

引き続きパイプラインに vRO ワークフロー タスクを含めて、開発環境、テスト環境、および本番環境でタスクを自動化できるようにします。

Code Stream でのパイプラインのトリガ

7

特定のイベントが発生したときに、 Code Stream がパイプラインをトリガするように設定できます。

例：

- Docker トリガでは、新しいアーティファクトが作成または更新されたときにパイプラインを実行できます。
- Git トリガでは、開発者がコードを更新したときにパイプラインをトリガできます。
- Gerrit トリガでは、開発者がコードをレビューしたときにパイプラインをトリガできます。

この章には、次のトピックが含まれています。

Code Stream で Docker トリガを使用して、継続的な配信パイプラインを実行する方法

Code Stream の管理者または開発者は、 Code Stream の Docker トリガを使用できます。Docker トリガは、ビルド アーティファクトの作成または更新時に必ずスタンドアローンの継続的デリバリ (CD) パイプラインを実行します。Docker トリガは CD パイプラインを実行します。これにより、新しいまたは更新されたアーティファクトがコンテナ イメージとして Docker Hub リポジトリにプッシュされます。CD パイプラインは、自動ビルドの一部として実行できます。

たとえば、更新されたコンテナ イメージを CD パイプラインを使用して継続的に展開するには、Docker トリガを使用します。コンテナ イメージが Docker レジストリにチェックインされると、Docker Hub の Webhook は、イメージが変更されたことを Code Stream に通知します。この通知は、更新されたコンテナ イメージで実行されるように CD パイプラインをトリガし、イメージを Docker Hub リポジトリにアップロードします。

Docker トリガを使用するには、 Code Stream でいくつかの手順を実行します。

表 7-1. Docker トリガを使用する方法

実行する操作	操作の詳細情報
Docker レジストリ エンドポイントを作成する	Code Stream がパイプラインをトリガするには、Docker レジストリ エンドポイントが必要です。エンドポイントがない場合は、Docker トリガの Webhook を追加するときに、そのエンドポイントを作成するオプションを選択できます。 Docker レジストリ エンドポイントには、Docker Hub リポジトリへの URL が含まれています。
パイプラインの実行時に Docker パラメータを自動挿入する入力パラメータをパイプラインに追加する	Docker パラメータをパイプラインに挿入できます。パラメータには、Docker イベント所有者名、イメージ、リポジトリ名、リポジトリ名前空間、タグを含めることができます。 CD パイプラインには、パイプラインがトリガされる前に Docker Webhook がパイプラインに渡す入力パラメータを含めます。
Docker Webhook を作成する	Code Stream で Docker Webhook を作成すると、対応する Webhook も Docker Hub に作成されます。 Code Stream の Docker Webhook は、Webhook に含まれる URL を介して Docker Hub に接続します。 Webhook は相互に通信し、アーティファクトが Docker Hub で作成または更新されるとパイプラインをトリガします。 Code Stream の Docker Webhook を更新または削除すると、Docker Hub の Webhook も更新または削除されます。
パイプラインに Kubernetes タスクを追加して設定する	Docker Hub リポジトリでアーティファクトが作成または更新されると、パイプラインがトリガされます。次に、パイプラインを使用して Kubernetes クラスタ内の Docker ホストにアーティファクトを展開します。
ローカル YAML 定義をタスクに含める	展開タスクに適用する YAML 定義には、Docker コンテナ イメージが含まれます。プライベート所有のリポジトリからイメージをダウンロードする必要がある場合は、Docker 構成の [シークレット] を含むセクションを YAML ファイルに含める必要があります。スマート パイプライン テンプレートを使用する前の Code Stream での CICD ネイティブ ビルドの計画の CD 部分を参照してください

Docker Hub リポジトリでアーティファクトが作成または更新されると、Docker Hub の Webhook は、 Code Stream の Webhook に通知します。これにより、パイプラインがトリガされます。次のアクションが発生します。

1 Docker Hub は、Webhook 内の URL に POST 要求を送信します。

2 Code Stream は、Docker トリガを実行します。

3 Docker トリガは、CD パイプラインを起動します。

4 CD パイプラインは、アーティファクトを Docker Hub リポジトリにプッシュします。

5 Code Stream は、Docker Webhook をトリガします。これは、アーティファクトを Docker ホストに展開する CD パイプラインを実行します。

この例では、開発 Kubernetes クラスタにアプリケーションを展開する、 Code Stream の Docker エンドポイントと Docker Webhook を作成します。手順には、仮想マシンが Webhook の URL に投稿するペイロードのサンプル コード、使用される API コード、およびセキュア トークンを使用した認証コードが含まれます。

前提条件

- 継続的デリバリ (CD) パイプラインが Code Stream インスタンスにあることを確認します。また、アプリケーションを展開する 1 つ以上の Kubernetes タスクが含まれていることも確認します。4 章 Code Stream でコードをネイティブにビルド、統合、および配信することを計画する を参照してください。
- CD パイプラインが開発用にアプリケーションを展開できる、既存の Kubernetes クラスタにアクセスできることを確認します。
- Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。

手順

1 Docker レジストリ エンドポイントを作成する

   a [エンドポイント] をクリックします。

   b [新規エンドポイント] をクリックします。

   c 既存のプロジェクト名の入力を開始します。

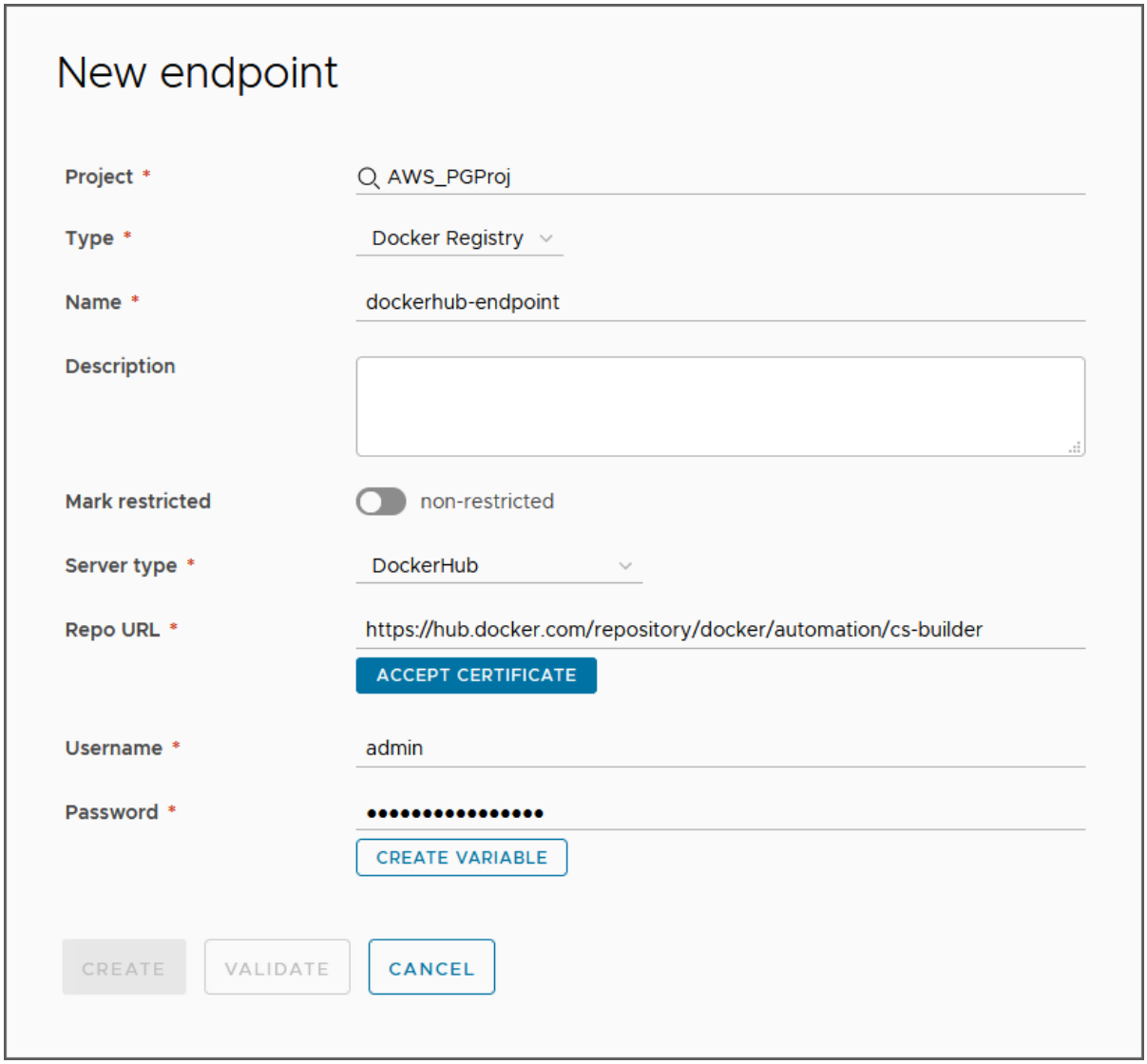
   d タイプには [Docker レジストリ] を選択します。

   e 適切な名前を入力します。

   f サーバ タイプには [DockerHub] を選択します。

g　Docker Hub リポジトリへの URL を入力します。

h　リポジトリにアクセスできる名前とパスワードを入力します。



2　CD パイプラインで、パイプラインの実行時に入力プロパティを Docker パラメータの自動挿入に設定します。

3 Docker Webhook を作成する

a [トリガ] - [Docker] の順にクリックします。

b [Docker の新規 Webhook] をクリックします。

c プロジェクトを選択します。

d 適切な名前を入力します。

e Docker レジストリ エンドポイントを選択します。

エンドポイントがまだない場合は、[エンドポイントの作成] をクリックして作成します。

f トリガする Webhook 用の Docker のパラメータが挿入されるパイプラインを選択します。手順 2 を参照してください。

カスタムで追加された入力パラメータを使用してパイプラインが設定されている場合、[入力パラメータ] リストにはパラメータと値が表示されます。イベントをトリガするパイプラインに渡される入力パラメータの値を入力できます。値は、空白のままにすることもできます。デフォルト値が定義されている場合は、デフォルト値を使用することもできます。

入力タブのパラメータの詳細については、タスクの手動追加を行う前の Code Stream での CICD ネイティブ ビルドの計画を参照してください。

g API トークンを入力します。

CSP API トークンは、 Code Stream との外部 API 接続を認証します。API トークンを取得するには、次の手順を実行します。

1 [トークンの生成] をクリックします。

2 ユーザー名とパスワードに関連付けられているメール アドレスを入力し、[生成] をクリックします。

生成するトークンは 6 か月有効です。これは更新トークンとも呼ばれます。

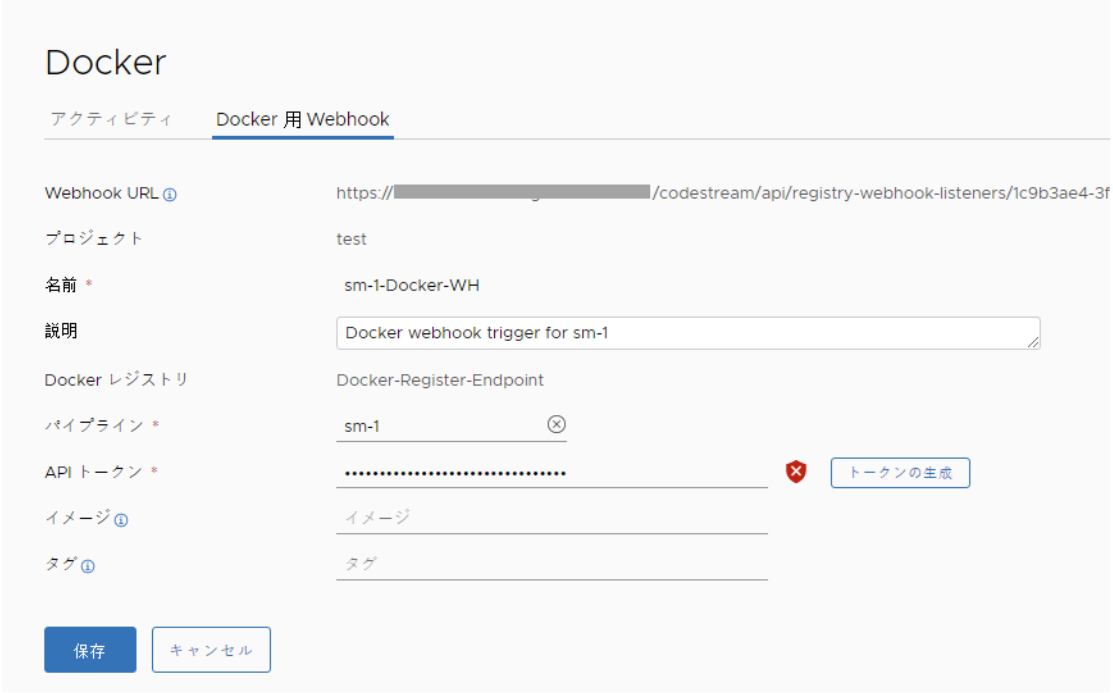
- 後で使用するためにトークンを変数として保持するには、[変数の作成] をクリックし、変数の名前を入力して [保存] をクリックします。
- 後で使用するためにトークンをテキスト値で保持するには、[コピー] をクリックし、トークンをテキスト ファイルに貼り付けてローカルに保存します。

後で使用するために、変数の作成とテキスト ファイルによるトークンの保存の両方を選択することもできます。

3 [閉じる] をクリックします。

h ビルド イメージを入力します。

i タグを入力します。



j [保存] をクリックします。

Docker Webhook が有効な場合、Webhook カードが表示されます。Docker Webhook をトリガーしてパイプラインを実行せずに、Docker Hub リポジトリにダミー プッシュする場合は、[無効にする] をクリックします。

4 CD パイプラインで、Kubernetes 展開タスクを構成します。

a Kubernetes タスクのプロパティで、開発 Kubernetes クラスタを選択します。

b [作成] アクションを選択します。

c ペイロード ソースの [ローカル定義] を選択します。

d 次に、ローカル YAML ファイルを選択します。

たとえば、Docker Hub は、このローカル YAML 定義を Webhook の URL にペイロードとして送信する場合があります。

```
{
"callback_url": "https://registry.hub.docker.com/u/svendowideit/testhook/hook/
2141b5bi5i5b02bec211i4eeih0242eg11000a/",
"push_data": {
"images": [
"27d47432a69bca5f2700e4dff7de0388ed65f9d3fb1ec645e2bc24c223dc1cc3",
"51a9c7c1f8bb2fa19bcd09789a34e63f35abb80044bc10196e304f6634cc582c",
"..."
],
"pushed_at": 1.417566161e+09,
"pusher": "trustedbuilder",
"tag": "latest"
},
"repository": {
"comment_count": 0,
"date_created": 1.417494799e+09,
"description": "",
"dockerfile": "#\n# BUILD\u0009\u0009docker build -t svendowideit/apt-cacher .\n#
RUN\u0009\u0009docker run -d -p 3142:3142 -name apt-cacher-run apt-cacher\n#\n# and
then you can run containers with:\n# \u0009\u0009docker run -t -i -rm -e http_proxy
http://192.168.1.2:3142/ debian
bash\n#\nFROM\u0009\u0009ubuntu\n\n\nVOLUME\u0009\u0009[\/var/cache/apt-cacher-ng\]
\nRUN\u0009\u0009apt-get update ; apt-get install -yq apt-cacher-ng\n\nEXPOSE
\u0009\u00093142\nCMD\u0009\u0009chmod 777 /var/cache/apt-cacher-ng ; /etc/init.d/apt-
cacher-ng start ; tail -f /var/log/apt-cacher-ng/*\n",
"full_description": "Docker Hub based automated build from a GitHub repo",
"is_official": false,
"is_private": true,
"is_trusted": true,
"name": "testhook",
"namespace": "svendowideit",
"owner": "svendowideit",
"repo_name": "svendowideit/testhook",
"repo_url": "https://registry.hub.docker.com/u/svendowideit/testhook/",
"star_count": 0,
"status": "Active"
}
}
```

Docker Hub で Webhook を作成する API は、次のフォームを使用します。`https://cloud.docker.com/v2/repositories/%3CUSERNAME%3E/%3CREPOSITORY%3E/webhook_pipeline/`

JSON コードの本文は次のようになります。

```
{
"name": "demo_webhook",
```

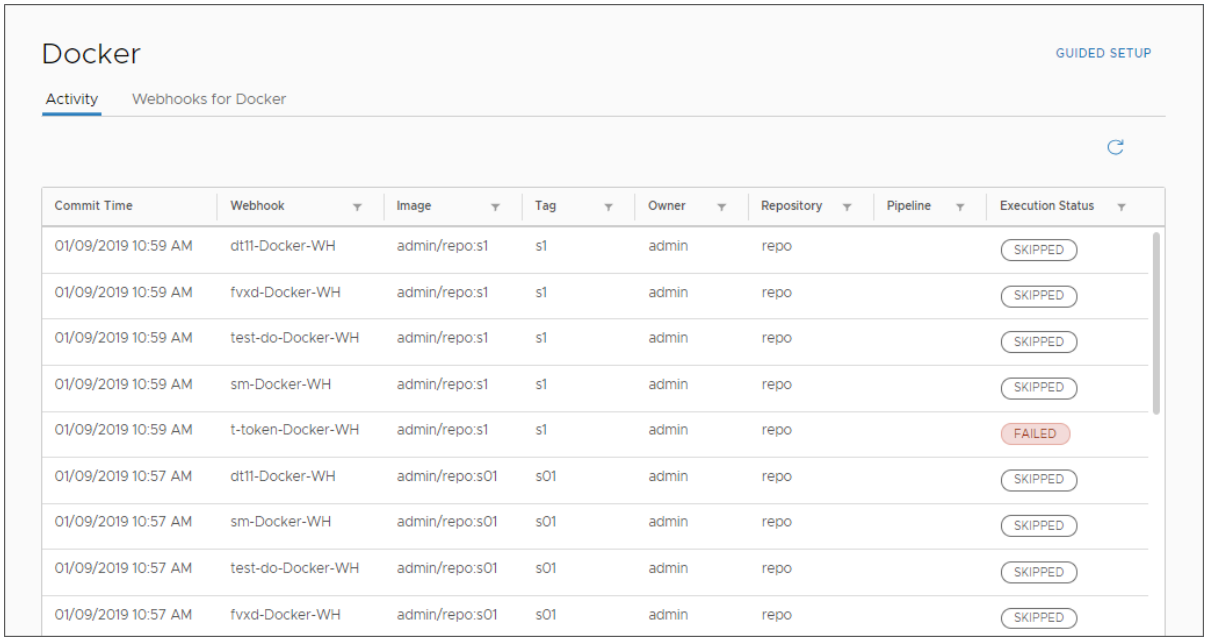
```
"webhooks": [
{
"name": "demo_webhook",
"hook_url": "http://www.google.com"
}
]
}
```

Docker Hub サーバからイベントを受信するために、 Code Stream で作成する Docker Webhook の認証方式では、Webhook のランダムな文字列トークンを使用して、許可された認証メカニズムを使用します。セキュアなトークンに基づいてイベントをフィルタリングします。これは、`hook_url` に追加できます。

Code Stream は、設定されたセキュアなトークンを使用することにより、Docker Hub サーバからのすべての要求を検証できます。例：`hook_url = IP:Port/pipelines/api/docker-hub-webhooks?secureToken = ""`

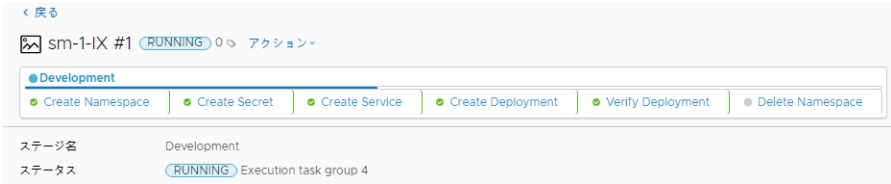
5 Docker Hub リポジトリに Docker アーティファクトを作成します。または、既存のアーティファクトを更新します。

6 更新がトリガされたことを確認して Docker Webhook アクティビティを表示するには、[トリガ] - [Docker] - [アクティビティ] の順にクリックします。



7 [実行] をクリックし、実行中のパイプラインを監視します。

8 実行中のステージをクリックして、パイプラインの実行中にタスクを表示します。



結果

完了です。CD パイプラインを継続的に実行するように、Docker トリガを設定します。これで、パイプラインは新しい Docker アーティファクトや更新された Docker アーティファクトを Docker Hub リポジトリにアップロードできるようになりました。

次のステップ

新規または更新されたアーティファクトが開発 Kubernetes クラスタの Docker ホストに展開されていることを確認します。

Code Stream で Git トリガを使用してパイプラインを実行する方法

Code Stream 管理者または開発者は、Git トリガを使用して Code Stream と Git ライフサイクルを統合できます。GitHub、GitLab、または Bitbucket Enterprise でコードを変更すると、対応するイベントが Webhook を使用して Code Stream と通信し、パイプラインをトリガします。Webhook は、同じネットワークから Cloud Assembly とエンタープライズ バージョンの両方にアクセスできる場合に、GitLab、GitHub、Bitbucket のオンプレミス エンタープライズ バージョンと連携します。

Code Stream で Git 用の Webhook を追加すると、GitHub、GitLab、または Bitbucket リポジトリにも Webhook が作成されます。後で Webhook を更新または削除すると、そのアクションによって GitHub、GitLab、または Bitbucket の Webhook も更新または削除されます。

Webhook 定義には、監視対象のリポジトリのブランチ上にある Git エンドポイントを含める必要があります。Webhook を作成するために、 Code Stream は Git エンドポイントを使用します。エンドポイントがない場合には、Webhook を追加するときに作成できます。この例では、GitHub に Git エンドポイントが事前定義されていることを前提としています。

注： Webhook を作成するには、Git エンドポイントは認証にプライベート トークンを使用する必要があります。パスワードは使用できません。

Webhook の構成画面で同じ Git エンドポイントを使用して、ブランチ名に別の値を指定することにより、複数のブランチに対して複数の Webhook を作成できます。同じ Git リポジトリ内の別のブランチに別の Webhook を作成する際に、複数のブランチに対して Git エンドポイントのクローン作成を複数回行う必要はありません。代わりに、Webhook 内でブランチ名を指定すれば、Git エンドポイントを再利用できるようになります。Git Webhook のブランチがエンドポイントのブランチと同じである場合は、Git Webhook 画面でブランチ名を入力する必要はありません。

この例では、Git トリガを GitHub リポジトリと併用する方法を示しますが、その前提条件には、別の Git サーバ タイプが使用されている場合に必要な準備が含まれています。

前提条件

- Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。
- 監視対象の GitHub ブランチに Git エンドポイントがあることを確認します。 Code Stream と Git の連携方法を参照してください。
- Git リポジトリに Webhook を作成する権限があることを確認します。
- GitLab で webhook を設定する場合、GitLab Enterprise のデフォルトのネットワーク設定を変更して、送信要求を有効にし、ローカルの Webhook を作成できるようにします。

  注： この変更は、GitLab Enterprise でのみ必要です。これらの設定は、GitHub または Bitbucket には適用されません。

  a GitLab Enterprise インスタンスに管理者としてログインします。

  b `http://{gitlab-server}/admin/application_settings/network` などの URL を使用して、ネットワーク設定に移動します。

  c [送信要求] を展開し、次をクリックします。

    - Webhook および Web サービスからローカル ネットワークへの要求を許可する。
    - システム フックからローカル ネットワークへの要求を許可する。
- トリガするパイプラインについて、パイプラインの実行時に Git パラメータを挿入するための入力プロパティが設定されていることを確認します。

入力パラメータについては、タスクの手動追加を行う前の Code Stream での CICD ネイティブ ビルドの計画を参照してください。

手順

1 Code Stream で、[トリガ] - [Git] の順にクリックします。

2 [Git の Webhook] タブをクリックし、[新規の Git 用 Webhook] をクリックします。

a プロジェクトを選択します。

b Webhook のわかりやすい名前と説明を入力します。

c 監視対象のブランチ用に構成された Git エンドポイントを選択します。

Webhook を作成すると、Webhook の定義には現在のエンドポイントの詳細が含まれます。

- エンドポイントで Git タイプ、Git サーバ タイプ、または Git リポジトリの URL を後で変更すると、Webhook は元のエンドポイントの詳細を使用して Git リポジトリにアクセスするため、パイプラインをトリガーできなくなります。Webhook を削除して、エンドポイントを使用して再度作成する必要があります。
- エンドポイントで認証タイプ、ユーザー名、またはプライベート トークンを後で変更すると、Webhook は引き続き機能します。
- BitBucket リポジトリを使用している場合、リポジトリの URL は `https://api.bitbucket.org/{user}/{repo name}` または `http(s)://{bitbucket-enterprise-server}/rest/api/1.0/users/{username}/repos/{repo name}` の形式にする必要があります。

注： ベーシック認証にパスワードを使用する Git エンドポイントを使用して Webhook を作成した場合は、その Webhook を削除し、認証にプライベート トークンを使用する Git エンドポイントを使用して再定義する必要があります。

Code Stream と Git の連携方法を参照してください。

d （オプション） Webhook で監視するブランチを入力します。

ブランチが未指定のままにされた場合、Webhook は Git エンドポイント用に構成されたブランチを監視します。

e （オプション） Webhook のシークレット トークンを生成します。

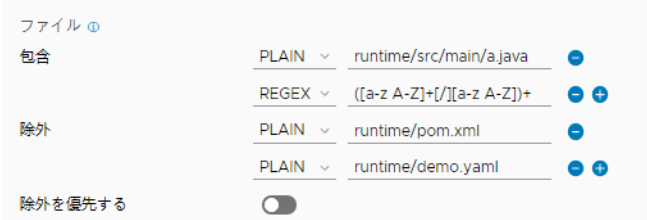
シークレット トークンを使用する場合は、 Code Stream によって Webhook のランダムな文字列トークンが生成されます。Webhook は、Git イベント データを受信すると、シークレット トークンとともにデータを送信します。 Code Stream は、この情報を使用して、設定済みの GitHub インスタンス、リポジトリ、ブランチなど想定されるソースが呼び出し元であるかどうかを判断します。シークレット トークンによって、新たなセキュリティ レイヤーが加わります。Git イベント データが適切なソースから取得されていることを確認するためのものです。

f （オプション） ファイルの包含または除外をトリガの条件として指定します。

- ファイルの包含。コミット内のファイルのいずれかが包含パスまたは正規表現に指定されているファイルと一致すると、コミットでパイプラインがトリガされます。正規表現を指定した場合、 Code Stream は変更セット内のファイル名がその正規表現の式に一致するときにのみパイプラインをトリガします。正規表現フィルタは、1 つのリポジトリに複数のパイプラインのトリガを設定する場合に便利です。
- ファイルの除外。コミット内のすべてのファイルが除外パスまたは正規表現に指定されているファイルと一致すると、パイプラインはトリガされません。
- 除外を優先する。[除外を優先する] を有効にすると、コミット内のファイルのいずれかが除外パスまたは正規表現に指定されているファイルと一致しても、パイプラインがトリガされなくなります。デフォルトの設定はオフです。

ファイルの包含とファイルの除外の両方が条件として満たされる場合、パイプラインはトリガされません。

次の例では、ファイルの包含も除外もトリガの条件にしています。



- ファイル包含の場合、`runtime/src/main/a.java` や他の Java ファイルに変更を加えてコミットすると、イベント設定に設定されたパイプラインがトリガされます。
- ファイル除外の場合、両方のファイルに変更を加えただけでコミットしても、イベント設定に設定されたパイプラインはトリガされません。

g Git イベントの場合、[プッシュ] または [プル] 申請を選択します。

h API トークンを入力します。

CSP API トークンは、 Code Stream との外部 API 接続を認証します。API トークンを取得するには、次の手順を実行します。

1 [トークンの生成] をクリックします。

2 ユーザー名とパスワードに関連付けられているメール アドレスを入力し、[生成] をクリックします。

生成するトークンは 6 か月有効です。これは更新トークンとも呼ばれます。

- 後で使用するためにトークンを変数として保持するには、[変数の作成] をクリックし、変数の名前を入力して [保存] をクリックします。
- 後で使用するためにトークンをテキスト値で保持するには、[コピー] をクリックし、トークンをテキスト ファイルに貼り付けてローカルに保存します。

後で使用するために、変数の作成とテキスト ファイルによるトークンの保存の両方を選択することもできます。

3 [閉じる] をクリックします。

i Webhook によってトリガされるパイプラインを選択します。

カスタムで追加された入力パラメータがパイプラインに含まれている場合、[入力パラメータ] リストにはパラメータと値が表示されます。イベントをトリガするパイプラインに渡される入力パラメータの値を入力できます。値は、空白のままにすることもできます。デフォルト値が定義されている場合は、デフォルト値を使用することもできます。

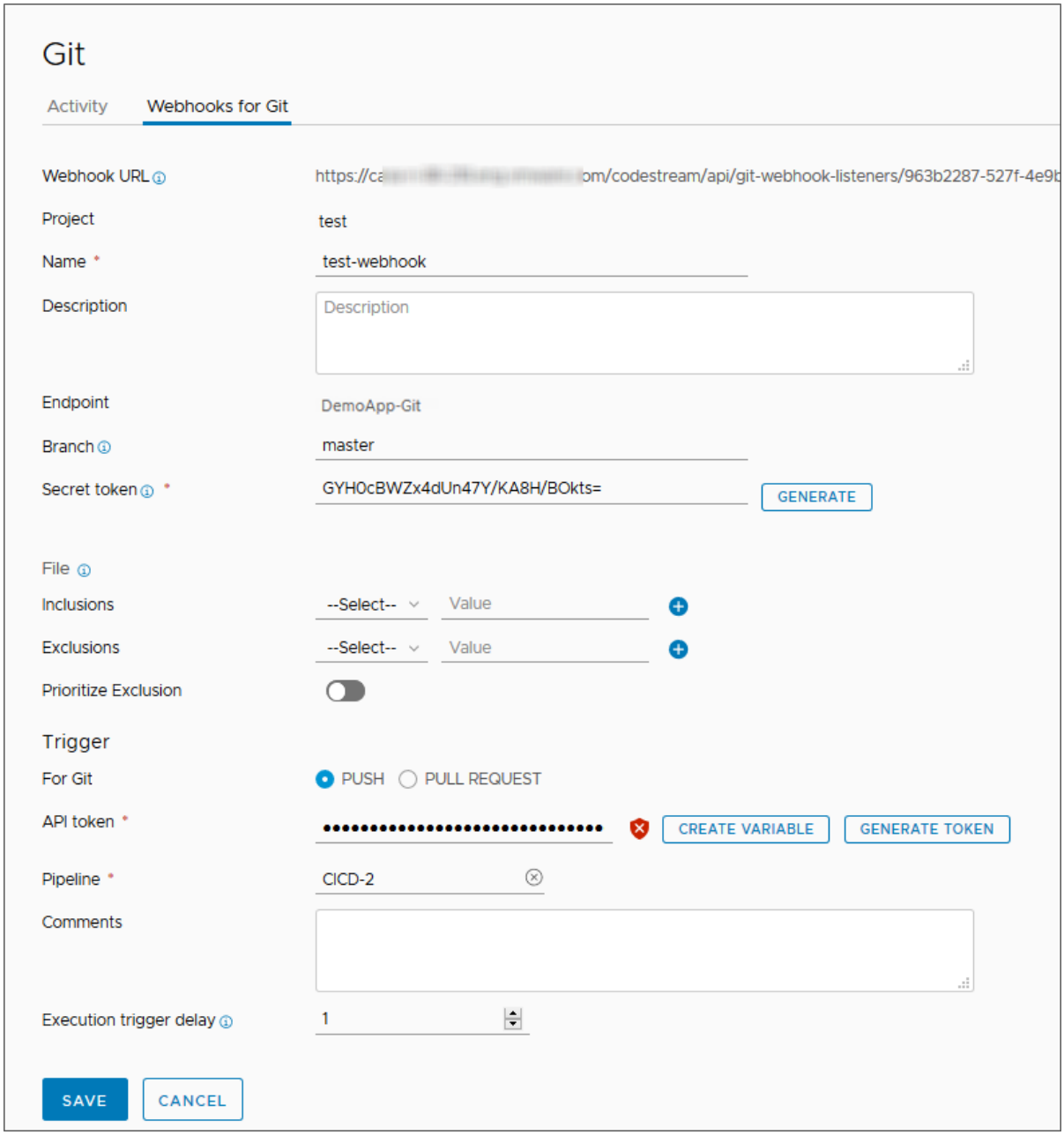
Git トリガ用の自動挿入の入力パラメータについては、前提条件を参照してください。

j [作成] をクリックします。

Webhook が新しいカードとして表示されます。

3 Webhook カードをクリックします。

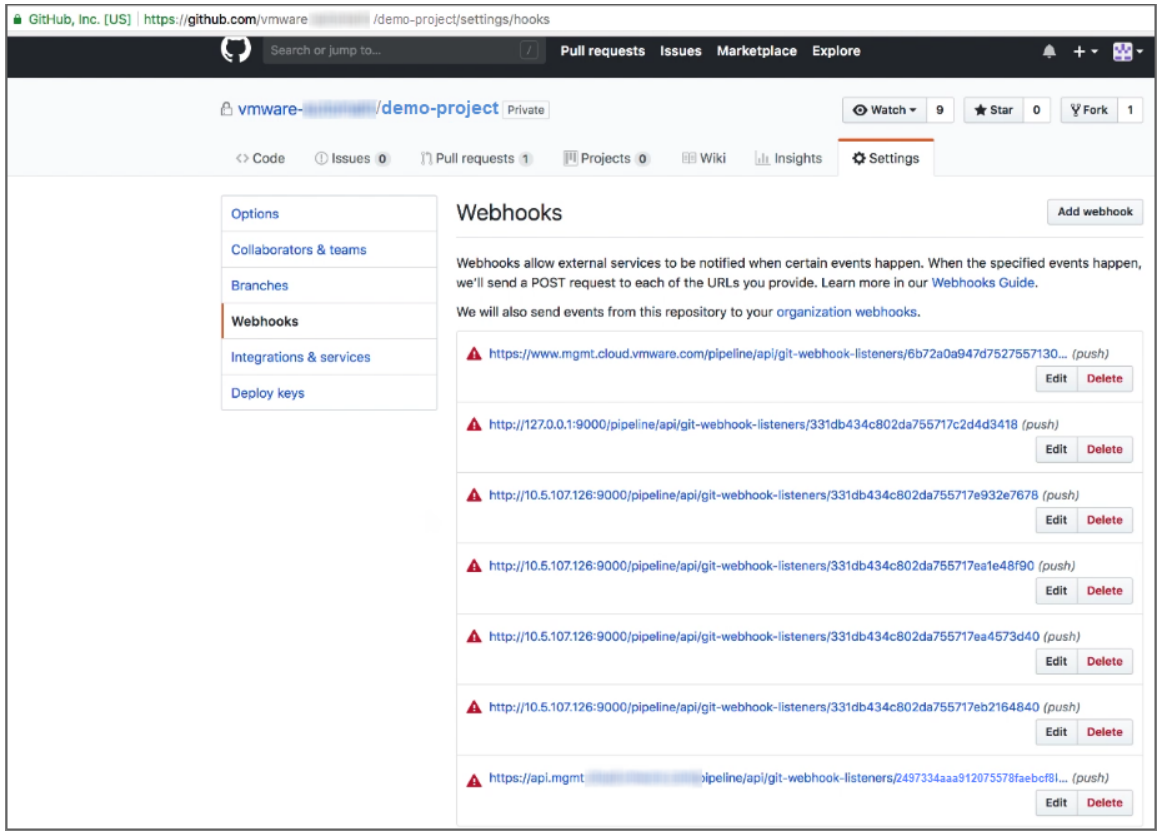
Webhook データ フォームを再び表示すると、フォームの上部に Webhook の URL が追加されています。Git Webhook は、Webhook URL を経由して GitHub リポジトリに接続します。


Git
Activity
Webhooks for Git
Webhook URL
https://ca om/codestream/api/git-webhook-listeners/963b2287-527f-4e9b
Project
test
Name *
test-webhook
Description
Description
Endpoint
DemoApp-Git
Branch
master
Secret token *
GYH0cBWZx4dUn47Y/KA8H/BOkts=
GENERATE
File
Inclusions
--Select--
Value
Exclusions
--Select--
Value
Prioritize Exclusion
Trigger
For Git
PUSH
PULL REQUEST
API token *
CREATE VARIABLE
GENERATE TOKEN
Pipeline *
CICD-2
Comments
Execution trigger delay
1
SAVE
CANCEL

4 新しいブラウザ ウィンドウで、Webhook を使用して接続されている GitHub リポジトリを開きます。

a Code Stream で追加した Webhook を表示するには、[設定] タブをクリックし、[Webhooks] を選択します。

Webhook リストの下部に、同じ Webhook URL が表示されます。



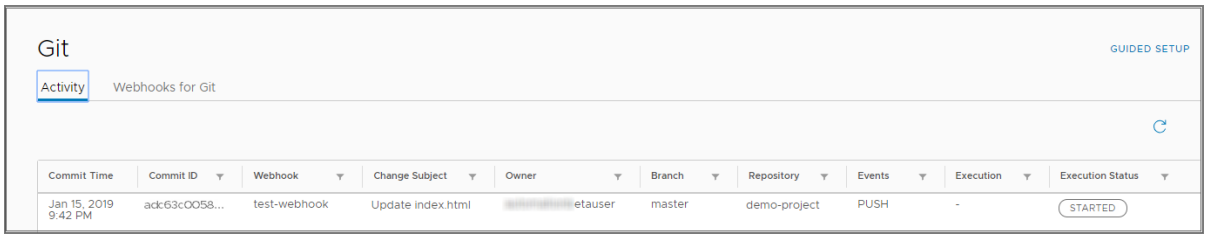
b コードを変更するには、[コード] タブをクリックし、ブランチでファイルを選択します。ファイルを編集した後、変更をコミットします。

c Webhook URL が機能していることを確認するには、[設定] タブをクリックし、[Webhooks] を再度選択します。

Webhook リストの下部に表示される Webhook URL の横に、緑色のチェックマークが示されます。

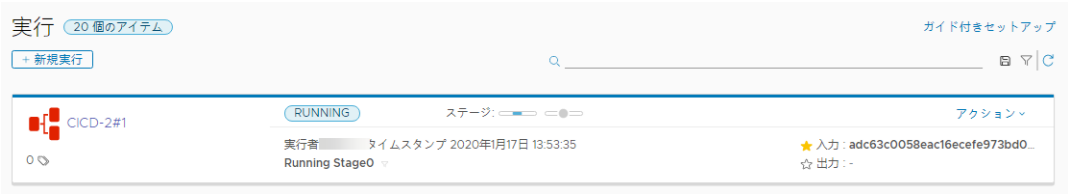
5 Code Stream に戻ると、Git Webhook のアクティビティが表示されます。[トリガ] - [Git] - [アクティビティ] の順にクリックします。

実行ステータスで、パイプラインの実行が開始されたことを確認します。



6 [実行] をクリックし、パイプラインを実行中に追跡します。

パイプラインの実行を確認するには、[更新] を押します。



結果

完了です。Git のトリガが正常に使用されました。

Code Stream で Gerrit トリガを使用してパイプラインを実行する方法

Code Stream 管理者または開発者は、Gerrit トリガを使用して Code Stream と Gerrit コード レビュー ライフサイクルを統合できます。Gerrit プロジェクトでパッチ セットの作成、ドラフトの発行、コード変更のマージを行ったり、変更を Git ブランチに直接プッシュすると、イベントがパイプラインの実行をトリガします。

Gerrit トリガを追加する場合は、Gerrit リスナーと、Gerrit サーバ上の Gerrit プロジェクトを選択し、Gerrit イベントを構成します。この例では、最初に Gerrit リスナーを構成し、次に 3 つの異なるパイプラインの 2 つのイベントを使用して、Gerrit トリガでこのリスナーを使用します。

前提条件

- Code Stream で、プロジェクトのメンバーであることを確認します。メンバーでない場合は、プロジェクトにメンバーとして追加するように Code Stream 管理者に依頼します。 Code Stream でプロジェクトを追加する方法を参照してください。
- Code Stream で Gerrit エンドポイントが設定されていることを確認します。 Code Stream を Gerrit と統合する方法を参照してください。
- Gerrit リリース バージョンを把握していることを確認します。
- パイプラインをトリガするために、パイプラインの入力プロパティが [Gerrit] として設定されていることを確認します。これにより、パイプラインは、実行中に Gerrit パラメータを入力として受け取ることができます。

入力パラメータについては、タスクの手動追加を行う前の Code Stream での CICD ネイティブ ビルドの計画を参照してください。

手順

1 Code Stream で、[トリガ] - [Gerrit] の順にクリックします。

2 （オプション）[リスナー] タブをクリックし、[新規リスナー] をクリックします。

注： Gerrit トリガに使用する予定の Gerrit リスナーがすでに定義されている場合は、この手順をスキップします。

a プロジェクトを選択します。

b Gerrit リスナーの名前を入力します。

c Gerrit エンドポイントを選択します。

d API トークンを入力します。

CSP API トークンは、 Code Stream との外部 API 接続を認証します。API トークンを取得するには、次の手順を実行します。

1 [トークンの生成] をクリックします。

2 ユーザー名とパスワードに関連付けられているメール アドレスを入力し、[生成] をクリックします。

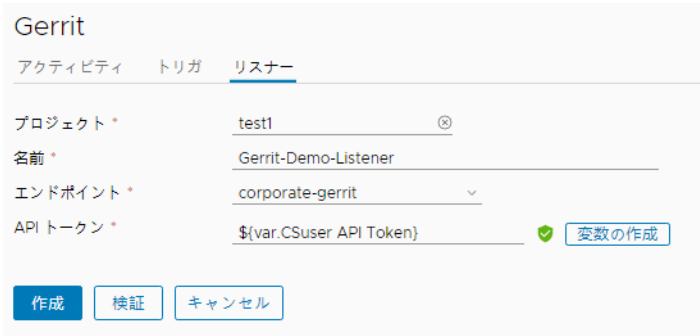
生成するトークンは 6 か月有効です。これは更新トークンとも呼ばれます。

- 後で使用するためにトークンを変数として保持するには、[変数の作成] をクリックし、変数の名前を入力して [保存] をクリックします。
- 後で使用するためにトークンをテキスト値で保持するには、[コピー] をクリックし、トークンをテキスト ファイルに貼り付けてローカルに保存します。

後で使用するために、変数の作成とテキスト ファイルによるトークンの保存の両方を選択することもできます。

3 [閉じる] をクリックします。

変数を作成した場合は、ドル記号によるバインドを使用して入力した変数名が API トークンに表示されます。トークンをコピーした場合、API トークンにはマスクされたトークンが表示されます。



e トークンとエンドポイントの詳細を検証するには、[検証] をクリックします。

トークンは、あと 90 日で有効期限が切れます。

f [作成] をクリックします。

g リスナー カードで [接続] をクリックします。

リスナーが Gerrit サーバ上のすべてのアクティビティの監視を開始し、そのサーバ上で有効になっているすべてのトリガを待機します。そのサーバでのトリガの待機を停止するには、トリガを無効にします。

注： リスナーに接続されている Gerrit エンドポイントを更新するには、エンドポイントを更新する前にリスナーを切断する必要があります。

- [構成] - [トリガ] - [Gerrit] の順にクリックします。
- [リスナー] タブをクリックします。
- 更新するエンドポイントに接続されているリスナーで、[切断] をクリックします。

3 [トリガ] タブをクリックしてから、[新規トリガ] をクリックします。

4 Gerrit サーバ上のプロジェクトを選択します。

5 名前を入力します。

Gerrit トリガ名は一意である必要があります。

6 設定された Gerrit リスナーを選択します。

Code Stream は、Gerrit リスナーを使用して、サーバで使用可能な Gerrit プロジェクトのリストを表示します。

7 Gerrit サーバ上のプロジェクトを選択します。

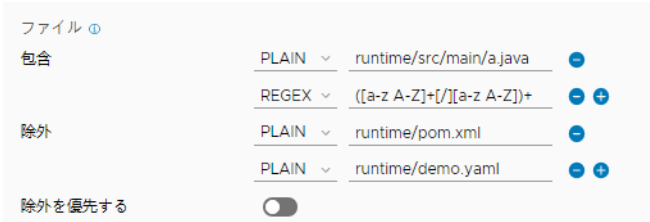
8 Gerrit リスナーが監視するリポジトリのブランチを入力します。

9 （オプション） ファイルの包含または除外をトリガの条件として指定します。

- パイプラインをトリガするファイルの包含を指定します。コミット内のファイルのいずれかが包含パスまたは正規表現に指定されているファイルと一致すると、パイプラインがトリガされます。正規表現を指定した場合、Code Stream は変更セット内のファイル名がその正規表現の式に一致するパイプラインのみをトリガします。正規表現フィルタは、1 つのリポジトリに複数のパイプラインのトリガを設定する場合に便利です。
- ファイルの除外を指定して、パイプラインがトリガされないようにします。コミット内のすべてのファイルが除外パスまたは正規表現に指定されているファイルと一致すると、パイプラインはトリガされません。
- [除外を優先する] をオンに切り替えると、パイプラインがトリガされなくなります。コミットのファイルのいずれかが除外パスまたは正規表現で指定されたファイルと一致する場合でも、パイプラインはトリガされません。[除外を優先する] のデフォルト設定は、オフとなっています。

ファイルの包含とファイルの除外の両方が条件として満たされる場合、パイプラインはトリガされません。

次の例では、ファイルの包含も除外もトリガの条件にしています。



- ファイルの包含の場合、`runtime/src/main/a.java` や他の Java ファイルに変更を加えてコミットすると、イベント設定で指定されたパイプラインがトリガされます。
- ファイルの除外の場合、両方のファイルに変更を加えてコミットしても、イベント設定で指定されたパイプラインはトリガされません。

10 [新しい構成] をクリックします。

a Gerrit イベントの場合、[パッチセットの作成]、[ドラフトの公開]、または [変更のマージ] を選択します。または、Git への直接プッシュによって Gerrit をバイパスする場合は、[直接 Git プッシュ] を選択します。

注： Gerrit リリース バージョン 2.15 では、ドラフトの変更とドラフトの変更セットはサポートされなくなりました。したがって、Gerrit リリース バージョン 2.15 以降を実行している場合、[ドラフトの公開] はサポートされているイベントではありません。

b トリガするパイプラインを選択します。

カスタムで追加された入力パラメータがパイプラインに含まれている場合、[入力パラメータ] リストにはパラメータと値が表示されます。イベントをトリガするパイプラインに渡される入力パラメータの値を入力できます。値を空白のままにすることも、デフォルト値を使用することもできます。

注： デフォルト値が定義されている場合：

- 入力パラメータに値を入力すると、パイプライン モデルで定義されているデフォルト値がオーバーライドされます。
- パイプライン モデルのパラメータ値が変更されても、トリガ設定のデフォルト値は変更されません。

Gerrit トリガ用の自動挿入の入力パラメータについては、前提条件を参照してください。

c [パッチセットの作成]、[ドラフトの公開]、[変更のマージ] では、デフォルトで一部のアクションがラベルとともに表示されます。ラベルの変更や、コメントの追加ができます。その後、パイプラインが実行されると、パイプラインの [実行された操作] として、ラベルまたはコメントが [アクティビティ] タブに表示されます。

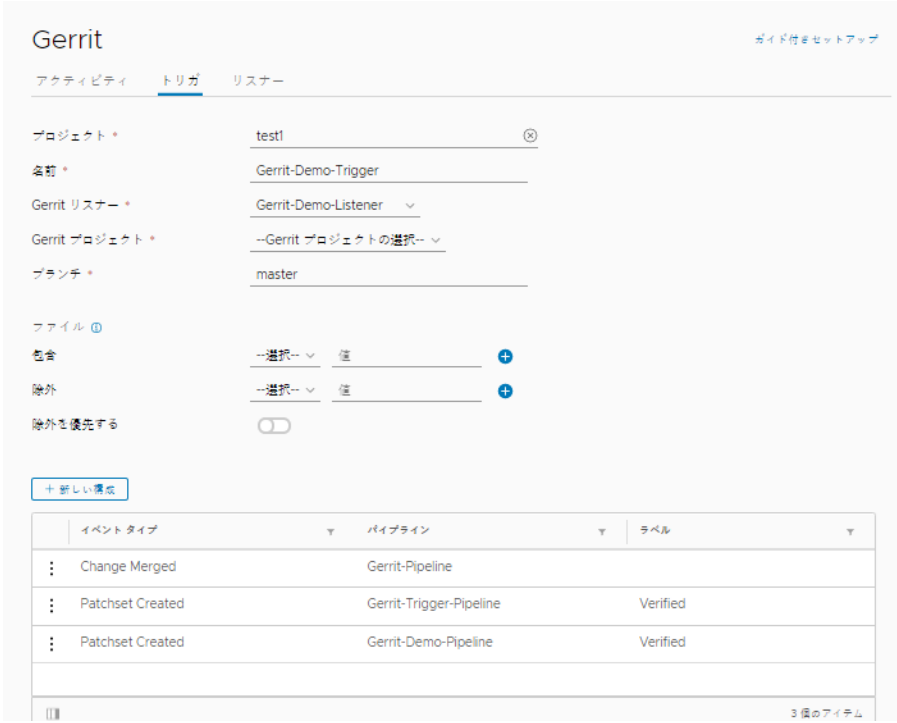
Gerrit イベント構成では、成功コメントまたは失敗コメントの変数を使用してコメントを入力できます。たとえば、`${var.success}` と `${var.failure}` を使用します。

d [保存] をクリックします。

複数のパイプラインに複数のトリガ イベントを追加するには、[新しい構成] を再度クリックします。

次の例では、3 つのパイプラインのイベントを表示できます。

- Gerrit プロジェクトで [変更のマージ] イベントが発生した場合、[Gerrit-Pipeline] という名前のパイプラインがトリガされます。
- Gerrit プロジェクトで [パッチセットの作成] イベントが発生した場合、[Gerrit-Trigger-Pipeline] および [Gerrit-Demo-Pipeline] という名前のパイプラインがトリガされます。



11 [作成] をクリックします。

Gerrit トリガは、[トリガ] タブに新規カードとして表示され、デフォルトでは [無効] に設定されています。

12 トリガ カードで、[有効] をクリックします。

トリガを有効にすると、Gerrit リスナーを使用できるようになり、Gerrit プロジェクトのブランチで発生するイベントの監視が開始されます。

ファイルの包含条件またはファイルの除外条件は同じであるが、トリガの作成時に含めたものとは異なるリポジトリを持つトリガを作成するには、トリガ カードで [アクション] - [クローン作成] の順にクリックします。次に、クローン作成されたトリガで [開く] をクリックし、パラメータを変更します。

結果

完了です。3 つの異なるパイプラインの 2 つのイベントに Gerrit トリガが正常に設定されました。

次のステップ

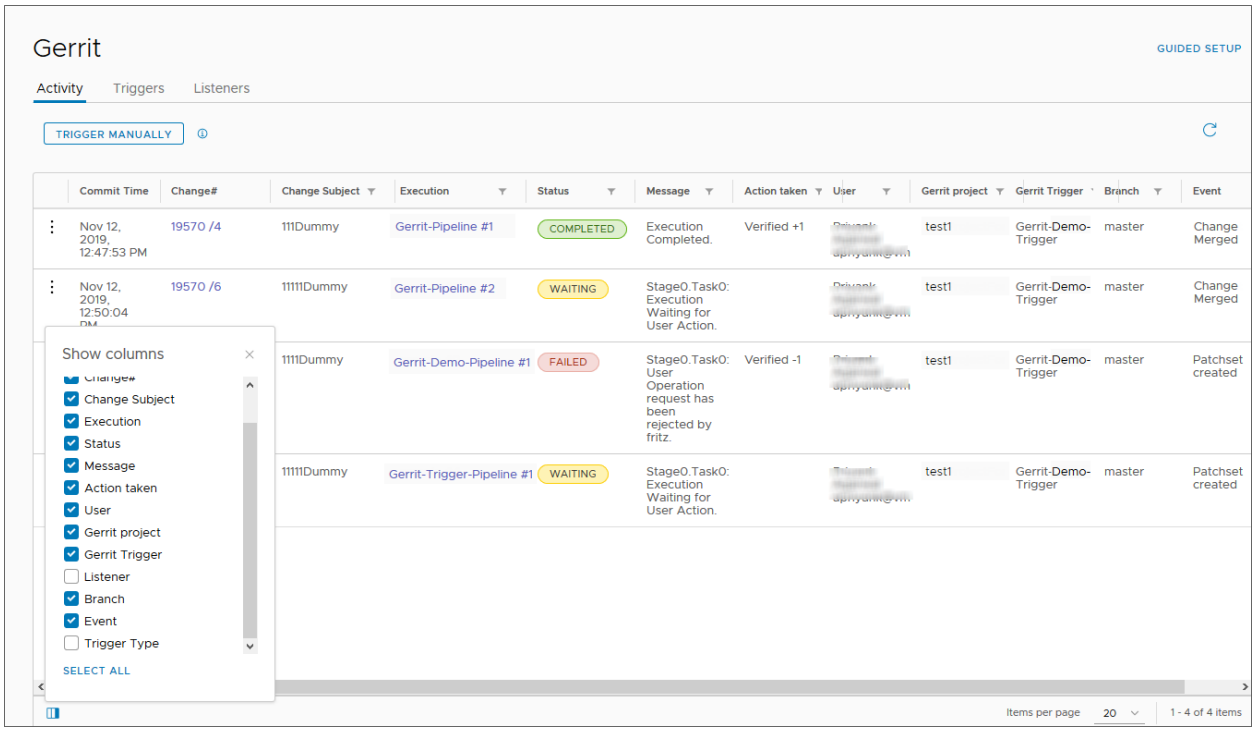
Gerrit プロジェクトのコード変更をコミットした後、[アクティビティ] タブで Code Stream の Gerrit イベントを確認します。アクティビティのリストに、トリガ設定のすべてのパイプライン実行に対応するエントリが含まれていることを確認します。

イベントが発生した場合、特定のタイプのイベントに関連する Gerrit トリガ内のパイプラインのみが実行されます。この例では、パッチ セットが作成された場合、[Gerrit-Trigger-Pipeline] と [Gerrit-Demo-Pipeline] のみが実行されます。

[アクティビティ] タブの列には、各 Gerrit トリガ イベントについての情報が示されています。表の下に表示される列のアイコンをクリックすると、表示される列を選択できます。

- トリガが直接の Git プッシュであった場合、[サブジェクトの変更] 列と [実行] 列は空になります。

- [Gerrit トリガー] 列には、イベントを作成したトリガが表示されます。
- [リスナー] 列はデフォルトでオフになっています。選択すると、イベントを受信した Gerrit リスナーが列に表示されます。複数のトリガに関連付けられている 1 つのリスナーを表示できます。
- [トリガー タイプ] 列はデフォルトでオフになっています。選択すると、トリガのタイプが [自動] または [手動] として列に表示されます。
- 他の列には、[コミット時間]、[変更 #]、[ステータス]、[メッセージ]、[実行された操作]、[ユーザー]、[Gerrit プロジェクト]、[ブランチ]、[イベント] が含まれます。



完了した実行や失敗したパイプライン実行のアクティビティを制御するには、[アクティビティ] 画面の任意のエントリの左側にある 3 つのドットをクリックします。

- パイプライン モデルの誤りや別の問題が原因でパイプラインを実行できない場合は、誤りを修正し、[再実行] を選択して、パイプラインを再実行します。
- ネットワーク接続の問題や別の問題が原因でパイプラインを実行できない場合は、[再開] を選択して、同じパイプラインの実行を再開し、ランタイムを保存します。
- [実行の表示] を使用して、パイプライン実行ビューを開きます。パイプラインを実行して結果を確認する方法を参照してください。
- [削除] を使用して、[アクティビティ] 画面からエントリを削除します。

Gerrit イベントがパイプラインのトリガに失敗した場合は、[手動でトリガー] をクリックして、Gerrit トリガを選択します。次に、変更 ID を入力して、[実行] をクリックします。

Code Stream でのパイプラインの監視

8

Code Stream の管理者または開発者には、 Code Stream のパイプラインのパフォーマンスに関する判断材料が必要です。開発からテストを経て本番環境に至るまでパイプラインがどの程度効果的にコードをリリースしているかを把握しておく必要があります。

判断材料を得るには、 Code Stream ダッシュボードを使用してパイプライン実行のトレンドと結果を監視します。デフォルトのパイプライン ダッシュボードを使用すると、1 つのパイプラインを監視できるほか、複数のパイプラインを監視するためのカスタム ダッシュボードを作成できます。

- パイプライン メトリックには、平均時間などの統計情報が含まれます。これは、パイプライン ダッシュボードで確認できます。
- 複数のパイプライン全体のメトリックを表示するには、カスタム ダッシュボードを使用します。

この章には、次のトピックが含まれています。

- Code Stream でパイプライン ダッシュボードに表示される内容
- Code Stream でカスタム ダッシュボードを使用してパイプラインのキー パフォーマンス インジケータを追跡する方法

Code Stream でパイプライン ダッシュボードに表示される内容

パイプライン ダッシュボードには、特定のパイプラインの実行結果が表示されます。トレンド、発生数の多い障害、正常に完了した変更などです。パイプラインを作成すると、 Code Stream がパイプライン ダッシュボードを作成します。

このダッシュボードには、パイプラインの実行結果を表示するウィジェットが含まれています。

パイプライン実行ステータス数ウィジェット

一定期間内に実行されたパイプラインの総数を、完了、失敗、キャンセル済みのステータスでグループ化して表示できます。より長い期間または短い期間でのパイプラインの実行ステータスの変化を確認するには、表示期間を変更します。

パイプライン実行の統計ウィジェット

パイプライン実行の統計には、一定期間内でのパイプラインのリカバリ、デリバリ、または失敗までの平均時間が含まれます。

次の各状態は、すべてのパイプライン実行に適用されます。

- 完了
- 失敗
- 待機中
- 実行中
- キャンセル済み
- キューに格納済み
- 未開始
- ロールバック中
- ロールバックが完了しました
- ロールバックに失敗しました
- 一時停止中

表 8-1. 平均時間の測定

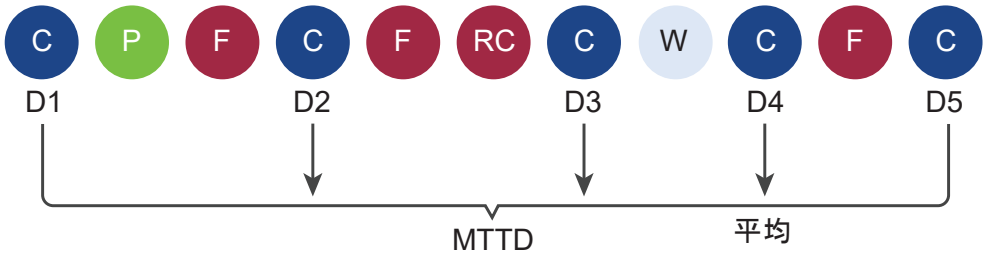
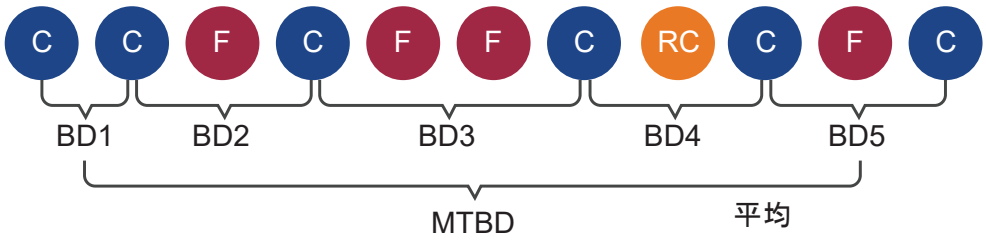
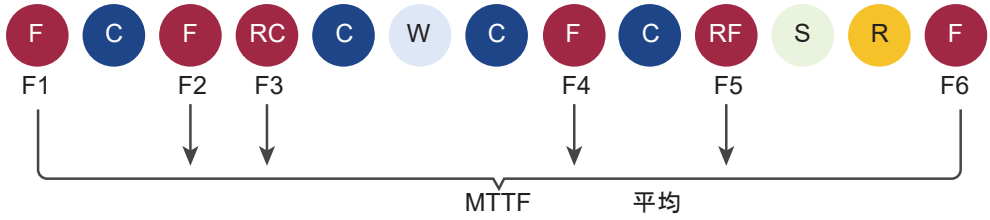
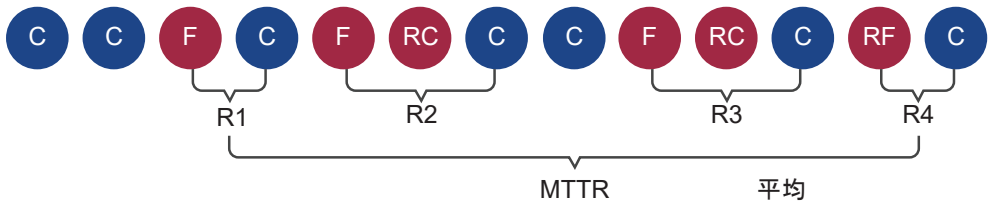
測定対象...	詳しい内容...
CI の平均	継続的な統合フェーズでかかった平均時間（CI タスク タイプで、時間単位で測定）。
平均デリバリ時間 (MTTD)	一定期間内に完了したすべての実行の平均時間。D1、D2 などは、完了した各実行のデリバリにかかった時間です。 
デリバリ間の平均時間 (MTBD)	一定期間内の成功したデリバリ間の平均経過時間。2 つの連続する完了した実行の間の経過時間は、成功したデリバリ間の時間です（BD1、BD2 など）。MTBD は、本番環境が更新される頻度を示します。 

表 8-1. 平均時間の測定 （続き）

測定対象...	詳しい内容...
平均失敗時間 (MTTF)	一定期間内に FAILED、ROLLBACK_COMPLETED、または ROLLBACK_FAILED 状態で終了した実行の平均実行時間。F1、F2 などは、実行が FAILURE、ROLLBACK_COMPLETED、または ROLLBACK FAILED で終了するまでの時間です。 
平均リカバリ時間 (MTTR)	一定期間内の失敗からリカバリまでの平均時間。失敗からリカバリまでの時間は、最終的なステータスが FAILED、ROLLBACK_COMPLETED、または ROLLBACK_FAILED である実行から、その直後に COMPLETED ステータスで成功した実行までの経過時間です。R1、R2 などは、FAILED または ROLLBACK_FAILED の各実行からリカバリまでの時間です。 

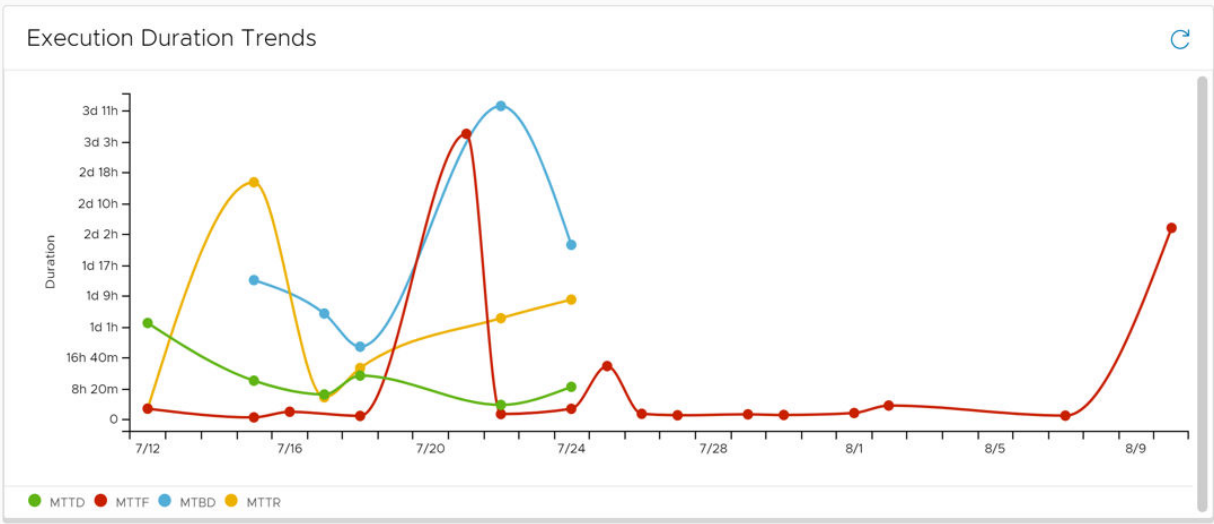
上位の失敗ステージ ウィジェットおよび上位の失敗タスク ウィジェット

2 つのウィジェットに、パイプライン内で失敗が上位のステージとタスクが表示されます。各測定値は、各パイプラインおよびプロジェクトの開発環境および開発後環境での失敗発生数と割合を示します（1 週間または 1 か月の平均）。上位の障害を表示して、リリース自動化プロセスの問題のトラブルシューティングを行います。

たとえば、過去 7 日間などの特定期間に失敗したタスクを表示するように設定して、その期間に失敗した上位のタスクを確認できます。環境またはパイプラインを変更してから、再びパイプラインを実行する場合、過去 14 日間などの長い期間にわたって失敗した上位のタスクを確認すると、変更された可能性があることがわかる場合があります。その場合、リリースの自動化プロセスの変更によって、パイプラインの実行の成功率が向上したことがわかります。

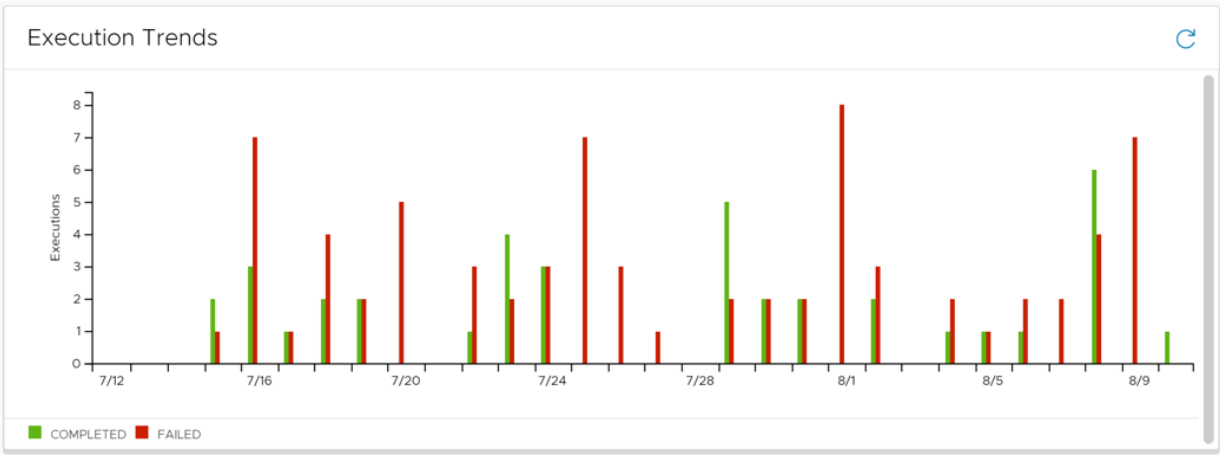
パイプライン実行時間のトレンド ウィジェット

パイプライン実行時間のトレンドには、一定期間内の MTTD、MTTF、MTBD、および MTTR が表示されます。



パイプライン実行のトレンド ウィジェット

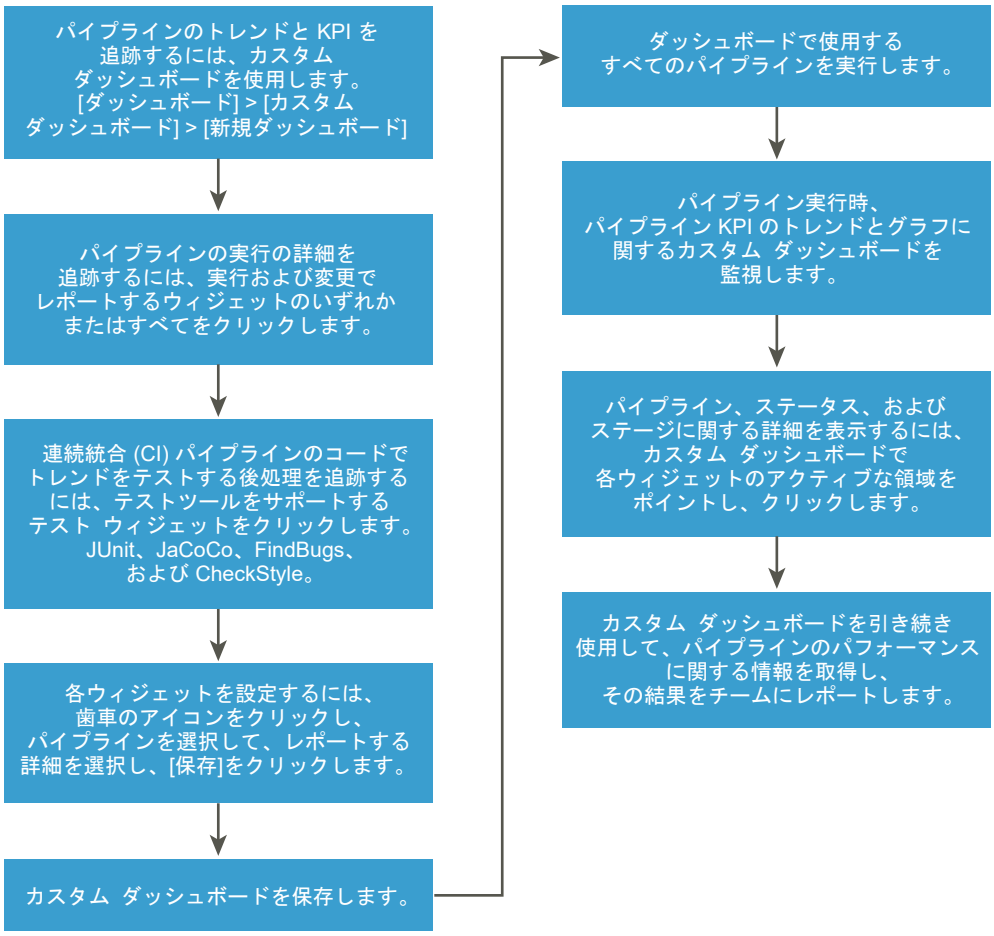
パイプライン実行のトレンドには、一定期間内のパイプラインの日別の総実行数がステータスごとにグループ化されて示されます。現在の日付を除いて、ほとんどの日次集計では、COMPLETED および FAILED の実行のみが表示されます。



Code Stream でカスタム ダッシュボードを使用してパイプラインのキー パフォーマンス インジケータを追跡する方法

Code Stream の管理者または開発者は、実行した 1 つ以上のパイプラインについて確認したい結果が表示されるように、カスタム ダッシュボードを作成できます。たとえば、複数のパイプラインから KPI とメトリックを収集し、プロジェクト全体のダッシュボードを作成できます。実行時に警告や障害がレポートされた場合は、ダッシュボードを使用して、その障害のトラブルシューティングを行うことができます。

カスタム ダッシュボードを使用してパイプラインのトレンドとキー パフォーマンス インジケータを追跡するには、ダッシュボードにウィジェットを追加し、パイプラインについてレポートするようにウィジェットを構成します。



前提条件

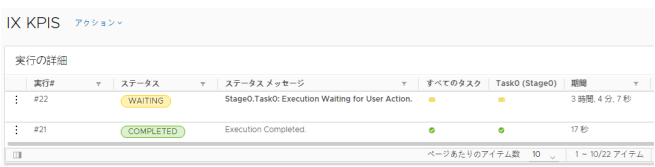
- 1 つ以上のパイプラインが配置されていることを確認します。ユーザー インターフェイスで、[パイプライン] をクリックします。
- 監視するパイプラインが正常に実行されたことを確認します。[実行] をクリックします。

手順

1 カスタム ダッシュボードを作成するには、[ダッシュボード] - [カスタム ダッシュボード] - [新規ダッシュボード] の順にクリックします。

2 パイプラインの特定のトレンドとキー パフォーマンス インジケータについて報告するようにダッシュボードをカスタマイズするには、ウィジェットをクリックします。

たとえば、パイプラインのステータス、ステージ、タスク、実行時間、および実行者の詳細を表示するには、[実行の詳細] ウィジェットをクリックします。また、後処理で JUnit、JaCoCo、FindBugs、Checkstyle の各ウィジェットを使用して、継続的インテグレーション (CI) パイプラインのトレンドを追跡できます。



3 追加する各ウィジェットを構成します。

a ウィジェットで、歯車アイコンをクリックします。

b パイプラインを選択し、使用可能なオプションを設定し、表示する列を選択します。

c ウィジェット設定を保存するには、[保存] をクリックします。

d カスタム ダッシュボードを保存するには、[保存] をクリックし、[閉じる] をクリックします。

4 パイプラインの詳細な情報を表示するには、ウィジェットのアクティブな領域をクリックします。

たとえば、[実行の詳細] ウィジェットで、[ステータス] 列のエントリをクリックすると、パイプラインの実行に関する詳細な情報が表示されます。[最近加えられた変更] ウィジェットで、パイプラインのステージとタスクのサマリを表示するには、アクティブなリンクをクリックします。

結果

完了です。パイプラインのトレンドと KPI を監視するカスタム ダッシュボードを作成しました。

次のステップ

引き続き Code Stream でパイプラインのパフォーマンスを監視します。その結果をマネージャおよびチームと共有して、アプリケーションをリリースするプロセスの強化を続けます。

Code Stream の詳細

9

Code Stream の管理者や開発者が、 Code Stream の詳細やユーザーのために実行できることを確認するための方法が多数用意されています。

パイプラインとその実行方法、エンドポイントの追加方法、プロジェクトの追加方法などの詳細については、このドキュメントを参照してください。

ロールによって提供される権限について理解します。制限付きリソースの使用方法と、パイプラインの承認を要求する方法について説明します。 Code Stream でユーザー アクセスと承認を管理する方法を参照してください。

パイプライン、実行、またはエンドポイントに特定のジョブまたはコンポーネントが配置されている場所を検出するには、検索の値を参照してください。

この章には、次のトピックが含まれています。

- Code Stream での検索について
- Code Stream 管理者および開発者向けのその他のリソース

Code Stream での検索について

特定のアイテムまたは他のコンポーネントが配置されている場所を検索するには検索を使用します。たとえば、有効または無効なパイプラインを検索することがあります。パイプラインは無効になっていると実行できないためです。

検索可能な内容

次の場所で検索できます。

- プロジェクト
- エンドポイント
- パイプライン
- 実行
- パイプライン ダッシュボード、カスタム ダッシュボード
- Gerrit トリガおよびサーバ
- Git Webhook
- Docker Webhook

次の列ベースのフィルタ検索を実行できます。

- ユーザー操作
- 変数
- Gerrit、Git、および Docker のトリガ アクティビティ

各トリガの [アクティビティ] 画面でグリッドベースのフィルタ検索を実行できます。

検索の機能

検索の条件は、表示されている画面によって異なります。画面ごとに異なる検索条件があります。

検索場所	検索に使用する基準
パイプライン ダッシュボード	プロジェクト、名前、説明、タグ、リンク
カスタム ダッシュボード	プロジェクト、名前、説明、リンク（ダッシュボードのアイテムの UUID）
実行	`<key>:<value>` 形式による名前、コメント、理由、タグ、インデックス、ステータス、プロジェクト、表示、実行者、実行者（自分）、リンク（実行の UUID）、入力パラメータ、出力パラメータ、またはステータス メッセージ
パイプライン	名前、説明、状態、タグ、作成者、作成者（自分）、更新者、更新者（自分）、プロジェクト
プロジェクト	名前、説明
エンドポイント	名前、説明、タイプ、更新者、プロジェクト
Gerrit トリガ	名前、ステータス、プロジェクト
Gerrit サーバ	名前、サーバ URL、プロジェクト
Git Webhook	名前、サーバ タイプ、リポジトリ、ブランチ、プロジェクト

補足説明：

- リンクは、ダッシュボードのパイプライン、実行、またはウィジェットの UUID です。
- 入力パラメータ、出力パラメータ、およびステータス メッセージの表記と例は次のとおりです。
  - 表記：`input.<inputKey>:<inputValue>`

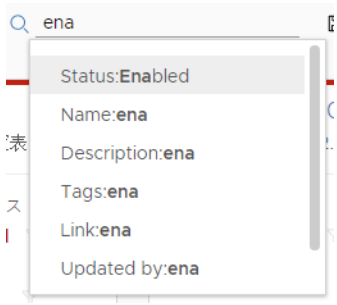
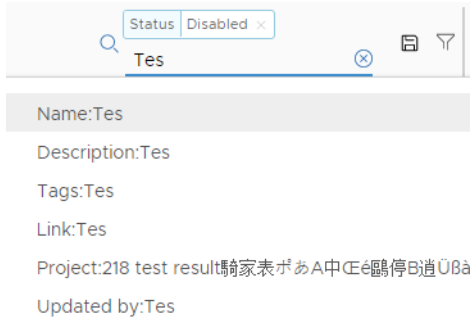
    例：**`input.GERRIT_CHANGE_OWNER_EMAIL:joe_user`**
  - 表記：`output.<outputKey>:<outputValue>`

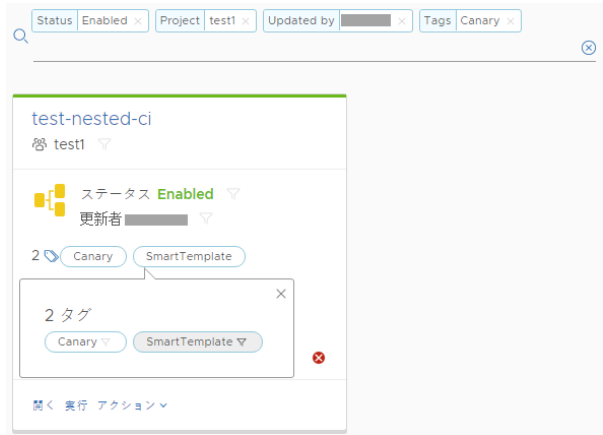
    例：**`output.BuildNo:29`**
  - 表記：`statusMessage:<value>`

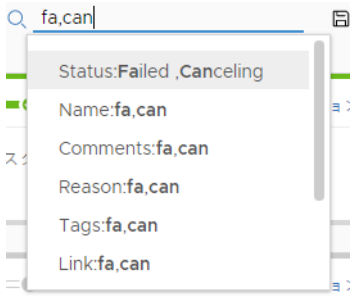
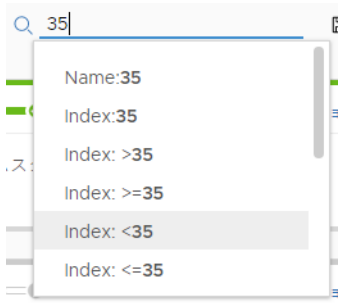
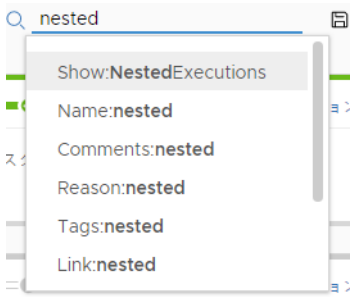
    例：**`statusMessage:Execution failed`**
- ステータスまたは状態は検索画面によって異なります。
  - 実行の場合、設定可能な値には、completed、failed、rollback_failed、または canceled が含まれます。

  - パイプラインの場合、設定可能な状態の値には、有効、無効、またはリリース済みが含まれます。
  - トリガーの場合、設定可能な状態の値には、有効または無効が含まれます。
- 実行者（自分）、作成者（自分）、または作成者（自分）とは、自分、つまりログインしているユーザーを指します。

検索は、有効なすべての画面の右上に表示されます。空白になっている検索欄に入力し始めると、Code Stream は画面のコンテキストを認識し、検索のためのオプションを提示します。

検索に使用可能な方法	入力方法
検索パラメータの一部を入力します。 たとえば、有効なパイプラインをすべてリストするステータス フィルタを追加するには、**ena** と入力します。	
見つかる項目の数を減らすには、フィルタを追加します。 たとえば、**Tes** と入力して、名前フィルタを追加します。フィルタは、既存の [Status:disabled] フィルタを使用して AND として機能し、名前に **Tes** が付いた無効なパイプラインのみを表示します。 さらにフィルタを追加するときには、残っている [名前]、[説明]、[タグ]、[リンク]、[プロジェクト]、[更新者] のオプションが表示されます。	

表示されるアイテムの数を減らすには、パイプラインまたはパイプラインの実行のプロパティで、フィルタ アイコンをクリックします。
  - パイプラインの場合は、[ステータス]、[タグ]、[プロジェクト]、[更新者] のそれぞれにフィルタ アイコンがあります。
  - 実行の場合は、[タグ]、[実行者]、[ステータス メッセージ] のそれぞれにフィルタ アイコンがあります。
たとえば、パイプライン カードでアイコンをクリックして、[SmartTemplate] タグのフィルタを以下の既存のフィルタに追加します。[Status:Enabled]、[Project:test]、[Updated by:user]、[Tags:Canary]。
	

検索に使用可能な方法	入力方法
すべてのアイテムを 2 つの実行状態に含めるには、コンマ区切り記号を使用します。 たとえば、**fa,can** を入力して、OR として機能するステータス フィルタを作成して、すべての失敗またはキャンセルされた実行をリストします。	
インデックス範囲内のすべてのアイテムを含めるには、数値を入力します。 たとえば、**35** を入力し、**<** を選択して、インデックス番号が 35 未満のすべての実行をリストします。	
タスクとしてモデル化されるパイプラインは、ネストされた実行になり、デフォルトによるすべての実行はリストされません。 ネストされた実行を表示するには、**nested** と入力して、[Show] フィルタを選択します。	

お気に入りの検索を保存する方法

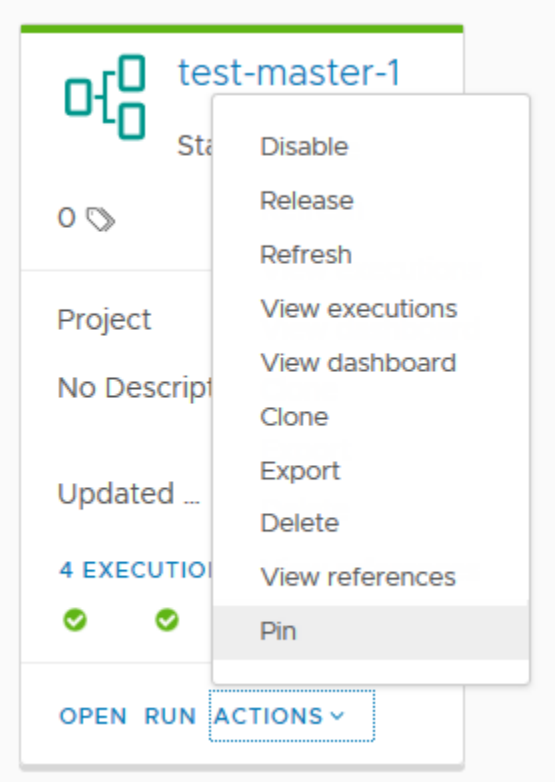
お気に入りの検索を保存すると、検索領域の横にあるディスク アイコンをクリックすることによって各画面で使用できます。

- 検索のパラメータを入力し、アイコンをクリックしてその検索を保存すると、検索に **my enabled** などの名前が追加されます。
- 検索を保存した後、検索にアクセスするにはそのアイコンをクリックします。また、[管理] を選択して、保存された検索のリストで名前変更、削除、または移動することもできます。

検索はユーザー名に関連付けられ、検索が適用される画面にのみ表示されます。たとえば、パイプラインの画面で [Status:enabled] に **my enabled** という名前の検索を保存した場合、[Status:enabled] がトリガに有効な検索であるにも関わらず、**my enabled** 検索は Gerrit トリガの画面では使用できません。

お気に入りのパイプラインを保存可能

お気に入りのパイプラインまたはダッシュボードがある場合は、パイプライン画面またはダッシュボード画面の上部に常に表示されるように固定することができます。パイプライン カードで、[アクション] - [固定] の順にクリックします。



Code Stream 管理者および開発者向けのその他のリソース

Code Stream の管理者または開発者は、 Code Stream の詳細について学ぶことができます。

表 9-1. 管理者向けのその他のリソース

詳細を知りたい内容	参照リソース
管理者が Code Stream を使用できるその他の方法： ■ クラウド ネイティブ アプリケーションのテストおよびリリースを自動化するためのパイプラインを構成する。 ■ テストを通じて開発者のソース コードを自動化して、本番環境に対してテストする。 ■ 開発者がプライマリ ブランチに変更をコミットする前にテストするためのパイプラインを構成する。 ■ 重要なパイプライン メトリックを追跡する。	Code Stream ■ vRealize Automation のドキュメント ■ vRealize Automation 製品 Web サイト VMware ハンズオン ■ vRealize Automation コミュニティを使用する。 ■ VMware Learning Zone を使用する。 ■ VMware のブログを検索する。 ■ VMware のハンズオン ラボを試す。

表 9-2. 開発者向けのその他のリソース

詳細を知りたい内容	参照リソース
開発者が Code Stream を使用できるその他の方法： ■ パブリックおよびプライベートのレジストリ イメージを使用して新しいアプリケーションまたはサービスの環境をビルドする。 ■ 最新の安定したビルドからブランチを作成できるように開発環境をセットアップする。 ■ 最新のコード変更およびアーティファクトを使用して開発環境を更新する。 ■ 他の依存関係サービスの最新の安定したビルドに対するコミットされていないコード変更をテストする。 ■ プライマリ CICD パイプラインにコミットされた変更によって他のサービスが中断されたときに通知を受け取る。	Code Stream ■ vRealize Automation のドキュメント ■ vRealize Automation 製品 Web サイト VMware ハンズオン ■ vRealize Automation コミュニティを使用する。 ■ VMware Learning Zone を使用する。 ■ VMware のブログを検索する。 ■ VMware のハンズオン ラボを試す。