AWS Well-Architected フレームワーク

信頼性の柱

信頼性の柱: AWS Well-Architected フレームワーク

Table of Contents

信頼性の柱 – AWS Well-Architected フレームワーク

発行日: 2024 年 11 月 6 日 (ドキュメントの改訂)

このホワイトペーパーは、AWS Well-Architected フレームワークの信頼性の柱に焦点を当てています。お客様が Amazon Web Services (AWS) 環境の設計、配信、メンテナンスを行う際にベストプラクティスを適用するためのガイダンスを提供します。

序章

AWS Well-Architected フレームワークは、AWS でワークロードを構築する際に行う決定の長所と短所を理解するのに役立ちます。このフレームワークを使用することで、信頼性、安全性、効率性、コスト効率に優れ、持続可能なワークロードをクラウド内で設計および運用するための、アーキテクチャ上のベストプラクティスを学ぶことができます。アーキテクチャをベストプラクティスに照らして評価し、改善すべき領域を特定するための一貫した方法を提供します。当社は、Well-Architected ワークロードを備えることで、ビジネス成功の可能性が大幅に高まると確信しています。

AWS Well-Architected フレームワークは 6 つの柱に基づいています。

- 運用上の優秀性
- セキュリティ
- 信頼性
- パフォーマンス効率
- コスト最適化
- 持続可能性

このホワイトペーパーでは、信頼性の柱をお客様のソリューションに適用する方法について説明します。従来のオンプレミス環境では、単一障害点、自動化の欠如、伸縮性の欠如が原因で、信頼性の実現が困難な場合があります。このホワイトペーパーで紹介するプラクティスを採用することで、強固な基盤、一貫した変更管理、実証済みの障害復旧プロセスを備えた、回復力の高いアーキテクチャを構築できます。

このホワイトペーパーの対象者は、最高技術責任者 (CTO)、アーキテクト、開発者、オペレーションチームメンバーなどの技術担当者です。このホワイトペーパーを読むことで、信頼性の高いクラウドアーキテクチャを設計するための AWS のベストプラクティスと戦略を理解することができます。

このホワイトペーパーには、高レベルの詳細な実装情報、アーキテクチャパターン、その他リソースの参考資料が含まれています。

信頼性

信頼性の柱には、意図した機能を期待どおりに、正しく、一貫して実行するワークロードの能力が含まれます。これには、ワークロードのライフサイクル全体を通じてワークロードを運用およびテストする能力が含まれます。本資料では、AWS に信頼性の高いワークロードを実装するための、詳細なベストプラクティスのガイダンスを提供します。

トピック

回復力に関する責任共有モデル

回復力については、AWS とお客様で責任を共有します。ディザスタリカバリ (DR) と可用性が回復力の一環として、この責任共有モデルにおいてどのように運用されるのかを理解することが重要です。

AWS の責任 - クラウドの回復力

AWS は、AWS クラウドで提供されるすべてのサービスを実行するインフラストラクチャの回復力について責任を負います。このインフラストラクチャは、 AWS クラウドサービスを実行するハードウェア、ソフトウェア、ネットワーキング、施設で構成されています。AWS は、商業的に合理的な努力を払ってこれらの AWS クラウドサービスを利用可能にし、AWS サービス レベル アグリーメント (SLA) を満たすか、それを超えるサービスの可用性を確保します。

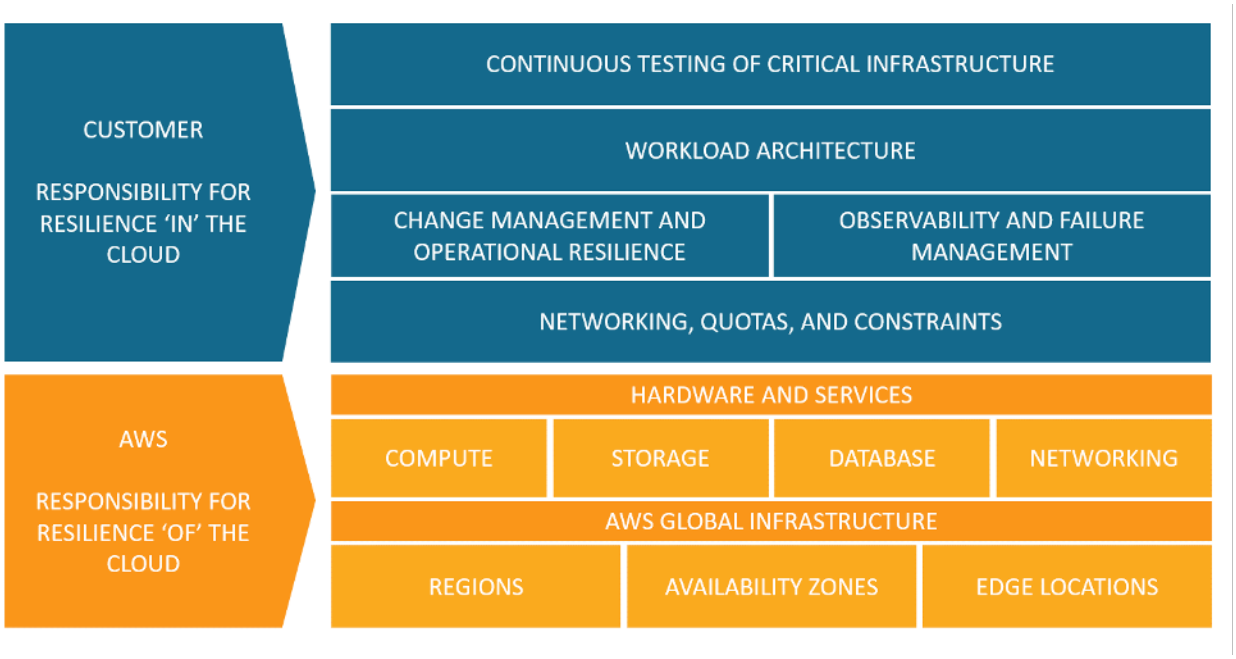
AWS グローバルクラウドインフラストラクチャは、お客様が回復力の高いワークロードアーキテクチャを構築できるように設計されています。各 AWS リージョンは完全に分離されており、複数のアベイラビリティーゾーンで構成されています。アベイラビリティーゾーンは、インフラストラクチャの物理的に分離された部分です。アベイラビリティーゾーンは、ワークロードの回復力に影響を及ぼす可能性のある障害を分離し、リージョン内のその他のゾーンへの影響を回避します。ただし、同時に AWS リージョン内のすべてのゾーンは、高帯域幅、低レイテンシーのネットワーキングで相互接続されています。ゾーン間をつなぐのは、高スループット、低レイテンシーのネットワーキングを提供する、完全な冗長性を備えた専用メトロファイバーです。ゾーン間のすべてのトラフィックは暗号化されています。ゾーン間の同期レプリケーションを実行するために、十分なネットワークパフォー

マンスが提供されます。アプリケーションを AZ 間でパーティショニングすると、企業は、停電、落雷、竜巻、台風などの問題からよりよく隔離され、保護されます。

お客様の責任 - クラウドの回復力

お客様の責任は、選択した AWS クラウドサービスによって異なります。選択したサービスにより、お客様が回復力に関する責任の一環として実行する必要がある、設定作業の量が決まります。例えば、Amazon Elastic Compute Cloud (Amazon EC2) のようなサービスでは、必要となる回復力の設定と管理をすべてお客様が実行する必要があります。Amazon EC2 インスタンスをデプロイするお客様は、複数の場所 (AWS アベイラビリティーゾーンなど) に Amazon EC2 インスタンスをデプロイし、自動スケーリングなどのサービスを使用して自己修復を実装し、インスタンスにインストールされたアプリケーションに回復力のあるワークロードアーキテクチャのベストプラクティスを使用する責任があります。Amazon S3 や Amazon DynamoDB などのマネージドサービスについては、インフラストラクチャレイヤー、オペレーティングシステム、プラットフォームの運用を AWS が行い、お客様はエンドポイントにアクセスしてデータを保存、取得します。お客様は、バックアップ、バージョニング、レプリケーション戦略など、データの回復力を管理する責任があります。

AWS リージョン内の複数のアベイラビリティーゾーンにワークロードをデプロイすることは、問題を単一のアベイラビリティーゾーンに分離することでワークロードを保護するように設計された高可用性戦略の一環です。これにより、その他のアベイラビリティーゾーンの冗長性を使用して、リクエストの処理を継続できます。マルチ AZ アーキテクチャは、ワークロードをより適切に分離し、停電、落雷、竜巻、地震などの問題から保護するように設計された DR 戦略の一環でもあります。DR 戦略として、複数の AWS リージョンを利用することもできます。例えば、アクティブ/パッシブ設定では、アクティブなリージョンがリクエストを処理できなくなった場合に、ワークロードのサービスはアクティブなリージョンから DR リージョンにフェイルオーバーします。



お客様と AWS の、クラウド内およびクラウドの回復力に対する責任。

お客様は、AWS サービスを使用して回復力の目標を達成できます。お客様は、システムの以下の側面を管理して、クラウド内の回復力を実現する責任を負います。特定の各サービスの詳細については、「AWS のドキュメント」を参照してください。

ネットワーキング、クォータ、制約

- この分野の責任共有モデルのベストプラクティスについては、「基礎」で詳しく説明しています。
- 必要に応じて、含めるサービスの Service Quotas と制約を把握し、予想される負荷リクエストの増加に基づいて、スケーリングのための十分な余裕を持ってアーキテクチャを計画します。
- 可用性、冗長性、スケーラビリティに優れたネットワークトポロジを設計します。

変更管理と運用上の回復力

- 変更管理には、環境に変更を導入および管理する方法が含まれます。変更を実装するには、ランブックを構築し、アプリケーションとインフラストラクチャのデプロイ戦略を最新の状態に保つ必要があります。
- ワークロードリソースをモニタリングするための回復力のある戦略では、技術メトリクスとビジネスメトリクス、通知、自動化、分析を含むすべてのコンポーネントを考慮します。
- クラウド内のワークロードは、使用量の減損や変動に伴う需要のスケーリングの変化に適応する必要があります。

オブザーバビリティ管理と障害管理

- ワークロードがコンポーネントの障害に耐えられるようにヒーリングを自動化するには、モニタリングによる障害の監視が必要です。
- 障害管理には、データのバックアップ、ワークロードがコンポーネントの障害に耐えられるようにするためのベストプラクティスの適用、ディザスタリカバリの計画が必要です。

ワークロードアーキテクチャ

- ワークロードアーキテクチャには、ビジネスドメインに関するサービスを設計する方法、障害を防ぐために SOA と分散システム設計を適用する方法、スロットリング、再試行、キュー管理、タイムアウト、緊急対策などの機能を組み込む方法が含まれます。
- ベストプラクティスとジャンプスタートの実装に合わせて、実証済みの AWS ソリューション、Amazon Builders Library、サーバーレスパターンを活用します。
- 継続的な改善を採用すると、システムを分散サービスに分解し、スケーリングとイノベーションを加速できます。AWS マイクロサービスガイダンスとマネージドサービスオプションを使用して、変更とイノベーションを導入する能力を簡素化し、高速化します。

重要なインフラストラクチャの継続的テスト

- 信頼性のテストでは、機能レベル、パフォーマンスレベル、カオスレベルでのテストと、インシデント分析とゲームデープラクティスを採用して、十分に把握していない問題を解決するための専門知識を構築します。
- すべてをクラウドに移行した (オールイン) アプリケーションとハイブリッドアプリケーションの両方で、問題発生時やコンポーネントの停止時にアプリケーションがどのように動作するかを把握しておくと、停止から迅速かつ確実に復旧できます。
- 繰り返し可能な実験を作成し、文書化して、期待どおりにいかない場合にシステムがどのように動作するかを把握しておきます。このようなテストは、全体的な回復力の有効性を証明するものであり、実際の障害シナリオに直面する前に、運用手順のフィードバックループを提供します。

設計原則

クラウドには、信頼性の向上に役立ついくつかの原則があります。ベストプラクティスについてこれから説明しますが、以下の原則を覚えておいてください。

- 障害から自動的に復旧する: システムでワークロードの主要業績評価指標 (KPI) をモニタリングすることで、しきい値を超えた場合に自動化を実行できます。この場合の KPI は、サービスの運用の技術的側面ではなく、ビジネス価値に関する指標である必要があります。これにより、障害発生時の自動通知と追跡が可能になり、障害に対処する、または障害を修正するための復旧プロセスを自動化できます。より複雑な自動化を使用すると、障害が発生する前に修正を予期できます。
- リカバリ手順をテストする: オンプレミス環境では、ワークロードが特定のシナリオで動作することを実証するためのテストを行うことがよくあります。復旧戦略を検証するためにテストを実施することはあまりありません。クラウドでは、どのようにワークロードに障害が発生するかをテストし、復旧の手順を検証できます。オートメーションを使用してさまざまな障害をシミュレートすることも、以前に障害が発生したシナリオを再現することもできます。このアプローチでは、実際の障害シナリオが発生する前にテストを行い、修正できる障害経路が公開されるため、リスクが軽減されます。
- 水平方向にスケールしてワークロード全体の可用性を高める: 1 つの大規模なリソースを複数の小規模なリソースに置き換えることで、1 つの障害がシステム全体に及ぼす影響を軽減します。リクエストを複数の小規模なリソースに分散することで、一般的な障害点を共有しないようにします。
- 容量の推測をやめる: オンプレミスのワークロードにおける障害の一般的な原因はリソースの飽和状態で、ワークロードに対する需要がそのワークロードの容量を超えたときに発生します (サービス妨害攻撃の目標となることがよくあります)。クラウドでは、需要とワークロード使用率をモニタリングし、リソースの追加と削除を自動化することで、プロビジョニングが過剰にも過小にもならない、需要を満たす最適なレベルを維持できます。制限はまだありますが、いくつかのクオータは制御可能で、そのほかのクオーターも管理できます (「Service Quotas と制約の管理」を参照)。
- 自動化の変更を管理する: インフラストラクチャに対する変更は、自動化を使用して実行する必要があります。管理する必要がある変更には、自動化に対する変更が含まれており、それを追跡して確認することができます。

定義

このホワイトペーパーでは、クラウドの信頼性を対象として、次の 4 つの分野のベストプラクティスについて説明します。

- 基礎
- ワークロードアーキテクチャ
- 変更管理
- 障害管理

信頼性を実現するには、基盤、つまりワークロードに対して Service Quotas とネットワークトポロジを適応させた環境を作ることから始める必要があります。分散システムにおけるワークロードのアーキテクチャは、障害を防止および軽減するように設計する必要があります。ワークロードは需要または要件の変化に対応する必要があり、障害を検出して自動的に復旧するように設計する必要があります。

トピック

回復力、および信頼性のコンポーネント

クラウド内のワークロードの信頼性はいくつかの要因に依存しますが、その基本となるのは回復力です。

- 回復力は、インフラストラクチャまたはサービスの中断から復旧し、需要に合わせて動的にコンピューティングリソースを取得し、設定ミスや一時的なネットワーク問題のような障害を軽減するワークロードの能力です。

ワークロードの信頼性に影響を与えるその他の要因には、次のようなものがあります。

- 運用上の優秀性。これには、変更の自動化、障害対応のためのプレイブックの利用、アプリケーションに本番運用の準備ができていることを確認するための運用準備状況レビュー (ORR) が含まれます。
- セキュリティ。これには、可用性に影響を与える、悪意のある行為によるデータまたはインフラストラクチャへの危害を防止することが含まれます。例えば、データの安全性を確保するには、バックアップを暗号化します。
- パフォーマンス効率。これには、最大リクエストレートの設計とワークロードに対するレイテンシーの最小化が含まれます。
- コスト最適化。これには、静的な安定性を達成するために EC2 インスタンスにより多く費用をかけるか、より多くの容量が必要な場合に自動スケーリングに依存するかどうかといったトレードオフが含まれます。

回復力は、このホワイトペーパーにおける最大の焦点です。

他の 4 つの要素も重要であり、AWS Well-Architected フレームワークのそれぞれの柱によって力バーされています ここに記載されているベストプラクティスの多くは、信頼性の面にも対応しますが、焦点は回復力です。

利用可能な状況

可用性 (サービス可用性とも呼ばれます) は、回復力を数量的に測定するためによく使用されるメトリクスであると同時に、的を絞った回復力目標でもあります。

- 可用性は、ワークロードが使用可能な時間の割合です。

「使用可能」とは、取り決めた機能を必要なときに正常に実行できることを意味します。

この割合 (%) は、月、年、直近 3 年などの時間単位で計算します。可能な限り厳密に解釈すると、予定された中断や予定外の中断を含め、アプリケーションが正常に動作しないときは、可用性が下がることになります。可用性は次のように定義されます。

$$Availability = \frac{Available\ for\ Use\ Time}{Total\ Time}$$

- 可用性は、一定期間 (通常は 1 か月または 1 年) の稼働時間の割合 (例: 99.9%) です。
- 一般的には「9 の数」で省略して表現され、例えば、「ファイブナイン」は 99.999% の可用性という意味になります。
- 一部のお客様は、計画されたサービスのダウンタイム (計画メンテナンスなど) を計算式の合計時間から除外することを選択します。ただし、このような時間にもユーザーがサービスを利用したい可能性があるため、この方法はお勧めしません。

アプリケーションの可用性における一般的な設計目標と、この目標の達成中、1 年以内に中断が発生する可能性のある最大時間を以下の表に示します。この表には、可用性レベルごとによく知られているアプリケーションの種類の例が示されています。このドキュメント全体を通して、これらの可用性の値を参考にします。

利用可能な状況	最大利用不可能時間 (年間)	アプリケーションのカテゴリ
99%	3 日と 15 時間	バッチ処理、データの抽出、転送、ロードジョブ

利用可能な状況	最大利用不可能時間 (年間)	アプリケーションのカテゴリ
99.9%	8 時間 45 分	ナレッジ管理、プロジェクト追跡などの社内ツール
99.95%	4 時間 22 分	オンラインコマース、POS
99.99%	52 分	動画配信、ブロードキャストワークロード
99.999%	5 分	ATM トランザクション、通信ワークロード

リクエストに基づく可用性の測定。サービスの場合、「使用可能な時間」の代わりに、成功したリクエスト数と失敗したリクエスト数をカウントする方が容易かもしれません。この場合、次の計算を使用できます。

$$Availability = \frac{Successful\ Responses}{Valid\ Requests}$$

これは多くの場合、1 分間または 5 分間で測定されます。次に、これらの期間の平均から、1 か月の稼働時間率 (時間ベースの可用性測定) を計算できます。特定の期間に受信したリクエストがなかった場合、その時間の可用性は 100% になります。

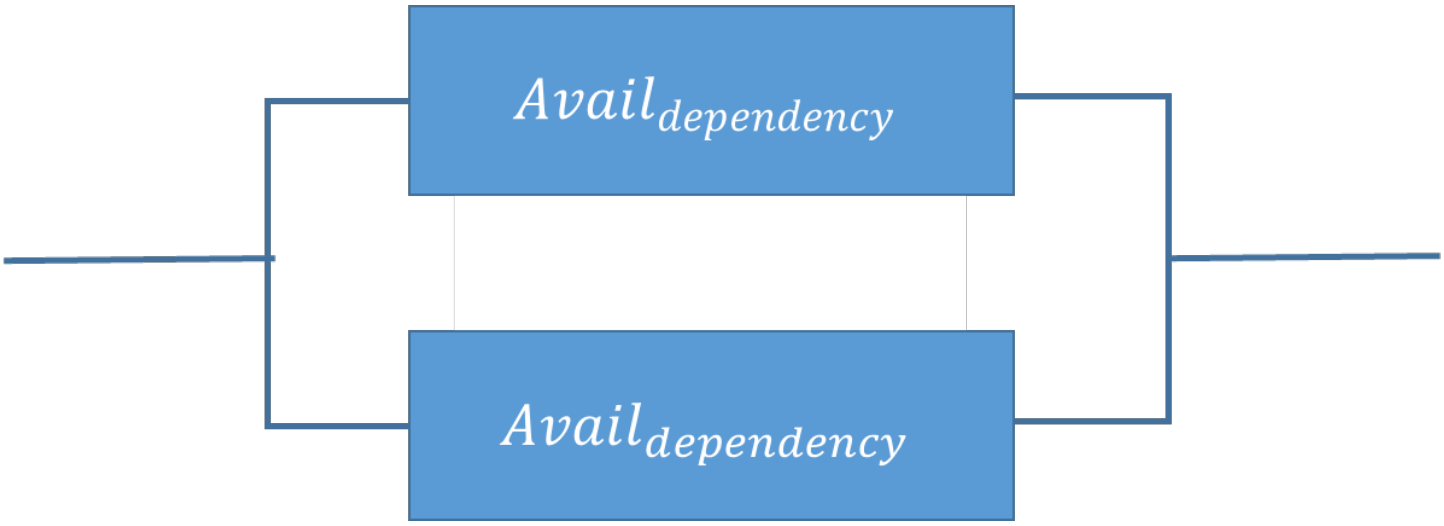
ハードな依存関係を持つ可用性を計算する。多くのシステムは他のシステムとハードな依存関係にあり、依存するシステムでサービス停止が起こると、呼び出す側のシステムにも影響します。この反対はソフトな依存関係で、依存関係にあるシステムに障害が起こると、アプリケーションがそれを補完します。ハードな依存関係が存在する場合、呼び出す側のシステムにおける可用性は、依存するシステムの可用性の積になります。例えば、可用性 99.99% を実現するように設計されたシステムが、同様に可用性が 99.99% である他の 2 つのシステムに依存する場合、このワークロードの可用性は、理論的には 99.97% になります。

$Avail_{invok} \times Avail_{dep1} \times Avail_{dep2} = Avail_{workload}$

99.99% × 99.99% × 99.99% = 99.97%

したがって、お客様自身で可用性を計算する場合は、このシステムとの依存関係と、それらの可用性の設計目標を理解することが重要です。

冗長コンポーネントの可用性を計算する 独立した、冗長化されたコンポーネント (複数のアベイラビリティーゾーンの冗長リソースなど) をシステムが使用する場合、理論的な可用性は、100% からそのコンポーネントの障害率の積を引いたものになります。例えば、あるシステムが 2 つの独立したコンポーネントを利用し、それぞれ 99.9% の可用性を持つ場合、この依存関係の実効可用性は 99.9999% になります。



$Avail_{effective} = Avail_{MAX} - ((100\% - Avail_{dependency}) \times (100\% - Avail_{dependency}))$

99.9999% = 100% − (0.1%×0.1%)

ショートカット計算: 計算内のすべてのコンポーネントの可用性が 9 のみで構成されている場合は、9 の桁の数を合計するだけで答えが得られます。上記の例では、2 つの独立した、冗長なコンポーネントは 9 が 3 つの可用性を持つため、結果は 9 が 6 つになります。

依存するシステムの可用性を計算する 依存関係によっては、多くの AWS のサービスの可用性設計目標など、可用性に関するガイダンスを提供するものもあります。ただし、これを利用できない場合 (メーカーが可用性情報を公開していないコンポーネントなど)、推測する 1 つの方法は、平均故障間隔 (MTBF) と平均復旧時間 (MTTR) を特定することです。可用性の推測値は、次の方法で計算できます。

$$Avail_{EST} = \frac{MTBF}{MTBF + MTTR}$$

例えば、MTBF が 150 日、MTTR が 1 時間なら、可用性の推測値は 99.97% です。

詳細については、可用性の計算に役立つ、「Availability and Beyond: Understanding and improving the resilience of distributed systems on AWS」を参照してください。

可用性のコスト 通常は、アプリケーションの可用性レベルを高く設計すればコストは増大するため、設計に着手する前に可用性の正確なニーズを特定することが重要です。可用性レベルが高いと、網羅的な障害シナリオのもとで、厳しいテスト条件と検証条件が課されることになります。このような場合、あらゆる種類の障害からの回復に自動化が必要になり、システム運用のすべての側面が同様に構築され、同じ基準に基づいてテストされることが必要になります。例えば、容量の追加や削除、更新されたソフトウェアや設定変更のデプロイまたはロールバック、システムデータの移行などが、可用性の設計目標を満たすように実施される必要があります。極めて高いレベルの可用性を前提にソフトウェア開発のコストを積み上げると、システムのデプロイ速度が遅くなるため、イノベーションが難しくなります。このため、これに対するガイダンスは、基準を適用しながら、システム運用におけるライフサイクル全体にわたって適切な可用性の目標を検討することです。

可用性の設計目標が高いシステムにおけるもう一つのコスト増大の原因は、依存関係にあるシステムの選択にあります。目標レベルが高ければ、依存関係を持つソフトウェアまたはサービスの選択肢が狭くなり、前述のようにそれに基づくコストも増大していきます。可用性の設計目標が高くなるほど、リレーショナルデータベースのような多目的のサービスは少なくなり、目的に特化したサービスが多くなります。これは、後者の方が評価、テスト、自動化が容易で、含まれているが使用されていない機能との予期せぬ相互作用の可能性が低いためです。

ディザスタリカバリ (DR) 目標

可用性の目標に加えて、回復力戦略には、災害イベント時にワークロードを復旧する戦略に基づいた、ディザスタリカバリ (DR) 目標も含める必要があります。ディザスタリカバリは、自然災害、大規模な技術的障害、または攻撃や エラーなどの人為的脅威に対応する、1 回限りの復旧目標に焦点を当てています。これは、コンポーネントの障害、負荷の急増、ソフトウェアのバグに対応して、一定期間の平均回復力を測定する可用性とは異なります。

組織によって定義された目標復旧時間 (RTO) RTO とは、サービスが中断してから復旧するまでに経過した時間の、許容される最大値のことです。これにより、サービスが使用不可のときに許容される時間枠が決まります。

組織によって定義された目標復旧時点 (RPO) RPO とは、データが最後に復旧した時点を起点とする経過時間の、許容される最大値のことです。これにより、最後の回復時点からサービスが中断されるまでの間に許容できるデータ損失の程度が決まります。

RPO (目標復旧時点) 、RTO (目標復旧時間) 、災害イベントの関係。

RTO は、停止の開始からワークロード復旧までの時間を測定する点で、MTTR (平均復旧時間) と似ています。ただし、MTTR は、一定期間にわたって複数の可用性に影響を与えるイベントに対して取られる平均値であり、RTO は、可用性に影響を与える単一のイベントの目標値、つまり許容される最大値です。

可用性ニーズを理解する

最初に、アプリケーションの可用性をアプリケーション全体の単一ターゲットとして考えてしまうケースがよくあります。しかし多くの場合、詳しく分析してみれば、アプリケーションやサービスでは、特定の側面に応じて求められる可用性の要件もさまざまであることに気付くはずです。例えば、システムによっては、既存データを取得するよりも、新規データを受信して保存する機能を優先させる場合があります。また、システムの構成や環境を変更するオペレーションよりも、リアルタイムオペレーションを優先させるシステムもあります。1 日のうち特定の時間帯に極めて高い可用性が要求されるサービスでも、その時間帯以外は長時間の中断を許容できる可能性もあります。これらは、1 つのアプリケーションを分解し、それぞれの構成要素の可用性要件を評価する方法のほんの一例です。この方法のメリットは、システム全体を最も厳格な要件に合わせて設計するのではなく、特定のニーズに応じて可用性に労力 (および費用) をかけられることです。

推奨事項

各アプリケーションの固有の側面を厳密に評価し、必要に応じてビジネスニーズを反映して、可用性とディザスタリカバリの設計目標を差別化してください。

通常、AWS では、サービスを「データプレーン」と「コントロールプレーン」に分けて考えます。コントロールプレーンで環境を設定しつつ、データプレーンではリアルタイムのサービスを提供します。例えば、Amazon EC2 インスタンス、Amazon RDS データベース、Amazon DynamoDB テーブルの読み取り/書き込みオペレーションは、すべてデータプレーンによる操作です。逆に、EC2 インスタンスや RDS データベースの起動、DynamoDB のテーブルメタデータの追加や変更などは、すべてコントロールプレーンによる操作です。これらすべての機能には高レベルの可用性が重要となりますが、一般的にデータプレーンの可用性設計の目標は、コントロールプレーンより高くなります。したがって、高い可用性要件を持つワークロードでは、コントロールプレーンの操作に対するランタイム依存を避ける必要があります。

AWS のお客様の多くは、類似のアプローチを採用して、アプリケーションを厳密に評価し、さまざまな可用性ニーズを持つサブコンポーネントを特定しています。次に、さまざまな側面に応じた可用性設計目標の調整を行い、システムを設計するための適切な作業を行います。AWS は、99.999% 以上の可用性を持つサービスを含め、広い範囲の可用性設計目標を持つアプリケーションを開発する豊富な経験を有しています。AWSソリューションアーキテクト (SA) は、お客様の可用性目標に対する適切な設計を支援します。設計プロセスの初期段階から AWS を導入することで、当社が可用性目標の達成を支援する能力も高まります。可用性の計画は、実際のワークロードが始動する直前にだけ立てるものではありません。運用上の経験を積み、実際のイベントから学び、さまざまな種類の障害に耐えながら、設計を改良し続けることも必要です。これにより、実装を改善するための適切な作業を行うことができます。

ワークロードに求められる可用性のニーズは、ビジネスのニーズと重要度に合わせる必要があります。まず、RTO、RPO、および可用性を定義し、ビジネスにとって重要なことのフレームワークを明確にすることで、各ワークロードを評価できます。このようなアプローチでは、ワークロードの実装に関与する担当者が、フレームワークと、ワークロードがビジネスニーズに与える影響を理解している必要があります。

基礎

基盤となる要件は、単一のワークロードまたはプロジェクトの範囲を超える要件です。システムを設計する前に、信頼性に影響を与える基本的な要件を満たしておく必要があります。例えば、データセンターへの十分なネットワーク帯域幅が必要です。

オンプレミス環境では、依存関係により、このような要件が長いリードタイムの原因となる可能性があるため、初期計画に組み込んでおく必要があります。ただし、AWS では、このような基本的な要件のほとんどが既に組み込まれており、必要に応じて変更できます。クラウドは、ほぼ制限を持たないように設計されています。つまり、十分なネットワーク性能とコンピューティング性能の要件を満たすのは AWS の責任であり、お客様はリソースのサイズと割り当てを需要に応じて自由に変更できます。

以下のセクションでは、このような信頼性について考慮すべき点に焦点を当てたベストプラクティスについて説明します。

トピック

- サービスクォータと制約を管理する
- ネットワークトポロジを計画する

サービスクォータと制約を管理する

クラウドベースのワークロードアーキテクチャには、サービスクォータ (サービスの制限とも言います) があります。これらのクォータは、サービスを不正使用されないようにするために、必要以上のリソースを誤ってプロビジョニングすることを防ぎ、API オペレーションのリクエストレートを制限するためのものです。また、光ファイバーケーブルにビットをプッシュダウンできる速度や、物理ディスク上のストレージの量などのリソースの制限もあります。

AWS Marketplace のアプリケーションを使用している場合、アプリケーションの制限を理解する必要があります。サードパーティーのウェブサービスや SaaS を使用している場合は、これらの制限も認識しておく必要があります。

ベストプラクティス

- REL01-BP01 サービスクォータと制約を認識する
- REL01-BP02 アカウントおよびリージョンをまたいでサービスクォータを管理する
- REL01-BP03 アーキテクチャを通じて、固定サービスクォータと制約に対応する

- REL01-BP04 クォータをモニタリングおよび管理する
- REL01-BP05 クォータ管理を自動化する
- REL01-BP06 フェイルオーバーに対応するために、現在のクォータと最大使用量の間に十分なギャップがあることを確認する

REL01-BP01 サービスクォータと制約を認識する

デフォルトのクォータに注意して、ワークロードアーキテクチャに対するクォータ引き上げリクエストを管理しましょう。ディスクやネットワークなど、どのクラウドリソースの制約が潜在的に大きな影響を与えるかを知っておきましょう。

期待される成果: お客様は、主要メトリクスのモニタリング、インフラストラクチャのレビュー、自動修復の手順などに、適切なガイドラインを設定して、サービスクォータや制約がサービスの低下や中断につながるレベルに達していないことを確認することによって、AWS アカウントでのサービスの低下や中断を防ぐことができます。

一般的なアンチパターン:

- 使用しているサービスのハードまたはソフト上のクォータや制限を理解せずにワークロードをデプロイする。
- 必要なクォータの分析と再設定、またはサポートへの事前連絡をせずに、代替ワークロードをデプロイする。
- クラウドサービスには制限がなく、料金、制限、回数、数量を気にせずにサービスを使用できると考えている。
- クォータは自動的に増加すると考えている。
- クォータリクエストのプロセスやスケジュールを知らない。
- クラウドサービスのデフォルトのクォータが、リージョン間で比較する各サービスで同一だと考えている。
- サービスの制約は破ることが可能で、システムによりリソースの制約を超えて自動スケールまたは制限の増加が追加されると考えている。
- リソースの使用率にストレスをかけるためにピーク時トラフィックでアプリケーションをテストしていない。
- 必要なリソースサイズを分析せずにリソースをプロビジョニングする。
- 実際に必要な分または予想されるピークを遥かに超えるリソースタイプを選択することでキャパシティを過剰プロビジョニングする。

- 新規顧客イベントや新技術のデプロイに先駆けて、新しいレベルのトラフィックのキャパシティ要件を評価していない。

このベストプラクティスを活用するメリット: サービスクォータとリソースの制約をモニタリングおよび自動管理することで、エラーを予防できます。ベストプラクティスに従っていないと、顧客のサービスにおけるトラフィックパターンの変化により、停止または機能低下が起こる可能性があります。これらの値をすべてのリージョンとすべてのアカウントでモニタリングし管理することで、有害なイベントや計画外のイベントにおける、アプリケーションの回復力が向上します。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

Service Quotas は、250 を超える AWS のサービスのクォータを一元的に管理できる AWS のサービスです。クォータ値を検索できるほか、Service Quotas コンソールから、または AWS SDK を使用して、クォータの増加をリクエストし追跡することもできます。AWS Trusted Advisor には、特定のサービスの特定の要素の使用状況およびクォータを表示する、サービスクォータチェックが用意されています。サービスごとのデフォルトのサービスクォータは、それぞれのサービスの AWS ドキュメントにも記載されています (例: Amazon VPC クォータを参照のこと)。

スロットルされた API のレート制限など、一部のサービス上の制限は、Amazon API Gateway 内で使用量プランを変更することで設定できます。それぞれのサービス上の構成として設定される制限には、プロビジョンド IOPS、割り当てられた Amazon RDS ストレージ、Amazon EBS ボリューム割り当てなどがあります。Amazon Elastic Compute Cloud には独自のサービスの制限ダッシュボードがあり、インスタンス、Amazon Elastic Block Store、Elastic IP アドレスの制限を管理するのに役立ちます。サービスクォータがアプリケーションのパフォーマンスに影響を及ぼし、ニーズに合わせて調整できないような事例が発生した場合は、サポートに連絡し、緩和策の有無についてお問い合わせください。

サービスクォータはその性質上、リージョン固有である場合も、グローバルである場合もあります。クォータに達している AWS サービスを使用すると、通常の使用で予想どおりに動作しないことや、サービスの停止や機能低下を招くことがあります。例えば、あるサービスクォータは特定のリージョンで使用される DL Amazon EC2 の数を制限していますが、Auto Scaling グループ (ASG) を使用したトラフィックスケーリングイベントの最中に、この制限に達する場合があります。

各アカウントのサービスクォータについて、使用量を定期的に評価し、そのアカウントにおける適切なサービス制限を判断する必要があります。このようなサービスクォータは、意図せず必要以上のリソースをプロビジョニングすることを防止する運用上のガードレールとして存在しています。

また、API オペレーションにおけるリクエスト率を制限し、不正使用からサービスを保護する役目も持っています。

サービスの制約は、サービスクォータとは異なります。サービスの制約は、リソースタイプごとに決まっている特定のリソースの制限を表します。これらはストレージ容量だったり (例えば gp2 のサイズ制限は 1 GB ~ 16 TB)、ディスクスループットだったりします。制限に達する可能性のある使用量について、リソースタイプの制約を監督し定期的に評価することが不可欠です。予期せず制約に達した場合、アカウントのアプリケーションまたはサービスが機能低下または停止する恐れがあります。

サービスクォータがアプリケーションのパフォーマンスに影響を及ぼし、ニーズに合わせて調整できないような事例がある場合は、サポートに連絡し、緩和策の有無についてお問い合わせください。修正されたクォータの調整に関する詳細は、「REL01-BP03 アーキテクチャを通じて、固定サービスクォータと制約に対応する」を参照してください。

Service Quotas のモニタリングや管理に役立つ AWS サービスやツールは多数あります。これらのサービスやツールは、クォータレベルの自動または手動チェックの提供に活用するものです。

- AWS Trusted Advisor は、いくつかのサービスの一部の側面に対する利用状況とクォータを表示する、サービスクォータチェックを提供しています。クォータに迫っているサービスの特定に役立ちます。
- AWS マネジメントコンソールでは、サービスのクォータ値の表示、管理、新しいクォータのリクエスト、クォータリクエストのステータスのモニタリング、クォータの履歴の表示を行う手段を提供しています。
- AWS CLI および CDK は、サービスクォータのレベルと使用量を自動で管理およびモニタリングする、プログラムによる手段を提供します。

実装手順

Service Quotas の場合:

- AWS Service Quotas をレビューします。
- 既存のサービスクォータを把握するには、使用されているサービス (IAM Access Analyzer など) を確認します。サービスクォータで制御されている AWS のサービスは約 250 あります。次に、各アカウントとリージョンで使用されている可能性のある特定のサービスクォータ名を特定します。各リージョンには約 3,000 のサービスクォータ名があります。
- このクォータ分析を AWS Config で強化して、AWS アカウントで使用されているすべての AWS リソースを特定します。

- AWS CloudFormation データを使用して、使用されている AWS リソースを特定します。AWS マネジメントコンソール で作成された、または `list-stack-resources` AWS CLI コマンドを使用して作成されたリソースを確認します。テンプレート自体にデプロイされるように設定されたリソースも確認できます。
- デプロイコードを見て、ワークロードに必要なすべてのサービスを決定します。
- 適用するサービスクォータを決定します。Trusted Advisor および Service Quotas からプログラムでアクセスできる情報を使用します。
- サービスクォータが制限に近づいたか達した場合にアラートで知らせる、自動モニタリングの方法を作成します (「REL01-BP02 アカウントおよびリージョンをまたいでサービスクォータを管理する」および「REL01-BP04 クォータをモニタリングおよび管理する」を参照のこと)。
- サービスクォータが 1 つのリージョンで変更されたときに同一アカウント内の他のリージョンで変更されたかどうかを確認する、自動化されたプログラムによる方法を作成します (「REL01-BP02 アカウントおよびリージョンをまたいでサービスクォータを管理する」および「REL01-BP04 クォータをモニタリングおよび管理する」を参照のこと)。
- アプリケーションログやメトリクスのスキャンを自動化して、クォータまたはサービス制約のエラーが発生しているか判断します。エラーが発生している場合は、モニタリングシステムにアラートを送信します。
- 特定のサービスでクォータの引き上げが必要であると判断された場合に、クォータに必要とされる変更を測定するための手順を作成します (「REL01-BP05 クォータ管理を自動化する」を参照のこと)。
- サービスクォータの変更をリクエストするプロビジョニングおよび承認ワークフローを作成します。これには、リクエストが拒否または一部承認された場合の例外ワークフローを含めます。
- 本稼働環境またはロード済み環境にロールアウトする前に、新しい AWS サービスをプロビジョニングして使用するにあたって、サービスクォータをレビューするエンジニアリング手段を作成します (例: ロードテストアカウント)。

サービス制約の場合:

- 読み取りがリソースの制約に近づいているリソースについてアラートを発報するモニタリングおよびメトリクス手段を作成します。必要に応じて CloudWatch を活用してメトリクスまたはログをモニタリングします。
- 制約のある各リソースについて、アプリケーションまたはシステムにとって有意なアラートのしきい値を設定します。

- 制約が使用量に近い場合、リソースタイプを変更するワークフローまたはインフラストラクチャ管理手順を作成します。このワークフローには、ベストプラクティスとして負荷テストを含め、新しいタイプが新しい制約のある適切なリソースタイプであることを検証します。
- 既存の手順やプロセスを使用して、特定したリソースを推奨の新しいリソースタイプに移行します。

リソース

関連するベストプラクティス:

- REL01-BP02 アカウントおよびリージョンをまたいでサービスクォータを管理する
- REL01-BP03 アーキテクチャを通じて、固定サービスクォータと制約に対応する
- REL01-BP04 クォータをモニタリングおよび管理する
- REL01-BP05 クォータ管理を自動化する
- REL01-BP06 フェイルオーバーに対応するために、現在のクォータと最大使用量の間に十分なギャップがあることを確認する
- REL03-BP01 ワークロードをセグメント化する方法を選択する
- REL10-BP01 複数の場所にワークロードをデプロイする
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する
- REL11-BP03 すべてのレイヤーの修復を自動化する
- REL12-BP04 カオスエンジニアリングを使用して回復力をテストする

関連ドキュメント:

- AWS Well-Architected Framework の信頼性の柱: 可用性
- AWS Service Quotas (旧称: サービスの制限)
- AWS Trusted Advisor ベストプラクティスチェック (「Service Limits」セクションを参照)
- AWS Answers の AWS Limit Monitor
- Amazon EC2 サービスの制限
- Service Quotas とは
- How to Request Quota Increase
- Service endpoints and quotas

- Service Quotas User Guide
- のクォータモニタAWS
- AWS 障害分離境界
- 冗長性を実装した可用性
- AWS for Data
- 継続的インテグレーションとは?
- 継続的デリバリーとは?
- APN Partner: partners that can help with configuration management
- における Account-per-Tenant 型 SaaS 環境のライフサイクル管理AWS
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower

関連動画:

- AWS Live re:Inforce 2019 - Service Quotas
- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo

関連ツール:

- Amazon CodeGuru Reviewer
- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge
- Amazon DevOps Guru
- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager

- AWS Marketplace

REL01-BP02 アカウントおよびリージョンをまたいでサービスクォータを管理する

複数のアカウントまたはリージョンをご利用の場合は、本番ワークロードを実行するすべての環境で適切なクォータをリクエストしてください。

期待される成果: 複数のアカウントまたはリージョンにまたがる設定、またはゾーン、リージョン、アカウントのフェイルオーバーを使用したレジリエンス設計になっている設定において、サービスクォータの枯渇によってサービスやアプリケーションが影響を受けないようにします。

一般的なアンチパターン:

- 1 つの分離リージョンでのリソース使用量の増加を許可するが、他のリージョンではキャパシティを維持するメカニズムがない。
- 分離リージョン内のすべてのクォータを手動で設定する。
- 主要ではないリージョンの能力が低下している際に、将来のクォータの必要性について、回復力のあるアーキテクチャ (アクティブやパッシブなど) の影響を考慮していない。
- ワークロードが実行されているすべてのリージョンやアカウントにおいて、クォータを定期的に評価し、必要な変更を行っていない。
- 複数のリージョンやアカウントにまたがって緩和をリクエストする際に、クォータリクエストテンプレートを活用していない。
- クォータの緩和は、コンピューティング予約リクエストのように、コストに影響があるという誤った考えの下、サービスクォータを更新していない。

このベストプラクティスを活用するメリット: リージョンでのサービスが利用不可になった際に、セカンダリリージョンやアカウントで現在の負荷を処理できることを検証します。これにより、リージョンを利用できない場合に発生するエラー数や機能低下のレベルを削減できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

サービスクォータの追跡はアカウントごとに行います。特に明記されていない限り、クォータは AWS リージョン 固有です。テストと開発が妨げられないように、本番環境に加えて、該当するすべ

ての非本番環境でもクォータを管理します。高レベルの回復性を維持するには、サービスクォータを継続的に評価する必要があります (自動または手動で)。

アクティブ/アクティブ、アクティブ/パッシブ - ホット、アクティブ/パッシブ - コールド、アクティブ/パッシブ - パイロットライトの各アプローチを使用した設計の実装により、複数のリージョンにまたがるワークロードが増えると、すべてのリージョンおよびアカウントのクォータレベルを把握することが不可欠になってきます。サービスクォータが正しく設定されている場合、過去のトラフィックパターンが常に適した指標になるとは限りません。

同じくらい重要なのが、サービスクォータ名の制限が各リージョンで常に同じとは限らないことです。あるリージョンでは値が 5 であり、別のリージョンでは値が 10 であることもありえます。負荷時の一貫した回復力を提要するために、このようなクォータの管理は、すべての同じサービス、アカウント、リージョンを網羅する必要があります。

異なるリージョン (アクティブリージョンまたはパッシブリージョン) 間のすべてのサービスクォータの差異を調整し、これらの差異を継続的に調整するプロセスを作成します。パッシブリージョンのフェイルオーバーのテスト計画は、ピーク時のアクティブキャパシティまでスケールすることはめったにありません。つまり、ゲームデーや机上演習では、リージョン間のサービスクォータの差異を見つけられない場合があり、したがって適切な制限を維持できない場合があります。

サービスクォータのドリフトとは、特定の名前のクォータにおけるサービスクォータの制限が、すべてのリージョンではなく、1 つのリージョンで変更される状況であり、追跡と評価が非常に重要になります。トラフィックを伴う、またはトラフィックを伴う可能性があるリージョンでのクォータの変更を、考慮する必要があります。

- サービスの要件、レイテンシー、およびディザスタリカバリ (DR) 要件に基づいて、関連するアカウントとリージョンを選択します。
- 関連するすべてのアカウント、リージョン、アベイラビリティーゾーン全体のサービスクォータを特定します。制限の対象範囲はアカウントとリージョンです。これらの値を比較して差異を把握する必要があります。

実装手順

- 使用量のリスクレベルを超えている可能性がある Service Quotas の値をレビューします。AWS Trusted Advisor は 80% および 90% のしきい値で違反を警告します。
- パッシブリージョンのサービスクォータの値をレビューします (アクティブ/パッシブ設計の場合)。プライマリリージョンで障害があった場合に、負荷が正常にセカンダリリージョンで実行されることを検証します。

- サービスクォータのドリフトが同一アカウント内のリージョン間で発生し、それに伴って制限が変更された場合の評価を自動化します。
- お客様の組織単位 (OU) がサポートされている方法で構成されている場合、サービスクォータテンプレートを更新して、クォータの変更を反映し、複数のリージョンやアカウントに適用する必要があります。
  - テンプレートを作成して、リージョンをクォータの変更に関連付けます。
  - 既存のすべてのサービスクォータテンプレートをレビューして、変更が必要かどうかを確認します (リージョン、制限、アカウント)。

リソース

関連するベストプラクティス:

- REL01-BP01 サービスクォータと制約を認識する
- REL01-BP03 アーキテクチャを通じて、固定サービスクォータと制約に対応する
- REL01-BP04 クォータをモニタリングおよび管理する
- REL01-BP05 クォータ管理を自動化する
- REL01-BP06 フェイルオーバーに対応するために、現在のクォータと最大使用量の間に十分なギャップがあることを確認する
- REL03-BP01 ワークロードをセグメント化する方法を選択する
- REL10-BP01 複数の場所にワークロードをデプロイする
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する
- REL11-BP03 すべてのレイヤーの修復を自動化する
- REL12-BP04 カオスエンジニアリングを使用して回復力をテストする

関連ドキュメント:

- AWS Well-Architected Framework の信頼性の柱: 可用性
- AWS Service Quotas (旧称: サービスの制限)
- AWS Trusted Advisor ベストプラクティスチェック (「Service Limits」セクションを参照)
- AWS Answers の AWS Limit Monitor
- Amazon EC2 サービスの制限

- Service Quotas とは
- How to Request Quota Increase
- Service endpoints and quotas
- Service Quotas User Guide
- のクォータモニタAWS
- AWS 障害分離境界
- 冗長性を実装した可用性
- AWS for Data
- 継続的インテグレーションとは?
- 継続的デリバリーとは?
- APN Partner: partners that can help with configuration management
- における Account-per-Tenant 型 SaaS 環境のライフサイクル管理AWS
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower

関連動画:

- AWS Live re:Inforce 2019 - Service Quotas
- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo

関連サービス:

- Amazon CodeGuru Reviewer
- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge
- Amazon DevOps Guru

- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager
- AWS Marketplace

REL01-BP03 アーキテクチャを通じて、固定サービスクォータと制約に対応する

Service Quotas、サービスの制約、物理リソースの制限には変更できないものもあることに注意します。このような制限が信頼性に影響を及ぼさないように、アプリケーションとサービスのアーキテクチャを設計します。

例えば、ネットワーク帯域幅、サーバーレス関数呼び出しのペイロードサイズ、API Gateway のスロットルバーストレート、データベースへの同時ユーザー接続数などは変更できません。

期待される成果: アプリケーションやサービスは、通常のトラフィック状況および高トラフィック状況で期待されるとおりに動作します。アプリケーションやサービスは、リソースの固定制約またはサービスクォータの制限内で機能するように設計されています。

一般的なアンチパターン:

- 単一のサービスリソースを使用する設計を選択し、スケーリング時に障害が発生する原因となる設計上の制約があることを認識していない。
- テスト中にサービスの固定クォータに到達してしまう非現実的なベンチマークを採用している。(例: バースト制限を利用しながら長時間テストを実行する)
- 固定サービスクォータを超えてもスケールしたり変更したりできない設計を選択する。(例: 256KB のペイロードサイズの SQS)
- 高トラフィックイベント中に危険にさらされる可能性があるサービスクォータのしきい値をモニタリングしたり警告したりするために、オブザーバビリティが設計および実装されていない。

このベストプラクティスを活用するメリット: 予測されるあらゆるサービス負荷レベルで、アプリケーションが中断や低下することなく実行されることを確認できます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

ソフトサービスクォータ、つまりより高い容量ユニットに置き換えられるリソースとは異なり、AWS サービスの固定クォータは変更できません。つまり、アプリケーションの設計で使用する場合、このようなタイプのすべての AWS サービスの容量のハード制限を評価する必要があります。

ハード制限は Service Quotas コンソールに表示されます。列に が表示されている場合ADJUSTABLE = No、サービスにはハード制限があります。ハード制限は、一部のリソース設定ページにも表示されます。例えば、Lambda には調整できない特定のハード制限があります。

例としては、Lambda 関数で実行する Python アプリケーションを設計する場合、Lambda が 15 分以上実行される可能性があるかを判断するためにアプリケーションを評価する必要があります。コードがこのサービスクォータ制限を超過して実行される可能性がある場合は、代替テクノロジーや設計を検討する必要があります。本番環境へのデプロイ後にこの制限に達すると、修正されるまでアプリケーションの機能が低下し、中断することになります。ソフトクォータとは異なり、このような制限は、重大度 1 の緊急事態が発生した場合でも、変更できません。

アプリケーションがテスト環境にデプロイされたら、ハード制限に到達できるかどうかを確認する戦略を行使する必要があります。ストレステスト、負荷テスト、カオステストは、導入テスト計画の一環とする必要があります。

実装手順

- 使用できる AWS サービスの完全なリストをアプリケーションの設計段階で確認します。
- これらすべてのサービスについて、ソフトクォータ制限とハードクォータ制限を確認します。すべての制限が Service Quotas コンソールに表示されているとは限りません。サービスによっては、このような制限について、別の場所で説明されています。
- アプリケーションを設計する際は、ビジネス上の成果、ユースケース、依存するシステム、可用性目標、ディザスタリカバリオブジェクトなど、ワークロードのビジネスとテクノロジーの推進要因を確認します。ビジネスとテクノロジーの推進要因に従って、ワークロードに適した分散システムを特定するプロセスを進めます。
- リージョン全体とアカウント全体にわたるサービス負荷を分析します。ハード制限の多くは、サービスのリージョンに基づいていますが、アカウントに基づく制限もあります。
- ゾーン障害時とリージョン障害時のリソース使用状況について、回復力あるアーキテクチャを分析します。「アクティブ/アクティブ」、「アクティブ/パッシブ – ホット」、「アクティブ/パッシブ – コールド」、「アクティブ/パッシブ – パイロットライト」のアプローチを使用するマルチリージョン設計を進めると、このような障害ケースはより高い使用状況につながり、ハード制限に達するユースケースを生み出す可能性が出てきます。

リソース

関連するベストプラクティス:

- REL01-BP01 サービスクォータと制約を認識する
- REL01-BP02 アカウントおよびリージョンをまたいでサービスクォータを管理する
- REL01-BP04 クォータをモニタリングおよび管理する
- REL01-BP05 クォータ管理を自動化する
- REL01-BP06 フェイルオーバーに対応するために、現在のクォータと最大使用量の間に十分なギャップがあることを確認する
- REL03-BP01 ワークロードをセグメント化する方法を選択する
- REL10-BP01 複数の場所にワークロードをデプロイする
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する
- REL11-BP03 すべてのレイヤーの修復を自動化する
- REL12-BP04 カオスエンジニアリングを使用して回復力をテストする

関連ドキュメント:

- AWS Well-Architected Framework の信頼性の柱: 可用性
- AWS Service Quotas (旧称: サービスの制限)
- AWS Trusted Advisor ベストプラクティスチェック (「Service Limits」セクションを参照)
- AWS Answers の AWS Limit Monitor
- Amazon EC2 サービスの制限
- Service Quotas とは
- How to Request Quota Increase
- Service endpoints and quotas
- Service Quotas User Guide
- のクォータモニタAWS
- AWS 障害分離境界
- 冗長性を実装した可用性
- AWS for Data
- 継続的インテグレーションとは?

- 継続的デリバリーとは?
- APN Partner: partners that can help with configuration management
- における Account-per-Tenant 型 SaaS 環境のライフサイクル管理AWS
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower
- Actions, resources, and condition keys for Service Quotas

関連動画:

- AWS Live re:Inforce 2019 - Service Quotas
- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo
- AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small

関連ツール:

- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge
- Amazon DevOps Guru
- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager
- AWS Marketplace

REL01-BP04 クォータをモニタリングおよび管理する

予想される使用量を評価し、クォータを必要に応じて引き上げて、使用量を予定どおり増やせるようにします。

期待される成果: 管理およびモニタリングするアクティブで自動化されたシステムが導入されます。このような運用ソリューションを採用すると、使用量がクォータのしきい値に近づいているかどうかを確認することができます。クォータの変更が要請されると、これらは積極的に修正されます。

一般的なアンチパターン:

- サービスクォータのしきい値をチェックするようにモニタリングを設定していない。
- ハード制限の値は変更できないにもかかわらず、モニタリングを設定していない。
- ソフトクォータの変更を要請して確保するのに必要な時間は即時または短期間であると想定している。
- サービスクォータに近づいた際のアラームは設定していても、アラートの対応方法に関するプロセスがない。
- AWS Service Quotas でサポートされているサービスのアラームのみを設定し、他の AWS サービスのモニタリングを行っていない。
- 「アクティブ/アクティブ」、「アクティブ/パッシブ – ホット」、「アクティブ/パッシブ – コールド」、「アクティブ/パッシブ – パイロットライト」のアプローチなど、複数リージョンの回復力ある設計のクォータ管理を考慮していない。
- リージョン間のクォータの違いを評価していない。
- 特定のクォータ引き上げ要請について、すべてのリージョンのニーズを評価していない。
- マルチリージョンクォータ管理向けのテンプレートを活用していない。

Benefits of establishing this best practice: AWS Service Quotas を自動的に追跡し、これらのクォータに対する使用状況をモニタリングすることで、クォータ制限に近づいていることを確認できます。このモニタリングデータを使用して、クォータの枯渇による低下を制限することもできます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

サポートされているサービスについては、評価してアラートまたはアラームを送信できるさまざまなサービスを設定することで、クォータをモニタリングできます。これにより、使用状況のモニタリングがサポートされ、クォータに近づいていることのアラートが送信されます。このアラームは、AWS Config、Lambda 関数、Amazon CloudWatch、または AWS Trusted Advisor からトリガーできます。CloudWatch Logs のメトリクスフィルターを使用して、ログのパターンを検索および抽出し、使用量がクォータのしきい値に近づいているかどうかを判断することもできます。

実装手順

モニタリング:

- 現在のリソース消費 (バケットやインスタンスなど) を把握します。現在のリソース消費を収集するには、Amazon EC2 `DescribeInstances` API などのサービス API オペレーションを使用します。
- 以下を使用して、サービスに不可欠で適用できる現在のクォータをキャプチャします。
  - AWS Service Quotas
  - AWS Trusted Advisor
  - AWS ドキュメント
  - AWS サービス固有のページ
  - AWS Command Line Interface (AWS CLI)
  - AWS Cloud Development Kit (AWS CDK)
- AWS Service Quotas を使用します。これは、250 を超える AWS のサービスのクォータを一元的に管理するのに役立つ AWS のサービスです。
- さまざまなしきい値で現在のサービス制限をモニタリングするには、Trusted Advisor サービス制限を使用します。
- リージョンの使用状況の増加をチェックするには、サービスクォータ履歴 (コンソールまたは AWS CLI) を使用します。
- 各リージョンと各アカウントのサービスクォータの変化を比較して、必要に応じて同等性を確立します。

管理:

- 自動化: AWS Config カスタムルールを設定し、リージョン間でサービスクォータをスキャンして、違いを比較します。
- 自動化: リージョン全体にわたるサービスクォータをスキャンするスケジュールされた Lambda 関数を設定して、違いを比較します。
- 手動: リージョン全体にわたるサービスクォータをスキャンするために、AWS CLI、API、または AWS コンソールを介してサービスクォータをスキャンして、違いを比較します。違いについてのレポートを作成します。
- リージョン間でクォータの違いが確認された場合は、必要に応じてクォータの変更を要請します。
- すべての変更の結果を確認します。

リソース

関連するベストプラクティス:

- REL01-BP01 サービスクォータと制約を認識する
- REL01-BP02 アカウントおよびリージョンをまたいでサービスクォータを管理する
- REL01-BP03 アーキテクチャを通じて、固定サービスクォータと制約に対応する
- REL01-BP05 クォータ管理を自動化する
- REL01-BP06 フェイルオーバーに対応するために、現在のクォータと最大使用量の間に十分なギャップがあることを確認する
- REL03-BP01 ワークロードをセグメント化する方法を選択する
- REL10-BP01 複数の場所にワークロードをデプロイする
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する
- REL11-BP03 すべてのレイヤーの修復を自動化する
- REL12-BP04 カオスエンジニアリングを使用して回復力をテストする

関連ドキュメント:

- AWS Well-Architected Framework の信頼性の柱: 可用性
- AWS Service Quotas (旧称: サービスの制限)
- AWS Trusted Advisor ベストプラクティスチェック (「Service Limits」セクションを参照)
- AWS Answers の AWS Limit Monitor
- Amazon EC2 サービスの制限
- Service Quotas とは
- How to Request Quota Increase
- Service endpoints and quotas
- Service Quotas User Guide
- AWS のクォータモニタ
- AWS 障害分離境界
- 冗長性を実装した可用性
- AWS for Data
- 継続的インテグレーションとは?

- 継続的デリバリーとは?
- APN パートナー: 設定管理を支援できるパートナー
- AWS における Account-per-Tenant 型 SaaS 環境のライフサイクル管理
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower
- 「Service Quotas のアクション、リソース、および条件キー」

関連動画:

- AWS Live re:Inforce 2019 - Service Quotas
- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo
- AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small

関連ツール:

- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge
- Amazon DevOps Guru
- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager
- AWS Marketplace

REL01-BP05 クォータ管理を自動化する

Service Quotas (AWS サービスの制限とも呼ばれます) は、AWS アカウント のサービスリソースの最大数です。各 AWS サービスは、一連のクォータとそのデフォルト値を定義します。必要なすべて

のリソースへのワークロードアクセスを提供するには、Service Quotas の値を増やす必要がある場合があります。

AWS リソースのワークロード消費の増加は、ワークロードの安定性を脅かし、クォータを超えた場合にユーザーエクスペリエンスに影響を与える可能性があります。ワークロードが制限に達したときに警告するツールを実装し、クォータ引き上げリクエストを自動的に作成することを検討してください。

期待される成果: クォータは、各 AWS アカウント およびリージョンで実行されているワークロードに対して適切に設定されます。

一般的なアンチパターン:

- ワークロード要件を満たすためにクォータを適切に考慮して調整できていない。
- スプレッドシートなど、古くなる可能性のある方法を使用して、クォータと使用状況を追跡している。
- 定期的なスケジュールのサービス制限のみを更新している。
- 既存のクォータを確認し、必要に応じて Service Quotas の引き上げをリクエストする運用プロセスが組織にない。

このベストプラクティスを活用するメリット:

- ワークロードの耐障害性の向上: AWS リソースクォータを超えるとエラーが発生するのを防ぎます。
- ディザスタリカバリの簡素化: プライマリリージョンに構築された自動クォータ管理メカニズムは、別の AWS リージョン の DR セットアップ中に再利用できます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

AWS Service Quotas コンソール、AWS Command Line Interface (AWS CLI)、AWS SDK などのメカニズムを通じて、現在のクォータを表示し、進行中のクォータ消費を追跡します。構成管理データベース (CMDB) および IT サービス管理 (ITSM) システムを AWS Service Quotas API と統合することもできます。

クォータ使用量が定義されたしきい値に達した場合に自動アラートを生成し、アラートを受信したときにクォータ引き上げリクエストを送信するプロセスを定義します。基盤となるワークロードがビジ

ネスにとって重要な場合は、クォータ引き上げリクエストを自動化できますが、成長フィードバックループなどのランナウェイアクションのリスクを回避するために、自動化を慎重にテストします。

クォータの増加が小さくなると、多くの場合、自動的に承認されます。より大きなクォータリクエストは、AWS サポートによって手動で処理する必要があり、確認と処理にさらに時間がかかる場合があります。複数のリクエストや大幅な増加リクエストを処理するには、追加の時間が必要です。

実装手順

- Service Quotas の自動モニタリングを実装し、ワークロードのリソース使用率の増加がクォータの制限に近づいた場合にアラートを発行します。例えば、AWS のクォータモニタは、Service Quotas の自動モニタリングを提供できます。このツールは AWS Organizations と統合され、Cloudformation StackSets を使用してデプロイされるため、新しいアカウントは作成時に自動的にモニタリングされます。
- Service Quotas リクエストテンプレートや AWS Control Tower などの機能を使用して、新しいアカウントの Service Quotas のセットアップを簡素化します。
- 現在の Service Quotas のダッシュボードをすべての AWS アカウント およびリージョンで作成し、クォータを超えないように必要に応じて参照します。Cloud Intelligence Dashboards の一部である Trusted Advisor Organizational (TAO) Dashboard を使用すると、このようなダッシュボードをすぐに使い始めることができます。
- サービス制限の拡大を追跡します。Consolidated Insights from Multiple Accounts (CIMA) は、すべてのリクエストの組織レベルのビューを提供します。
- 非本番環境アカウントで低いクォータしきい値を設定して、アラート生成とクォータ増加リクエストの自動化をテストします。これらのテストを本番稼働アカウントで実行しないでください。

リソース

関連するベストプラクティス:

- OPS10-BP07 イベントへの対応を自動化する

関連ドキュメント:

- APN パートナー: 設定管理を支援できるパートナー
- AWS Marketplace: 制限の追跡に役立つ CMDB 製品
- AWS Service Quotas (旧称: サービスの制限)
- AWS Trusted Advisor ベストプラクティスチェック (「Service Limits」セクションを参照)

- AWS のクォータモニタソリューション - AWS ソリューション
- Service Quotas とは
- Service Quotas リクエストテンプレートとは

関連動画:

- AWS Live re:Inforce 2019 - Service Quotas
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower

関連ツール:

- AWS のクォータモニタ

REL01-BP06 フェイルオーバーに対応するために、現在のクォータと最大使用量の間に十分なギャップがあることを確認する

この記事では、リソースクォータと使用量の間にスペースを維持する方法と、それが組織にどのように役立つかについて説明します。リソースの使用が終了した後も、そのリソースの使用クォータは引き続き考慮される可能性があります。これにより、リソースに障害が発生したり、アクセスできなくなったりする可能性があります。障害が発生したリソースまたはアクセスできないリソースと、代替のリソースの合計リソース数がクォータ内に収まることを確認して、リソースの傷害を防ぎます。このギャップを算出する際は、ネットワーク障害、アベイラビリティーゾーンの不具合、またはリージョン規模の不具合などのユースケースを考慮します。

期待される成果: 現在のサービスのしきい値を使用して、規模を問わず、リソースやリソースへのアクセスで障害に対応できます。リソースプランニングでは、ゾーン障害、ネットワーク障害、さらにはリージョン規模の不具合が考慮されています。

一般的なアンチパターン:

- フェイルオーバーシナリオを考慮せずに、現在のニーズに基づいてサービスクォータを設定する。
- ピーク時のサービスクォータの計算時に静的安定性の原則を考慮しない。
- 各リージョンに求められる合計クォータを計算する際に、アクセスできないリソースの可能性を考慮しない。
- AWS サービス障害分離境界と、予測される異常な使用パターンを考慮していないサービスがある。

このベストプラクティスを活用するメリット: サービス中断イベントがアプリケーションの可用性に影響を与えるような場合、クラウドを使用すると、このようなイベントを軽減または復旧するための戦略を実装できます。戦略の例としては、サービスの制限を使い果たさずに、アクセスできないリソースを置き換える追加リソースを作成してフェイルオーバー条件に対応することが挙げられます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

クォータ制限を評価する際は、何らかの劣化が原因で発生する可能性のあるフェイルオーバーのケースを検討します。次のフェイルオーバーケースを検討してください。

- 中断された、またはアクセスできない VPC。
- アクセスできないサブネット。
- リソースのアクセシビリティに影響するアベイラビリティーゾーンの低下。
- ネットワークルートまたは受信ポイントと送信ポイントがブロックされたり、変更されたりしている。
- リソースへのアクセスに影響を及ぼすレベルまで低下したリージョン。
- リージョンまたはアベイラビリティーゾーンでの障害の影響を受けるリソースのサブセット。

ビジネスへの影響は異なる可能性があり、フェイルオーバーの決定は状況ごとに異なります。アプリケーションまたはサービスのフェイルオーバーを決定する前に、フェイルオーバーロケーションとリソースのクォータでリソース容量計画に対処します。

各サービスのクォータを確認するときは、アクティビティのピークが通常のピークよりも高いことを考慮してください。このようなピークは、ネットワークまたはアクセス許可のためにアクセス不可能ですがアクティブなリソースに関連している可能性があります。終了していないアクティブなリソースは、サービスクォータ制限に対してカウントされます。

実装手順

- フェイルオーバーまたはアクセスできなくなった場合に対応するため、サービスクォータと最大使用量の間に十分なギャップを維持します。
- サービスクォータを決定します。一般的なデプロイパターン、可用性要件、消費の増加を考慮します。
- 必要に応じてクォータの引き上げをリクエストします。クォータ引き上げリクエストの待機時間を予測します。

- 信頼性の要件 (「9 の数」としても知られる) を決定します。
- コンポーネント、アベイラビリティーゾーン、リージョンの喪失などの潜在的な障害シナリオを理解します。
- デプロイ手法 (例えば、Canary、ブルー/グリーン、レッド/ブラック、ローリングなど) を確立します。
- 現在のクォータ制限に適切なバッファを含めます。バッファの例は 15% です。
- 必要に応じて、静的安定性 (ゾーンおよびリージョン) の計算を含めます。
- 消費の増加を計画し、消費の傾向を監視します。
- 最も重要なワークロードへの静的安定性の影響を考慮します。すべてのリージョンとアベイラビリティーゾーンで静的に安定性のあるシステムに相当するリソースを評価します。
- オンデマンドキャパシティ予約を使用して、フェイルオーバーが発生する前に容量をスケジュールすることを検討します。これは実装すると、フェイルオーバー発生時、最も重要なビジネススケジュール中に、適切な量および種類のリソースを取得するうえで予測できるリスクの軽減に役立つ戦略です。

リソース

関連するベストプラクティス:

- REL01-BP01 サービスクォータと制約を認識する
- REL01-BP02 アカウントおよびリージョンをまたいでサービスクォータを管理する
- REL01-BP03 アーキテクチャを通じて、固定サービスクォータと制約に対応する
- REL01-BP04 クォータをモニタリングおよび管理する
- REL01-BP05 クォータ管理を自動化する
- REL03-BP01 ワークロードをセグメント化する方法を選択する
- REL10-BP01 複数の場所にワークロードをデプロイする
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する
- REL11-BP03 すべてのレイヤーの修復を自動化する
- REL12-BP04 カオスエンジニアリングを使用して回復力をテストする

関連ドキュメント:

- AWS Well-Architected Framework の信頼性の柱: 可用性

- AWS Service Quotas (旧称: サービスの制限)
- AWS Trusted Advisor ベストプラクティスチェック (「Service Limits」セクションを参照)
- AWS Answers の AWS Limit Monitor
- Amazon EC2 サービスの制限
- Service Quotas とは
- How to Request Quota Increase
- Service endpoints and quotas
- Service Quotas User Guide
- AWS のクォータモニタ
- AWS 障害分離境界
- 冗長性を実装した可用性
- AWS for Data
- 継続的インテグレーションとは?
- 継続的デリバリーとは?
- APN Partner: partners that can help with configuration management
- AWS における Account-per-Tenant 型 SaaS 環境のライフサイクル管理
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower
- Actions, resources, and condition keys for Service Quotas

関連動画:

- AWS Live re:Inforce 2019 - Service Quotas
- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo
- AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small

関連ツール:

- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge
- Amazon DevOps Guru
- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager
- AWS Marketplace

ネットワークトポロジを計画する

多くの場合、ワークロードは複数の環境に存在します。その中には、複数のクラウド環境 (パブリックアクセスとプライベートの両方) や、場合によっては既存のデータセンターのインフラストラクチャが含まれます。計画する際は、システム内およびシステム間の接続、パブリック IP アドレスの管理、プライベート IP アドレスの管理、ドメイン名解決といったネットワークに関する項目も考慮に含めなければなりません。

IP アドレスベースのネットワークを使用するシステムを構築する際は、予想される障害を見越してネットワークトポロジとアドレス指定を考慮し、さらに将来の成長と他のシステムやネットワークと統合できる余地を残しておくように計画する必要があります。

Amazon Virtual Private Cloud (Amazon VPC) では、AWS クラウドのプライベートな隔離されたセクションをプロビジョニングすることで、仮想ネットワーク内で AWS リソースを起動できます。

ベストプラクティス

- REL02-BP01 ワークロードのパブリックエンドポイントに高可用性ネットワーク接続を使用する
- REL02-BP02 クラウド環境とオンプレミス環境のプライベートネットワーク間の冗長接続をプロビジョニングする
- REL02-BP03 拡張性と可用性を考慮した IP サブネットの割り当てを確実に行う
- REL02-BP04 多対多メッシュよりもハブアンドスポークトポロジを優先する
- REL02-BP05 接続されているすべてのプライベートアドレススペースにおいて、重複しないプライベート IP アドレス範囲を指定する

REL02-BP01 ワークロードのパブリックエンドポイントに高可用性ネットワーク接続を使用する

ワークロードのパブリックエンドポイントに高可用性ネットワーク接続を構築すると、接続の喪失によるダウンタイムを低減し、ワークロードの可用性と SLA を向上できます。これを実現するには、可用性の高い DNS、コンテンツ配信ネットワーク、API ゲートウェイ、負荷分散、またはリバースプロキシを使用します。

期待される成果: パブリックエンドポイントの高可用性ネットワーク接続を計画、構築、運用化することが重要です。接続が失われたためにワークロードにアクセスできなくなった場合、ワークロードが実行中で利用可能であっても、顧客はシステムがダウンしているとみなします。ワークロードのパブリックエンドポイントに対する可用性と回復力に優れたネットワーク接続と、ワークロード自体の回復力のあるアーキテクチャを組み合わせることで、可能な限り最高レベルの可用性とサービスレベルを顧客に提供できます。

AWS Global Accelerator、Amazon CloudFront、Amazon API Gateway、AWS Lambda 関数 URL、AWS AppSync API、Elastic Load Balancing (ELB) はすべて、高可用性のパブリックエンドポイントを提供します。Amazon Route 53 は、ドメイン名解決のための高可用性 DNS サービスを提供し、パブリックエンドポイントアドレスを解決できることを確認できます。

また、ロードバランシングとプロキシ処理のために、AWS Marketplace のソフトウェアアプライアンスを評価することもできます。

一般的なアンチパターン:

- 高可用性に向けて DNS とネットワーク接続を計画せず、高可用性ワークロードを設計する。
- 個別のインスタンスまたはコンテナでパブリックインターネットアドレスを使用し、DNS 経由でそれらのアドレスへの接続を管理する。
- サービスの検索に、ドメイン名ではなく、IP アドレスを使用する。
- パブリックエンドポイントへの接続が失われるシナリオをテストしていない。
- ネットワークスループットのニーズと分散パターンを分析していない。
- ワークロードのパブリックエンドポイントへのインターネットにおけるネットワーク接続が中断される可能性を考慮したシナリオをテストおよび計画していない。
- 大規模な地理的領域にコンテンツ (ウェブページ、静的アセット、またはメディアファイル) を提供し、コンテンツ配信ネットワークを使用しない。
- 分散サービス拒否 (DDoS) 攻撃に備えた計画をしていない。DDoS 攻撃は、正当なトラフィックを遮断し、ユーザーの可用性を低下させるリスクがあります。

このベストプラクティスを活用するメリット: 可用性と回復力に優れたネットワーク接続を設計することで、ユーザーはワークロードにアクセスして利用できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

パブリックエンドポイントへの高可用性ネットワーク接続を構築する際の中核となるのが、トラフィックのルーティングです。トラフィックがエンドポイントに到達できることを確認するには、ドメイン名を DNS が対応する IP アドレスに解決することができる必要があります。ドメインの DNS レコードを管理するには、Amazon Route 53 などの高可用性かつスケーラブルなドメインネームシステム (DNS) を使用します。Amazon Route 53 が提供するヘルスチェックも利用できます。ヘルスチェックは、アプリケーションが到達可能かつ利用可能で、機能していることを確認します。ヘルスチェックは、ウェブページや特定の URL を要求するなどのユーザーの動作を模倣するように設定できます。障害が発生した場合、Amazon Route 53 は DNS 解決リクエストに応答し、トラフィックを正常なエンドポイントのみに転送します。Amazon Route 53 が提供する位置情報 DNS およびレイテンシーベースルーティング機能の使用を検討することもできます。

ワークロード自体の可用性が高いことを確認するには、Elastic Load Balancing (ELB) を使用します。Amazon Route 53 は、トラフィックを ELB にターゲットとするために使用できます。ELB は、このトラフィックを、ターゲットのコンピューティングインスタンスに分散します。サーバーレスソリューションには、Amazon API Gateway と AWS Lambda を組み合わせて使用できます。また、複数の AWS リージョンでワークロードを実行することもできます。マルチサイトのアクティブ/アクティブパターンを使用すると、ワークロードは複数のリージョンからのトラフィックを処理できます。マルチサイトのアクティブ/パッシブパターンの場合、ワークロードはアクティブリージョンからのトラフィックを処理し、データはセカンダリリージョンにレプリケートされ、プライマリリージョンで障害が発生した場合にアクティブになります。その後、Route 53 のヘルスチェックを使用して、プライマリリージョンの任意のエンドポイントからセカンダリリージョンのエンドポイントへの DNS フェイルオーバーを制御して、ワークロードが到達可能であり、ユーザーが利用できることを確認することができます。

Amazon CloudFront は、世界中のエッジロケーションのネットワークを使用してリクエストを処理することにより、低レイテンシーかつ高データ転送速度でコンテンツを配信する、シンプルな API を提供します。コンテンツ配信ネットワークは、ユーザーの所在地に近いロケーションにあるか、近いロケーションでキャッシュされているコンテンツを提供することで、顧客にサービスを提供します。これにより、コンテンツの負荷がサーバーから CloudFront のエッジロケーションへと移動するため、アプリケーションの可用性も向上します。エッジロケーションとリージョンのエッジキャッシュは、視聴者の近くにコンテンツのキャッシュコピーを保持するため、ワークロードの取得が迅速化し、到達可能性と可用性が向上します。

ユーザーが地理的に分散しているワークロードの場合、AWS Global Accelerator を使用すると、アプリケーションの可用性とパフォーマンスを向上できます。AWS Global Accelerator は、1 つまたは複数の AWS リージョンでホストされているアプリケーションへの固定エントリポイントとして機能するエニーキャスト静的 IP アドレスを提供します。これにより、トラフィックがユーザーにできるだけ近い AWS グローバルネットワークに入ることができ、ワークロードの到達可能性と可用性が向上します。AWS Global Accelerator を使用すると、TCP、HTTP、および HTTPS ヘルスチェックを使用して、アプリケーションエンドポイントの健全性もモニタリングできます。エンドポイントの正常性または設定に変更が生じると、正常なエンドポイントへのユーザートラフィックのリダイレクトが許可され、最高のパフォーマンスと可用性がユーザーに提供されます。さらに、AWS Global Accelerator は障害を分離するように設計されており、独立したネットワークゾーンによって提供される 2 つの静的 IPv4 アドレスを使用して、アプリケーションの可用性を向上します。

DDoS 攻撃からの保護として、AWS は、AWS Shield Standard を提供しています。Shield Standard は自動的に有効にされており、SYN/UDP フラッド攻撃やリフレクション攻撃などの一般的なインフラストラクチャ (レイヤー 3 および 4) 攻撃から保護し、AWS 上のアプリケーションの高可用性をサポートします。より高度で大規模な攻撃 (UDP フラッド攻撃など)、State-Exhaustion 攻撃 (TCP SYN フラッドなど) に対する追加の保護、および Amazon Elastic Compute Cloud (Amazon EC2)、Elastic Load Balancing (ELB)、Amazon CloudFront、AWS Global Accelerator、Route 53 上で実行されるアプリケーションの保護には、AWS Shield Advanced の使用を検討できます。HTTP POST や GET フラッド攻撃などのアプリケーションレイヤー攻撃からの保護には、AWS WAF を使用します。AWS WAF を使用すると、IP アドレス、HTTP ヘッダー、HTTP ボディ、URI 文字列、SQL インジェクション、クロスサイトスクリプティング条件を使用して、リクエストをブロックするか許可するかを決定できます。

実装手順

1. 高可用性の DNS の設定: Amazon Route 53 は、可用性と拡張性に優れたドメインネームシステム (DNS) のウェブサービスです。Route 53 は、ユーザーリクエストを AWS またはオンプレミスで実行されるインターネットアプリケーションに接続します。詳細については、「Amazon Route 53 を DNS サービスとして設定する」を参照してください。
2. ヘルスチェックの設定: Route 53 を使用する場合、正常なターゲットのみが解決可能であることを確認します。Route 53 ヘルスチェックの作成と DNS フェイルオーバーの設定から始めます。ヘルスチェックを設定する際には、以下を考慮することが重要です。
   a. Amazon Route 53 でヘルスチェックの正常性を判断する方法
   b. ヘルスチェックの作成、更新、削除
   c. ヘルスチェックのステータス監視と通知の受信
   d. Amazon Route 53 DNS のベストプラクティス

3. DNS サービスをエンドポイントに接続します。
    a. Elastic Load Balancing をトラフィックのターゲットとして使用する場合は、ロードバランサーのリージョンエンドポイントを指す Amazon Route 53 を使用してエイリアスレコードを作成します。エイリアスレコードの作成中に、[ターゲットの正常性の評価] オプションを [あり] に設定します。
    b. サーバーレスワークロードまたはプライベート API については、API Gateway を使用する場合、Route 53 を使用してトラフィックを API Gateway にルーティングします。
4. コンテンツ配信ネットワークを決定します。
    a. ユーザーに近いエッジロケーションを使用してコンテンツを配信するには、CloudFront がコンテンツを配信する方法を理解することから始めます。
    b. 簡単な CloudFront ディストリビューションの使用から始めます。これにより、CloudFront は、コンテンツの配信元と、コンテンツ配信の追跡および管理方法に関する詳細を認識できます。CloudFront のディストリビューションを設定する際には、以下の側面を理解して、考慮することが重要です。
        i. CloudFront エッジロケーションでのキャッシュの仕組み
        ii. CloudFront キャッシュから直接提供されるリクエストの比率 (キャッシュヒット率) の向上
        iii. Amazon CloudFront Origin Shield の使用
        iv. CloudFront オリジンフェイルオーバーによる高可用性の最適化
5. アプリケーションレイヤー保護の設定: AWS WAF は、可用性低下、セキュリティの侵害、リソースの過剰消費といった、一般的なウェブのエクスプロイトやボットから保護します。詳細を理解するには、「how AWS WAF works」を確認し、アプリケーションレイヤーの HTTP POST および GET フラッド攻撃からの保護を実装する準備が整ったら、「Getting started with AWS WAF」を確認してください。AWS WAF は CloudFront と組み合わせて使用することもできます。ドキュメントについては、「how AWS WAF works with Amazon CloudFront features」を参照してください。
6. 追加の DDoS 保護の設定: デフォルトでは、すべての AWS のお客様が追加料金なしで、ウェブサイトやアプリケーションをターゲットする一般的で最も頻繁に発生するネットワーク層およびトランスポート層の DDoS 攻撃に対する AWS Shield Standard を使用した保護を受けています。Amazon EC2、Elastic Load Balancing、Amazon CloudFront、AWS Global Accelerator、Amazon Route 53 で実行されているインターネット接続アプリケーションの保護を強化するには、AWS Shield Advanced を検討し、「examples of DDoS resilient architectures」を確認してください。ワークロードとパブリックエンドポイントを DDoS 攻撃から保護するには、「Getting started with AWS Shield Advanced」を確認してください。

リソース

関連するベストプラクティス:

- REL10-BP01 複数の場所にワークロードをデプロイする
- REL11-BP04 復旧中はコントロールプレーンではなくデータプレーンを利用する
- REL11-BP06 イベントが可用性に影響する場合に通知を送信する

関連ドキュメント:

- APN パートナー: ネットワークの計画を支援できるパートナー
- ネットワークインフラストラクチャ向け AWS Marketplace
- とはAWS Global Accelerator
- Amazon CloudFront とは何ですか?
- Amazon Route 53 とは?
- Elastic Load Balancing とは?
- ネットワーク接続機能 - クラウド基盤の確立
- Amazon API Gateway とは何ですか?
- AWS WAF、AWS Shield、AWS Firewall Manager とは
- Amazon Application Recovery Controller とは
- DNS フェイルオーバーのカスタムヘルスチェックの設定

関連動画:

- AWS re:Invent 2022 - Improve performance and availability with AWS Global Accelerator
- AWS re:Invent 2020: Global traffic management with Amazon Route 53
- AWS re:Invent 2022 - Operating highly available Multi-AZ applications
- AWS re:Invent 2022 - Dive deep on AWS networking infrastructure
- AWS re:Invent 2022 - Building resilient networks

関連する例:

- Amazon Application Recovery Controller (ARC) を使用したディザスタリカバリ

- AWS Global Accelerator ワークショップ

REL02-BP02 クラウド環境とオンプレミス環境のプライベートネットワーク間の冗長接続をプロビジョニングする

クラウド環境とオンプレミス環境のプライベートネットワーク間の接続に冗長性を実装して、接続の耐障害性を実現します。これは、ネットワーク障害発生時にも接続を維持できるように、2 つ以上のリンクとトラフィックパスをデプロイすることで可能になります。

一般的なアンチパターン:

- 単一のネットワーク接続に依存することで単一障害点が発生する。
- 1 つの VPN トンネルのみを使用するか、同じアベイラビリティーゾーンで終端する複数のトンネルを使用する。
- 1 社の ISP に VPN 接続を依存しているため、ISP が停止すると完全なネットワーク障害につながる可能性がある。
- ネットワーク中断時のトラフィック再ルーティングに不可欠な BGP などの動的ルーティングプロトコルを実装していない。
- VPN トンネルの帯域幅制限を無視し、バックアップ機能を過大評価している。

このベストプラクティスを活用するメリット: クラウド環境と企業/オンプレミス環境の間に冗長接続を実装することにより、2 つの環境間で、依存するサービスが確実に通信できるようになります。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

AWS Direct Connect を使用してオンプレミスネットワークを AWS に接続する場合、複数のオンプレミスのロケーションと複数の AWS Direct Connect のロケーションにある個別のデバイスで終端する個別の接続を使用して、ネットワークの耐障害性を最大限に高める (99.99% の SLA) ことができます。このトポロジは、デバイス障害、接続問題、およびロケーション全体での停止に対する耐障害性を提供します。または、複数のロケーション (各オンプレミスのロケーションを 1 つの Direct Connect ロケーションに接続) への 2 つの個別の接続を使用することで、高い耐障害性 (99.9% の SLA) を実現できます。このアプローチにより、ファイバーの切断やデバイスの障害による接続の中断を防ぎ、ロケーション全体での障害を軽減できます。Direct Connect Resiliency Toolkit は、AWS Direct Connect トポロジの設計に役立ちます。

プライマリ AWS Direct Connect 接続に対する費用対効果の高いバックアップとして、AWS Transit Gateway で終端する AWS Site-to-Site VPN を検討することもできます。この設定により、複数の VPN トンネルでの等コストマルチパス (ECMP) ルーティングが可能になり、各 VPN トンネルの上限が 1.25 Gbps であっても、最大 50 Gbps のスループットが可能になります。ただし、ネットワークの中断を最小限に抑え、安定した接続を提供するには、AWS Direct Connect が依然として最も効果的な選択肢であることに注意してください。

インターネット経由で VPN を使用してクラウド環境をオンプレミスのデータセンターに接続する場合は、単一のサイト間 VPN 接続の一部として 2 つの VPN トンネルを設定します。高可用性を実現するために、各トンネルが異なるアベイラビリティーゾーンで終端するようにし、オンプレミスデバイスの障害を防ぐために冗長ハードウェアを使用する必要があります。さらに、1 社のインターネットサービスプロバイダー (ISP) の停止によって VPN 接続が完全に中断してしまうことを防ぐために、オンプレミスの口ケーションで異なる ISP による複数のインターネット接続を利用することを検討してください。ルーティングとインフラストラクチャが多様な ISP、特に AWS エンドポイントに複数の物理パスがある ISP を選択すると、高い接続可用性が得られます。

複数の AWS Direct Connect 接続と複数の VPN トンネル (または両方の組み合わせ) による物理的な冗長性に加え、ボーダーゲートウェイプロトコル (BGP) の動的ルーティングを実装することも重要です。動的 BGP は、リアルタイムのネットワーク状態と設定されているポリシーに基づいて、1 つのパスから別のパスにトラフィックを自動的に再ルーティングします。この動的な動作は、リンクまたはネットワーク障害の発生時にネットワークの可用性とサービスの継続性を維持するうえで特に役立ちます。代替パスが瞬時に選択されるため、ネットワークの耐障害性と信頼性が高まります。

実装手順

- AWS とオンプレミス環境間に可用性の高い接続を確保します。
  - 個別にデプロイされたプライベートネットワーク間で複数の AWS Direct Connect 接続または VPN トンネルを使用します。
  - 複数の Direct Connect 口ケーションを使用して、高い可用性を確保します。
  - 複数の AWS リージョンを使用している場合は、2 つ以上のリージョンで冗長性を確保します。
- 可能な場合は AWS Transit Gateway を使用して、VPN 接続 を終端します。
- AWS Marketplace アプライアンスを評価して VPN を終端するか、SD-WAN を AWS に拡張します。AWS Marketplace アプライアンスを使用する場合は、さまざまなアベイラビリティーゾーンで高可用性を実現するために、冗長インスタンスをデプロイします。
- オンプレミス環境への冗長接続を確保します。
  - 可用性の二ーズを満たすために、複数の AWS リージョンへの冗長接続が必要な場合があります。

- Direct Connect Resiliency Toolkit を使用して開始します。

リソース

関連ドキュメント:

- AWS Direct Connect の耐障害性に関するレコメンデーション
- 冗長な Site-to-Site VPN 接続を使用してフェイルオーバーを提供する
- ルーティングポリシーと BGP コミュニティ
- Active/Active and Active/Passive Configurations in AWS Direct Connect
- APN パートナー: ネットワークの計画を支援できるパートナー
- ネットワークインフラストラクチャ向け AWS Marketplace
- Amazon Virtual Private Cloud Connectivity Options ホワイトペーパー
- スケーラブルでセキュアなマルチ VPC の AWS ネットワークインフラストラクチャの構築
- 冗長な Site-to-Site VPN 接続を使用してフェイルオーバーを提供する
- Direct Connect Resiliency Toolkit を使用して開始する
- VPC エンドポイントおよび VPC エンドポイントサービス (AWS PrivateLink)
- Amazon VPC とは?
- Transit Gateway とは
- What is AWS Site-to-Site VPN?
- Direct Connect ゲートウェイの操作

関連動画:

- AWS re:Invent 2018: Amazon VPC 向けの高度な VPC 設計と新機能
- AWS re:Invent 2019: 多くの VPC に対応した AWS Transit Gateway のリファレンスアーキテクチャ

REL02-BP03 拡張性と可用性を考慮した IP サブネットの割り当てを確実に行う

Amazon VPC の IP アドレス範囲は、将来の拡張やアベイラビリティーゾーン間でのサブネットへの IP アドレスの割り当てを考慮して、ワークロードの要件を満たすための十分な大きさが必要です。

これには、ロードバランサー、EC2 インスタンス、コンテナベースのアプリケーションが含まれます。

ネットワークトポロジの計画は、IP アドレススペースの定義から始めます。プライベート IP アドレス範囲 (RFC 1918 ガイドラインに準拠) は、VPC ごとに割り当てる必要があります。このプロセスの一環として、次の要件を満たすようにします。

- リージョンごとに複数の VPC 用の IP アドレススペースを割り当てます。
- VPC 内で複数のアベイラビリティーゾーンを網羅できるように、複数のサブネット用のスペースを確保します。
- 将来の拡張のために、未使用の CIDR ブロックスペースを VPC 内に残しておくことを検討します。
- 機械学習用のスポットフリート、Amazon EMR クラスター、Amazon Redshift クラスターなど、使用する可能性のある Amazon EC2 インスタンスの一時的なフリートのニーズを満たす IP アドレススペースがあることを確認します。各 Kubernetes ポッドにはデフォルトで VPC CIDR ブロックからルーティング可能なアドレスが割り当てられるため、Amazon Elastic Kubernetes Service (Amazon EKS) などの Kubernetes クラスターについても同様の検討が必要です。
- 各サブネット CIDR ブロックの最初の 4 つの IP アドレスと最後の IP アドレスはリザーブドのため、使用できません。
- VPC に割り当てられた最初の VPC CIDR ブロックは変更または削除できませんが、重複していない CIDR ブロックは VPC に追加できます。サブネット IPv4 CIDR は変更できませんが、IPv6 CIDR は変更できます。
- 利用可能な VPC CIDR ブロックの最大サイズは /16、最小サイズは /28 です。
- 他の接続ネットワーク (VPC、オンプレミス、その他のクラウドプロバイダー) を検討し、IP アドレススペースが重複しないようにしてください。詳細については、「REL02-BP05接続されているすべてのプライベートアドレススペースにおいて、重複しないプライベート IP アドレス範囲を指定する」を参照してください。

期待できる成果: IP サブネットをスケールできるため、将来の成長に対応し、無駄を回避できます。

一般的なアンチパターン:

- 将来の成長が考慮されていないため、CIDR ブロックが小さすぎて再構成が必要になり、ダウンタイムが発生する可能性がある。
- Elastic Load Balancer が使用できる IP アドレスの数を不正確に見積もる。
- 多数の高トラフィックロードバランサーを同じサブネットにデプロイする。

- IP アドレスの消費をモニタリングできない状態で、自動スケーリングメカニズムを使用する。
- 将来の成長予測をはるかに超える過剰に大きな CIDR 範囲を定義したせいで、アドレス範囲が重複する他のネットワークとのピアリングが困難になる可能性がある。

このベストプラクティスを活用するメリット: これにより、ワークロードの増大に対応し、スケールアップ時に引き続き可用性を提供できます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

拡張、規制コンプライアンス、他のネットワークとの統合に対応できるようにネットワークを計画します。適切に計画しないと、拡張の見積もりが甘くなったり、規制コンプライアンスが変わったり、取得やプライベートネットワーク接続の実装が困難になったりする場合があります。

- サービス要件、レイテンシー、規制、ディザスタリカバリ (DR) 要件に基づいて、関連する AWS アカウントとリージョンを選択します。
- リージョン別 VPC デプロイのニーズを明確にします。
- VPC のサイズを明確にします。
  - マルチ VPC 接続をデプロイするかどうかを判断します。
    - Transit Gateway とは
    - 単一リージョンの複数 VPC 接続
  - 規制要件のためにネットワークの分離が必要かどうかを判断します。
  - 現在および将来のニーズに合わせて、適切なサイズの CIDR ブロックを持つ VPC を作成します。
    - 成長予測が不確かな場合は、将来の再構成のリスクを軽減するために、大きめの CIDR ブロックを選択しておいた方がよいでしょう。
  - デュアルスタック VPC の一部として、サブネットの IPv6 アドレス指定を使用することを検討してください。IPv6 は、多数の IPv4 アドレスが必要となる一時的なインスタンスやコンテナのフリートを含む、プライベートサブネットでの使用に適しています。

リソース

関連する Well-Architected のベストプラクティス:

- REL02-BP05 接続されているすべてのプライベートアドレススペースにおいて、重複しないプライベート IP アドレス範囲を指定する

関連ドキュメント:

- APN パートナー: ネットワークの計画を支援できるパートナー
- ネットワークインフラストラクチャ向け AWS Marketplace
- Amazon Virtual Private Cloud Connectivity Options ホワイトペーパー
- 複数のデータセンターの HA ネットワーク接続
- 単一リージョンの複数 VPC 接続
- Amazon VPC とは?
- IPv6 on AWS
- リファレンスアーキテクチャの IPv6
- Amazon EKS が IPv6 サポートを開始
- VPC に関する推奨事項 - Classic Load Balancer
- アベイラビリティーゾーンサブネット - Application Load Balancer
- アベイラビリティーゾーン - Network Load Balancer

関連動画:

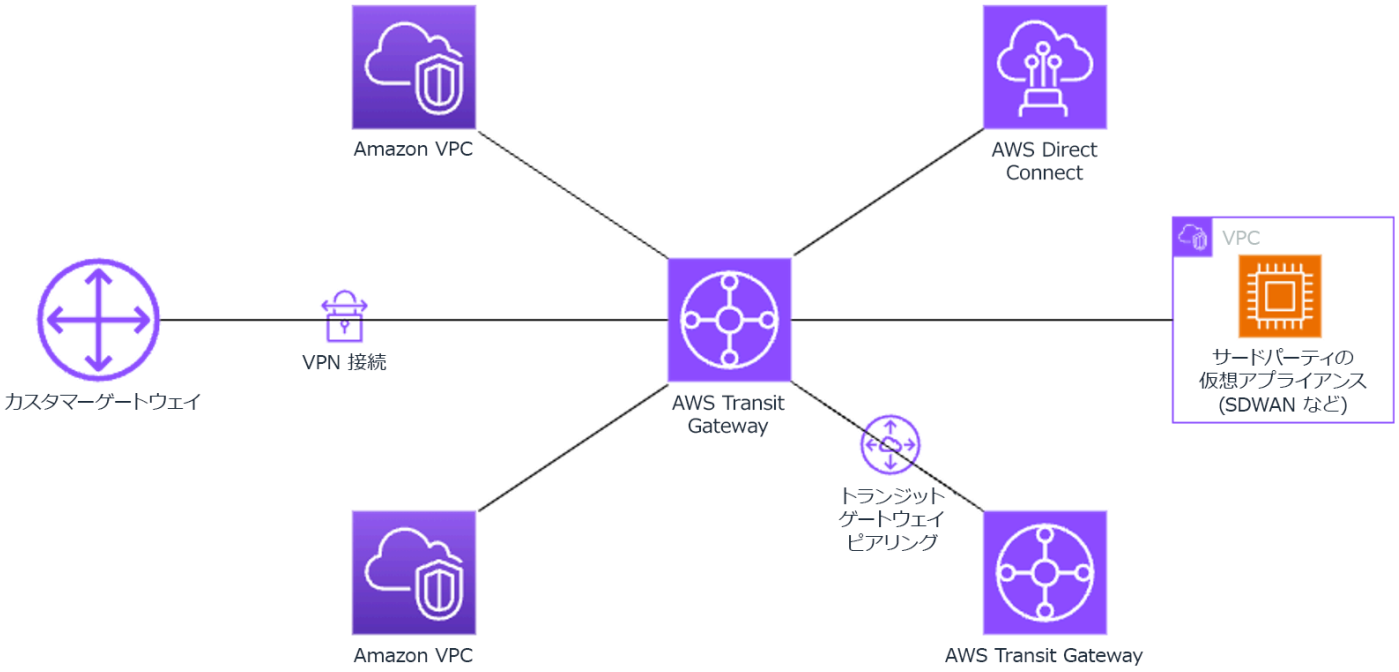
- AWS re:Invent 2018: Amazon VPC 向けの高度な VPC 設計と新機能 (NET303)
- AWS re:Invent 2019: 多くの VPC に対応した AWS Transit Gateway のリファレンスアーキテクチャ (NET406-R1)
- AWS re:Invent 2023: AWS の新機能のご紹介 成長と柔軟性を考慮したネットワーク設計 (NET310)

REL02-BP04 多対多メッシュよりもハブアンドスポークトポロジを優先する

仮想プライベートクラウド (VPC) やオンプレミスネットワークなど、複数のプライベートネットワークを接続する場合は、メッシュトポロジではなくハブアンドスポークトポロジを選択します。各ネットワークが互いに直接接続され、複雑さと管理オーバーヘッドが増加するメッシュトポロジとは異なり、ハブアンドスポークアーキテクチャでは、接続が 1 つのハブを介して一元化されます。こ

の一元化により、ネットワーク構造が簡素化され、運用性、スケーラビリティ、コントロールが強化されます。

AWS Transit Gateway は、AWS でハブアンドスポークネットワークを構築するために設計された、スケーラブルで可用性の高いマネージドサービスです。このサービスは、ネットワークのセントラルハブとして機能し、ネットワークのセグメンテーション、一元化されたルーティング、クラウド環境とオンプレミス環境両方へのシンプルな接続を提供します。次の図は、AWS Transit Gateway を使用してハブアンドスポークトポロジを構築する方法を示しています。



期待される成果: 仮想プライベートクラウド (VPC) とオンプレミスネットワークを中央ハブ経由で接続しました。スケーラビリティの高いクラウドルーターとして機能するハブを介してピアリング接続を設定します。複雑なピアリング関係を使用する必要がないため、ルーティングが簡素化されます。ネットワーク間のトラフィックは暗号化され、ネットワークを分離できます。

一般的なアンチパターン:

- 複雑なネットワークピアリングルールを構築している。
- 相互に通信すべきでないネットワーク間のルート (相互依存性のない個別のワークロードなど) を提供している。
- ハブインスタンスのガバナンスが効果的ではない。

このベストプラクティスを活用するメリット: 接続するネットワークの数が増えるにつれて、メッシュ接続の管理と拡張はますます困難になります。メッシュアーキテクチャには、追加のインフラストラクチャコンポーネント、設定要件、デプロイに関する考慮事項など、追加の課題があります。メッシュでは、データプレーンとコントロールプレーンコンポーネントを管理およびモニタリングするための追加のオーバーヘッドも必要になります。メッシュアーキテクチャの高可用性を提供する方法、メッシュのヘルスとパフォーマンスをモニタリングする方法、メッシュコンポーネントのアップグレードを処理する方法について考える必要があります。

一方、ハブアンドスポークモデルは、複数のネットワークにわたる一元的なトラフィックルーティングを確立します。これにより、データプレーンとコントロールプレーンコンポーネントの管理とモニタリングに、よりシンプルなアプローチが実現します。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

Network Services アカウントが存在しない場合は、アカウントを作成します。ハブを組織の Network Services アカウントに配置します。このアプローチにより、ハブはネットワークエンジニアによって一元管理されます。

ハブアンドスポークモデルのハブは、仮想プライベートクラウド (VPC) とオンプレミスネットワーク間を流れるトラフィックの仮想ルーターとして機能します。このアプローチにより、ネットワークの複雑さが軽減され、ネットワーク問題のトラブルシューティングが容易になります。

相互接続する VPC、AWS Direct Connect、Site-to-Site VPN 接続を含むネットワーク設計を検討してください。

各 Transit Gateway VPC アタッチメントに個別のサブネットの使用を検討してください。各サブネットに対して、小さな CIDR (/28 など) を使用して、コンピューティングリソース用のアドレス空間が増えるようにします。さらに、ネットワーク ACL を 1 つ作成し、ハブに関連付けられたすべてのサブネットに関連付けます。ネットワーク ACL は、インバウンド方向とアウトバウンド方向の両方で開いたままにします。

通信するネットワーク間でのみルートが提供されるように、ルーティングテーブルを設計して実装します。相互に通信すべきでないネットワーク間のルートを省略します (相互依存関係のない個別のワークロード間など)。

実装手順

1. ネットワークを計画します。接続するネットワークを決定し、重複する CIDR 範囲を共有していないことを確認します。

2. AWS Transit Gateway を作成し、VPC をアタッチします。
3. 必要に応じて、VPN 接続または Direct Connect ゲートウェイを作成し、それらを Transit Gateway に関連付けます。
4. Transit Gateway のルートテーブルを設定して、接続されている VPC と他の接続間のトラフィックのルーティング方法を定義します。
5. パフォーマンスとコストを最適化するために Amazon CloudWatch を使用し、必要に応じて構成のモニタリングと調整を行います。

リソース

関連するベストプラクティス:

- REL02-BP03 拡張性と可用性を考慮した IP サブネットの割り当てを確実に行う
- REL02-BP05 接続されているすべてのプライベートアドレススペースにおいて、重複しないプライベート IP アドレス範囲を指定する

関連ドキュメント:

- Transit Gateway とは
- Transit Gateway 設計のベストプラクティス
- スケーラブルでセキュアなマルチ VPC の AWS ネットワークインフラストラクチャの構築
- AWS Transit Gateway のリージョン間ピアリングを使用したグローバルネットワークの構築
- Amazon Virtual Private Cloud の接続オプション
- APN パートナー: ネットワークの計画を支援できるパートナー
- AWS Marketplaceネットワークインフラストラクチャ向け

関連動画:

- AWS re:Invent 2023 - AWS ネットワーキングの基礎
- AWS re:Invent 2023 – 高度な VPC 設計と新機能

関連するワークショップ:

- AWS Transit Gateway ワークショップ

REL02-BP05 接続されているすべてのプライベートアドレススペースにおいて、重複しないプライベート IP アドレス範囲を指定する

各 VPC の IP アドレス範囲が、ピア接続時、Transit Gateway 経由での接続時、または VPN 経由での接続時に重複しないようにする必要があります。VPC とオンプレミス環境間の、または使用する他のクラウドプロバイダーとの IP アドレスの競合を回避してください。必要に応じてプライベート IP アドレス範囲を割り当てる方法を用意する必要もあります。これを自動化するには、IP アドレス管理 (IPAM) システムが役立ちます。

期待される成果:

- VPC、オンプレミス環境、または他のクラウドプロバイダー間で IP アドレス範囲が競合しません。
- 適切に IP アドレスを管理することで、ネットワークインフラストラクチャを容易にスケールし、規模の拡大やネットワーク要件の変更に対応できるようになります。

一般的なアンチパターン:

- オンプレミス、社内ネットワーク、または他のクラウドプロバイダーにあるものと同じ IP 範囲を VPC で使用する。
- ワークロードのデプロイに使用されている VPC の IP 範囲を追跡しない。
- スプレッドシートなど、手動の IP アドレス管理プロセスに依存する。
- CIDR ブロックサイズの過剰/過小なサイズ設定によって無駄な IP アドレスが発生する、またはワークロード用のアドレススペースに不足が発生する。

このベストプラクティスを活用するメリット: ネットワークを積極的に計画することで、相互接続されたネットワークで同じ IP アドレスが複数出現しないようにできます。これにより、異なるアプリケーションを使用しているワークロードの一部でルーティングの問題が発生するのを防ぐことができます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

Amazon VPC IP Address Manager などの IPAM を使用して、CIDR の使用をモニタリングおよび管理します。AWS Marketplace は、複数の IPAM を提供しています。AWS での予想される使用量を評

価して、CIDR の範囲を既存の VPC に追加し、計画的な使用量の増加を可能にする VPC を作成します。

実装手順

- 現在の CIDR 消費量 (VPC、サブネットなど) を把握します。
  - サービス API オペレーションを使用して、現在の CIDR 消費量を収集します。
  - Amazon VPC IP Address Manager を使用して、リソースを検出します。
- 現在のサブネットの使用量を把握します。
  - サービス API オペレーションを使用して、各リージョンの VPC ごとにサブネットを収集します。
  - Amazon VPC IP Address Manager を使用して、リソースを検出します。
- 現在の使用量を記録します。
- 重複する IP 範囲を作成したかどうかを確認します。
- 予備容量を計算します。
- 重複している IP 範囲を特定します。新しいアドレス範囲に移行するか、重複する範囲を接続する必要がある場合は、プライベート NAT ゲートウェイや AWS PrivateLink などの手法を使用することを検討します。

リソース

関連するベストプラクティス:

- ネットワークの保護

関連ドキュメント:

- APN パートナー: ネットワークの計画を支援できるパートナー
- AWS Marketplaceネットワークインフラストラクチャ向け
- Amazon Virtual Private Cloud Connectivity Options ホワイトペーパー
- 複数のデータセンターの HA ネットワーク接続
- Connecting Networks with Overlapping IP Ranges
- Amazon VPC とは?
- IPAM とは

関連動画:

- AWS re:Invent 2023 - 高度な VPC 設計と新機能
- AWS re:Invent 2019: 多くの VPC に対応した AWS Transit Gateway のリファレンスアーキテクチャ
- AWS re:Invent 2023 - Ready for what's next? 成長と柔軟性を考慮したネットワーク設計
- AWS re:Invent 2021 - {New Launch} Manage your IP addresses at scale on AWS

ワークロードアーキテクチャ

信頼性の高いワークロードの実現は、ソフトウェアとインフラストラクチャの両方について事前に設計を決定することから始まります。アーキテクチャの選択は、Well-Architected の 6 つの柱のすべてにわたって、ワークロードの動作に影響を与えます。高い信頼性を保つには、特定のパターンに従う必要があります。

次のセクションでは、高い信頼性を保つためにこのようなパターンで使用するベストプラクティスについて説明します。

トピック

- ワークロードサービスアーキテクチャを設計する
- 障害を防ぐために分散システムでの操作を設計する
- 障害を軽減するため、または障害に耐えるために分散システムでの相互作用を設計する

ワークロードサービスアーキテクチャを設計する

サービス指向アーキテクチャ (SOA) またはマイクロサービスアーキテクチャを使用して、スケーラビリティと信頼性に優れたワークロードを構築します。サービス指向アーキテクチャ (SOA) は、サービスインターフェイスを介してソフトウェアコンポーネントを再利用できるようにする方法です。マイクロサービスアーキテクチャは、その一歩先を行き、コンポーネントをさらに小さくシンプルにしています。

サービス指向アーキテクチャ (SOA) インターフェイスは一般的な通信標準を使用しているため、新しいワークロードに迅速に組み込むことができます。相互依存する分割不可能なユニットで構成されたモノリスアーキテクチャを構築するプラクティスは、SOA に置き替えられました。

AWS では、長く SOA を使用してきましたが、現在はマイクロサービスを使用してシステムを構築しています。マイクロサービスには多くの魅力がありますが、可用性の点で重要なのは、マイクロサービスが小さくてシンプルであるということです。マイクロサービスでは、各種のサービスに求められる可用性を区別して、最も高い可用性ニーズを持つマイクロサービスに特化して投資を行うことができます。例えば、Amazon.com で製品情報ページ (「詳細ページ」) を配信するには、ページの個別の部分を作成するために何百ものマイクロサービスが呼び出されます。製品と料金の詳細を表示するために不可欠なサービスはいくつかありますが、そのサービスが利用できないときは、ページ上のコンテンツの大部分を単純に削除できます。顧客が製品を購入できる場合に、エクスペリエンスを提供するための写真やレビューなどは不要です。

ベストプラクティス

- REL03-BP01 ワークロードをセグメント化する方法を選択する
- REL03-BP02 特定のビジネスドメインと機能に重点を置いたサービスを構築する
- REL03-BP03 API ごとにサービス契約を提供する

REL03-BP01 ワークロードをセグメント化する方法を選択する

アプリケーションの回復力要件を決定する際に、ワークロードのセグメント化は重要です。モノリシックアーキテクチャはできるだけ避ける必要があります。代わりに、どのアプリケーションコンポーネントをマイクロサービスに分けられるかを注意深く検討します。アプリケーションの要件によっては、最終的にサービス指向アーキテクチャ (SOA) とマイクロサービスの組み合わせになることもあります。ステートレス化が可能なワークロードは、マイクロサービスとしてデプロイすることができます。

期待できる成果:ワークロードは、サポート可能でスケーラブルであり、可能な限り疎結合である必要があります。

ワークロードのセグメント化方法を選択する場合は、複雑さとメリットのバランスを考慮してください。新製品のローンチ時に適しているものは、最初からスケールするように構築されたワークロードが必要とするものとは異なります。既存のモノリスをリファクタリングする場合、アプリケーションがステートレスへの分解をどの程度サポートできるかを検討する必要があります。サービスをより細かく分割すると、明確に定義された小規模なチームがサービスを開発し、管理できるようになります。ただし、サービスが細かくなると、レイテンシーの増加、デバッグの複雑化、運用負荷の増大など、複雑な問題が発生する可能性があります。

一般的なアンチパターン:

- マイクロサービス Death Star とは、アトミックコンポーネントが強く依存しあっているために、1つの失敗がより大きな失敗となり、コンポーネントがモノリスのように柔軟性が低く、壊れやすくなっている状態のことです。

このベストプラクティスを活用するメリット:

- より特化したセグメントは、高い俊敏性、組織の柔軟性、およびスケーラビリティにつながります。
- サービス中断の影響が小さくなります。

- アプリケーションコンポーネントには異なる可用性要件があり、より特化したセグメント化によってサポートすることができます。
- ワークロードをサポートするチームの責任が明確に定義されます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

ワークロードをセグメント化する方法に基づいて、アーキテクチャタイプを選択します。SOA またはマイクロサービスアーキテクチャ (まれにモノリシックアーキテクチャ) を選択します。モノリシックアーキテクチャから開始する場合でも、それがモジュラー型で、ユーザーの導入に合わせて製品がスケールされるにつれて最終的に SOA またはマイクロサービスに進化できることを確認する必要があります。SOA とマイクロサービスは、それぞれより細かなセグメントを提供し、最新のスケーラブルで信頼性の高いアーキテクチャとして好まれていますが、特にマイクロサービスアーキテクチャをデプロイする際は、トレードオフを考慮しなければなりません。

主なトレードオフとしては、分散コンピューティングアーキテクチャを採用することになり、ユーザーのレイテンシー要件を達成するのが難しくなることと、ユーザーインタラクションのデバッグとトレースがさらに複雑になることが挙げられます。AWS X-Ray をこの問題の解決に役立てることができます。管理するアプリケーションの数が増え、複数の独立したコンポーネントをデプロイする必要があるため、運用が複雑になることも考慮する必要があります。

モノリシック、サービス指向、マイクロサービスアーキテクチャ

実装手順

- アプリケーションのリファクタリングやビルドに適したアーキテクチャを決定します。SOA とマイクロサービスは、それぞれがより細かなセグメント化を実現するため、最新のスケーラブルで信頼性の高いアーキテクチャとして好まれています。SOA は、マイクロサービスの複雑さを回避しながら、より細かなセグメント化を達成するための優れた折衷案となり得ます。詳細については、マイクロサービスのトレードオフを参照してください。
- ワークロードが適していて、組織がサポートできる場合は、最高の俊敏性と信頼性を実現するために、マイクロサービスアーキテクチャを使用する必要があります。詳細については、AWS でのマイクロサービスの実装を参照してください。
- モノリスを細かなコンポーネントにリファクタリングするには、Strangler Fig のパターンに従うことを検討してください。これには、特定のアプリケーションコンポーネントを新しいアプリケーションやサービスに徐々に置き換えることが含まれます。AWS Migration Hub Refactor Spaces は、増分リファクタリングの開始点として機能します。詳細については、ストラングラーパターンを使用してオンプレミスのレガシーワークロードをシームレスに移行するを参照してください。
- マイクロサービスを実装するには、これらの分散サービスと相互に通信するためのサービス検出メカニズムが必要になる場合があります。AWS App Mesh はサービス指向アーキテクチャで使用することができ、信頼性の高いサービスの検出とアクセス性を提供します。AWS Cloud Map は、動的 DNS ベースのサービス検出にも使用できます。
- モノリスから SOA に移行する場合、Amazon MQ は、クラウド内のレガシーアプリケーションを再設計するときに、サービスバスとしてギャップを埋めるのに役立ちます。
- 単一の共有されたデータベースがある既存のモノリスには、データを再編成して細かなセグメントにする方法を選択します。これは、ビジネスユニット、アクセスパターン、またはデータ構造によって行うことができます。リファクタリングプロセスのこの時点では、リレーショナルまたは非リレーショナル (NoSQL) タイプのデータベースを選択して進めていく必要があります。詳細については、SQL から NoSQL へを参照してください。

実装計画に必要な工数レベル: 高

リソース

関連するベストプラクティス:

- REL03-BP02 特定のビジネスドメインと機能に重点を置いたサービスを構築する

関連ドキュメント:

- Amazon API Gateway: OpenAPI を使用した REST API の設定
- サービス指向アーキテクチャとは
- 境界付けられたコンテキスト (ドメイン駆動設計の中心的なパターン)
- AWS でのマイクロサービスの実装
- マイクロサービスのトレードオフ
- Microservices - a definition of this new architectural term
- AWS でのマイクロサービス
- AWS App Mesh とは

関連する例:

- イテレーティブアプリモダナイゼーションワークショップ

関連動画:

- Delivering Excellence with Microservices on AWS

REL03-BP02 特定のビジネスドメインと機能に重点を置いたサービスを構築する

サービス指向アーキテクチャ (SOA) は、ビジネスニーズに合わせて明確に定義された機能を備えたサービスを定義します。マイクロサービスは、ドメインモデルと制限付きコンテキストを使用して、ビジネスコンテキストの境界に沿ってサービスの境界を描きます。ビジネスドメインと機能に重点を置くことで、チームがサービスの独立した信頼性要件を定義しやすくなります。コンテキストに制限があると、ビジネスロジックが分離されてカプセル化されるため、チームは障害の処理方法について、より的確に判断できるようになります。

期待される成果: エンジニアとビジネス関係者は共同で境界のあるコンテキストを定義し、それを使用して、特定のビジネス機能を果たすサービスとしてシステムを設計します。これらのチームは、イベントストーミングなどの確立された手法を使用して要件を定義します。新しいアプリケーションは、境界を明確に定義し、疎結合するサービスとして設計されます。既存のモノリスは境界コンテキストに分解され、システム設計は SOA またはマイクロサービスアーキテクチャに移行します。モノリスをリファクタリングする際には、バブルコンテキストやモノリスの分解パターンなどの確立されたアプローチが適用されます。

ドメイン指向のサービスは、状態を共有しない 1 つ以上のプロセスとして実行されます。サービスは需要の変動に個別に対応し、ドメイン固有の要件に照らして障害シナリオを処理します。

一般的なアンチパターン:

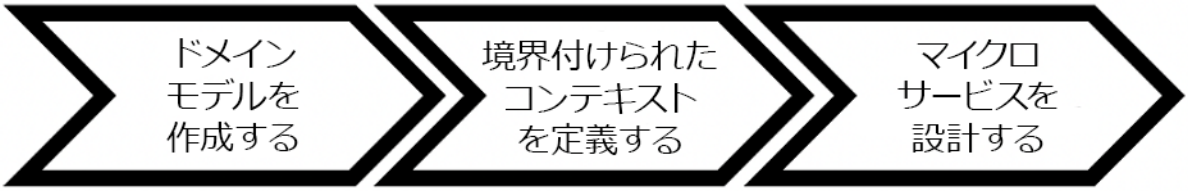
- チームは、特定のビジネスドメインではなく、UI や UX、ミドルウェア、データベースなどの特定の技術ドメインを中心に形成される。
- アプリケーションがドメインの担当範囲にまたがっている。限定されたコンテキストにまたがるサービスは、メンテナンスが難しく、大規模なテスト作業が必要になり、複数のドメインチームがソフトウェア更新に参加する必要がある。
- ドメインエンティティライブラリと同様に、ドメイン依存関係がサービス間で共有されるため、あるサービスドメインを変更すると、他のサービスドメインも変更する必要がある。
- サービス契約とビジネスロジックはエンティティを共通かつ一貫したドメイン言語で表現していないため、翻訳層が発生し、システムが複雑になり、デバッグ作業が増加する。

このベストプラクティスを活用するメリット: アプリケーションは、ビジネスドメインによって区切られた個別のサービスとして設計され、共通のビジネス言語を使用します。サービスは個別にテストおよびデプロイできます。サービスは、実装されたドメインのドメイン固有の回復力要件を満たします。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

ドメイン駆動型設計 (DDD) は、ビジネスドメインを中心にソフトウェアを設計および構築するための基本的なアプローチです。ビジネスドメインに焦点を当てたサービスを構築する際には、既存のフレームワークを使用すると便利です。既存のモノリシックアプリケーションを扱う場合は、確立された手法を提供する分解パターンを利用してアプリケーションをモダナイズし、サービスにすることができます。



ドメイン駆動型設計

実装手順

- チームはイベントストーミングワークショップを開催して、軽量の付箋形式でイベント、コマンド、集計、ドメインをすばやく特定できます。
- ドメインエンティティと関数がドメインコンテキストで形成されたら、境界コンテキストを使用してドメインをサービスに分割できます。境界コンテキストでは、類似の特徴と属性を共有するエンティティがグループ化されます。モデルをコンテキストに分割すると、マイクロサービスを境界で区切る方法のテンプレートが現れます。
  - 例えば、Amazon.com ウェブサイトエンティティには、パッケージ、配送、スケジュール、料金、割引、通貨などがあります。
  - パッケージ、配送、スケジュールは出荷コンテキストにグループ化され、料金、割引、通貨は料金設定コンテキストにグループ化されます。
- モノリスをマイクロサービスに分解すると、マイクロサービスをリファクタリングするためのパターンが概説されます。ビジネス機能、サブドメイン、またはトランザクション別に分解するパターンを使用することは、ドメイン駆動のアプローチに適しています。
- バブルコンテキストなどの戦術的な手法を使用すると、事前に書き直したり、DDD に全面的にコミットしたりすることなく、既存のアプリケーションまたはレガシーアプリケーションに DDD を導入できます。バブルコンテキストアプローチでは、サービスマッピングと調整、または新しく定義されたドメインモデルを外部の影響から保護する破損防止層を使用して、小さな境界コンテキストが確立されます。

チームがドメイン分析を行い、エンティティとサービス契約を定義したら、AWS のサービスを活用し、ドメイン駆動型の設計をクラウドベースのサービスとして実装できます。

- ドメインのビジネスルールを実践するテストを定義することから開発を始めましょう。テスト駆動型開発 (TDD) と動作駆動型開発 (BDD) は、チームがサービスをビジネス上の問題の解決に集中させるのに役立ちます。
- ビジネスドメインの要件とマイクロサービスアーキテクチャに最適な AWS のサービスを選択します。
  - AWS サーバーレスを使用すると、チームはサーバーやインフラストラクチャの管理ではなく、特定のドメインロジックに集中できます。
  - AWS のコンテナを使用すると、インフラストラクチャの管理が簡素化されるため、ドメイン要件に集中できます。
  - 目的別データベースは、ドメイン要件を最適なデータベースタイプに一致させるのに役立ちます。

- AWS にヘキサゴナルアーキテクチャを構築することで、ビジネスドメインから逆算してサービスにビジネスロジックを構築し、機能要件を満たして統合アダプターをアタッチするためのフレームワークの概要が示されます。インターフェイスの詳細と AWS のサービスのビジネスロジックを分離するパターンは、チームがドメインの機能に集中し、ソフトウェアの品質を向上させるのに役立ちます。

リソース

関連するベストプラクティス:

- REL03-BP01 ワークロードをセグメント化する方法を選択する
- REL03-BP03 API ごとにサービス契約を提供する

関連ドキュメント:

- AWS マイクロサービス
- AWS でのマイクロサービスの実装
- モノリスをマイクロサービスに分割する方法
- レガシーシステムに囲まれているときの DDD の使用開始
- エリック・エヴァンスのドメイン駆動設計: ソフトウェアの核心にある複雑さに立ち向かう
- AWS でヘキサゴナルアーキテクチャを構築する
- マイクロサービスへのモノリスの分解
- イベントストーミング
- 制限されたコンテキスト間のメッセージ
- マイクロサービス
- テスト駆動型開発
- 動作駆動型開発

関連する例:

- AWSでのクラウドネイティブマイクロサービスの設計 (DDD/EventStormingWorkshop より)

関連ツール:

- AWS クラウドデータベース
- AWS でのサーバーレス
- AWS でのコンテナ

REL03-BP03 API ごとにサービス契約を提供する

サービス契約とは、機械が読み取れる API 定義で定義された、API プロデューサーとコンシューマー間の文書化された契約です。契約バージョニング戦略により、コンシューマーは既存の API を引き続き使用し、準備ができたらアプリケーションを新しい API に移行できます。プロデューサーのデプロイは、契約がある限りいつでも行うことができます。サービスチームは、選択した技術スタックを使用して、API 契約の条件を満たすことができます。

期待される成果: サービス指向またはマイクロサービスアーキテクチャで構築されたアプリケーションは、ランタイム依存関係を統合しながら独立して動作できます。API コンシューマーまたはプロデューサーに変更をデプロイしても、双方が共通の API 契約に従っていれば、システム全体の安定性が損なわれることはありません。サービス API を介して通信するコンポーネントは、相互にほとんど、またはまったく影響を与えずに、独立した機能リリース、ランタイム依存関係のアップグレード、またはディザスタリカバリ (DR) サイトへのフェイルオーバーを実行できます。さらに、ディスクリートサービスでは、他のサービスを一斉にスケールインしなくても、吸収するリソース需要を個別にスケールできます。

一般的なアンチパターン:

- 厳密に型指定されたスキーマを使用しないサービス API を作成します。その結果、API バインディングの生成に使用できない API や、プログラムで検証できないペイロードが生成されます。
- バージョニング戦略を採用していないため、API コンシューマーに更新とリリースを強制します。または、サービス契約が進化するときに失敗します。
- ドメインコンテキストや言語での統合の失敗を説明するのではなく、基盤となるサービス実装の詳細を漏らすエラーメッセージ。
- API 契約を使用せずにテストケースを開発し、モック API 実装を使用しないことで、サービスコンポーネントを個別にテストできます。

このベストプラクティスを活用するメリット: API サービス契約を介して通信するコンポーネントで構成された分散型システムでは、信頼性を向上させることができます。開発者は、コンパイル中にタイプチェックを行って、リクエストとレスポンスが API 契約に従っていること、および必須フィールドが存在することを確認し、開発プロセスの早期に潜在的な問題を発見できます。API 契約

は、API 用のわかりやすい自己文書化インターフェイスを提供し、さまざまなシステムやプログラミング言語間の相互運用性を向上させます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

ビジネスドメインを特定し、ワークロードのセグメント化を決定したら、サービス API を開発できます。まず、機械が読み取れる API のサービス契約を定義し、次に API バージョニング戦略を実装します。REST、GraphQL、非同期イベントなどの一般的なプロトコルでサービスを統合する準備ができたら、AWS のサービスをアーキテクチャに組み込み、コンポーネントを厳密に型指定された API 契約と統合できます。

サービス API 契約の AWS のサービス

Amazon API Gateway、AWS AppSync、Amazon EventBridge などの AWS のサービスをアーキテクチャに組み込み、アプリケーションで API サービス契約を使用します。Amazon API Gateway は、ネイティブ AWS サービスやその他のウェブサービスと直接統合するのに役立ちます。API Gateway は、OpenAPI 仕様とバージョニングをサポートしています。AWS AppSync は、クエリ、ミューテーション、サブスクリプションのサービスインターフェイスを定義する GraphQL スキーマを定義して設定する、マネージド GraphQL エンドポイントです。Amazon EventBridge は、イベントスキーマを使用してイベントを定義し、イベントのコードバインディングを生成します。

実装手順

- まず、API の契約を定義します。契約では、API の機能を説明するだけでなく、API の入出力用に厳密に型指定されたデータオブジェクトとフィールドを定義します。
- API Gateway で API を設定すると、エンドポイントの OpenAPI 仕様をインポートおよびエクスポートできます。
  - OpenAPI 定義をインポートすると、API の作成が簡素化され、AWS Serverless Application Model や AWS Cloud Development Kit (AWS CDK) などのコードツールとしての AWS インフラストラクチャと統合できます。
  - API 定義をエクスポートすると、API テストツールとの統合が簡素化され、サービス利用者に統合仕様が提供されます。
- AWS AppSync で GraphQL スキーマファイルを定義して、GraphQL API を定義して管理し、コントラクトインターフェイスを生成して、複雑な REST モデル、複数のデータベーステーブル、またはレガシー サービスとのやり取りを簡素化できます。

- AWS AppSync と統合された AWS Amplify プロジェクトは、アプリケーションで使用するための厳密に型指定された JavaScript クエリファイルと、Amazon DynamoDB テーブル用の AWS AppSync GraphQL クライアントライブラリを生成します。
- Amazon EventBridge からサービスイベントを利用する場合、イベントは、スキーマレジストリに既に存在するスキーマや OpenAPI 仕様で定義したスキーマに従います。レジストリでスキーマを定義すると、スキーマ契約からクライアントバインディングを生成して、コードをイベントと統合することもできます。
- API の拡張またはバージョニング。オプションフィールドまたは必須フィールドのデフォルト値で構成できるフィールドを追加する場合、API を拡張する方が簡単なオプションです。
  - REST や GraphQL などのプロトコルの JSON ベースの契約は、契約の拡張に適しています。
  - SOAP のようなプロトコルの XML ベースの契約をサービスコンシューマーとテストして、契約拡張の可能性を判断する必要があります。
- API をバージョニングするときは、ロジックを単一のコードベースで管理できるように、ファサードを使用してバージョンをサポートするプロキシバージョニングの実装を検討してください。
  - API Gateway を使用すると、リクエストとレスポンスのマッピングを使用して、新しいフィールドにデフォルト値を提供したり、リクエストまたはレスポンスから削除されたフィールドを削除したりするファサードを確立し、契約の変更の吸収を簡素化できます。このアプローチにより、基盤となるサービスが単一のコードベースを維持できます。

リソース

関連するベストプラクティス:

- REL03-BP01 ワークロードをセグメント化する方法を選択する
- REL03-BP02 特定のビジネスドメインと機能に重点を置いたサービスを構築する
- REL04-BP02 疎結合の依存関係を実装する
- REL05-BP03 再試行呼び出しを制御および制限する
- REL05-BP05 クライアントタイムアウトを設定する

関連ドキュメント:

- API (アプリケーションプログラミングインターフェイス) とは
- AWS でのマイクロサービスの実装
- マイクロサービスのトレードオフ

- Microservices - a definition of this new architectural term
- AWS でのマイクロサービス
- OpenAPI への API Gateway 拡張機能の使用
- OpenAPI 仕様
- GraphQL: スキーマとタイプ
- Amazon EventBridge のコードバインディング

関連する例:

- Amazon API Gateway: OpenAPI を使用した REST API の設定
- Amazon API Gateway to Amazon DynamoDB CRUD application using OpenAPI
- サーバーレス時代のモダンアプリケーション統合パターン: API Gateway サービスの統合
- Amazon CloudFront でのヘッダーベースの API Gateway バージョニングの実装
- AWS AppSync: クライアントアプリケーションをビルドする

関連動画:

- Using OpenAPI in AWS SAM to manage API Gateway

関連ツール:

- Amazon API Gateway
- AWS AppSync
- Amazon EventBridge

障害を防ぐために分散システムでの操作を設計する

分散システムは、サーバーやサービスなどのコンポーネントを相互接続するために通信ネットワークに依存しています。これらのネットワークでデータ損失や遅延が発生しても、ワークロードは確実に動作する必要があります。分散システムのコンポーネントは、他のコンポーネントやワークロードに悪影響を及ぼさない方法で動作する必要があります。これらのベストプラクティスは障害を防ぎ、平均故障間隔 (MTBF) を改善します。

ベストプラクティス

REL04-BP01 依存している分散システムの種類を特定する

分散システムには、同期、非同期、またはバッチの 3 つの種類があります。同期システムは、リクエストを可能な限り迅速に処理し、HTTP/S、REST、またはリモートプロシージャコール (RPC) プロトコルを使用して同期リクエスト呼び出しと応答呼び出しを行うことで相互に通信する必要があります。非同期システムは、個々のシステムを結合することなく、中間サービスを介して非同期的にデータを交換することによって相互に通信します。バッチシステムは大量の入力データを受け取り、人の介入なしに自動データプロセスを実行し、出力データを生成します。

望ましい結果: 同期、非同期、バッチの依存関係と効果的に相互作用するワークロードを設計します。

一般的なアンチパターン:

- ワークロードは依存関係からの応答を無期限に待つため、リクエストが受信されたかどうかが不明なまま、ワークロードクライアントがタイムアウトすることがあります。
- ワークロードは、相互に同期呼び出しを行う一連の依存システムを使用しています。これには、チェーン全体で正常に処理が行われる前に、各システムが利用可能な状態にあり、システムでリクエストを正常に処理する必要があるため、動作が脆弱になり、全体的な可用性が損なわれる可能性があります。
- ワークロードは依存関係と非同期で通信し、重複したメッセージを受信する場合が多いのにもかかわらず、1 回のみのメッセージ処理が保証されるという概念に基づいています。
- 適切なバッチスケジューリングツールを使用していないため、ワークロードは同じバッチジョブを同時に実行します。

このベストプラクティスを活用する利点: 特定のワークロードでは、同期、非同期、バッチのいずれかの通信スタイルを 1 つ以上実装するのが一般的です。このベストプラクティスは、それぞれの通信スタイルに関連するさまざまなトレードオフを特定し、ワークロードが依存関係の中断に耐えられるようにするのに役立ちます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

以下のセクションでは、各種類の依存関係に関する一般的な実装ガイダンスと固有の実装ガイダンスの両方について説明します。

一般的な質問、または機能要望

- 依存関係が提供するパフォーマンスと信頼性のサービスレベル目標 (SLO) が、ワークロードのパフォーマンスと信頼性の要件を満たしていることを確認します。
- AWS オブザーバビリティサービスを使用して応答時間とエラー率をモニタリングし、依存関係がワークロードに必要なレベルでサービスを提供していることを確認します。
- 依存関係と通信する際に、ワークロードが直面する可能性のある課題を特定します。分散システムには、アーキテクチャの複雑さ、運用上の負担、コストが増加する可能性のあるさまざまな課題があります。一般的な課題には、レイテンシー、ネットワークの中断、データ損失、スケーリング、データ複製の遅延などがあります。
- 堅牢なエラー処理とログ記録を実装して、依存関係で問題が発生した場合の問題のトラブルシューティングに役立ててください。

同期依存関係

同期通信では、ワークロードは依存関係にリクエストを送信し、応答を待っている操作をブロックします。依存関係がリクエストを受け取ると、すぐに処理を試み、応答をワークロードに送り返します。同期通信の大きな課題は、一時的な結合が発生するため、ワークロードとその依存関係を同時に利用できるようにする必要があることです。ワークロードで依存関係との同期通信が必要な場合は、以下のガイダンスを検討します。

- ワークロードが 1 つの機能を実行するために複数の同期依存関係に依存するべきではありません。リクエストを正常に完了させるためにパス内のすべての依存関係が利用可能である必要があるため、依存関係の連鎖は全体的な脆弱性を高めます。
- 依存関係が正常でない場合や利用できない場合のエラー処理と再試行戦略を決定します。バイモーダル動作の使用は避けてください。バイモーダル動作とは、通常モードと障害モードでワークロードが異なる動作を示す場合をいいます。バイモーダル動作の詳細については、「REL11-BP05 静的安定性を使用してバイモーダル動作を防止する」を参照してください。
- ワークロードを待機させるより、フェイルファストの方がよいことを覚えておいてください。例えば、「AWS Lambda デベロッパーガイド」では、Lambda 関数を呼び出すときに再試行や失敗を処理する方法について説明します。

- ワークロードが依存関係を呼び出す際のタイムアウトを設定します。これにより、応答を待つ時間が長すぎたり、無期限に待ったりすることを回避できます。このトピックに関する役立つ説明は、「レイテンシーを考慮した Amazon DynamoDB アプリケーションのための AWS Java SDK HTTP リクエスト設定のチューニング」に記載されています。
- 1 つのリクエストを処理するためのワークロードから依存関係への呼び出し回数を最小限に抑えます。呼び出し回数の多さは、結合とレイテンシーの増加につながります。

非同期依存関係

ワークロードを依存関係から一時的に切り離すには、非同期で通信する必要があります。非同期アプローチを使用すると、ワークロードの依存関係や一連の依存関係からの応答の送信を待つことなく、他の処理を実行できます。

ワークロードで依存関係との非同期通信が必要な場合は、以下のガイダンスを検討します。

- ユースケースと要件に基づいて、メッセージングとイベントストリーミングのどちらを使用するかを決定します。メッセージングを使用すると、メッセージブローカーを介してメッセージを送受信することで、ワークロードが依存関係と通信できます。イベントストリーミングを使用すると、ワークロードとその依存関係は、ストリーミングサービスを使用して、できるだけ早く処理する必要のあるデータを継続的なストリームとして配信されるイベントを公開およびサブスクライブできます。

- メッセージングとイベントストリーミングではメッセージの処理方法が異なるため、以下に基づいてトレードオフを決定する必要があります。
  - メッセージ優先度: メッセージブローカーは、通常のメッセージよりも先に優先度の高いメッセージを処理できます。イベントストリーミングでは、すべてのメッセージの優先度が同じになります。
  - メッセージ消費: メッセージブローカーは、コンシューマーがメッセージを受信したかどうか確認します。イベントストリーミングのコンシューマーは、最後に読んだメッセージを常に把握しておく必要があります。
  - メッセージの順序: メッセージングでは、先入れ先出し (FIFO) アプローチを使用しない限り、メッセージの送信順序を正確に受信することは保証されません。イベントストリーミングでは、データが生成された順序が常に保持されます。
  - メッセージの削除: メッセージングでは、コンシューマーはメッセージを処理した後に削除する必要があります。イベントストリーミングサービスはメッセージをストリームに追加し、メッ

セージの保存期間が終了するまでストリームに残ります。この削除ポリシーにより、イベントストリーミングはメッセージの再生に適したものになります。

- 依存関係がいつ作業を完了したかをワークロードがどのように認識するかを定義します。ワークロードが Lambda 関数を非同期的に呼び出すと、Lambda はリクエストをキューに入れ、追加情報を含まない成功のレスポンスを返します。処理が完了すると、Lambda 関数は成功または失敗に基づいて設定可能な送信先に結果を送信できます。
- べき等性を活用して、重複メッセージを処理するワークロードを構築します。べき等性とは、同じメッセージに対してワークロードが複数回生成されても、ワークロードの結果は変化しないことを指します。ネットワーク障害が発生した場合、または確認応答が受信されていない場合、メッセージングサービスまたはストリーミングサービスがメッセージを再配信することに注意してください。
- ワークロードが依存関係から応答を受け取らない場合は、ワークロードはリクエストを再送信します。リトライ回数を制限して、ワークロードの CPU、メモリ、ネットワークリソースの消費を抑え、他のリクエストを処理できるようにすることを検討してください。「AWS Lambda ドキュメント」では、非同期呼び出しのエラーを処理する方法を示しています。
- 適切なオブザーバビリティ、デバッグ、トレースツールを活用して、ワークロードの非同期通信とその依存関係を管理し運用します。Amazon CloudWatch を使用して、メッセージングおよびイベントストリーミングサービスをモニタリングできます。また、AWS X-Ray を使用してワークロードを計測し、問題のトラブルシューティングに関するインサイトをすばやく得ることもできます。

バッチ依存関係

バッチシステムは、手動操作なしで、入力データを受け取り、処理するための一連のジョブを開始し、いくつかの出力データを生成します。データサイズにもよりますが、ジョブは数分から、場合によっては数日かかることもあります。ワークロードで依存関係とのバッチ通信を行う場合は、以下のガイダンスを検討します。

- ワークロードでバッチジョブを実行する時間枠を定義します。ワークロードでは、例えば 1 時間ごとまたは月末に、バッチシステムを呼び出す繰り返しパターンを設定できます。
- データ入力と処理済みデータ出力の場所を定義します。Amazon Simple Storage Service (Amazon S3)、Amazon Elastic File System (Amazon EFS)、Amazon FSx for Lustre などのストレージサービスを使用すると、大規模なワークロードのファイル読み書きに対応できます。
- ワークロードで複数のバッチジョブを呼び出す必要がある場合は、AWS Step Functions を活用して、AWS またはオンプレミスで実行されるバッチジョブのオーケストレーションを簡素化できます。このサンプルプロジェクトは、Step Functions、AWS Batch、および Lambda を使用したバッチジョブのオーケストレーションを示しています。

- バッチジョブをモニタリングして、ジョブの完了に本来よりも時間がかかっているなどの異常がないかを確認します。CloudWatch Container Insights などのツールを使用して、AWS Batch 環境やジョブをモニタリングできます。この場合、ワークロードによって次のジョブの開始が停止し、関連するスタッフに例外が通知されます。

リソース

関連ドキュメント:

- AWS クラウド Operations: モニタリングとオブザーバビリティ
- Amazon Builders' Library: 分散システムの課題
- REL11-BP05 静的安定性を使用してバイモーダル動作を防止する
- AWS Lambda デベロッパーガイド: AWS Lambda でのエラー処理と自動再試行
- レイテンシーを考慮した Amazon DynamoDB アプリケーションのための AWS Java SDK HTTP リクエスト設定のチューニング
- AWS メッセージング
- ストリーミングデータとは
- AWS Lambda デベロッパーガイド: 非同期呼び出し
- Amazon Simple Queue Service に関するよくある質問: FIFO キュー
- Amazon Kinesis Data Streams デベロッパーガイド: 重複レコードの処理
- Amazon Simple Queue Service デベロッパーガイド: Amazon SQS で使用できる CloudWatch メトリクス
- Amazon Kinesis Data Streams デベロッパーガイド: Amazon CloudWatch による Amazon Kinesis Data Streams Service のモニタリング
- AWS X-Ray デベロッパーガイド: AWS X-Rayの概念
- GitHub の AWS サンプル: AWS Step Functions 複雑なオーケストレーターアプリ
- AWS Batch ユーザーガイド: AWS Batch CloudWatch Container Insights

関連動画:

- AWS Summit SF 2022 - AWS によるフルスタックのオブザーバビリティとアプリケーションモニタリング (COP310)

関連ツール:

- Amazon CloudWatch
- Amazon CloudWatch Logs ()
- AWS X-Ray
- Amazon Simple Storage Service (Amazon S3)
- Amazon Elastic File System (Amazon EFS)
- Amazon FSx for Lustre
- AWS Step Functions
- AWS Batch

REL04-BP02 疎結合の依存関係を実装する

キューイングシステム、ストリーミングシステム、ワークフロー、ロードバランサーなどの依存関係は、疎結合されています。疎結合は、コンポーネントの動作をそれに依存する他のコンポーネントから分離するのに役立ち、弾力性と俊敏性を高めます。

キューイングシステム、ストリーミングシステム、ワークフローなどの依存関係を疎結合化すると、システムへの変更や障害の影響を最小限に抑えることができます。疎結合化により、コンポーネントの動作が依存する他のシステムに影響を与えないように分離され、回復力と俊敏性が向上します。

密結合のシステムでは、あるコンポーネントを変更すると、そのコンポーネントに依存する他のコンポーネントも変更しなければならなくなり、結果として、すべてのコンポーネントのパフォーマンスが低下する可能性があります。疎結合はこの依存関係を壊すため、依存コンポーネントが知る必要があるのは、バージョン管理されて公開されたインターフェイスのみです。依存関係があるコンポーネント間に疎結合を実装すると、あるコンポーネントの障害が別のコンポーネントに影響を及ぼさないように隔離することができます。

疎結合では、コードの変更やコンポーネントへの機能の追加を自由にできる一方で、そのコンポーネントに依存する他のコンポーネントへのリスクを最小限に抑えることができます。また、回復力を細分化でき、コンポーネントレベルでスケールアウトしたり、依存関係の根本的な実装さえも変更したりできます。

疎結合によって弾力性をさらに向上させるには、可能な場合はコンポーネント間のやりとりを非同期にします。このモデルは、即時応答を必要とせず、リクエストが登録されていることの確認で十分な状況では、どのような対話にも最適です。イベントを生成するコンポーネントと、イベントを消費するコンポーネントがあります。2 つのコンポーネントは、直接的なポイントツーポイントのやりとりではなく、通常、Amazon SQS キューのような中間的な耐久性の高いストレージレイヤーや

Amazon Kinesis のようなストリーミングデータプラットフォーム、または AWS Step Functions を介して統合されます。

図 4: 疎結合されたキューイングシステムやロードバランサーなどの依存関係

Amazon SQS キューと AWS Step Functions は、疎結合の中間レイヤーを追加する方法のうちの 2 つにすぎません。Amazon EventBridge を使用してイベント駆動型アーキテクチャを AWS クラウドに構築することもできます。Amazon EventBridge は、クライアント (イベントプロデューサー) が依存するサービス (イベントコンシューマー) から抽象化できます。Amazon Simple Notification Service (Amazon SNS) は、高スループットのプッシュベースの多対多メッセージングが必要な場合に効果的なソリューションです。Amazon SNS トピックを使用すると、パブリッシャーシステムは、メッセージを多数のサブスクライバーエンドポイントにファンアウトして、並列処理できます。

キューにはいくつかの利点がありますが、ほとんどのハードリアルタイムシステムでは、しきい値の時間 (多くの場合、数秒) よりも長時間かかっているリクエストは古くなっているとみなされ (クライアントが停止し、応答を待機しなくなる)、処理されません。このように、古くなったリクエストの代わりに、より新しい (そしておそらくまだ有効な) リクエストを処理することができます。

期待される成果: 疎結合の依存関係を実装すると、コンポーネントレベルへの障害の面積を最小限に抑えることができ、問題の診断と解決に役立ちます。また、開発サイクルが簡素化され、チームはモジュールレベルで変更を実装できるようになり、その部分に依存する他のコンポーネントのパフォーマンスに影響は及びません。このアプローチでは、リソースのニーズに基づいてコンポーネントレベルでスケールアウトできるだけでなく、コスト効率良くコンポーネントを活用できるようになります。

一般的なアンチパターン:

- モノリシックワークロードのデプロイ。
- リクエストのフェイルオーバーや非同期処理を行うことはできない状態で、ワークロード層間で直接 API を呼び出す。
- 共有データを使用した密結合。疎結合のシステムでは、共有データベースや他の形で密結合されたデータストレージを介したデータの共有を避ける必要があります。そうしたデータ共有が密結合を持ち込み、スケーラビリティを妨げる可能性があります。
- バックプレッシャーを無視する。ワークロードには、コンポーネントが同じ速度でデータを処理できない場合に、データの受信を遅らせたり停止したりする機能が必要です。

このベストプラクティスを活用するメリット: 疎結合は、コンポーネントの動作をそれに依存する他のコンポーネントから隔離するのに役立ち、弾力性と俊敏性を高めます。1 つのコンポーネントの障害は他のコンポーネントから隔離されます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

疎結合の依存関係を実装します。疎結合のアプリケーションを構築するためのさまざまなソリューションがあります。フルマネージド型のキュー、自動化されたワークフロー、イベントへの対応、API などを実装するサービスがこれに当たりますが、いずれも、コンポーネントの動作を他のコンポーネントから隔離するのに役立ち、弾力性と俊敏性を高めます。

- イベント駆動型アーキテクチャの構築: Amazon EventBridge は、疎結合で分散されたイベント駆動型アーキテクチャの構築に役立ちます。
- 分散システムにキューを実装する: Amazon Simple Queue Service (Amazon SQS) を使用して、分散システムを統合および疎結合化できます。
- コンポーネントをマイクロサービスとしてコンテナ化: マイクロサービスを使用すると、チームは十分に定義された API を通して通信する小型の独立コンポーネントから構成されるアプリケーションを構築できます。Amazon Elastic Container Service (Amazon ECS) および Amazon Elastic Kubernetes Service (Amazon EKS) は、コンテナ化をすばやく実現できます。
- Step Functions を使用してワークフローを管理する: Step Functions は、複数の AWS サービスを柔軟なワークフローに調整するのに役立ちます。
- パブリッシュ/サブスクライブ (pub/sub) メッセージングアーキテクチャを活用する: Amazon Simple Notification Service (Amazon SNS) は、パブリッシャーからサブスクライバー (プロデューサーおよびコンシューマーとも呼ばれます) へのメッセージ配信を提供します。

実装手順

- イベント駆動型アーキテクチャのコンポーネントは、イベントによって開始されます。イベントは、ユーザーがカートに商品を追加するなど、システム内で発生するアクションです。アクションが正常に実行されると、システムの次のコンポーネントを起動するイベントが生成されます。
  - Amazon EventBridge を使用したイベント駆動型アプリケーションの構築
  - AWS re:Invent 2022: Amazon EventBridge を使用したイベント駆動型統合の設計
- 分散型メッセージングシステムには、キューベースのアーキテクチャを確立するために主に 3 つの部分を実装する必要があります。これには、分散型システムのコンポーネント、分離のために使用するキュー (Amazon SQS サーバー上で分散される)、キュー内のメッセージが該当します。

一般的なシステムには、キューへのメッセージ配信を開始するプロデューサーと、キューからメッセージを受信するコンシューマーがあります。キューは、冗長性を確保するために、複数の Amazon SQS サーバーにメッセージを格納します。

  - Amazon SQS の基本的なアーキテクチャ
  - Amazon Simple Queue Service を使用して分散アプリケーション間でメッセージを送信する
- マイクロサービスをうまく利用すれば、疎結合のコンポーネントが独立したチームによって管理されるため、保守性とスケーラビリティが向上します。また、変更が必要になっても、動作を分離し、単一のコンポーネントに限定できます。
  - AWS でのマイクロサービスの実装
  - Let's Architect! コンテナを使用するマイクロサービスの設計
- AWS Step Functions では、分散型アプリケーションの構築、プロセスの自動化、マイクロサービスのオーケストレーションなどを行うことができます。複数のコンポーネントをオーケストレーションして 1 つのワークフローとしてまとめ、自動化することで、アプリケーション内の依存関係を分離できます。
  - AWS Step Functions と AWS Lambda を使用してサーバーレスワークフローを作成する
  - AWS Step Functions の開始方法

リソース

関連ドキュメント:

- Amazon EC2: べき等性を保証する
- Amazon Builders' Library: 分散システムの課題
- Amazon Builders' Library: 信頼性、動作の継続、1 杯のおいしいコーヒー
- What Is Amazon EventBridge?
- Amazon Simple Queue Service とは
- モノリスから卒業する
- AWS Step Functions および Amazon SQS を使用してキューベースのマイクロサービスをオーケストレーションする
- Amazon SQS の基本的なアーキテクチャ
- キューベースのアーキテクチャ

関連動画:

- AWS New York Summit 2019: イベント駆動型アーキテクチャと Amazon EventBridge 入門 (MAD205)
- AWS re:Invent 2018: ループを閉じ、発想を開く: 大小さまざまなシステムをコントロールする方法 ARC337 (疎結合、継続動作、静的安定性を含む)
- AWS re:Invent 2019: イベント駆動型アーキテクチャへの移行 (SVS308)
- AWS re:Invent 2019: Amazon SQS と Lambda を使用するスケーラブルなサーバーレスイベント駆動型アプリケーション
- AWS re:Invent 2022: Amazon EventBridge を使用したイベント駆動型統合の設計
- AWS re:Invent 2017: Elastic Load Balancing の詳細とベストプラクティス

REL04-BP03 継続動作を行う

負荷が急激に大きく変化すると、システム障害が発生することがあります。例えば、ワークロードで何千台ものサーバーのヘルスをモニタリングするヘルスチェックを実行する場合、毎回同じサイズのペイロード (現在の状態の完全なスナップショット) を送信しています。障害が発生しているサーバーがなくても、またはそのすべてに障害が発生していても、ヘルスチェックシステムは、大規模で急激な変更なしに常に作業を行っています。

例えば、ヘルスチェックシステムが 100,000 台のサーバーをモニタリングしている場合、通常のサーバー障害率が軽いときは、その負荷はわずかです。しかし、重大なイベントによってこれらのサーバーの半分が異常な状態になると、ヘルスチェックシステムは、通知システムを更新し、クライアントに状態を通知しようとして過負荷になるでしょう。したがって、ヘルスチェックシステムは毎回現在の状態のフルスナップショットを送信する必要があります。それぞれがビットで表される 100,000 個のサーバーヘルス状態は、12.5 KB のペイロードにすぎません。サーバーに障害が発生していないか、またはすべてに発生しているかにかかわらず、ヘルスチェックシステムは定期的に作業を行っているため、大規模の急激な変化はシステムの安定性を脅かすものではありません。これは実際に Amazon Route 53 がエンドポイントのヘルスチェック (IP アドレスなど) によってエンドユーザーがどのようにルーティングされているかを調べる際の方法です。

このベストプラクティスを活用しない場合のリスクレベル: 低

実装のガイダンス

- 負荷が急激に変化してシステム障害が発生しないように、継続動作を行います。
- 疎結合の依存関係を実装します。キューイングシステム、ストリーミングシステム、ワークフロー、ロードバランサーなどの依存関係は、疎結合されています。疎結合は、コンポーネントの動作をそれに依存する他のコンポーネントから分離するのに役立ち、弾力性と俊敏性を高めます。

- Amazon Builders' Library: 信頼性、動作の継続、1 杯のおいしいコーヒー
- AWS re:Invent 2018: ループを閉じ、発想を開く: 大小さまざまなシステムをコントロールする方法 ARC337 (継続動作を含む)
  - 100,000 台のサーバーをモニタリングするヘルスチェックシステムの例の場合、成功または失敗の数に関係なく、ペイロードサイズが一定になるように、ワークロードを設計します。

リソース

関連ドキュメント:

- Amazon EC2: べき等性を保証する
- Amazon Builders' Library: 分散システムの課題
- Amazon Builders' Library: 信頼性、動作の継続、1 杯のおいしいコーヒー

関連動画:

- AWS New York Summit 2019: イベント駆動型アーキテクチャと Amazon EventBridge 入門 (MAD205)
- AWS re:Invent 2018: ループを閉じ、発想を開く: 大小さまざまなシステムをコントロールする方法 ARC337 (継続動作を含む)
- AWS re:Invent 2018: ループを閉じ、発想を開く: 大小さまざまなシステムをコントロールする方法 ARC337 (疎結合、継続動作、静的安定性を含む)
- AWS re:Invent 2019: イベント駆動型アーキテクチャへの移行 (SVS308)

REL04-BP04 変更操作をべき等にする

べき等性のサービスは、各リクエストが 1 回だけ処理することを約束します。そのため、同一のリクエストを複数回行っても、リクエストを 1 回行ったのと同じ効果しかありません。これにより、リクエストが誤って複数回処理されることを恐れる必要がなくなるため、クライアントが再試行しやすくなります。これを行うには、クライアントは、リクエストが繰り返されるたびに使用されるべき等性トークンを使用して API リクエストを発行できます。べき等性サービス API は、システムの基盤となる状態が変更された場合でも、トークンを使用してリクエストが最初に完了したときに返された応答と同じ応答を返します。

分散システムでは、アクションを最大で 1 回 (クライアントがリクエストを 1 回だけ行う)、または少なくとも 1 回 (クライアントが成功を確認するまでリクエストを続ける) 実行するのは比較的簡単です。同一のリクエストを複数回行っても、リクエストを 1 回行ったのと同じ効果を持つように、アクションが確実に 1 回実行されることを保証するのはより困難です。API でべき等性トークンを使用すると、サービスは、重複レコードや副作用を生むことなく、変更リクエストを 1 回または複数回受け取ることができます。

期待される成果: すべてのコンポーネントとサービスにわたってべき等性を保証するための、一貫性があり、十分に文書化され、広く採用されているアプローチが得られます。

一般的なアンチパターン:

- 必要でない場合でも、無差別にべき等性を適用している。
- べき等性を実装するための過度に複雑なロジックを導入している。
- タイムスタンプは、べき等性のキーとして使用している。これにより、クロックスキューや、複数のクライアントが同じタイムスタンプを使用して変更を適用することが原因で、不正確さが生じる可能性があります。
- べき等性を保つためにペイロード全体を保存している。このアプローチでは、リクエストごとに完全なデータペイロードを保存し、新しいリクエストごとに上書きします。これにより、パフォーマンスが低下し、スケーラビリティに影響する可能性があります。
- サービス間でキーの生成に一貫性がない。一貫したキーがないと、サービスは重複するリクエストを認識しない可能性があり、意図しない結果になる可能性があります。

このベストプラクティスを活用するメリット:

- スケーラビリティの向上: システムは、追加のロジックや複雑な状態管理を実行することなく、再試行や重複したリクエストを処理できます。
- 信頼性の向上: べき等性により、サービスは複数の同一リクエストを一貫した方法で処理できるようになり、予期しない副作用や重複レコードのリスクが軽減されます。これは、ネットワーク障害や再試行が頻繁に発生する分散システムでは特に重要です。
- データ整合性の向上: 同じリクエストが同じレスポンスを生成するため、べき等性は分散システム間でデータ整合性を維持するのに役立ちます。これは、トランザクションとオペレーションの整合性を維持するために不可欠です。
- エラー処理: べき等性トークンを使用すると、エラー処理がより簡単になります。問題によりクライアントが応答を受信しなかった場合、同じべき等性トークンを使用してリクエストを安全に再送信できます。

- 運用上の透明性: べき等性を使用すると、モニタリングと ログ記録が向上します。サービスは、べき等性トークンを使用してリクエストをログに記録できるため、問題の追跡とデバッグが容易になります。
- API 契約の簡素化: クライアント側とサーバー側のシステム間の契約を簡素化し、誤ったデータ処理が行われる恐れを軽減します。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

分散システムでは、アクションを最大 1 回 (クライアントは 1 つのリクエストのみを行う) または少なくとも 1 回 (クライアントは成功が確認されるまでリクエストを続ける) 実行するのは比較的簡単です。ただし、確実に 1 回だけ動作を実装するのは困難です。これを実現するには、クライアントはリクエストごとにべき等性トークンを生成して提供する必要があります。

べき等性トークンを使用することによって、サービスは新しいリクエストと繰り返しのリクエストを区別できます。サービスがべき等性トークンを含むリクエストを受信すると、そのトークンが既に使用されているかどうかが確認されます。トークンが使用されている場合、サービスは保存されたレスポンスを取得して返します。トークンが新しい場合、サービスはリクエストを処理し、レスポンスをトークンと共に保存し、レスポンスを返します。このメカニズムにより、すべてのレスポンスがべき等になり、分散システムの信頼性と一貫性が向上します。

べき等性は、イベント駆動型アーキテクチャの重要な動作でもあります。これらのアーキテクチャは通常、Amazon SQS、Amazon MQ、Amazon Kinesis Streams、Amazon Managed Streaming for Apache Kafka (MSK) などのメッセージキューによってサポートされます。状況によっては、1 回だけ公開されたメッセージが誤って複数回配信されることがあります。パブリッシャーがべき等性トークンを生成してメッセージに含める場合、受信した重複メッセージの処理によって、同じメッセージに対するアクションが繰り返されないように要求します。コンシューマーは受信した各トークンを追跡し、重複したトークンを含むメッセージを無視する必要があります。

サービスとコンシューマーは、受信したべき等性トークンを、呼び出すダウンストリームサービスにも渡す必要があります。同様に、処理チェーン内のすべてのダウンストリームサービスは、メッセージを複数回処理することの副作用を避けるために、べき等性が実装されていることを確認する責任があります。

実装手順

1. べき等性オペレーションを特定する

どのオペレーションにべき等性が必要かを判断します。通常、これには POST、PUT、DELETE HTTP メソッドとデータベースの挿入、更新、または削除オペレーションが含まれます。読み取り専用クエリなど、状態を変更しない操作では、副作用がない限り、通常はべき等性は必要ありません。

2. 一意の識別子を使用する

   送信者から送信される各べき等オペレーションのリクエストには、リクエストに直接、またはメタデータ (HTTP ヘッダーなど) の一部として、一意のトークンを含めます。これにより、受信者は重複するリクエストやオペレーションを認識して処理できます。トークンに一般的に使用される識別子には、Universally Unique Identifiers (UUID) と K-Sortable Unique Identifiers (KSUID) があります。

3. 状態の追跡と管理

   ワークロード内の各オペレーションまたはリクエストの状態を維持します。これは、べき等性トークンと対応する状態 (保留中、完了、失敗など) をデータベース、キャッシュ、またはその他の永続的ストアに保存することによって実現できます。この状態情報により、ワークロードは重複するリクエストやオペレーションを識別して処理できます。

   必要に応じて、ロック、トランザクション、オプティミスティック同時実行コントロールなどの適切な同時実行コントロールメカニズムを使用して、一貫性と原子性を維持します。これには、べき等性トークンを記録し、リクエストの処理に関連するすべての変更操作を実行するプロセスが含まれます。これにより、競合状態を防ぎ、べき等操作が正しく実行されることを確認できます。

   ストレージとパフォーマンスを管理するために、データストアから古いべき等性トークンを定期的に削除します。ストレージシステムがサポートしている場合は、データの有効期限タイムスタンプ (有効期限または TTL 値とも呼ばれます) の使用を検討してください。べき等性トークンの再利用の可能性は時間の経過と共に減少します。

   べき等性トークンと関連する状態を保存するために通常使用される一般的な AWS ストレージオプションは次のとおりです。

   - Amazon DynamoDB: DynamoDB は、低レイテンシーのパフォーマンスと高可用性を提供する NoSQL データベースサービスであり、べき等性関連データの保存に最適です。DynamoDB のキー値およびドキュメントデータモデルにより、べき等性トークンと関連する状態情報を効率的に保存および取得できます。また、アプリケーションが挿入時に TTL 値を設定すると、DynamoDB はべき等性トークンを自動的に期限切れにすることもできます。

- Amazon ElastiCache: ElastiCache は、高スループット、低レイテンシー、低コストでべき等性トークンを保存できます。ElastiCache (Redis) と ElastiCache (Memcached) の両方とも、アプリケーションが挿入時に TTL 値を設定すると、べき等性トークンを自動的に期限切れにすることもできます。
- Amazon Relational Database Service (RDS): 特にアプリケーションが他の目的でリレーショナルデータベースを既に使用している場合は、Amazon RDS を使用してべき等性トークンと関連する状態情報を保存できます。
- Amazon Simple Storage Service (S3): Amazon S3 は、べき等性トークンと関連メタデータの保存に使用できる、スケーラビリティと耐久性の高いオブジェクトストレージサービスです。S3 のバージョニング機能は、べき等操作の状態を維持するのに特に役立ちます。ストレージサービスの選択は、通常、べき等性関連データの量、必要なパフォーマンス特性、耐久性と可用性の必要性、べき等性メカニズムが全体的なワークロードアーキテクチャとどのように統合されるかなどの要因によって決まります。

4. べき等性オペレーションを実装する

   API とワークロードコンポーネントをべき等になるように設計します。べき等性チェックをワークロードコンポーネントに組み込みます。リクエストを処理する前、またはオペレーションを実行する前に、一意の識別子が既に処理されているかどうかを確認します。存在する場合は、オペレーションを再度実行する代わりに、前の結果を返します。例えば、クライアントがユーザーの作成リクエストを送信した場合、同じ一意の識別子を持つユーザーが既に存在するかどうかを確認します。ユーザーが存在する場合は、新しいユーザー情報を作成する代わりに、既存のユーザー情報を返す必要があります。同様に、キューコンシューマーが重複したべき等性トークンを含むメッセージを受信した場合、コンシューマーはそのメッセージを無視する必要があります。

   リクエストのべき等性を検証する包括的なテストスイートを作成します。成功したリクエスト、失敗したリクエスト、重複したリクエストなど、幅広いシナリオをカバーする必要があります。

   ワークロードが AWS Lambda 関数を活用する場合は、AWS Lambda の Powertools を検討してください。Powertools for AWS Lambda は、サーバーレスのベストプラクティスを実装し、AWS Lambda 関数を操作する際のデベロッパーの速度を向上させるデベロッパーツールキットです。特に、Lambda 関数を再試行しても安全なべき等操作に変換するユーティリティを提供します。

5. べき等性を明確に伝える

   API とワークロードコンポーネントをドキュメント化して、操作のべき等性を明確に伝えます。これにより、クライアントは予想される動作と、ワークロードと確実にやり取りする方法を理解できます。

6. モニタリングと監査

モニタリングと監査のメカニズムを実装して、予期しないレスポンスの変動や過剰な重複リクエストの処理など、レスポンスのべき等性に関連する問題を検出します。これにより、ワークロードの問題や予期しない動作を検出して調査できます。

リソース

関連するベストプラクティス:

- REL05-BP03 再試行呼び出しを制御および制限する
- REL06-BP01 ワークロードのすべてのコンポーネントをモニタリングする (生成)
- REL06-BP03 通知を送信する (リアルタイム処理とアラーム)
- REL08-BP02 デプロイの一部として機能テストを統合する

関連ドキュメント:

- The Amazon Builders' Library: Making retries safe with idempotent APIs
- Amazon Builders' Library: 分散システムの課題
- Amazon Builders' Library: 信頼性、動作の継続、1 杯のおいしいコーヒー
- Amazon Elastic Container Service: Ensuring idempotency
- Lambda 関数をべき等にするにはどうすればよいですか?
- Ensuring idempotency in Amazon EC2 API requests

関連動画:

- Building Distributed Applications with Event-driven Architecture - AWS Online Tech Talks
- AWS re:Invent 2023 - Building next-generation applications with event-driven architecture
- AWS re:Invent 2023 - Advanced integration patterns & trade-offs for loosely coupled systems
- AWS re:Invent 2023 - Advanced event-driven patterns with Amazon EventBridge
- AWS re:Invent 2018 - Close Loops and Opening Minds: How to Take Control of Systems, Big and Small ARC337 (includes loose coupling, constant work, static stability)
- AWS re:Invent 2019 - Moving to event-driven architectures (SVS308)

関連ツール:

- Idempotency with AWS Lambda Powertools (Java)
- Idempotency with AWS Lambda Powertools (Python)
- AWS Lambda Powertools GitHub page

障害を軽減するため、または障害に耐えるために分散システムでの相互作用を設計する

分散システムは、サーバーやサービスなどのコンポーネントを相互接続するために通信ネットワークを利用しています。このネットワークでデータの損失やレイテンシーがあっても、ワークロードは確実に動作する必要があります。分散システムのコンポーネントは、他のコンポーネントやワークロードに悪影響を及ぼさない方法で動作する必要があります。これらのベストプラクティスに従うことで、ワークロードはストレスや障害に耐え、より迅速に復旧し、障害の影響を軽減できます。これにより、平均復旧時間 (MTTR) が向上します。

これらのベストプラクティスは障害を防ぎ、平均故障間隔 (MTBF) を改善します。

ベストプラクティス

- REL05-BP01 該当するハードな依存関係をソフトな依存関係に変換するため、グレースフルデグラデーションを実装する
- REL05-BP02 リクエストのスロットル
- REL05-BP03 再試行呼び出しを制御および制限する
- REL05-BP04 フェイルファストとキューの制限
- REL05-BP05 クライアントタイムアウトを設定する
- REL05-BP06 可能な限りシステムをステートレスにする
- REL05-BP07 緊急レバーを実装する

REL05-BP01 該当するハードな依存関係をソフトな依存関係に変換するため、グレースフルデグラデーションを実装する

アプリケーションコンポーネントは、依存関係が使用できなくなっても、引き続きコア機能を実行する必要があります。少し古いデータ、代替データ、またはまったくデータを提供していない可能性があります。これにより、局所的な障害によるシステム全体の機能への影響を最小限に抑えながら、中心的なビジネス価値を提供できます。

期待される成果: コンポーネントの依存関係が異常な場合でも、コンポーネント自体は機能しますが、パフォーマンスが低下します。コンポーネントの故障モードは通常の動作とみなしてください。ワークフローは、このような障害が完全な障害につながらないように、あるいは少なくとも予測可能で回復可能な状態になるように設計する必要があります。

一般的なアンチパターン:

- 必要な中核的なビジネス機能が特定されていない。依存関係に障害が発生してもコンポーネントが機能することをテストしていません。
- エラーに関するデータを提供しない場合や、複数の依存関係のうち 1 つしか使用できず、結果の一部が返される場合もあります。
- トランザクションが部分的に失敗すると、一貫性のない状態になる。
- 中央パラメータストアにアクセスする代替手段がない。
- 更新に失敗した結果、その結果を考慮せずにローカルステートを無効化または空にする。

このベストプラクティスを活用するメリット: グレースフルデグラデーションを行うと、システム全体の可用性が向上し、障害が発生しても最も重要な機能の機能が維持されます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

グレースフルデグラデーションを実装することで、依存関係の障害がコンポーネントの機能に与える影響を最小限に抑えることができます。コンポーネントが依存関係の障害を検出し、他のコンポーネントや顧客への影響を最小限に抑える方法で回避するのが理想的です。

グレースフルデグラデーションを考慮した設計とは、依存関係の設計時に潜在的な障害モードを考慮することを意味します。障害モードごとに、コンポーネントのほとんどの機能、または少なくとも最も重要な機能を発信者または顧客に提供する方法を用意してください。これらの考慮事項は、テストや検証が必要な追加要件になる可能性があります。理想的には、1 つまたは複数の依存関係に障害が発生した場合でも、コンポーネントがコア機能を許容範囲内で実行できることが理想的です。

これは技術的な議論であると同時にビジネス上の議論でもあります。すべてのビジネス要件は重要であり、可能であれば満たす必要があります。ただし、すべてが満たされない場合に何が起こるかをたずねることは依然として理にかなっています。システムは可用性と一貫性を保つように設計できますが、1 つの要件を削除しなければならない状況では、どちらの要件がより重要でしょうか。支払い処理については、一貫性があるかもしれません。リアルタイムアプリケーションの場合、可用性が高く

なる可能性があります。カスタマー向けウェブサイトの場合、答えはカスタマーの期待するものによって異なる場合があります。

これが何を意味するかは、コンポーネントの要件と、そのコア機能とみなすべき内容によって異なります。例えば、次のようになります。

- e コマースウェブサイトでは、パーソナライズされたレコメンデーション、上位ランクの商品、顧客の注文状況など、複数の異なるシステムからのデータがランディングページに表示される場合があります。上流システムの 1 つに障害が発生した場合でも、エラーページを顧客に表示するのではなく、他のシステムすべてを表示する方が理にかなっています。
- バッチ書き込みを実行するコンポーネントは、個々の操作のいずれかが失敗した場合でも、バッチの処理を続行できます。再試行メカニズムを実装するのは簡単なはずです。これは、どの操作が成功し、どの操作が失敗したか、なぜ失敗したかについての情報を呼び出し元に返すか、失敗したリクエストをデッドレターキューに入れて非同期再試行を実装することで実現できます。失敗した操作に関する情報も記録する必要があります。
- トランザクションを処理するシステムは、個々の更新がすべて実行されたか、まったく実行されないかを確認する必要があります。分散トランザクションでは、同じトランザクションの後の操作が失敗した場合に備えて、Saga パターンを使用して以前の操作をロールバックできます。ここでの中心的な機能は一貫性を維持することです。
- タイムクリティカルなシステムは、タイムリーに応答しない依存関係に対処しなければなりません。このような場合は、サーキットブレーカーパターンを使用できます。依存関係からの応答がタイムアウトし始めると、システムは追加の呼び出しが行われないクローズ状態に切り替えることができます。
- アプリケーションはパラメータストアからパラメータを読み取ることができます。デフォルトのパラメータセットを使用してコンテナイメージを作成し、パラメータストアが利用できない場合にこれらを使用すると便利です。

なお、コンポーネントに障害が発生した場合の経路は検査が必要で、主要経路よりも大幅に簡潔でなければなりません。一般的には、フォールバック戦略は避けるべきです。

実装手順

外部依存関係と内部依存関係を特定します。どのような種類の障害が発生する可能性があるかを検討してください。障害発生時に上流と下流のシステムやカスタマーへの悪影響を最小限に抑える方法を考えてください。

依存関係の一覧と、失敗した場合に正常にデグレードする方法は次のとおりです。

1. 依存関係の部分的な障害: コンポーネントは、1 つのシステムへの複数の要求、または複数のシステムへの 1 つの要求のいずれかとして、下流システムに対して複数の要求を行うことができます。ビジネスの状況によっては、これに対するさまざまな処理方法が適切な場合があります (詳細については、実装ガイダンスの前述の例を参照してください)。
2. 高負荷のためにダウンストリームシステムがリクエスト処理不可: ダウンストリームシステムへのリクエストが一貫して失敗している場合、再試行を続けることは意味がありません。これにより、既に過負荷になっているシステムに追加の負荷がかかり、回復が困難になる可能性があります。ここでは、ダウンストリームシステムへのコールの失敗を監視するサーキットブレーカーパターンを利用できます。大量のコールが失敗すると、ダウンストリームシステムへのリクエストの送信が停止され、ダウンストリームシステムが再び使用可能かどうかをテストするコールがたまにしか送信されません。
3. パラメータストアが使用不可: パラメータストアを変換するには、ソフト依存関係キャッシュを使用するか、コンテナイメージやマシンイメージに含まれる適切なデフォルトを使用できます。これらのデフォルトは最新の状態に保ち、テストスイートに含める必要があることに注意してください。
4. モニタリングサービスまたは非機能的依存関係が停止: コンポーネントが断続的にログ、メトリクス、またはトレースを中央監視サービスに送信できない場合でも、通常どおりビジネス機能を実行するのが最善策です。メトリクスを長時間ログに記録したりプッシュしたりしないことは、ほとんどの場合受け入れられません。また、ユースケースによっては、コンプライアンス要件を満たすために完全な監査エントリが必要になる場合があります。
5. リレーショナルデータベースのプライマリインスタンスが停止している可能性がある: Amazon Relational Database Service は、ほぼすべてのリレーショナルデータベースと同様に、プライマリライターインスタンスを 1 つだけ持つことができます。これにより、書き込みワークロードの単一障害点が生じ、スケーリングがより困難になります。これは、可用性を高めるためにマルチ AZ 構成を使用するか、スケーリングを向上させるために Amazon Aurora Serverless 構成を使用することで部分的に軽減できます。可用性要件が非常に高い場合は、プライマリライターにまったく依存しない方が理にかなっています。読み取り専用のクエリには、リードレプリカを使用できます。これにより、冗長性が確保され、スケールアップだけでなくスケールアウトも可能です。書き込みは、例えば、Amazon Simple Queue Service キューにバッファリングできるため、プライマリが一時的に使用できなくなっても、カスタマーからの書き込み要求を引き続き受け付けることができます。

リソース

関連ドキュメント:

- Amazon API Gateway: API リクエストを調整してスループットを向上させる
- Circuit Breaker (「Release It!」書籍よりサーキットブレーカーをまとめたもの)
- AWS でのエラーの再試行とエクスポネンシャルバックオフ
- Michael Nygard 著「Release It! Design and Deploy Production-Ready Software」
- The Amazon Builders' Library: 分散システムでのフォールバックの回避
- Amazon Builders' Library: 乗り越えられないキューバックログの回避
- The Amazon Builders' Library: キャッシングの課題と戦略
- The Amazon Builders' Library: ジッターを伴うタイムアウト、再試行、およびバックオフ

関連動画:

- Retry, backoff, and jitter: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)

REL05-BP02 リクエストのスロットル

リクエストを制限して、予想外の需要の増加によるリソースの枯渇を緩和します。スロットリングレートを下回るリクエストは処理されますが、定義された制限を超えるリクエストは拒否され、リクエストがスロットリングされたことを示すメッセージが返されます。

期待される成果: 突然のカスタマートラフィックの増加、フラッディング攻撃、または再試行ストームによる大量のスパイクは、リクエストスロットリングによって軽減され、サポートされているリクエスト量の通常の処理をワークロードが継続できるようになります。

一般的なアンチパターン:

- API エンドポイントのスロットルは実装されていないか、予想される量を考慮せずにデフォルト値のままになっています。
- API エンドポイントは負荷テストされておらず、スロットリング制限もテストされていません。
- リクエストのサイズや複雑さを考慮せずにリクエストレートをスロットリングできます。
- 最大リクエストレートまたは最大リクエストサイズをテストしますが、両方を一緒にテストするわけではありません。
- リソースは、テストで設定したのと同じ制限にプロビジョニングされません。
- アプリケーション (A2A) API コンシューマーへの適用を目的とした使用プランは設定も検討もされていません。
- 水平方向にスケールするキューコンシューマーには、最大同時実行設定は設定されていません。

- IP アドレスごとのレート制限は実装されていません。

このベストプラクティスを活用するメリット: スロットル制限を設定したワークロードは、予期しない量のスパイクが発生しても、正常に動作し、受け入れられたリクエストの負荷を正常に処理できます。API やキューへのリクエストの急なスパイクや持続的なスパイクはスロットリングされ、リクエスト処理リソースを使い果たすことはありません。レート制限は、単一の IP アドレスまたは API コンシューマーからの大量のトラフィックがリソースを使い果たして他のコンシューマーに影響を与えないように、個々のリクエスタを制限します。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

サービスは、既知のキャパシティのリクエストを処理するように設計する必要があります。このキャパシティは、負荷テストによって確立できます。リクエストの到着率が制限を超えると、適切なレスポンスからリクエストがスロットリングされたことが通知されます。これにより、コンシューマーはエラーを処理して後で再試行できます。

サービスにスロットリングの実装が必要な場合は、トークンがリクエストにカウントされるトークンバケットアルゴリズムの実装を検討してください。トークンは 1 秒あたりのスロットルレートで補充され、リクエストごとに 1 つのトークンで非同期に空になります。

トークンバケットアルゴリズム。

Amazon API Gateway は、アカウントとリージョンの制限に則ってトークンバケットアルゴリズムを実装します。使用プランに従ってクライアントごとに設定することが可能です。さらに、Amazon

Simple Queue Service (Amazon SQS) と Amazon Kinesis を使用することで、リクエストをバッファリングしてリクエストレートを均衡にし、対処可能なリクエストのスロットリングレートを高めることができます。最後に、AWS WAF を使用してレート制限を実装することで、異常に高い負荷を発生させる特定の API コンシューマーをスロットリングします。

実装手順

API Gateway で API のスロットリング制限を設定し、制限を超過したときに「`429 Too Many Requests`」エラーを返すようにします。AWS AppSync および API Gateway エンドポイントで AWS WAF を使用すれば、IP アドレスごとにレート制限を有効にできます。さらに、システムが非同期処理に対応できる場合は、メッセージをキューまたはストリームに入れてサービスクライアントへの応答を高速化できます。これにより、より高いスロットルレートにバーストできます。

非同期処理の場合、Amazon SQS を AWS Lambda のイベントソースとして設定しているときは、最大同時実行数を設定することで、イベント率の上昇によって、ワークロードやアカウント内の他のサービスに必要な、使用可能なアカウントの同時実行クォータが消費されることを回避できます。

API Gateway ではトークンバケットのマネージド実装が行われますが、API Gateway を使用できない場合は、お使いのサービス用のトークンバケットの、言語固有のオープンソース実装 (「参考文献」内の「関連する例」を参照) を利用できます。

- API Gateway のスロットリング制限は、リージョンごとのアカウントレベル、ステージごとの API、使用プランのレベルごとの API キーで理解し、設定します。
- AWS WAF レート制限ルールを API Gateway および AWS AppSync エンドポイントに適用してフラッドから保護し、悪意のある IP をブロックします。A2A コンシューマー向けの AWS AppSync API キーにレート制限ルールを設定することもできます。
- AWS AppSync API のレート制限よりも高度なスロットリング制御が必要かどうかを検討し、必要な場合は AWS AppSync エンドポイントの前に API Gateway を設定します。
- Amazon SQS キューが Lambda キューコンシューマーのトリガーとして設定されているときは、最大同時実行数は、サービスレベルの目標達成に十分に対応できる値、かつ他の Lambda 関数に影響を与える同時実行の制限を消費しない値に設定します。Lambda でキューを使用する場合は、同じアカウントおよびリージョン内の他の Lambda 関数に予約された同時実行を設定することを検討します。
- API Gateway を、Amazon SQS または Kinesis とのネイティブサービス統合と共に使用して、リクエストをバッファリングします。

- API Gateway を使用できない場合は、言語固有のライブラリを調べて、ワークロード用のトークンバケットアルゴリズムを実装してください。サンプルセクションを確認して、適切なライブラリを見つけるために独自の調査を行ってください。
- 設定する予定の、または引き上げを許可する予定の制限をテストし、テストした制限を文書化します。
- テストで設定した上限を超えて制限を増やさないでください。制限を増やす場合は、増やす前に、プロビジョニングされたリソースが既にテストシナリオのものと同等かそれ以上であることを確認してください。

リソース

関連するベストプラクティス:

- REL04-BP03 継続動作を行う
- REL05-BP03 再試行呼び出しを制御および制限する

関連ドキュメント:

- Amazon API Gateway: スループットを高めるために API リクエストをスロットリングする
- AWS WAF: レートベースのルールステートメント
- Introducing maximum concurrency of AWS Lambda when using Amazon SQS as an event source
- AWS Lambda: 最大同実行数

関連する例:

- The three most important AWS WAF rate-based rules
- Java Bucket4j
- Python token-bucket
- Node token-bucket
- .NET System Threading Rate Limiting

関連動画:

- Implementing GraphQL API security best practices with AWS AppSync

関連ツール:

- Amazon API Gateway
- AWS AppSync
- Amazon Simple Queue Service
- Amazon Kinesis
- AWS WAF
- Virtual Waiting Room on AWS

REL05-BP03 再試行呼び出しを制御および制限する

エクスポネンシャルバックオフを使用して、各再試行の間隔を徐々に長くしてリクエストを再試行します。再試行間隔をランダム化するために、再試行間にジッターを導入します。最大再試行回数を制限します。

期待される成果: 分散ソフトウェアシステムの一般的なコンポーネントには、サーバー、ロードバランサー、データベース、DNS サーバーが含まれます。通常の操作では、これらのコンポーネントは一時的なエラーや限定的なエラーを含むリクエストに応答できます。また、再試行してもエラーが続くリクエストにも応答できます。クライアントがサービスにリクエストを行うと、そのリクエストはメモリ、スレッド、接続、ポート、またはその他の限られたリソースを含むリソースを消費します。再試行の制御と制限は、リソースを解放して消費を最小限に抑え、負荷がかかっているシステムコンポーネントに負荷がかからないようにするための戦略です。

クライアントのリクエストがタイムアウトになったり、エラーレスポンスが返されたりした場合は、再試行するかどうかを決定する必要があります。再試行する場合は、ジッターと最大再試行値によるエクスポネンシャルバックオフを行います。その結果、バックエンドのサービスとプロセスの負荷が軽減され、自己修復にかかる時間が短縮されるため、復旧が速くなり、リクエストサービスが正常に処理されます。

一般的なアンチパターン:

- エクスポネンシャルバックオフ、ジッター、最大再試行値を追加せずに再試行を実装します。バックオフとジッターは、同じ間隔で意図せずに調整された再試行による人為的なトラフィックのスパイクを防ぐのに役立ちます。
- その効果をテストしたり、再試行シナリオをテストせずに再試行が既に SDK に組み込まれていたりすることを前提に再試行を実装します。

- 公開されている依存関係のエラーコードを理解できず、許可の欠如、設定エラー、または手動による介入なしでは解決できないと思われるその他の状態を示す明確な原因がある エラーを含め、すべてのエラーを再試行することになります。
- 根本的な問題を明らかにして対処できるように、繰り返し発生するサービス障害の監視や警告など、オブザーバビリティのプラクティスには触れていません。
- 組み込みまたはサードパーティーの再試行機能で十分な場合は、カスタムの再試行メカニズムを開発します。
- アプリケーションスタックの複数のレイヤーで再試行すると、再試行が複雑になり、再試行の大混乱の中でさらにリソースを消費します。これらのエラーが依存しているアプリケーションの依存関係にどのように影響するかを必ず理解し、再試行は 1 つのレベルでのみ実装してください。
- べき等性を持たないサービスコールを再試行すると、結果が重複するなどの予期しない影響が発生します。

このベストプラクティスを活用するメリット: 再試行は、リクエストが失敗したときにクライアントが希望する結果を得るのに役立ちますが、必要な応答を得るまでにサーバーの時間を多く消費します。障害がまれな場合や一時的な場合は、再試行しても問題ありません。リソースの過負荷が原因で障害が発生した場合、再試行は事態を悪化させる可能性があります。クライアントの再試行にジッターを伴うエクスポネンシャルバックオフを追加することで、リソース過負荷が原因で障害が発生した場合でも、サーバーを回復できます。ジッターを使用すると、リクエストがスパイクに陥るのを防ぎ、バックオフによって通常のリクエスト負荷に再試行を追加することによる負荷のエスカレーションが軽減されます。最後に、メタステーブル障害の原因となるバックログが作成されないように、最大再試行回数または経過時間を設定することが重要です。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

再試行呼び出しを制御および制限します。エクスポネンシャルバックオフを使用して、徐々に長い間隔で再試行します。再試行間隔をランダム化するジッターを導入し、再試行の最大数を制限します。

一部の AWS SDK は、デフォルトで再試行とエクスポネンシャルバックオフを実装しています。ワークロードに該当する場合は、これらの組み込み AWS 実装を使用してください。べき等性を持たせて再試行することでクライアントの可用性が向上するサービスを呼び出すときも、同様のロジックをワークロードに実装します。タイムアウトの時間と、再試行をいつ停止するのかをユースケースに基づいて決めます。こうした再試行のユースケースに対応するテストシナリオを構築し、実行してください。

実装手順

- アプリケーションスタック内の最適なレイヤーを決定して、アプリケーションが依存するサービスの再試行を実装してください。
- 選択した言語に対してエクスポネンシャルバックオフとジッターを伴う実証済みの再試行戦略を実装している既存の SDK に注意し、独自の再試行実装を作成するよりもこれらを優先してください。
- 再試行を行う前に、サービスがべき等性を持っていることを確認します。再試行を実装したら、必ずテストを行い、本番環境で定期的に実行するようにしてください。
- AWS サービス API を呼び出すときは、AWS SDK と AWS CLI を使用し、再試行の設定オプションを把握します。デフォルトがユースケースに適しているかどうかを判断し、テストし、必要に応じて調整します。

リソース

関連するベストプラクティス:

- REL04-BP04 変更操作をべき等にする
- REL05-BP02 リクエストのスロットル
- REL05-BP04 フェイルファストとキューの制限
- REL05-BP05 クライアントタイムアウトを設定する
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する

関連ドキュメント:

- AWS でのエラーの再試行とエクスポネンシャルバックオフ
- Amazon Builders' Library: ジッターを伴うタイムアウト、再試行、およびバックオフ
- Exponential Backoff and Jitter
- Making retries safe with idempotent APIs

関連する例:

- Spring Retry
- Resilience4j Retry

関連動画:

- Retry, backoff, and jitter: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)

関連ツール:

- AWS SDKs and Tools: Retry behavior
- AWS Command Line Interface: AWS CLI 再試行

REL05-BP04 フェイルファストとキューの制限

サービスがリクエストに正常に応答できない場合は、すぐに失敗します。これにより、リクエストに関連付けられたリソースが解放され、リソースが不足した場合にサービスを復旧できます。フェイルファストは確立されたソフトウェア設計パターンであり、これを活用して信頼性の高いワークロードをクラウド上に構築できます。キューイングは、負荷をスムーズにし、非同期処理が許容できる場合にクライアントがリソースを解放できるようにする、確立されたエンタープライズ統合パターンでもあります。サービスが通常の状態では正常に応答できるが、リクエストのレートが高すぎると失敗する場合は、キューを使用してリクエストをバッファします。ただし、長いキューのバックログの蓄積は許可しないでください。クライアントが既に処理を停止している古いリクエストを処理する原因となる可能性があるためです。

期待される成果: システムにリソースの競合、タイムアウト、例外、またはグレー障害が発生してサービスレベル目標を達成できない場合、フェイルファスト戦略を使用するとシステムをより迅速に回復できます。トラフィックの急増を吸収する必要があり、非同期処理に対応できるシステムでは、バックエンドサービスへのリクエストをバッファリングするキューを使用して、クライアントがリクエストを迅速にリリースできるようにすることで信頼性を向上できます。リクエストをキューにバッファリングする際には、克服できないバックログを回避するためにキュー管理戦略が実装されます。

一般的なアンチパターン:

- メッセージキューを実装するが、システムに障害が発生したことを検出するデッドレターキュー (DLQ) やアラームを DLQ ボリュームに設定しない。
- キュー内のメッセージの経過時間を測定するのではなく、キューのコンシューマーが遅れたり、エラーが発生して再試行が発生したりするタイミングを把握するためのレイテンシーの測定です。
- 業務上の必要がなくなった場合に、これらのメッセージを処理する価値がない場合に、未処理のメッセージをキューから消去しない。

- 先入れ先出し (FIFO) キューを後入れ先出し (LIFO) キューに設定すると、クライアントのニーズにより適切に対応できます。例えば、厳密な順序付けが不要で、バックログ処理により新規リクエストや時間的制約のあるリクエストがすべて遅延し、その結果、すべてのクライアントでサービスレベル違反が発生するような場合です。
- 仕事の受け入れを管理してリクエストを内部キューに入れる API を公開する代わりに、内部キューをクライアントに公開します。
- 1 つのキューに多数の作業リクエストタイプをまとめると、リソース需要がリクエストタイプ全体に分散され、バックログの状態が悪化する可能性があります。
- 異なるモニタリング、タイムアウト、リソース割り当てが必要な場合でも、複雑なリクエストと単純なリクエストを同じキューで処理します。
- エラーを適切に処理できる上位レベルのコンポーネントに例外をバブリングするフェイルファストメカニズムをソフトウェアで実装するために、入力を検証したり、アサーションを使用したりしない。
- リクエストルーティングから障害のあるリソースを削除しない。特に、クラッシュや再起動、断続的な依存関係の障害、容量の低下、ネットワークのパケットロスなどにより、障害がグレーで成功と失敗の両方を示している場合。

このベストプラクティスを活用するメリット: フェイルファストなシステムはデバッグや修正が容易で、多くの場合、リリースが本稼働環境にパブリッシュされる前に、コーディングや構成上の問題を明らかにすることができます。効果的なキューイング戦略を組み込んだシステムは、トラフィックの急増や断続的なシステム障害状態に対する回復力と信頼性が向上します。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

フェイルファスト戦略は、ソフトウェアソリューションにコード化することも、インフラストラクチャに構成することもできます。キューは、高速に障害が発生するだけでなく、システムコンポーネントを切り離してスムーズに負荷をかけるための単純でありながら強力なアーキテクチャ手法です。Amazon CloudWatch には、障害をモニタリングし、警告する機能があります。システムに障害が発生していることが判明したら、障害が発生したリソースからフェイルアウェイするなどの緩和策を講じることができます。システムが Amazon SQS やその他キューテクノロジーを使用してキューを実装し、負荷を軽減している場合、そのシステムは、キューのバックログやメッセージ使用の失敗の、管理方法を検討しておく必要があります。

実装手順

- プログラムによるアサーションまたは特定のメトリクスをソフトウェアに実装し、それらを使用してシステムの問題を明示的に警告します。Amazon CloudWatch を使用すると、アプリケーションのログパターンや SDK の計測に基づいてメトリクスとアラームを作成することができます。
- CloudWatch メトリクスとアラームを使用して、リソースに障害が発生して処理に遅延が発生したり、リクエストの処理が繰り返し失敗したりしないようにします。
- Amazon SQS を使用してリクエストを受け入れ、リクエストを内部キューに追加し、メッセージ生成クライアントに成功メッセージで応答する API を設計することで非同期処理を使用します。これにより、バックエンドキューのコンシューマーがリクエストを処理している間、クライアントはリソースを解放して他の作業に進むことができます。
- 現在とメッセージのタイムスタンプを比較することで、メッセージをキューから取り出すたびに CloudWatch メトリクスを生成し、キューの処理遅延を測定およびモニタリングします。
- 障害によってメッセージ処理が正常に行われなかった、またはサービスレベル契約の範囲内で処理できない量のトラフィックが急増した場合は、古いトラフィックや過剰なトラフィックをスピルオーバーキューに振り分けます。これにより、キャパシティに空きがあれば、新しい作業や古い作業を優先的に処理できます。この手法は LIFO 処理の近似値であり、すべての新規作業で通常のシステム処理が可能になります。
- 処理できないメッセージをバックログから後で調査して解決できる場所に移動するには、デッドレターキューまたはリドライブキューを使用します。
- 再試行するか、許容範囲内であれば、メッセージのタイムスタンプと現在を比較して、要求元のクライアントに関係のないメッセージは破棄して、古いメッセージを削除してください。

リソース

関連するベストプラクティス:

- REL04-BP02 疎結合の依存関係を実装する
- REL05-BP02 リクエストのスロットル
- REL05-BP03 再試行呼び出しを制御および制限する
- REL06-BP02 メトリクスを定義および計算する (集計)
- REL06-BP07 システムを通じたリクエストのエンドツーエンドのトレースをモニタリングする

関連ドキュメント:

- 乗り越えられないキューバックログの回避
- Fail Fast
- Amazon SQS キュー内のメッセージのバックログの増加を防ぐにはどうすればよいですか?
- Elastic Load Balancing: Zonal Shift
- Amazon Application Recovery Controller: トラフィックフェイルオーバーのルーティングコントロール

関連する例:

- Enterprise Integration Patterns: Dead Letter Channel

関連動画:

- AWS re:Invent 2022 - Operating highly available Multi-AZ applications

関連ツール:

- Amazon Simple Queue Service
- Amazon MQ
- AWS IoT Core
- Amazon CloudWatch

REL05-BP05 クライアントタイムアウトを設定する

接続とリクエストにタイムアウトを適切に設定し、体系的に検証します。また、デフォルト値には依存しないでください。これらはワークロードの詳細を認識していないためです。

期待される成果: クライアントのタイムアウトには、完了までに異常に時間がかかるリクエストを待つことに関連するクライアント、サーバー、およびワークロードにかかるコストを考慮する必要があります。タイムアウトの正確な原因を知ることはできないため、クライアントはサービスの知識を活用して、考えられる原因と適切なタイムアウトを予測する必要があります。

クライアント接続は、設定された値に基づいてタイムアウトします。タイムアウトが発生すると、クライアントはバックオフして再試行するかサーキットブレーカーを開くか、いずれかを決定します。これらのパターンは、根本的なエラー状態を悪化させる可能性のあるリクエストの発行を回避します。

一般的なアンチパターン:

- システムタイムアウトまたはデフォルトタイムアウトを認識していない。
- 通常のリクエスト完了タイミングを認識していない。
- リクエストが完了するまでに異常に時間がかかる原因や、これらの完了を待つことによってクライアント、サービス、またはワークロードのパフォーマンスが低下する原因を認識していない。
- ネットワークに障害が発生して、タイムアウトに達したときだけリクエストが失敗する確率や、より短いタイムアウトを採用しないことでクライアントとワークロードのパフォーマンスにコストがかかることを認識していない。
- 接続とリクエストの両方のタイムアウトシナリオはテストされていません。
- タイムアウトの設定が高すぎると、待機時間が長くなり、リソースの使用率が高くなる可能性があります。
- タイムアウトの設定が低すぎると、人為的な障害が発生します。
- サーキットブレーカーや再試行などのリモート呼び出しのタイムアウトエラーを処理するパターンを見落としています。
- サービス呼び出しエラー率、遅延に関するサービスレベル目標、および遅延異常値のモニタリングは考慮していません。これらのメトリクスから、タイムアウトが積極的または許容範囲が広いかを判断できます。

このベストプラクティスを活用するメリット: リモート呼び出しのタイムアウトは、リモート呼び出しの応答が異常に遅い場合やタイムアウトエラーがサービスクライアントによって適切に処理される場合にリソースを節約できるように、タイムアウトを適切に処理するように設定され、システムがタイムアウトを適切に処理するように設計されています。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

サービス依存関係呼び出しに接続タイムアウトとリクエストタイムアウトの両方を設定します。またこの設定は、通常プロセス全体のすべての呼び出しにも行います。多くのフレームワークにはタイムアウト機能が組み込まれていますが、デフォルト値が無限であるか、サービス目標の許容範囲を超えているものもあるので注意してください。値が高すぎると、クライアントがタイムアウトの発生を待機している間もリソースが消費され続けるため、タイムアウトの有用性が低下します。値が小さすぎると、再試行されるリクエストが多くなりすぎるため、バックエンドのトラフィックが増加し、レイテンシーが高くなってしまいます。場合によっては、すべてのリクエストが再試行されることになるため、完全な機能停止につながる恐れもあります。

タイムアウト戦略を決定する際には、次の点を考慮してください。

- リクエストの内容、ターゲットサービスの障害、またはネットワークパーティションの障害により、リクエストの処理に通常よりも時間がかかる場合があります。
- 異常に高価なコンテンツを含むリクエストは、サーバーとクライアントのリソースを不必要に消費する可能性があります。この場合、これらのリクエストをタイムアウトさせて再試行しないことで、リソースを節約できます。また、サービスは、スロットルやサーバー側のタイムアウトにより、異常にコストが大きいコンテンツから身を守る必要があります。
- サービスの障害により異常に時間がかかるリクエストは、タイムアウトして再試行できます。リクエストと再試行のサービスコストを考慮する必要がありますが、原因が局所的な障害である場合は、再試行してもコストはかからず、クライアントリソースの消費量を削減できます。障害の性質によっては、タイムアウトによってサーバーリソースが解放されることもあります。
- リクエストまたはレスポンスがネットワークから配信されなかったために完了までに時間がかかるリクエストは、タイムアウトして再試行できます。リクエストまたはレスポンスが配信されなかったため、タイムアウトの長さに関係なく失敗に終わったことになります。この場合、タイムアウトしてもサーバーリソースは解放されませんが、クライアントリソースが解放され、ワークロードのパフォーマンスが向上します。

再試行やサーキットブレーカーなどの確立された設計パターンを活用して、タイムアウトをスムーズに処理し、フェイルファストアプローチをサポートします。AWSSDK と AWS CLI を使用すると、接続タイムアウトとリクエストタイムアウトの両方を設定でき、エクスポネンシャルバックオフとジッターによる再試行も行えます。AWS Lambda 関数はタイムアウトの設定をサポートしており、AWS Step Functions を併用すれば、ローコードのサーキットブレーカーを構築して、AWS サービスおよび SDK との事前構築済みの統合を活用できます。AWS App MeshEnvoy はタイムアウトとサーキットブレーカーの機能を備えています。

実装手順

- リモートサービス呼び出しのタイムアウトを設定し、組み込みの言語タイムアウト機能またはオープンソースのタイムアウトライブラリを活用してください。
- ワークロードが AWS SDK を使用して呼び出しを行う場合は、ドキュメントで言語固有のタイムアウト設定を確認してください。
  - Python
  - PHP
  - .NET
  - Ruby

  - Java
  - Go
  - Node.js
  - C++
- ワークロードで AWS SDK または AWS CLI コマンドを使用するときは、`connectTimeoutInMillis` と `tlsNegotiationTimeoutInMillis` の AWS 設定デフォルト を設定し、デフォルトのタイムアウト値を設定します。
- コマンドラインオプション の `cli-connect-timeout` と `cli-read-timeout` を適用して、AWS のサービスの 1 回限りの AWS CLI コマンドを制御します。
- リモートサービス呼び出しのタイムアウトをモニタリングし、エラーが続く場合はアラームを設定して、エラーシナリオにプロアクティブに対処できるようにします。
- コールエラー率、レイテンシーに関するサービスレベル目標、レイテンシーの外れ値に関する CloudWatch メトリクスと CloudWatch 異常検出を実装すると、過度にアグレッシブなタイムアウトや許容範囲のタイムアウトの管理に関するインサイトが得られます。
- Lambda 関数でタイムアウトを設定します。
- API Gateway クライアントは、タイムアウトを処理するときに独自に再試行を行う必要があります。API Gateway は、ダウンストリームの統合に対しては 50 ミリ秒から 29 秒までの統合タイムアウトをサポートしており、統合リクエストがタイムアウトしたときは再試行を行いません。
- タイムアウト時のリモート呼び出しを回避するには、サーキットブレーカーのパターンを実装します。呼び出しが失敗しないように回線を開き、呼び出しが正常に応答したら回線を閉じます。
- コンテナベースのワークロードについては、「App Mesh Envoy」の機能を確認して、組み込みのタイムアウトとサーキットブレーカーを活用します。
- AWS Step Functions を使用して、リモートサービスを呼び出すための (特に、ワークロードを簡素化する目的で AWS ネイティブの SDK と、サポートされている Step Functions 統合とを呼び出すための)、ローコードのサーキットブレーカーを作成します。

リソース

関連するベストプラクティス:

- REL05-BP03 再試行呼び出しを制御および制限する
- REL05-BP04 フェイルファストとキューの制限
- REL06-BP07 システムを通じたリクエストのエンドツーエンドのトレースをモニタリングする

関連ドキュメント:

- AWS SDK: 再試行とタイムアウト
- Amazon Builders' Library: ジッターを伴うタイムアウト、再試行、およびバックオフ
- Amazon API Gateway のクォータと重要な注意点
- AWS Command Line Interface: コマンドラインオプション
- AWS SDK for Java 2.x: API タイムアウトの設定
- AWSBotocore using the config object and Config Reference
- AWS SDK for .NET: Retries and Timeouts
- AWS Lambda: AWS Lambda 関数の設定

関連する例:

- Using the circuit breaker pattern with AWS Step Functions and Amazon DynamoDB
- Martin Fowler: CircuitBreaker

関連ツール:

- AWS SDK
- AWS Lambda
- Amazon Simple Queue Service
- AWS Step Functions
- AWS Command Line Interface

REL05-BP06 可能な限りシステムをステートレスにする

状態を必要としないシステム、または状態をオフロードするシステム (異なるクライアントリクエスト間にディスクやメモリ内のローカルに保存されたデータへの依存がない) にしてください。これにより、可用性に影響を与えることなく、サーバーをいつでも置き換えることができます。

ユーザーまたはサービスがアプリケーションと対話するとき、セッションを形成する一連のやりとりを頻繁に実行します。セッションは、ユーザーがアプリケーションを使用している間、リクエスト間で持続するユーザー固有のデータです。ステートレスアプリケーションは、以前のやりとりの知識を必要とせず、セッション情報を保存しません。

ステートレスな設計にすれば、あとは AWS Lambda や AWS Fargate などのサーバーレスコンピューティングサービスを利用できます。

サーバーの置き換えに加えて、ステートレスアプリケーションのもう 1 つの利点は、利用可能なコンピューティングリソース (EC2 インスタンスや AWS Lambda 関数など) がどのようなリクエストにも対応できるため、水平方向にスケールできることです。

このベストプラクティスを活用するメリット: ステートレスに設計されたシステムは水平スケーリングへの適応性が高いため、トラフィックや需要の変化に応じてキャパシティを増やしたり減らしたりすることが可能です。また、本質的に耐障害性に優れており、アプリケーション開発に柔軟性と俊敏性をもたらします。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

アプリケーションをステートレスにします。ステートレスアプリケーションは、水平スケーリングが可能であり、個別ノードの障害に耐性があります。アーキテクチャ内の状態を維持するアプリケーションのコンポーネントを分析して理解します。これにより、ステートレス設計への移行で考えられる影響を評価できます。ステートレスアーキテクチャはユーザーデータを切り離し、セッションデータをオフロードします。各コンポーネントを個別にスケールし、変化するワークロードの需要に対応することでリソースの使用率を最適化できる柔軟性が得られます。

実装手順

- アプリケーション内のステートフルなコンポーネントを特定して理解します。
- ユーザーデータをコアアプリケーションロジックから分離して管理することで、データを切り離します。
  - Amazon Cognito では、アイデンティティプール、ユーザープール、Amazon Cognito Sync などの機能を使用することでユーザーデータをアプリケーションコードから切り離すことができます。
  - AWS Secrets Manager を使用すると、シークレットを一元化された安全な場所に保管することでユーザーデータを切り離すことができます。つまり、アプリケーションコードにシークレットを保存する必要がないため、安全性が高まります。
  - イメージやドキュメントなどの構造化されていない大規模なデータを保存するときは Amazon S3 の使用を検討します。アプリケーションは必要に応じてこのデータを取得できるため、メモリに保存する必要はありません。

    - ユーザープロファイルなどの情報を保存するときは Amazon DynamoDB を使用します。アプリケーションでは、ほぼリアルタイムでこのデータをクエリできます。
- セッションデータをデータベース、キャッシュ、または外部ファイルにオフロードします。
    - セッションデータのオフロードに使用できる AWS のサービスには、Amazon ElastiCache、Amazon DynamoDB、Amazon Elastic File System (Amazon EFS)、Amazon MemoryDB などがあります。
- 選択したストレージソリューションでは、どの状態とユーザーデータを持続しておく必要があるのかを特定した後、ステートレスアーキテクチャを設計します。

リソース

関連するベストプラクティス:

- REL11-BP03 すべてのレイヤーの修復を自動化する

関連ドキュメント:

- Amazon Builders' Library: 分散システムでのフォールバックの回避
- Amazon Builders' Library: 乗り越えられないキューバックログの回避
- Amazon Builders' Library: キャッシングの課題と戦略
- AWS のステートレスなウェブ層

REL05-BP07 緊急レバーを実装する

緊急レバーは、ワークロードの可用性に対する影響を軽減できる迅速なプロセスです。

緊急レバーは、既知のテスト済みのメカニズムを使用して、コンポーネントや依存関係の動作を無効にしたり、スロットリングしたり、変更したりするためのものです。その効果として、想定外の需要増によるリソースの枯渇が原因となるワークロードの障害を軽減し、ワークロード内の重要ではないコンポーネントの障害の波及を抑制できます。

期待される成果: 緊急レバーを実装することで、ワークロードに欠かせないコンポーネントの可用性を維持するための、問題がないことが確認されているプロセスを確立できます。緊急レバーが作動している間、ワークロードは意図的に性能を落とし (グレースフルデグラデーション)、ビジネスに不可欠な機能を引き続き実行します。グレースフルデグラデーションの詳細は、「REL05-BP01 該当す

るハードな依存関係をソフトな依存関係に変換するため、グレースフルデグラデーションを実装する」を参照してください。

一般的なアンチパターン:

- 重要ではない依存関係に障害が発生した場合に、主要ワークロードの可用性に影響が波及する。
- 重要ではないコンポーネントに障害が起きている間に、重要なコンポーネントの動作をテストまたは検証しない。
- 緊急レバーの作動または作動解除に関する決定的な基準が明確に定義されていない。

このベストプラクティスを活用するメリット: 緊急レバーを実装すれば、予期せぬ需要の急増や、重要度の低い依存関係における障害などに対処するためのプロセスを確立してリゾルバーに提供することで、ワークロードに不可欠なコンポーネントの可用性を高めることができます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

- ワークロードの重要なコンポーネントを特定します。
- 重要ではないコンポーネントに障害が起きても耐えられるように、ワークロードの重要なコンポーネントを設計し、構築します。
- 重要ではないコンポーネントで障害が発生している最中に、重要なコンポーネントの動作を検証するためのテストを実施します。
- 緊急レバーの手続き開始の基準となる適切な指標やトリガーを定義し、監視します。
- 緊急レバーを構成する手順 (手動または自動) を定義します。

実装手順

- ワークロード内のビジネスクリティカルなコンポーネントを特定します。
  - ワークロードの技術的なコンポーネントをそれぞれ適切なビジネス機能にマッピングし、重要または非重要にランク分けします。Amazon の重要な機能および非重要な機能の例については、「Any Day Can Be Prime Day: How Amazon.com Search Uses Chaos Engineering to Handle Over 84K Requests Per Second」を参照してください。
  - これは技術上の決定でもビジネス上の決定でもあり、組織やワークロードによって異なります。
- 重要ではないコンポーネントに障害が起きても耐えられるように、ワークロードの重要なコンポーネントを設計し、構築します。

    - 依存関係の分析では、想定される障害モードをすべて検討し、緊急レバーのメカニズムを通じて、ダウンストリームのコンポーネントも重要な機能を利用できるか検証します。
- 緊急レバーが作動している間に、重要なコンポーネントの動作を検証するためのテストを実施してください。
    - バイモーダル動作は防止してください。詳細については、「REL11-BP05 静的安定性を使用してバイモーダル動作を防止する」を参照してください。
- 緊急レバーの手続き開始の基準となる指標を定義して監視し、警戒します。
    - ワークロードに応じて、監視対象として適切な指標を判断してください。指標の例としては、レイテンシーや、依存関係へのリクエストの失敗回数などが該当します。
- 緊急レバーを構成する手順 (手動または自動) を定義します。
    - これには、負荷制限、リクエストのスロットリング、グレースフルデグラデーションの実装などのメカニズムが含まれます。

リソース

関連するベストプラクティス:

- REL05-BP01 該当するハードな依存関係をソフトな依存関係に変換するため、グレースフルデグラデーションを実装する
- REL05-BP02 リクエストのスロットル
- REL11-BP05 静的安定性を使用してバイモーダル動作を防止する

関連ドキュメント:

- 安全なハンズオフデプロイメントの自動化
- プライムデーがいつ来ても大丈夫: Amazon.com の検索機能がカオスエンジニアリングで 1 秒に 84,000 件以上のリクエストを処理する方法

関連動画:

- AWS re:Invent 2020: Reliability, consistency, and confidence through immutability

変更管理

ワークロードを信頼できる形で運用するには、ワークロードやその環境に対する変化を予測してそれに対応することが不可欠です。変更には、需要の急増などのワークロードに負荷がかかる変更や、機能のデプロイやセキュリティパッチの適用といった内部からの変更があります。

次のセクションでは、変更管理のベストプラクティスについて説明します。

トピック

ワークロードリソースをモニタリングする

ログやメトリクスは、ワークロードの状態に関するインサイトを得るための強力なツールです。ログやメトリクスをモニタリングし、しきい値を超えたときや、重要なイベントが発生したときに通知を送信するようにワークロードを設定することができます。モニタリングにより、ワークロードは、低パフォーマンスのしきい値を超えたときや障害が発生したときにそれを認識できるため、それに応じて自動的に復旧できます。

モニタリングは、可用性の要件を満たしていることを確認する上で必要不可欠です。障害を効果的に検出するにはモニタリングが欠かせません。最悪の障害モードは「サイレント」障害です。この場合、機能は正常に機能しなくなっていますが、間接的なものを除き、検出する方法がありません。それにいち早く気付くのは、お客様ではなくてその顧客です。問題発生時にアラートを送信するのが、モニタリングの主な目的です。アラートは可能な限りシステムから分離する必要があります。サービスの中断によりアラートの機能が無効化されると、中断がより長時間になります。

AWS では、アプリケーションを複数のレベルで測定しています。これにより、各リクエスト、すべての依存関係、プロセス内の主要なオペレーションについて、レイテンシー、エラー率、可用性の記録を行っています。また、成功した操作のメトリクスも記録しています。これにより、切迫した問題が発生する前にそれを発見することができます。考慮するのは、平均レイテンシーだけではありません。99.9 パーセンタイルや 99.99 パーセンタイルなど、レイテンシーの外れ値により焦点を当てています。これは、1,000 または 10,000 のうちのたった 1 つのリクエストが遅かった場合でも、エクスペリエンスの満足度が低下するためです。また、平均値は許容できるかもしれませんが、リクエスト 100 件のうちの 1 件に極端なレイテンシーが発生すれば、トラフィックが増加したときに問題化します。

AWS のモニタリングは、次の 4 つの個別のフェーズで構成されています。

1. 生成 – ワークロードのすべてのコンポーネントをモニタリングする
2. 集計 – メトリクスを定義して計算する
3. リアルタイム処理とアラーム – 通知を送信し、応答を自動化する
4. ストレージと分析

ベストプラクティス

- REL06-BP01 ワークロードのすべてのコンポーネントをモニタリングする (生成)
- REL06-BP02 メトリクスを定義および計算する (集計)
- REL06-BP03 通知を送信する (リアルタイム処理とアラーム)
- REL06-BP04 レスポンスを自動化する (リアルタイム処理とアラーム)
- REL06-BP05 ログの分析
- REL06-BP06 モニタリングの範囲とメトリクスを定期的に確認する
- REL06-BP07 システムを通じたリクエストのエンドツーエンドのトレースをモニタリングする

REL06-BP01 ワークロードのすべてのコンポーネントをモニタリングする (生成)

ワークロードのコンポーネントは、Amazon CloudWatch またはサードパーティーのツールを使ってモニタリングします。AWS サービスを AWS Health ダッシュボードでモニタリングします。

フロントエンド、ビジネスロジック、ストレージ層など、ワークロードのすべてのコンポーネントをモニタリングする必要があります。主要なメトリクスと、必要に応じてそれをログから抽出する方法を定義し、対応するアラームイベントを起動させるためのしきい値を設定します。メトリクスがワークロードの重要業績評価指標 (KPI) に関連していることを確認し、メトリクスとログを使用して、サービス低下の早期警告サインを識別します。例えば、1 分間に正常に処理されたオーダー数など、ビジネス成果に関するメトリクスは、CPU 使用率などの技術的メトリクスより早く、ワークロード問題を示すことができます。AWS Health ダッシュボードは、AWS リソースの基盤となる AWS のサービスのパフォーマンスと可用性をパーソナライズして表示するために使用します。

クラウドでのモニタリングは新しい機会をもたらします。ほとんどのクラウドプロバイダーは、カスタマイズ可能なフックを開発して、ワークロードの複数のレイヤーをモニタリングする際に役立つインサイトを提供しています。Amazon CloudWatch などの AWS サービスは、統計的な機械学習アル

ゴリズムを応用して、システムとアプリケーションのメトリクスを継続的に分析し、正常なベースラインを決定し、最小限のユーザー介入で異常を表面化します。異常検出アルゴリズムは、メトリクスの季節的な変化と傾向の変化を考慮します。

AWS では、豊富なモニタリングおよびログ情報を公開しており、これらを使用して、ワークロード固有のメトリクスと需要変化プロセスを定義し、機械学習の知識に関わらず、機械学習技法を適応させることができます。

さらに、すべての外部エンドポイントをモニタリングし、それらがベースとなる実装から独立していることを確認します。このアクティブモニタリングは、合成トランザクション (「ユーザー canary」ともいう。「カナリアデプロイ」と混同しないこと) で行うことができます。これは、ワークロードのクライアントが実行するアクションに相当する多くの共通タスクを定期的に実行するものです。これらのタスクは、短期間に保ち、テスト中にワークロードに負荷をかけすぎないようにしてください。Amazon CloudWatch Synthetics を使用すると、Synthetic canaries を作成してエンドポイントと API をモニタリングすることができます。合成 canary クライアントノードと AWS X-Ray コンソールを組み合わせて、選択した期間中にエラー、障害、スロットリング率で問題が発生している合成 canary を特定することもできます。

期待される成果:

ワークロードのすべてのコンポーネントから重要なメトリクスを収集して使用し、ワークロードの信頼性と最適なユーザーエクスペリエンスを確保します。ワークロードがビジネス成果を達成していないことを検出した場合は、障害を迅速に宣言して、インシデントから復旧できます。

一般的なアンチパターン:

- ワークロードへの外部インターフェイスのみをモニタリングする。
- ワークロード固有のメトリクスを生成せず、ワークロードが使用している AWS から提供されるメトリクスにのみ依存する。
- ワークロードの技術的メトリクスを使用するだけで、ワークロードが貢献する非技術的な KPI に関するメトリクスをモニタリングしない。
- 本番トラフィックとシンプルなヘルスチェックに依存して、ワークロード状態をモニタリングし、評価する。

このベストプラクティスを活用するメリット: ワークロードのすべての階層でモニタリングすることで、ワークロードを構成するコンポーネントの問題をより迅速に予測し、解決できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

1. 可能な限りログを有効にします。ワークロードのすべてのコンポーネントからモニタリングデータを取得する必要があります。S3 Access Logs など、追加のロギングをオンにして、ワークロードがワークロード固有のデータをログに記録できるようにします。Amazon ECS、Amazon EKS、Amazon EC2、Elastic Load Balancing、AWS Auto Scaling、Amazon EMR などのサービスから、CPU、ネットワーク I/O、ディスク I/O の平均、に関するメトリクスを収集します。CloudWatch にメトリクスをパブリッシュする AWS のサービスの一覧については、「CloudWatch メトリクスを発行する AWS のサービス」を参照してください。
2. デフォルトのメトリクスをすべてレビューし、データ収集にギャップがないか確認します。すべてのサービスはデフォルトのメトリクスを生成します。デフォルトのメトリクスを収集することで、ワークロードのコンポーネント間の依存関係と、コンポーネントの信頼性とパフォーマンスがワークロードに及ぼす影響をより深く理解できます。メトリクスは、AWS CLI または API を使用して作成し、CloudWatch にパブリッシュすることもできます。
3. すべてのメトリクスを評価して、ワークロード内の各 AWS サービスに対してどのメトリクスでアラートを発するかを決定します。ワークロードの信頼性に大きな影響を持つメトリクスのサブセットを選択することもできます。重要なメトリクスとしきい値に焦点を当てることで、アラートの数を絞り込み、偽陽性を最小限に抑えることができます。
4. アラートを定義し、アラートが起動した後のワークロードの復旧プロセスを定義します。アラートを定義することで、通知とエスカレーションを迅速に行い、インシデントからの復旧に必要なステップに従い、所定の目標復旧時間 (RTO) を満たすことができます。Amazon CloudWatch Alarms を使用すると、定義されたしきい値に基づいて自動ワークフローを呼び出し、回復手順を開始することができます。
5. 合成トランザクションを使用して、ワークロードの状態に関する関連データを収集することを検討しましょう。合成モニタリングは、顧客と同じルートに従って同じアクションを実行するため、ワークロードに顧客のトラフィックがない場合でも、継続的にカスタマーエクスペリエンスを検証することが可能になります。合成トランザクションを使用すると、顧客が問題を検出する前に問題を検出できます。

リソース

関連するベストプラクティス:

- REL11-BP03 すべてのレイヤーの修復を自動化する

関連ドキュメント:

- AWS Health Dashboard の使用開始 – アカウントヘルスの確認
- CloudWatch メトリクスを発行する AWS のサービス
- Network Load Balancer のアクセスログ
- Application Load Balancer のアクセスログ
- AWS Lambda での Amazon CloudWatch Logs の使用
- サーバーアクセスログによるリクエストのログ記録
- Classic Load Balancer のアクセスログの有効化
- Amazon S3 へのログデータのエクスポート
- CloudWatch エージェントをインストールする
- カスタムメトリクスを発行する
- Amazon CloudWatch ダッシュボードの使用
- Amazon CloudWatch メトリクスを使用する
- 合成モニタリング (canary)
- Amazon CloudWatch Logs とは

ユーザーガイド:

- 証跡の作成
- CloudWatch エージェントを使用してメトリクス、ログ、トレースを収集する
- Amazon ECS のモニタリングツール
- VPC フローログ
- Amazon DevOps Guru とは
- What is AWS X-Ray?

関連ブログ:

- Amazon CloudWatch Synthetics と AWS X-Ray でのデバッグ

関連する例:

- Amazon Builders' Library: 運用の可視性を高めるために分散システムを装備する
- つのオブザーバビリティワークショップ

REL06-BP02 メトリクスを定義および計算する (集計)

ワークロードコンポーネントからメトリクスとログを収集し、そこから関連する集計メトリクスを計算します。これらのメトリクスは、ワークロードの広範で深いオブザーバビリティを提供し、耐障害性の態勢を大幅に改善できます。

オブザーバビリティは、ワークロードコンポーネントからメトリクスを収集し、それらを表示してアラートを発するだけではありません。これは、ワークロードの動作を全体的に理解することです。この動作情報は、ワークロード内のすべてのコンポーネントから取得されます。これには、ワークロードが依存するクラウドサービス、適切に作成されたログ、メトリクスが含まれます。このデータにより、ワークロードの全体的な動作を監督し、すべてのコンポーネントとすべての作業単位のやり取りを詳細に把握できます。

期待される成果:

- ワークロードコンポーネントと AWS サービスの依存関係からログを収集し、簡単にアクセスして処理できる一元的な場所に公開します。
- ログには、忠実度が高く正確なタイムスタンプが含まれています。
- ログには、トレース識別子、ユーザーまたはアカウント識別子、リモート IP アドレスなど、処理コンテキストに関する関連情報が含まれています。
- おおまかな視点からワークロードの動作を表す集計メトリクスをログから作成します。
- 集約ログをクエリして、ワークロードに関する深く関連するインサイトを取得し、実際の問題と潜在的な問題を特定できます。

一般的なアンチパターン:

- ワークロードが実行されるコンピューティングインスタンスや、ワークロードが使用するクラウドサービスから、関連するログやメトリクスを収集することはない。
- ビジネスキーパフォーマンスインジケーター (KPI) に関連するログとメトリクスのコレクションは無視する。
- ワークロード関連のテレメトリを集約や相関関係なしに個別に分析する。
- メトリクスとログの有効期限が早すぎ、トレンド分析や問題の定期的な識別が妨げられる。

これらのベストプラクティスを活用するメリット: より多くの異常を検出し、ワークロードのさまざまなコンポーネント間でイベントとメトリクスを関連付けることができます。メトリクスだけでは利

用できないことが多いログに含まれる情報に基づいて、ワークロードコンポーネントからインサイトを作成できます。大規模にログをクエリすることにより、障害の原因をより迅速に特定できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

ワークロードとそのコンポーネントに関連するテレメトリデータのソースを特定します。このデータは、オペレーティングシステム (OS) や Java などのアプリケーションランタイムなどのメトリクスを発行するコンポーネントだけでなく、アプリケーションやクラウドサービスのログからも取得されます。例えば、ウェブサーバーは通常、タイムスタンプ、処理レイテンシー、ユーザー ID、リモート IP アドレス、パス、クエリ文字列などの詳細情報を使用して各リクエストを記録します。これらのログの詳細レベルは、詳細なクエリを実行し、他の方法では利用できないメトリクスを生成するのに役立ちます。

適切なツールとプロセスを使用してメトリクスとログを収集します。Amazon EC2 インスタンスで実行されているアプリケーションによって生成されたログは、Amazon CloudWatch Agent などのエージェントによって収集され、Amazon CloudWatch Logs などの中央ストレージサービスに発行されます。AWS Lambda や Amazon Elastic Container Service などの AWS マネージドコンピューティングサービスは、自動的に CloudWatch Logs にログを発行します。Amazon CloudFront、Amazon S3、Elastic Load Balancing、Amazon API Gateway などのワークロードで使用される AWS ストレージおよび処理サービスのログ収集を有効にします。

動作パターンをより明確に把握し、相関する問題を関連するコンポーネントのグループに分離するのに役立つディメンションでテレメトリデータを強化します。追加すると、コンポーネントの動作をより詳細なレベルで観察し、相関する障害を検出し、適切な修復手順を実行できるようになります。便利なディメンションの例としては、アベイラビリティーゾーン、EC2 インスタンス ID、コンテナタスクまたはポッド ID などがあります。

メトリクスとログを収集したら、クエリを記述し、それらから集計メトリクスを生成して、正常な動作と異常な動作の両方に関する有用なインサイトを提供できます。例えば、Amazon CloudWatch Logs Insights を使用してアプリケーションログからカスタムメトリクスを導き出し、Amazon CloudWatch Metrics Insights を使用してメトリクスを大規模にクエリし、Amazon CloudWatch Container Insights を使用してコンテナ化されたアプリケーションとマイクロサービスからメトリクスとログを収集、集約、要約でき、AWS Lambda 関数を使用している場合は Amazon CloudWatch Lambda Insights を使用できます。集約エラーレートメトリクスを作成するには、コンポーネントログにエラーレスポンスやメッセージが見つかるたびにカウンターを増分したり、既存のエラーレートメトリクスの集約値を計算したりできます。このデータを使用して、パフォーマンスが最も低いリ

クエストやプロセスなどのテール動作を示すヒストグラムを生成できます。また、CloudWatch Logs の異常検出などのソリューションを使用して、このデータをリアルタイムでスキャンして異常パターンを検出することもできます。これらのインサイトはダッシュボードに配置して、ニーズや好みに応じて整理できます。

ログのクエリは、ワークロードコンポーネントによって特定のリクエストがどのように処理されたかを理解し、ワークロードの回復力に影響を与えるリクエストパターンやその他のコンテキストを明らかにするのに役立ちます。アプリケーションやその他のコンポーネントの動作に関する知識に基づいて、事前にクエリを調査して準備しておくと、必要に応じてクエリをより簡単に実行できるため便利です。例えば、CloudWatch Logs Insights を使用すると、CloudWatch Logs に保存されているログデータをインタラクティブに検索、分析できます。Amazon Athena を使用すると、ペタバイト規模で、多くの AWS のサービスを含む複数のソースからのログをクエリすることもできます。

ログ保持ポリシーを定義するときは、履歴ログの価値を考慮してください。履歴ログは、ワークロードのパフォーマンスにおける長期的な使用状況と動作のパターン、リグレッション、改善を特定するのに役立ちます。完全に削除されたログは後で分析することはできません。ただし、履歴ログの価値は長期間にわたって低下する傾向があります。必要に応じてニーズのバランスを取り、適用される可能性のある法的または契約上の要件に準拠するポリシーを選択してください。

実装手順

1. オブザーバビリティデータの収集、ストレージ、分析、表示メカニズムを選択します。
2. ワークロードの適切なコンポーネント (Amazon EC2 インスタンスやサイドカーコンテナなど) にメトリクスコレクターとログコレクターをインストールして設定します。これらのコレクターが予期せず停止した場合に自動的に再起動するように設定します。一時的な発行の失敗がアプリケーションに影響を与えたり、データが失われたりしないように、コレクターのディスクまたはメモリバッファリングを有効にします。
3. ワークロードの一部として使用する AWS のサービスのログオンを有効にし、必要に応じて選択したストレージサービスにそれらのログを転送します。詳細な手順については、各サービスのユーザーガイドまたはデベロッパーガイドを参照してください。
4. テレメトリデータに基づくワークロードに関連する運用メトリクスを定義します。これらは、ビジネス KPI 関連のメトリクスを含むワークロードコンポーネントから出力される直接メトリクスや、合計、レート、パーセンタイル、ヒストグラムなどの集計計算の結果に基づく場合があります。これらのメトリクスはログアナライザーを使用して計算し、必要に応じてダッシュボードに配置します。
5. 必要に応じて、ワークロードコンポーネント、リクエスト、またはトランザクションの動作を分析するための適切なログクエリを準備します。

6. コンポーネントログのログ保持ポリシーを定義して有効にします。ポリシーで許可されているよりも古いログは定期的に削除します。

リソース

関連するベストプラクティス:

- REL06-BP01 ワークロードのすべてのコンポーネントをモニタリングする (生成)
- REL06-BP03 通知を送信する (リアルタイム処理とアラーム)
- REL06-BP04 レスポンスを自動化する (リアルタイム処理とアラーム)
- REL06-BP05 ログの分析
- REL06-BP06 モニタリングの範囲とメトリクスを定期的に確認する
- REL06-BP07 システムを通じたリクエストのエンドツーエンドのトレースをモニタリングする

関連ドキュメント:

- Amazon CloudWatch の仕組み
- Amazon Managed Prometheus
- Amazon Managed Grafana。
- Analyzing Log Data with CloudWatch Logs Insights
- Amazon CloudWatch Lambda Insights
- Amazon CloudWatch Container Insights
- CloudWatch Metrics Insights を使用して CloudWatch メトリクスをクエリする
- AWS オープンテレメトリー用ディストロ
- Amazon CloudWatch Logs Insights のサンプルクエリ
- Amazon CloudWatch Synthetics と AWS X-Ray でのデバッグ
- ログデータの検索およびフィルタリング
- Amazon S3 に直接ログを送信する
- Amazon Builders' Library: 運用の可視性を高めるために分散システムを装備する

関連するワークショップ:

- つのオブザーバビリティワークショップ

関連ツール:

- AWS Distro for OpenTelemetry (GitHub)

REL06-BP03 通知を送信する (リアルタイム処理とアラーム)

組織は、潜在的な問題を検出すると、その問題に迅速かつ効果的に対応するために、適切な担当者とシステムにリアルタイムの通知とアラートを送信します。

期待される成果: サービスとアプリケーションのメトリクスに基づいて関連するアラームを設定することで、運用にかかわるイベントに迅速に対応できます。アラームのしきい値を超えると、適切な担当者とシステムに通知され、根本的な問題に対処できます。

一般的なアンチパターン:

- アラームのしきい値を過度に高く設定し、重要な通知が送信されなくなる。
- アラームのしきい値を低くしすぎたことにより、過剰な通知のノイズが原因で重要なアラートへの対処が行われなくなる。
- 使用率が変わってもアラームとそのしきい値を更新しない。
- 自動アクションで対処するのが最適なアラームに対して、自動アクションを生成する代わりに担当者に通知を送信することで、余計な通知が送信されてしまう。

このベストプラクティスを活用するメリット: 適切な担当者とシステムにリアルタイムの通知とアラートを送信することで、問題を早期に検出し、運用にかかわるインシデントに迅速に対応できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

アプリケーションの可用性に影響を与え、自動対応のトリガーとなる可能性のある問題を検出しやすくするために、ワークロードにはリアルタイム処理とアラーム機能が備わっている必要があります。組織は、重要なイベントが発生したり、メトリクスがしきい値を超えたりしたときに通知を受け取ることができるよう、定義されたメトリクスを使用してアラートを作成することで、リアルタイムの処理とアラームの発行を実施できます。

Amazon CloudWatch では、静的しきい値、異常検出、およびその他の基準に基づく CloudWatch アラームを使用して、メトリクスアラームと複合アラームを作成できます。CloudWatch を使用して設定できるアラームの種類の詳細については、CloudWatch ドキュメントのアラームに関するセクションを参照してください。

CloudWatch ダッシュボードを使用して、チーム向けに AWS リソースのメトリクスとアラートをカスタマイズ表示できます。CloudWatch コンソールのカスタマイズ可能なホームページでは、複数のリージョンのリソースを 1 つのビューでモニタリングできます。

アラームは、Amazon SNS トピックへの通知の送信、Amazon EC2 アクションまたは Amazon EC2 Auto Scaling アクションの実行、OpsItem または AWS Systems Manager のインシデントの作成など、1 つまたは複数のアクションを実行できます。

Amazon CloudWatch は Amazon SNS を使用して、アラームの状態が変化したときに通知を送信し、パブリッシャー (プロデューサー) からサブスクライバー (コンシューマー) にメッセージを配信します。Amazon SNS 通知の設定の詳細については、「Configuring Amazon SNS」を参照してください。

CloudWatch アラームが作成、更新、削除されたり、状態が変更されたりするたびに、CloudWatch は EventBridge にイベントを送信します。こうしたイベントで EventBridge を使用して、アラームの状態が変わるたびに通知したり、Systems Manager Automation を使用してアカウント内のイベントを自動的にトリガーしたりするなどのアクションを実行するルールを作成できます。

AWS Health で最新情報を入手してください。AWS Health は、AWS クラウド リソースの正常性に関する信頼できるソースです。AWS Health を使用して、確認されたサービスイベントの通知を受け取ることで、影響を軽減するための手順をすばやく実行できます。AWS User Notifications を通じて E メールやチャットチャネルに、目的に合った AWS Health イベント通知を作成し、Amazon EventBridge を通じてモニタリングツールやアラートツールをプログラムで統合します。AWS Organizations を使用する場合、アカウント間で AWS Health イベントを集約します。

EventBridge を使うべき場合と Amazon SNS を使うべき場合

EventBridge と Amazon SNS はどちらもイベント駆動型アプリケーションの開発に使用できます。どちらを選ぶかは、具体的なニーズによって異なります。

Amazon EventBridge は、独自のアプリケーション、SaaS アプリケーション、AWS サービスからのイベントに反応するアプリケーションを構築する場合に推奨されます。EventBridge は、サードパーティーの SaaS パートナーと直接統合する唯一のイベントベースのサービスです。EventBridge は、デベロッパーがアカウントにリソースを作成することなく、200 を超える AWS サービスからイベントを自動的に取り込みます。

EventBridge では、定義済みの JSON ベースの構造がイベントに使用されており、ターゲットに転送するイベントを選択する際にイベント本文全体に適用されるルールを作成できます。EventBridge は現在、AWS Lambda、Amazon SQS、Amazon SNS、Amazon Amazon Kinesis Data Streams、Amazon Data Firehose など、20 を超える AWS サービスをターゲットとしてサポートしています。

Amazon SNS は、高いファンアウトを必要とするアプリケーション (数千または数百万のエンドポイント) に推奨されます。よく見られるパターンは、お客様が Amazon SNS をルールのターゲットとして使用し、必要なイベントをフィルタリングして複数のエンドポイントに分散させるというものです。

メッセージは構造化されておらず、任意の形式にすることができます。Amazon SNS では Lambda、Amazon SQS、HTTP/S エンドポイント、SMS、モバイルプッシュ、メールの 6 種類のターゲットへのメッセージ転送をサポートしています。Amazon SNS の通常のレイテンシーは 30 ミリ秒未満です。AWS のさまざまなサービス (Amazon EC2、Amazon S3、Amazon RDS など 30 以上のサービス) で、Amazon SNS メッセージを送信するようにサービスを設定できます。

実装手順

1. Amazon CloudWatch アラームを使用してアラームを作成します。
    a. メトリクスアラームは、単一の CloudWatch メトリクス、または CloudWatch メトリクスに依存する式をモニタリングします。アラームは、メトリクスまたは式の値としきい値との比較に基づいて、複数の時間間隔にわたって 1 つまたは複数のアクションを開始します。アクションでは、Amazon SNS トピックに通知を送信したり、Amazon EC2 アクションまたは Amazon EC2 Auto Scaling アクションを実行したりできます。また、AWS Systems Manager で OpsItem またはインシデントを作成できます。
    b. 複合アラームは、作成した他のアラームのアラーム条件を考慮するルール式で構成されます。複合アラームは、すべてのルール条件が満たされた場合にのみアラーム状態になります。複合アラームのルール式で指定されるアラームには、メトリクスアラームや追加の複合アラームを含めることができます。複合アラームは、状態が変更されたときに Amazon SNS 通知を送信できます。また、ALARM 状態になったときに Systems Manager の OpsItems またはインシデントを作成できますが、EC2 アクションまたは Auto Scaling アクションを実行することはできません。
2. Amazon SNS 通知を設定します。CloudWatch アラームを作成する際には、アラームの状態が変化したときに通知を送信する Amazon SNS トピックを含めることができます。
3. 指定された CloudWatch アラームに一致するルールを EventBridge に作成します。各ルールは、Lambda 関数を含む複数のターゲットをサポートします。例えば、使用可能なディスク容量

が少なくなったときに起動するアラームを定義できます。このアラームにより、領域をクリーンアップする Lambda 関数が EventBridge ルールを介してトリガーされます。EventBridge ターゲットの詳細については、「EventBridge targets」を参照してください。

リソース

関連する Well-Architected のベストプラクティス:

- REL06-BP01 ワークロードのすべてのコンポーネントをモニタリングする (生成)
- REL06-BP02 メトリクスを定義および計算する (集計)
- REL12-BP01 プレイブックを使用して障害を調査する

関連ドキュメント:

- Amazon CloudWatch
- CloudWatch Logs insights
- Amazon CloudWatch でのアラームの使用
- Amazon CloudWatch ダッシュボードの使用
- Amazon CloudWatch メトリクスを使用する
- Amazon SNS 通知の設定
- CloudWatch 異常検出
- CloudWatch Logs data protection
- Amazon EventBridge
- Amazon Simple Notification Service

関連動画:

- reinvent 2022 observability videos
- AWS re:Invent 2022 - Observability best practices at Amazon

関連する例:

- つのオブザーバビリティワークショップ
- Amazon EventBridge to AWS Lambda with feedback control by Amazon CloudWatch Alarms

REL06-BP04 レスポンスを自動化する (リアルタイム処理とアラーム)

自動化を使用して、イベントが検出されたときにアクションを実行します (例えば、障害が発生したコンポーネントを交換します)。

アラームの自動リアルタイム処理が実装されているため、アラームがトリガーされたときにシステムが迅速に是正措置を講じ、障害やサービスの低下を防ぐことができます。アラームへの自動対応には、障害が起きたコンポーネントの交換、コンピューティングキャパシティの調整、正常なホスト、アベイラビリティーゾーン、その他のリージョンへのトラフィックのリダイレクト、オペレーターへの通知などがあります。

期待される成果: リアルタイムのアラームが特定され、アラームの自動処理が設定され、サービスレベル目標とサービスレベルアグリーメント (SLA) を達成するための適切なアクションが呼び出されます。自動処理は、単一コンポーネントの自己修復アクティビティからサイト全体のフェイルオーバーまで多岐にわたります。

一般的なアンチパターン:

- 主要なリアルタイムアラームの明確なインベントリまたはカタログがない。
- 重大なアラームへの自動対応 (例えば、コンピューティングが枯渇しそうになると、オートスケーリングが行われる) が欠如している。
- アラームへの対応が矛盾している。
- オペレーターがアラート通知を受け取ったときに従うべき標準作業手順書 (SOP) がない。
- 構成変更がモニタリングされていない。構成変更が検出されないと、ワークロードのダウンタイムが生じる可能性があります。
- 意図しない構成変更を取り消す戦略がない。

このベストプラクティスを活用するメリット: アラーム処理を自動化することで、システムの回復力を向上させることができます。システムが自動的に是正措置を講じるため、人が介入することでミスが生じやすい手作業を減らすことができます。ワークロードの可用性の目標を達成し、サービスの中断を低減します。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

アラートを効果的に管理し、対応を自動化するには、重要度と影響に基づいてアラートを分類し、対応手順を文書化し、対応計画を立ててからタスクをランク付けします。

特定のアクション (たいていはランブックに詳細が記載されている) が必要なタスクを特定し、ランブックとプレイブックをすべて調べて、どのタスクを自動化できるか判断します。アクションを定義できる場合、たいていは自動化できます。アクションを自動化できない場合は、手作業による手順を SOP に記録し、オペレーターにその手順の訓練をします。手作業のプロセスは継続的に見直し、アラートへの対応を自動化する計画を立て、実践できる余地がないか検討してください。

実装手順

1. アラームのインベントリを作成する: すべてのアラームのリストを取得するには、Amazon CloudWatch コマンド `describe-alarms` を使用して AWS CLI を使用できます。設定したアラームの数によっては、ページ分割を使用して各呼び出しのアラームのサブセットを取得するか、または AWS SDK を使用して API コールを使用してアラームを取得できます。
2. すべてのアラームアクションを文書化する: 手動か自動かにかかわらず、すべてのアラームとそのアクションでランブックを更新します。AWS Systems Manager には、定義済みのランブックが用意されています。詳細については、「ランブックの使用」を参照してください。ランブックコンテンツを表示する方法の詳細については、「View runbook content」を参照してください。
3. アラームアクションを設定して管理する: アクションが必要なアラームについては、CloudWatch SDK を使用して自動アクションを指定します。例えば、アラームに応じてアクションを作成して有効にする、またはアラームに応じてアクションを無効にする形で、CloudWatch アラームに基づいて Amazon EC2 インスタンスの状態を自動的に変更できます。

   Amazon EventBridge を使用すると、アプリケーションの可用性の問題やリソースの変更などのシステムイベントに自動的に対応できます。ルールを作成して、注目しているイベントと、イベントがルールに一致した場合に実行するアクションを指定できます。自動的に開始できるアクションには、AWS Lambda 関数の呼び出し、Amazon EC2 Run Command の呼び出し、Amazon Kinesis Data Streams へのイベントのリレー、「EventBridge を使用して Amazon EC2 を自動化する」の表示が含まれます。
4. 標準作業手順書 (SOP): アプリケーションコンポーネントに基づいて、AWS Resilience Hub は複数の SOP テンプレートを推奨します。これらの SOP を使用して、アラートが発生した場合にオペレーターが従うべきプロセスをすべて文書化できます。また、Resilience Hub のレコメンデーションに基づいて SOP を作成することもできます。その場合は、回復ポリシーを関連付けた Resilience Hub のアプリケーションと、そのアプリケーションに対する回復力評価の履歴が必要です。SOP のレコメンデーションは、回復力の評価を受けて作成されます。

   Resilience Hub は Systems Manager と連携して、SOP の基礎として使用できる多数の SSM ドキュメントを提供することで、SOP の手順を自動化します。例えば、Resilience Hub は既存の

SSM 自動化ドキュメントに基づいてディスク容量を追加するための SOP を推奨する場合があります。

5. Amazon DevOps Guru を使用して自動化アクションを実行する: Amazon DevOps Guru を使用して、異常な動作についてアプリケーションリソースを自動的にモニタリングし、的を絞ったレコメンデーションを提供することにより、問題の識別を速めて修復時間を短縮できます。DevOps Guru を使用すると、Amazon CloudWatch メトリクス、AWS Config、AWS CloudFormation、AWS X-Ray など、複数のソースからの運用データのストリームをほぼリアルタイムでモニタリングできます。また、DevOps Guru を使用して OpsCenter で OpsItems を自動的に作成し、イベントを EventBridge に送信して追加の自動化を行うこともできます。

リソース

関連するベストプラクティス:

- REL06-BP01 ワークロードのすべてのコンポーネントをモニタリングする (生成)
- REL06-BP02 メトリクスを定義および計算する (集計)
- REL06-BP03 通知を送信する (リアルタイム処理とアラーム)
- REL08-BP01 デプロイなどの標準的なアクティビティにランブックを使用する

関連ドキュメント:

- AWS Systems Manager Automation
- AWS リソースのイベントでトリガーされる EventBridge ルールの作成
- 1 つのオブザーバビリティワークショップ
- Amazon Builders' Library: 運用の可視性を高めるために分散システムを装備する
- Amazon DevOps Guru とは
- オートメーションドキュメント (プレイブック) の使用

関連動画:

- AWS re:Invent 2022 - Observability best practices at Amazon
- AWS re:Invent 2020: Automate anything with AWS Systems Manager
- Introduction to AWS Resilience Hub
- Create Custom Ticket Systems for Amazon DevOps Guru Notifications

- Enable Multi-Account Insight Aggregation with Amazon DevOps Guru

関連する例:

- Amazon CloudWatch and Systems Manager Workshop

REL06-BP05 ログの分析

ログファイルとメトリクスの履歴を収集し、これらを分析して、幅広いトレンドとワークロードの洞察が得られます。

Amazon CloudWatch Logs Insights は、シンプルかつ強力なクエリ言語をサポートし、ログデータの分析に使用できます。Amazon CloudWatch Logs ではさらに、シームレスにデータを Amazon S3 に送ってデータを使用したり、または Amazon Athena に送ってデータをクエリしたりできるサブスクリプションもサポートしています。豊富な種類のフォーマットのクエリがサポートされています。詳細については、Amazon Athena ユーザーガイドでサポートされる SerDes およびデータ形式を参照してください。巨大なログファイルセットの分析では、Amazon EMR クラスターを実行してペタバイト規模の分析を実行できます。

集計、処理、保存、分析を実行できる多数のツールが AWS パートナーやサードパーティーによって提供されています。このようなツールには、New Relic、Splunk、Loggly、Logstash、CloudHealth、Nagios などがあります。ただし、システムやアプリケーションログの外で行うデータ生成は各クラウドプロバイダーに固有であり、また多くの場合サービスごとに固有です。

モニタリングプロセスで見落とされがちな点は、データ管理です。モニタリングのためのデータ保存要件を決定し、それに応じたライフサイクルポリシーを適用する必要があります。Amazon S3 は、S3 バケットレベルのライフサイクル管理をサポートしています。このライフサイクル管理には、バケット内のパスごとに異なる管理方法を適用できます。ライフサイクルの最終段階では、データを Amazon Glacier に移行して長期保存し、保存期間の終了後には期限切れにすることができます。S3 Intelligent-Tiering ストレージクラスは、パフォーマンスへの影響や運用のオーバーヘッドなしに、データを最も費用対効果の高いアクセス階層に自動的に移動することにより、コストを最適化できるように設計されています。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

- CloudWatch Logs Insights を使用すると、Amazon CloudWatch Logs のログデータをインタラクティブに検索し分析することが可能になります。
  - CloudWatch Logs Insights を使用したログデータの分析
  - Amazon CloudWatch Logs Insights のサンプルクエリ
- Amazon CloudWatch Logs を使用して、Amazon Athena でデータのクエリを実行できる Amazon S3 にログを送信します。
  - Amazon Athena で Amazon S3 サーバーアクセスログを分析する方法を教えてください。
    - サーバーアクセスログバケットの S3 ライフサイクルポリシーを作成します。ライフサイクルポリシーを設定して、定期的にログファイルを削除します。これにより、各クエリで Athena が分析するデータの量が減ります。
      - S3 バケットのライフサイクルポリシーを作成する方法

リソース

関連ドキュメント:

- Amazon CloudWatch Logs Insights のサンプルクエリ
- CloudWatch Logs Insights を使用したログデータの分析
- Amazon CloudWatch Synthetics と AWS X-Ray でのデバッグ
- S3 バケットのライフサイクルポリシーを作成する方法
- Amazon Athena で Amazon S3 サーバーアクセスログを分析する方法を教えてください。
- 1 つのオブザーバビリティワークショップ
- Amazon Builders' Library: 運用の可視性を高めるために分散システムを装備する

REL06-BP06 モニタリングの範囲とメトリクスを定期的に確認する

ワークロードモニタリングの実装方法を頻繁に確認し、ワークロードとそのアーキテクチャの進化に合わせて更新します。モニタリングの定期的な監査は、障害インジケータの見逃しや見落としのリスクを低減し、ワークロードが可用性の目標を達成するのに役立ちます。

効果的なモニタリングは主要なビジネスメトリクスに根ざしており、ビジネスの優先順位が変化するにつれて進化します。監視レビュープロセスでは、サービスレベルインジケーター (SLI) を重視し、

インフラストラクチャ、アプリケーション、クライアント、ユーザーからのインサイトを取り入れる必要があります。

期待される成果: 効果的なモニタリング戦略があり、重要なイベントや変更の後だけでなく、定期的にレビューおよび更新されます。ワークロードとビジネス要件が進化しても、主要なアプリケーションヘルスインジケータが依然として適切であることを確認します。

一般的なアンチパターン:

- デフォルトのメトリクスのみを収集している。
- モニタリング戦略は設定するが、確認することはない。
- 主要な変更がデプロイされる際に、モニタリングについて話し合わない。
- 古いメトリクスを信頼して、ワークロードの状態を判断している。
- 運用チームは、古いメトリクスとしきい値が原因で誤検出アラートの対処に追われている。
- モニタリングされていないアプリケーションコンポーネントのオブザーバビリティが不足している。
- モニタリングでは低レベルの技術メトリクスのみに焦点を当て、ビジネスメトリクスを除外している。

このベストプラクティスを活用するメリット: モニタリングを定期的に確認すると、潜在的な問題を予測し、それらを検出できることを確認できます。また、以前のレビューで見逃した可能性のある死角を発見できるため、問題を検出する能力がさらに向上します。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

運用準備状況レビュー (ORR) プロセス中にモニタリングメトリクスと範囲を確認します。一貫したスケジュールで定期的な運用準備状況レビューを実行して、現在のワークロードと設定したモニタリングの間にギャップがあるかどうかを評価します。運用パフォーマンスのレビューと知識の共有を定期的に行うことで、運用チームのパフォーマンスを向上させることができます。既存のアラートのしきい値がまだ適切かどうかを検証し、運用チームが誤検出アラートを受信している状況や、モニタリングすべきアプリケーションのモニタリングされていない状況を確認します。

耐障害性分析フレームワークは、プロセスの進行に役立つガイダンスを提供します。フレームワークの焦点は、潜在的な障害モードと、その影響を軽減するために使用できる予防および修正コントロー

ルを特定することです。この知識は、モニタリングとアラートを行う適切なメトリクスとイベントを特定するのに役立ちます。

実装手順

1. ワークロードダッシュボードの定期的なレビューをスケジュールし、実施します。検査する深度に応じて異なる頻度にすることができます。
2. メトリクスの傾向を検査します。メトリクス値と履歴値を比較して、調査が必要なものを示唆している可能性がある傾向があるかどうかを確認します。これには、レイテンシーの増加、主要なビジネス機能の減少、失敗レスポンスの増加などがあります。
3. メトリクスの外れ値や異常を検査します。これは平均または中央値でマスキングできます。時間枠内の最高値と最低値を調べ、通常の境界をはるかに超えている観測結果の原因を調査します。これらの原因を引き続き削除すると、ワークロードパフォーマンスの一貫性の向上に応じて、期待されるメトリクスの境界を絞ることができます。
4. 行動の急変を探します。メトリクスの数量または方向性の突然の変化は、アプリケーションに変更があったこと、または追跡するためにさらなるメトリクスを追加する必要がある外部要因があることを示唆している可能性があります。
5. 現在のモニタリング戦略に、引き続きアプリケーションとの関連性があるかどうかを確認します。以前のインシデントの分析 (または耐障害性分析フレームワーク) に基づいて、モニタリングスコープに組み込む必要があるアプリケーションの追加の側面があるかどうかを評価します。
6. リアルユーザーモニタリング (RUM) メトリクスを確認して、アプリケーション機能のカバレッジにギャップがないかどうかを確認します。
7. 変更管理プロセスをレビューします。必要に応じて手順を更新し、変更を承認する前に実行する必要があるモニタリング分析ステップを含めます。
8. 運用準備状況の確認とエラープロセスの修正の一環として、モニタリングレビューを実装します。

リソース

関連するベストプラクティス

- REL06-BP01 ワークロードのすべてのコンポーネントをモニタリングする (生成)
- REL06-BP02 メトリクスを定義および計算する (集計)
- REL06-BP07 システムを通じたリクエストのエンドツーエンドのトレースをモニタリングする
- REL12-BP02 インシデント後の分析を実行する

- REL12-BP06 定期的にゲームデーを実施する

関連ドキュメント:

- correction of error (COE) を開発すべき理由
- Amazon CloudWatch ダッシュボードの使用
- 運用を可視化するためのダッシュボードの構築
- マルチ AZ の高度なレジリエンスパターン - グレー障害
- Amazon CloudWatch Logs Insights のサンプルクエリ
- Amazon CloudWatch Synthetics と AWS X-Ray でのデバッグ
- つのオブザーバビリティワークショップ
- Amazon Builders' Library: 運用の可視性を高めるために分散システムを装備する
- Amazon CloudWatch ダッシュボードの使用
- AWS Observability Best Practices
- 耐障害性分析フレームワーク
- 耐障害性分析フレームワーク - オブザーバビリティ
- 運用準備状況レビュー - ORR

REL06-BP07 システムを通じたリクエストのエンドツーエンドのトレースをモニタリングする

サービスコンポーネントで処理されるリクエストをトレースすることで、製品チームではより簡単に問題の分析とデバッグを行い、パフォーマンスを向上させることができます。

期待される成果: すべてのコンポーネントを網羅的にトレースできるワークロードは、デバッグが容易で、根本原因の発見を簡略化することで、エラーやレイテンシーの解決までの平均時間 (MTTR) を改善します。エンドツーエンドのトレースによって影響を受けるコンポーネントを検出し、エラーやレイテンシーの根本原因の詳細調査にかかる時間を短縮できます。

一般的なアンチパターン:

- トレースは一部のコンポーネントに使用されますが、すべてのコンポーネントで使用されるわけではありません。例えば、AWS Lambda のトレースを行わない場合は、ワークロードの急増でのコールドスタートが原因で生じたレイテンシーを明確に把握できない可能性があります。

- Synthetic Canaries やリアルユーザーモニタリング (RUM) には、トレースは設定されていません。Canary や RUM を使用しない場合、クライアントインタラクションのテレメトリがトレース分析から除外され、パフォーマンスプロファイルが不完全な状態になります。
- ハイブリッドワークロードには、クラウドネイティブとサードパーティーのトレースツールの両方が含まれていますが、単一のトレースソリューションを選択し、完全に統合する手段は講じられていません。選択したトレースソリューションに基づいて、クラウドネイティブのトレース SDK を使用して、クラウドネイティブではないコンポーネントを測定するか、サードパーティーツールを使用してクラウドネイティブのトレーステレメトリを取り込むように設定する必要があります。

このベストプラクティスを活用するメリット: 開発チームでは、問題についてアラートを受けることで、ロギング、パフォーマンス、障害に対するコンポーネントごとの相関関係など、システムコンポーネントの相互作用の全体像を把握できます。トレースによって根本原因を視覚的に把握しやすくなるため、根本原因の究明に費やす時間を短縮できます。チームではコンポーネントの相互作用を詳細に理解することで、問題を解決する際に、より適切で迅速な意思決定を行うことができます。システムトレースの分析によって、ディザスタリカバリ (DR) フェイルオーバーの開始時期、自己修復戦略を実行する場所などの意思決定を改善することで、最終的にサービスに対する顧客満足度の向上につながります。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

分散アプリケーションを運用するチームでは、トレースツールを使用して相関識別子を設定し、リクエストのトレースを収集して、接続されたコンポーネントのサービスマップを作成できます。サービスクライアント、ミドルウェアゲートウェイ、イベントバス、コンピューティングコンポーネント、キーバリューストアやデータベースを含むストレージなど、すべてのアプリケーションコンポーネントをリクエストトレースに含める必要があります。エンドツーエンドのトレース設定に Synthetic Canaries とリアルユーザーモニタリングを組み込んで、リモートクライアントとのやり取りやレイテンシーを測定することで、サービスレベル契約や目標に対するシステムのパフォーマンスを正確に評価できます。

AWS X-Ray および Amazon CloudWatch アプリケーションモニタリングの測定サービスを使用して、アプリケーションを通過するリクエストの全体像を把握できます。X-Ray は、アプリケーションのテレメトリを収集し、ペイロード、関数、トレース、サービス、API 全般の可視化およびフィルター処理が可能で、ノーコードまたはローコードのシステムコンポーネントに対して有効にできます。CloudWatch アプリケーションのモニタリングには ServiceLens が含まれており、トレースをメトリクス、ログ、アラームと統合します。CloudWatch アプリケーションモニタリングには、エンド

ポイントと API をモニタリングするための Synthetics や、ウェブアプリケーションクライアントを測定するためのリアルユーザーモニタリングも含まれています。

実装手順

- Amazon S3、AWS Lambda、Amazon API Gateway など、サポートされているすべてのネイティブサービスで AWS X-Ray を使用します。これらの AWS サービスでは、インフラストラクチャをコードとして、AWS SDK、または AWS マネジメントコンソールを使用して設定を切り替え、X-Ray を有効にできます。
- 測定アプリケーション AWS Distro for Open Telemetry および X-Ray またはサードパーティーの収集エージェント。
- プログラミング言語固有の実装については、AWS X-Ray 開発者ガイドを参照してください。これらのドキュメントでは、HTTP リクエスト、SQL クエリ、アプリケーションのプログラミング言語固有のその他のプロセスを測定する方法について詳しく説明します。
- Amazon CloudWatch Synthetic の Canary および Amazon CloudWatch RUM の X-Ray トレースを使用して、エンドユーザークライアントからダウンストリームの AWS インフラストラクチャを経由するリクエストパスを分析します。
- リソースの健全性と Canary テレメトリに基づき CloudWatch メトリクスとアラームを設定することで、チームでは迅速に問題についてアラートを発し、ServiceLens でトレースやサービスマップを詳しく調査できます。
- プライマリトレースソリューションにサードパーティー製ツールを使用している場合は、Datadog、New Relic、Dynatrace などのサードパーティートレースツールの X-Ray 統合を有効にします。

リソース

関連するベストプラクティス:

- REL06-BP01 ワークロードのすべてのコンポーネントをモニタリングする (生成)
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する

関連ドキュメント:

- とはAWS X-Ray
- Amazon CloudWatch : アプリケーションモニタリング
- Amazon CloudWatch Synthetics と AWS X-Ray でのデバッグ

- Amazon Builders' Library: 運用の可視性を高めるために分散システムを装備する
- Integrating AWS X-Ray with other AWS services
- AWS Distro for OpenTelemetry と AWS X-Ray
- Amazon CloudWatch : 合成モニタリングを使用する
- Amazon CloudWatch: CloudWatch RUM を使用する
- Set up Amazon CloudWatch synthetics canary and Amazon CloudWatch alarm
- 可用性およびその他: AWS の分散システムの回復力の理解と向上

関連する例:

- 1 つのオブザーバビリティワークショップ

関連動画:

- AWS re:Invent 2022 - How to monitor applications across multiple accounts
- How to Monitor your AWS Applications

関連ツール:

- AWS X-Ray
- Amazon CloudWatch
- Amazon Route 53

需要の変化に適応するようにワークロードを設計する

スケーラブルなワークロードでは、リソースを自動的に追加または削除することで、任意の時点での需要により合致できる伸縮性を得られます。

ベストプラクティス

- REL07-BP01 リソースの取得またはスケーリング時に自動化を使用する
- REL07-BP02 ワークロードの障害を検出したときにリソースを取得する
- REL07-BP03 ワークロードにより多くのリソースが必要であることを検出した時点でリソースを取得する

- REL07-BP04 ワークロードの負荷テストを実施する

REL07-BP01 リソースの取得またはスケーリング時に自動化を使用する

クラウドの信頼性の基盤は、インフラストラクチャとリソースのプログラム定義、プロビジョニング、管理です。自動化は、リソースのプロビジョニングを合理化し、一貫性のある安全なデプロイを促進し、インフラストラクチャ全体でリソースを拡張するのに役立ちます。

期待される成果: Infrastructure as Code (IaC) を管理する。バージョン管理システム (VCS) でインフラストラクチャコードを定義して維持します。AWS リソースのプロビジョニングを自動化されたメカニズムに委任し、Application Load Balancer (ALB)、Network Load Balancer (NLB)、Auto Scaling グループなどのマネージドサービスを活用します。継続的インテグレーション/継続的デリバリー (CI/CD) のパイプラインを使用してリソースをプロビジョニングすることで、Auto Scaling 設定の更新を含むリソースの更新がコードの変更によって自動的に開始されます。

一般的なアンチパターン:

- コマンドラインまたは AWS マネジメントコンソール (click-ops とも呼ばれる) でリソースを手動でデプロイしている。
- アプリケーションコンポーネントまたはリソースを緊密に結合し、その結果として柔軟性のないアーキテクチャを作成している。
- ビジネス要件、トラフィックパターン、または新しいリソースタイプの変化に適応しない柔軟性のないスケーリングポリシーを実装している。
- 予想される需要を満たすためのキャパシティを手動で見積もっている。

このベストプラクティスを活用するメリット: Infrastructure as Code (IaC) を使用すると、インフラストラクチャをプログラムで定義できます。これにより、アプリケーションの変更と同じソフトウェア開発ライフサイクルでインフラストラクチャの変更を管理することができるため、一貫性と再現性が向上し、エラーが発生しやすい手動タスクのリスクが軽減されます。自動化された配信パイプラインで IaC を実装することにより、リソースのプロビジョニングと更新のプロセスをさらに合理化できます。手動による介入を必要とせずに、インフラストラクチャの更新を確実かつ効率的に展開できます。この俊敏性は、変動する需要に合わせてリソースをスケーリングするときに特に重要です。

IaC および配信パイプラインと併せて、動的で自動化されたリソーススケーリングを実現できます。主要なメトリクスをモニタリングし、事前定義されたスケーリングポリシーを適用することにより、Auto Scaling は必要に応じてリソースを自動的にプロビジョニングまたはプロビジョニング解除できるため、パフォーマンスとコスト効率が向上します。これにより、アプリケーションやワークロードの要件の変化に応じて、手動エラーや遅延が発生する可能性が低くなります。

IaC、自動化された配信パイプライン、Auto Scaling を組み合わせることで、組織は自信を持って環境をプロビジョニング、更新、スケーリングできるようになります。この自動化は、応答性、耐障害性、効率的なマネージド型のクラウドインフラストラクチャを維持するために不可欠です。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

CI/CD パイプラインと AWS アーキテクチャの Infrastructure as Code (IaC) で自動化を設定するには、Git などのバージョン管理システムを選択して IaC テンプレートと設定を保存します。これらのテンプレートは、AWS CloudFormation などのツールを使用して記述できます。開始するには、これらのテンプレート内でインフラストラクチャコンポーネント (AWS VPC、Amazon EC2 Auto Scaling グループ、Amazon RDS データベースなど) を定義します。

次に、これらの IaC テンプレートを CI/CD パイプラインと統合してデプロイプロセスを自動化します。AWS CodePipeline のシームレスな AWS ネイティブソリューションを活用するか、サードパーティーの CI/CD ソリューションを使用できます。バージョン管理リポジトリに変更が発生したときにアクティブ化するパイプラインを作成します。IaC テンプレートをリントして検証するステージを含むようにパイプラインを設定し、インフラストラクチャをステージング環境にデプロイし、自動テストを実行し、最後に本番環境にデプロイします。必要に応じて承認ステップを組み込み、変更に対するコントロールを維持します。この自動パイプラインは、デプロイを高速化するだけでなく、環境間の一貫性と信頼性も高めることができます。

IaC で Amazon EC2 インスタンス、Amazon ECS タスク、データベースレプリカなどのリソースの Auto Scaling を設定し、必要に応じて自動でスケールアウトとスケールインを提供します。このアプローチは、アプリケーションの可用性とパフォーマンスを向上させ、需要に基づいてリソースを動的に調整することによってコストを最適化します。サポートされているリソースのリストについては、「Amazon EC2 Auto Scaling」と「AWS Auto Scaling」を参照してください。

実装手順

1. ソースコードリポジトリを作成および使用して、インフラストラクチャの設定を制御するコードを保存します。このリポジトリに変更をコミットして、進行中の変更を反映します。
2. インフラストラクチャを最新の状態に保ち、意図した状態からの不整合 (ドリフト) を検出するには、AWS CloudFormation などの Infrastructure as Code ソリューションを選択します。
3. IaC プラットフォームを CI/CD パイプラインと統合してデプロイを自動化します。
4. リソースの自動スケーリングに適したメトリクスを決定して収集します。

5. ワークロードコンポーネントに適したスケールアウトおよびスケールインポリシーを使用して、リソースの自動スケーリングを設定します。予測可能な使用パターンには、スケジュールされたスケーリングの使用を検討してください。
6. デプロイをモニタリングして障害とリグレッションを検出します。CI/CD プラットフォーム内にロールバックメカニズムを実装して、必要に応じて変更を元に戻します。

リソース

関連ドキュメント:

- AWS Auto Scaling: スケーリングプランの仕組み
- AWS Marketplace: 自動スケーリングで使用できる製品
- DynamoDB Auto Scaling によるスループットキャパシティの自動管理
- Auto Scaling グループでロードバランサーを使用する
- AWS Global Accelerator とは?
- Amazon EC2 Auto Scaling とは
- What is AWS Auto Scaling?
- Amazon CloudFront とは何ですか?
- Amazon Route 53 とは?
- Elastic Load Balancing とは?
- Network Load Balancer とは?
- 「Application Load Balancer とは?」
- Jenkins を AWS CodeBuild および AWS CodeDeploy と統合する
- AWS CodePipeline を使用して 4 ステージのパイプラインを作成する

関連動画:

- Back to Basics: Deploy Your Code to Amazon EC2
- AWS Supports You | Starting Your Infrastructure as Code Solution Using AWS CloudFormation Templates
- Streamline Your Software Release Process Using AWS CodePipeline
- Monitor AWS Resources Using Amazon CloudWatch Dashboards
- Create Cross Account & Cross Region CloudWatch Dashboards | Amazon Web Services

REL07-BP02 ワークロードの障害を検出したときにリソースを取得する

可用性が影響を受ける場合、必要に応じてリソースをリアクティブにスケールし、ワークロードの可用性を復元します。

まず、ヘルスチェックとこのチェックの基準を設定して、リソースの不足が可用性に影響を与えるタイミングを示す必要があります。次に、適切な担当者に通知してリソースを手動でスケールするか、オートメーションを開始してリソースを自動的にスケールします。

スケーリングはワークロードに合わせて手動で調整できます。例えば、Auto Scaling グループの EC2 インスタンスの数の変更や、DynamoDB テーブルのスループットの変更は、AWS マネジメントコンソールまたは AWS CLI で行うことができます。ただし、可能な限り自動化を使用する必要があります (「リソースを取得またはスケールするときに自動化を使用する」を参照)。

期待される成果: 障害やカスタマーエクスペリエンスの低下が検知された時点で、可用性を回復するためのスケーリングアクティビティ (自動または手動) が開始されます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

ワークロードのすべてのコンポーネントにオブザーバビリティとモニタリングを実装して、カスタマーエクスペリエンスを監視し、障害を検知します。必要なリソースをスケールする手順 (手動または自動) を定義します。詳細については、「REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する」を参照してください。

実装手順

- 必要なリソースをスケールする手順 (手動または自動) を定義します。
  - スケーリングの手順は、ワークロード内のさまざまなコンポーネントの設計方法に応じて異なります。
  - また、使用されている基盤のテクノロジーによっても異なります。
    - AWS Auto Scaling を使用するコンポーネントでは、スケーリングプランを使用して、リソースをスケールするための一連の指示を設定できます。AWS CloudFormation を使用または AWS リソースにタグを追加する場合、アプリケーションごとに異なる一連のリソース用にスケーリングプランを設定できます。Auto Scaling は、各リソースに合わせてカスタマイズされたスケーリング戦略のレコメンデーションを提供します。スケーリングプランを作成すると、Auto Scaling は、動的スケーリングと予測スケーリング方法を組み合わせて、スケーリ

ング戦略をサポートします。詳細については、「How scaling plans work」を参照してください。

- Amazon EC2 Auto Scaling は、アプリケーションの負荷を処理するために適切な数の Amazon EC2 インスタンスを利用できるようにします。Auto Scaling グループと呼ばれる EC2 インスタンスの集合を作成します。Auto Scaling グループごとにインスタンスの最小数と最大数を指定でき、グループがこれらの制限を下回る/上回ることがないように Amazon EC2 Auto Scaling が調整します。詳細については、「What is Amazon EC2 Auto Scaling?」を参照してください。
- Amazon DynamoDB Auto Scaling は Application Auto Scaling サービスを使用し、実際のトラフィックパターンに応じてプロビジョンドスループットキャパシティをユーザーに代わって動的に調節します。これにより、テーブルまたはグローバルセカンダリインデックスで、プロビジョニングされた読み込み/書き込みキャパシティーが拡張され、トラフィックの急激な増加をスロットリングなしに処理できるようになります。詳細については、「DynamoDB Auto Scaling によるスループットキャパシティの自動管理」を参照してください。

リソース

関連するベストプラクティス:

- REL07-BP01 リソースの取得またはスケーリング時に自動化を使用する
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する

関連ドキュメント:

- AWS Auto Scaling: スケーリングプランの仕組み
- DynamoDB Auto Scaling によるスループットキャパシティの自動管理
- Amazon EC2 Auto Scaling とは

REL07-BP03 ワークロードにより多くのリソースが必要であることを検出した時点でリソースを取得する

クラウドコンピューティングの最も重要な機能の 1 つは、リソースを動的にプロビジョニングできることです。

従来のオンプレミスコンピューティング環境では、ピーク需要に対応するために十分なキャパシティを事前に特定してプロビジョニングする必要があります。これは、高価であることと、ワークロード

のピーク時のキャパシティのニーズを過小評価している場合、可用性にリスクが生じるという点で問題です。

クラウドでは、このようなことを行う必要はありません。代わりに、必要に応じてコンピューティング、データベース、その他のリソースキャパシティをプロビジョニングして、現在および予測される需要を満たすことができます。Amazon EC2 Auto Scaling や Application Auto Scaling などの自動化されたソリューションは、指定したメトリクスに基づいてリソースをオンラインにすることができます。これにより、スケーリングプロセスが簡単で予測可能になり、常に十分なリソースを確保することによって、ワークロードの信頼性が大幅に向上します。

期待される成果: コンピューティングやその他のリソースの自動スケーリングを需要を満たすように設定します。追加のリソースをオンラインにする間、トラフィックのバーストに対応できるように、スケーリングポリシーでは十分なヘッドルームを確保してください。

一般的なアンチパターン:

- 一定数のスケーラブルなリソースをプロビジョニングする。
- 実際の需要と相関しないスケーリングメトリクスを選択する。
- 需要のバーストに対応するためのスケーリングプランに十分なヘッドルームを確保できていない。
- スケーリングポリシーがキャパシティを追加するタイミングが遅すぎるため、追加のリソースをオンラインにする際のキャパシティの枯渇やサービスの低下を招いている。
- 最小リソース数と最大リソース数を正しく設定できないため、スケーリングに失敗する。

このベストプラクティスを活用するメリット: 現在の需要を満たすのに十分なリソースを確保することは、ワークロードの高可用性を提供し、定義されたサービスレベル目標 (SLO) を遵守するために不可欠です。自動スケーリングを使用すると、現在および予測されている需要に対応するためにワークロードに必要な適切な量のコンピューティング、データベース、その他のリソースを提供できます。ピーク時のキャパシティのニーズを判断し、それに対応するためにリソースを静的に割り当てる必要はありません。代わりに、需要が増加すると、それに対応するためにより多くのリソースを割り当てることができ、需要が落ちた後は、リソースを非アクティブ化してコストを削減できます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

まず、ワークロードコンポーネントが自動スケーリングに適しているかどうかを判断します。これらのコンポーネントは、同じリソースを提供し、同じように動作するため、水平スケーラブルと呼ばれます。水平スケーラブルなコンポーネントの例としては、同様に構成された EC2 インスタンス、Amazon Elastic Container Service (ECS) タスク、Amazon Elastic Kubernetes Service (EKS) で

実行されているポッドなどがあります。これらのコンピューティングリソースは通常、ロードバランサーの背後に配置されており、レプリカと呼ばれます。

その他のレプリケートされたリソースには、データベースのリードレプリカ、Amazon DynamoDB テーブル、Amazon ElastiCache (Redis OSS) クラスターなどがあります。サポートされているリソースの完全なリストについては、「AWS services that you can use with Application Auto Scaling」を参照してください。

コンテナベースのアーキテクチャでは、2 つの異なる方法でスケールする必要がある場合があります。まず、水平スケーラブルなサービスを提供するコンテナをスケールする必要があるかもしれません。次に、コンピューティングリソースをスケールして、新しいコンテナ用のスペースを確保する必要があります。レイヤーごとに異なる自動スケーリングメカニズムがあります。ECS タスクをスケールするには、Application Auto Scaling を使用できます。Kubernetes ポッドをスケールするには、Horizontal Pod Autoscaler (HPA) または Kubernetes Event-driven Autoscaling (KEDA) を使用できます。コンピューティングリソースをスケールするには、ECS の場合はキャパシティプロバイダーが使用でき、Kubernetes の場合は Karpenter または Cluster Autoscaler を使用できます。

次に、自動スケーリングの実行方法を選択します。メトリクスベースのスケーリング、スケジュールされたスケーリング、予測スケーリングの 3 つの主要なオプションがあります。

メトリクスベースのスケーリング

メトリクスベースのスケーリングは、1 つ以上のスケーリングメトリクスの値に基づいてリソースをプロビジョニングします。スケーリングメトリクスとは、ワークロードの需要に対応するメトリクスです。適切なスケーリングメトリクスを決定する良い方法は、非本番環境で負荷テストを実行することです。負荷テスト中は、スケーラブルなリソースの数を一定に保ったまま、需要 (スループット、同時実行数、シミュレート対象ユーザー数など) を徐々に増やします。次に、需要の増加に応じて増加 (または減少) するメトリクスを探し、逆に需要の減少に応じて減少 (または増加) するメトリクスを探します。一般的なスケーリングメトリクスには、CPU 使用率、ワークキューの深さ (Amazon SQS キューなど)、アクティブユーザー数、ネットワークスループットが含まれます。

> **Note**
>
> AWS は、ほとんどのアプリケーションで、アプリケーションがウォームアップするにつれてメモリ使用率が増加し、その後、安定した値に達することを観察しました。需要が減少すると、通常、メモリ使用率は並行して減少するのではなく、上昇したままになります。メモリ使用率は、需要に応じて増減するという両方向の需要に対応していないため、このメトリクスを自動スケーリング用に選択する前に慎重に検討してください。

メトリクスベースのスケーリングは、潜在的なオペレーションです。使用率メトリクスが自動スケーリングメカニズムに伝達されるまでに数分かかることがあります。これらのメカニズムは通常、応答する前に、需要の増加を示す明確なシグナルを待ちます。その後、自動スケーラーが新しいリソースを作成すると、フルサービスになるまでさらに時間がかかる場合があります。このため、スケーリングメトリクスのターゲットをフル稼働 (例えば、CPU 使用率 90%) に近づけすぎないようにすることが重要です。そうすることにより、追加のキャパシティがオンラインになる前に、既存のリソースキャパシティが枯渇するリスクがあります。一般的なリソース使用率のターゲットは、需要パターンと追加のリソースのプロビジョニングに必要な時間に応じて、最適な可用性を得るために 50～70% の範囲になります。

スケジュールに基づくスケーリング

スケジュールされたスケーリングは、カレンダーまたは時間帯に基づいてリソースをプロビジョニングまたは削除します。これは、平日の営業時間中のピーク使用率やセールスイベントなど、需要が予測可能なワークロードに頻繁に使用されます。Amazon EC2 Auto Scaling と Application Auto Scaling はどちらも、スケジュールされたスケーリングをサポートしています。KEDA の cron スケーラーは、Kubernetes ポッドのスケジュールされたスケーリングをサポートしています。

予測スケーリング

予測スケーリングでは、機械学習を使用して、予想される需要に基づいてリソースを自動的にスケールします。予測スケーリングは、指定した使用率メトリクスの履歴値を分析して、その将来の値を継続的に予測します。次に、予測値を使用してリソースをスケールアップまたはスケールダウンします。Amazon EC2 Auto Scaling は、予測スケーリングを実行できます。

実装手順

1. ワークロードコンポーネントが自動スケーリングに適しているかどうかを判断します。
2. メトリクスベースのスケーリング、スケジュールされたスケーリング、予測スケーリングの中から、どのようなスケーリングメカニズムがワークロードに最適かを判断します。
3. コンポーネントに適した自動スケーリングメカニズムを選択します。Amazon EC2 インスタンスの場合は、Amazon EC2 Auto Scaling を使用します。その他の AWS のサービスについては、Application Auto Scaling を使用します。Kubernetes ポッド (Amazon EKS クラスターで実行されているものなど) の場合は、Horizontal Pod Autoscaler (HPA) または Kubernetes Event-driven Autoscaling (KEDA) を検討してください。Kubernetes または EKS ノードの場合は、Karpenter と Cluster Auto Scaler (CAS) を検討してください。
4. メトリクスのスケーリングまたはスケジュールされたスケーリングについては、負荷テストを実行して、ワークロードに適したスケーリングメトリクスとターゲット値を決定します。スケ

ジュールされたスケーリングでは、選択した日時に必要なリソースの数を決定します。予想されるピークトラフィックに対応するために必要なリソースの最大数を決定します。

5. 上記で収集した情報に基づいて、自動スケーラーを設定します。詳細は、自動スケーリングサービスのドキュメントを参照してください。最大スケーリング制限と最小スケーリング制限が正しく設定されていることを確認します。
6. スケーリング設定が期待どおりに機能していることを確認します。非本番環境で負荷テストを実行し、システムの反応を観察し、必要に応じて調整します。本番稼働で自動スケーリングを有効にする場合は、予期しない動作を通知する適切なアラームを設定します。

リソース

関連ドキュメント:

- Amazon EC2 Auto Scaling とは
- AWS Prescriptive Guidance: Load testing applications
- AWS Marketplace: 自動スケーリングで使用できる製品
- DynamoDB Auto Scaling によるスループットキャパシティの自動管理
- 新機能 – Machine Learning を中核とする EC2 の予測スケーリング
- Amazon EC2 Auto Scaling のスケジュールされたスケーリング
- Telling Stories About Little's Law

REL07-BP04 ワークロードの負荷テストを実施する

負荷テスト手法を採用して、スケーリングアクティビティがワークロード要件を満たすかどうかを測定します。

持続的な負荷テストを実行することが重要です。負荷テストによってブレークポイントを発見し、ワークロードのパフォーマンスをテストします。AWS は、本稼働ワークロードのスケールをモデル化する、一次的なテスト環境のセットアップを容易にします。クラウド上では、本稼働スケールのテスト環境をオンデマンドで作成し、テスト完了後にリソースを解放できます。テスト環境の支払いは実行時にのみ発生するため、オンプレミスでテストを実施する場合と比べて、わずかなコストで本番環境をシミュレートできます。

本番環境での負荷テストは、ゲームデーの一部として考える必要もあります。その中で、顧客の使用率が低い時間帯に本稼働システムに負荷をかけ、担当者全員がテスト結果を解釈して、発生した問題に対処できるようにします。

一般的なアンチパターン:

- 本番環境と同じ設定ではないデプロイで負荷テストを実行する。
- ワークロード全体ではなく、ワークロードの個々の部分に対してのみ負荷テストを実行する。
- 実際のリクエストの代表的なセットではなく、リクエストのサブセットを使用して負荷テストを実行する。
- 予想される負荷を下回る小さな安全率に対して負荷テストを実行する。

このベストプラクティスを活用するメリット: 負荷がかかるとアーキテクチャのどのコンポーネントに障害が発生するかを把握し、問題への対処に間に合うように、その負荷に近づいていることを示す、監視するべきメトリクスを特定して、障害の影響を防ぐことができます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

- 負荷テストを実行して、容量を追加または削除する必要があるワークロードの側面を特定します。負荷テストには、本番環境で受け取るものと同様の代表的なトラフィックを使用する必要があります。設定したメトリクスを監視しながら負荷を増やし、リソースを追加または削除する必要があるタイミングをどのメトリクスが示しているのかを判断します。
  - AWS での分散負荷テスト: 接続された数千のユーザーをシミュレートする
    - リクエストの組み合わせを特定します。さまざまなリクエストが混在している可能性があるため、トラフィックの組み合わせを特定するときは、さまざまな時間枠を確認する必要があります。
    - ロードドライバーを実装します。カスタムコード、オープンソース、または商用ソフトウェアを使用して、ロードドライバーを実装します。
    - 最初は小さな容量に対して負荷テストを実施します。1 つのインスタンスまたはコンテナと同じくらいの容量に負荷をかけることで、すぐに効果が現れます。
    - 大きな容量に対して負荷テストを実施します。この効果は分散された負荷によって異なるため、できるだけ本番環境に近い環境でテストする必要があります。

リソース

関連ドキュメント:

- AWS での分散負荷テスト: 接続された数千のユーザーをシミュレートする

- 「アプリケーションの負荷テスト」

関連動画:

- AWS Summit ANZ 2023: Accelerate with confidence through AWS Distributed Load Testing

変更の実装

新しい機能をデプロイし、ワークロードとオペレーティング環境が既知の適切にパッチが適用されたソフトウェアを実行していることを確認するには、変更を制御する必要があります。これらの変更がコントロールされていない場合、変更による影響を予測したり、変更によって発生する問題に対応することが困難になります。

リスクを最小限に抑えるための追加のデプロイパターン

機能トグル (別名、機能フラグ) は、アプリケーションの設定オプションです。特定の機能をオフにしてソフトウェアをデプロイすると、その機能はユーザー側には表示されません。この機能はその後、カナリアデプロイのようにオンにしたり、あるいは変更ベースを 100% に設定して影響を確認したりできます。デプロイに問題が発生した場合は、ロールバックしないで単純に機能をオフに戻すことができます。

障害部分を切り離したゾーンデプロイ: AWS が自社のデプロイ向けに策定した最も重要なルールの 1 つが、同じリージョンにある複数のアベイラビリティーゾーンに同時にアクセスしないことです。これは、アベイラビリティーゾーンを独立させて可用性を正しく計算するために重要となります。デプロイにあたり、このような考慮事項を念頭に置くことを推奨します。

運用準備状況レビュー (ORR)

AWS では、運用準備状況の確認を実施して、テストの完全性、モニタリング能力、さらには SLA に対するアプリケーションパフォーマンスの監査を評価し、障害時や運用における異常があったときにデータを提供することが有効であると考えています。正式な ORR は初回の本稼働デプロイの前に実施されます。AWS は定期的 (年 1 回、または重要なパフォーマンス期間の前) に ORR を実施して、想定された運用状況から逸脱していないことを確認します。運用準備状況の詳細については、「AWS Well-Architected フレームワーク」の「運用上の優秀性の柱」を参照してください。

ベストプラクティス

- REL08-BP01 デプロイなどの標準的なアクティビティにランブックを使用する
- REL08-BP02 デプロイの一部として機能テストを統合する

- REL08-BP03 デプロイの一部として回復力テストを統合する
- REL08-BP04 イミュータブルなインフラストラクチャを使用してデプロイする
- REL08-BP05 自動化を使用して変更をデプロイする

REL08-BP01 デプロイなどの標準的なアクティビティにランブックを使用する

ランブックは、特定の成果を達成するための事前定義された手順です。手動または自動のどちらでも、標準的なアクティビティを実行するにはランブックを使用します。例えば、ワークロードのデプロイ、ワークロードへのパッチの適用、DNS の変更などがあります。

例えば、デプロイ中のロールバックの安全性を確保するために、プロセスを配置します。顧客側の中断なしで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上で重要です。

ランブックの手順については、有効で効果的な手動プロセスから始めて、それをコードで実装し、必要に応じて自動実行を呼び出します。

高度に自動化された最新のワークロードに対しても、ランブックはゲームデーの実行や、厳格なレポートおよび監査の要件を満たすのに役立ちます。

プレイブックは特定のインシデントに対応するために使用し、ランブックは特定の成果を達成するために使用します。多くの場合、ランブックは日常的なアクティビティ用で、プレイブックは非日常的なイベントに対応するために使用します。

一般的なアンチパターン:

- 本番環境の設定に対して、計画されていない変更を実行する。
- デプロイを高速化するために計画の手順をスキップして、デプロイを失敗させる。
- 変更を戻すことができるかどうかをテストせずに変更を加える。

このベストプラクティスを活用するメリット: 効果的な変更計画を作成すると、影響を受けるすべてのシステムを認識できるため、変更を正常に実行する能力が向上します。テスト環境で変更を検証すると、信頼性が強化されます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

- ランブックに手順を文書化することで、一貫性を保ち、汎用イベントにすみやかに対応できるようになります。
- Infrastructure as Code の原則を適用して、インフラストラクチャを定義します。AWS CloudFormation (または信頼できるサードパーティー) を使用してインフラストラクチャを定義することで、バージョン管理ソフトウェアを使用して、バージョンおよび変更の追跡を行うことができます。
  - AWS CloudFormation (または信頼できるサードパーティープロバイダー) を使用して、インフラストラクチャを定義します。
    - What is AWS CloudFormation?
  - 優れたソフトウェア設計の原則を使用して、単一または分離されたテンプレートを作成します。
    - 実装にあたって、必要な権限、テンプレート、責任者を決定します。
      - によるアクセスの制御AWS Identity and Access Management
    - Git などの一般的なテクノロジーに基づくホスト型ソースコード管理システムを使用して、ソースコードと Infrastructure as Code (IaC) の構成を保存します。

リソース

関連ドキュメント:

- APN パートナー: 自動化されたデプロイソリューションの作成を支援できるパートナー
- AWS Marketplace: products that can be used to automate your deployments
- What is AWS CloudFormation?

関連する例:

- プレイブックとランブックによるオペレーションの自動化

REL08-BP02 デプロイの一部として機能テストを統合する

必要な機能を検証する単体テストや統合テストなどの技法を使用します。

ユニットテストは、コードの最小機能ユニットをテストして動作を検証するプロセスです。統合テストでは、各アプリケーション機能がソフトウェア要件に従って機能することを検証します。ユニット

テストはアプリケーションの一部を個別にテストすることに重点を置いていますが、統合テストでは副作用 (例えば、ミューテーションオペレーションによってデータが変更された場合の影響) を考慮します。いずれの場合も、テストをデプロイパイプラインに統合する必要があり、成功基準を満たしていない場合、パイプラインは停止またはロールバックされます。このようなテストは、パイプラインの本稼働前にステージングされた、本稼働前環境で実行されます。

これらのテストがビルドおよびデプロイアクションの一部として自動的に実行されると、最良の結果が得られます。例えば、デベロッパーは AWS CodePipeline を使用して、CodePipeline が変更を自動的に検出するソースリポジトリに変更をコミットします。アプリケーションが構築され、ユニットテストが実行されます。ユニットテストが完了すると、ビルドされたコードがテスト用のステージングサーバーにデプロイされます。ステージングサーバーから、CodePipeline は統合やロードなどの色々なテストを実行します。これらのテストが正常に完了すると、CodePipeline はテストおよび承認されたコードを本番稼働インスタンスにデプロイします。

期待される成果: オートメーションを使用してユニットテストと統合テストを実行し、コードが期待どおりに動作することを確認します。これらのテストはデプロイプロセスに統合され、テスト失敗によりデプロイが中止されます。

一般的なアンチパターン:

- デプロイのタイムラインを早めるために、デプロイプロセス中にテストの失敗や計画を無視またはバイパスする。
- テストをデプロイパイプラインの外部で手動で実行する。
- 手動の緊急ワークフローにより、自動化のテストステップを省略する。
- 自動テストを、本番環境とあまり似ていない環境で実行する。
- アプリケーションの進化に合わせて、柔軟性が低く、メンテナンス、更新、スケーリングが難しいテストスイートを構築する。

このベストプラクティスを活用するメリット: デプロイプロセス中の自動テストは問題を早期に検出するため、バグや予期しない動作で本番稼働にリリースされるリスクが軽減されます。ユニットテストでは、コードが目的どおりに動作し、API 契約が守られていることを確認します。統合テストでは、システムが指定された要件に従って動作することを検証します。これらのタイプのテストでは、ユーザーインターフェイス、API、データベース、ソースコードなどのコンポーネントの想定された動作順序を検証します。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

ソフトウェアの記述にはテスト駆動型の開発 (TDD) アプローチを採用し、コードを指定して検証するためのテストケースを開発します。開始するには、関数ごとにテストケースを作成します。テストが失敗した場合は、テストに合格するための新しいコードを記述します。このアプローチは、各関数の期待される結果を検証するのに役立ちます。ソースコードリポジトリにコードをコミットする前に、ユニットテストを実行し、合格することを確認します。

CI/CD パイプラインの構築、テスト、デプロイの各段階の一環として、ユニットテストと統合テストの両方を実装します。テストを自動化し、アプリケーションの新しいバージョンをデプロイする準備ができたら、テストを自動的に開始します。成功条件を満たさない場合、パイプラインは停止またはロールバックされます。

アプリケーションがウェブまたはモバイルアプリの場合は、複数のデスクトップブラウザまたは実際のデバイスで自動統合テストを実行します。このアプローチは、さまざまなデバイスにわたるモバイルアプリの互換性と機能を検証するのに特に役立ちます。

実装手順

1. 機能コード (テスト駆動型の開発、または TDD) を記述する前にユニットテストを記述します。ユニットテストの記述と実行が非機能的なコーディング要件となるように、コードガイドラインを確立します。
2. 識別されたテスト可能な機能をカバーする一連の自動統合テストを作成します。これらのテストでは、ユーザーインタラクションをシミュレートし、期待される結果を検証する必要があります。
3. 統合テストを実行するために必要なテスト環境を作成します。これには、本番環境を厳密に模倣するステージング環境または本番稼働前環境が含まれる場合があります。
4. AWS CodePipeline コンソールまたは AWS Command Line Interface (CLI) を使用して、ソース、ビルド、テスト、デプロイの各ステージを設定します。
5. コードの構築とテストが完了したら、アプリケーションをデプロイします。AWS CodeDeploy は、ステージング (テスト) 環境と本番環境にデプロイできます。これらの環境には、Amazon EC2 インスタンス、AWS Lambda 関数、またはオンプレミスサーバーが含まれる場合があります。アプリケーションをすべての環境にデプロイするには、同じデプロイメカニズムを使用する必要があります。
6. パイプラインの進行状況と各ステージのステータスをモニタリングします。品質チェックを使用して、テストのステータスに基づいてパイプラインをブロックします。パイプラインステージの障害や、パイプラインの完了に関する通知を受け取ることもできます。

7. テストの結果を継続的にモニタリングし、より注意が必要なパターン、リグレッション、または領域を探します。この情報を使用して、テストスイートを改善し、より堅牢なテストが必要なアプリケーションの領域を特定し、デプロイプロセスを最適化します。

リソース

関連するベストプラクティス:

- REL07-BP04 ワークロードの負荷テストを実施する
- REL08-BP03 デプロイの一部として回復力テストを統合する
- REL12-BP04 カオスエンジニアリングを使用して回復力をテストする

関連ドキュメント:

- AWS 規範ガイダンス: テスト自動化
- 継続的デリバリーと継続的インテグレーション
- 機能テストのインジケータ
- パイプラインのモニタリング
- AWS CodeBuild で AWS CodePipeline を使用してコードをテストし、ビルドを実行する
- AWS Device Farm

REL08-BP03 デプロイの一部として回復力テストを統合する

意図的にシステムに障害を導入することで回復力テストを統合し、障害発生時のシステムの能力を測定します。回復力テストは、システムでの予期しない障害の特定に重点を置いているため、通常デプロイサイクルに統合されるユニットテストや機能テストとは異なります。本番稼働前に回復力テストの統合を始めても問題ありませんが、ゲームデーの一環として、これらのテストを本番環境に実装するという目標を設定します。

期待される成果 回復力テストは、本番環境の劣化に耐えられるシステムの能力に対する信頼を構築するのに役立ちます。テストによって障害につながる可能性のある弱点を特定することで、システムを改善し、障害や劣化を自動的かつ効率的に軽減できます。

一般的なアンチパターン:

- デプロイプロセスにおけるオブザーバビリティとモニタリングの欠如

- システム障害の解決を人間に依存する
- 低品質の分析メカニズム
- システムの既知の問題に重点を置き、未知の問題点を特定するためのテストを行わない
- 障害を特定できるが、解決できない
- 検出結果とランブックが文書化されていない

このベストプラクティスを活用する利点: デプロイに統合された回復力テストは、システムの未知の問題のうち、テストを実行しないと気付かないものを特定するのに役立ちます。これらの問題は本番環境のダウンタイムにつながる可能性があります。システムでのこれらの未知の問題の特定は、検出結果の文書化、CI/CD プロセスへのテストの統合、およびランブックの作成に役立ち、効率的で反復可能なメカニズムを通じて、障害の緩和を簡素化します。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

システムのデプロイに統合できる最も一般的な回復力テストの形式は、ディザスタリカバリとカオスエンジニアリングです。

- 大規模なデプロイには、ディザスタリカバリ計画と標準運用手順 (SOP) の更新を含めます。
- 信頼性テストを自動デプロイパイプラインに統合します。AWS Resilience Hub などのサービスを CI/CD パイプラインに統合して、すべてのデプロイの一部として自動的に評価される、継続的な耐障害性評価を確立します。
- AWS Resilience Hub でアプリケーションを定義します。耐障害性評価では、アプリケーションの AWS Systems Manager ドキュメントとして復旧手順を作成するのに役立つコードスニペットが生成され、推奨される Amazon CloudWatch モニタとアラームのリストが提供されます。
- ディザスタリカバリ計画と SOP が更新されたら、ディザスタリカバリテストを実施して効果を確認します。ディザスタリカバリテストは、イベント後にシステムを復旧して通常の運用に戻ることができるかどうかを判断するのに役立ちます。さまざまなディザスタリカバリ戦略をシミュレートし、計画が稼働時間の要件を満たすのに十分であるかどうかを確認できます。一般的なディザスタリカバリ戦略には、バックアップと復元、パイロットライト、コールドスタンバイ、ウォームスタンバイ、ホットスタンバイ、アクティブ - アクティブなどがあり、コストと複雑さは戦略によって異なります。ディザスタリカバリテストの前に、目標復旧時間 (RTO) と目標復旧時点 (RPO) を定義して、シミュレートする戦略の選択を簡素化することをお勧めします。AWS には、計画とテストの開始に役立つ AWS Elastic Disaster Recovery などのディザスタリカバリツールが用意されています。

- カオスエンジニアリングのテストでは、ネットワーク障害やサービス障害などのシステムの中断を発生させます。制御された障害を使用してシミュレーションを行うことで、発生した障害の影響を抑えながら、システムの脆弱性を発見できます。他の戦略と同様に、AWS Fault Injection Service のようなサービスを使用して、非本番環境で制御された障害シミュレーションを実行し、本番環境にデプロイする前に確信を得ることができます。

リソース

関連ドキュメント:

- レジリエンステストで障害に関する実験を行いリカバリに備える
- アプリケーションを AWS Resilience Hub と AWS CodePipeline で継続的に評価する
- AWS でのディザスタリカバリ (DR) アーキテクチャ、パートI: クラウドでのリカバリの戦略
- カオスエンジニアリングを使用したワークロードのレジリエンスの検証
- カオスエンジニアリングの原則
- カオスエンジニアリングワークショップ

関連動画:

- AWS re:Invent 2020: Testing Resilience using Chaos Engineering
- Improve Application Resilience with AWS Fault Injection Service
- Prepare & Protect Your Applications From Disruption With AWS Resilience Hub

REL08-BP04 イミュータブルなインフラストラクチャを使用してデプロイする

イミュータブルなインフラストラクチャは、本稼働ワークロードで更新、セキュリティパッチ適用、設定変更がインプレースで行われないように義務付けるモデルです。変更が必要な場合、アーキテクチャは新しいインフラストラクチャに構築され、本番環境にデプロイされます。

イミュータブルなインフラストラクチャのデプロイ戦略に従って、ワークロードデプロイの信頼性、一貫性、再現性を高めましょう。

期待される成果: イミュータブルなインフラストラクチャでは、ワークロード内でインフラストラクチャリソースを実行するためのインプレース変更はできません。代わりに、変更の必要が生じた場合は、必要な変更をすべて適用した新しいインフラストラクチャリソース一式を、既存のリソースと並

行してデプロイします。このデプロイは自動的に検証され、検証に合格すると、トラフィックが新しいリソース一式に徐々にシフトします。

このデプロイ戦略は、ソフトウェアの更新、セキュリティパッチの適用、インフラストラクチャの変更、構成の更新、アプリケーションの更新などに適用されます。

一般的なアンチパターン:

- 実行中のインフラストラクチャリソースにインプレース変更を実装する。

このベストプラクティスを活用するメリット:

- 環境全体での一貫性の向上: 環境全体でインフラストラクチャリソースに違いがないため、一貫性が向上し、テストが簡素化されます。
- 設定ドリフトの削減: インフラストラクチャリソースを既知のバージョン管理された設定に置き換えることで、インフラストラクチャが既知のテスト済みで信頼できる状態に設定され、設定ドリフトを回避できます。
- 信頼性の高いアトミックデプロイ: デプロイは正常に完了するか、何も変更されないため、デプロイプロセスの一貫性と信頼性が向上します。
- デプロイの簡素化: アップグレードをサポートする必要がないため、デプロイが簡素化されます。新たにデプロイするだけでアップグレードされます。
- 迅速なロールバックとリカバリプロセスによる安全なデプロイ: 以前の動作バージョンが変更されないため、デプロイはより安全です。エラーが検出された場合はロールバックできます。
- セキュリティ体制の強化: インフラストラクチャへの変更を許可しないことで、リモートアクセスメカニズム (SSH など) を無効にすることができます。これにより、攻撃ベクトルが減少し、組織のセキュリティ体制が強化されます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

Automation

イミュータブルなインフラストラクチャのデプロイ戦略を定義するときは、再現性を高め、人為的ミスの可能性を最小限に抑えるために、可能な限り自動化を使用することをお勧めします。詳細については、「REL08-BP05 自動化による変更のデプロイ」および「安全かつハンドオフのデプロイの自動化」を参照してください。

Infrastructure as Code (IaC) では、インフラストラクチャのプロビジョニング、オーケストレーション、およびデプロイのステップがプログラムによって説明的、宣言的な方法で定義され、ソース管理システムに保存されます。IaC を活用することで、インフラストラクチャのデプロイを簡単に自動化し、インフラストラクチャの不変性を実現できます。

デプロイパターン

ワークロードの変更が必要な場合、イミュータブルなインフラストラクチャのデプロイ戦略では、必要な変更をすべて適用済みの、新しいインフラストラクチャリソース一式をデプロイすることが義務付けられています。この新しいリソースセットでは、ユーザーへの影響を最小限に抑えるロールアウトパターンに従うことが重要です。このデプロイには、主に 2 つの戦略があります。

カナリアデプロイは、通常、単一のサービスインスタンス (Canary) で実行される新しいバージョンに、少数の顧客を誘導する方法です。次に、発生した動作の変更やエラーを詳細に調べます。重大な問題が発生した場合は、canary からトラフィックを削除して、ユーザーを以前のバージョンに戻すことができます。デプロイが成功したら、変更やエラーをモニタリングしながら、完全にデプロイされるまで、希望の速度でデプロイを続行できます。AWS CodeDeploy では、デプロイ設定で カナリアデプロイを有効にすることができます。

ブルー/グリーンデプロイはカナリアデプロイに似ていますが、アプリケーションのフリート全体が並行してデプロイされる点が異なります。2 つのスタック (青と緑) 間でデプロイを交互に行います。この場合も、トラフィックを新しいバージョンに送信したときにデプロイに問題が発生した場合は、古いバージョンにフォールバックできます。通常、すべてのトラフィックが一度に切り替えられますが、各バージョンへのトラフィックの一部と Amazon Route 53 の加重 DNS ルーティング機能を使用して、新しいバージョンの採用をダイヤルアップすることもできます。AWS CodeDeploy と AWS Elastic Beanstalk は、ブルー/グリーンデプロイを有効にするデプロイ構成で設定できます。

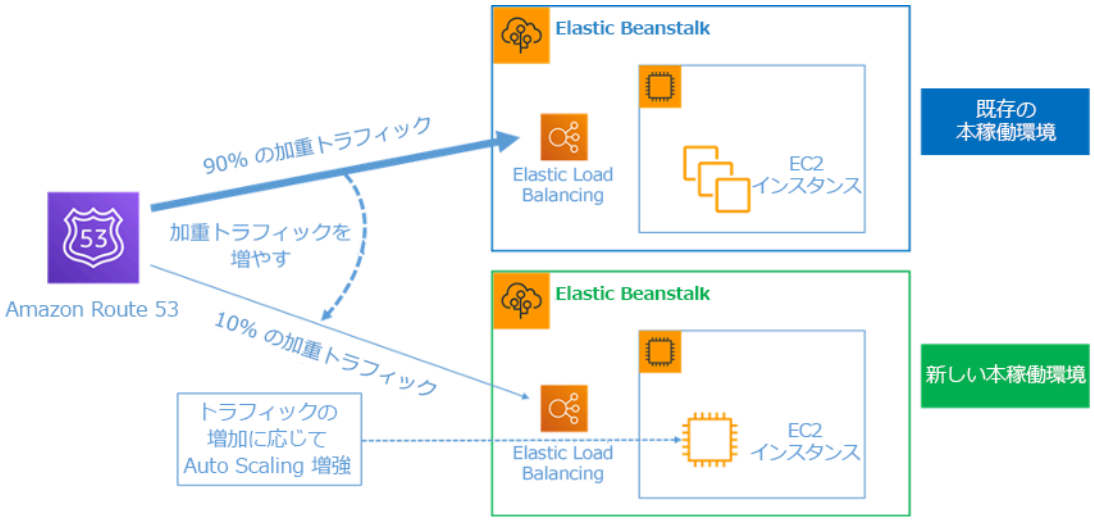


図 8: AWS Elastic Beanstalk と Amazon Route 53 によるブルー/グリーンデプロイ

ドリフト検出

ドリフトは、インフラストラクチャリソースの状態や構成が想定とは異なる変更として定義されます。適切に管理されていない構成変更は、イミュータブルなインフラストラクチャの概念と矛盾します。イミュータブルなインフラストラクチャを正常に実装するには、このような構成変更を検出し、修復する必要があります。

実装手順

- 実行中のインフラストラクチャリソースのインプレース変更は禁止します。
  - AWS Identity and Access Management (IAM) を使用すると、AWS のサービスとリソースにアクセスできるユーザーまたは内容を指定し、きめ細かいアクセス許可を一元管理し、アクセスを分析して、AWS 全体のアクセス許可を調整できます。
- インフラストラクチャリソースのデプロイを自動化して再現性を高め、人為的ミスを最小限に抑えます。
  - AWS ホワイトペーパーの「DevOps の概要」で説明されているように、自動化は AWS サービスの基礎であり、すべてのサービスと機能で内部的にサポートされています。
  - Amazon マシンイメージ (AMI)を事前に作成すると、起動時間を短縮できます。EC2 Image Builder は、カスタマイズされた、安全かつ最新の Linux または Windows カスタム AMI の作成、メンテナンス、検証、共有、デプロイを自動化するのに役立つフルマネージド AWS サービスです。
  - 次のサービスは自動化に対応しています。
    - AWS Elastic Beanstalk は、Java、.NET、PHP、Node.js、Python、Ruby、Go、Docker で開発されたウェブアプリケーションを、Apache、NGINX、Passenger、IIS などの一般的なサーバーに迅速にデプロイしてスケールするためのサービスです。
    - AWS Proton は、プラットフォームチームがインフラストラクチャのプロビジョニング、コードのデプロイ、モニタリング、更新に必要なさまざまなツールを接続および調整するのに役立ちます。AWS Proton は、サーバーレスおよびコンテナベースのアプリケーションの Infrastructure as code (IaC) のプロビジョニングおよびデプロイを可能にします。
  - IaC を活用することで、インフラストラクチャのデプロイを簡単に自動化し、インフラストラクチャの不変性を実現できます。AWS は、プログラムにより記述的、宣言的な方法でインフラストラクチャの作成、デプロイ、メンテナンスを可能にするサービスを提供しています。
    - AWS CloudFormation は、開発者が順序正しく予測可能な方法でAWS リソースを作成するのに役立ちます。リソースは、JSON または YAML 形式でテキストファイルに書き込まれます。テンプレートには、作成および管理されるリソースのタイプに応じて、特定の構文と構造が必要です。任意のコードエディタを使用して JSON または YAML でリソースを作成し、

バージョン管理システムにチェックインすると、CloudFormation が、指定されたサービスを安全で繰り返し可能な方法で構築します。

  - AWS Serverless Application Model (AWS SAM) は、AWS でサーバーレスアプリケーションを構築するために使用できるオープンソースフレームワークです。AWS SAM は他の AWS サービスと統合された、CloudFormation の拡張機能です。
  - AWS Cloud Development Kit (AWS CDK) は、使い慣れたプログラミング言語を使用して、クラウドアプリケーションリソースをモデル化およびプロビジョニングするために使用できるオープンソースのソフトウェア開発フレームワークです。AWS CDK により、TypeScript、Python、Java、.NET を使用してアプリケーションインフラストラクチャをモデル化できます。AWS CDK は CloudFormation をバックグラウンドで使用し、安全で繰り返し可能な方法でリソースをプロビジョニングします。
  - AWS クラウドコントロール API では、開発者がクラウドインフラストラクチャを簡単に、一貫した方法で管理できるように、作成、読み取り、更新、削除、一覧表示 (CRUDL) API の共通セットが導入されています。Cloud Control API の共通 API を使用すると、開発者は AWS およびサードパーティーサービスのライフサイクルを均一に管理できます。
- ユーザーへの影響を最小限に抑えるデプロイパターンを実装します。
  - カナリアデプロイ:
    - API Gateway の Canary リリースデプロイの設定
    - AWS App Mesh を使用して、Amazon ECS のカナリアデプロイでパイプラインを作成する
  - ブルー/グリーンデプロイ: AWS ホワイトペーパーの「ブルー/グリーンデプロイ」では、ブルー/グリーンデプロイ戦略を実装する手法の例について説明しています。
- 構成または状態のドリフトを検出します。詳細については、「スタックとリソースに対するアンマネージド型構成変更の検出」を参照してください。

リソース

関連するベストプラクティス:

- REL08-BP05 自動化を使用して変更をデプロイする

関連ドキュメント:

- Automating safe, hands-off deployments
- Leveraging AWS CloudFormation to create an immutable infrastructure at Nubank
- Infrastructure as Code

- Implementing an alarm to automatically detect drift in AWS CloudFormation stacks

関連動画:

- AWS re:Invent 2020: Reliability, consistency, and confidence through immutability

REL08-BP05 自動化を使用して変更をデプロイする

デプロイとパッチ適用を自動化することで、悪影響を排除します。

本稼働システムに変更を加えることは、多くの組織にとって最大級のリスクの 1 つです。当社は、ソフトウェアで対処するビジネス上の問題と同じくらい、デプロイを最優先の課題であると考えています。これは今日、変更のテストとデプロイ、容量の追加と削除、データの移行など、実運用のあらゆる場所における自動化の導入を意味します。

期待される成果: 広範な本稼働前テスト、自動ロールバック、ずらされた本稼働デプロイを使用して、自動デプロイの安全性をリリースプロセスに構築します。この自動化により、デプロイの失敗による本番環境への潜在的な影響が最小限に抑えられ、開発者は本番環境へのデプロイを積極的に監視する必要がなくなります。

一般的なアンチパターン:

- 手動で変更を行う。
- 手動の緊急ワークフローにより、自動化のステップを省略する。
- スケジュールを短縮するために、確立された計画やプロセスを無視する。
- ベイク時間を考慮せずに、急激なフォローオンデプロイを実行する。

このベストプラクティスを活用するメリット: 自動化を使用してすべての変更をデプロイすることで、人為的ミスが発生する可能性を排除し、本番環境を変更する前にテストする機能を提供します。本番環境の稼働前にこのプロセスを実行することで、計画が完了していることを確認できます。さらに、リリースプロセスに自動ロールバックを組み込むことで、本番環境の問題を特定し、ワークロードを以前の動作状態に戻すことができます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

デプロイパイプラインを自動化します。デプロイパイプラインを使用すると、自動テストおよび異常の検出を呼び出せるようになります。また、本番環境へのデプロイを行う前の特定のステップでパイプラインを休止したり、変更を自動的にロールバックしたりできます。このために不可欠な部分は、継続的インテグレーションと継続的デリバリー/デプロイ (CI/CD) 文化の導入です。これにより、コミットまたはコードの変更は、ビルドおよびテスト段階から本番環境へのデプロイまで、さまざまな自動化されたステージゲートを通過します。

運用上の最も難しい手順には人間を関与させることが一般通念で推奨されていますが、最も難しい手順については、まさにこの理由から自動化を推奨します。

実装手順

デプロイを自動化して手動操作をなくすには、以下の手順に従います。

- コードを安全に保存するためのコードリポジトリを設定する: Git などの一般的なテクノロジーに基づくホスト型ソースコード管理システムを使用して、ソースコードと Infrastructure as code (IaC) 構成を保存します。
- ソースコードをコンパイルし、テストを実行して、デプロイアーティファクトを作成するように継続的統合サービスを設定する: この目的のためにビルドプロジェクトを設定するには、「コンソールを使用した AWS CodeBuild の開始方法」を参照してください。
- アプリケーションのデプロイを自動化し、エラーが発生しやすい手動デプロイに依存せずにアプリケーションの更新の複雑さを処理するデプロイサービスを設定する: AWS CodeDeploy は、Amazon EC2、AWS Fargate、AWS Lambda、オンプレミスサーバーなどのさまざまなコンピューティングサービスへのソフトウェアデプロイを自動化します。これらのステップを設定するには、「CodeDeploy の開始方法」を参照してください。
- リリースパイプラインを自動化する継続的な配信サービスを設定して、アプリケーションとインフラストラクチャの更新をより迅速かつ確実にする: リリースパイプラインの自動化に役立つ AWS CodePipeline の使用を検討してください。詳細については、「CodePipeline チュートリアル」を参照してください。

リソース

関連するベストプラクティス:

- OPS05-BP04 構築およびデプロイ管理システムを使用する
- OPS05-BP10 統合とデプロイを完全自動化する

- OPS06-BP02 デプロイをテストする
- OPS06-BP04 テストとロールバックを自動化する

関連ドキュメント:

- AWS CodePipeline を利用したネストされた AWS CloudFormation スタックの継続的デリバリー
- APN パートナー: 自動化されたデプロイソリューションの作成を支援できるパートナー
- AWS Marketplace: products that can be used to automate your deployments
- Webhook を使用してチャットメッセージを自動化する
- Amazon Builders' Library: デプロイ時におけるロールバックの安全性の確保
- Amazon Builders' Library: 継続的デリバリーによる高速化
- とはAWS CodePipeline
- What is CodeDeploy?
- AWS Systems Manager Patch Manager
- Amazon SESとは
- Amazon Simple Notification Service とは

関連動画:

- AWS Summit 2019: AWS における CI/CD

障害管理

> 障害は発生するものであり、最終的にはすべてが時間の経過とともにフェイルオーバーします。つまり、ルーターからハードディスクまで、TCP パケットを破壊するオペレーティングシステムからメモリユニットまで、そして一時的なエラーから永続的な障害まで、どれもが対象となるのです。これは、最高品質のハードウェアを使用しているか、最低料金のコンポーネントを使用しているかにかかわらず、当たり前のことです - Werner Vogels、CTO - Amazon.com

低レベルのハードウェアコンポーネントの障害は、オンプレミスのデータセンターで毎日対処する必要がある問題です。ただし、クラウドでは、お客様はこれらのタイプの障害のほとんどから保護されるはずです。例えば、Amazon EBS ボリュームは、特定のアベイラビリティーゾーンに配置され、単一のコンポーネントに障害が発生したときに保護できるように、自動的にレプリケートされます。すべての EBS ボリュームは、99.999% の可用性を実現するように設計されています。Amazon S3 オブジェクトは、最低 3 つのアベイラビリティーゾーンに保存され、年間 99.999999999% のオブジェクト耐久性を実現しています。クラウドプロバイダーに関係なく、障害がワークロードに影響を与える可能性があります。したがって、ワークロードの信頼性を確保する必要がある場合は、回復力を持たせる手順を実行しなければなりません。

ここで説明するベストプラクティスを適用するための前提条件として、ワークロードを設計、実装、および運用する担当者がビジネス目標とこれを達成するための信頼性目標を確実に把握しているようにする必要があります。その担当者は、信頼性要件を認識し、トレーニングを受ける必要があります。

以下のセクションでは、障害を管理してワークロードに影響を与えるのを防ぐためのベストプラクティスについて説明します。

トピック

- データのバックアップ方法
- 障害部分を切り離してワークロードを保護する
- コンポーネントの障害に耐えられるようにワークロードを設計する
- テストの信頼性
- ディザスタリカバリ (DR) を計画する

データのバックアップ方法

目標復旧時間 (RTO) と目標復旧時点 (RPO) の要件を満たすように、データ、アプリケーション、設定をバックアップします。

ベストプラクティス

- REL09-BP01 バックアップが必要なすべてのデータを特定してバックアップする、またはソースからデータを再現する
- REL09-BP02 バックアップを保護し、暗号化する
- REL09-BP03 データバックアップを自動的に実行する
- REL09-BP04 データの定期的な復旧を行ってバックアップの完全性とプロセスを確認する

REL09-BP01 バックアップが必要なすべてのデータを特定してバックアップする、またはソースからデータを再現する

ワークロードが使用するデータサービスとリソースのバックアップ機能を理解し、使用します。ほとんどのサービスは、ワークロードデータをバックアップする機能を提供します。

期待される成果: データソースが識別され、重要性に基づいて分類されます。次に、RPO に基づいてデータ復旧戦略を確立します。この戦略には、これらのデータソースをバックアップするか、他のソースからデータを再生成する能力を持つことが含まれます。データを喪失した場合は、実装された戦略によって、定義された RPO および RTO 内でデータを復旧または再生成できます。

クラウド成熟フェーズ: 基礎

一般的なアンチパターン:

- ワークロードのすべてのデータソースとそれらの重要性を認識していない。
- 重要なデータソースのバックアップを取っていない。
- 重要性を評価基準として使用せずに、一部のデータソースのみのバックアップを取る。
- RPO が定義されていないか、バックアップの頻度が RPO を満たしていない。
- バックアップが必要かどうか、またはデータを他のソースから再生成できるかどうかを評価していない。

このベストプラクティスを活用するメリット: バックアップが必要な場所を特定し、バックアップを作成するメカニズムを実装するか、外部ソースからデータを再生成できるようにすることで、停止時にデータを復元し、復旧する能力が高まります。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

すべての AWS データストアは、バックアップ機能を備えています。Amazon RDS や Amazon DynamoDB などのサービスは、ポイントインタイムリカバリ (PITR) を有効にする自動バックアップを追加でサポートします。これにより、現在時刻の 5 分前までの任意の時刻に、バックアップを復元することができます。多くの AWS のサービスでは、バックアップを別の AWS リージョンにコピーできます。AWS Backup は、AWS のサービス間でデータ保護を一元化および自動化する機能を提供するツールです。AWS Elastic Disaster Recovery では、完全なサーバーワークロードをコピーし、オンプレミス、クロス AZ、またはクロスリージョンの継続的なデータ保護を維持し、目標復旧時点 (RPO) を秒単位で測定できます。

Amazon S3 をセルフマネージドデータソースおよび AWS マネージドデータソースのバックアップ先として使用できます。Amazon EBS、Amazon RDS、Amazon DynamoDB などの AWS サービスには、バックアップを作成する機能が組み込まれています。サードパーティーのバックアップソフトウェアも使用できます。

AWS Storage Gateway または AWS DataSync を使用して、オンプレミスのデータを AWS クラウドにバックアップできます。Amazon S3 バケットを使用して、このデータを AWS に保存できます。Amazon S3 は、Amazon Glacier や Amazon Glacier Deep Archive などの複数のストレージ階層を提供し、データストレージのコストを削減します。

他のソースからデータを再生成することによって、データリカバリのニーズを満たすこともできます。例えば、Amazon ElastiCache レプリカノードまたは Amazon RDS リードレプリカを使用して、プライマリを喪失した場合にデータを再現できます。このようなソースを使用して目標復旧時点 (RPO) と目標復旧時間 (RTO) を満たせる場合は、バックアップが不要な可能性があります。別の例として、Amazon EMR を使用している場合、Amazon S3 から Amazon EMR にデータを再現できる限り、HDFS データストアをバックアップする必要がない可能性があります。

バックアップ戦略を選択するときは、データの復旧にかかる時間を考慮してください。データの復旧に必要な時間は、バックアップの種類 (バックアップ戦略の場合) やデータ再生成メカニズムの複雑さに応じて異なります。この時間は、ワークロードの RTO 以内である必要があります。

実装手順

1. ワークロードのすべてのデータソースを特定する。データは、データベース、ボリューム、ファイルシステム、ログ記録システム、オブジェクトストレージなどの多数のリソースに保管できます。「リソース」セクションを参照して、データが保管されているさまざまな AWS のサービスに関する関連ドキュメントと、これらのサービスが提供するバックアップ機能を確認してください。
2. 重要性に基づいてデータソースを分類する。データセットごとにワークロードに対する重要度が異なるため、回復力の要件も異なります。例えば、一部のデータは重要度が高く、ゼロに近い RPO を必要とするかもしれませんが、その他のデータは重要度が低く、より高い RPO や部分的なデータ損失に耐えられるかもしれません。同様に、データセットごとに RTO の要件も異なります。
3. AWS またはサードパーティーのサービスを使用して、データのバックアップを作成します。AWS Backup は、AWS でさまざまなデータソースのバックアップを作成できるマネージドサービスです。AWS Elastic Disaster Recovery は、AWS リージョンへの 1 秒未満の自動データ複製を処理します。AWS サービスのほとんどは、バックアップを作成する機能をネイティブで備えています。AWS Marketplace には、これらの機能を提供する多数のソリューションも用意されています。以下に記載されている「リソース」を参照して、さまざまな AWS のサービスからデータバックアップを作成する方法に関する情報を確認してください。
4. バックアップしないデータにデータ再生成メカニズムを確立する。さまざまな理由から、他のソースから再現できるデータはバックアップしないという選択をすることもあるでしょう。バックアップの保管にコストがかかるため、バックアップを作成するよりも、必要なときにソースからデータを再現したほうが安いという状況もあるかもしれません。別の例は、バックアップからの復元にかかる時間が、ソースからデータを再現するよりも長く、結果として RTO に反する場合です。このような状況では、トレードオフを考慮して、データ復旧が必要なときにこれらのソースからデータを再現する方法について、十分に定義されたプロセスを確立してください。例えば、データの分析を行うために、Amazon S3 からデータウェアハウス (Amazon Redshift など)、または MapReduce クラスター (Amazon EMR など) にデータをロードしてある場合、これは他のソースから再現できるデータの例にあてはまるかもしれません。これらの分析結果がどこかに保管されているか再現可能である限り、データウェアハウスまたは MapReduce クラスターで発生した障害によって、データが失われることはありません。ソースから再現できる例としては他にも、キャッシュ (Amazon ElastiCache など) や RDS リードレプリカなどが挙げられます。
5. データをバックアップするサイクルを確立する。データソースのバックアップ作成は定期的なプロセスであり、頻度は RPO に依存します。

実装計画に必要な工数レベル: 中

リソース

関連するベストプラクティス:

REL13-BP01 ダウンタイムやデータ消失に関する復旧目標を定義する

REL13-BP02 復旧目標を満たすため、定義された復旧戦略を使用する

関連ドキュメント:

- とはAWS Backup
- AWS DataSync とは
- ボリュームゲートウェイとは
- APN パートナー: バックアップを支援できるパートナー
- AWS Marketplace: バックアップに活用できる製品
- Amazon EBS スナップショット
- Amazon EFS のバックアップ
- Amazon FSx for Windows File Server のバックアップ
- ElastiCache for Redis のバックアップと復元
- Neptune での DB クラスタースナップショットの作成
- DB スナップショットの作成
- スケジュールに従ってトリガーする EventBridge ルールの作成
- Amazon S3 によるクロスリージョンレプリケーション
- EFS-to-EFS AWS Backup
- Amazon S3 へのログデータのエクスポート
- オブジェクトのライフサイクル管理
- DynamoDB のオンデマンドバックアップと復元
- DynamoDB のポイントインタイムリカバリ
- Amazon OpenSearch Service でのインデックススナップショットの操作
- AWS Elastic Disaster Recovery とは

関連動画:

- AWS re:Invent 2021 - AWS によるバックアップ、ディザスタリカバリ、ランサムウェア保護
- AWS Backup デモ: クロスアカウントおよびクロスリージョンバックアップ

- AWS re:Invent 2019: AWS Backup の詳細と Rackspaceについて (STG341)

REL09-BP02 バックアップを保護し、暗号化する

認証と承認を使用して、バックアップへのアクセスを制御し、検出します。暗号化によりバックアップのデータ保全性が損なわれることを防止、検出します。

セキュリティコントロールを実装して、バックアップデータへの不正アクセスを防止します。バックアップを暗号化して、データの機密性と整合性を保護します。

一般的なアンチパターン:

- データに対するのと同一の、バックアップおよび復元オートメーションへのアクセスを設定する。
- バックアップを暗号化しない。
- 削除や改ざんから保護するためのイミュータビリティを実装していない。
- 本稼働システムとバックアップシステムに同じセキュリティドメインを使用する。
- 定期的なテストでバックアップの整合性を検証していない。

このベストプラクティスを活用するメリット:

- バックアップを保護することで、データの改ざんを防止し、データの暗号化により、誤って公開されたデータへのアクセスが防止されます。
- バックアップインフラストラクチャをターゲットとするランサムウェアやその他のサイバー脅威に対する保護を強化しました。
- 検証済みの復旧プロセスにより、サイバーインシデント後の復旧時間が短縮されました。
- セキュリティインシデント中のビジネス継続性機能を改善しました。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

AWS Identity and Access Management (IAM) などの認証と承認を使用して、バックアップへのアクセスを制御、検出します。暗号化によりバックアップのデータ保全性が損なわれることを防止、検出します。

Amazon S3 は、保管中のデータを暗号化するための方法をいくつかサポートしています。Amazon S3 はサーバー側の暗号化を使用して、オブジェクトを暗号化されていないデータとして受け入れて

から、保存時に暗号化します。クライアント側の暗号化を使用すると、ワークロードアプリケーションはデータを Amazon S3 に送信する前に暗号化することに対して責任を負います。どちらの方法でも、AWS Key Management Service (AWS KMS) を使ってデータキーを作成して保存することもできます。また、自分でキーを用意し、そのキーを管理することもできます。AWS KMS を使用すると、IAM を使用してポリシーを設定し、データキーと復号化されたデータにアクセスできるユーザーとアクセスできないユーザーにわけることができます。

Amazon RDS では、データベースの暗号化を選択すると、バックアップも暗号化されます。DynamoDB バックアップは常に暗号化されます。AWS Elastic Disaster Recovery を使用すると、転送中および保管中のすべてのデータが暗号化されます。Elastic Disaster Recovery を使用すると、デフォルトの Amazon EBS 暗号化ボリューム暗号化キーまたはカスタムのカスタマーマネージドキーのいずれかを使用して、保管中のデータを暗号化できます。

サイバーレジリエンスに関する考慮事項

サイバー脅威に対するバックアップセキュリティを強化するには、暗号化に加えて以下の追加のコントロールを実装することを検討してください。

- AWS Backup Vault Lock または Amazon S3 Object Lock を使用してイミュータビリティを実装し、保持期間中にバックアップデータが変更または削除されるのを防ぎ、ランサムウェアや悪意のある削除から保護します。
- 重要なシステムの AWS Backup 論理エアギャップボールトを使用して、本番稼働環境とバックアップ環境の間に論理的な分離を確立し、両方の環境の侵害を同時に防止するのに役立つ分離を作成します。
- AWS Backup 復元テストを使用してバックアップの整合性を定期的に検証し、バックアップが破損しておらず、サイバーインシデント後に正常に復元できることを確認します。
- マルチパーティー承認を使用して重要な復旧オペレーションに対する AWS Backup マルチパーティー承認を実装し、複数の指定された承認者からの認可を要求することで、許可されていない復旧や悪意のある復旧の試みを防止します。

実装手順

1. 各データストアで暗号化を使用します。ソースデータが暗号化されている場合、バックアップも暗号化されます。
   - Amazon RDS で暗号化を使用します。RDS インスタンスの作成時に、AWS Key Management Service を使用して、保管時の暗号化を設定できます。

- Amazon EBS ボリュームを暗号化します。デフォルトの暗号化を設定するか、ボリュームの作成時に一意のキーを指定できます。
- 必要な Amazon DynamoDB 暗号化を使用します。DynamoDB は、保管中のデータをすべて暗号化します。AWS 所有の AWS KMS キーを使用するか、AWS マネージド KMS キーを使用して、アカウントに保存されるキーを指定できます。
- Amazon EFS に保存されているデータを暗号化します。ファイルシステムを作成するときに暗号化を設定します。
- 送信元と送信先のリージョンで暗号化を設定します。KMS に保存されているキーを使用して Amazon S3 で保管時の暗号化を設定できますが、キーはリージョン固有です。レプリケーションを設定するときに、送信先キーを指定できます。
- デフォルトまたはカスタムの Elastic Disaster Recovery 用 Amazon EBS 暗号化を使用するかどうかを選択します。このオプションでは、ステージングエリアのサブネットディスクとレプリケートしたディスク上に保管中のレプリケートされたデータを暗号化します。

2. バックアップにアクセスするための最小特権のアクセス許可を実装します。セキュリティのベストプラクティスに従って、バックアップ、スナップショット、およびレプリカへのアクセスを制限します。
3. 重要なバックアップのイミュータビリティを設定します。重要なデータについては、AWS Backup Vault Lock または S3 Object Lock を実装して、指定された保持期間中の削除や変更を防止します。実装の詳細については、「AWS Backup Vault Lock」を参照してください。
4. バックアップ環境の論理的な分離を作成します。サイバー脅威からの保護を強化する必要がある重要なシステムには、AWS Backup 論理エアギャップボールトを実装します。実装のガイダンスについては、「Building cyber resiliency with AWS Backup logically air-gapped vault」を参照してください。
5. バックアップ検証プロセスを実装します。AWS Backup 復元テストを設定して、バックアップが破損しておらず、サイバーインシデント後に正常に復元できることを定期的に確認します。詳細については、「Validate recovery readiness with AWS Backup restore testing」を参照してください。
6. 機密性の高い復旧オペレーションのマルチパーティー承認を設定します。重要なシステムでは、復旧を続行する前に、複数の指定された承認者からの認可を要求する AWS Backup マルチパーティー承認を実装します。実装の詳細については、「Improve recovery resilience with AWS Backup support for Multi-party approval」を参照してください。

リソース

関連ドキュメント:

- AWS Marketplace: バックアップに活用できる製品
- Amazon EBS 暗号化
- Amazon S3: 暗号化によるデータの保護
- CRR 追加設定: AWS KMS に保存された暗号化キーを使用したサーバー側の暗号化 (SSE) で作成されたオブジェクトをレプリケートする
- 保管時の DynamoDB 暗号化
- Amazon RDS リソースを暗号化する
- Amazon EFS のデータとメタデータの暗号化
- でのバックアップの暗号化AWS
- 暗号化テーブルの管理
- セキュリティの柱 - AWS Well-Architected フレームワーク
- とは AWS Elastic Disaster Recovery
- FSISEC11: How are you protecting against ransomware?
- Ransomware Risk Management on AWS Using the NIST Cyber Security Framework
- Building cyber resiliency with AWS Backup logically air-gapped vault
- Validate recovery readiness with AWS Backup restore testing
- Improve recovery resilience with AWS Backup support for Multi-party approval

関連する例:

- Implementing Bi-Directional Cross-Region Replication (CRR) for Amazon S3

REL09-BP03 データバックアップを自動的に実行する

目標復旧時点 (RPO) によって通知される定期的なスケジュール、またはデータセット内の変更に基づいて、バックアップが自動的に行われるように設定します。データ損失の少ない重要なデータセットは頻繁に自動バックアップする必要があります。多少の損失は許容できる重要度の低いデータは、バックアップの頻度を少なくすることができます。

期待される成果: 一定の周期でデータソースのバックアップを作成する自動化されたプロセス。

一般的なアンチパターン:

- バックアップを手動で実行する。
- バックアップ機能があるが、自動化にバックアップが含まれていないリソースを使用する。

このベストプラクティスを活用するメリット: バックアップを自動化することで、バックアップが RPO に基づいて定期的に作成され、作成されない場合はアラートが送信されます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

AWS Backup を使用して、AWS データソースの自動データバックアップを作成できます。Amazon RDS インスタンスは 5 分ごとにほぼ連続的にバックアップでき、Amazon S3 オブジェクトは 15 分ごとにほぼ連続的にバックアップできます。これにより、バックアップ履歴内の特定の時点へのポイントインタイムリカバリ (PITR) が可能になります。Amazon EBS ボリューム、Amazon DynamoDB テーブル、Amazon FSx ファイルシステムなど、その他の AWS データソースについては、AWS Backup は 1 時間ごとの頻度で自動バックアップを実行できます。これらのサービスはネイティブバックアップ機能も提供します。ポイントインタイムリカバリによる自動バックアップを提供する AWS サービスには、Amazon DynamoDB、Amazon RDS、Amazon Keyspaces (Apache Cassandra 向け) などがあります。これらは、バックアップ履歴内の特定の時点に復元できます。その他の AWS データストレージサービスのほとんどが、1 時間ごとの定期バックアップをスケジュールする機能を提供しています。

Amazon RDS および Amazon DynamoDB は、ポイントインタイムリカバリと継続的なバックアップを提供します。Amazon S3 バージョニングは、有効化すると自動的に行われます。Amazon EBS スナップショットの作成、保持、削除を自動化するには、Amazon Data Lifecycle Manager を使うことができます。また、Amazon EBS-backed Amazon マシンイメージ (AMI) とその基盤となる Amazon EBS スナップショットの作成、コピー、廃止、および登録解除も自動化できます。

AWS Elastic Disaster Recovery は、ソース環境 (オンプレミスまたは AWS) からターゲットの復旧リージョンへの継続的なブロックレベルのレプリケーションを提供します。ポイントインタイム Amazon EBS スナップショットは、このサービスが自動的に作成し、管理します。

バックアップの自動化と履歴を一元的に確認できるようにするために、AWS Backup は完全マネージド型の、ポリシーベースのバックアップソリューションを提供します。AWS Storage Gateway を

使用して、クラウド内およびオンプレミスの複数の AWS のサービスにわたってデータのバックアップを一元化および自動化します。

バージョニングに加えて、Amazon S3 はレプリケーション機能も備えています。S3 バケット全体を同じまたは異なる AWS リージョンにある別のバケットに自動的にレプリケートできます。

実装手順

1. 手動でバックアップされているデータソースを特定します。詳細については、「REL09-BP01 バックアップが必要なすべてのデータを特定してバックアップする、またはソースからデータを再現する」を参照してください。
2. ワークロードの RPO を決定します。詳細については、「REL13-BP01 ダウンタイムやデータ消失に関する復旧目標を定義する」を参照してください。
3. 自動バックアップまたはマネージドサービスを使用します。AWS Backup はフルマネージド型のバックアップサービスであり、AWS のサービス、クラウド内、およびオンプレミス間で簡単に一元化およびデータ保護を自動化できます。AWS Backup のバックアッププランを使用して、バックアップするリソースと、これらのバックアップを作成する頻度を定義するルールを作成します。この頻度は、ステップ 2 で確立した RPO によって通知される必要があります。AWS Backup を使用して自動バックアップを作成する方法の実践的なガイダンスについては、「データのバックアップと復元のテスト」を参照してください。データを保存するほとんどの AWS サービスでは、バックアップ機能がネイティブで提供されています。例えば、RDS は、ポイントインタイムリカバリ (PITR) 付きの自動バックアップに利用できます。
4. オンプレミスデータソースおよびメッセージキューなど、自動バックアップソリューションやマネージドサービスによってサポートされていないデータソースに関しては、信頼できるサードパーティーソリューションを使用して自動バックアップを作成することを検討してください。または、AWS CLI または SDK を使用してこれを行うオートメーションを作成することができます。AWS Lambda 関数または AWS Step Functions を使用して、データバックアップの作成にかかわるロジックを定義し、Amazon EventBridge を使用して RPO に基づく頻度で呼び出せます。

実装計画に必要な工数レベル: 低

リソース

関連ドキュメント:

- APN パートナー: バックアップを支援できるパートナー
- AWS Marketplace: バックアップに活用できる製品

- スケジュールに従ってトリガーする EventBridge ルールの作成
- AWS Backup とは
- AWS Step Functions とは
- AWS Elastic Disaster Recovery とは

関連動画:

- AWS re:Invent 2019: AWS Backup の詳細と Rackspace について (STG341)

REL09-BP04 データの定期的な復旧を行ってバックアップの完全性とプロセスを確認する

復旧テストを実行して、バックアッププロセスの実装が目標復旧時間 (RTO) と目標復旧時点 (RPO) を満たしていることを検証します。

期待される成果: バックアップからのデータを、十分に定義されたメカニズムを使用して定期的に復旧することで、ワークロードについて確立された目標復旧時間 (RTO) 内での復旧が可能であることを確認できます。バックアップからの復元により、オリジナルデータを含むリソースになり、破損したりアクセス不能になっていたりするデータがなく、目標復旧時点 (RPO) 内のデータ損失であることを確認します。

一般的なアンチパターン:

- バックアップを復元しますが、復元が使用可能であることを確認するためのデータのクエリや取得は行いません。
- バックアップが存在することを前提とする。
- システムのバックアップが完全に動作可能であり、そこからデータを復旧できることを前提とする。
- バックアップからデータを復元または復旧する時間がワークロードの RTO の範囲内であることを前提とする。
- バックアップに含まれるデータがワークロードの RPO の範囲内であることを前提とする。
- ランブックを使用せずに、または確立された自動手順の外部で、必要に応じて復元する。

このベストプラクティスを活用するメリット: バックアップの復旧をテストすることで、データの紛失や破損を心配せずに、必要なときにデータを復元できること、ワークロードの RTO 内で復元と復

旧が可能であること、ならびにデータ損失がワークロードの RPO の範囲内であることを確認できます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

バックアップおよび復元機能をテストすることで、停止時にこれらのアクションを実行できるという安心感が得られます。定期的にバックアップを新しい場所に復元して、テストを実行し、データの完全性を確認します。実行する必要がある一般的なテストには、すべてのデータが利用可能かどうか、破損していないかどうか、アクセス可能かどうか、データ損失がワークロードの RPO 内に収まるかどうかを確認することなどがあります。そのようなテストは、復旧メカニズムが十分に高速であり、ワークロードの RTO に対応できることを確認するのにも役立ちます。

AWS を使用して、テスト環境を構築し、そこにバックアップを復元して RTO および RPO が機能するかを評価し、データコンテンツと完全性のテストを実行できます。

さらに、Amazon RDS および Amazon DynamoDB はポイントインタイムリカバリ (PITR) を許可します。継続的バックアップを使用すると、データセットを指定された日時の状態に復元できます。

すべてのデータが使用可能であり、破損しておらず、アクセス可能であり、データ損失がワークロードの RPO の範囲内であることを確認します。そのようなテストは、復旧メカニズムが十分に高速であり、ワークロードの RTO に対応できることを確認するのにも役立ちます。

AWS Elastic Disaster Recovery は、Amazon EBS ボリュームの継続的なポイントインタイムリカバリのスナップショットを提供します。ソースサーバーがレプリケートされると、設定されたポリシーに基づいて時間の経過とともにポイントインタイム状態が記録されます。Elastic Disaster Recovery は、トラフィックをリダイレクトせずにテストおよびドリル目的でインスタンスを起動して、スナップショットの整合性を検証するのに役立ちます。

実装手順

1. バックアップ中のデータソースと、これらのバックアップの保存場所を特定します。実装のガイダンスについては、「REL09-BP01 バックアップが必要なすべてのデータを特定してバックアップする、またはソースからデータを再現する」を参照してください。
2. 各データソースのデータ検証の基準を確立します。データのタイプが異なると、プロパティも異なり、異なる検証メカニズムが必要になることがあります。本番環境での使用を決定する前に、このデータを検証する方法を考慮してください。データを検証するための一般的な方法としては、データタイプ、フォーマット、チェックサム、サイズ、またはこれらの組み合わせなど、データとバックアッププロパティをカスタム検証ロジックで使用することです。例えば、復元さ

れたリソースと、バックアップが作成された時点でのデータソースの間でチェックサム値を比較します。

3. データの重要性に基づいた RTO と RPO を確立します。実装のガイダンスについては、「REL13-BP01 ダウンタイムやデータ消失に関する復旧目標を定義する」を参照してください。
4. データ復旧機能を評価します。バックアップおよび復元戦略をレビューして、RTO および RPO を満たせるかどうかを理解し、必要に応じて戦略を調整します。AWS Resilience Hub を使用して、ワークロードのアセスメントを実行できます。アセスメントは、回復力ポリシーに対してアプリケーション設定を評価し、RTO および RPO 目標を満たすことができるかどうかを報告します。
5. 本番環境で確立済みのデータ復元プロセスを使用して、テスト復元を行います。プロセスは、オリジナルデータソースのバックアップ方法、バックアップそのもののフォーマットとストレージ場所、またはデータが他のソースから再生成されるかどうかによって異なります。例えば、AWS Backup などのマネージドサービスを使用している場合、バックアップを新しいリソース に復元するだけで済みます。AWS Elastic Disaster Recovery を使用した場合、リカバリドリルを起動します。
6. データ検証のために以前に確立した基準に基づいて、復元されたリソースからのデータ復旧を検証します。復元され、復旧されたデータには、バックアップ時点で最新のレコードまたはアイテムが含まれていますか? このデータはワークロードの RPO の範囲内ですか?
7. 復元と復旧に必要な時間を測定し、確立された RTO と比較します。このプロセスは、ワークロードの RTO の範囲内ですか? 例えば、復元プロセスが開始されたときのタイムスタンプと復旧検証が完了したときのタイムスタンプを比較して、このプロセスの所要時間を計算します。すべての AWS API 呼び出しにはタイムスタンプが付けられており、AWS CloudTrail で情報を確認できます。この情報から復元プロセスが開始したときの詳細がわかりますが、検証が完了したときの終了タイムスタンプが検証ロジックによって記録される必要があります。自動プロセスを使用する場合、Amazon DynamoDB などのサービスを使用してこの情報を保存できます。さらに、多くの AWS のサービスは、特定のアクションが発生したときのタイムスタンプ付きの情報を提供するイベント履歴を備えています。AWS Backup 内では、バックアップおよび復元アクションはジョブと呼ばれ、ジョブにはメタデータの一部としてタイムスタンプ情報が含まれ、復元と復旧の所要時間の測定に使用できます。
8. 復元と復旧に必要な時間がワークロードについて確立された RTO を超えている場合は、関係者に通知します。このラボで示すように自動化を実装する場合、Amazon Simple Notification Service (Amazon SNS) などのサービスを使用して、E メールや SMS などのプッシュ通知を関係者に送信できます。これらのメッセージは、Amazon Chime、Slack、Microsoft Teams などのメッセージングアプリケーションに発行したり、AWS Systems Manager OpsCenter を使用してタスクを OpsItems として作成したりできます。

9. このプロセスを定期的に実行するように自動化します。例えば、AWS Lambda や AWS Step Functions の状態マシンなどのサービスを使用して、復元および復旧プロセスを自動化でき、Amazon EventBridge を使用して、下のアーキテクチャ図に示されているように、このオートメーションワークフローを定期的に呼び出すことができます。AWS Backup でデータリカバリの検証を自動化する方法を学びます。この Well-Architected ラボでは、いくつかのステップを自動化するための方法を解説します。

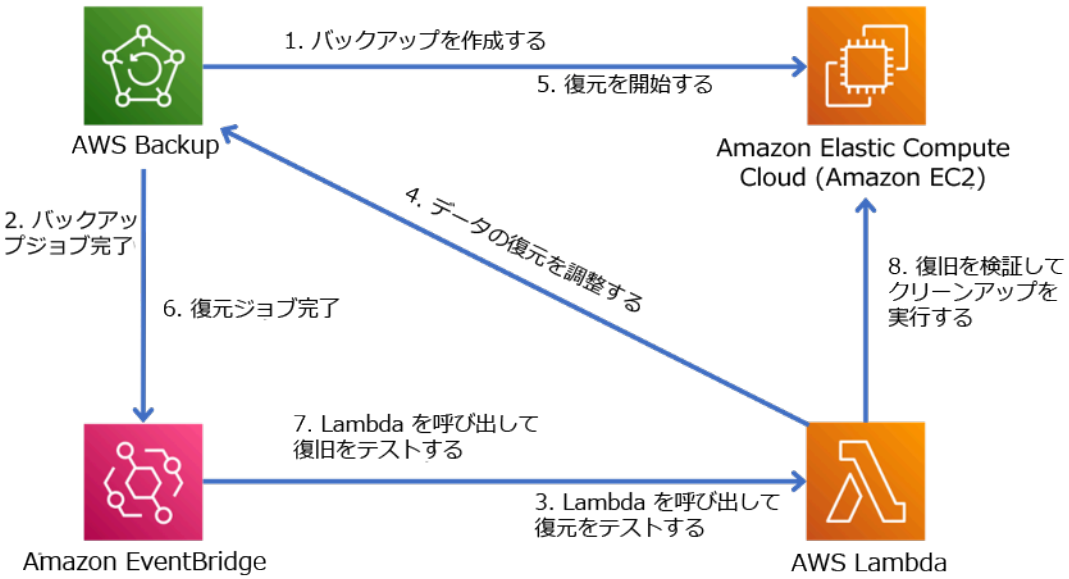


図 9: 自動化されたバックアップおよび復元プロセス

実装計画に必要な工数レベル: 検証基準の複雑さに応じて中から高

リソース

関連ドキュメント:

- AWS Backup でデータリカバリの検証を自動化する
- APN パートナー: バックアップを支援できるパートナー
- AWS Marketplace: バックアップに活用できる製品
- スケジュールに従ってトリガーする EventBridge ルールの作成
- DynamoDB のオンデマンドバックアップと復元
- AWS Backup とは
- AWS Step Functions とは
- AWS Elastic Disaster Recovery とは

- AWS Elastic Disaster Recovery

障害部分を切り離してワークロードを保護する

障害分離は、コンポーネントまたはシステムの障害の影響を定義された境界に制限します。適切な分離により、境界の外部のコンポーネントは障害の影響を受けません。複数の障害分離境界にわたってワークロードを実行すると、障害に対する回復力が高まる可能性があります。

ベストプラクティス

- REL10-BP01 複数の場所にワークロードをデプロイする
- REL10-BP02 単一のロケーションに制約されるコンポーネントのリカバリを自動化する
- REL10-BP03 バルクヘッドアーキテクチャを使用して影響範囲を制限する

REL10-BP01 複数の場所にワークロードをデプロイする

ワークロードのデータとリソースを複数のアベイラビリティーゾーンに分散するか、必要に応じて複数の AWS リージョンにまたがって分散します。

AWS のサービス設計の基本的な原則は、基盤となる物理インフラストラクチャを含む単一障害点を回避することです。AWS は、リージョンと呼ばれる複数の地理的ロケーションで、クラウドコンピューティングリソースとサービスをグローバルに提供します。各リージョンは物理的および論理的に独立しており、3 つ以上のアベイラビリティーゾーン (AZ) で構成されています。アベイラビリティーゾーンは地理的には互いに近接していますが、物理的には分離され、隔離されています。アベイラビリティーゾーンとリージョン間でワークロードを分散すると、火災、洪水、気象関連の災害、地震、人為的ミスなどの脅威のリスクが軽減されます。

ワークロードに適した高可用性を提供するロケーション戦略を作成します。

期待される成果: 本番稼働ワークロードは、耐障害性と高可用性を実現するために、複数のアベイラビリティーゾーン (AZ) またはリージョンに分散されます。

一般的なアンチパターン:

- 本番稼働ワークロードを 1 つのアベイラビリティーゾーンにのみ存在させる。
- マルチ AZ アーキテクチャでビジネス要件を満たせるときに、マルチリージョンアーキテクチャを実装する。
- デプロイまたはデータが非同期化され、設定ドリフトやデータのレプリケート不足が発生する。

- 回復力要件とマルチロケーション要件がアプリケーションコンポーネント間で異なる場合に、コンポーネント間の依存関係を考慮しない。

このベストプラクティスを活用するメリット:

- ワークロードは、電力や環境制御の障害、自然災害、アップストリームサービスの障害、AZ またはリージョン全体に影響を及ぼすネットワークの問題などのインシデントに対して、より耐性を持つようになります。
- Amazon EC2 インスタンスの広範なインベントリにアクセスし、特定の EC2 インスタンスタイプを起動するときに InsufficientCapacityExceptions (ICE) が発生する可能性を減らすことができます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

リージョンの少なくとも 2 つのアベイラビリティーゾーン (AZ) にすべての本番稼働ワークロードをデプロイして運用します。

複数のアベイラビリティーゾーンを使用する

アベイラビリティーゾーンは、火災、洪水、竜巻などのリスクによる相関障害を回避するために、物理的に互いに分離されたリソースホスティングの場所です。各アベイラビリティーゾーンには、ユーティリティ電源接続、バックアップ電源、機械サービス、ネットワーク接続などの独立した物理インフラストラクチャがあります。この配置により、これらのコンポーネントのいずれかの障害は、影響を受けるアベイラビリティーゾーンのみに制限されます。例えば、AZ 全体のインシデントにより、影響を受けるアベイラビリティーゾーンで EC2 インスタンスが使用できなくなった場合でも、他のアベイラビリティーゾーンのインスタンスは引き続き利用できます。

物理的に分離されているにもかかわらず、同じ AWS リージョン 内のアベイラビリティーゾーンは、高スループット、低レイテンシー (1 桁ミリ秒) のネットワークを提供できるほど近い位置にあります。ユーザーエクスペリエンスに大きな影響を与えることなく、ほとんどのワークロードのアベイラビリティーゾーン間でデータを同期的にレプリケートできます。つまり、アクティブ/アクティブまたはアクティブ/スタンバイの設定でリージョンのアベイラビリティーゾーンを使用できます。

ワークロードに関連するすべてのコンピューティングは、複数のアベイラビリティーゾーンに分散する必要があります。これには、Amazon EC2 インスタンス、AWS Fargate タスク、VPC にアタッチされた AWS Lambda 関数が含まれます。EC2 Auto Scaling、Amazon Elastic Container Service (ECS)、Amazon Elastic Kubernetes Service (EKS) などの AWS コンピューティングサービスは、ア

ベイラビリティーゾーン間でコンピューティングを起動および管理する方法を提供します。必要に応じて別のアベイラビリティーゾーンでコンピューティングを自動的に置き換えるように設定して、アベイラビリティーを維持します。利用可能なアベイラビリティーゾーンにトラフィックを転送するには、コンピューティングの前に、Application Load Balancer や Network Load Balancer などのロードバランサーを配置します。AWS Load Balancer は、アベイラビリティーゾーンに障害が発生した場合に、トラフィックを利用可能なインスタンスに再ルーティングできます。

また、ワークロードのデータをレプリケートし、複数のアベイラビリティーゾーンで使用できるようにする必要があります。Amazon S3、Amazon Elastic File Service (EFS)、Amazon Aurora、Amazon DynamoDB、Amazon Simple Queue Service (SQS)、Amazon Kinesis Data Streams などの一部の AWS マネージドデータサービスは、デフォルトで複数のアベイラビリティーゾーンにデータをレプリケートし、アベイラビリティーゾーンの障害に対して堅牢です。Amazon Relational Database Service (RDS)、Amazon Redshift、Amazon ElastiCache などの他の AWS マネージドデータサービスでは、マルチ AZ レプリケーションを有効にする必要があります。これらのサービスを有効にすると、アベイラビリティーゾーンの障害が自動的に検出され、利用可能なアベイラビリティーゾーンにリクエストがリダイレクトされ、復旧後に必要に応じて顧客の介入なしにデータの再レプリケートが行われます。使用する各 AWS マネージドデータサービスのユーザーガイドをよく読み、マルチ AZ 機能、動作、操作を理解してください。

Amazon Elastic Block Store (EBS) ボリュームや Amazon EC2 インスタンスストレージなどのセルフマネージドストレージを使用している場合は、マルチ AZ レプリケーションを自分で管理する必要があります。

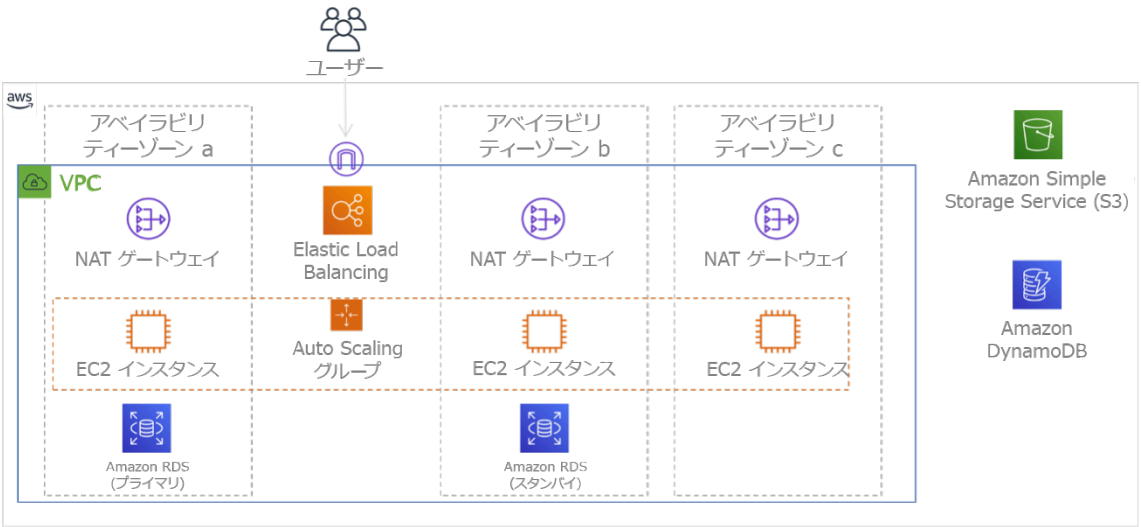


図 9: 3 つのアベイラビリティーゾーンにまたがってデプロイされる多階層アーキテクチャ。Amazon S3 と Amazon DynamoDB は常に自動的にマルチ AZ です。ELB も 3 つのゾーンすべてにデプロイされます。

複数の AWS リージョン を使用する

非常に高い耐障害性を必要とするワークロード (重要なインフラストラクチャ、医療関連のアプリケーション、厳格な顧客要件または義務付けられたアベイラビリティー要件を持つサービスなど) がある場合は、単一の AWS リージョン で提供できるものを超える追加の可用性が必要になることがあります。この場合、少なくとも 2 つの AWS リージョン にわたってワークロードをデプロイして運用する必要があります (データレジデンシー要件で許可されている場合)。

AWS リージョン は、世界中のさまざまな地理的地域と複数の大陸にあります。AWS リージョン は、アベイラビリティーゾーンのみよりもさらに物理的に分離および隔離されています。AWS サービスは、いくつかの例外を除き、この設計を利用して、異なるリージョン間で完全に独立して動作します (リージョンサービスとも呼ばれます)。AWS リージョンal サービスの障害は、別のリージョンのサービスに影響を与えないように設計されています。

複数のリージョンでワークロードを操作する場合は、追加の要件を考慮する必要があります。異なるリージョンのリソースは互いに独立しているため、各リージョンでワークロードのコンポーネントを複製する必要があります。これには、コンピューティングおよびデータサービスに加えて、VPC などの基盤インフラストラクチャが含まれます。

注: マルチリージョン設計を検討するときは、ワークロードが 1 つのリージョンで実行できることを確認します。あるリージョンのコンポーネントが別のリージョンのサービスまたはコンポーネントに依存するリージョン間に依存関係を作成すると、障害のリスクが高まり、信頼性体制が大幅に弱まる可能性があります。

マルチリージョンのデプロイを容易にし、一貫性を維持するために、AWS CloudFormation StackSets は AWS インフラストラクチャ全体を複数のリージョンにレプリケートできます。AWS CloudFormation は、設定ドリフトを検出し、リージョン内の AWS リソースが同期していないときに通知することもできます。多くの AWS サービスは、重要なワークロードアセットに対してマルチリージョンレプリケーションを提供します。例えば、EC2 Image Builder は、使用する各リージョンへのビルドのたびに EC2 マシンイメージ (AMI) を発行できます。Amazon Elastic Container Registry (ECR) は、コンテナイメージを選択したリージョンにレプリケートできます。

選択した各リージョンにもデータをレプリケートする必要があります。多くの AWS マネージドデータサービスは、Amazon S3、Amazon DynamoDB、Amazon RDS、Amazon Aurora、Amazon Redshift、Amazon Elasticache、Amazon EFS などのクロスリージョンレプリケーション機能を備えています。Amazon DynamoDB グローバルテーブルは、サポートされている任意のリージョンでの書き込みを受け入れ、他のすべての設定済みリージョン間でデータをレプリケートします。他のサービスでは、他のリージョンに読み取り専用レプリカが含まれているため、書き込みのプライマリリージョンを指定する必要があります。ワークロードが使用する AWS マネージドデータサービスご

とに、そのユーザーガイドとデベロッパーガイドを参照して、マルチリージョンの機能と制限を理解してください。書き込みの転送先、トランザクションの機能と制限、レプリケーションの実行方法、リージョン間の同期をモニタリングする方法に特に注意してください。

AWS には、リクエストトラフィックをリージョンデプロイに非常に柔軟にルーティングする機能もあります。例えば、Amazon Route 53 を使用して DNS レコードを設定し、ユーザーに最も近い利用可能なリージョンにトラフィックを誘導できます。または、DNS レコードをアクティブ/スタンバイ構成で構成し、1 つのリージョンをプライマリとして指定し、プライマリリージョンに異常が発生した場合にのみリージョンレプリカにフォールバックすることもできます。Route 53 ヘルスチェックを設定し、異常なエンドポイントを検出し、自動フェイルオーバーを実行し、さらに Amazon Application Recovery Controller (ARC) を使用して、必要に応じて手動でトラフィックを再ルーティングするための高可用性ルーティング制御を実現できます。

高可用性のために複数のリージョンで運用しないことを選択した場合でも、ディザスタリカバリ (DR) 戦略の一環として複数のリージョンを検討してください。可能であれば、ワークロードのインフラストラクチャコンポーネントとデータを、セカンダリリージョンのウォームスタンバイまたはパイロットライト構成でレプリケートします。この設計では、VPC、Auto Scaling グループ、コンテナオーケストレーター、その他のコンポーネントなどのプライマリリージョンからベースラインインフラストラクチャをレプリケートしますが、スタンバイリージョンの可変サイズのコンポーネント (EC2 インスタンスやデータベースレプリカの数など) を最小操作可能なサイズに設定します。また、プライマリリージョンからスタンバイリージョンへの継続的なデータレプリケーションも用意します。インシデントが発生した場合は、スタンバイリージョンのリソースをスケールアウトまたは拡張し、プライマリリージョンに昇格させることができます。

実装手順

1. ビジネス関係者やデータレジデンシーのエキスパートと協力して、リソースとデータをホストするためにどの AWS リージョン を使用できるかを決定します。
2. ビジネスおよび技術の関係者と協力してワークロードを評価し、その耐障害性ニーズがマルチ AZ アプローチ (単一の AWS リージョン) で満たされるかどうか、またはマルチリージョンアプローチ (複数のリージョンが許可されている場合) が必要かどうかを判断します。複数のリージョンを使用すると可用性が高まりますが、複雑さとコストが増加する可能性があります。評価する際には、次の要素を考慮してください。
   a. ビジネス目的と顧客要件: アベイラビリティーゾーンまたはリージョンでワークロードに影響を与えるインシデントが発生すると、どのくらいのダウンタイムが許容されますか？「REL13-BP01 ダウンタイムやデータ損失に関する復旧目標を定義する」で説明されているように、復旧ポイントの目的を評価します。

b. ディザスタリカバリ (DR) の要件: どのような潜在的な災害に対して保険をかけたいですか? 単一のアベイラビリティーゾーンからリージョン全体まで、さまざまな影響範囲でデータ損失や長期的に利用不可になる可能性を考慮してください。アベイラビリティーゾーン間でデータとリソースをレプリケートし、1 つのアベイラビリティーゾーンで持続的な障害が発生した場合は、別のアベイラビリティーゾーンでサービスを復旧できます。リージョン間でデータとリソースをレプリケートする場合、別のリージョンでサービスを復旧できます。

3. コンピューティングリソースを複数のアベイラビリティーゾーンにデプロイします。

a. VPC で、異なるアベイラビリティーゾーンに複数のサブネットを作成します。インシデント発生時でもワークロードを処理するために必要なリソースを収容できる大きさになるようにそれぞれを構成します。詳細については、「REL02-BP03 拡張性と可用性を考慮した IP サブネットの割り当てを確実に行う」を参照してください。

b. Amazon EC2 インスタンスを使用している場合は、EC2 Auto Scaling を使用してインスタンスを管理します。Auto Scaling グループの作成時に前のステップで選択したサブネットを指定します。

c. Amazon ECS または Amazon EKSの AWS Fargate コンピューティングを使用している場合は、ECS Service の作成、ECS タスクの起動、または EKS の Fargate プロファイルの作成の最初のステップで選択したサブネットを選択します。

d. VPC で実行する必要がある AWS Lambda 関数を使用している場合は、Lambda 関数の作成時に最初のステップで選択したサブネットを選択します。VPC 構成を持たない機能については、AWS Lambda が自動的に可用性を管理します。

e. ロードバランサーなどのトラフィックディレクターをコンピューティングリソースの前に配置します。クロスゾーン負荷分散が有効になっている場合、AWS Application Load Balancer と Network Load Balancer は、アベイラビリティーゾーンの障害が原因で EC2 インスタンスやコンテナなどのターゲットに到達できない場合にそれを検出し、正常なアベイラビリティーゾーンのターゲットにトラフィックを再ルーティングします。クロスゾーン負荷分散を無効にする場合は、Amazon Application Recovery Controller (ARC) を使用してゾーンシフト機能を提供します。サードパーティーのロードバランサーを使用している場合、または独自のロードバランサーを実装している場合は、異なるアベイラビリティーゾーンにまたがる複数のフロントエンドでロードバランサーを設定します。

4. ワークロードのデータを複数のアベイラビリティーゾーンにレプリケートします。

a. Amazon RDS、Amazon ElastiCache、Amazon FSx などの AWS マネージドデータサービスを使用する場合は、そのユーザーガイドを読んで、データのレプリケーションと耐障害性の機能を理解します。必要に応じてクロス AZ レプリケーションとフェイルオーバーを有効にします。

b. Amazon S3、Amazon EFS、Amazon FSx などの AWS マネージドストレージサービスを使用する場合は、高い耐久性が求められるデータに対してシングル AZ または 1 つのゾーン構成を使用しないでください。これらのサービスにはマルチ AZ 設定を使用します。各サービスのユーザーガイドを確認して、マルチ AZ レプリケーションがデフォルトで有効になっているか、有効にする必要があるかを判断します。

c. セルフマネージド型データベース、キュー、またはその他のストレージサービスを実行する場合は、アプリケーションの手順またはベストプラクティスに従って、マルチ AZ レプリケーションを調整します。アプリケーションのフェイルオーバー手順を理解します。

5. AZ の障害を検出し、正常なアベイラビリティーゾーンにトラフィックを再ルーティングするように DNS サービスを設定します。Amazon Route 53 を Elastic ロードバランサーと組み合わせて使用すると、自動的にこれを実行できます。Route 53 は、ヘルスチェックを使用して正常な IP アドレスのみを持つクエリに応答するフェイルオーバーレコードで設定することもできます。フェイルオーバーに使用される DNS レコードでは、レコードキャッシュが復旧を妨げるのを防ぐために、短い有効期間 (TTL) 値 (60 秒以下など) を指定します (Route 53 エイリアスレコードは適切な TTL を提供します)。

複数の AWS リージョン を使用する場合の追加ステップ

1. 選択したリージョン全体で、ワークロードで使用されるすべてのオペレーティングシステム (OS) とアプリケーションコードをレプリケートします。必要に応じて、Amazon EC2 Image Builder などのソリューションを使用して Amazon EC2 マシンイメージ (AMI) をレプリケートします。Amazon ECR クロスリージョンレプリケーションなどのソリューションを使用して、レジストリに保存されているコンテナイメージをレプリケートします。アプリケーションリソースの保存に使用される Amazon S3 バケットのリージョンレプリケーションを有効にします。
2. コンピューティングリソースと設定メタデータ (AWS Systems Manager Parameter Store に保存されているパラメータなど) を複数のリージョンにデプロイします。前のステップで説明したのと同じ手順を使用しますが、ワークロードに使用するリージョンごとに設定をレプリケートします。AWS CloudFormation などの Infrastructure as code (IaC) ソリューションを使用して、リージョン間で設定を均一に再現します。ディザスタリカバリにパイロットライト設定でセカンダリリージョンを使用している場合は、コンピューティングリソースの数を最小値に減らしてコストを削減し、それに応じて復旧までの時間を長くすることができます。
3. プライマリリージョンからセカンダリリージョンにデータをレプリケートします。

a. Amazon DynamoDB グローバルテーブルは、サポートされている任意のリージョンから書き込むことができるデータのグローバルレプリカを提供します。Amazon RDS、Amazon Aurora、Amazon Elasticache などの他の AWS マネージドデータサービスでは、プライマリ

(読み取り/書き込み) リージョンとレプリカ (読み取り専用) リージョンを指定します。リージョンレプリケーションの詳細については、各サービスのユーザーガイドとデベロッパーガイドを参照してください。

b. セルフマネージドデータベースを実行している場合は、アプリケーションの手順またはベストプラクティスに従って、マルチリージョンレプリケーションを調整します。アプリケーションのフェイルオーバー手順を理解します。

c. ワークロードで AWS EventBridge を使用している場合は、選択したイベントをプライマリリージョンからセカンダリリージョンに転送する必要がある場合があります。そのためには、セカンダリリージョンのイベントバスをプライマリリージョンの一致イベントのターゲットとして指定します。

4. リージョン間で同じ暗号化キーを使用するかどうか、またどの程度使用するかを検討します。セキュリティと使いやすさのバランスをとる一般的なアプローチは、リージョンローカルデータと認証にリージョンスコープキーを使用し、グローバルスコープキーを使用して、異なるリージョン間でレプリケートされるデータの暗号化を行うことです。AWS Key Management Service(KMS) は、リージョン間で共有されるキーを安全に分散および保護するためのマルチリージョンキーをサポートしています。

5. AWS Global Accelerator を検討して、正常なエンドポイントを含むリージョンにトラフィックを誘導することによって、アプリケーションの可用性を向上させます。

リソース

関連するベストプラクティス:

- REL02-BP03 拡張性と可用性を考慮した IP サブネットの割り当てを確実に行う
- REL11-BP05 静的安定性を使用してバイモーダル動作を防止する
- REL13-BP01 ダウンタイムやデータ損失に関する復旧目標を定義する

関連ドキュメント:

- AWS グローバルインフラストラクチャ
- ホワイトペーパー: AWS 障害分離境界
- Amazon EC2 Auto Scaling のレジリエンス
- Amazon EC2 Auto Scaling: 例: アベイラビリティーゾーン間でインスタンスを分散する
- EC2 Image Builder の仕組み
- Amazon ECS がタスクをコンテナインスタンスに配置する方法 (Fargate を含む)

- AWS Lambda での回復力
- Amazon S3: オブジェクトレプリケーションの概要
- Amazon ECR でのプライベートイメージレプリケーション
- グローバルテーブル: DynamoDB を使用した複数リージョンレプリケーション
- Amazon Elasticache for Redis OSS: グローバルデータストアを使用した AWS リージョン 間のレプリケーション
- Amazon RDS の耐障害性
- Amazon Aurora Global Database の使用
- AWS Global Accelerator デベロッパーガイド
- AWS KMS のマルチリージョンキー
- Amazon Route 53: DNS フェイルオーバーの設定
- Amazon Application Recovery Controller (ARC) デベロッパーガイド
- AWS リージョン間での Amazon EventBridge イベントの送受信
- AWS のサービスによるマルチリージョンアプリケーションの作成 (ブログシリーズ)
- Disaster Recovery (DR) Architecture on AWS, Part I: Strategies for Recovery in the Cloud
- AWS でのディザスタリカバリ (DR) アーキテクチャ、パート III: パイロットライトとウォームスタンバイ

関連動画:

- AWS re:Invent 2018: マルチリージョンアクティブ/アクティブアプリケーション用アーキテクチャパターン
- AWS re:Invent 2019: AWS グローバルネットワークインフラストラクチャの革新性とオペレーション

REL10-BP02 単一のロケーションに制約されるコンポーネントのリカバリを自動化する

ワークロードのコンポーネントを実行できるのが単一のアベイラビリティーゾーンまたはオンプレミスのデータセンターでのみである場合は、定義した復旧目標内でワークロードを全面的に再構築する機能を実装する必要があります。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

技術的な制約のためにワークロードを複数のロケーションにデプロイするベストプラクティスが不可能な場合は、回復性を確保するための代替パスを採り入れる必要があります。このような場合、必要なインフラストラクチャを再作成し、アプリケーションを再デプロイし、必要なデータを再作成する機能を自動化する必要があります。

例えば、Amazon EMR は同じアベイラビリティーゾーンで特定のクラスターのすべてのノードを起動します。これは、同じゾーンでクラスターを実行すると、データアクセス率が高くなり、ジョブフローのパフォーマンスが向上するためです。このコンポーネントがワークロードの回復力のために必要な場合は、クラスターとそのデータを再デプロイする方法が必要です。また、Amazon EMR では、マルチ AZ を使用する以外の方法で冗長性をプロビジョニングする必要があります。複数のノードをプロビジョニングすることが可能です。EMR File System (EMRFS) を使用することで EMR のデータを Amazon S3 に保存することができ、そのデータを、今度は複数のアベイラビリティーゾーンまたは AWS リージョンを横断してレプリケートすることができます。

同様に、Amazon Redshift でも、選択した AWS リージョン内の、ランダムに選択されたアベイラビリティーゾーンにクラスターがデフォルトでプロビジョニングされます。すべてのクラスターノードが同じゾーンにプロビジョニングされます。

オンプレミスのデータセンターにデプロイされたサーバーベースのステートフルなワークロードの場合、AWS Elastic Disaster Recovery を使用して AWS のワークロードを保護できます 既に AWS でホストされてる場合は、Elastic Disaster Recovery を使用することでワークロードを別のアベイラビリティーゾーンまたはリージョンに保護することができます。Elastic Disaster Recovery は、軽量のステージングエリアへの、ブロックレベルの継続的なレプリケーションを行い、オンプレミスおよびクラウドベースのアプリケーションの高速かつ信頼性の高い復旧を実現します。

実装手順

1. 自己修復を実装します。可能であれば自動スケーリングを利用して、インスタンスとコンテナをデプロイします。自動スケーリングを利用できない場合は、EC2 インスタンスの自動復旧機能を利用するか、Amazon EC2 または ECS のコンテナのライフサイクルイベントを利用して自己修復自動化を実装します。
    - 単一インスタンス IP アドレスや、プライベート IP アドレス、Elastic IP アドレス、インスタンスメタデータを必要としないインスタンスとコンテナのワークロードには、Amazon EC2 Auto Scaling グループを使用します。
        - 起動テンプレートのユーザーデータを使用して、ほとんどのワークロードを自己修復できるオートメーションを実装できます。

- 単一インスタンス IP アドレスや、プライベート IP アドレス、Elastic IP アドレス、インスタンスメタデータを必要とするワークロードには、Amazon EC2 インスタンスの自動復旧機能を使用します。
  - 自動復旧は、インスタンスの障害が検出されると、復旧ステータスアラートを SNS トピックに送信します。
- オートスケーリングや EC2 の復旧機能を利用できない場合は、Amazon EC2 インスタンスのライフサイクルイベントや Amazon ECS イベントを利用して自己修復を自動化します。
  - 必要なプロセスロジックに従ってコンポーネントを修復するオートメーションを呼び出すには、イベントを利用します。
- 単一のロケーションに制限されているステートフルワークロードは AWS Elastic Disaster Recovery を使用して保護します。

リソース

関連ドキュメント:

- Amazon ECS イベント
- Amazon EC2 Auto Scaling のライフサイクルフック
- インスタンスの耐障害性
- Amazon ECS サービスを自動的にスケールする
- Amazon EC2 Auto Scaling とは
- AWS Elastic Disaster Recovery

REL10-BP03 バルクヘッドアーキテクチャを使用して影響範囲を制限する

バルクヘッドアーキテクチャ (セルベースアーキテクチャとも呼ばれる) を実装して、ワークロード内の障害の影響を限られた数のコンポーネントに制限します。

期待される成果: セルベースアーキテクチャでは、複数の分離されたワークロードのインスタンスを使用します。各インスタンスはセルと呼ばれます。各セルは独立しており、その他のセルとは状態を共有せず、ワークロードのリクエスト全体のサブセットを処理します。これにより、不適切なソフトウェア更新などの障害による個別のセルおよび処理中のリクエストへの予測される影響が軽減されます。例えばワークロードが 10 個のセルを使用して 100 個のリクエストを処理していると、障害が発生した場合、リクエスト全体の 90% は障害の影響を受けません。

一般的なアンチパターン:

- セルを際限なく増殖させる。
- コードの更新やデプロイをすべてのセルに同時に適用する。
- セル間で状態またはコンポーネントを共有する (ルーターレイヤーを除く)。
- 複雑なビジネスロジックやルーティングロジックをルーターレイヤーに追加する。
- セル間のインタラクションを最小に抑えない。

このベストプラクティスを活用するメリット: セルベースアーキテクチャでは、多くの一般的な障害はセル自体に封じ込められるため、障害の分離を強化できます。このように障害を分離することで、コードのデプロイに失敗したり、リクエストが破損したり特定の障害モードを呼び出したり (ポイズンピルリクエスト) するなど、他の方法では封じ込めが難しい障害が起きても回復が可能になります。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

船ではバルクヘッドが使用されており、船体が破損しても、船体の一部のセクションのみに封じ込めることができます。複雑なシステムでは、このパターンを模して障害分離を実現する場合がよくあります。障害部分を分離した境界は、ワークロード内の障害の影響を限られた数のコンポーネントに限定します。境界の外部のコンポーネントは、障害の影響を受けません。障害を分離した複数の境界を使用して、ワークロードへの影響を制限することができます。AWS では、複数のアベイラビリティーゾーンとリージョンを使用して障害分離を実現できます。この障害分離の概念はワークロードのアーキテクチャにも拡張できます。

ワークロード全体は、パーティションキーによってセルに分割されます。このキーは、サービスの粒度に従うか、サービスのワークロードをセル間のインタラクションを最小限にして分割できる自然な方法に従う必要があります。パーティションキーの例には、カスタマー ID、リソース ID、またはほとんどの API コールで簡単にアクセスできるその他のパラメータなどがあります。セルルーティングレイヤーは、パーティションキーに基づいて個々のセルにリクエストを分散し、クライアントに単一のエンドポイントを提示します。

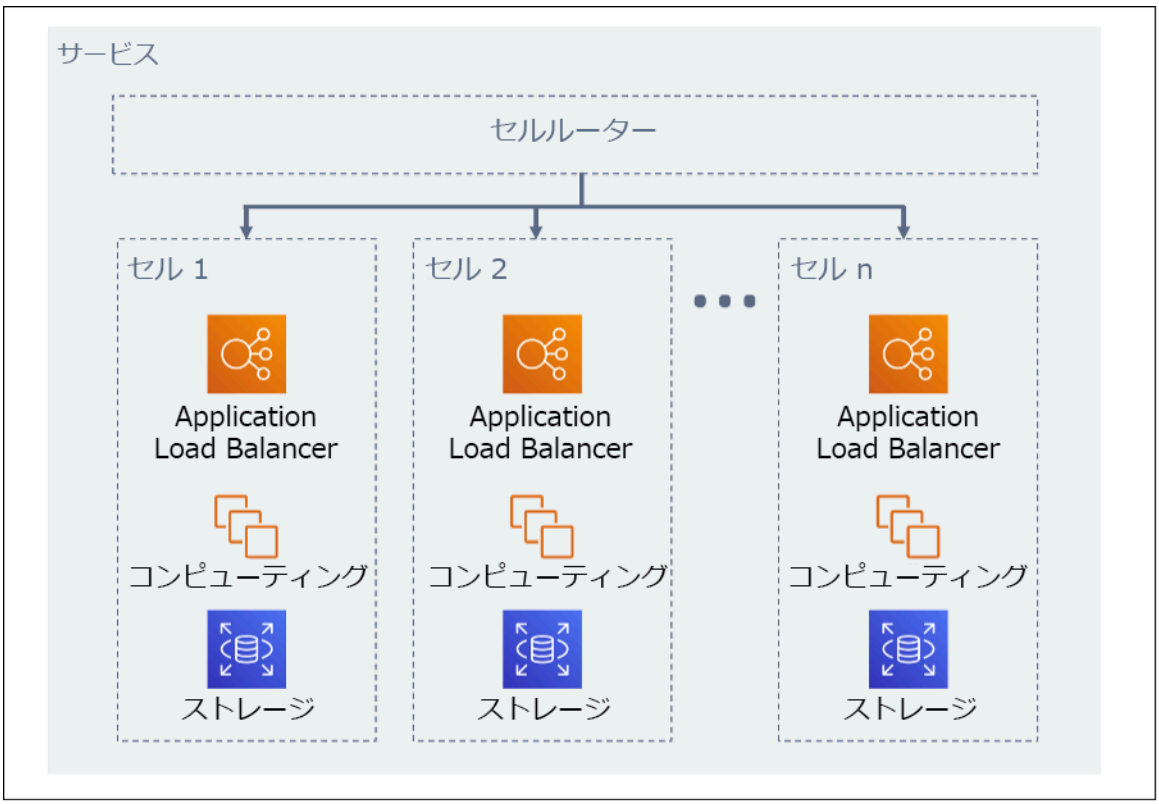


図 11: セルベースアーキテクチャ

実装手順

セルベースアーキテクチャを設計する場合、以下のとおり、考慮すべき設計上の考慮事項がいくつかあります。

1. パーティションキー: パーティションキーを選択するときは特に注意が必要です。
   - このキーは、サービスの粒度またはサービスのワークロードを最小限のクロスセルインタラクションで分割できる自然な方法に沿って分割を行う必要があります。例えば、`customer ID` や `resource ID` などがあります。
   - パーティションキーは、あらゆるリクエストにおいて、直接またはその他のパラメータによって決定論的に容易に推測できる方法で使用できる必要があります。
2. 永続的なセルマッピング: アップストリームのサービスは、リソースのライフサイクルを通じて 1 つのセルとしかやり取りできません。
   - ワークロードによっては、あるセルから別のセルにデータを移行するためにセル移行戦略が必要となる場合があります。セルの移行が必要になる可能性のあるシナリオには、ワークロード内の特定のユーザーまたはリソースが過剰に増大して、専用のセルが必要になる、といった場合があります。

- セルは、セル間で状態またはコンポーネントを共有すべきではありません。
- つまり、セル間のインタラクションは回避するか、最小限に抑える必要があります。これは、このようなインタラクションにより、セル間の依存関係が形成され、障害分離による改善が妨げられるためです。

3. ルーターレイヤー: ルーターレイヤーはセル間で共有されたコンポーネントであるため、セルと同じ区分化の戦略をとることはできません。
   - ルーターレイヤーでは、パーティションマッピングアルゴリズムを計算効率の高い方法で使用して、リクエストを個別のセルに分散することをお勧めします。例えば、暗号化ハッシュ関数と合同算術を組み合わせてパーティションキーをセルにマップするなどの方法があります。
   - マルチセルへの影響を回避するには、ルーティングレイヤーを可能な限りシンプルかつ水平方向にスケーラブルなものとする必要があります。この場合、このレイヤー内での複雑なビジネスロジックは避ける必要があります。この方法には期待される動作をいつでも簡単に把握できるという利点があり、完全なテスト容易性が実現します。Colm MacCárthaigh が「信頼性、動作の継続、1 杯のおいしいコーヒー」で述べているとおり、信頼性の高いシステムを生み反脆弱性を最小限に抑えるものは、シンプルな設計と継続的な取り組みです。
4. セルのサイズ: セルにはサイズの上限を設定する必要があり、それを超えるサイズは許容すべきではありません。
   - 最大サイズを特定するには、限界点に達して安全なオペレーションの限界が明らかになるまで徹底的にテストを実施する必要があります。テストプラクティスの実装方法の詳細については、「REL07-BP04 ワークロードの負荷テストを実施する」を参照してください。
   - 全体的なワークロードの増大は、セルの追加によるべきです。これにより、需要の拡大に合わせてワークロードをスケールできるようになります。
5. マルチ AZ またはマルチリージョン戦略: さまざまな障害に対抗するには、耐障害性を多層的に活用する必要があります。
   - 回復力のためには、複数の防御層を構築するアプローチを使用する必要があります。1 つの層では、複数の AZ を使用して高可用性アーキテクチャを構築することによって、小規模で一般的な混乱に対して保護します。もう 1 つの防御層では、広範囲の自然災害やリージョンレベルの混乱など、まれな出来事に対して保護します。この 2 番目の層には、複数の AWS リージョンにまたがるアプリケーションの設計が必要です。ワークロードのためのマルチリージョン戦略の実装は、国の広い範囲に影響を与える自然災害や、リージョン全体に及ぶ技術的障害に対する保護に役立ちます。マルチリージョンアーキテクチャの実装はかなり複雑になることがあり、通常、ほとんどのワークロードには不要である点に注意してください。詳細については、「REL10-BP01 複数の場所にワークロードをデプロイする」を参照してください。

6. コードのデプロイ: コードの変更は、すべてのセルに同時にデプロイするのではなく段階的にデプロイする方法が推奨されます。
   - これにより、不適切なデプロイや人的エラーによる複数のセルでの障害発生可能性を最小限に抑えることができます。詳細については、「安全なハンズオフデプロイメントの自動化」を参照してください。

リソース

関連するベストプラクティス:

- REL07-BP04 ワークロードの負荷テストを実施する
- REL10-BP01 複数の場所にワークロードをデプロイする

関連ドキュメント:

- 信頼性、動作の継続、1 杯のおいしいコーヒー
- AWS and Compartmentalization
- シャッフルシャーディングを使ったワークロードの分離
- 安全なハンズオフデプロイメントの自動化

関連動画:

- AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small
- AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338)
- Shuffle-sharding: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)
- AWS Summit ANZ 2021 - Everything fails, all the time: Designing for resilience

コンポーネントの障害に耐えられるようにワークロードを設計する

高い可用性と低い平均復旧時間 (MTTR) の要件を持つワークロードは、回復力を考慮した設計をする必要があります。

ベストプラクティス

- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する

- REL11-BP02 正常なリソースにフェイルオーバーする
- REL11-BP03 すべてのレイヤーの修復を自動化する
- REL11-BP04 復旧中はコントロールプレーンではなくデータプレーンを利用する
- REL11-BP05 静的安定性を使用してバイモーダル動作を防止する
- REL11-BP06 イベントが可用性に影響する場合に通知を送信する
- REL11-BP07 可用性の目標と稼働時間のサービスレベルアグリーメント (SLA) を満たす製品を設計する

REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する

ワークロードの状態を継続的にモニタリングすることで、お客様と自動化システムが障害やパフォーマンスの低下の発生をすみやかに把握できるようにします。ビジネス価値に基づいて主要業績評価指標 (KPI) をモニタリングします。

復旧および修復メカニズムはすべて、問題を迅速に検出する機能から開始する必要があります。技術的な障害を最初に検出して、解決できるようにするのです。ただし、可用性はワークロードがビジネス価値を提供する能力に基づいているため、これを測定する主要業績評価指標 (KPI) が検出および修正戦略の一部である必要があります。

期待される成果: ワークロードの重要なコンポーネントは個別にモニタリングされ、障害が発生したタイミングと場所で障害を検出してアラートを出します。

一般的なアンチパターン:

- アラームが設定されていないため、停止は通知なしで発生する。
- アラームは存在しますが、そのしきい値では対応するために十分な時間がない。
- メトリクスは、目標復旧時間 (RTO) を満たすのに十分な頻度で収集されない。
- ワークロードの顧客向けインターフェイスのみがアクティブにモニタリングされる。
- 技術的なメトリクスのみ収集し、ビジネス関数のメトリクスは収集しない。
- ワークロードのユーザーエクスペリエンスを測定するメトリクスがない。
- 作成されたモニタが多すぎる。

このベストプラクティスを活用するメリット: すべてのレイヤーで適切なモニタリングを行うことで、検出までの時間を短縮して、復旧時間を短縮できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

モニタリングの対象となるすべてのワークロードを特定します。モニタリングする必要があるワークロードのすべてのコンポーネントを特定したら、次はモニタリングの間隔を決定します。モニタリングの間隔は、障害の検出にかかる時間に基づいた復旧を開始する速さに直接影響します。平均検出時間 (MTTD) は、障害が発生してから修理作業が開始されるまでの時間です。サービスのリストは広範囲かつ完全でなければなりません。

モニタリングは、アプリケーション、プラットフォーム、インフラストラクチャ、ネットワークを含むアプリケーションスタックのすべてのレイヤーを対象とする必要があります。

モニタリング戦略では、Gray failure の影響を考慮する必要があります。Gray failure の詳細については、ホワイトペーパー「Advanced Multi-AZ Resilience Patterns」の「Gray failures」を参照してください。

実装手順

- モニタリング間隔は、どの程度迅速に復旧する必要があるかによって異なります。復旧時間は復旧にかかる時間によって決まるため、この時間と目標復旧時間 (RTO) を考慮して、収集の頻度を決定する必要があります。
- コンポーネントとマネージドサービスの詳細なモニタリングを設定します。
  - EC2 インスタンスと Auto Scaling の詳細モニタリングが必要かどうかを判断します。詳細モニタリングは 1 分間隔メトリクスを提供し、デフォルトのモニタリングは 5 分間隔メトリクスを提供します。
  - RDS の拡張モニタリングが必要かどうかを判断します。拡張モニタリングでは、RDS インスタンスのエージェントを使用して、さまざまなプロセスまたはスレッドに関する有益な情報を取得します。
  - Lambda、API Gateway、Amazon EKS、Amazon ECS、すべてのロードバランサーのタイプに不可欠なサーバーレスコンポーネントの、モニタリング要件を決定します。
  - Amazon S3、Amazon FSx、Amazon EFS、Amazon EBS のストレージコンポーネントのモニタリング要件を決定します。
- ビジネスの重要業績評価指標 (KPI) を測定するカスタムメトリクスを作成します。ワークロードには主要なビジネス機能が実装されており、いつ間接的な問題が発生したのかを特定するのに役立つ KPI として使用される必要があります。
- ユーザー Canary を使用して、障害に対するユーザーエクスペリエンスをモニタリングします。顧客の行動を実行およびシミュレートできる合成トランザクションテスト (「canary テスト」ともい

う。「カナリアデプロイ」と混同しないこと) は、最も重要なテストプロセスの 1 つです。さまざまなリモートロケーションからワークロードエンドポイントに対してこれらのテストを常に実行します。

- ユーザーのエクスペリエンスを追跡するカスタムメトリクスを作成します。顧客のエクスペリエンスを測定できる場合は、コンシューマーエクスペリエンスが低下するタイミングを判断できます。
- ワークロードのいずれかの要素が正常に動作していない場合にこれを検出し、リソースを自動的にスケールする時期を示す、アラームを設定します。アラームは、ダッシュボードに視覚的に表示したり、Amazon SNS または E メールを介して送信したり、Auto Scaling と連携させてワークロードリソースをスケールアップ/スケールダウンするのに使用したりすることができます。
- ダッシュボードを作成してメトリクスを視覚化します。ダッシュボードは、傾向や異常値などの潜在的な問題の指標を視覚的に確認したり、調査対象となり得る問題の存在を示したりするために使用できます。
- サービスに分散トレースモニタリングを作成します。分散型モニタリングにより、アプリケーションと基盤となるサービスがどのように動作しているかを理解して、パフォーマンスの問題やエラーの根本原因を特定し、トラブルシューティングすることができます。
- モニタリングシステム (CloudWatch または X-Ray を使用) のダッシュボードとデータ収集を別のリージョンとアカウントに作成します。
- AWS Health でサービスの低下について最新情報を入手してください。AWS User Notifications を通じて E メールやチャットチャネルに、目的に合った AWS Health イベント通知を作成し、Amazon EventBridge を通じてモニタリングツールやアラートツールをプログラムで統合します。

リソース

関連するベストプラクティス:

- 可用性
- REL11-BP06 イベントが可用性に影響する場合に通知を送信する

関連ドキュメント:

- canary の使用 (Amazon CloudWatch Synthetics
- インスタンスの詳細モニタリングを有効または無効にする
- 拡張モニタリング
- Auto Scaling グループとインスタンスを Amazon CloudWatch でモニタリングする

- カスタムメトリクスをパブリッシュする
- Amazon CloudWatch でのアラームの使用
- Amazon CloudWatch ダッシュボードの使用
- クロスアカウントクロスリージョンダッシュボード
- AWS X-Ray のよくある質問
- 可用性について

関連動画:

- グレー障害の緩和

関連する例:

- つのオブザーバビリティワークショップ: X-Ray の探究

関連ツール:

- CloudWatch
- CloudWatch X-Ray

REL11-BP02 正常なリソースにフェイルオーバーする

リソース障害の発生時に、正常なリソースが引き続きリクエストに対応します。ロケーション障害 (アベイラビリティーゾーンや AWS リージョンなど) に対しては、障害のないロケーションの正常なリソースにフェイルオーバーするシステムを用意します。

サービスを設計するときは、リソース、アベイラビリティーゾーン、またはリージョンに負荷を分散します。そのため、個々のリソースの障害は、残りの正常なリソースにトラフィックをシフトすることによって緩和できます。障害発生時にサービスがどのように検出され、ルーティングされるかを検討してください。

障害復旧を念頭に置いてサービスを設計します。AWS では、障害からの復旧時間とデータへの影響を最小限に抑えるサービスを設計しています。当社のサービスは主にデータストアを使用しており、リクエストが認識されるのは、リージョン内の複数のレプリカにわたりデータが永続的に保存された後です。これらのサービスは、セル単位の分離とアベイラビリティーゾーンにより提供される障害切

り分けを活用するように構成されています。当社は、運用上の手順の多くで自動化を幅広く使用しています。また、中断から迅速に復旧するために、置換と再起動の機能を最適化しています。

フェイルオーバーを可能にするパターンとデザインは、AWS プラットフォームサービスごとに異なります。AWS ネイティブのマネージドサービスの多くは、複数のアベイラビリティーゾーン (Lambda や API Gateway など) にネイティブに対応しています。他の AWS サービス (EC2 や EKS など) では、AZ 間でのリソースまたはデータストレージのフェイルオーバーをサポートするための特定のベストプラクティス設計が必要です。

モニタリングは、フェイルオーバーリソースが正常であることを確認し、リソースのフェイルオーバーの進行状況を追跡して、ビジネスプロセスの復旧をモニタリングするために設定する必要があります。

期待される成果: システムは、新しいリソースを自動または手動で使用してパフォーマンスの低下から復旧できます。

一般的なアンチパターン:

- 障害を想定した計画が、計画と設計の段階に含まれていない。
- RTO と RPO が確立されていない。
- モニタリングが不十分で、障害が発生しているリソースを検出できない。
- 障害ドメインの適切な隔離。
- マルチリージョンのフェイルオーバーが考慮されていない。
- フェイルオーバーを決定する際の障害検出の感度が高すぎる、または過剰である。
- フェイルオーバー設計のテストや検証を行っていない。
- オートヒーリングのオートメーションを実行したが、ヒーリングが必要とされたことは通知しない。
- すぐにフェイルバックするのを防ぐための減衰期間を十分に設けていない。

このベストプラクティスを活用するメリット: 適切に機能低下し、迅速に回復することで、障害発生時でも信頼性を維持する耐障害性の高いシステムを構築できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

Elastic Load Balancing や Amazon EC2 Auto Scaling などの AWS サービスは、複数のリソースおよびアベイラビリティーゾーンへの負荷分散に役立ちます。そのため、個々のリソース (EC2 インスタ

ンスなど) の障害や、アベイラビリティーゾーンの障害を、残りの正常なリソースにトラフィックをシフトすることによって緩和できます。

マルチリージョンのワークロードの場合、設計はさらに複雑です。例えば、クロスリージョンリードレプリカを使用すると、データを複数の AWS リージョンにデプロイできます。ただし、リードレプリカをプライマリに昇格させ、トラフィックを新しいエンドポイントに向けるには、やはりフェイルオーバーが必要です。Amazon Route 53、Amazon Application Recovery Controller (ARC)、Amazon CloudFront、AWS Global Accelerator は、AWS リージョン間でトラフィックをルーティングするために役立ちます。

Amazon S3、Lambda、API Gateway、Amazon SQS、Amazon SNS、Amazon SES、Amazon Pinpoint、Amazon ECR、AWS Certificate Manager、EventBridge、Amazon DynamoDB などの AWS サービスは、AWS によって複数のアベイラビリティーゾーンに自動的にデプロイされます。障害が発生した場合、これらの AWS サービスはトラフィックを正常なロケーションに自動的にルーティングします。データは複数のアベイラビリティーゾーンに冗長的に保存され、使用可能な状態を維持します。

Amazon RDS、Amazon Aurora、Amazon Redshift、Amazon EKS、Amazon ECS の場合、マルチ AZ は設定オプションです。フェイルオーバーが開始すると、AWS はトラフィックを正常なインスタンスに転送させることができます。このフェイルオーバーアクションは、AWS が行うことも、必要に応じてお客様が実行することもできます。

Amazon EC2 インスタンス、Amazon Redshift、Amazon ECS タスク、Amazon EKS ポッドの場合は、デプロイ先のアベイラビリティーゾーンを選択します。設計によっては、Elastic Load Balancing に、異常なゾーンにあるインスタンスを検出してトラフィックを正常なゾーンにルーティングするソリューションが用意されています。Elastic Load Balancing は、オンプレミスのデータセンター内のコンポーネントにトラフィックをルーティングすることもできます。

マルチリージョンのトラフィックフェイルオーバーの場合、再ルーティングでは、インターネットドメインを定義し、ヘルスチェックなどのルーティングポリシーを割り当ててトラフィックを正常なリージョンにルーティングする手段として、Amazon Route 53、Amazon Application Recovery Controller、AWS Global Accelerator、Route 53 Private DNS for VPC、または CloudFront を活用できます。AWS Global Accelerator はアプリケーションへの固定エントリポイントとして機能する静的 IP アドレスを提供し、インターネットの代わりに AWS グローバルネットワークを使用して任意の AWS リージョンのエンドポイントにルーティングすることで、パフォーマンスと信頼性を高めます。

実装手順

- すべての適切なアプリケーションとサービスのフェイルオーバー設計を作成します。各アーキテクチャコンポーネントを分離し、各コンポーネントの RTO と RPO を満たすフェイルオーバー設計を作成します。
- フェイルオーバープランに必要なすべてのサービスを使用して、下位環境 (開発環境やテスト環境など) を構成します。Infrastructure as Code (IaC) を使用してソリューションをデプロイし、再現性を確保します。
- フェイルオーバー設計を実装してテストするために、2 つ目のリージョンなどの復旧サイトを設定します。必要に応じて、テスト用のリソースを一時的に設定して、追加コストを抑えることができます。
- どのフェイルオーバープランを AWS で自動化するか、どのフェイルオーバープランを DevOps プロセスで自動化できるか、どのフェイルオーバープランを手動で行うかを判断します。各サービスの RTO と RPO を文書化して測定します。
- フェイルオーバープレイブックを作成し、各リソース、アプリケーション、サービスをフェイルオーバーするためのすべての手順を含めます。
- フェイルバックプレイブックを作成し、各リソース、アプリケーション、サービスをフェイルバックするためのすべての手順を (タイミングとともに) 含めます。
- プレイブックを開始してリハーサルするための計画を立てます。シミュレーションとカオステストを使用して、プレイブックの手順と自動化をテストします。
- ロケーション障害 (アベイラビリティーゾーンや AWS リージョンなど) に対しては、障害のないロケーションの正常なリソースにフェイルオーバーするシステムを用意します。フェイルオーバーテストの前に、クォータ、自動スケーリングのレベル、実行中のリソースを確認してください。

リソース

関連する Well-Architected のベストプラクティス:

- REL13 - 災害対策 (DR) を計画する
- REL10 - 障害部分を切り離してワークロードを保護する

関連ドキュメント:

- RTO と RPO のターゲットを設定する
- Route 53 加重ルーティングを使用したフェイルオーバー

- Amazon Application Recovery Controller を使用したディザスタリカバリ
- EC2 with autoscaling
- EC2 Deployments - Multi-AZ
- ECS Deployments - Multi-AZ
- Amazon Application Recovery Controller を使用したトラフィックの切り替え
- Lambda を使用した Application Load Balancer とフェイルオーバー
- ACM Replication and Failover
- Parameter Store Replication and Failover
- ECR クロスリージョンレプリケーションとフェイルオーバー
- Secrets Manager クロスリージョンレプリケーションの設定
- 新機能 – Amazon Elastic File System (EFS) のレプリケーション
- EFS クロスリージョンレプリケーションとフェイルオーバー
- ネットワークフェイルオーバー
- MRAP を使用した S3 エンドポイントフェイルオーバー
- S3 のクロスリージョンレプリケーションを作成する
- AWS でのクロスリージョンフェイルオーバーとグレースフルフェイルバックに関するガイダンス
- マルチリージョンの Global Accelerator によるフェイルオーバー
- DRS によるフェイルオーバー

関連する例:

- でのディザスタリカバリAWS
- の Elastic Disaster RecoveryAWS

REL11-BP03 すべてのレイヤーの修復を自動化する

障害を検出したら、自動化機能を使用して修復するアクションを実行します。パフォーマンスの低下は、内部のサービスメカニズムによって自動的に修復される場合もあれば、修復アクションによってリソースを再起動または削除する必要がある場合もあります。

セルフマネージドアプリケーションやクロスリージョン修復では、復旧設計と自動修復プロセスを既存のベストプラクティスから引き出すことができます。

リソースを再起動または削除する機能は、障害を修復するための重要なツールです。ベストプラクティスは、可能な限りサービスをステートレスにすることです。これにより、リソースの再起動時のデータまたは可用性が失われるのを防ぎます。クラウドでは、再起動の一環として、リソース全体 (コンピューティングインスタンス、サーバーレス関数など) を置き換えることができます (通常はそうする必要があります)。再起動自体は、障害から復旧するための簡単で信頼できる方法です。ワークロードでは、さまざまなタイプの障害が発生します。障害は、ハードウェア、ソフトウェア、通信、オペレーションなどさまざまな部分で発生する可能性があります。

再起動または再試行は、ネットワークリクエストにも適用されます。依存関係にあるシステムからエラーが返された場合、ネットワークのタイムアウトの場合と依存関係にあるシステムの障害の両方に同じ復旧アプローチを適用します。どちらのイベントもシステムに類似の影響を与えるため、どちらかのイベントを特例とするのではなく、エクスポネンシャルバックオフとジッターで限定的に再試行するという類似の戦略を適用します。再起動の機能は、復旧指向コンピューティングと高可用性クラスターアーキテクチャを特徴とする復旧メカニズムです。

期待される成果: 障害の検出を修正するために、自動アクションが実行されます。

一般的なアンチパターン:

- 自動スケーリングなしでリソースをプロビジョニングする。
- インスタンスまたはコンテナにアプリケーションを個別にデプロイする。
- 自動復旧を使用せずに、複数の口ケーションにデプロイできないアプリケーションをデプロイします。
- 自動スケーリングと自動復旧が修復に失敗するアプリケーションを手動で修復する。
- フェイルオーバーデータベースの自動化はない。
- トラフィックを新しいエンドポイントに再ルーティングする自動化された方法が足りない。
- ストレージのレプリケーションはない。

このベストプラクティスを活用するメリット: 自動修復により、平均回復時間を短縮し、可用性を向上させることができます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

Amazon EKS やその他の Kubernetes サービスの設計には、レプリカセットまたはステートフルセットの最小値と最大値の両方、およびクラスターとノードグループの最小サイズが含まれている必要が

あります。これらのメカニズムは、Kubernetes コントロールプレーンを使用して障害を自動的に修正しながら、継続的に利用可能な処理リソースを最小限に抑えます。

コンピューティングクラスターを使用してロードバランサーを介してアクセスされる設計パターンでは、Auto Scaling グループを活用する必要があります。Elastic Load Balancing (ELB) は、受信アプリケーショントラフィックを 1 つ以上のアベイラビリティーゾーン (AZ) にある複数のターゲットと仮想アプライアンスに自動的に分散します。

ロードバランシングを使用しないクラスター化されたコンピューティングベースの設計では、少なくとも 1 台のノードが失われることを想定したサイズにする必要があります。これにより、新しいノードを復旧している間も、容量が減少する可能性がある状態でサービスが稼働し続けることができます。サービスの例としては、Mongo、DynamoDB Accelerator、Amazon Redshift、Amazon EMR、Cassandra、Kafka、MSK-EC2、Couchbase、ELK、Amazon OpenSearch Service などがあります。これらのサービスの多くは、追加の自動修復機能を使用して設計できます。一部のクラスターテクノロジーでは、ノードが失われたときにアラートを生成し、新しいノードを再作成するための自動または手動のワークフローをトリガーする必要があります。このワークフローは、AWS Systems Manager を使用して自動化でき、問題を迅速に修正できます。

Amazon EventBridge を使用すれば、CloudWatch アラームなどのイベントや、その他の AWS サービスの状態の変化などを、モニタリングおよびフィルタリングすることができます。イベント情報に基づいて、AWS Lambda、Systems Manager Automation、または他のターゲットを呼び出して、ワークロードに対してカスタム修正ロジックを実行できます。Amazon EC2 Auto Scaling は、EC2 インスタンスの状態をチェックするように設定できます。インスタンスが実行中以外の状態にある場合、またはシステムステータスが損なわれている場合、Amazon EC2 Auto Scaling はインスタンスが異常であるとみなして代替インスタンスを起動します。大規模な置き換え (アベイラビリティーゾーン全体の喪失など) の場合、静的安定性が高可用性のために優先されます。

実装手順

- Auto Scaling グループを使用して、ワークロードに階層をデプロイします。Auto Scaling は、ステートレスなアプリケーションで自己修復を実行し、キャパシティを追加および削除できます。
- 前述のコンピューティングインスタンスの場合は、ロードバランシングを使用して適切なロードバランサーのタイプを選択します。
- Amazon RDS の修復を検討します。スタンバイインスタンスでは、スタンバイインスタンスへの自動フェイルオーバーを設定します。Amazon RDS リードレプリカの場合、リードレプリカをプライマリにするには自動化されたワークフローが必要です。
- 複数のロケーションにデプロイできないアプリケーションがデプロイされている EC2 インスタンスに自動復旧を実装し、障害時の再起動を許容できます。自動復旧は、アプリケーションが複数の

ロケーションにデプロイできない場合に、障害が発生したハードウェアを交換してインスタンスを再起動するために使用できます。インスタンスメタデータおよび関連する IP アドレスは保持されます。また、EBS ボリュームと Amazon Elastic File System または File Systems for Lustre および Windows へのマウントポイントも保持されます。AWS OpsWorks を使用することで、レイヤーレベルで EC2 インスタンスの自動修復を設定できます。

- 自動スケーリングまたは自動復旧を使用できない場合、または自動復旧が失敗した場合は、AWS Step Functions と AWS Lambda を使用して自動復旧を実装します。自動スケーリングを使用できず、さらに、自動復旧が使用できないか、自動復旧が失敗した場合は、AWS Step Functions と AWS Lambda を使用して修復を自動化できます。
- Amazon EventBridge を使用すれば、CloudWatch アラームなどのイベントや、その他の AWS サービスの状態の変化などを、モニタリングおよびフィルタリングすることができます。イベント情報に基づいて、AWS Lambda (または他のターゲット) を呼び出し、ワークロードに対してカスタム修正ロジックを実行できます。

リソース

関連するベストプラクティス:

- 可用性
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する

関連ドキュメント:

- AWS Auto Scaling の仕組み
- Amazon EC2 自動復旧
- Amazon Elastic Block Store (Amazon EBS)
- Amazon Elastic File System (Amazon EFS)
- Amazon FSx for Lustre とは
- Amazon FSx for Windows File Server とは
- AWS OpsWorks: 障害の発生したインスタンスを自動修復機能によって交換する
- What is AWS Step Functions?
- とはAWS Lambda
- What Is Amazon EventBridge?

- Amazon CloudWatch でのアラームの使用
- Amazon RDS フェイルオーバー
- SSM - Systems Manager Automation
- 回復力のあるアーキテクチャのベストプラクティス

関連動画:

- Automatically Provision and Scale OpenSearch Service
- Amazon RDS Failover Automatically

関連する例:

- Amazon RDS フェイルオーバーワークショップ

関連ツール:

- CloudWatch
- CloudWatch X-Ray

REL11-BP04 復旧中はコントロールプレーンではなくデータプレーンを利用する

コントロールプレーンはリソースの作成、読み取り、記述、更新、削除、一覧表示 (CRUDL) に使用される管理 API を提供し、データプレーンは日々のサービストラフィックを処理します。回復力に影響を与える可能性のあるイベントへの回復または軽減対応を実装するときは、サービスの回復、再スケーリング、復元、修復、またはフェイルオーバーに最小限のコントロールプレーンのオペレーションを使用することに焦点を当ててください。これらのパフォーマンスの低下イベント中は、データプレーンのアクションがどのアクティビティよりも優先されるはずです。

例えば、新しいコンピューティングインスタンスの起動、ブロックストレージの起動、キューサービスの記述などは、すべてコントロールプレーンのアクションです。コンピューティングインスタンスを起動後、コントロールプレーンは、容量のある物理ホストの検索、ネットワークインターフェイスの割り当て、ローカルブロックストレージボリュームの準備、認証情報の生成、セキュリティルールの追加など、複数のタスクを実行する必要があります。コントロールプレーンは複雑なオーケストレーションになりがちです。

期待される成果: リソースに障害が発生すると、システムは、トラフィックを障害のあるリソースから正常なリソースに移行することで、自動または手動で回復できます。

一般的なアンチパターン:

- トラフィックを再ルーティングするための DNS レコードの変更への依存。
- リソースのプロビジョニングが不十分なため、障害が発生したコンポーネントの交換をコントロールプレーンのスケーリング操作に依存。
- あらゆるカテゴリの障害を修復するために、広範囲にわたるマルチサービス、マルチ API コントロールプレーンアクションに依存。

このベストプラクティスを活用するメリット: 自動修復の成功率が高くなると、平均復旧時間が短縮され、ワークロードの可用性が向上します。

このベストプラクティスを活用しない場合のリスクレベル: 中。特定の種類のサービス低下の場合、コントロールプレーンが影響を受けます。コントロールプレーンを広範囲に使用して修復すると、復旧時間 (RTO) と平均復旧時間 (MTTR) が長くなる可能性があります。

実装のガイダンス

データプレーンのアクションを制限するには、サービスの復元に必要なアクションについて各サービスを評価します。

DNS トラフィックをシフトするときは Amazon Application Recovery Controller を活用します。これらの機能により、障害から回復するアプリケーションの機能を継続的にモニタリングし、複数の AWS リージョン、アベイラビリティーゾーン、およびオンプレミスにまたがってアプリケーションの回復を管理できます。

Route 53 ルーティングポリシーにコントロールプレーンが使用されているため、復旧の際にコントロールプレーンに依存しないようにします。Route 53 データプレーンは、DNS クエリに応答し、ヘルスチェックを実行し、評価します。これらはグローバルに配布され、「100% 利用可能」のサービスレベルアグリーメント (SLA) 向けに設計されています。

Route 53 のリソースを作成、更新、削除する Route 53 管理 API およびコンソールは、コントロールプレーンで実行します。コントロールプレーンは、DNS の管理に必要な強力な一貫性と耐久性を重視するように設計されています。これを達成するために、コントロールプレーンは単一のリージョン、米国東部 (バージニア北部) に配置されています。どちらのシステムも非常に高い信頼性で構築されていますが、コントロールプレーンは SLA には含まれません。まれに、データプレーンの回復

力設計によって可用性を維持できるときでも、コントロールプレーンでは維持でない場合があります。ディザスタリカバリおよびフェイルオーバーメカニズムについては、データプレーンの機能を使用して、可能な限り最善の信頼性を提供してください。

コンピューティングインフラストラクチャは静的に安定するように設計して、インシデント中にコントロールプレーンを使用しないようにします。例えば、Amazon EC2 インスタンスを使用している場合は、新しいインスタンスを手動でプロビジョニングしたり、Auto Scaling グループにインスタンスの追加を指示したりすることは避けてください。回復性を最大限に高めるには、フェイルオーバーに使用するクラスターに十分な容量をプロビジョニングしてください。この容量のしきい値を制限する必要がある場合は、エンドツーエンドのシステム全体にスロットルを設定して、限られたリソースに到達するトラフィックの合計を安全に制限してください。

Amazon DynamoDB、Amazon API Gateway、ロードバランサー、AWS Lambda サーバーレスなどのサービスでは、これらのサービスを使用するとデータプレーンを活用できます。ただし、新しい関数、ロードバランサー、API ゲートウェイ、または DynamoDB テーブルの作成はコントロールプレーンのアクションであり、イベントの準備やフェイルオーバーアクションのリハーサルとして、パフォーマンス低下の前に完了しておく必要があります。Amazon RDS では、データプレーンのアクションによりデータへのアクセスが可能になります。

データプレーン、コントロールプレーン、および、AWS が高可用性の目標を満たすサービスをいかに構築しているのか、についての詳細は、「アベイラビリティーゾーンを使用した静的安定性」を参照してください。

データプレーンでの運用と、コントロールプレーンでの運用を理解します。

実装手順

パフォーマンス低下のイベント後に復元する必要のある各ワークロードについて、フェイルオーバーランブック、高可用性設計、自動修復設計、または HA リソース復元プランの評価を行います。コントロールプレーンのアクションとみなされる可能性のある各アクションを特定します。

コントロールアクションをデータプレーンアクションに変更することを検討します。

- 自動スケーリング (コントロールプレーン) を事前スケールされた Amazon EC2 リソース (データプレーン) に変更します。
- Amazon EC2 インスタンススケーリング (コントロールプレーン) を AWS Lambda スケーリング (データプレーン) に変更します。
- Kubernetes を使用するあらゆる設計とコントロールプレーンのアクションの性質を評価します。ポッドの追加は Kubernetes のデータプレーンのアクションです。アクションはノードの追加では

なく、ポッドの追加に限定する必要があります 過剰にプロビジョニングされたノードを使用することは、コントロールプレーンのアクションを制限するための推奨される方法です。

データプレーンのアクションが同じ修復に影響を与えられる代替アプローチを検討してください。

- Route 53 レコードの変更 (コントロールプレーン) または Amazon Application Recovery Controller (データプレーン)
- 自動化がさらに進んだアップデート用の Route 53 ヘルスチェック

サービスがミッションクリティカルな場合は、影響を受けていないリージョンでより多くのコントロールプレーンとデータプレーンのアクションを実行できるように、セカンダリリージョンのサービスを検討してください。

- プライマリリージョンの Amazon EC2 Auto Scaling または Amazon EKS と、セカンダリリージョンの Amazon EC2 Auto Scaling または Amazon EKS とを比較し、セカンダリリージョンにトラフィックをルーティングします (コントロールプレーンのアクション)。
- セカンダリプライマリでリードレプリカを作成するか、プライマリリージョンで同じアクションを試みる (コントロールプレーンのアクション)

リソース

関連するベストプラクティス:

- 可用性
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する

関連ドキュメント:

- APN Partner: partners that can help with automation of your fault tolerance
- AWS Marketplace: products that can be used for fault tolerance
- Amazon Builders' Library: 小さなサービスに制御を任せて分散型システムのオーバーヘッドを回避する
- Amazon DynamoDB API (コントロールプレーンとデータプレーン)
- AWS Lambda の実行 (コントロールプレーンとデータプレーンに分割)
- AWS Elemental MediaStore Data Plane

- Amazon Application Recovery Controller を使用して回復力の高いアプリケーションを構築する、パート 1: 単一リージョンスタック
- Amazon Application Recovery Controller を使用して回復力の高いアプリケーションを構築する、パート 2: マルチリージョンスタック
- Amazon Route 53 を用いたディザスタリカバリ (DR) のメカニズム
- Amazon Application Recovery Controller とは
- Kubernetes Control Plane and data plane

関連動画:

- Back to Basics - Using Static Stability
- Building resilient multi-site workloads using AWS global services

関連する例:

- Amazon Application Recovery Controller の紹介
- Amazon Builders' Library: 小さなサービスに制御を任せて分散型システムのオーバーヘッドを回避する
- Amazon Application Recovery Controller を使用して回復力の高いアプリケーションを構築する、パート 1: 単一リージョンスタック
- Amazon Application Recovery Controller を使用して回復力の高いアプリケーションを構築する、パート 2: マルチリージョンスタック
- アベイラビリティーゾーンを使用した静的安定性

関連ツール:

- Amazon CloudWatch
- AWS X-Ray

REL11-BP05 静的安定性を使用してバイモーダル動作を防止する

ワークロードは静的に安定し、1 つの通常モードでのみ動作する必要があります。バイモーダル動作とは、通常モードと障害モードでワークロードが異なる動作を示す場合をいいます。

例えば、別のアベイラビリティーゾーン (AZ) で新しいインスタンスを起動して、AZ の障害からの復旧を試みることができます。これにより、障害モード中にバイモーダル応答が発生する可能性があります。バイモーダル動作を防止するために、静的に安定し、1 つのモードでのみ動作するワークロードを構築する必要があります。この例では、これらのインスタンスは障害発生前に 2 番目の AZ でプロビジョニングしておく必要があります。この静的に安定した設計により、確実にワークロードが単一のモードでのみ動作するようになります。

期待される成果: ワークロードは、通常モードと障害モードでバイモーダル動作を示しません。

一般的なアンチパターン:

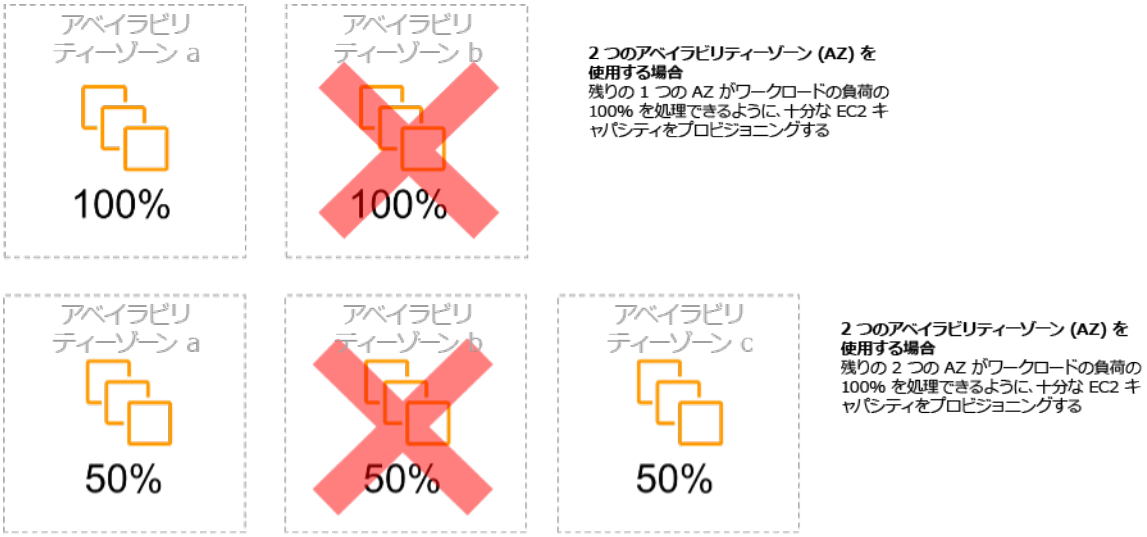
- 障害の範囲に関係なく、リソースが常にプロビジョニング可能であると仮定する。
- 障害発生時にリソースを動的に取得しようとする。
- 障害が発生するまで、ゾーンまたはリージョン間で適切なリソースをプロビジョニングしない。
- コンピューティングリソースにのみ静的に安定した設計を検討する。

このベストプラクティスを活用するメリット: 静的に安定した設計で実行されるワークロードは、通常のイベント時でも障害発生時でも予測可能な結果を得ることができます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

バイモーダル動作とは、例えばアベイラビリティーゾーン (AZ) に障害が発生した場合に新しいインスタンスの起動に依存するなど、通常モードと障害モードでワークロードが異なる動作を示す場合をいいます。バイモーダル動作の例は、安定した Amazon EC2 設計により、1 つの AZ が削除された場合にワークロードの負荷を処理するのに十分な数のインスタンスが各 AZ にプロビジョニングされる場合です。Elastic Load Balancing または Amazon Route 53 ヘルスチェックを使用して、障害のあるインスタンスから負荷を分散します。トラフィックがシフトした後、AWS Auto Scaling を使用して、障害が発生したゾーンのインスタンスを非同期で置き換え、正常なゾーンで起動します。EC2 インスタンスやコンテナなどのコンピューティングデプロイの静的安定性があると、信頼性が最も高くなります。



複数のアベイラビリティーゾーン (AZ) にわたる EC2 インスタンスの静的安定性

これは、このモデルのコストと、すべてのレジリエンスケースでワークロードを維持することのビジネス価値に照らして検討する必要があります。コンピューティングキャパシティを少なくプロビジョニングし、障害発生時に新しいインスタンスの起動に依存するほうがコストは低くなりますが、大規模な障害 (AZ やリージョンの障害など) の場合、このアプローチは、運用プレーンと、影響を受けていないゾーンまたはリージョンでリソースが十分に利用可能であることの両方に依存するため、あまり効果的ではありません。

また、ソリューションでは、信頼性とワークロードに必要なコストを比較検討する必要があります。静的に安定したアーキテクチャは、複数の AZ に分散されているコンピューティングインスタンス、データベースリードレプリカ設計、Kubernetes (Amazon EKS) クラスター設計、マルチリージョンフェイルオーバーアーキテクチャなど、さまざまなアーキテクチャに適用されます。

各ゾーンでより多くのリソースを使用することにより、より静的に安定した設計を実装することもできます。ゾーンを追加することで、静的安定性に必要な追加のコンピューティング量を減らすことができます。

バイモーダル動作の例に、ネットワークのタイムアウトにより、システム全体の設定状態の再読み込みが始まることがあります。これにより想定外の負荷が別のコンポーネントに加わり、そのコンポーネントで障害が発生し、想定外の結果につながる可能性があります。この負のフィードバックループは、ワークロードの可用性に影響を与えます。そこで、静的に安定し、1 つのモードでのみ動作するシステムを構築する必要があります。静的に安定した設計は、一定の作業を行い、常に一定の周期で設定状態を更新します。呼び出しに失敗すると、ワークロードは以前にキャッシュされた値を使用し、アラームを開始します。

バイモーダル動作のもう 1 つの例は、障害発生時にクライアントがワークロードキャッシュをバイパスできるようにすることです。これは、クライアントのニーズに対応するソリューションのように思われるかもしれませんが、ワークロードの需要を大幅に変更し、障害が発生する可能性が高くなります。

重要なワークロードを評価して、どのワークロードにこのタイプのレジリエンス設計が必要かを判断します。重要と思われるものについては、各アプリケーションコンポーネントを確認する必要があります。静的安定性評価を必要とするサービスの種類の例は次のとおりです。

- コンピューティング: Amazon EC2、EKS-EC2、ECS-EC2、EMR-EC2
- データベース: Amazon Aurora、Amazon RDS、Amazon Redshift。
- ストレージ: Amazon S3 (単一ゾーン)、Amazon EFS (マウント)、Amazon FSx (マウント)
- ロードバランサー: 特定の設計で

実装手順

- 静的に安定し、1 つのモードでのみ動作するシステムを構築します。この場合、1 つのアベイラビリティーゾーンまたはリージョンが削除された場合にワークロード容量を処理するのに十分な数のインスタンスを、各アベイラビリティーゾーンまたはリージョンにプロビジョニングします。正常なリソースへのルーティングには、次のようなさまざまなサービスを使用できます。
  - クロスリージョン DNS ルーティング
  - MRAP Amazon S3 マルチリージョンのルーティング
  - AWS Global Accelerator
  - Amazon Application Recovery Controller
- 単一のプライマリインスタンスまたはリードレプリカが失われた場合に備えて、データベースリードレプリカを設定します。トラフィックがリードレプリカによって処理されている場合、各アベイラビリティーゾーンと各リージョンでの量は、そのゾーンまたはリージョンに障害が発生した場合の全体的な必要量と同じにします。
- アベイラビリティーゾーンに障害が発生した場合に保存されるデータに対して静的に安定するように設計された Amazon S3 ストレージに重要なデータを設定します。Amazon S3 One Zone-IA ストレージクラスを使用する場合、そのゾーンが失われると、保存されたデータへのアクセスが最小限に抑えられるため、静的に安定しているとみなすべきではありません。
- ロードバランサーは、誤って、または設計により、特定のアベイラビリティーゾーン (AZ) を処理するように設定されている場合があります。この場合、静的に安定した設計では、ワークロードをより複雑な設計の複数の AZ に分散することが考えられます。セキュリティ、レイテンシー、ま

たはコスト上の理由から、元の設計を使用してゾーン間のトラフィックを削減できる場合があります。

リソース

関連する Well-Architected のベストプラクティス:

- 可用性
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する
- REL11-BP04 復旧中はコントロールプレーンではなくデータプレーンを利用する

関連ドキュメント:

- ディザスタリカバリディザスタリカバリプランの依存関係を最小化する
- The Amazon Builders' Library: アベイラビリティーゾーンを使用した静的安定性
- 障害分離境界
- アベイラビリティーゾーンを使用した静的安定性
- マルチゾーン RDS
- ディザスタリカバリディザスタリカバリプランの依存関係を最小化する
- クロスリージョン DNS ルーティング
- MRAP Amazon S3 マルチリージョンのルーティング
- AWS Global Accelerator
- Amazon Application Recovery Controller
- 1 ゾーン Amazon S3
- クロスゾーンロードバランシング

関連動画:

- Static stability in AWS: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)

REL11-BP06 イベントが可用性に影響する場合に通知を送信する

しきい値の違反が検出されると、イベントによって引き起こされた問題が自動的に解決された場合でも、通知が送信されます。

自動ヒーリング機能により、ワークロードの信頼性を高めることができます。ただし、対処する必要のある根本的な問題もあいまいになる可能性があります。根本原因の問題を解決できるように、自動ヒーリングによって対処されたものを含む問題のパターンを検出できるように、適切なモニタリングとイベントを実装します。

回復力を備えたシステムは、低下イベントがすぐに適切なチームに通知されるよう設計されています。これらの通知は、1 つまたは複数の通信チャネルを通じて送信する必要があります。

期待される成果: エラー率、レイテンシー、その他の主要業績評価指標 (KPI) メトリクスなどのしきい値を超えると、直ちにオペレーションチームにアラートが送信されます。これにより、これらの問題をできるだけ早く解決し、ユーザーへの影響を回避するか最小限に抑えることができます。

一般的なアンチパターン:

- 送信するアラームが多すぎる。
- 実行できないアラームを送信する。
- アラームのしきい値の設定が高すぎる (感度が高すぎる) か、低すぎる (感度が低すぎる)。
- 外部依存関係に対するアラームを送信しない。
- モニタリングとアラームを設計する際にグレー障害を考慮していない。
- ヒーリングのオートメーションを実行しているが、ヒーリングが必要となったことを適切なチームに通知しない。

このベストプラクティスを活用するメリット: 復旧の通知により、運用チームやビジネスチームはサービスの低下を認識し、直ちに対応して平均検出時間 (MTTD) と平均修復時間 (MTTR) の両方を最小限に抑えることができます。復旧イベントの通知により、発生頻度の低い問題を無視することもなくなります。

このベストプラクティスを活用しない場合のリスクレベル: 中 適切なモニタリングとイベント通知のメカニズムを実装しないと、自動ヒーリングで対処されるものも含め、問題のパターンを検出できなくなる可能性があります。チームは、ユーザーがカスタマーサービスに連絡した場合、または偶然に見つけた場合にしか、システムの低下を認識できなくなります。

実装のガイダンス

モニタリング戦略の定義において、アラームのトリガーは一般的なイベントです。このイベントには、アラームの識別子、アラームの状態 (`IN ALARM` や `OK`) およびアラームをトリガーした原因の詳細が含まれる可能性があります。多くの場合、1 件のアラームイベントが検出されると、1 件の E メール通知が送信されます。これは、1 件のアラームにつき 1 つのアクションの例です。アラーム通

知は、問題があることを適切な担当者に通知するため、オブザーバビリティにおいて非常に重要です。ただし、オブザーバビリティソリューションでイベントに対するアクションが洗練されると、人間の介入を必要とせずに問題を自動的に修正できます。

KPI モニタリングアラームを設定すると、しきい値を超えたときに適切なチームにアラートが送信されます。これらのアラートを使用して、低下の修復を試みる自動プロセスをトリガーすることもできます。

より複雑なしきい値のモニタリングには、複合アラームを検討する必要があります。複合アラームでは、多くの KPI モニタリングアラームを使用して、運用上のビジネスロジックに基づいてアラートを作成します。CloudWatch アラームは、Amazon SNS 統合または Amazon EventBridge を使用して、E メールを送信するか、サードパーティーのインシデント追跡システムにインシデントをログ記録するように設定できます。

実装手順

モニタリング対象のワークロードに応じて、次のようなさまざまなタイプのアラームを作成します。

- アプリケーションアラームは、ワークロードの一部が正常に動作していないことを検出するために使用します。
- インフラストラクチャアラームは、リソースをスケールするタイミングを示します。アラームは、ダッシュボードに視覚的に表示したり、Amazon SNS または E メールを介して送信したり、Auto Scaling と連携させてワークロードリソースをスケールイン/スケールアウトするのに使用したりすることができます。
- シンプルな静的アラームを作成して、メトリクスが指定された評価期間の静的しきい値を超える状況をモニタリングできます。
- 複合アラームは、複数のソースからの複雑なアラームに対応できます。
- アラームを作成したら、適切な通知イベントを作成します。Amazon SNS API を直接呼び出して、通知を送信し、修正やコミュニケーションのための自動化をリンクすることができます。
- AWS Health でサービスの低下について最新情報を入手してください。AWS User Notifications を通じて E メールやチャットチャネルに、目的に合った AWS Health イベント通知を作成し、Amazon EventBridge を通じてモニタリングツールやアラートツールをプログラムで統合します。

リソース

関連する Well-Architected のベストプラクティス:

- 可用性

関連ドキュメント:

- 静的しきい値に基づいて CloudWatch アラームを作成する
- Amazon EventBridge とは
- Amazon Simple Notification Service とは
- カスタムメトリクスを発行する
- Amazon CloudWatch でのアラームの使用
- CloudWatch 複合アラームの設定
- re:Invent 2022 で公開された AWS オブザーバビリティに関する最新情報

関連ツール:

- CloudWatch
- CloudWatch X-Ray

REL11-BP07 可用性の目標と稼働時間のサービスレベルアグリーメント (SLA) を満たす製品を設計する

可用性の目標と稼働時間のサービスレベルアグリーメント (SLA) を満たすように製品を設計します。可用性目標またはアップタイム SLA を公開するか、非公開で同意する場合は、アーキテクチャと運用プロセスが SLA をサポートするように設計されていることを確認します。

期待される成果: 各アプリケーションには、可用性の目標とパフォーマンスメトリクスの SLA が定義されています。これらのメトリクスは、ビジネス上の成果を満たすためにモニタリングおよび管理できます。

一般的なアンチパターン:

- SLA を設定せずにワークロードを設計およびデプロイする。
- 合理的な理由やビジネス要件なしに SLA メトリクスが高すぎに設定されている。
- 依存関係とその基盤となる SLA を考慮せずに SLA を設定する。
- 回復力の共有責任モデルを考慮せずにアプリケーションが設計される。

このベストプラクティスを活用するメリット: 主要な復元力の目標に基づいてアプリケーションを設計すると、ビジネス目標と顧客の期待を満たすことができます。このような目標は、さまざまなテクノロジーを評価し、さまざまなトレードオフを考慮に入れたアプリケーション設計プロセスの促進につながります。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

アプリケーションの設計では、ビジネス、オペレーション、財務上の目的に基づくさまざまな要件を考慮する必要があります。運用上の要件の範囲内で、ワークロードを適切にモニタリングおよびサポートできるように、ワークロードには特定の回復性メトリクス目標が必要です。ワークロードのデプロイ後は、回復性メトリクスの設定や派生の作成を行うべきではありません。これは設計段階で定義し、さまざまな決定とトレードオフのガイドとしての役割を果たす必要があります。

- 各ワークロードには、独自の回復性メトリクスセットが必要です。このようなメトリクスは、その他のビジネスアプリケーションとは異なる場合があります。
- 依存関係を減らすと、可用性にプラスの影響を与えることができます。各ワークロードは、独自の依存関係と SLA を考慮に入れる必要があります。通常、ワークロードの目標以上の可用性目標を持つ依存関係を選択します。
- 可能であれば、依存関係が損なわれてもワークロードが正常に動作できるように、疎結合の設計を検討します。
- コントロールプレーンの依存関係を減らします。特に、復旧時または機能低下時の依存関係を低減します。ミッションクリティカルなワークロードに対して静的安定性がある設計を評価します。リソースを節約して、ワークロード内のこのような依存関係の可用性を向上します。
- オブザーバビリティと計測は、平均検出時間 (MTTD) と平均修復時間 (MTTR) の短縮と SLA の達成のために重要です。
- 分散システムの可用性を向上させる要素は、障害の頻度が少ない (MTBF が長い)、障害検出時間が短い (MTTD が短い)、修理時間が短い (MTTR が短い) という 3 つです。
- ワークロードの回復性メトリクスを確立し、それを満たすことは、効果的な設計の基本となります。このような設計では、設計の複雑性、サービスの依存関係、パフォーマンス、スケーリング、コストのトレードオフを考慮する必要があります。

実装手順

- 以下の質問を検討し、ワークロードの設計を確認して、文書化します。
  - コントロールプレーンはワークロードのどの個所で使用されますか。

  - ワークロードはどのように耐障害性を実装しますか。
  - どのようなスケーリング、自動スケーリング、冗長性、高可用性コンポーネントの設計パターンがありますか。
  - どのようなデータ整合性と可用性の要件がありますか。
  - リソース節約またはリソースの静的安定性に関する考慮事項はありますか。
  - どのようなサービスの依存関係がありますか。
- ステークホルダーと協力して、ワークロードアーキテクチャに基づいて SLA メトリクスを定義します。ワークロードで使用されるすべての依存関係の SLA を検討します。
- SLA 目標の設定後、SLA を満たすようにアーキテクチャを最適化します。
- SLA を満たす設計が定まったら、運用上の変更、プロセスの自動化、MTTD および MTTR の短縮についても重視するランブックを導入します。
- デプロイ後、SLA についてモニタリングしてレポートを作成します。

リソース

関連するベストプラクティス:

- REL03-BP01 ワークロードをセグメント化する方法を選択する
- REL10-BP01 複数の場所にワークロードをデプロイする
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する
- REL11-BP03 すべてのレイヤーの修復を自動化する
- REL12-BP04 カオスエンジニアリングを使用して回復力をテストする
- REL13-BP01 ダウンタイムやデータ消失に関する復旧目標を定義する
- ワークロードの状態の理解

関連ドキュメント:

- 冗長性を実装した可用性
- 信頼性の柱 - 可用性
- 可用性の測定
- AWS 障害分離境界
- 回復性に関する責任共有モデル
- アベイラビリティーゾーンを使用した静的安定性

- AWS サービスレベルアグリーメント (SLA)
- Guidance for Cell-based Architecture on AWS
- AWS インフラストラクチャ
- マルチ AZ の高度なレジリエンスパターン (ホワイトペーパー)

関連サービス:

- Amazon CloudWatch
- AWS Config
- AWS Trusted Advisor

テストの信頼性

本番環境のストレスに耐えられるようにワークロードを設計した後、ワークロードが意図したとおりに動作し、期待する弾力性を実現することを確認する唯一の方法が、テストを行うことです。

バグまたはパフォーマンスのボトルネックがワークロードの信頼性に影響を与える可能性があるため、ワークロードが機能の要件と非機能要件を満たしていることを検証するためにテストします。ワークロードの弾#性をテストして、本番環境でしか表面化しない潜在的なバグを見つけられるようにします。このようなテストを定期的に行います。

ベストプラクティス

- REL12-BP01 プレイブックを使用して障害を調査する
- REL12-BP02 インシデント後の分析を実行する
- REL12-BP03 スケーラビリティおよびパフォーマンス要件をテストする
- REL12-BP04 カオスエンジニアリングを使用して回復力をテストする
- REL12-BP05 定期的にゲームデーを実施する

REL12-BP01 プレイブックを使用して障害を調査する

調査プロセスをプレイブックに文書化することで、よく理解されていない障害シナリオに対する一貫性のある迅速な対応が可能になります。プレイブックは、障害シナリオの原因となる要因を特定するために実行される事前定義されたステップです。プロセスステップの結果は、問題が特定されるか、エスカレーションされるまで、次のステップを決定するために使用されます。

プレイブックは、対応措置を効果的に実行できるようにするために立てる必要があるプロアクティブな計画です。本番環境でプレイブックに含まれていない障害シナリオが発生した場合は、まず問題に対処します (火を消します)。その後、振り返って問題に対処するために実行した手順を見て、これらの手順を用いてプレイブックに新しいエントリを追加します。

プレイブックは特定のインシデントに対応するために用いられる一方、ランブックは特定の結果を達成するために使用されます。多くの場合、ランブックは日常的なアクティビティに用いられる一方、プレイブックは非日常的なイベントに応えるために使用します。

一般的なアンチパターン:

- 問題の診断やインシデントへの対応を行うためのプロセスを知ることなくワークロードのデプロイを計画する。
- イベントを調査するときに、ログとメトリクスを収集するシステムに関する計画外の決定。
- データを取得するためにメトリクスとイベントを十分な期間保持していない。

このベストプラクティスを活用するメリット: プレイブックをキャプチャすることで、プロセスへの一貫した遵守が実現できます。プレイブックを成文化することによって、手動のアクティビティによるエラーの発生が抑制されます。プレイブックのオートメーションは、チームメンバーの介入の必要性をなくし、または介入の開始時に追加情報を提供することによって、イベントへの対応時間を短縮します。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

- プレイブックを使用して問題を特定します。プレイブックは、問題を調査するための文書化されたプロセスです。プロセスをプレイブックに文書化することで、障害シナリオに対する一貫性のある迅速な対応が可能になります。プレイブックには、十分なスキルを持った人物が該当する情報の収集、障害の潜在的原因の特定、障害の切り分け、寄与する要因の特定 (インシデント後の分析の実行) を行うために必要な情報とガイダンスが含まれている必要があります。
  - プレイブックをコードとして実装します。プレイブックをスクリプト化することにより、運用をコードとして実行し、一貫性を保ち、手動プロセスによって発生するエラーを抑制または低減します。プレイブックは、問題に寄与する要因を特定するために必要となり得るさまざまなステップを表す複数のスクリプトで構成できます。ランブックのアクティビティは、プレイブックのアクティビティの一部として呼び出されるまたは実行されるか、特定されたイベントへの応答としてプレイブックの実行を引き起こす場合があります。
    - AWS Systems Manager を使った運営計画の自動化

- AWS Systems Manager Run Command
- AWS Systems Manager Automation
- AWS Lambda とは
- What Is Amazon EventBridge?
- Amazon CloudWatch でのアラームの使用

リソース

関連ドキュメント:

- AWS Systems Manager Automation
- AWS Systems Manager Run Command
- AWS Systems Manager を使った運営計画の自動化
- Amazon CloudWatch でのアラームの使用
- canary の使用 (Amazon CloudWatch Synthetics)
- What Is Amazon EventBridge?
- AWS Lambda とは

関連する例:

- プレイブックとランブックによるオペレーションの自動化

REL12-BP02 インシデント後の分析を実行する

顧客に影響を与えるイベントを確認し、寄与する要因と予防措置を特定します。この情報を使用して、再発を制限または回避するための緩和策を開発します。迅速で効果的な対応のための手順を開発します。対象者に合わせて調整された、寄与因子と是正措置を必要に応じて伝えます。必要に応じて根本原因を他の人に伝える方法を確立します。

既存のテストで問題が見つからなかった理由を評価します。テストがまだ存在しない場合は、このケースのテストを追加します。

期待される成果: チームが合意済みの一貫したアプローチで、インシデント後の分析を取り扱います。そのメカニズムの 1 つがエラーの修正 (COE) プロセスです。COE プロセスは、チームがインシ

デントの根本原因を特定、理解、対処するのに役立つと同時に、同じインシデントの再発を防止するメカニズムとガードレールの構築にも有益です。

一般的なアンチパターン:

- 寄与因子を見つけるが、他の潜在的な問題やリスクの軽減策についてさらに詳しく調べない。
- 人的エラーの原因を特定するだけで、人的ミスを防止し得るトレーニングやオートメーションを実施しない。
- 原因究明よりも責任を追及するばかりで恐怖心を煽り、オープンなコミュニケーションが妨げられる。
- インサイトを共有できない。インシデント分析の結果を少人数のグループで抱え込み、他の人が教訓を活かせなくなります。
- 組織の知識をキャプチャするメカニズムがない。学んだ教訓を最新のベストプラクティスという形で保存しないと貴重なインサイトが失われ、同様または類似の根本原因でインシデントが再発することになります。

このベストプラクティスを活用するメリット: インシデント後の分析を実施し、結果を共有することで、他のワークロードが同じ寄与因子を実装した場合のリスクを軽減し、インシデントが発生する前に軽減策または自動復旧を実装できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

優れたインシデント後の分析は、システムの別の場所で使用されているアーキテクチャパターンの問題に対して、共通のソリューションを提案する機会になります。

COE プロセスの基礎は、問題を文書化して対処することです。重要な根本原因を文書化し、その原因をレビューして確実に解決するための標準化した手法を定義しておくことを推奨します。インシデント後の分析プロセスには明確に担当者を割り当てます。インシデントの調査とフォローアップの監督を担当するチームまたは個人を指名してください。

責任を追及するのではなく、学習と改善を重視する文化を醸成してください。個人の責任を問うのではなく、インシデントの再発防止が目標であることを強調します。

インシデント後の分析を実施するための明確な手順を策定します。これらの手順では、実行すべき手順、収集する情報、分析中に対処すべき重要な問題をまとめておく必要があります。インシデントを徹底的に調査し、直接的な原因だけでなく、根本原因と寄与因子を特定します。5 つの why 分析などの手法を用いて、根本的な問題を掘り下げてください。

インシデント分析から学んだ教訓のリポジトリを維持します。こうした組織の知識は、将来のインシデントや予防策の参考になります。インシデント後の分析から得られた結果とインサイトを共有し、誰でも参加できるインシデント後のレビューミーティングを開催して、学んだ教訓について話し合うことを検討してください。

実装手順

- インシデント後の分析を行う際は、非難の場にならないようにします。これにより、インシデントにかかわった人々は、提案された是正措置を冷静に検討し、誠実な自己評価とチーム間のコラボレーションに努めることができます。
- 重大な問題を文書化する標準化された方法を定義します。このようなドキュメントの構造の例は次のとおりです。
  - 何が起きたのか。
  - 顧客とビジネスにどのような影響がありましたか?
  - 根本原因は何でしたか?
  - これをサポートするデータはありますか?
    - 例えば、メトリクスとグラフ
  - 重要な柱、特にセキュリティとは何を意味するでしょうか?
    - ワークロードを設計するときには、ビジネスの状況に応じて、柱の間でトレードオフを行います。これらのビジネス上の意思決定がエンジニアリングの優先度を左右する可能性があります。開発環境では信頼性を妥協することでコストを削減するという最適化を#う場合や、ミッションクリティカルなソリューションでは、信頼性を最適化するためにコストをかける場合などがあります。セキュリティは常に最優先事項です。顧客を保護する必要があるからです。
  - どのような教訓を学びましたか?
  - どのような是正措置を講じましたか?
    - アクション項目
    - 関連項目
- インシデント後の分析を実施するための明確に定義された標準運用手順を作成します。
- 標準化されたインシデント報告プロセスを用意します。初回のインシデントレポート、ログ、コミュニケーション、インシデントの発生中に取られたアクションなど、すべてのインシデントを包括的に文書化します。
- インシデントが生じても必ずしもシステム停止を伴うわけではありません。ニアミスである場合や、システムがビジネス機能を果たしながら予想外の方法で動作している場合があります。
- フィードバックと教訓に基づいて、インシデント後の分析プロセスを継続的に改善します。

- 重要な検出結果をナレッジ管理システムでキャプチャし、開発者ガイドやデプロイ前のチェックリストに追加すべきパターンを検討します。

リソース

関連ドキュメント:

- correction of error (COE) を開発すべき理由

関連動画:

- Amazon's approach to failing successfully
- AWS re:Invent 2021 - Amazon Builders' Library: Operational Excellence at Amazon

REL12-BP03 スケーラビリティおよびパフォーマンス要件をテストする

負荷テストなどの技法を使用して、ワークロードがスケーリングおよびパフォーマンス要件を満たしていることを検証します。

クラウドでは、ワークロードの本番環境規模のテスト環境をオンデマンドで作成できます。運用環境の動作を正確に予測できない可能性がある、スケールダウンされたテスト環境に依存する代わりに、クラウドを使用して、予想される運用環境を厳密に反映したテスト環境をプロビジョニングできます。この環境は、アプリケーションが直面する実際の条件をより正確にシミュレーションしてテストするのに役立ちます。

パフォーマンステストの取り組みと並行して、基本リソース、スケーリング設定、Service Quotas、回復性設計が負荷状態で期待どおりに動作することを検証することが重要です。この包括的なアプローチにより、最も厳しい条件下でも、アプリケーションを必要に応じて確実にスケーリングおよび実行できることを確認します。

期待される成果: ワークロードは、ピーク負荷にさらされている場合でも、期待される動作を維持します。アプリケーションが成長および進化するにつれて発生する可能性のあるパフォーマンス関連の問題には、積極的に対処します。

一般的なアンチパターン:

- 本番環境と厳密に一致しないテスト環境を使用する。

- 負荷テストは、デプロイメントの継続的インテグレーション (CI) パイプラインの統合された一部としてではなく、個別の 1 回限りのアクティビティとして扱う。
- レスポンスタイム、スループット、スケーラビリティのターゲットなど、明確で測定可能なパフォーマンス要件を定義していない。
- 非現実的または不十分な負荷シナリオでテストを実行し、ピーク負荷、急激なスパイク、持続的な高負荷をテストできない。
- 予想される負荷制限を超えてワークロードをストレステストすることはない。
- 不十分または不適切な負荷テストとパフォーマンスプロファイリングツールを使用する。
- パフォーマンスメトリクスを追跡し、異常を検出するための包括的なモニタリングおよびアラートシステムが不足している。

このベストプラクティスを活用するメリット:

- 負荷テストは、本番環境に移行する前に、システムのパフォーマンスの潜在的なボトルネックを特定するのに役立ちます。本番環境レベルのトラフィックとワークロードをシミュレートすると、レスポンスタイムの遅延、リソースの制約、システム障害など、システムが負荷の処理に苦労する可能性のある領域を特定できます。
- さまざまな負荷条件下でシステムをテストすると、ワークロードのサポートに必要なリソース要件をよりよく理解できます。この情報は、リソース配分について十分な情報に基づいた意思決定を行い、リソースの過剰プロビジョニングや過少プロビジョニングを防ぐのに役立ちます。
- 潜在的な障害点を特定するには、ワークロードが高負荷条件下でどのように機能するかを監視します。この情報は、必要に応じて耐障害性メカニズム、フェイルオーバー戦略、冗長性対策を適切に実装することによって、ワークロードの信頼性と回復力を向上させるのに役立ちます。
- パフォーマンスの問題を早期に特定して対処できるため、システム停止、レスポンス時間の遅延、ユーザーの不満などによるコストのかかる結果を回避できます。
- テスト中に収集された詳細なパフォーマンスデータとプロファイリング情報は、本番環境で発生する可能性のあるパフォーマンス関連の問題のトラブルシューティングに役立ちます。これにより、インシデント対応と解決が迅速になり、ユーザーと組織のオペレーションへの影響が軽減されます。
- 特定の業界では、プロアクティブなパフォーマンステストを行うことにより、ワークロードがコンプライアンス基準を満たすのに役立ち、罰則や法的問題のリスクが軽減されます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

最初のステップは、スケーリングとパフォーマンス要件のすべての側面をカバーする包括的なテスト戦略を定義することです。まず、スループット、レイテンシーヒストグラム、エラー率など、ビジネスニーズに基づいてワークロードのサービスレベル目標 (SLO) を明確に定義します。次に、平均使用量から突然の急増や持続的なピーク負荷まで、さまざまな負荷シナリオをシミュレートできる一連のテストを設計し、ワークロードの動作が SLO を満たしていることを確認します。開発プロセスの早い段階でパフォーマンスの低下を検出するには、これらのテストを自動化し、継続的インテグレーションおよびデプロイパイプラインに統合する必要があります。

スケーリングとパフォーマンスを効果的にテストするには、適切なツールとインフラストラクチャに投資します。これには、現実的なユーザートラフィックを生成できる負荷テストツール、ボトルネックを特定するためのパフォーマンスプロファイリングツール、主要なメトリクスを追跡するためのモニタリングソリューションが含まれます。重要なのは、テスト環境がインフラストラクチャと環境条件の点で本番環境と厳密に一致していることを確認して、テスト結果を可能な限り正確にする必要があります。本番環境のようなセットアップを確実にレプリケートしてスケーリングしやすくするには、Infrastructure as code およびコンテナベースのアプリケーションを使用します。

スケーリングとパフォーマンステストは継続的なプロセスであり、1 回限りのアクティビティではありません。包括的なモニタリングとアラートを実装して、本番環境でのアプリケーションのパフォーマンスを追跡し、このデータを使用してテスト戦略と最適化の取り組みを継続的に改善します。パフォーマンスデータを定期的に分析して、新しい問題を特定し、新しいスケーリング戦略をテストし、最適化を実装してアプリケーションの効率と信頼性を向上させます。反復的なアプローチを採用し、本番データから常に学習する場合、アプリケーションがさまざまなユーザーの需要に適応し、時間の経過と共に回復力と最適なパフォーマンスを維持できることを確認できます。

実装手順

1. レスポンスタイム、スループット、スケーラビリティの目標など、明確で測定可能なパフォーマンス要件を確立します。これらの要件は、ワークロードの使用パターン、ユーザーの期待、ビジネスニーズに基づいている必要があります。
2. 本番環境の負荷パターンとユーザーの動作を正確に模倣できる負荷テストツールを選択して設定します。
3. テスト結果の精度を向上させるために、インフラストラクチャや環境条件など、本番環境に近いテスト環境を設定します。
4. 平均使用パターンからピーク負荷、急速なスパイク、持続的な高負荷まで、幅広いシナリオをカバーするテストスイートを作成します。開発プロセスの早い段階でパフォーマンスのリグレッ

ションを検出するために、テストを継続的インテグレーションおよびデプロイパイプラインに統合します。

5. 負荷テストを実行して実際のユーザートラフィックをシミュレートし、アプリケーションがさまざまな負荷条件下でどのように動作するかを理解します。アプリケーションのストレステストを行うには、予想される負荷を超えて、レスポンス時間の低下、リソースの枯渇、システム障害などの動作を観察します。これにより、アプリケーションの限界点を特定し、スケーリング戦略を通知するのに役立ちます。負荷を段階的に増やしてワークロードのスケーラビリティを評価し、パフォーマンスへの影響を測定してスケーリング制限を特定し、将来の容量ニーズに備えます。
6. 包括的なモニタリングとアラートを実装し、パフォーマンスメトリクスを追跡し、異常を検出し、しきい値を超えたときにスケーリングアクションまたは通知を開始します。
7. パフォーマンスデータを継続的にモニタリングして分析し、改善すべき分野を特定します。テスト戦略と最適化の取り組みを反復します。

リソース

関連するベストプラクティス:

- REL01-BP04 クォータをモニタリングおよび管理する
- REL06-BP01 ワークロードのすべてのコンポーネントをモニタリングする (生成)
- REL06-BP03 通知を送信する (リアルタイム処理とアラーム))

関連ドキュメント:

- 「アプリケーションの負荷テスト」
- Distributed Load Testing on AWS
- アプリケーションパフォーマンスのモニタリング
- Amazon EC2 テストポリシー

関連する例:

- AWS での分散負荷テスト (GitHub)

関連ツール:

- Amazon CodeGuru Profiler

- Amazon CloudWatch RUM
- Apache JMeter
- K6
- Vegeta
- Hey
- ab
- wrk
- Distributed Load Testing on AWS

REL12-BP04 カオスエンジニアリングを使用して回復力をテストする

悪条件下でシステムがどのように反応するかを理解するために、本番環境またはできるだけそれに近い環境で定期的にカオス実験を行います。

期待される成果:

イベント発生時のワークロードの既知の期待動作を検証する回復力テストに加えて、フォールトインジェクション実験や想定外の負荷の注入という形でカオスエンジニアリングを適用し、ワークロードの回復力を定期的に検証します。カオスエンジニアリングと回復力テストの両方を組み合わせることで、ワークロードがコンポーネントの障害に耐え、想定外の中断から影響を最小限に抑えて回復できることを確信できます。

一般的なアンチパターン:

- 回復力を考慮した設計でありながら、障害発生時にワークロードが全体としてどのように機能するかを検証していない。
- 実際の条件および予期される負荷による実験を一切行わない。
- 実験をコードとして処理しないか、開発サイクルを通して維持しない。
- CI/CD パイプラインの一部、またはデプロイの外部のどちらとしても、カオス実験を実行しない。
- どの障害で実験するかを考慮する際に、過去のインシデント後の分析を無視する。

このベストプラクティスを活用するメリット: ワークロードの回復力を検証するために障害を発生させることで、回復力設計の復旧手順が実際の障害発生時にも機能するという確信を得られます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

カオスエンジニアリングは、サービスプロバイダー、インフラストラクチャ、ワークロード、コンポーネントレベルにおいて、現実世界の障害 (シミュレーション) を継続的に発生させる機能を提供し、顧客には最小限の影響しか与えません。これにより、チームは障害から学び、ワークロードの回復力を観察、測定、改善することができます。また、イベント発生時にアラートが発せられ、チームに通知されることを確認することもできます。

カオスエンジニアリングを継続的に実施することで、放置しておくと可用性やオペレーションに悪影響を及ぼす可能性がある、ワークロードの欠陥が浮き彫りになります。

> **Note**
>
> カオスエンジニアリングとは、あるシステムで実験を行い、本稼働時の混乱状態に耐えることができるかどうかの確信を得るための手法です。– カオスエンジニアリングの原則

システムがこれらの混乱に耐えられる場合は、カオス実験を自動化されたリグレッションテストとして維持する必要があります。このように、カオス実験はシステム開発ライフサイクル (SDLC) の一部および CI/CD パイプラインの一部として実行される必要があります。

ワークロードがコンポーネントの障害に耐えられることを確認するために、実際のイベントを実験の一部として挿入します。例えば、Amazon EC2 インスタンスの喪失やプライマリ Amazon RDS データベースインスタンスのフェイルオーバーを実験し、ワークロードに影響がないこと (または最小限の影響にとどまること) を確認します。コンポーネントの障害の組み合わせを使用して、アベイラビリティーゾーンでの中断によって発生する可能性のあるイベントをシミュレートします。

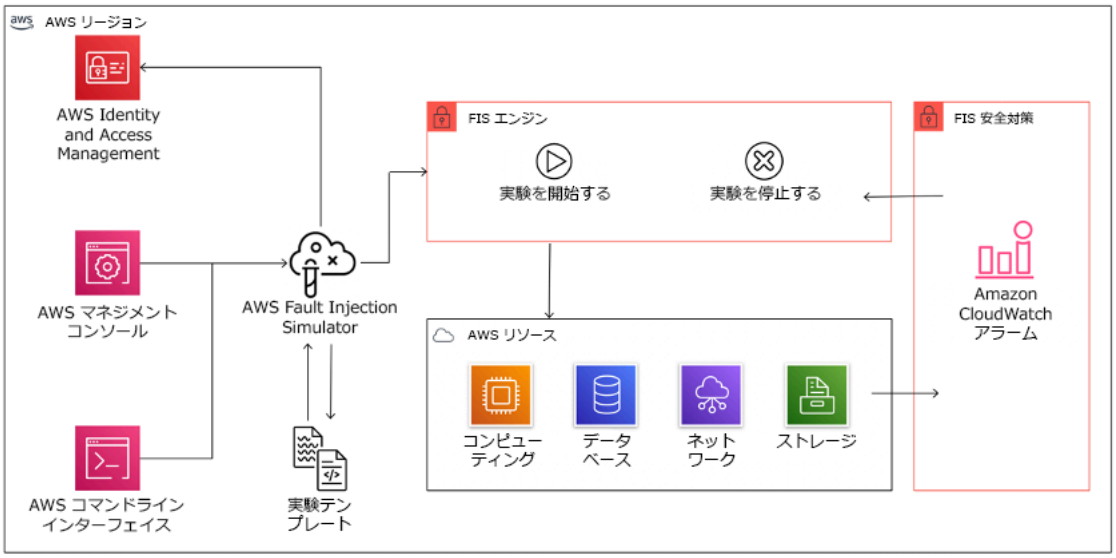
アプリケーションレベルの障害 (クラッシュなど) では、メモリや CPU の枯渇などのストレス要因から始めます。

断続的なネットワーク中断による外部依存関係のフォールバックまたはフェイルオーバーメカニズムを検証するために、数秒から数時間続く指定期間の間、コンポーネントがサードパーティープロバイダーへのアクセスをブロックすることで、そのようなイベントをシミュレートする必要があります。

その他の劣化状態では、機能の低下や応答の遅れが発生し、サービスの中断につながることがよくあります。このパフォーマンス低下の一般的な原因は、主要サービスのレイテンシー増加と、信頼性の低いネットワーク通信 (パケットのドロップ) です。レイテンシー、メッセージのドロップ、DNS 障害などのネットワークへの影響を含むこれらの障害実験には、名前の解決、DNS サービスへの到達、依存サービスへの接続ができないことなどが含まれる可能性があります。

カオスエンジニアリングのツール:

AWS Fault Injection Service (AWS FIS) は、フォールトインジェクション実験を実行するフルマネージドサービスであり、CD パイプラインの一部として、またはパイプラインの外で使用することができます。AWS FIS は、カオスエンジニアリングのゲームデー中に使用するのに適しています。Amazon EC2、Amazon Elastic Container Service (Amazon ECS)、Amazon Elastic Kubernetes Service (Amazon EKS)、Amazon RDS など、さまざまな種類のリソースに障害を同時発生させるのに役立ちます。これらの障害には、リソースの終了、強制フェイルオーバー、CPU またはメモリへのストレス、スロットリング、レイテンシー、パケット損失などが含まれます。Amazon CloudWatch アラームと連携しているため、ガードレールとして停止条件を設定し、想定外の影響を与えた場合に実験をロールバックすることができます。



AWS Fault Injection Service を AWS リソースと統合することで、ワークロードのフォールトインジェクション実験を実行できるようになります。

フォールトインジェクション実験を実行するための、いくつかのサードパーティーオプションがあります。これには、Chaos Toolkit、Chaos Mesh、Litmus Chaos などのオープンソースツールや、Gremlin などの商用オプションが含まれます。AWS に挿入できる障害の範囲を拡張するために、AWS FIS を Chaos Mesh および Litmus Chaos と統合することで、複数のツール間でフォールトインジェクションワークフローを調整できます。例えば、AWS FIS 障害アクションを使用して、ランダムに選択した割合のクラスターノードを終了する間に、Chaos Mesh または Litmus 障害を使用して、ポッドの CPU のストレステストを実行することができます。

実装手順

1. どの障害を実験に使用するかを決定します。

回復力に対するワークロードの設計を評価します。Well-Architected Framework のベストプラクティスを使用して作成するこのような設計では、重要な依存関係、過去のイベント、既知の問題、コンプライアンス要件に基づくリスクを考慮します。回復力を維持するために想定した設計の各要素と、それを低減するために設計する障害をリストアップします。このようなリストの作成の詳細については、「Operational Readiness Review」ホワイトペーパーを参照してください。このホワイトペーパーでは、過去のインシデントの再発を防ぐためのプロセスを作成する方法について説明します。障害モードと影響の分析 (FMEA) プロセスにより、障害とそれがワークロードに与える影響をコンポーネントレベルで分析するためのフレームワークが提供されます。FMEA については、Adrian Cockcroft の「障害モードと継続的回復力」で詳しく説明しています。

2. 各障害に優先度を割り当てます。

   「高」「中」「低」などの大まかな分類から始めます。優先度を評価するには、障害の発生頻度と障害によるワークロード全体への影響を考慮します。

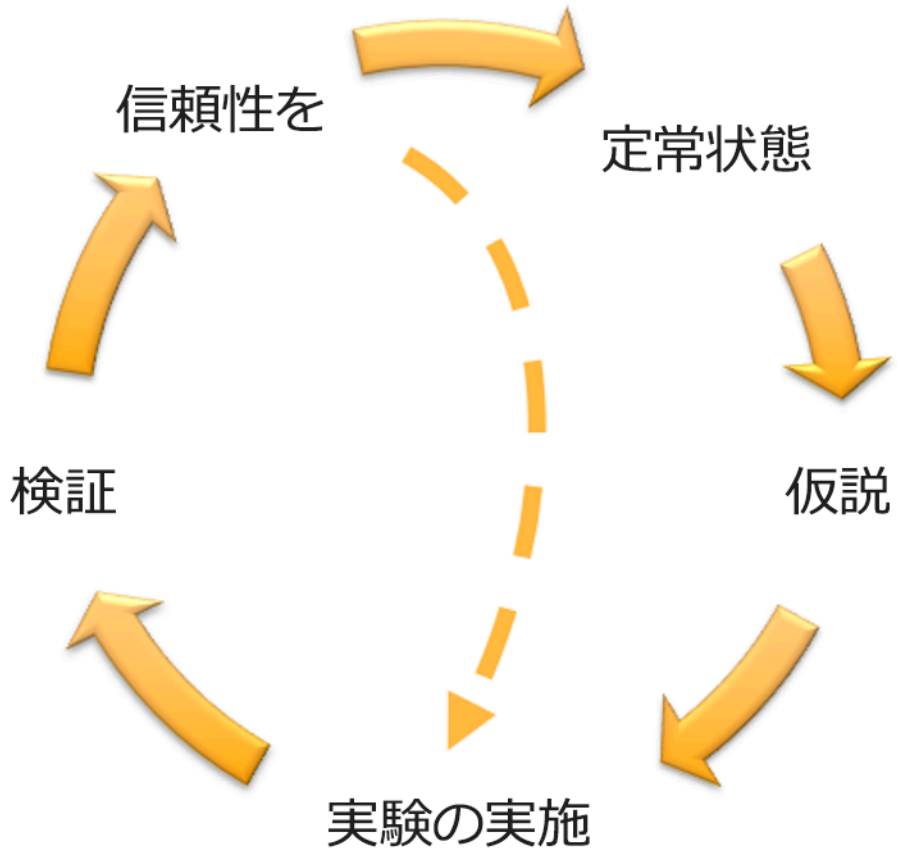
   ある障害の発生頻度を考慮する場合、利用可能であれば、そのワークロードの過去のデータを分析します。利用できない場合は、類似の環境で実行されている他のワークロードのデータを使用します。

   ある障害の影響を考慮する場合、一般的に、障害の範囲が大きければ大きいほど、影響も大きくなります。ワークロードの設計と目的も考慮します。例えば、ソースデータストアにアクセスする機能は、データ変換や分析を行うワークロードにとって重要です。この場合、アクセス障害、スロットルアクセス、レイテンシー挿入の実験を優先させることになります。

   障害発生後の分析は、障害モードの頻度と影響の両方を理解するための良いデータソースとなります。

   割り当てた優先度を使用して、どの障害を最初に実験するか、および新しいフォールトインジェクション実験を開発する順序を決定します。

3. 実施するそれぞれの実験に関して、次の図のカオスエンジニアリングと継続的な回復力のフライホイールに従います。



Adrian Hornsby による、科学的手法を用いたカオスエンジニアリングと継続的な回復力のフライホイール。

a. 定常状態とは、正常な動作を示すワークロードの測定可能な出力であると定義します。

   ワークロードは、信頼性が高く、期待どおりに動作していれば、定常状態を示します。したがって、定常状態を定義する前に、ワークロードが正常であることを検証します。一定の割合の障害は許容範囲内である可能性があるため、定常状態は、必ずしも障害発生時にワークロードに影響が及ばないことを意味するものではありません。定常状態は実験中に観察される基準値であり、定義した仮説が次のステップで想定どおりにならない場合に、異常を強調します。

   例えば、決済システムの定常状態は、成功率 99%、往復時間 500ms で 300TPS を処理することと定義できます。

b. ワークロードがどのように障害に対応するかについての仮説を立てます。

良い仮説は、定常状態を維持するために、ワークロードがどのように障害を軽減すると予想されるかに基づいています。仮説は、特定の種類の障害が発生した場合、システムまたはワークロードが定常状態を維持することを示します。なぜならば、ワークロードは特定の緩和策を講じて設計されているからです。特定の種類の障害および緩和策は、仮説の中で特定する必要があります。

次のテンプレートを仮説に使用できます (ただし、他の表現も許容されます)。

> **Note**
>
> #####が発生した場合、#######のワークロードには、##################を維持するための###############。

例えば、次のようになります。

- Amazon EKS ノードグループの 20% のノードが停止しても、[Transaction Create API] は 100 ミリ秒未満で 99% のリクエストに対応し続けます (定常状態)。Amazon EKS ノードは 5 分以内に回復し、ポッドはスケジューリングされて、実験開始後 8 分以内にトラフィックを処理するようになります。3 分以内にアラートが発せられます。
- Amazon EC2 インスタンスの単一障害が発生した場合、注文システムの Elastic Load Balancing ヘルスチェックにより、Amazon EC2 Auto Scaling が障害インスタンスを置き換える間、Elastic Load Balancing は残りの健全なインスタンスにのみリクエストを送信し、サーバーサイド (5xx) エラーの増加を 0.01% 未満に維持します (定常状態)。
- プライマリ Amazon RDS データベースインスタンスに障害が発生した場合、サプライチェーンデータ収集ワークロードはフェイルオーバーし、スタンバイ Amazon RDS データベースインスタンスに接続して、データベースの読み取りまたは書き込みエラーを 1 分未満に維持します (定常状態)。

c. 障害を挿入して実験を実行する。

実験はデフォルトでフェイルセーフであり、ワークロードが耐えることができる必要があります。ワークロードが失敗することがわかっている場合は、実験を実行しないでください。カオスエンジニアリングは、既知の未知、または未知なる未知を見つけるために使用される必要があります。既知の未知は、認識しているが完全には理解していないモノであり、未知なる未知は、認識も完全に理解もしていないモノです。壊れているとわかっているワークロードに対して実験を行っても、新しいインサイトを得ることはできません。実験は慎重に計画し、影響の範囲を明確にし、予期せぬ乱れが発生した場合に適用できるロールバックメカニズムを用意

する必要があります。デューディリジェンスによりワークロードが実験に耐えられることがわかったら、実験を進めてください。障害を挿入するには、いくつかの方法があります。AWS のワークロードの場合、AWS FIS はアクションと呼ばれる多くの事前定義された障害シミュレーションを提供します。AWS Systems Manager ドキュメントを使用して、AWS FIS で実行するカスタムアクションを定義することもできます。

カオス実験にカスタムスクリプトを使用することは、スクリプトがワークロードの現在の状態を理解する機能を持ち、ログを出力でき、可能であればロールバックと停止条件のメカニズムを提供しない限り、お勧めできません。

カオスエンジニアリングをサポートする効果的なフレームワークやツールセットは、実験の現在の状態を追跡し、ログを出力し、実験の制御された実行をサポートするためのロールバックメカニズムを提供します。AWS FIS のように、実験範囲を明確に定義し、実験によって予期せぬ乱れが生じた場合に実験をロールバックする安全なメカニズムを備えた実験を行うことができる、確立されたサービスから始めてみてください。AWS FIS を使用したさまざまな実験については、「Resilient and Well-Architected Apps with Chaos Engineering lab」も参照してください。AWS Resilience Hub はワークロードを分析し、AWS FIS で実装および実行することを選択できる実験を作成します。

> **Note**
>
> すべての実験について、その範囲と影響を明確に理解します。本番環境で実行する前に、まず非本番環境で障害をシミュレートすることをお勧めします。

可能であれば、実験はコントロールと実験的なシステムデプロイの両方をスピンアップするカナリアデプロイを使用して、実際の負荷下の本番環境で実行する必要があります。オフピークの時間帯に実験を行うことは、本番で初めて実験を行う際に潜在的な影響を軽減するための良い方法です。また、実際の顧客トラフィックを使用するとリスクが高すぎる場合は、本稼働インフラストラクチャの制御環境と実験環境に対して合成トラフィックを使用し、実験を実行することができます。本番環境での実験が不可能な場合は、できるだけ本番環境に近い本番稼働前の環境で実験を行ってください。

実験が本稼働トラフィックや他のシステムに許容範囲を超えて影響を与えないように、ガードレールを確立してモニタリングする必要があります。停止条件を設定し、定義したガードレールのメトリクスでしきい値に達した場合は実験を停止するようにします。これには、ワークロードの定常状態のメトリクスと、障害を挿入するコンポーネントに対するメトリクスを含める必要があります。合成モニタ (ユーザー canary とも呼ばれます) は、通常、ユーザープロキ

シとして含める必要があるメトリクスの 1 つです。AWS FIS の停止条件は実験テンプレートの一部としてサポートされており、テンプレートごとに最大 5 つの停止条件を許可します。

カオスの原則の 1 つは、実験の範囲とその影響を最小化することです。

短期的な悪影響は許容する必要がありますが、実験の影響を最小化し、抑制することは、カオスエンジニアの責任であり義務です。

範囲や潜在的な影響を検証する方法として、本番環境で直接実験を行うのではなく、まず非本番環境で実験を行い、実験中に停止条件のしきい値が想定どおりに作動するか、例外をキャッチするためのオブザーバビリティがあるかどうかを確認することが挙げられます。

フォールトインジェクション実験を実行する場合は、すべての責任者に情報が十分に伝達されるようにします。オペレーションチーム、サービス信頼性チーム、カスタマーサポートなどの適切なチームとコミュニケーションをとり、実験がいつ実行され、何が期待されるかを伝えます。これらのチームには、何か悪影響が見られた場合に実験を行っているチームに知らせるための、コミュニケーションツールを提供します。

ワークロードとその基盤となるシステムを、元の既知の良好な状態に復元する必要があります。多くの場合、ワークロードの回復力のある設計が自己回復を行います。しかし、一部の障害設計や実験の失敗により、ワークロードが予期せぬ障害状態に陥ることがあります。実験が終了するまでにこのことを認識し、ワークロードとシステムを復旧させる必要があります。AWS FIS では、アクションのパラメータ内にロールバック設定 (ポストアクションとも呼ばれます) を設定することができます。ポストアクションは、ターゲットをアクションが実行される前の状態に戻します。自動化 (AWS FIS の使用など) であれ手動であれ、これらのポストアクションは、障害を検出して処理する方法を説明するプレイブックの一部である必要があります。

d. 仮説を検証する。

カオスエンジニアリングの原則は、ワークロードの定常状態を検証する方法について、以下のようなガイダンスを示しています。

システムの内部属性ではなく、測定可能な出力に焦点を当てます。その出力を短期間測定することによって、システムの定常状態のプロキシが構成されます。システム全体のスループット、エラーレート、レイテンシーのパーセンタイルはすべて、定常状態の動作を表す重要なメトリクスになり得ます。カオスエンジニアリングでは、実験中のシステムの動作パターンに焦点を当て、システムがどのように動作するかを検証するのではなく、システムが実際に動作することを検証します。

先の 2 つの例では、サーバーサイド (5xx) エラーの増加率が 0.01% 未満、データベースの読み取りと書き込みのエラーが 1 分未満という、定常状態のメトリクスが含まれています。

5xx エラーは、ワークロードのクライアントが直接経験する障害モードの結果であるため、良いメトリクスです。データベースエラーの測定は、障害の直接的な結果として適切ですが、顧客からのリクエストの失敗や、顧客に表面化したエラーなど、顧客への影響も測定して補足する必要があります。さらに、ワークロードのクライアントが直接アクセスする API や URI に、合成モニタリング (ユーザー canary とも呼ばれます) を含める必要があります。

e. 回復力を高めるためのワークロード設計を改善する。

定常状態が維持されなかった場合は、AWS Well-Architected の信頼性の柱のベスト プラクティスを適用して、障害を軽減するためにワークロードの設計をどのように改善できるかを調査します。その他のガイダンスとリソースは、「AWS ビルダーライブラリ」にあります。このライブラリには、ヘルスチェックの改善方法や、アプリケーションコードでのバックオフによる再試行方法などに関する記事がホストされています。

これらの変更を実施した後、再度実験を行い (カオスエンジニアリングフライホイールの点線で表示)、その効果を判断します。検証の結果、仮説が正しいことがわかれば、ワークロードは定常状態になり、このサイクルが続きます。

4. 実験を定期的に実施する。

カオス実験はサイクルであり、実験はカオスエンジニアリングの一環として定期的に実施する必要があります。ワークロードが実験の仮説を満たしたら、CI/CD パイプラインのリグレッション部分として継続的に実行されるように、実験を自動化する必要があります。これを行う方法については、AWS CodePipeline を使用して AWS FIS 実験を実行する方法に関するブログを参照してください。CI/CD パイプラインでの AWS FIS 反復実験に関するこのラボでは、実践的に作業できます。

フォールトインジェクション実験は、ゲームデーの一部でもあります (REL12-BP05 定期的にゲームデーを実施する を参照)。ゲームデーでは、障害やイベントをシミュレートし、システム、プロセス、チームの対応を検証します。その目的は、例外的なイベントの発生時にチームが実行することになっているアクションを実際に実行することです。

5. 実験結果をキャプチャし、保存する。

フォールトインジェクション実験の結果は、キャプチャおよび保持される必要があります。実験結果や傾向を後で分析できるように、必要なデータ (時間、ワークロード、条件など) をすべて含めておきましょう。結果の例として、ダッシュボードのスクリーンショット、メトリクスのデー

タベースからの CSV ダンプ、実験中のイベントや観察結果を手書きで記録したものなどがあります。AWS FIS を使用した実験ログ記録も、このデータキャプチャの一部です。

リソース

関連するベストプラクティス:

- REL08-BP03 デプロイの一部として回復力テストを統合する
- REL13-BP03 ディザスタリカバリの実装をテストし、実装を検証する

関連ドキュメント:

- とはAWS Fault Injection Service
- とはAWS Resilience Hub
- カオスエンジニアリングの原則
- カオスエンジニアリング: 最初の実験を計画する
- 回復力エンジニアリング: 失敗から学ぶ
- カオスエンジニアリングのストーリー
- 分散システムでのフォールバックの回避
- カオス実験のカナリアデプロイ

関連動画:

- AWS re:Invent 2020: Testing resiliency using chaos engineering (ARC316)
- AWS re:Invent 2019: Improving resiliency with chaos engineering (DOP309-R1)
- AWS re:Invent 2019: Performing chaos engineering in a serverless world (CMY301)

関連ツール:

- AWS Fault Injection Service
- AWS Marketplace: Gremlin Chaos Engineering Platform
- Chaos Toolkit
- Chaos Mesh
- Litmus

REL12-BP05 定期的にゲームデーを実施する

ワークロードに影響するイベントや障害に対応するための手順を定期的に実行するために、ゲームデーを実施します。本番稼働シナリオの処理を担当するのと同じチームを関与させます。これらの演習は、本番稼働イベントによって引き起こされるユーザーへの影響を防ぐための対策を講じるのに役立ちます。現実的な条件で対応手順を実践すると、実際のイベントが発生する前にギャップや弱点を特定して対処できます。

ゲームデーは、本番稼働のような環境でのイベントをシミュレートして、システム、プロセス、チームの応答をテストします。その目的は、例外的なイベントの発生時にチームが実行することになっているアクションを実際に実行することです。これは、改善できる箇所を把握し、組織がイベントに対応することを経験するのに役立ちます。これらは定期的に実施し、チームが対応方法を理解し、定着した習慣を構築できるようにする必要があります。

ゲームデーは、チームが本番イベントをより自信を持って処理できるように準備します。練習の行き届いたチームは、さまざまなシナリオを迅速に検出して対応できます。これにより、準備状況とレジリエンス体制が大幅に改善されます。

期待される成果: レジリエンスゲームの日数は、一貫したスケジュールに基づいて実行します。これらのゲームデーは、ビジネスを行ううえで通常かつ期待される部分と見なされます。組織には準備が文化として構築されており、本番稼働の問題が発生した場合、チームは効果的に対応し、問題を効率的に解決し、顧客への影響を軽減する準備ができています。

一般的なアンチパターン:

- 手順は文書化するものの、実行したことはない。
- テスト演習では、ビジネスの意思決定者を除外する。
- ゲームデーは実行するが、関連するすべてのステークホルダーに通知しているわけではない。
- 技術的な障害のみに焦点を当て、利害関係者は関与させない。
- ゲームデーから学んだ教訓を復旧プロセスに取り込むことはしない。
- 障害やバグについてチームを非難する。

このベストプラクティスを活用するメリット:

- 対応スキルの向上: ゲーム当日、チームはシミュレーションイベント中に職務を練習し、コミュニケーションメカニズムをテストします。これにより、本番稼働状況でより調整された効率的な対応が可能になります。

- 依存関係を特定して対処する: 複雑な環境では、さまざまなシステム、サービス、コンポーネント間で複雑な依存関係が発生することがよくあります。ゲームデーは、これらの依存関係を特定して対処し、重要なシステムとサービスがランブックの手順で適切にカバーされ、タイムリーにスケールアップまたは復旧できることを検証するのに役立ちます。
- レジリエンスの文化を育む: ゲームデーは、組織内でレジリエンスの考え方を育むのに役立ちます。部門横断的なチームやステークホルダーを関与させる場合、これらの演習は、組織全体でレジリエンスの重要性に対する認識、コラボレーション、共有理解を促進します。
- 継続的な改善と適応: 定期的なゲームデーは、回復力戦略を継続的に評価して適応させるのに役立ちます。これにより、状況の変化に直面しても、関連性と有効性が維持されます。
- システムの信頼性を高める: ゲームデーを成功させることで、システムの中断に耐え、障害から回復する能力に対する信頼を築くことができます。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

必要なレジリエンス対策を設計して実装したら、ゲームデーを実施して、すべてが本番稼働で計画どおりに機能することを確認します。ゲームデー、特に最初の日には、すべてのチームメンバーが参加し、すべての利害関係者と参加者に、日付、時刻、シミュレートされたシナリオについて事前に通知する必要があります。

ゲーム当日、関係チームは、所定の手順に従ってさまざまなイベントや潜在的なシナリオをシミュレートします。参加者は、これらのシミュレートされたイベントの影響を注意深くモニタリングし、評価します。システムが設計どおりに動作する場合、自動検出、スケーリング、自己修復メカニズムがアクティブ化され、ユーザーへの影響はほとんどまたはまったくありません。チームが悪影響に気付いた場合、テストをロールバックし、該当するランブックに文書化された自動手段または手動介入のいずれかによって、特定された問題を修正します。

レジリエンスを継続的に向上させるには、学んだ教訓を文書化し、取り入れることが重要です。このプロセスは、ゲームデーからのインサイトを体系的にキャプチャし、システム、プロセス、チームの機能を強化するために使用するフィードバックループです。

システムコンポーネントやサービスが予期せず失敗する可能性のある実際のシナリオを再現するには、ゲームデーの演習としてシミュレートされた障害を挿入します。チームは、システムの耐障害性と耐障害性をテストし、制御された環境でインシデント対応と復旧プロセスをシミュレートできます。

AWS では、Infrastructure as Code (IaC) を使用して、本番環境のレプリカを使用してゲームデーを実行できます。このプロセスを通じて、本番環境によく似た安全な環境でテストできます。さまざまな障害シナリオを作成するには、AWS Fault Injection Service を検討してください。Amazon CloudWatch や AWS X-Rayなどのサービスを使用して、ゲーム期間中のシステム動作をモニタリングします。AWS Systems Manager を使用してプレイブックを管理および実行し、AWS Step Functions を使用して定期的なゲームデーワークフローをオーケストレーションします。

実装手順

- ゲームデープログラムを確立する: ゲームデーの頻度、範囲、目的を定義する構造化プログラムを作成します。これらの演習の計画と実行には、主要な利害関係者と対象分野のエキスパートを関与させます。
- ゲームデーの準備：
  1. ゲームデーの焦点となる重要なビジネスクリティカルなサービスを特定します。これらのサービスをサポートする人材、プロセス、テクノロジーをカタログ化してマッピングします。
  2. ゲームデーのアジェンダを設定し、イベントに参加する関連チームの準備をします。計画されたシナリオをシミュレートし、適切な復旧プロセスを実行するオートメーションサービスを用意します。AWS Fault Injection Service、AWS Step Functions、AWS Systems Manager などの AWS サービスは、障害の注入や復旧アクションの開始など、ゲームデーのさまざまな側面を自動化するのに役立ちます。
- シミュレーションを実行する： ゲーム当日に、計画されたシナリオを実行します。シミュレーションされたイベントに対する人、プロセス、テクノロジーの反応を観察し、文書化します。
- 演習後のレビューを実施する： ゲームの終了後、遡及セッションを実施して学んだ教訓を確認します。改善すべき分野と、運用の回復力を向上させるために必要なアクションを特定します。検出結果を文書化し、必要な変更を追跡して、レジリエンス戦略を強化し、完了への準備を整えます。

リソース

関連するベストプラクティス:

- REL12-BP01 プレイブックを使用して障害を調査する
- REL12-BP04 カオスエンジニアリングを使用して回復力をテストする
- OPS04-BP01 主要業績評価指標を特定する
- OPS07-BP03 ランブックを使用して手順を実行する
- OPS10-BP01 イベント、インシデント、問題管理のプロセスを使用する

関連ドキュメント:

- AWS GameDay とは

関連動画:

- AWS re:Invent 2023 - Practice like you play: How Amazon scales resilience to new heights

関連する例:

- AWS ワークショップ - 嵐を乗り切る: 回復力のあるシステムの制御された混乱を解き放つ
- 独自のゲームデーを構築して運用レジリエンスをサポート

ディザスタリカバリ (DR) を計画する

バックアップと冗長ワークロードコンポーネントを配置することは、DR 戦略の出発点です。RTO と RPO は、ワークロードを回復するための目標です。これは、ビジネスニーズに基づいて設定します。ワークロードのリソースとデータのロケーションと機能を考慮して、目標を達成するための戦略を実装します。ワークロードのディザスタリカバリを提供することのビジネス価値を伝えるには、中断の可能性と復旧コストも重要な要素となります。

可用性とディザスタリカバリは、どちらも、障害のモニタリング、複数のロケーションへのデプロイ、自動フェイルオーバーなど、同じベストプラクティスに依存しています。ただし、可用性がワークロードのコンポーネントに焦点を合わせているのに対して、ディザスタリカバリはワークロード全体の個別のコピーに焦点を合わせています。ディザスタリカバリの目標はアベイラビリティーとは異なり、災害後の復旧時間が焦点です。

ベストプラクティス

- REL13-BP01 ダウンタイムやデータ消失に関する復旧目標を定義する
- REL13-BP02 復旧目標を満たすため、定義された復旧戦略を使用する
- REL13-BP03 ディザスタリカバリの実装をテストし、実装を検証する
- REL13-BP04 DR サイトまたはリージョンでの設定ドリフトを管理する
- REL13-BP05 復旧を自動化する

REL13-BP01 ダウンタイムやデータ消失に関する復旧目標を定義する

障害は、さまざまな方法でビジネスに影響を与える可能性があります。まず、障害が発生するとサービスの中断 (ダウンタイム) が発生する可能性があります。2 つ目は、障害によりデータが失われたり、一貫性がなくなったり、古くなったりする可能性があります。障害への対応方法と障害からの復旧方法をガイドするために、ワークロードごとに目標復旧時間 (RTO) と目標復旧時点 (RPO) を定義します。目標復旧時間 (RTO) は、サービスの中断から復旧までの最大許容遅延時間です。目標復旧時点 (RPO) は、最後に実行されたデータ復旧時点からの最大許容時間です。

期待される成果: すべてのワークロードには、技術的な考慮事項とビジネスへの影響に基づいて指定された RTO と RPO があります。

一般的なアンチパターン:

- 復旧目標が指定されていない。
- 任意の復旧目標を選択する。
- 過度に寛大で、ビジネス目標を満たさない復旧目標を選択する。
- ダウンタイムとデータ損失の影響を評価していない。
- 復旧時間ゼロやデータ損失ゼロなど、ワークロード設定では達成できないおそれのある非現実的な復旧目標を選択する。
- 実際のビジネス目標よりも厳格な復旧目標を選択する。これにより、ワークロードが必要とするよりもコストが高く、複雑な DR 実装を強いられている。
- 依存するワークロードの復旧目標と互換性のない復旧目標を選択する。
- 規制とコンプライアンスの要件を考慮していない。

このベストプラクティスを活用するメリット: ワークロードに RTO と RPO を設定すると、ビジネスニーズに基づいて明確で測定可能な復旧目標を確立できます。これらの目標を設定すると、それに合わせてカスタマイズされたディザスタリカバリ (DR) 計画を作成できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

ディザスタリカバリ計画の指針となるマトリックスまたはワークシートを作成します。マトリックスで、ビジネスへの影響 (重大、高、中、低など) と、それぞれにターゲットとする関連する RTO と RPO に基づいて、異なるワークロードカテゴリまたは階層を作成します。次のマトリックスは、従うことができる例を示しています (RTO と RPO の値は異なる場合があります)。

ディザスタリカバリマトリックス						
		目標復旧時点				
		1 分未満	1 時間未満	6 時間未満	1 日未満	1 日以上
目標復旧時間	10 分未満	重要	重要	高	中	中
	2 時間未満	重要	高	中	中	低
	8 時間未満	高	中	中	低	低
	24 時間未満	中	中	低	低	低
	24 時間以上	中	低	低	低	低

ディザスタリカバリマトリックスの例

各ワークロードについて、ダウンタイムとデータ損失がビジネスに与える影響を調査して理解します。影響は通常、ダウンタイムとデータ損失と共に大きくなりますが、影響の形状はワークロードタイプによって異なります。例えば、最大 1 時間のダウンタイムによる影響は小さいかもしれませんが、その後、影響が急速に拡大する可能性があります。影響には、財務上の影響 (収益の損失など)、評判上の影響 (顧客の信頼の喪失を含む)、運用上の影響 (給与の損失や生産性の低下など)、規制上のリスクなど、さまざまな形態があります。完了したら、ワークロードを適切な階層に割り当てます。

障害の影響を分析するときは、次の質問を考慮してください。

1. ビジネスに許容できない影響が生じる前にワークロードが利用できなくなる最大時間はどれくらいですか。
2. ワークロードの中断により、ビジネスにどのような影響が及ぶでしょうか。財務、評判、運用、規制など、あらゆる種類の影響を考慮してください。
3. ビジネスに許容できない影響が生じる前に、失われたり回復不能になったりする可能性のあるデータの最大量はどれくらいですか?
4. 失われたデータは、他のソース (派生データとも呼ばれます) から再作成できますか? その場合、ワークロードデータの再作成に使用されるすべてのソースデータの RPO も考慮してください。
5. このワークロードが依存するワークロード (ダウンストリーム) の復旧目標と可用性の期待値は何ですか? ワークロードの目標は、ダウンストリームの依存関係の復旧能力を考慮して達成可能である必要があります。このワークロードの復旧能力を向上させる可能性のあるダウンストリーム依存関係の回避策または軽減策を検討してください。

6. このワークロードに依存するワークロード (アップストリーム) の復旧目標と可用性の期待値は何ですか? アップストリームのワークロード目標によっては、このワークロードに、最初に想定されるよりも厳密な復旧能力が必要になる場合があります。
7. インシデントのタイプに基づいて、さまざまな復旧目標がありますか? 例えば、インシデントがアベイラビリティーゾーンまたはリージョン全体に影響するかどうかに応じて、RTO と RPO が異なる場合があります。
8. 復旧目標は、特定のイベント中または 1 年のうちに変化しますか? 例えば、ホリデーショッピングシーズン、スポーツイベント、特別セール、新製品の発売などに合わせて、異なる RTO と RPO を設定できます。
9. 復旧目標は、どのような事業部門や組織のディザスタリカバリ戦略と整合していますか?
10. 考慮すべき法的または契約上の影響はありますか? 例えば、特定の RTO または RPO でサービスを提供する契約上の義務はありますか? それらを満たさなかった場合、どのような罰則が科される可能性がありますか?
11. 規制またはコンプライアンス要件を満たすために、データ整合性を維持する必要はありますか?

次のワークシートは、各ワークロードの評価に役立ちます。このワークシートは、特定のニーズに応じて修正できます (質問を追加するなど)。

ステップ 2: 主な質問	ワークロードに適用されますか?	ワークロード RTO	ワークロード RPO	RTO 調整	RPO 調整	手順
[1] ワークロードの最大ダウン時間						サービス停止の発生から復旧までの時間で測定
[2] 失われるデータの最大量						復元可能な最後の既知のデータセットからの時間で測定
[3a] アップストリームの依存関係						最も厳しいアップストリームリカバリ目標を入力します
[3b] ダウンストリームの依存関係						最も緩いダウンストリームリカバリ目標を入力します
[3a] 調整されたアップストリームの依存関係						アップストリームの値が現在の値よりも小さく、ダウンストリームの値が大きい場
[3b] 調整されたダウンストリームの依存関係						合は、依存関係を調整し、調整した値をここに入力します
[3] 依存関係						アップストリームの依存関係を満たすために値を下げるか、ダウンストリームの依存関係の機能に基づいて値を上げます
ステップ 2: その他の質問						質問に該当する場合は、ご記入ください。該当しない場合は、飛ばしてください
ベースの RTO/RPO						上記の RTO と RPO の値をここに入れます
[4] サービス停止の種類	[ ]はい / [ ]いいえ					イベントタイプについて要件が最も厳しい復旧目標を入力します
[5] 特定の時間ベースの目標	[ ]はい / [ ]いいえ					時間について要件が最も厳しい復旧目標を入力します
[6] 顧客の混乱	[ ]はい / [ ]いいえ					ダウンタイムまたはデータ損失の関数として影響を受ける顧客をグラフ化します。これをもとに、顧客への影響を考慮して許容される最大 RTO と RPO を入力します
[7] 評判への影響	[ ]はい / [ ]いいえ					ビジネスと連携し、評判への影響を考慮した最大の RTO と RPO を決定します
[8] 運用上の影響	[ ]はい / [ ]いいえ					運用上の影響に基づいて、最大 RTO と RPO を入力します
[9] 組織の調整	[ ]はい / [ ]いいえ					LOB および組織の要件に従って、このタイプのワークロードの最大 RTO と RPO を入力します
[10] 契約上の義務	[ ]はい / [ ]いいえ					契約上の義務に基づいて、最大 RTO と RPO を入力します
[11] 規制コンプライアンス	[ ]はい / [ ]いいえ					適用される規制コンプライアンスに基づいて、最大 RTO と RPO を入力します
その他の質問に基づくターゲット						Q の 4 ～ 11 から最小値 (より厳しい値) をここに入力します
調整済みターゲット						上記の目標に対応できない場合は、ステークホルダーと協力して制約を緩め、ここに新しい最小値を入力します
調整済み RTO/RPO						ベースの RPO/RTO 値、または調整済みターゲットのいずれか低い方を入力します
ステップ 3						
事前定義されたカテゴリまたは層にマッピング						両方の値を下方 (より厳密) に調整して、最も近い定義された層に合わせます

ワークシート

実装手順

1. 各ワークロードを担当するビジネスステークホルダーと技術チームを特定し、連携します。
2. ワークロードが組織に与える影響について、重要度のカテゴリまたは階層を作成します。カテゴリの例としては、重要、高、中、低などがあります。カテゴリごとに、ビジネス目標と要件を反映する RTO と RPO を選択します。
3. 前のステップで作成したインパクトカテゴリの 1 つを各ワークロードに割り当てます。ワークロードがカテゴリにどのようにマッピングされるかを決定するには、ビジネスにとってのワークロードの重要性と中断やデータ損失の影響を考慮し、上記の質問を参考にしてください。これにより、ワークロードごとに RTO と RPO が作成されます。
4. 前のステップで決定した各ワークロードの RTO と RPO を検討します。ワークロードのビジネスチームと技術チームを関与させて、目標を調整する必要があるかどうかを判断します。例えば、ビジネスステークホルダーは、より厳格なターゲットが必要であると判断する可能性があります。あるいは、技術チームは、利用可能なリソースと技術的な制約で目標を達成できるようにターゲットを変更する必要があると判断することもできます。

リソース

関連するベストプラクティス:

- REL09-BP04 データの定期的な復旧を行ってバックアップの完全性とプロセスを確認する
- REL12-BP01 プレイブックを使用して障害を調査する
- REL13-BP02 復旧目標を満たすため、定義された復旧戦略を使用する
- REL13-BP03 ディザスタリカバリの実装をテストし、実装を検証する

関連ドキュメント:

- AWS アーキテクチャブログ: ディザスタリカバリシリーズ
- AWS でのワークロードの災害対策: クラウド内での復旧 (AWS ホワイトペーパー)
- AWS Resilience Hub による障害耐性ポリシーの管理
- APN Partner: partners that can help with disaster recovery
- AWS Marketplace: ディザスタリカバリに活用できる商品

関連動画:

- AWS re:Invent 2018: マルチリージョンアクティブ/アクティブアプリケーション用アーキテクチャパターン
- Disaster Recovery of Workloads on AWS

REL13-BP02 復旧目標を満たすため、定義された復旧戦略を使用する

ワークロードの復旧目標を満たすディザスタリカバリ (DR) 戦略を定義します。バックアップと復元、スタンバイ (アクティブ/パッシブ)、またはアクティブ/アクティブなどの戦略を選択します。

期待される成果: 各ワークロードについて、定義され、実装された DR 戦略があり、ワークロードは DR 目標を達成できます。ワークロード間の DR 戦略では、再利用可能なパターンを利用します (以前に記載された戦略など)。

一般的なアンチパターン:

- 同じような DR 目標を持つワークロードについて、一貫性のない復旧手順を実装する。
- DR 戦略は、災害が発生したときにアドホックに実装すればよいとする。
- ディザスタリカバリが計画されていない。
- 復旧時にコントロールプレーンのオペレーションに依存する。

このベストプラクティスを活用するメリット:

- 定義された復旧戦略を使用すると、一般的なツールとテスト手順を使用できます。
- 定義された復旧戦略を使用すると、チーム間のナレッジ共有と、チームが所有するワークロードでの DR の実装が改善します。

このベストプラクティスを活用しない場合のリスクレベル: 高 計画され、実装され、テストされた DR 戦略がなければ、災害発生時に復旧目標を達成できない可能性があります。

実装のガイダンス

DR 戦略は、プライマリロケーションでワークロードを実行できなくなった場合に復旧サイトでワークロードに耐えられる能力に依存します。最も一般的な復旧目標は、RTO と RPO です (「REL13-BP01 ダウンタイムやデータ消失に関する復旧目標を定義する」を参照)。

単一の AWS リージョン内の複数のアベイラビリティーゾーン (AZ) にまたがる DR 戦略は、火災、洪水、大規模な停電などの災害イベントに対して影響を緩和できます。ワークロードを特定の AWS リージョンで実行できなくなるような、可能性の低いイベントに対する保護を実装する必要がある場合には、複数のリージョンを使用する DR 戦略を使用できます。

複数のリージョンにまたがる DR 戦略を設計するときには、以下のいずれかの戦略を選んでください。戦略は、コストと複雑さが低く、かつ RTO と RPO が長い順にリストされます。リカバリリージョンとは、ワークロードに使用されるプライマリ以外の AWS リージョンを指します。

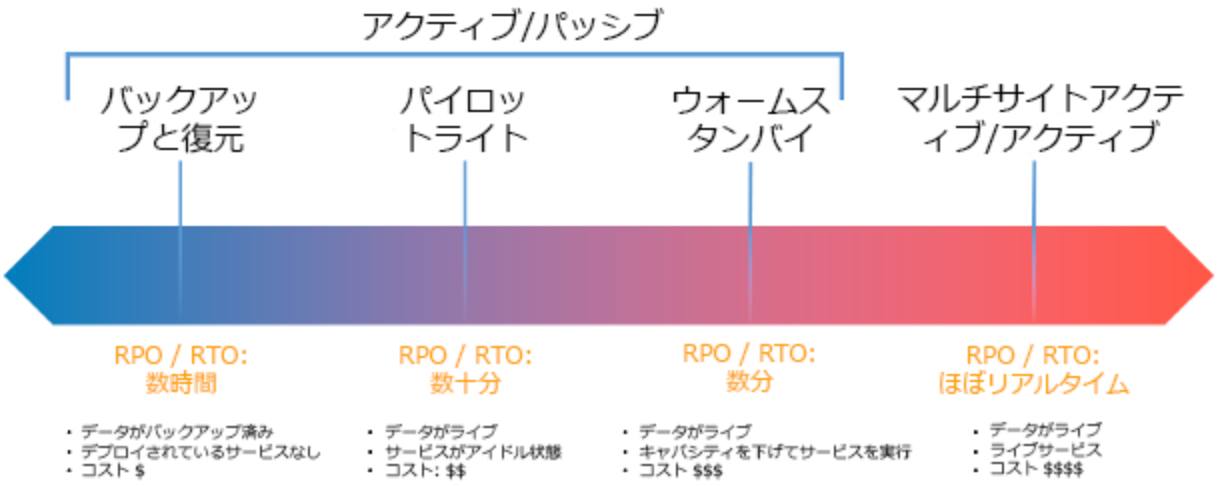


図 17: ディザスタリカバリ (DR) 戦略

- バックアップと復元 (数時間での RPO、24 時間以下での RTO): データとアプリケーションを復旧リージョンにバックアップします。自動化されたバックアップまたは連続バックアップを使用すると、ポイントインタイムリカバリ (PITR) が可能であり、場合によっては RPO を 5 分間に短縮できます。災害の際には、インフラストラクチャをデプロイし (インフラストラクチャをコードとして使用して RTO を削減)、コードをデプロイし、バックアップされたデータを復元して、復旧リージョンで災害から復旧します。
- パイロットライト (数分間の RPO、数十分間の RTO): コアワークロードインフラストラクチャのコピーを復旧リージョンにプロビジョニングします。データを復旧リージョンにレプリケートして、そこでバックアップを作成します。データベースやオブジェクトストレージなど、データのレプリケーションとバックアップのサポートに必要なリソースは、常にオンです。アプリケーションサーバーやサーバーレスコンピューティングなど、その他の要素はデプロイされませんが、必要なときには、必須の設定とアプリケーションコードで作成できます。

- ウォームスタンバイ (数秒間の RPO、数分間の RTO): 完全に機能する縮小バージョンのワークロードを復旧リージョンで常に実行している状態に保ちます。ビジネスクリティカルなシステムは完全に複製され、常に稼働していますが、フリートは縮小されています。データは復旧リージョンでレプリケートされ、使用可能です。復旧時には、システムをすばやくスケールアップして本番環境の負荷を処理できるようにします。ウォームスタンバイの規模が大きいほど、RTO とコントロールプレーンへの依存は低くなります。これを完全にスケールアップしたものはホットスタンバイと呼ばれます。
- マルチリージョン (マルチサイト) アクティブアクティブ (ゼロに近い RPO、ほぼゼロの RTO): ワークロードは複数の AWS リージョンにデプロイされ、そこからトラフィックにアクティブに対応します。この戦略では、複数のリージョン間でデータを同期する必要があります。2 つの異なるリージョンレプリカ内の同じレコードへの書き込みによって生じる矛盾を回避または処理する必要があり、これは複雑になることがあります。データレプリケーションは、データの同期に便利であり、特定のタイプの災害から保護しますが、ソリューションがポイントインタイムリカバリのためのオプションを含んでいない限り、データの破損や破壊からは保護しません。

> **Note**
>
> パイロットライトとウォームスタンバイの違いは、理解しにくいかもしれません。どちらも、プライマリリージョンアセットのコピーがある復旧リージョン内の環境を含みます。その違いは、パイロットライトが最初に追加アクションを取らなければリクエストを処理できないのに対して、ウォームスタンバイはトラフィックを直ちに (削減された能力レベルで) 処理できることです。パイロットライトでは、サーバーをオンにして、おそらく追加の (非コア) インフラストラクチャをデプロイし、スケールアップする必要があるのに対して、ウォームスタンバイでは、スケールアップするだけです (すべてが既にデプロイされ、実行しています)。RTO と RPO のニーズに基づいて、両者の中から選択してください。
> コストが懸念事項であり、ウォームスタンバイ戦略での定義と同様の RPO および RTO 目標の達成を目指す場合は、パイロットライトアプローチを採用して、改善された RPO および RTO 目標を提供する AWS Elastic Disaster Recovery などのクラウドネイティブソリューションを検討することができます。

実装手順

1. このワークロードの復旧要件を満たす DR 戦略を決定します。

DR 戦略を選ぶということは、ダウンタイムとデータ損失の削減 (RTO と RPO) 、戦略を実装するコストと複雑性のトレードオフです。必要以上に厳格な戦略の実装は、不要なコストにつながるため避けてください。

例えば、次の図では、許容可能な最大 RTO と、サービス復元戦略に費やすことができるコストの限界を決定しています。特定のビジネス目標の場合、パイロットライトまたはウォームスタンバイの DR 戦略は、RTO とコスト基準の両方を満たすことができます。

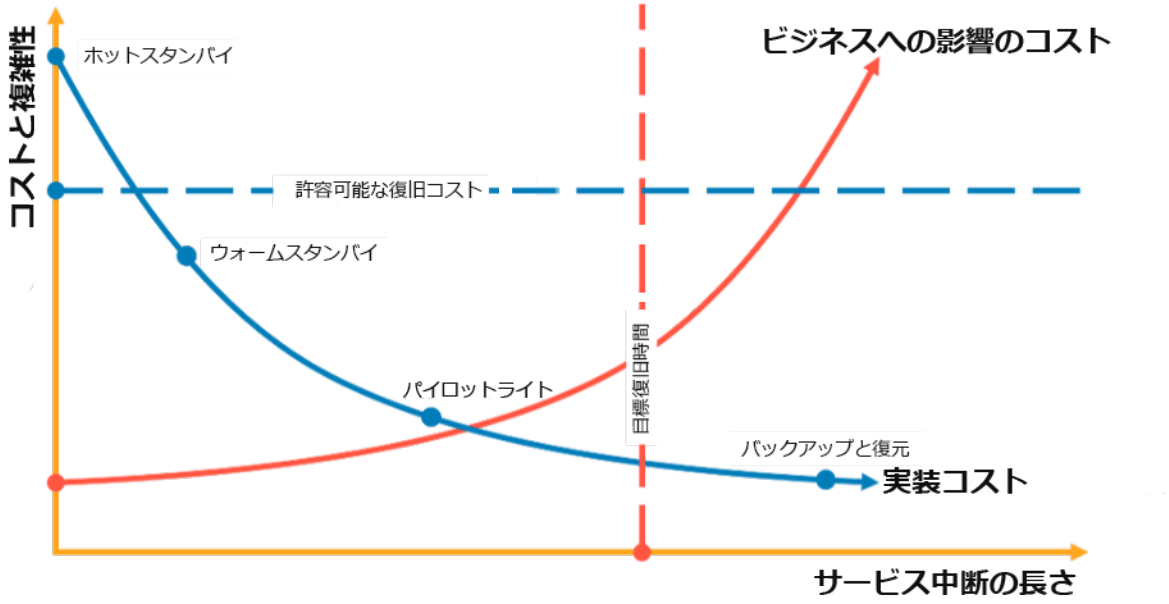


図 18: RTO とコストに基づく DR 戦略の選択を示すグラフ

詳細については、「ビジネス継続計画 (BCP)」を参照してください。

2. 選択した DR 戦略の実装パターンをレビューします。

このステップでは、選択した戦略の実装方法を理解します。戦略は、プライマリサイトと復旧サイトとしての AWS リージョンを使用して説明されています。ただし、単一リージョン内のアベイラビリティーゾーンを DR 戦略として使用することもでき、その場合は、これら複数の戦略の要素を利用します。

以下の手順では、戦略を特定のワークロードに適用できます。

バックアップと復元

バックアップと復元は、実装の複雑さが最も少ない戦略ですが、ワークロードの復元に必要な時間と労力が多く、より高い RTO と RPO につながります。常にデータのバックアップを取り、これらを別のサイト (別の AWS リージョンなど) にコピーすることをお勧めします。

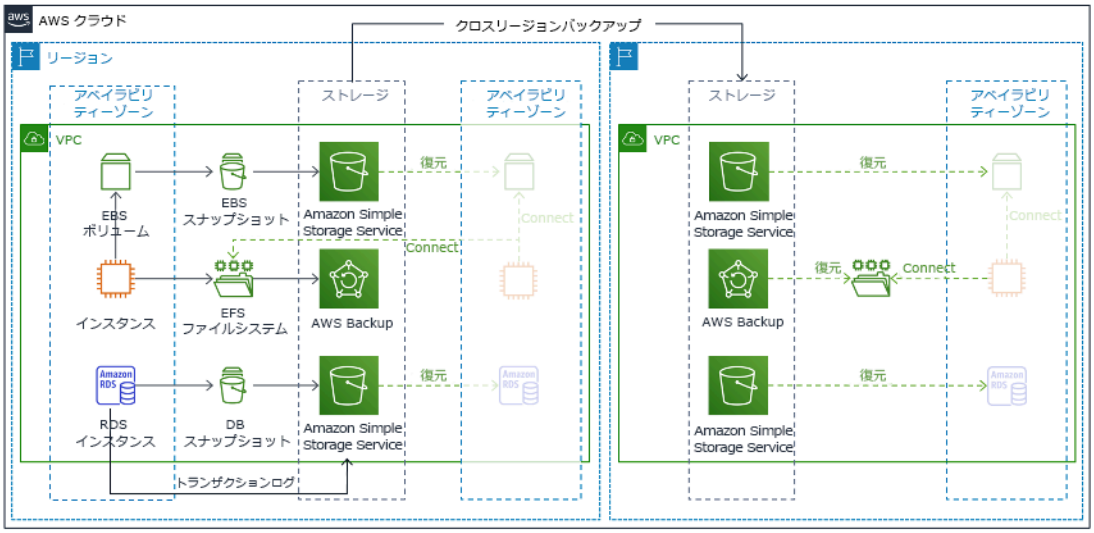


図 19: バックアップと復元アーキテクチャ

この戦略の詳細については、「AWS でのディザスタリカバリ (DR) アーキテクチャ、パート II: 迅速な復旧によるバックアップと復元」を参照してください。

パイロットライト

パイロットライトアプローチでは、プライマリリージョンから復旧リージョンにデータをレプリケートします。ワークロードインフラストラクチャに使用されるコアリソースは復旧リージョンにデプロイされますが、これを機能するスタックにするには、やはり追加のリソースと依存関係が必要です。例えば、図 20 では、コンピューティングインスタンスはデプロイされていません。

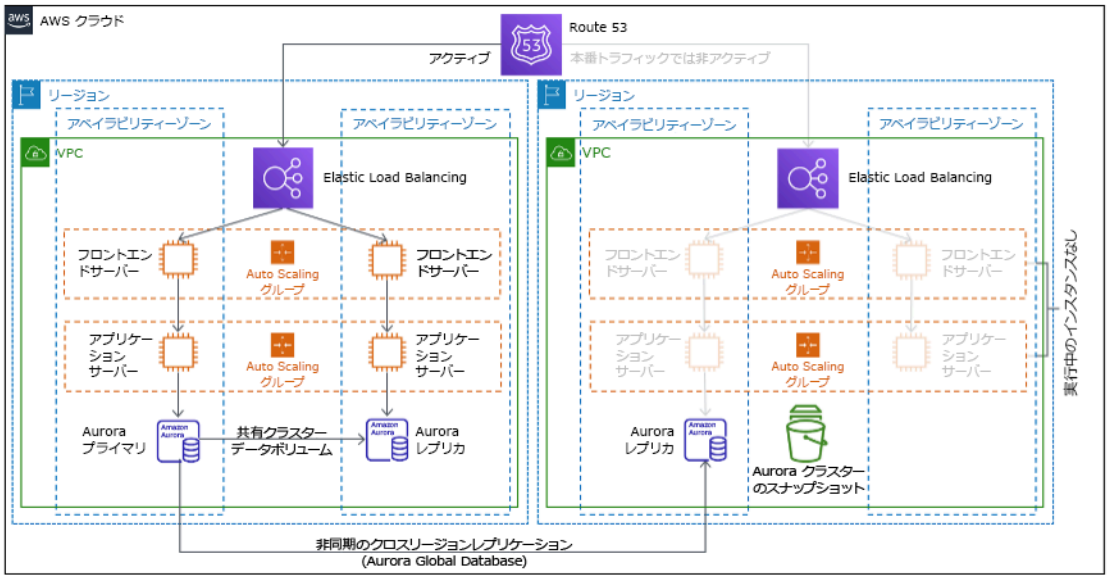


図 20: パイロットライトアーキテクチャ

この戦略の詳細については、「AWS でのディザスタリカバリ (DR) アーキテクチャ、パート III: パイロットライトとウォームスタンバイ」を参照してください。

ウォームスタンバイ。

ウォームスタンバイアプローチでは、本番稼働環境の完全に機能する縮小コピーを別のリージョンに用意します。このアプローチは、パイロットライトの概念を拡張して、ワークロードが別のリージョンに常駐するため、復旧時間が短縮されます。復旧リージョンが完全なキャパシティでデプロイされた場合は、ホットスタンバイと呼ばれます。

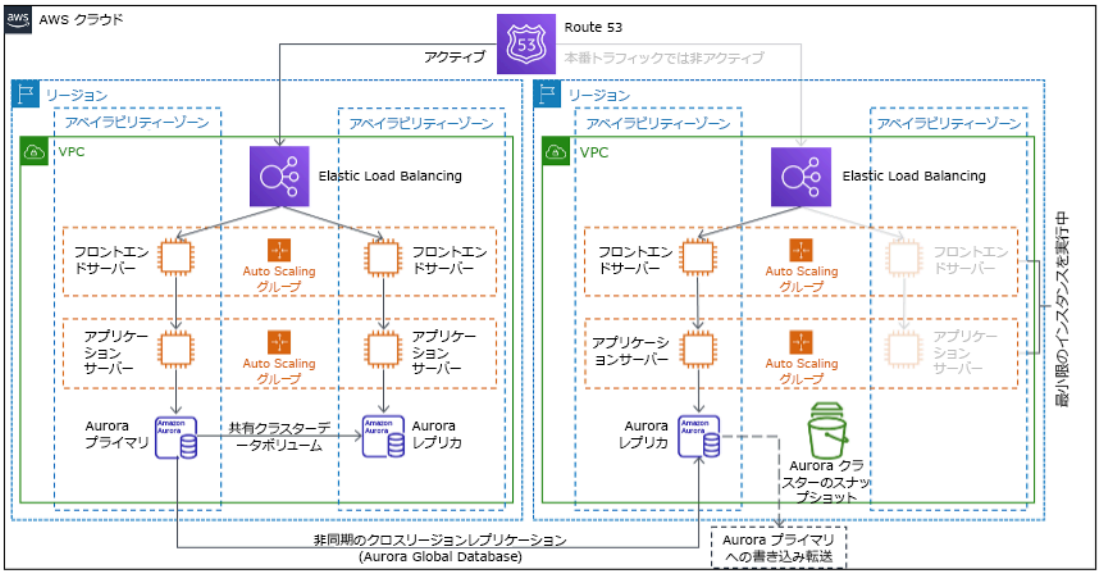


図 21: ウォームスタンバイアーキテクチャ

ウォームスタンバイまたはパイロットライトを使用するには、復旧リージョンのリソースをスケールアップする必要があります。必要に応じてキャパシティが利用可能であることを保証するには、EC2 インスタンスのキャパシティ予約の使用を検討してください。AWS Lambda を使用する場合、プロビジョニングされた同時実行はランタイム環境を提供して、関数の呼び出しにすぐに応答する準備を整えることができます。

この戦略の詳細については、「AWS でのディザスタリカバリ (DR) アーキテクチャ、パート III: パイロットライトとウォームスタンバイ」を参照してください。

マルチサイトアクティブ/アクティブ

マルチサイトアクティブ/アクティブ戦略の一部として、複数のリージョンでワークロードを同時実行できます。マルチサイトアクティブ/アクティブは、デプロイされたすべてのリージョンからのトラフィックを処理します。DR 以外の理由でこの戦略を選択することもあります。可用性を高めるためや、グローバルオーディエンスにワークロードをデプロイするときに (エンドポイントをエンドユーザーに近づけるためや、そのリージョン内のオーディエンスに対してローカライズされたスタックをデプロイするため) 使用できます。DR 戦略としては、ワークロードがデプロイされている AWS リージョンの 1 つでワークロードをサポートできない場合、そのリージョンは隔離され、残りのリージョンを使用して可用性を維持します。マルチサイトアクティブ/アクティブは、運用が最も複雑な DR 戦略であり、ビジネス要件上、必須の場合のみ選択してください。

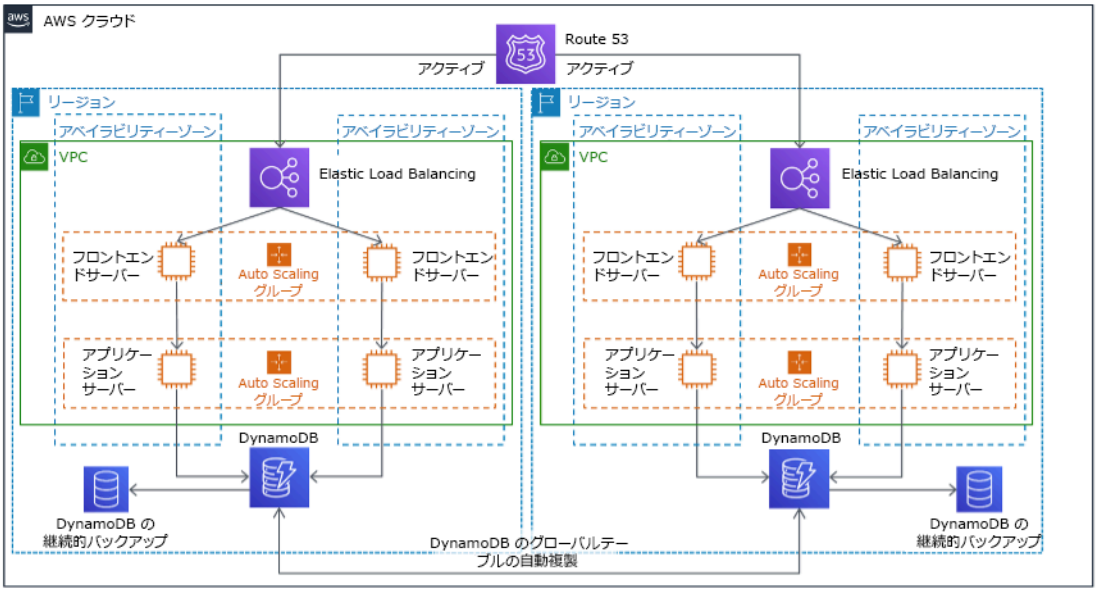


図 22: マルチサイトアクティブ/アクティブアーキテクチャ

この戦略の詳細については、「AWS のディザスタリカバリ (DR) アーキテクチャ、パート IV: マルチサイトアクティブ/アクティブ」を参照してください。

AWS Elastic Disaster Recovery

ディザスタリカバリのためのパイロットライトまたはウォームスタンバイ戦略を検討している場合、AWS Elastic Disaster Recovery はより優れた代替アプローチを提供できる場合があります。Elastic Disaster Recovery は、ウォームスタンバイと同様の RPO と RTO ターゲットを提供できますが、パイロットライトの低コストアプローチを維持します。Elastic Disaster Recovery は、プライマリリージョンからリカバリリージョンにデータをレプリケートします。継続的なデータ保護を使用して、数秒で測定される RPO と数分で測定できる RTO を実現します。パイロットライト戦略と同様、データのレプリケートに必要となるリソースのみが復旧リージョンにデプロイされるため、コストが抑えられます。Elastic Disaster Recovery では、サービスがフェイルオーバーまたはドリルの一環として開始されたコンピューティングリソースの回復を調整します。

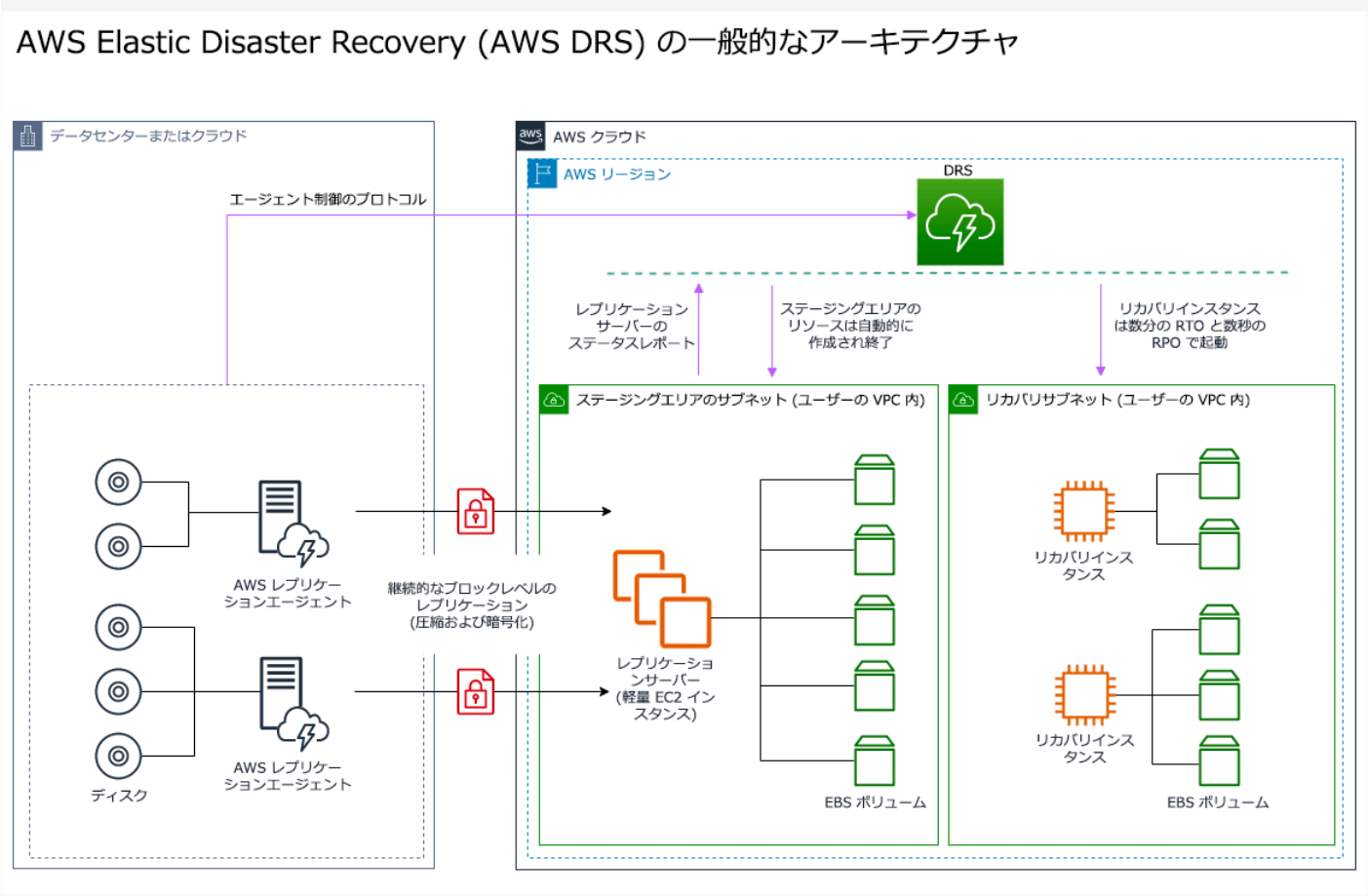


図 23: AWS Elastic Disaster Recovery アーキテクチャ

データを保護するためのその他のプラクティス

どの戦略でも、データ災害に対する緩和も必要です。連続的なデータレプリケーションは、特定のタイプの災害から保護しますが、戦略に、保存データのバージョニングまたはポイントインタイムリカバリのためのオプションが含まれていない限り、データの破損や破壊からは保護しません。復旧サイトにレプリケートしたデータもバックアップして、レプリカに加えて、ポイントインタイムバックアップを作成する必要があります。

単一の AWS リージョン内の複数のアベイラビリティーゾーン (AZ) の使用

単一のリージョン内の複数の AZ を使用する場合、DR 実装は上記の戦略の複数の要素を使用します。まず、図 23 に示されているとおり、複数の AZ を使用して、高可用性 (HA) アーキテクチャを作成する必要があります。このアーキテクチャでは、Amazon EC2 インスタンスと Elastic Load Balancer にリソースが複数の AZ にデプロイされ、リクエストがアクティブに処理されるため、マルチサイトアクティブ/アクティブアプローチを使用します。このアーキテクチャはホット

スタンバイでもあります。プライマリ Amazon RDS インスタンスが停止 (または AZ 自体が停止) すると、スタンバイインスタンスはプライマリに昇格されます。

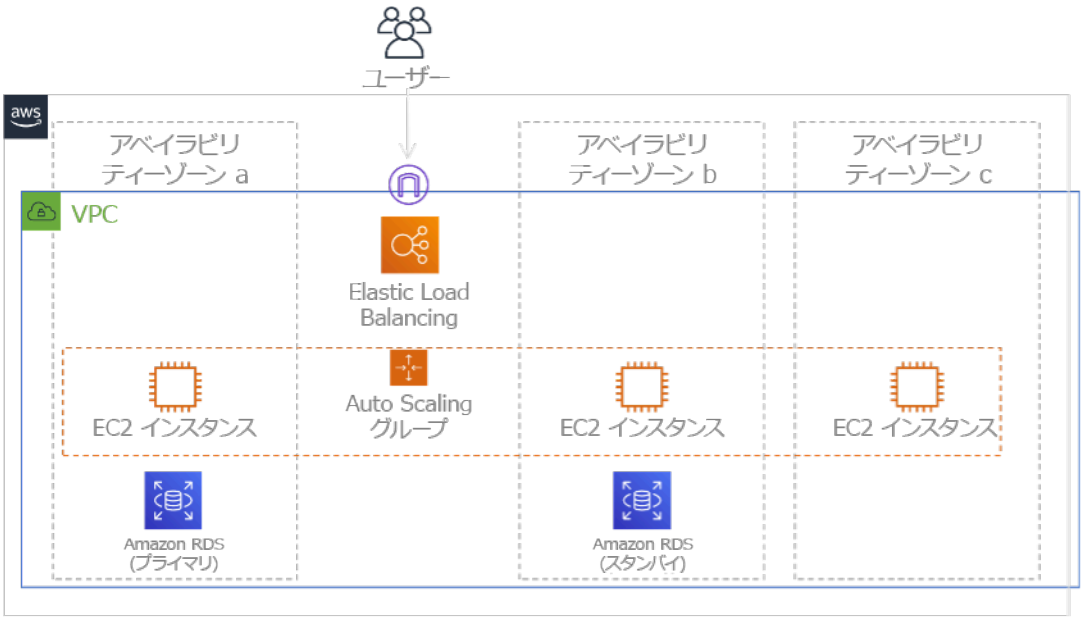


図 24: マルチ AZ アーキテクチャ

この HA アーキテクチャに加えて、ワークロードの実行に必要なすべてのデータのバックアップを追加する必要があります。これは、Amazon EBS ボリュームまたは Amazon Redshift クラスターなど、単一のゾーンに制約されるデータの場合に特に重要です。AZ に障害が発生した場合、このデータを別の AZ に復元する必要があります。可能な場合には、追加の保護層として、データバックアップも別の AWS リージョンにコピーしてください。

単一リージョンに対するあまり一般的でない代替アプローチとして、マルチ AZ DR があります。これはブログ記事「Amazon Application Recovery Controller を使用して回復力の高いアプリケーションを構築する、パート 1: 単一リージョンスタック」で説明されています。ここでは、戦略は、AZ 間の分離をできるだけ高く維持して、リージョンのように動作させることです。この代替戦略を使用すると、アクティブ/アクティブまたはアクティブ/パッシブアプローチを選ぶことができます。

> **Note**
>
> ワークロードによっては、規制によるデータレジデンシー要件があります。現在 AWS リージョンが 1 つだけの地域のワークロードにこれが該当する場合、マルチリージョンではビジネスニーズに適しません。マルチ AZ 戦略は、ほとんどの災害に対して良好な保護を提供します。

3. ワークロードのリソースと、それらの設定がフェイルオーバー前 (正常なオペレーション時) に復旧リージョンでどうなるかを評価します。

   インフラストラクチャと AWS リソースには、AWS CloudFormation などのコードとしてインフラストラクチャ、または Hashicorp Terraform などのサードパーティーツールを使用します。複数のアカウントとリージョンに単一の操作でデプロイするには、AWS CloudFormation StackSets を使用できます。マルチサイトアクティブ/アクティブとホットスタンバイ戦略の場合、復旧リージョンにデプロイされるインフラストラクチャはプライマリリージョンと同じリソースを持ちます。パイロットライトとウォームスタンバイ戦略の場合、デプロイされたインフラストラクチャを本番稼働で使用するには追加のアクションが必要です。CloudFormation パラメータおよび条件付きロジックを使用すると、デプロイされるスタックがアクティブかスタンバイかを単一のテンプレートで制御できます。Elastic Disaster Recovery を使用する場合、サービスがアプリケーション設定とコンピューティングリソースの復元をレプリケートおよび調整します。

   すべての DR 戦略では、データソースを AWS リージョン内でバックアップし、それらのバックアップをリカバリリージョンにコピーします。AWS Backup は、これらのリソースのバックアップを設定、スケジュール、モニタリングする一元的なビューを提供します。パイロットライト、ウォームスタンバイ、およびマルチサイトアクティブ/アクティブの場合、Amazon Relational Database Service (Amazon RDS) DB インスタンスまたは Amazon DynamoDB テーブルなど、プライマリリージョンのデータを復旧リージョンのデータリソースにレプリケートする必要があります。したがって、これらのデータリソースはライブであり、復旧リージョンのリクエストに対応できます。

   リージョン間での AWS サービスの運用方法の詳細については、「AWS のサービスによるマルチリージョンアプリケーションの作成」に関するブログシリーズを参照してください。

4. 必要なとき (災害発生時) に復旧リージョンをフェイルオーバーに備える方法を決定し、実装します。

   マルチサイトアクティブ/アクティブの場合、フェイルオーバーとは、リージョンを隔離して、残りのアクティブリージョンに頼ることを意味します。一般に、これらのリージョンはトラフィックを受け入れる準備ができています。パイロットライトとウォームスタンバイ戦略の場合、復旧アクションとして、図 20 の EC2 インスタンスなど、不足しているリソースやその他の不足リソースをデプロイする必要があります。

   上記の戦略のすべてで、データベースの読み取り専用インスタンスを昇格して、プライマリの読み書きインスタンスにしなければならない場合があります。

バックアップと復元の場合、バックアップからのデータの復元によって、EBS ボリューム、RDS DB インスタンス、DynamoDB テーブルなど、そのデータのリソースを作成します。インフラストラクチャを復元し、コードをデプロイする必要もあります。AWS Backup を使用して、データを復旧リージョンに復元できます。詳細については、「REL09-BP01 バックアップが必要なすべてのデータを特定してバックアップする、またはソースからデータを再現する」を参照してください。インフラストラクチャを再構築するには、Amazon Virtual Private Cloud (Amazon VPC)、サブネット、セキュリティグループに加えて、EC2 インスタンスなどのリソースの作成も必要です。復元プロセスの大部分を自動化できます。詳細については、このブログ記事を参照してください。

5. 必要なとき (災害発生時) にフェイルオーバーするトラフィックを再ルーティングする方法を決定し、実装します。

   このフェイルオーバー操作は、自動または手動で開始できます。ヘルスチェックまたはアラームに基づくフェイルオーバーの自動開始を使用するときには、不要なフェイルオーバー (誤ったアラーム) によって、使用できないデータやデータ損失などのコストが発生するため、注意が必要です。そのため、多くの場合、手動によるフェイルオーバーの開始が使用されます。この場合でも、フェイルオーバーのステップを自動化できるため、手動開始はボタンを押すようなものです。

   AWS サービスを使用するときに検討すべき、いくつかのトラフィック管理オプションがあります。1 つのオプションは、Amazon Route 53 を使用することです。Amazon Route 53 を使用すると、1 つ以上の AWS リージョンの複数の IP エンドポイントを Route 53 ドメイン名に関連付けることができます。手動で開始されるフェイルオーバーを実装するには、Amazon Application Recovery Controller を使用できます。これは、リカバリリージョンにトラフィックを再ルーティングするための高可用性データプレーン API を提供します。フェイルオーバーを実装するときには、「REL11-BP04 復旧中はコントロールプレーンではなくデータプレーンを利用する」で説明されているように、データプレーン操作を使用し、コントロールプレーンを避けてください。

   これらのオプションの詳細については、「ディザスタリカバリホワイトペーパー」のセクションを参照してください。

6. ワークロードをフェイルバックする方法のプランを設計します。

   フェイルバックとは、災害イベントの終息後、ワークロード操作をプライマリリージョンに戻すことを言います。インフラストラクチャとコードをプライマリリージョンにプロビジョニングするときには、一般に、最初に使用したのと同じステップに従い、コードとしてのインフラストラクチャとコードデプロイパイプラインに依存します。フェイルバックでの課題は、データストアを復元し、動作中の復旧リージョンとの一貫性を確認することです。

フェイルオーバー状態では、復旧リージョンのデータベースはライブであり、最新データを保持しています。目的は、復旧リージョンからプライマリリージョンへ再同期して、最新であることを確認することです。

いくつかの AWS のサービスは、これを自動的に行います。Amazon DynamoDB グローバルテーブルを使用している場合、プライマリリージョンのテーブルが使用できなくなった場合でも、オンラインに復帰すると、DynamoDB が保留中の書き込みの伝播を再開します。Amazon Aurora Global Database を使用していて、管理された計画的なフェイルオーバー を使用している場合、Aurora Global Database の既存のレプリケーショントポロジは維持されます。そのため、プライマリリージョンの以前の読み書きインスタンスがレプリカになり、復旧リージョンから更新を受け取ります。

これが自動でない場合、プライマリリージョンで復旧リージョンのデータベースのレプリカとしてデータベースを再確立する必要があります。多くの場合、これには、古いプライマリデータベースを削除して、新しいレプリカを作成する必要があります。

フェイルオーバー後、復旧リージョンでの実行を続行できる場合は、これを新しいプライマリリージョンにすることを検討してください。その場合でも、上記のすべてのステップを実行して、前のプライマリリージョンを復旧リージョンにします。一部の組織は、計画的ローテーションを実行して、プライマリリージョンと復旧リージョンを定期的に (3 か月ごとなど) 交換しています。

フェイルオーバーとフェイルバックに必要なすべてのステップをプレイブックに記載して、チームのメンバー全員が使用できるようにし、定期的にレビューする必要があります。

Elastic Disaster Recovery を使用する場合、サービスはフェイルバックプロセスの調整と自動化のサポートを提供します。詳細については、「フェイルバックの実行」を参照してください。

実装計画に必要な工数レベル: 高

リソース

関連するベストプラクティス:

- the section called “REL09-BP01 バックアップが必要なすべてのデータを特定してバックアップする、またはソースからデータを再現する”
- the section called “REL11-BP04 復旧中はコントロールプレーンではなくデータプレーンを利用する”

- the section called “REL13-BP01 ダウンタイムやデータ消失に関する復旧目標を定義する”

関連ドキュメント:

- AWS アーキテクチャブログ: ディザスタリカバリシリーズ
- AWS でのワークロードの災害対策: クラウド内での復旧 (AWS ホワイトペーパー)
- クラウド内での災害対策オプション
- Build a serverless multi-region, active-active backend solution in an hour
- Multi-region serverless backend — reloaded
- 別の AWS リージョンでのリードレプリカの作成
- Route 53: Configuring DNS Failover
- S3: クロスリージョンレプリケーション
- とはAWS Backup
- Amazon Application Recovery Controller とは
- AWS Elastic Disaster Recovery
- HashiCorp Terraform: 開始方法 - AWS
- APN Partner: partners that can help with disaster recovery
- AWS Marketplace: ディザスタリカバリに活用できる商品

関連動画:

- Disaster Recovery of Workloads on AWS
- AWS re:Invent 2018: マルチリージョンアクティブ/アクティブアプリケーション用アーキテクチャパターン (ARC209-R2)
- Get Started with AWS Elastic Disaster Recovery | Amazon Web Services

REL13-BP03 ディザスタリカバリの実装をテストし、実装を検証する

復旧サイトへの定期的なテストフェイルオーバーを実施して、適切な動作と、RTO および RPO が満たされることを確認します。

一般的なアンチパターン:

- 本番環境ではフェイルオーバーを実行しない。

このベストプラクティスを活用するメリット: ディザスタリカバリプランを定期的にテストすることで、必要なときに機能することや、チームが戦略の実行方法を把握していることを確認できます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

回避すべきパターンは、まれにしか実行されない復旧経路を作ることです。例えば、読み取り専用のクエリに使用されるセカンダリデータストアがあるとします。データストアの書き込み時にプライマリデータストアで障害が発生した場合、セカンダリデータストアにフェイルオーバーします。もしこのフェイルオーバーを頻繁にテストしない場合、セカンダリデータストアの機能に関する前提が正しくない可能性があります。セカンダリデータストアの容量は、最後にテストしたときには十分だったかもしれませんが、このシナリオでは負荷に耐えられなくなる可能性があります。エラー復旧がうまくいくのは頻繁にテストする経路のみであることは、これまでの経験からも明らかです。少数の復旧経路を用意することがベストであるのはそのためです。復旧パターンを確立して定期的にテストできます。復旧経路が複雑な場合や重大な場合に復旧経路が正常に機能するという確信を持つには、本番環境でその障害を定期的に実行する必要があります。前述の例では、その必要性に関係なく、スタンバイへのフェイルオーバーを定期的に行う必要があります。

実装手順

1. ワークロードを復旧用にエンジニアリングします。復旧経路を定期的にテストします。復旧指向コンピューティングは、回復を強化するシステムの以下の特性を特徴としています。隔離と冗長性、システム全体の変更のロールバック機能、正常性を監視し判断する機能、診断する機能、自動的な復旧、モジュラー設計、再起動する機能。復旧経路を訓練して、指定された時間内に指定された状態に復旧できるようにします。この復旧中にランブックを使用して問題を文書化し、次のテストの前に解決策を見つけます。
2. Amazon EC2 ベースのワークロードの場合、AWS Elastic Disaster Recovery を使用して DR 戦略のドリルインスタンスを実装および起動します。AWS Elastic Disaster Recovery は、ドリルを効率的に実行する機能を提供して、フェイルオーバーイベントの準備に役立つます。また、Elastic Disaster Recovery を使用すると、トラフィックをリダイレクトせずに、テストおよびドリル目的でインスタンスを頻繁に起動できます。

リソース

関連ドキュメント:

- APN パートナー: ディザスタリカバリを支援できるパートナー

- AWS アーキテクチャブログ: ディザスタリカバリシリーズ
- AWS Marketplace: ディザスタリカバリに活用できる商品
- AWS Elastic Disaster Recovery
- AWS 上のワークロードのディザスタリカバリ: クラウドにおけるリカバリ (AWS ホワイトペーパー)
- AWS Elastic Disaster Recovery フェイルオーバーの準備
- バークレー/スタンフォード大学の復旧指向コンピューティングプロジェクト
- AWS Fault Injection Simulator とは

関連動画:

- AWS re:Invent 2018: マルチリージョンアクティブ/アクティブアプリケーション用アーキテクチャパターン
- AWS re:Invent 2019: AWS のバックアップおよび復元、ディザスタリカバリソリューション

REL13-BP04 DR サイトまたはリージョンでの設定ドリフトを管理する

ディザスタリカバリ (DR) 手順を成功させるには、DR 環境がオンラインになった後、機能やデータに損失を与えることなく、ワークロードがタイムリーに通常の操作を再開できる必要があります。この目標を達成するには、DR 環境とプライマリ環境の間でインフラストラクチャ、データ、設定を一貫して維持することが重要です。

期待される成果: ディザスタリカバリサイトの設定とデータはプライマリサイトと同等であるため、必要に応じて迅速かつ完全な復旧が容易になります。

一般的なアンチパターン:

- プライマリロケーションに変更が加えられたときにリカバリロケーションを更新しないため、構成が古くなり、リカバリ作業の妨げになる可能性がある。
- プライマリロケーションとリカバリロケーション間のサービスの違いなどの潜在的な制限を考慮していないため、フェイルオーバー中に予期しない障害が発生する可能性がある。
- DR 環境の更新および同期を手動プロセスに依存しているため、ヒューマンエラーや不整合のリスクが高まる。
- 設定のドリフトを検出できないため、インシデントが発生する前に DR サイトの準備状況が誤って認識される。

このベストプラクティスを活用するメリット: DR 環境とプライマリ環境間の整合性により、インシデント後の復旧が成功する可能性が大幅に向上し、復旧手順が失敗するリスクが軽減されます。

このベストプラクティスを活用しない場合のリスクレベル: 高

実装のガイダンス

設定管理とフェイルオーバーの準備に対する包括的なアプローチは、DR サイトが一貫して更新され、プライマリサイトに障害が発生した場合に引き継ぐ準備ができていることを確認するのに役立ちます。

プライマリ環境とディザスタリカバリ (DR) 環境の一貫性を実現するには、配信パイプラインがプライマリサイトと DR サイトの両方にアプリケーションを分散していることを確認します。適切な評価期間 (時差デプロイとも呼ばれます) 後に DR サイトに変更をロールアウトして、プライマリサイトの問題を検出し、問題が広がる前にデプロイを停止します。モニタリングを実装して設定のドリフトを検出し、環境全体の変更とコンプライアンスを追跡します。DR サイトで自動修復を実行し、完全な一貫性を保ち、インシデント発生時に引き継ぐ準備を整えます。

実装手順

1. DR リージョンに、DR プランを正常に実行するために必要な AWS のサービスと機能が含まれていることを確認します。
2. Infrastructure as Code (IaC) を使用します。本番環境インフラストラクチャとアプリケーション構成テンプレートを正確に保ち、ディザスタリカバリ環境に定期的に適用します。AWS CloudFormation は、CloudFormation テンプレートで指定されている内容と実際にデプロイされている内容との間のドリフトを検出できます。
3. CI/CD パイプラインを設定して、プライマリサイトや DR サイトを含むすべての環境にアプリケーションとインフラストラクチャの更新をデプロイします。AWS CodePipeline などの CI/CD ソリューションはデプロイプロセスを自動化できるため、構成ドリフトのリスクを軽減できます。
4. プライマリ環境と DR 環境間のスタガーデプロイ。このアプローチでは、アップデートをまずプライマリ環境に展開してテストできるため、問題が DR サイトに伝播される前にプライマリサイトの問題を分離できます。このアプローチにより、欠陥が本番稼働サイトと DR サイトに同時にプッシュされるのを防ぎ、DR 環境の整合性を維持できます。
5. プライマリ環境と DR 環境の両方でリソース設定を継続的にモニタリングします。AWS Config などのソリューションは、構成のコンプライアンスを強制し、ドリフトを検出するのに役立ち、環境間で一貫した構成を維持するのに役立ちます。

6. 設定ドリフト、データレプリケーションの中断、遅延を追跡して通知するアラートメカニズムを実装します。
7. 検出された設定ドリフトの修復を自動化します。
8. プライマリ設定と DR 設定の間で継続的な整合性を検証するために、定期的な監査とコンプライアンスチェックをスケジュールします。定期的なレビューは、定義されたルールへのコンプライアンスを維持し、対処する必要がある不一致を特定するのに役立ちます。
9. AWS プロビジョニングされた容量、Service Quotas、スロットル制限、設定とバージョンの不一致をチェックします。

リソース

関連するベストプラクティス:

- REL01-BP01 Service Quotas と制約を認識する
- REL01-BP02 アカウントおよびリージョンをまたいで Service Quotas を管理する
- REL01-BP04 クォータをモニタリングおよび管理する
- REL13-BP03 ディザスタリカバリの実装をテストし、実装を検証する

関連ドキュメント:

- による非準拠 AWS リソースの修復AWS Config ルール
- AWS Systems Manager Automation
- AWS CloudFormation: スタックとリソースに対するアンマネージド型構成変更の検出
- AWS CloudFormation: CloudFormation スタック全体のドリフトを検出する
- AWS Systems Manager Automation
- AWS 上のワークロードのディザスタリカバリ: クラウドにおけるリカバリ (AWS ホワイトペーパー)
- でインフラストラクチャ設定管理ソリューションを実装するにはどうすればよいですか?AWS
- AWS Config ルール による非準拠 AWS リソースの修復

関連動画:

- AWS re:Invent 2018: マルチリージョンアクティブ/アクティブアプリケーション用アーキテクチャパターン (ARC209-R2)

関連する例:

- CloudFormation レジストリ
- AWS のクォータモニタ
- Amazon CloudWatch と AWS Lambda を使用して AWS CloudFormation の自動ドリフト修正を実装する
- AWS アーキテクチャブログ: ディザスタリカバリシリーズ
- AWS Marketplace: ディザスタリカバリに活用できる商品
- Automating safe, hands-off deployments

REL13-BP05 復旧を自動化する

信頼性、オブザーバビリティ、再現性のあるテスト済みの自動復旧メカニズムを実装して、障害のリスクとビジネスへの影響を減らします。

期待される成果: 復旧プロセスのために、十分に文書化され、標準化され、徹底的にテストされた自動化ワークフローを実装しました。リカバリオートメーションは、データ損失や利用不能のリスクが低い軽微な問題を自動的に修正します。重大なインシデントの復旧プロセスを迅速に呼び出し、プロセスの実行中に修復動作を観察し、危険な状況や障害が観察された場合はプロセスを終了できます。

一般的なアンチパターン:

- 復旧計画の一環として、障害が発生した、または劣化した状態にあるコンポーネントやメカニズムに依存する。
- 復旧プロセスに、コンソールアクセス (クリックオペレーションとも呼ばれます) などの手動による介入が必要である。
- データ損失や利用不能のリスクが高い状況では、復旧手順を自動的に開始する。
- 機能していない、または追加のリスクをもたらす復旧手順 (Andon コードや大きな赤色の停止ボタンなど) を中止するメカニズムを含めることができない。

このベストプラクティスを活用するメリット:

- 復旧オペレーションの信頼性、予測可能性、一貫性の向上。
- 目標復旧時間 (RTO) や目標復旧時点 (RPO) など、より厳格な復旧目標を達成する能力。
- インシデント発生時に復旧が失敗する可能性が低くなります。
- 人為的ミスが発生しやすい手動復旧プロセスに関連する障害のリスクを低減します。

このベストプラクティスを活用しない場合のリスクレベル: 中

実装のガイダンス

自動復旧を実装するには、AWS のサービスとベストプラクティスを使用する包括的なアプローチが必要です。まず、ワークロードの重要なコンポーネントと潜在的な障害点を特定します。人間の介入なしにワークロードとデータを障害から復旧できる自動プロセスを開発します。

Infrastructure as Code (IaC) の原則を使用してリカバリオートメーションを開発します。これで復旧環境をソース環境と一致させ、復旧プロセスをバージョン管理できます。複雑な復旧ワークフローを調整するには、AWS Systems Manager Automations や AWS Step Functions などのソリューションを検討してください。

復旧プロセスの自動化は大きなメリットをもたらし、目標復旧時間 (RTO) と目標復旧時点 (RPO) をより簡単に達成するのに役立ちます。ただし、ダウンタイムの増加やデータ損失など、予期しない状況が発生して障害が発生したり、独自の新しいリスクが発生したりする可能性があります。このリスクを軽減するには、進行中のリカバリオートメーションをすばやく停止する機能を提供します。停止したら、調査して修正のための手段を講じることができます。

サポートされているワークロードについては、AWS Elastic Disaster Recovery (AWS DRS) などのソリューションを検討して、自動フェイルオーバーを提供します。AWSDRS は、マシン (オペレーティングシステム、システム状態設定、データベース、アプリケーション、ファイルを含む) をターゲット AWS アカウント と優先リージョンのステージング領域に継続的に複製します。インシデントが発生した場合、AWS DRS はレプリケートされたサーバーを AWS のリカバリリージョンで完全にプロビジョニングされたワークロードに変換する処理を自動化します。

自動復旧のメンテナンスと改善は継続的なプロセスです。学んだ教訓に基づいて復旧手順を継続的にテストおよび改良し、復旧能力を強化できる新しい AWS のサービスや機能について最新情報を入手してください。

実装手順

1. 自動復旧の計画
    a. ワークロードアーキテクチャ、コンポーネント、依存関係を徹底的に見直し、自動復旧メカニズムを特定して計画します。ワークロードの依存関係をハード依存関係とソフト依存関係に分類します。ハード依存関係とは、ワークロードがそれなしでは動作できず、代替品を提供できない依存関係です。ソフト依存関係は、ワークロードが通常使用しているものですが、一時的な代替システムやプロセスに置き換えたり、グレースフルデグラデーションによって処理できる依存関係です。

b. 欠落または破損したデータを特定して復旧するプロセスを確立します。

c. 復旧アクションが完了した後、復旧した定常状態を確認するための手順を定義します。

d. キャッシュの事前ウォーミングや入力など、復旧したシステムをフルサービス対応にするために必要なアクションを検討してください。

e. 復旧プロセス中に発生する可能性のある問題と、それらを検出して修正する方法を検討します。

f. プライマリサイトとそのコントロールプレーンにアクセスできないシナリオを検討してください。プライマリサイトに依存することなく、復旧アクションを個別に実行できることを確認します。DNS レコードを手動でミューテーションすることなくトラフィックをリダイレクトするには、Amazon Application Recovery Controller (ARC) などのソリューションを検討してください。

2. 自動復旧プロセスの開発

a. ハンズフリーリカバリ用の自動障害検出とフェイルオーバーメカニズムを実装します。Amazon CloudWatch などのダッシュボードを構築して、自動復旧手順の進行状況と健全性について報告します。正常な復旧を検証する手順を含めます。処理中の復旧を中止するメカニズムを指定します。

b. 自動的に復旧できない障害のフォールバックプロセスとしてプレイブックを構築し、ディザスタリカバリプランを考慮します。

c. REL13-BP03 で説明されているように、復旧プロセスをテストします。

3. 復旧に備える

a. 復旧サイトの状態を評価し、重要なコンポーネントを事前にデプロイします。詳細については、「REL13-BP04」を参照してください。

b. 組織全体の関連するステークホルダーやチームが関与する復旧作業の明確なロール、責任、意思決定プロセスを定義します。

c. 復旧プロセスを開始する条件を定義します。

d. 必要に応じて、または安全と見なされた後に、復旧プロセスを元に戻してプライマリサイトにフォールバックする計画を作成します。

リソース

関連するベストプラクティス:

- REL07-BP01 リソースの取得またはスケーリング時に自動化を使用する
- REL11-BP01 ワークロードのすべてのコンポーネントをモニタリングして障害を検知する

- REL13-BP02 復旧目標を満たすため、定義された復旧戦略を使用する
- REL13-BP03 ディザスタリカバリの実装をテストし、実装を検証する
- REL13-BP04 DR サイトまたはリージョンでの設定ドリフトを管理する

関連ドキュメント:

- AWS アーキテクチャブログ: ディザスタリカバリシリーズ
- AWS 上のワークロードのディザスタリカバリ: クラウドにおけるリカバリ (AWS ホワイトペーパー)
- Amazon Route 53 ARC と AWS Step Functions を使用したディザスタリカバリオートメーションのオーケストレーション
- AWS CDK を使用して AWS Systems Manager Automation ランブックを構築する
- AWS Marketplace: ディザスタリカバリに使用できる製品
- AWS Systems Manager Automation
- AWS Elastic Disaster Recovery
- Elastic Disaster Recovery の自動フェイルオーバーとフェイルバック
- AWS Elastic Disaster Recovery リソース
- APN パートナー: ディザスタリカバリを支援できるパートナー

関連動画:

- AWS re:Invent 2018: マルチリージョンアクティブ/アクティブアプリケーション用アーキテクチャパターン (ARC209-R2)
- AWS re:Invent 2022: AWS On Air ft. AWS Elastic Disaster Recovery の AWS フェイルバック

結論

可用性と信頼性のトピックに詳しくない方も、ミッションクリティカルなワークロードの可用性を最大化するインサイトを求めている経験豊富な方も、このホワイトペーパーによって考えを刷新したり、新しいアイデアを導き出したり、新しい質問につなげたりしていただければ光栄です。これにより、ビジネスのニーズに基づいた適切な可用性のレベルと、それを実現するための信頼性を設計する方法について理解が深まることを願います。ここで提供されている設計、運用、復旧に関するレコメンデーションや、AWS ソリューションアーキテクトの知識と経験を活用することを推奨します。特に AWS で高レベルの可用性を達成したお客様の成功事例についてなど、ご意見やご感想をお待ちしております。アカウントチームに問い合わせるか、当社ウェブサイトからお問い合わせください。

寄稿者

本ドキュメントの寄稿者は次のとおりです。

- Amazon Web Services、Principal Solutions Architect、Michael Fischer
- Amazon Web Services、Principal Developer Advocate、Seth Eliot
- Amazon Web Services、Well-Architected ソリューションアーキテクト、Mahanth Jayadeva
- Amazon Web Services、プリンシパルソリューションアーキテクト、James Devine
- Amazon Web Services、Cloud Foundations シニアソリューションアーキテクト、Jason DiDomenico
- Amazon Web Services、プリンシパルソリューションアーキテクト、Marcin Bednarz
- Amazon Web Services、シニアソリューションアーキテクト、Tyler Applebaum
- Rodney Lester、プリンシパルソリューションズアーキテクト、Amazon Web Services
- Amazon Web Services、シニアソリューションアーキテクト、Joe Chapman
- Amazon Web Services、プリンシパルシステム開発エンジニア、Adrian Hornsby
- Amazon Web Services、S3 Vice President、Kevin Miller
- Amazon Web Services、プリンシパルテクニカルプログラムマネージャー、Shannon Richards
- Amazon Web Services、Fed Fin チーフテクノロジスト、Laurent Domb
- Amazon Web Services、シニアソリューションアーキテクト、Kevin Bell
- Amazon Web Services、プリンシパルクラウドレジリエンスアーキテクト、Rob Martell
- Amazon Web Services、シニアソリューションアーキテクトマネージャー DR、Priyam Reddy
- Amazon Web Services、プリンシパルテクノロジスト、Jeff Ferris
- Amazon Web Services、シニアソリューションアーキテクト、Matias Battaglia

詳細情報

詳細については、次を参照してください。

- AWS Well-Architected フレームワーク
- AWS アーキテクチャセンター

ドキュメントの改訂

このホワイトペーパーの更新に関する通知を受け取るには、RSS フィードにサブスクライブしてください。

変更	説明	日付
ベストプラクティスガイダンスの更新	ベストプラクティスは、REL 1、REL 2、REL 4、REL 6、REL 7、REL 8、REL 10、REL 12、REL 13 の各分野における新しいガイダンスで更新されました。ガイダンスが柱全体に拡張され、明確化されました。REL10-BP02 と REL12-BP03 では、ガイダンスが他のベストプラクティスに統合されています。柱全体のリソースが更新されました。	2024 年 11 月 6 日
ベストプラクティスガイダンスの更新	REL 2、4、5、6、7、8 のベストプラクティスの小規模な更新。	2024 年 6 月 27 日
ベストプラクティスガイダンスの更新	ベストプラクティスを更新して、以下の領域に関して新たなガイダンスを追加。障害を防ぐために分散システムでの操作を設計する、障害を軽減または障害に耐えるために分散システムでの操作を設計する、ワークロードリソースをモニタリングする、需要の変化に適応するようにワーク	2023 年 12 月 6 日

	ロードを設計する、変更の実装、テストの信頼性	
ベストプラクティスガイダンスの更新	ベストプラクティスを更新して、以下の領域に関して新たなガイダンスを追加。ワークロードリソースをモニタリングする、コンポーネントの障害に耐えられるようにワークロードを設計する	2023 年 10 月 3 日
ベストプラクティスガイダンスの更新	ベストプラクティスを更新して、以下の領域に関して新たなガイダンスを追加。ワークロードサービスアーキテクチャを設計する、障害を軽減または障害に耐えるために分散システムでの操作を設計する、ワークロードリソースをモニタリングする	2023 年 7 月 13 日
マイナーな更新	インクルーシブでない表現を削除。	2023 年 4 月 13 日
新しいフレームワークの更新	規範ガイダンスを使用してベストプラクティスを更新し、新しいベストプラクティスを追加。	2023 年 4 月 10 日
ホワイトペーパーの更新	新しい実装ガイダンスを使用してベストプラクティスを更新。	2022 年 12 月 15 日
マイナーな更新	図の番号を修正し、全体を通してマイナーな変更。	2022 年 11 月 17 日
ホワイトペーパーの更新	ベストプラクティスに加筆し、改善計画を追加。	2022 年 10 月 20 日

ホワイトペーパーの更新	「障害部分を切り離してワークロードを保護する」と「コンポーネントの障害に耐えられるようにワークロードを設計する」のセクションの信頼性の柱に、新しい 2 つのベストプラクティスを追加。	2022 年 5 月 5 日
ホワイトペーパーの更新	ディザスタリカバリのガイダンスを更新して、Route 53 Application Recovery Controller を追加 DevOps Guru への参照を追加。いくつかのリソースリンクの更新と編集上のマイナー変更。	2021 年 10 月 26 日
マイナーな更新	AWS Fault Injection Service (AWS FIS) に関する情報を追加。	2021 年 3 月 15 日
マイナーな更新	マイナーなテキストの更新。	2021 年 1 月 4 日

ホワイトペーパーの更新	付録 A を更新して、Amazon SQS、Amazon SNS、Amazon MQ の可用性設計目標を更新。 テーブルの行を見やすく並べなおし。可用性とディザスタリカバリの違いと、それらの回復力への貢献の説明を改善。マルチリージョンアーキテクチャ (可用性) とマルチリージョン戦略 (ディザスタリカバリ) の範囲を拡大。参照書籍を最新版に更新。可用性の計算を拡張して、リクエストベースの計算とショートカット計算を加筆。ゲームデーの説明を改善。	2020 年 12 月 7 日
マイナーな更新	付録 A を更新し、AWS Lambda の可用性設計の目標を更新	2020 年 10 月 27 日
マイナーな更新	付録 A を更新し、AWS Global Accelerator の可用性設計の目標を追加	2020 年 7 月 24 日

新しいフレームワークの更新	以下の大幅な更新とコンテンツの新規追加/改訂を実施。「ワークロードアーキテクチャ」のベストプラクティスのセクションを追加、「変更管理」と「障害管理」の各セクションのベストプラクティスを再編、リソースを更新、最新の AWS リソースおよびサービス (AWS Global Accelerator、AWS Service Quotas、AWS Transit Gateway など) を追加して更新、信頼性、可用性、回復力の定義を追加/更新、Well-Architected レビューに使用される AWS Well-Architected Tool (質問とベストプラクティス) と整合するようにホワイトペーパーを修正、設計原則を再整理、「障害から自動的に復旧する」を「復旧手順をテストする」の前に移動、図と等式のフォーマットを更新、「主なサービス」のセクションを削除し「主な AWS のサービス」の参照先をベストプラクティスに統合。	2020 年 7 月 8 日
マイナーな更新	壊れたリンクを修正	2019 年 10 月 1 日
ホワイトペーパーの更新	付録 A を更新	2019 年 4 月 1 日
ホワイトペーパーの更新	具体的な AWS Direct Connect ネットワーク推奨事項とサービス設計目標を追加	2018 年 9 月 1 日

ホワイトペーパーの更新	設計の原則と制限管理のセクションを追加。リンク更新、アップストリーム/ダウンストリームの不明瞭な用語を削除、信頼性の柱の残りのトピックの可用性のシナリオに明示的な参照を追加。	2018 年 6 月 1 日
ホワイトペーパーの更新	DynamoDB クロスリージョンソリューションを DynamoDB グローバルテーブルに変更サービス設計目標を追加	2018 年 3 月 1 日
マイナーな更新	可用性の計算を微修正してアプリケーションの可用性を追加	2017 年 12 月 1 日
ホワイトペーパーの更新	高可用性設計に関するガイダンスを更新し、概念、ベストプラクティス、実装例を追加。	2017 年 11 月 1 日
初版発行	信頼性の柱 - AWS Well-Architected フレームワークを発行しました。	2016 年 11 月 1 日

注意

お客様は、本書に記載されている情報を独自に評価する責任を負うものとします。本書は、(a) 情報提供のみを目的とし、(b) AWS の現行製品と慣行について説明しており、これらは予告なしに変更されることがあり、(c) AWS およびその関連会社、サプライヤー、またはライセンサーからの契約上の義務や保証をもたらすものではありません。AWS の製品やサービスは、明示または黙示を問わず、一切の保証、表明、条件なしに「現状のまま」提供されます。お客様に対する AWS の責任は AWS 契約によって規定されます。本書は、AWS とお客様との間で締結されるいかなる契約の一部でもなく、その内容を修正するものでもありません。

AWS 用語集

AWS の最新の用語については、「AWS の用語集リファレンス」の「AWS 用語集」を参照してください。