デプロイ時におけるロールバックの安全性の確保

Sandeep Pokkunuri

Amazon で私たちがソリューションを構築するときに指針とする原則の 1 つは、一方向だけのドアを通り抜けるのを回避することです。わかりやすく言うと、元に戻したり拡張したりするのが困難な選択肢は選ばないということです。私たちは、製品、機能、API、バックエンドシステムの設計からデプロイまで、ソフトウェア開発のすべての段階にこの原則を適用します。この記事では、私たちがこの原則をソフトウェア開発にどのように適用するかについて説明します。

デプロイを行うと、ソフトウェア環境がある状態 (バージョン) から別の状態に変わります。ソフトウェアは、どちらの状態でも完璧に機能するかもしれません。しかし、ソフトウェアは、前方移行 (アップグレードまたはロールフォワード) または後方移行 (ダウングレードまたはロールバック) の間または後に適切に機能しない可能性があります。ソフトウェアが適切に機能しない場合、サービスの中断を招き、その結果、お客様からの信頼を失ってしまいます。この記事では、両方のバージョンのソフトウェアが予期したとおりに機能することを前提とし、デプロイ中のロールフォワードまたはロールバックがエラーにつながらないようにすることに重点を置きます。

私たちは、新しいバージョンのソフトウェアをリリースする前に、ベータまたはガンマテスト環境で機能、同時性、パフォーマンス、拡張性、ダウンストリームでの障害処理など、ソフトウェアの複数の特性をテストします。このテストにより、新しいバージョンの問題を明らかにし、それを修正することができます。ただし、デプロイの成功を保証するだけでは必ずしも十分とは言えません。予期せぬ状況や本番環境でソフトウェアが最適に動作しないといった問題に直面することがあります。

Amazon では、デプロイのロールバックによってお客様の側でエラーが発生する可能性があるという状況に陥るのを回避したいと考えています。このような状況に陥るのを回避するため、私たちはデプロイの前に必ずロールバックの準備を入念に行います。あるバージョンのソフトウェアがエラーや前のバージョンで使用できた機能の中断なしにロールバックできる場合、このようなバージョンは後方互換と呼ばれます。私たちは、私たちのソフトウェアがすべてのリビジョンで後方互換になるように計画し、それを検証します。

Amazon のソフトウェアアップデートに対するアプローチについて詳しく説明する前に、スタンドアロンソフトウェアのデプロイと分散型ソフトウェアのデプロイのいくつかの違いについてお話したいと思います。

スタンドアロンソフトウェアと分散型ソフトウェアのデプロイの比較

1 台のデバイス上で 1 つのプロセスとして実行されるスタンドアロンソフトウェアの場合、デプロイはアトミックです。ソフトウェアの 2 つのバージョンが同時に実行されることは決してありません。スタンドアロンソフトウェアが状態を保っている場合、新しいバージョンは古いバージョンによって書き込まれた (シリアル化された) データを読み取る (シリアル化解除する) 必要があります (その逆も同様です)。この条件を満たせば、デプロイは安全にロールバックおよびロールフォワードできます。
分散システムでは、デプロイはより複雑になります。デプロイはローリング更新によって行われるため、可用性には影響しません。新しいバージョンはホストのサブセットに一度に展開されるため、その他のホストは引き続きリクエストに対応できます。一般に、これらのホストは、リモートプロシージャコール (RPC) や共有永続状態 (メタデータやチェックポイントなど) を介して相互に通信します。このような通信または共有状態により、新たな課題が提示されます。書き込み側と読み取り側が異なるバージョンのソフトウェアを実行し、その結果、データについて異なる解釈をする可能性があります。読み取り側は、データをまったく読み取れず、停止する可能性すらあります。

プロトコルの変更による問題

私たちは、ロールバック不能の最も一般的な理由がプロトコルの変更にあることを突き止めました。たとえば、データをディスクに保存しながらデータの圧縮を開始するコード変更について考えてみてください。新しいバージョンが圧縮されたデータを書き込んだら、ロールバックできなくなります。古いバージョンは、ディスクから読み取ったデータを解凍する必要があることがわからないからです。データが BLOB またはドキュメントストアに保存された場合、デプロイ中であっても他のサーバーはそのデータの読み取りに失敗します。このデータが 2 つのプロセス間またはサーバー間で渡されると、レシーバーはその読み取りに失敗します。

プロトコルの変更は非常に微妙である場合があります。たとえば、1 つの接続を介して非同期に通信する 2 つのサーバーを想定してください。アライブであることを互いに認識し続けるには、ハートビートを 5 秒間隔で相互に送信することに同意します。指定した時間内にサーバーがハートビートを認識しない場合、他方のサーバーはダウンしているとみなし、接続を閉じます。
次に、このハートビートの間隔を 10 秒に増やしたデプロイを想定してみましょう。コードコミットは数値だけの些細な変更と思われます。しかし、ロールフォワードもバックワードも安全ではなく

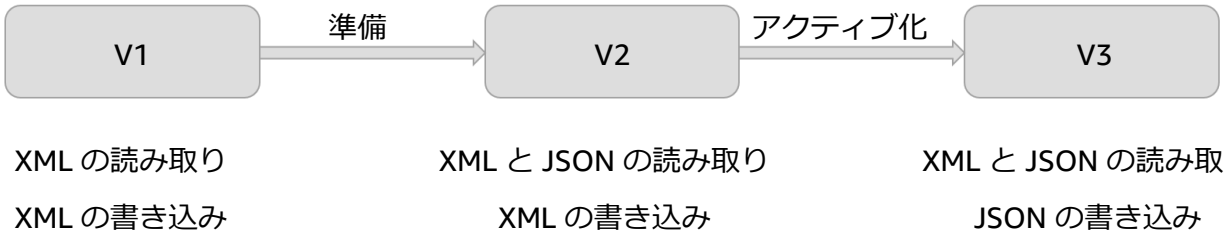
なります。デプロイ時に、新しいバージョンを実行しているサーバーは 10 秒間隔でハートビートを送信します。その結果、古いバージョンを実行しているサーバーは 5 秒を超えるハートビートを識別せず、新しいバージョンを実行しているサーバーとの接続を終了します。大規模なフリートの場合、複数の接続でこの状況が発生するため可用性が低下します。

コードの読み取りやドキュメントの設計で微妙な変更を分析するのは困難です。このため、各デプロイのロールフォワードとバックワードが安全であることを明確に検証します。

2 フェーズデプロイ技術

ロールバックが安全であることを確認する 1 つの方法として、一般的に 2 フェーズデプロイと呼ばれる技術を使用します。Amazon Simple Storage Service (Amazon S3) でデータ (書き込み対象、読み取り対象) を管理するサービスを使用した、次のような仮想シナリオを検討します。このサービスは、スケーリングと可用性を目的とし、複数のアベイラビリティーゾーンにまたがったサーバーのフリートで実行されます。

現在、サービスは XML 形式を使用してデータを維持しています。下図に示すように、バージョン V1 ではすべてのサーバーが XML を読み書きします。業務上の理由から、JSON 形式でデータを維持する必要性が生じます。この変更を 1 つのデプロイで行う場合、変更が適用されるサーバーは JSON で書き込みます。しかし、他のサーバーは JSON で読み取れません。この状況によってエラーが発生します。したがって、変更を 2 つの部分に分けて 2 フェーズデプロイを行います。



上図に示すように、最初のフェーズを準備と呼びます。このフェーズでは、バージョン V2 をデプロイし、すべてのサーバーが (XML に加えて) JSON を読み取る一方、XML を継続して書き込むように準備します。運用上の観点からこの変更による影響はありません。すべてのサーバーで引き続き XML を読み取り、すべてのデータを XML で書き込めます。この変更をロールバックする場合、サーバーは JSON を読み取れない状態に戻ります。JSON で書き込まれたデータがまだ存在しないため、これは問題ではありません。

上の図に示すように、2 つ目のフェーズをアクティブ化と呼びます。このフェーズでは、バージョン V3 をデプロイし、JSON 形式を書き込みで使用できるようサーバーをアクティブ化します。各サーバーにこの変更が適用されると、JSON での書き込みが開始されます。最初のフェーズで準備したため、この変更がまだ適用されていないサーバーでも JSON を読み取れます。この変更をロールバックする場合、一時的になったアクティブ化フェーズだったサーバーで書き込まれたすべてのデータは JSON です。アクティブ化フェーズでなかったサーバーで書き込まれたデータは XML です。V2 に示すように、ロールバック後もサーバーは引き続き XML と JSON のどちらも読み取れるため、この状況に問題はありません。

上の図では XML から JSON への形式変更のシリアル化を説明しましたが、一般的な技術は前述のプロトコルの変更に関するセクションに記載したすべての状況に適用されます。例として、サーバー間のハートビート期間を 5 秒から 10 秒に増やす必要があった過去のシナリオを振り返ってみましょう。準備フェーズでは、すべてのサーバーのハートビート期間を 10 秒に指定できますが、すべてのサーバーがハートビートを 5 秒おきに送信し続けます。アクティベートフェーズで、この頻度を 10 秒間隔に変更できます。

2 フェーズデプロイの注意事項

ここからは、2 フェーズのデプロイ手法に従う場合の注意事項について説明します。前出のセクションに記載されているシナリオ例で述べたように、これらの注意事項は多くの 2 フェーズデプロイに該当します。

多くのデプロイツールでは、変更が適用されたホストの数が少なければデプロイは成功とみなされ、正常とレポートされます。たとえば、AWS CodeDeploy には minimumHealthyHosts というデプロイ設定があります。

2 フェーズデプロイの例における重要な想定事項が、第 1 フェーズの終了までに、すべてのサーバーが XML と JSON を読み取るようアップグレードされていることです。第 1 フェーズで 1 つ以上のサーバーがアップグレードされなかった場合、第 2 フェーズの終了後にデータが読み取られません。そのため、準備フェーズではすべてのサーバーに変更が適用されたことをはっきりと確認します。

Amazon DynamoDB の開発時に、複数のマイクロサービスにまたがる膨大な数のサーバー間の通信プロトコルを変更することを決定しました。すべてのサーバーが準備フェーズに達してからアクティベートフェーズに進むよう、すべてのマイクロサービス間でデプロイを調整しました。万一に備え、各フェーズの最後にはデプロイが成功したことを 1 つ 1 つのサーバーではっきりと確認しました。

2 つの各フェーズではロールバックを安全に実行できますが、両方の変更をロールバックすることはできません。前述の例では、アクティベートフェーズの終わりにサーバーで JSON 形式のデータが書き込まれます。準備フェーズとアクティベートフェーズでの変更の前に使用されていたソフトウェアバージョンでは、JSON を読み取れません。したがって、念のため準備フェーズとアクティベートフェーズの間に長い時間を空けることにしています。この期間は待機期間と呼ばれ、通常は数日間に設定されます。待機期間を設定することで、旧バージョンへのロールバックが必要な事態を回避します。

アクティベートフェーズの終了後は、XML を読み取るソフトウェアの機能を安全に削除できなくなります。これは、準備フェーズの前に書き込まれたデータがすべて XML 形式であるためです。この機能は、すべてのオブジェクトが JSON で書き直されたことを確認した後で削除する必要があります。この処理はバックフィルと呼ばれます。このために、サービスでのデータの書き込みおよび読み取り中に同時に実行できる追加のツールが必要になる場合があります。

シリアル化のベストプラクティス

多くのソフトウェアでは、永続化やネットワーク上での転送を目的としてデータのシリアル化が行われます。ソフトウェアの進化に応じて、シリアル化のロジックが変更されるのは一般的なことです。これらの変更には、新しいフィールドの追加や形式の完全な変更が含まれます。年月を積み重ねる中で、シリアル化で従うべきベストプラクティスがいくつか確立されました。

- 通常、カスタムシリアル化形式は開発しません。

  カスタムシリアル化の初期ロジックは影響が少なく、むしろパフォーマンスを向上させる場合もあるかもしれません。しかし、その後の形式の反復処理により、JSON、Protocol Buffers、Cap'n Proto、FlatBuffers などの確立されたフレームワークではすでに解決されている問題が発生します。これらのフレームワークを適切に使用すると、エスケープ処理、後方互換性、属性存在追跡 (フィールドが明示的に設定されたか、暗黙的にデフォルト値が割り当てられたか) などの安全機能を活用できます。
- 変更ごとに、固有のバージョンをシリアライザに明示的に割り当てます。

  この処理は、ソースコードやビルドのバージョニングに関係なく行います。また、シリアライザバージョンをシリアル化されたデータとともに保存するか、メタデータに保存します。古いバージョンのシリアライザは、新しいソフトウェアでも引き続き機能します。通常、書き込まれたデータまたは読み取られたデータのバージョンのメトリクスを出力する

ことは有用です。エラーがあった場合に、オペレータに可視性が提供され、トラブルシューティング情報を確認できます。これはすべて、RPC と API バージョンにも適用されます。

- 制御できないデータ構造のシリアル化を回避します。

  たとえば、リフレクションを使用すると Java のコレクションオブジェクトをシリアル化できましたが、JDK をアップグレードしようとすると、そのようなクラスの基盤となる実装が変更され、逆シリアル化が失敗する可能性があります。このリスクは、チーム間で共有されるライブラリのクラスにも当てはまります。
- 通常、シリアライザは不明な属性の存在を許可するように設計されています。

  可能な場合、シリアライザはデータを書き戻す間、不明な属性を保持します。この処理により、新しいバージョンのソフトウェアを実行するサーバーによってシリアル化中に新しい属性がデータに追加された場合でも、古いバージョンを実行するサーバーが同じデータを更新する際に属性が消去されることはありません。したがって、2 フェーズデプロイは必要ありません。

多くのベストプラクティスと同様に、ガイドラインはすべてのアプリケーションとシナリオに適用できるわけではないことにご注意ください。

変更をロールバックしても問題がないことを確認する

一般的に、ソフトウェアの変更をロールフォワードおよびロールバックしても問題ないことを、アップグレード-ダウングレードテストを通じて明示的に検証します。この処理においては、本番環境を代表するテスト環境をセットアップします。長年にわたり、テスト環境をセットアップする際に避けるパターンをいくつか特定してきました。

変更がテスト環境のすべてのテストに合格したにもかかわらず、本番環境に変更をデプロイするとエラーが発生することがあります。あるケースでは、テスト環境の各サーバーにサービスが 1 つずつしかありませんでした。したがって、すべてのデプロイはアトミックであり、異なるバージョンのソフトウェアを同時に実行する可能性は排除されていました。現在は、テスト環境で本番環境ほどのトラフィックが見られない場合でも、本番環境と同様に、各サービスの背後にある異なるアベイラビリティーゾーンの複数のサーバーが使用されます。Amazon では倹約が推奨されますが、品質の保証に関しては当てはまりません。

別のケースでは、テスト環境に複数のサーバーがありましたが、ただし、テストを高速化するために、すべてのサーバーにデプロイを一括して実行していました。このアプローチでは、ソフトウェアの古いバージョンと新しいバージョンが同時に実行されることも回避されており、ロールフォワードでの問題は検出されませんでした。現在は、すべてのテスト環境と本番環境で同じデプロイ設定を使用しています。

マイクロサービス間の調整を伴う変更については、テスト環境と本番環境のマイクロサービス間で同じデプロイ順序が維持されます。ただし、ロールフォワードとロールバックの順序は異なる場合があります。たとえば、通常、シリアル化のコンテキストでは特定の順序に従います。つまり、ロールフォワード中は読み込みが書き込みに先行し、ロールバック中は書き込みが読み込みに先行します。テスト環境と実稼働環境の両方で、該当する順番で処理が行われます。

テスト環境のセットアップが本番環境に類似している場合は、可能な限り厳密に本番トラフィックをシミュレートします。たとえば、いくつかのレコード (またはメッセージ) が短時間で連続して作成されて読み込まれます。API はすべて継続して実行されます。次に、環境を 3 つの段階に分けます。各段階は、潜在的なバグを特定するために必要な期間継続されます。この期間は、すべての API、バックエンドワークフロー、バッチジョブを少なくとも 1 回実行するのに十分な時間になります。
最初の段階では、フリートの約半分に変更をデプロイして、ソフトウェアバージョンを共存させます。次の段階では、デプロイを完了させます。3 番目の段階では、ロールバックデプロイを開始し、すべてのサーバーが古いソフトウェアを実行するまで同じ手順を実行します。これらの段階でエラーや予期しない動作がなければ、テストが成功したとみなされます。

まとめ

お客様へのサービス提供の中断なしにデプロイをロールバックできるようにすることは、サービスの信頼性を高めるうえで不可欠です。ロールバックの安全性を明示的にテストすることで、エラーの発生しやすい手動分析に依存する必要がなくなります。私たちは、安全にロールバックできない変更を見つけた場合、通常はそれぞれ安全にロールバック/ロールフォワードできる 2 つの変更に分割します。