AWS ホワイトペーパー

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

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: AWS ホワイトペーパー

Table of Contents

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

公開日:2021 年 11 月 12 日 (ドキュメント履歴)

今日の企業は、クラウドとオンプレミスの両方で複雑な分散システムを運用しています。企業は、カスタマーにサービスを提供し、ビジネス上の成果を達成するために、これらのワークロードに回復力を持たせたいと考えています。本誌では、回復力の尺度としての可用性について一般的な理解を概説し、可用性の高いワークロードを構築するためのルールを確立し、ワークロードの可用性を向上させる方法についてガイダンスを提供します。

はじめに

可用性の高いワークロードを構築するとはどういう意味ですか。可用性をどのように測定するのですか。ワークロードの可用性を向上させるにはどうすればよいですか。このドキュメントは、このような質問に答えるのに役立ちます。3 つの主要なセクションに分かれています。最初のセクション「可用性の理解」は、主に理論上のものです。ここでは、可用性の定義とそれに影響する要因について、共通の理解を確立します。2 番目のセクション「可用性の測定」では、ワークロードの可用性を経験的に測定する方法について説明します。3 番目のセクション「可用性の高い分散システムの設計」では、最初のセクションで紹介したアイデアを実用的に応用します。AWSさらに、これらのセクション全体にわたり、本誌では、回復力のあるワークロードを構築するためのルールについて説明します。このドキュメントは、「AWS Well-Architected の信頼性の柱」に示されているガイダンスとベストプラクティスをサポートすることを目的としています。

本誌では、多くの代数数学が使用されています。重要なポイントは、数学そのものではなく、この数学が支持する概念です。とはいえ、課題を提示することも本誌の目的です。可用性の高いワークロードを運用する場合、構築したものが意図したとおりに達成されていることを数学的に証明できる必要があります。しっかりと考慮して構築された最高の設計でも、一貫して望ましい結果が得られるとは限りません。つまり、ソリューションの有効性を測定するメカニズムが必要であり、回復力が高く可用性の高い分散システムを構築して運用するには、ある程度の計算が必要です。

可用性について

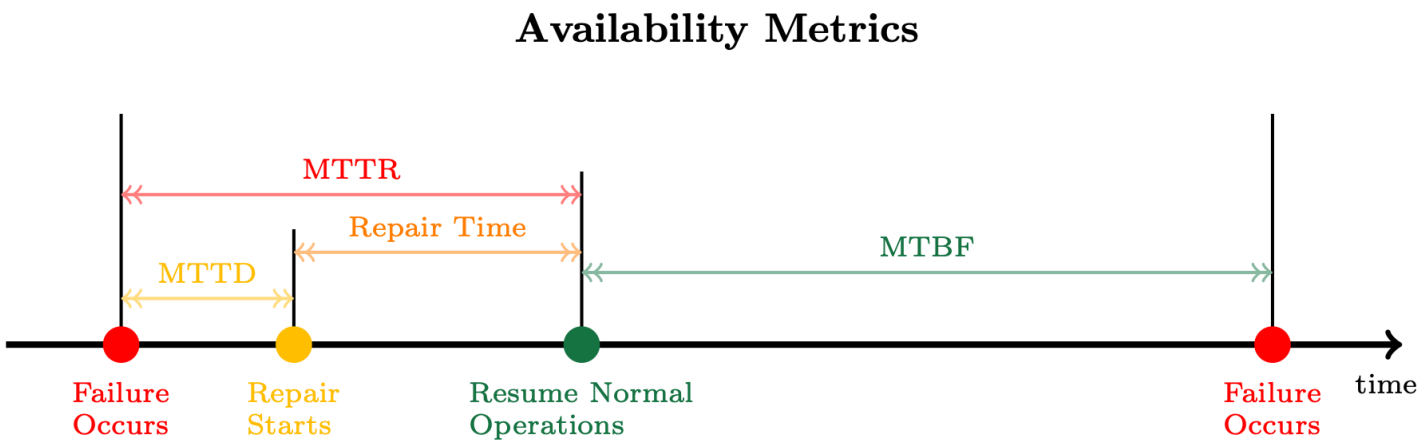
可用性は、回復力を定量的に測定する主要な方法の 1 つです。可用性 A は、ワークロードが使用可能な時間の割合として定義されます。これは、測定対象の合計時間 (予想される「稼働時間」に予想される「ダウンタイム」を加えたもの) に対する予想される (利用可能な)「稼働時間」の比率です。

$$A = \frac{uptime}{uptime + downtime}$$

方程式 1 - 可用性

この公式についてより詳しく理解するために、稼働時間とダウンタイムを測定する方法について説明します。まず知りたいのは、ワークロードがどのくらいの時間障害なく継続するかです。これを平均故障間隔 (MTBF) と呼びます。これは、ワークロードが通常動作を開始してから次の障害が発生するまでの平均時間です。次に、障害が発生してから回復するまでにどれくらいの時間がかかるかを求めます。

これを平均修理 (または回復) 時間 (MTTR) と呼びます。これは、障害が発生したサブシステムが修復されるか、またはサービスに戻る間、ワークロードが使用できない期間です。MTTR の重要な期間は、平均検出時間 (MTTD) です。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。次の図は、これらすべてのメトリクスがどのように関連しているかを示しています。



MTTD、MTTR、MTBF の関係

したがって、可用性 A は MTBF (ワークロードが稼働している時間)、MTTR (ワークロードがダウンしている時間) で表すことができます。

$$A = \frac{MTBF}{MTBF + MTTR}$$

方程式 2 - MTBF と MTTR の関係

また、ワークロードが「ダウン」している (つまり使用できない) 確率は、障害の確率 F です。

$$F = 1 - A$$

方程式 3 - 障害の確率

信頼性とは、指定された応答時間内において、要求されたときに適切な処理を実行できるワークロードの能力です。これが可用性の測定することです。ワークロードに障害が発生する頻度を減らす (MTBF が長い) か、修復時間が短くする (MTTR が短い) と、可用性が向上します。

> ルール 1
>
> 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。

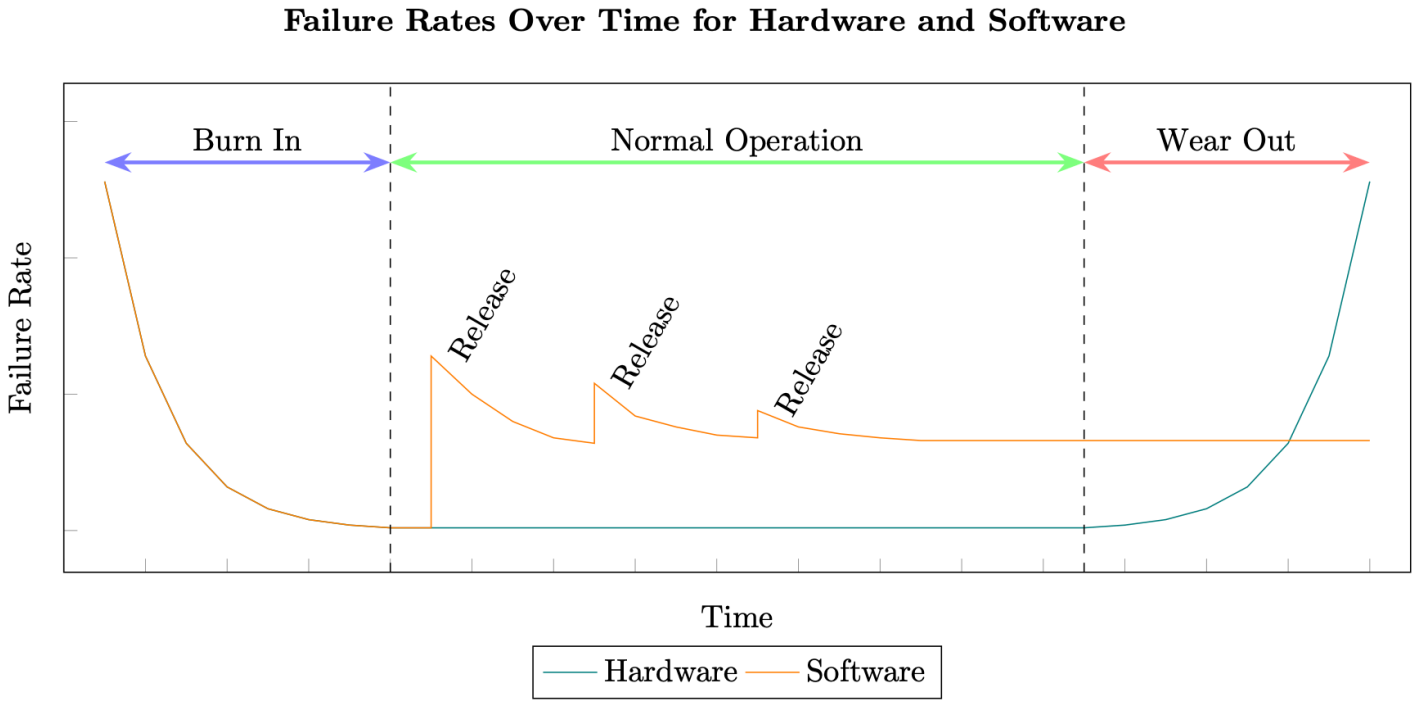
トピック

分散システムの可用性

分散システムは、ソフトウェアコンポーネントとハードウェアコンポーネントの両方で構成されています。一部のソフトウェアコンポーネントは、それ自体が別の分散システムである可能性がありま

す。基盤となるハードウェアコンポーネントとソフトウェアコンポーネントの両方の可能性は、結果として得られるワークロードの可用性に影響します。

MTBF と MTTR を使った可用性の計算は、ハードウェアシステムが基礎になっています。ただし、分散システムの障害の原因は、ハードウェアの一部で発生する障害の原因とは大きく異なります。メーカーがハードウェアコンポーネントが摩耗するまでの平均時間を一貫して計算できる場合、同じテストを分散システムのソフトウェアコンポーネントに適用することはできません。通常、ハードウェアは障害率の「バスタブ」曲線を描き、ソフトウェアは新しいリリースごとに追加される不具合によって時差曲線を描きます (「ソフトウェアの信頼性」を参照)。



ハードウェアとソフトウェアの障害率

さらに、分散システムのソフトウェアは通常、ハードウェアよりも指数関数的に高い速度で変化します。例えば、標準的な磁気ハードドライブの年間平均故障率 (AFR) は 0.93% で、実際には HDD の場合、摩耗期に達するまでに少なくとも 3 ~ 5 年の寿命があり、それよりも長い可能性があります (2020 年 Backblaze社のハードドライブのデータと統計を参照)。ハードドライブはその存続期間中に実質的に変化しません。一方、例えば Amazon は 3 ~ 5 年の間に、ソフトウェアシステムに 4 億 5,000 万から 7 億 5,000 万を超える変更を加える可能性があります。(Amazon Builders' Library — 安全で手間のかからないデプロイの自動化を参照)。

ハードウェアには「計画的陳腐化」という概念もあります。つまり、ハードウェアには寿命が組み込まれているため、一定期間が経過すると交換する必要があります。(「Great Lightbulb Conspiracy」

を参照)。ソフトウェアは、理論的にこの制約は適用されず、摩耗期間もなく、無期限に運用できます。

つまり、MTBF や MTTR の数値を取得するためにハードウェアで使用されるテストモデルや予測モデルは、ソフトウェアには適用されないということです。1970 年代以降、この問題を解決するためのモデル構築が何百回も試みられてきましたが、それらはすべて、一般的に予測モデリングと推定モデリングの 2 つのカテゴリに分類されます。(ソフトウェア信頼性モデルのリストを参照)。

したがって、分散システムの前向きな MTBF と MTTR の計算、つまり前向きな可用性の計算は、必ず何らかの予測または予測から導き出されます。これらは予測モデリング、確率的シミュレーション、履歴分析、または厳密なテストによって生成される場合がありますが、これらの計算によって稼働時間やダウンタイムが保証されるわけではありません。

過去に分散システムに障害が発生した理由は 2 度と発生しない可能性があります。将来的に発生する障害の理由は異なる可能性が高く、おそらくは不明です。また、将来の障害に備えて必要な回復メカニズムは、過去に使用されていたものとは異なり、かかる時間も大きく異なる場合があります。

また、MTBF と MTTR は平均値です。平均値と実際に表示される値には多少の差異があります (標準偏差 σ はこの変動を測定します)。したがって、実際の運用環境では、ワークロードの障害から回復までの時間が短くなったり長くなる可能性があります。

とは言え、分散システムを構成するソフトウェアコンポーネントの可用性は依然として重要です。ソフトウェアはさまざまな理由で障害を起こす可能性があり (次のセクションで詳しく説明します)、ワークロードの可用性に影響を与えます。したがって、可用性の高い分散システムでは、ハードウェアおよび外部ソフトウェアサブシステムと同様に、ソフトウェアコンポーネントの可用性の計算、測定、および向上に重点を置く必要があります。

> ルール 2
>
> ワークロード内のソフトウェアの可用性は、ワークロード全体の可用性にとって重要な要素であり、他のコンポーネントと同様に重視する必要があります。

MTBF と MTTR は分散システムでは予測が難しいものの、それでも可用性を向上させるための重要な分析情報が得られることに注意する必要があります。障害の頻度を減らす (MTBF を高くする) ことと、障害発生後の回復までの時間を短くする (MTTR を短くする) ことの両方が、経験的な可用性の向上につながります。

分散システムにおける障害の種類

分散システムで一般的に可用性に影響するバグには、ボーアバグとハイゼンバグ (「A Conversation with Bruce Lindsay」ACM Queue vol. 2、no. 8 – November 2004 参照) の 2 つのクラスがあります。

ボーアバグは、繰り返し発生するソフトウェアの問題です。同じ入力が与えられると、バグは常に同じ誤った出力を生成します (決定論的なボーアの原子模型のように、確実かつ簡単に検出できます)。この種のバグが発生するのは、ワークロードが本番環境に移行する頃までは稀です。

ハイゼンバグは一時的なバグです。つまり、特定の稀な状況でのみ発生するバグです。これらの条件は通常、ハードウェア (例えば、一時的なデバイス障害やレジスタサイズなどのハードウェア実装の仕様)、コンパイラの最適化と言語の実装、制限条件 (例えば、一時的にストレージが不足している)、競合条件 (例えば、マルチスレッドオペレーションにセマフォを使用しない) などに関連しています。

ハイゼンバグは本番環境のバグの大部分を占めており、見つけにくく、観察やデバッグを試みると動作が変わったり、消失するように見えるため、見つけるのが困難です。ただし、プログラムを再起動すると、動作環境が若干異なり、ハイゼンバグの原因となった状況が解消されるため、失敗した動作が成功する可能性があります。

したがって、本番環境でのほとんどの障害は一時的なものであり、動作を再試行しても再び障害が発生する可能性はほとんどありません。回復力を高めるには、分散システムがハイゼンバグに対する耐障害性を備えている必要があります。これを実現する方法については、「分散システムの MTBF の増加」のセクションで説明します。

依存関係のある可用性

前のセクションでは、ハードウェア、ソフトウェア、その他の分散システムはすべてワークロードのコンポーネントであると述べました。これらのコンポーネントを依存関係と呼びます。これは、ワークロードがその機能を提供するために依存するものです。ハードな依存関係とは、それなしではワークロードが機能しないものですが、ソフトな依存関係は、利用できないことに気付かれなかったり、一定期間許容されることがあります。ハードな依存関係は、ワークロードの可用性に直接影響します。

そのため、ワークロードの理論上の最大可用性を計算してみることをお勧めします。これは、ソフトウェア自体を含むすべての依存関係の可用性の積です (α_n は 1 つのサブシステムの可用性)。というのは、各依存関係は動作可能でなければならないからです。

$$A = \alpha_1 \times \alpha_2 \times \ldots \times \alpha_n$$

方程式 4 - 理論上の最大可用性

これらの計算に使用される可用性の数値は、通常 SLA やサービスレベル目標 (SLO) などに関連付けられています。SLA は、顧客が受けると予想されるサービスのレベル、サービスを測定するためのメトリクス、およびサービスレベルが達成されなかった場合の是正措置またペナルティ (通常は金額) を定義します。

上記の式を使うと、純粋に数学的に見れば、ワークロードは、その依存関係以の可用性は達成できないと結論づけることができます。しかし実際に一般的に見られるのは、そうではないということです。99.99% の可用性の SLA で 2 つまたは 3 つの依存関係を使用して構築されたワークロードでも、それ自体でも 99.99% 以上の可用性を達成できます。

これは、前のセクションで概説したように、これらの可用性の数値が推定値であるためです。これらは、障害がどのくらいの頻度で発生し、どれだけ早く修復できるかを推定または予測します。ダウンタイムを保証するものではありません。依存関係は、定められた可用性 SLA または SLO を頻繁に超過します。

また、依存関係は、パブリック SLA で定められている数値よりも、パフォーマンスを目標とする内部可用性目標が高い場合もあります。これにより、不明または不明の可能性がある事態が発生した場合に、SLA を満たすリスクをある程度軽減できます。

最後に、ワークロードには、SLA が不明であったり、SLA や SLO が提供されていない依存関係がある可能性があります。例えば、世界中のインターネットルーティングは、多くのワークロードにとって共通の依存関係となっていますが、グローバルトラフィックがどのインターネットサービスプロバイダーを使用しているか、SLA があるかどうか、プロバイダー間での一貫性のレベルについて知ることは困難です。

つまりこれらすべてが示唆するのは、理論上の最大可用性を計算しても、概算しか行われない可能性が高く、それだけでは正確ではなく、意味のある洞察も得られないということです。この計算からわかるのは、ワークロードが依存する要素が少ないほど、障害が発生する可能性が全体的に低くなるということです。1 未満の数の乗算が少なければ少ないほど、結果の値は大きくなります。

ルール 3

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。

この計算は、依存関係の選択プロセスの参考にもなります。選択プロセスは、ワークロードの設計方法、依存関係の冗長性を活用して可用性を向上させる方法、およびそれらの依存関係をソフトと見なすかハードと見なすかに影響します。ワークロードに影響を与える可能性のある依存関係は慎重に選択する必要があります。次のルールでは、その方法について説明します。

ルール 4

一般的には、可用性の目標がワークロードの目標と同等以上の依存関係を選択します。

冗長性を実装した可用性

ワークロードが複数の独立した冗長サブシステムを利用する場合、単一のサブシステムを使用する場合よりも高いレベルの理論上の可用性を実現できます。例えば、2 つの同一のサブシステムで構成されるワークロードについて考えてみましょう。サブシステム 1 またはサブシステム 2 のいずれかが動作していれば、完全に動作可能です。システム全体がダウンするには、両方のサブシステムが同時にダウンしている必要があります。

1 つのサブシステムの故障率が 1 − α の場合、2 つの冗長サブシステムが同時に停止する確率は、各サブシステムの故障率の積であり、$F = (1- \alpha_1) \times (1- \alpha_2)$ です。2 つの冗長サブシステムがあるワークロードの場合、式 (3) を使用すると、可用性は次のように定義されます。

$$A = 1 - F$$
$$F = (1 - \alpha_1) \times (1 - \alpha_2)$$
$$A = 1 - (1 - \alpha)^2$$

方程式 5

したがって、可用性が 99% の 2 つのサブシステムの場合、一方のサブシステムが故障する確率は 1% で、両方に障害が発生する確率は (1− 99%) × (1− 99%) = 0.01% です。これにより、2 つの冗長サブシステムを使用した場合の可用性が 99.99% になります。

これを一般化して、追加の冗長スペア s を組み込むこともできます。式 (5) では、1 つのスペアのみを想定していますが、複数のサブシステムが同時に消失した場合でも、可用性に影響を与えないように、1 つのワークロードに 2 つ、3 つ、またはそれ以上のスペアがある場合もあります。ワークロードに 3 つのサブシステムがあり、2 つがスペアである場合、3 つのサブシステムすべてが同時に障害

を起こす確率は (1−α) × (1−α) × (1−α) または (1− α) ^{3} です。一般に、s 個のスペアがあるワークロードは、s + 1 個のサブシステムに障害が発生した場合にのみエラーとなります。

n 個のサブシステムと s 個のスペアがあるワークロードの場合、f は障害モードの数、つまり n 個のうち s + 1 個のサブシステムで障害が発生する回数です。

これは事実上、二項定理、つまり n の集合から k 個の要素を選択する、つまり 「n が k を選択」 する組み合わせ論です。この場合、k は s + 1 です。

$$k = s + 1$$

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

$$\binom{n}{s+1} = \frac{n!}{(s+1)!\,\big(n-(s+1)\big)!}$$

$$f = \frac{n!}{(s+1)!\,(n-s-1)!}$$

方程式 6

次に、障害モードの数とスペアリングを組み込んだ一般的な可用性の近似値を生成できます。(これが概算である理由については、付録 2 のHighleymanらの「Breaking the Availability Barrier」を参照してください)。

$$s = \ Number\ of\ spares$$
$$\alpha = Availability\ of\ subcomponent$$
$$f = Number\ of\ failure\ modes$$
$$A = 1 - F \approx 1 - f(1-\alpha)^{s+1}$$

方程式 7

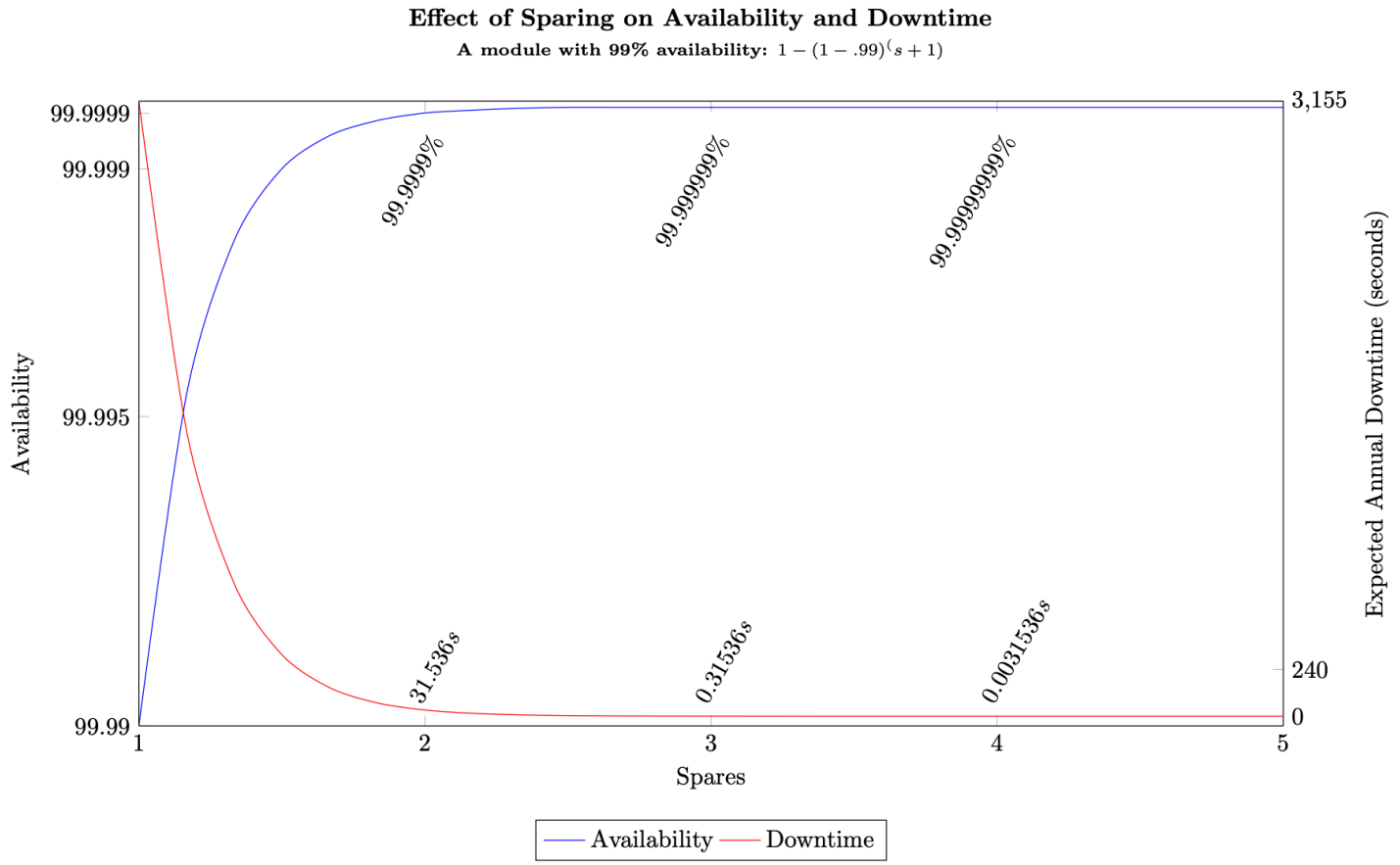
スペアリングは、独立して障害が発生するリソースを提供するあらゆる依存関係に適用できます。例えば、さまざまな AZ の Amazon EC2 インスタンス、またはさまざまな AWS リージョン の Amazon S3 バケットなどがあります。スペアを使用すると、その依存関係によって全体的な可用性が向上し、ワークロードの可用性目標を支援できるようになります。

> ルール 5
>
> スペアリングを使用すると、ワークロード内の依存関係の可用性が向上します。

ただし、スペアリングにはコストがかかります。スペアを追加するたびに元のモジュールと同じコストがかかるため、コストは少なくとも直線的に増加します。また、スペアを使用できるワークロードを構築すると、その複雑さも増します。依存関係の障害を特定し、その障害を原因とする作業を健全なリソースに振り分け、ワークロードの全体的なキャパシティを管理する方法について理解している必要があります。

冗長性は、最適化の問題です。スペアの数が少なすぎると、必要以上に頻繁にワークロードに障害が発生する可能性があります。また、スペアが多すぎると、ワークロードの実行コストが増加しすぎます。保証される追加の可用性よりもスペアの追加コストがかかるというしきい値が存在します。

スペアの場合の一般的な可用性の式、式 (7) を使用すると、可用性が 99.5% のサブシステムで、スペアが 2 台ある場合、ワークロードの可用性は $A \approx 1 - (1)(1-.995)^{3} = 99.9999875\%$ (年間約 3.94 秒のダウンタイム) となり、スペアが 10 台の場合、$A \approx 1 - (1)(1-.995)^{11} = 25.5\ 9's$ (おおよそのダウンタイムは、1 年あたり 1.26252×10^{-15}ms となり、実質的には 0 秒になります)。これら 2 つのワークロードを比較すると、スペアにかかるコストが 5 倍に増加し、ダウンタイムが 1 年で 4 秒短縮されました。ほとんどのワークロードでは、この程度の可用性の向上で、このコストの増加は正当化されません。次の図は、この関係を示しています。



スペアリングの増加によるリターンの減少

スペアが 3 台以上になると、1 年間に予想されるダウンタイムがわずか 1 秒未満になります。つまり、この時点以降は、収益が減少する領域に到達することになります。より高いレベルの可用性を実現するために「もっと追加したい」という衝動に駆られるかもしれませんが、実際には、コスト面でのメリットはすぐに消失します。サブシステム自体の可用性が 99% 以上であれば、スペアを 3 台以上使用しても実質的なメリットは得られません。

> **ルール 6**
>
> スペアリングのコスト効率には上限があります。必要な可用性を実現するには、必要最小限のスペアのみを使用します。

正しい数のスペアを選択するには、障害の単位について考慮する必要があります。例えば、ピークキャパシティを処理するために 10 の EC2 インスタンスが必要で、それらが 1 つの AZ にデプロイされているワークロードについて検討します。

AZ は障害隔離境界として設計されているため、障害の単位は 1 つの EC2 インスタンスだけではありません。AZ 全体に相当する EC2 インスタンスが同時に障害を起こす可能性があるためです。この場合は、別の AZ に冗長性を追加して、AZ に障害が発生した場合の負荷を処理する EC2 インスタンスを 10 個追加し、合計 20 個の EC2 インスタンスをデプロイします (静的安定性のパターンに従います)。

これはスペア EC2 インスタンスが 10 個あるように見えますが、実際にはスペア AZ が 1 つしかないため、収益が減少するポイントを超えていません。ただし、3 つの AZ を利用し、AZ ごとに 5 つの EC2 インスタンスをデプロイすると、可用性を高めると同時に、コスト効率をさらに高めることができます。

これにより、1 つのスペア AZ で合計 15 個の EC2 インスタンス (2 つの AZ で 20 個のインスタンスではなく) を提供し、1 つの AZ に影響するイベントが発生した場合にピークキャパシティに対応するために必要な合計 10 個のインスタンスを引き続き確保できます。そのため、ワークロード (インスタンス、セル、AZ、リージョン) が使用するすべての障害隔離境界で耐障害性を確保できるよう、スペアリングを組み込む必要があります。

CAP の定理

可用性について考えるもう 1 つの方法は、CAP 定理との関係です。この定理では、データを格納する複数のノードで構成される分散システムでは、次の 3 つの保証のうち 2 つ以上を同時に提供することはできません。

- C (一貫性): 一貫性が保証されない場合、読み取りリクエストはすべて最新の書き込みまたはエラーを受け取ります。
- A (可用性): ノードがダウンしていたり利用できなくなったりしても、すべてのリクエストはエラー以外の応答を受け取ります。
- P (耐パーティション): ノード間で任意の数のメッセージが失われても、システムは動作し続けます。

(詳細については、セス・ギルバートとナンシー・リンチ、「Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services」、ACM SIGACT News、第 33 巻 2 号 (2002)、51 ~ 59 ページを参照してください。)

ほとんどの分散システムは、ネットワーク障害に耐える必要があるため、ネットワークパーティションを許可する必要があります。つまり、これらのワークロードは、ネットワークパーティションが発生したときに一貫性と可用性のどちらかを選択する必要があります。ワークロードが可用性を選択した場合、常に応答が返されますが、データに一貫性がない可能性があります。一貫性を選択した場

合、ワークロードがデータの一貫性を確認できないため、ネットワークパーティションでエラーを返す可能性があります。

より高いレベルの可用性を提供することを目標とするワークロードでは、ネットワークパーティション中にエラーが返される (使用できない) ことを防止するため、可用性と耐パーティション (AP) を選択する場合があります。その結果、最終的な整合性または単調な整合性など、より緩やかな整合性モデルが必要になります。

耐障害性と障害隔離

可用性について考えるとき、これらの 2 つは重要な概念です。耐障害性とは、サブシステムの障害に耐え、可用性を維持する (確立された SLA の範囲内で適切な処理を行う) 能力です。フォールトトレランスを実装するために、ワークロードはスペア (または冗長) サブシステムを使用します。冗長セット内のサブシステムの 1 つに障害が発生すると、通常はほぼシームレスに、別のサブシステムが処理を引き継ぎます。この場合、スペアは真のスペアキャパシティであり、障害が発生したサブシステムの処理を 100% 引き継ぐことができます。真のスペアでは、ワークロードに悪影響を及ぼすには、複数のサブシステムで障害が発生する必要があります。

障害分離により、障害発生時の影響範囲を最小限に抑えることができます。これは通常、モジュール化によって実装されます。ワークロードは小さなサブシステムに分割され、それぞれ個別に障害が発生し、個別に修復できます。モジュールの障害は、モジュールを超えて拡散することはありません。この考えは、ワークロード内のさまざまな機能にまたがる垂直方向と、同じ機能を提供する複数のサブシステムにわたる水平方向の両方に及びます。これらのモジュールは、イベント中の影響範囲を制限する障害コンテナとして機能します。

コントロールプレーン、データプレーン、静的安定性のアーキテクチャパターンは、耐障害性と障害分離の実装を直接的にサポートします。Amazon Builders' Library の記事「可用性ゾーンを使用した静的安定性」には、これらの用語の適切な定義と、耐障害性が高く可用性の高いワークロードの構築にどのように適用されるかについて記載されています。このホワイトペーパーでは、「AWS での高可用性分散システムの設計」のセクションでこれらのパターンを使用しており、その定義についてもここにまとめています。

- コントロールプレーン — ワークロードのうち、リソースの変更 (追加、削除、修正) を行い、これらの変更を必要な場所に反映することを担当する部分です。コントロールプレーンは通常、データプレーンよりも複雑で可動部分が多いため、統計的に故障する可能性が高く、可用性が低くなります。
- データプレーン — 日常業務機能を提供するワークロードの一部です。データプレーンは、コントロールプレーンよりもシンプルで大量に動作する傾向があるため、可用性が高くなります。

- 静的安定性 — 依存関係が損なわれてもワークロードが正常に動作し続ける能力。実装方法の 1 つは、データプレーンからコントロールプレーンの依存関係を削除することです。また、ワークロードの依存関係を疎結合する方法もあります。ワークロードには、依存関係が提供しているはずの更新情報 (新しい情報、削除された情報、変更された情報など) が表示されない可能性があります。ただし、依存関係が損なわれる前に行っていたことはすべて機能し続けています。

ワークロードの障害について考えるとき、回復について検討できる大まかなアプローチが 2 つあります。1 つ目の方法は、障害が発生した後に、AWS Auto Scaling を使用して新しい容量を追加してそれに対応することです。2 つ目の方法は、そうした障害が発生する前に備えることです。例えば、ワークロードのインフラストラクチャをオーバープロビジョニングして、追加のリソースを必要とせずに正常に動作し続けることができるようにします。

静的に安定したシステムでは、後者のアプローチを使用します。障害発生時にも利用可能なスペアキャパシティを事前にプロビジョニングします。この方法では、障害から回復するための新しいキャパシティをプロビジョニングする際に、ワークロードのリカバリパスにあるコントロールプレーンに依存関係を構築しません。また、さまざまなリソースに新しいキャパシティをプロビジョニングするには時間がかかります。新しいキャパシティを待っている間に既存の需要によってワークロードが過負荷になり、さらにパフォーマンスが低下して、「ブラウンアウト」が発生したり、可用性が完全に喪失する可能性があります。ただし、事前にプロビジョニングされたキャパシティを利用することによるコストへの影響と可用性目標を比較して検討する必要もあります。

静的安定性では、高可用性ワークロードに対して次の 2 つのルールがあります。

ルール 7

特にリカバリ中は、データプレーンのコントロールプレーンに依存関係を作成しません。

ルール 8

可能な限り、依存関係が損なわれてもワークロードが正しく動作できるように、依存関係を疎結合します。

可用性の測定

前に説明したとおり、分散システムの将来を見据えた可用性モデルを作成することは困難で、必要な洞察が得られない場合があります。そこで有用なのは、ワークロードの可用性を測定する一貫した方法を開発することです。

可用性を稼働時間とダウンタイムとして定義すると、障害はバイナリーオプションとして表されます。つまりワークロードが稼働しているか、稼働していないかのどちらかです。

ただし、これは稀なケースです。障害にはある程度の影響があり、多くの場合、ワークロードの一部で発生し、一部のユーザーまたはリクエスト、一部の場所、または一部のレイテンシに影響を与える場合があります。これらはすべて部分的障害モードです。

また、MTTR と MTBF は、何がシステムの可用性に影響するのか、そしてそれをどのように改善するのかを理解するのに役立ちますが、その有用性は可用性の経験的な尺度としては役に立ちません。さらに、ワークロードは多くのコンポーネントで構成されています。例えば、支払い処理システムなどのワークロードは、多くのアプリケーションプログラミングインターフェイス (API) とサブシステムで構成されています。そこで、「ワークロード全体の可用性はどの程度か」というような質問は、実際には複雑で微妙な質問となります。

このセクションでは、可用性を経験的に測定する 3 つの方法、つまりサーバー側のリクエスト成功率、クライアント側のリクエスト成功率、および年間ダウンタイムについて説明します。

サーバー側とクライアント側のリクエスト成功率

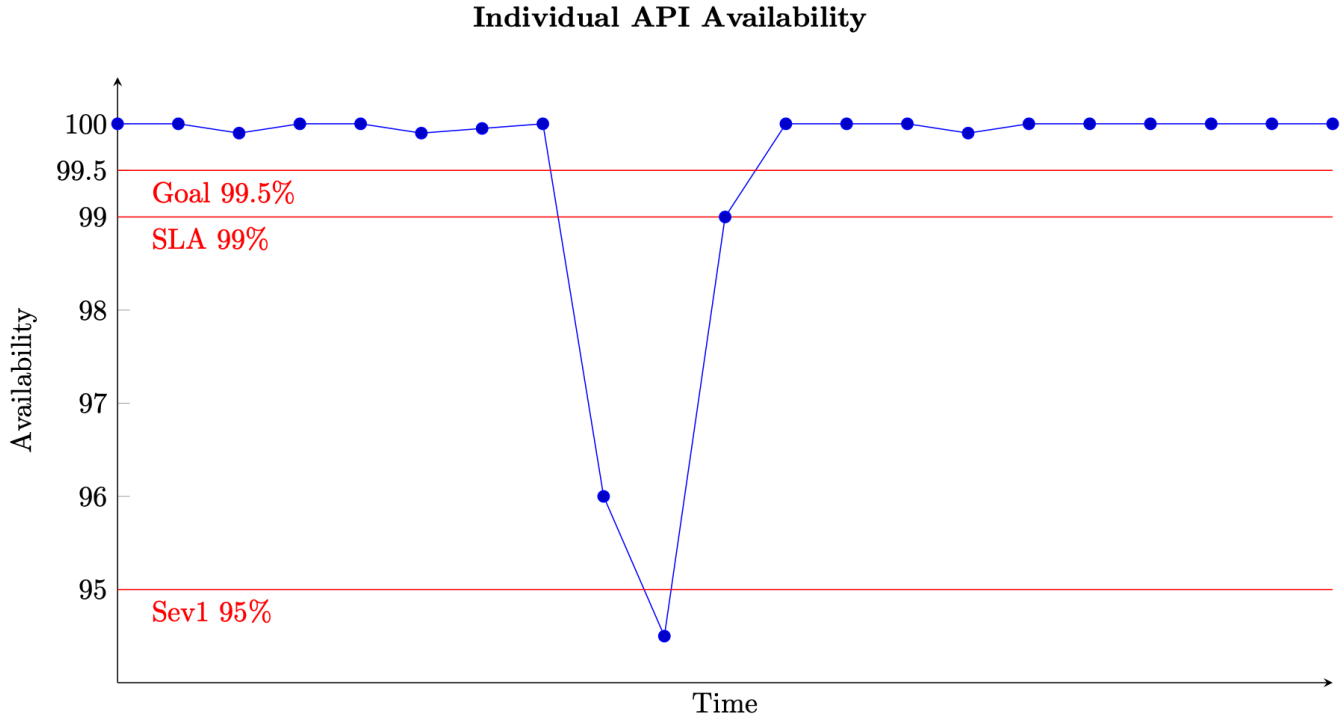
最初の 2 つの方法は非常に似ており、測定する観点のみが異なります。サーバー側のメトリクスは、サービスの計測から収集できます。しかし、これらは完全ではありません。クライアントがサービスにアクセスできない場合、それらのメトリクスを収集することはできません。クライアントエクスペリエンスを理解するには、失敗したリクエストに関するクライアントからのテレメトリに頼ってはいけません。より簡単なのは、サービスを定期的に調査してメトリックを記録するソフトウェアの canary を使用しカスタマートラフィックをシミュレートすることがクライアント側のメトリクスを収集することです。

この 2 つの方法では、可用性は、サービスが受け取った有効な作業単位と正常に処理された作業単位の合計に対する割合として計算されます (404 エラーになる HTTP リクエストなどの無効な作業単位は無視されます)。

$$A = \frac{Successfully\ Processed\ Units\ of\ Work}{Total\ Valid\ Units\ of\ Work\ Received}$$

方程式 8

リクエストベースのサービスの場合、作業単位は HTTP リクエストと同様にリクエストです。イベントベースまたはタスクベースのサービスの場合、作業単位はイベントまたはタスク (キュー外のメッセージの処理など) です。この可用性の測定は、1 分または 5 分といった短い時間間隔では意味があります。また、リクエストベースのサービスの API レベル別の場合など、きめ細かい観点からも最適です。次の図は、この方法で計算した場合、時間の経過に伴って可用性がどのように変化するかを示しています。グラフの各データポイントは、式 (8) を使用して 5 分間隔で計算されます (1 分または 10 分間隔など、他の時間軸も選択できます)。例えば、データポイント 10 では 94.5% の可用性が示されています。つまり、t+45 から t+50 までの間に、サービスが 1,000 件のリクエストを受け取ったとしても、そのうち正常に処理されたのはそのうちの 945 件だけでした。



単一の API の可用性を時系列で測定する例

このグラフには、API の可用性目標である 99.5%、カスタマーに提供するサービスレベルアグリーメント (SLA)、99% の可用性、および重要度の高いアラームのしきい値(95%) も示されています。こ

れらの各種しきい値のコンテキストが欠如すると、可用性のグラフではサービスの稼働状況に関する重要な分析情報が得られない可能性があります。

また、コントロールプレーンなどの大規模なサブシステムやサービス全体の可用性を追跡し、説明できることが理想です。その方法の 1 つは、各サブシステムの 5 分間のデータポイントの平均を取ることです。このグラフは前のグラフと似ていますが、より多数の入力を表しています。また、サービスを構成するすべてのサブシステムに同等の重みを与えています。別の方法として、サービス内のすべての API から受け取り、正常に処理されたすべてのリクエストを合計して、5 分間隔で可用性を計算することもできます。

ただし、後者の方法では、スループットが低く、可用性の低い個々の API は表されない可能性があります。簡単な例として、2 つの API を持つサービスについて考えてみましょう。

最初の API は 5 分以内に 1,000,000 件のリクエストを受け取り、そのうちの 999,000 件を正常に処理するため、99.9% の可用性を実現しています。2 番目の API は、同じ 5 分間にリクエストを 100 件受け取り、そのうちの 50 件しか正常に処理しないため、可用性が 50% です。

各 API からのリクエストを合計すると、合計で 1,000,100 件の有効なリクエストがあり、そのうちの 999,050 件が正常に処理され、サービス全体の可用性は 99.895% になります。ただし、前者の方法のように 2 つの API の可用性を平均すると、74.95%の可用性となり、実際の エクスペリエンスがより正しく表現している可能性があります。

どちらのアプローチも間違っていませんが、可用性のメトリクスが伝える内容を理解することが重要です。ワークロードが各サブシステムで同様のリクエスト量を受け取っている場合は、すべてのサブシステムのリクエストの合計を選択することもできます。このアプローチは、可用性とカスタマーエクスペリエンスの尺度としての「リクエスト」とその成功に焦点を当てています。あるいは、リクエスト量が異なっても、サブシステムの可用性を平均して、その重要度を等しく表すこともできます。このアプローチは、サブシステムおよびカスタマーエクスペリエンスのプロキシとしての各サブシステムの能力に焦点を当てています。

年間ダウンタイム

3 つ目のアプローチは、年間ダウンタイムを計算することです。この形式の可用性マトリクスは、長期的な目標設定とレビューに適しています。ダウンタイムがワークロードにとって何を意味するのかを定義する必要があります。これにより、特定の期間の合計時間 (分) に対してワークロードが「停止」状態ではない時間 (分) を基に可用性を測定できます。

ワークロードによっては、単一の API またはワークロード関数が 1 分または 5 分間隔で 95% の可用性を下回るようなものとして、ダウンタイムを定義できる場合があります (これは前述の可用性グラ

フで発生しました)。また、ダウンタイムが重要なデータプレーンのオペレーションのサブセットに適用されるときのみ検討する場合があるかもしれません。例えば、SQS の可用性に関する Amazon メッセージング (SQS、SNS) サービスレベルアグリーメントは SQS 送信、受信、削除 API に適用されます。

大規模で複雑なワークロードでは、システム全体の可用性メトリクスの定義が必要な場合があります。大規模な e コマースサイトの場合、システム全体のメトリクスはカスタマーの注文率のようなものです。この場合、5 分間のうちに、予測数量と比較して 10% 以上注文が減少した場合に、ダウンタイムとなる可能性があります。

いずれの方法でも、すべての停止期間を合計して年間の可用性を計算できます。例えば、ある暦年に 5 分間のダウンタイムが 27 回発生した場合 (データプレーン API の可用性が 95% を下回ったと定義)、全体のダウンタイムは 135 分で (5 分間は連続していた場合と分離していた場合がある)、つまり、年間可用性は 99.97% です。

この可用性を測定する追加の方法を使うと、クライアント側とサーバー側のメトリクスにはないデータや洞察が得られる可能性があります。例えば、障害が発生し、エラー率が大幅に上昇しているワークロードについて考えてみましょう。このワークロードのカスタマーは、そのサービスへの呼び出しをすべて停止する可能性があります。もしかすると、回路遮断器を作動させたり、ディザスタリカバリ計画に従って別のリージョンでサービスを利用しているかもしれません。失敗したレスポンスだけを測定すると、障害発生時のワークロードの可用性が実際に向上するかもしれませんが、これは障害が改善または解消されるからではなく、カスタマーがその使用を停止するためです。

レイテンシー

最後に、ワークロード内の処理単位のレイテンシーを測定することも重要です。可用性は一部として、確立された SLA の範囲内で作業を行うことと定義されます。レスポンスを返すのにクライアントのタイムアウトよりも時間がかかる場合、クライアントはリクエストが失敗し、ワークロードが利用できないと認識します。ただし、サーバー側では、リクエストが正常に処理されたように見える場合があります。

レイテンシーの測定は、可用性を評価するもう 1 つの方法となります。この測定では、パーセンタイルとトリミング平均を使用するのが適切な統計方法です。通常、50 パーセンタイル (P50 と TM50) と 99 パーセンタイル (P99 と TM99) で測定されます。レイテンシーは、クライアントエクスペリエンスを表す canary とサーバー側のメトリクスで測定する必要があります。P99 や TM99.9 など、あるパーセンタイルのレイテンシーの平均が目標 SLA を上回る場合は、常にそのダウンタイムを考慮して、年間ダウンタイムの計算に役立てることができます。

AWS での高可用性分散システムの設計

これまでのセクションでは、主にワークロードの理論的な可用性と、それによって実現できることについて説明しました。これらは、分散システムを構築する際に覚えておくべき重要な概念です。依存関係の選択プロセスや冗長性の実装方法を知るのに役立ちます。

これまで、MTTD、MTTR、MTBF と可用性の関係について説明しました。このセクションでは、これまでの理論に基づいた実践的なガイダンスを紹介します。つまり、高可用性を実現するためのエンジニアリングワークロードは、MTBF を増やし、MTTR と MTTD を減らすことを目的としています。

障害をすべて無くすことが理想的ですが、これは現実的ではありません。依存関係が深く積み重なっている大規模な分散システムでは、障害が発生することがあります。「壊れないものはない」(Amazon.com 最高技術責任者、Werner Vogels、「Amazon Web Services の 10 年間の経験から得た 10 の教訓」を参照)、「障害を法律で規定することはできません。だから迅速な検出と対応に重点を置くべきです」(ARC335 Amazon EC2 チーム創設メンバーの Chris Pinkham 氏、「障害に備える設計:AWS での耐障害性の高いシステムの設計」を参照)

障害が発生するかどうかを制御できないことは頻繁にあります。制御できるのは、障害をどれだけ早く検出して対処するかです。そのため、MTBF の増加は依然として高可用性の重要な要素ですが、MTTD と MTTR の削減は、カスタマーが制御できる最も大きな変更です。

トピック

- MTTD の削減
- MTTR の削減
- MTBF の増加

MTTD の削減

障害のある MTTD を減らすことで、障害を可能な限り早期発見することができます MTTD の短縮は、可観測性、つまりワークロードの状態を把握するためにどのように計測したかに基づいています。カスタマーは、問題がいつ発生したかを事前に特定する方法として、ワークロードの重要なサブシステムでカスタマーエクスペリエンス指標をモニタリングする必要があります (これらの指標の詳細については、「付録1 — MTTD と MTTR の重要なメトリクス」を参照)。カスタマーは Amazon CloudWatch Synthetics を使用して API やコンソールを監視する canary を作成して、ユーザーエクスペリエンスを積極的に測定できます。Elastic Load Balancing (ELB) ヘルスチェックや Amazon

Route 53 ヘルスチェックなど、MTTD を最小限に抑えるために使用できるヘルスチェックメカニズムは他にも多数あります。(「Amazon Builders' Library — ヘルスチェックの実装」を参照。)

また、モニタリングでは、システム全体と個々のサブシステムの両方の部分的な障害を検出できる必要があります。可用性、障害、レイテンシーのメトリックスでは、障害分離の境界の次元性を CloudWatch メトリクスディメンションとして使用する必要があります。例えば、us-east-1 リージョンの use1-az1 AZ にある、セルベースアーキテクチャの一部である単一の EC2 インスタンスが、コントロールプレーンサブシステムの一部であるワークロードの更新 API の一部であるとします。サーバーがメトリクスをプッシュすると、インスタンス ID、AZ、リージョン、API 名、サブシステム名をディメンションとして使用できます。これにより、可観測性を確保し、これらの各ディメンションにわたってアラームを設定して障害を検出できます。

MTTR の削減

障害が発見された後、残りの MTTR 時間では実際の修理または影響の軽減を実施します。障害を修復または軽減するには、何が問題なのかを知る必要があります。このフェーズの重要なメトリクスには、1/ 影響評価メトリクスと 2/ 運用健全性メトリクスの 2 つの主要なグループがあります。最初のグループは、障害発生時の影響の範囲を示し、影響を受けたカスタマー、リソース、またはワークロードの数または割合を測定します。2 番目のグループは、影響がある理由を特定するのに役立ちます。理由が明らかになると、オペレーターとオートメーションは障害に対応し、解決できます。これらのメトリクスの詳細については、「付録 1 — MTTD と MTTR の重要なメトリクス」を参照してください。

> ルール 9
>
> MTTD と MTTR を減らすには、可観測性と計測が不可欠です。

障害回避ルート

影響を軽減する最速のアプローチは、障害を回避するフェイルファストサブシステムを使用することです。このアプローチでは、冗長性を利用して、障害が発生したサブシステムの処理をスペアにすばやく移行することで MTTR を削減します。冗長性は、ソフトウェアプロセスから EC2 インスタンス、複数の AZ、複数のリージョンまで多岐にわたります。

スペアのサブシステムでは、MTTR をほぼゼロまで下げることができます。復旧時間は、スタンバイのスペアに処理を再ルーティングするのにかかる時間だけです。多くの場合、これは最小限のレイテンシーで行われ、定義された SLA の範囲内で作業を完了でき、システムの可用性を維持できま

す。これによっ発生するMTTR は、わずか、または気づかない程度の遅延であり、長期間利用できなくなるということはありません。

例えば、サービスが Application Load Balancer (ALB) の背後にある EC2 インスタンスを利用している場合、ヘルスチェックを 5 秒程度の短い間隔で設定し、ターゲットが異常とマークされるまでに必要なヘルスチェックの失敗は 2 回のみです。つまり、10 秒以内に障害を検出し、問題のあるホストへのトラフィックの送信を停止できます。この場合、障害が検出されるとすぐに軽減が行われるため、MTTR は MTTD と実質的に同じになります。

これこそが、高可用性または連続可用性のワークロードが達成しようとしていることです。障害が発生したサブシステムをすばやく検出して障害としてマークし、そのサブシステムへのトラフィックの送信を停止して、代わりに冗長サブシステムにトラフィックを送信することで、ワークロードの障害を迅速に回避することが理想です。

ただし、この種のフェイルファストメカニズムを使用すると、ワークロードが一時的なエラーに対して非常に敏感になるため注意が必要です。この例では、ロードバランサーのヘルスチェックが、依存関係またはワークフローのテスト (ディープヘルスチェックと呼ばれることが多い) ではなく、インスタンスのみのシャローまたは稼働状態とローカルヘルスチェックを実行していることを確認します。これにより、ワークロードに影響する一時的なエラーが発生したときに、インスタンスの不要な置き換えを防ぐことができます。

障害を回避するルーティングを成功させるには、可観測性とサブシステムの障害を検出する能力が不可欠です。また、影響の範囲を把握することで、影響を受けるリソースに異常または障害のマークを付けてサービスを停止し、回避できるようになります。例えば、1 つの AZ で部分的なサービス障害が発生した場合、計測により AZ にローカライズされた問題があることを特定し、復旧するまでその AZ のすべてのリソースを回避できる必要があります。

障害を回避するため、環境によっては追加のツールが必要になる場合もあります。前の ALB の背後にある EC2 インスタンスの例で、ある AZ のインスタンスがローカルヘルスチェックに合格しているのに、特定された AZ の障害が原因で別の AZ のデータベースに接続できなくなっているとします。この場合、負荷分散ヘルスチェックによってそれらのインスタンスがサービス停止になることはありません。ロードバランサーから AZ を削除したり、インスタンスのヘルスチェックを強制的に不合格するには、別の自動化メカニズムが必要になります。これは、影響の範囲が AZ であるかどうかを特定することに依存します。ロードバランサーを使用していないワークロードでは、特定の AZ のリソースが作業単位を受け付けたり、AZ から容量を完全に削除するのを防ぐために、同様の方法が必要になります。

場合によっては、冗長なサブシステムへの作業の移行を自動化できないこともあります。例えば、テクノロジーが独自のリーダーの選択を行わない場合のプライマリデータベースからセカンダリデー

タベースへのフェイルオーバーなどです。これは、AWSマルチリージョンアーキテクチャでは一般的なシナリオです。この種のフェイルオーバーでは、完了するまでにある程度のダウンタイムが必要で、すぐに元に戻すことができず、一定期間ワークロードは冗長にならないままです。そのため、意思決定プロセスに人が存在することが重要です。

厳密性の低い整合性モデルを採用できるワークロードでは、マルチリージョンのフェイルオーバーオートメーションを使用して障害を回避することで、MTTR を短縮できます。Amazon S3 クロスリージョンレプリケーションまたは Amazon DynamoDB グローバルテーブルなどの機能は、結果的に整合性のあるレプリケーションを通じてマルチリージョン機能を提供します。さらに、CAP 定理を考えるときには、緩和された整合性モデルを使用することが有益です。ステートフルサブシステムへの接続に影響するネットワーク障害が発生しても、ワークロードが整合性よりも可用性を優先した場合でも、障害を回避するルーティングの別の方法として、エラー以外のレスポンスを提供できます。

障害を回避するルーティングは、2 つの異なる方法で実装できます。最初の戦略は、障害が発生したサブシステムのすべての負荷を処理するのに十分なリソースを事前にプロビジョニングして、静的安定性を実装することです。これは、単一の EC2 インスタンスでも、AZ 全体の容量でもかまいません。障害発生時に新しいリソースをプロビジョニングしようとすると、MTTR が増加し、復旧パスのコントロールプレーンに依存関係が追加されます。ただし、追加料金が発生します。

2 つ目の戦略は、障害が発生したサブシステムから他のサブシステムへトラフィックの一部をルーティングし、残りの容量では処理できない余分なトラフィックを負荷分散することです。このパフォーマンスが低下している間、障害が発生した容量を補うために新しいリソースをスケールアップできます。この方法では MTTR が長くなり、コントロールプレーンへの依存が発生しますが、スタンバイ容量および予備容量のコストは低下します。

既知の正常な状態に戻す

修復中に軽減するためのもう 1 つの一般的な方法は、ワークロードを以前の既知の正常な状態に回復させることです。最近の変更が原因で障害が発生した可能性がある場合は、その変更をロールバックすることが前の状態に戻す方法の 1 つです。

また、一時的な状態が原因で障害が発生した場合は、ワークロードを再起動することで影響が軽減される可能性があります。これらの 2 つのシナリオをについて検証してみましょう。

デプロイ時に MTTD と MTTR を最小限に抑えるには、可観測性と自動化が必要です。デプロイプロセスでは、エラー率の増加、レイテンシーの増加、または異常が発生していないか、ワークロードを継続的にモニタリングする必要があります。これらが認識されると、デプロイプロセスが停止されます。

インプレースデプロイ、ブルー/グリーンデプロイ、ローリングデプロイなど、さまざまなデプロイ戦略があります。これらはそれぞれ、既知の正常な状態に戻るために異なるメカニズムを使用する場合があります。自動的に前の状態にロールバックしたり、トラフィックをブルー環境に戻したり、手動で操作することができます。

CloudFormation には、AWS CodeDeploy と同様に、スタックの作成および更新オペレーションの一部として自動的にロールバックする機能があります。CodeDeploy は、ブルー/グリーンデプロイとローリングデプロイもサポートしています。

これらの機能を活用して MTTR を最小限に抑えるには、すべてのインフラストラクチャとコードのデプロイをこれらのサービスを使って自動化することを検討してください。これらのサービスを使用できないシナリオでは、AWS Step Functions での saga パターンの実装により、失敗したデプロイをロールバックすることを検討してください。

再起動を検討する場合、いくつかの異なるアプローチがあります。これらは、作業時間が最長であるサーバーの再起動から、最短のスレッドの再起動まで多岐にわたります。次の表は、一部の再起動方法とおおよその完了までの時間をまとめたものです (桁違いを表記するためであり、これらは正確ではありません)。

障害復旧メカニズム	推定 MTTR
新しい仮想サーバーの起動と設定	15 分
ソフトウェアの再デプロイ	10 分
サーバーの再起動	5 分
コンテナの再起動または起動	2 秒
新しいサーバーレス関数の呼び出し	100 ミリ秒
プロセスの再起動	10 ミリ秒
スレッドの再起動	10 µs

表を見直してみると、コンテナやサーバーレス関数 (AWS Lambda など) を使うことは、MTTR にとって明確な利点がいくつか存在します。これらの MTTR は、仮想マシンを再起動したり、新しいマシンを起動するよりも桁違いに速いです。ただし、ソフトウェアのモジュール化による障害の分

離も有益です。障害を単一のプロセスまたはスレッドに抑えることができると、その障害からの復旧は、コンテナまたはサーバーを再起動するよりもはるかに高速です。

復旧の一般的なアプローチとしては、下から上へ、つまり 1/再起動、2/リブート、3/再イメージ化/再デプロイ、4/置換という順序で進めることができます。ただし、再起動の手順に入ると、障害を回避するルーティングの方が通常は迅速です (通常、最長で 3 ~ 4 分かかります)。そのため、再起動を試みた後で影響を最も早く軽減するには、障害を回避するルーティングをし、バックグラウンドで復旧プロセスを続行して容量をワークロードに戻します。

> ルール 10
>
> 問題解決ではなく、影響の軽減に焦点を当てる。最速の方法で通常のオペレーションに戻す。

障害の診断

検出後の復旧プロセスの一部には、診断期間があります。これは、オペレーターが問題が何かについて判断を試みる期間です。このプロセスには、ログのクエリ実行、運用健全性メトリクスの確認、トラブルシューティングのためのホストへのログインが含まれる場合があります。これらのアクションはすべて時間がかかるため、迅速化するためのツールやランブックを作成すると、MTTR の短縮にも役立ちます。

ランブックと自動化

同様に、問題が何か、そしてどのような一連の作業によりワークロードを修復できるかについて判断したら、オペレーターは通常、そのために一連のステップを実行する必要があります。例えば、障害発生後、ワークロードを修復する最速の方法は、再起動かもしれません。これには、順序付けられた複数のステップが必要になることがあります。これらのステップを自動化したり、またはオペレーターに特定の指示を与えるランブックを利用すると、プロセスが迅速になり、不注意による作業ミスのリスクが軽減されます。

MTBF の増加

可用性を向上させる最後の要素は、MTBF の増加です。これは、ソフトウェアとその実行に使用される AWS サービスの両方に当てはまります。

分散システムの MTBF の増加

MTBF を増やす方法の 1 つは、ソフトウェアの欠陥を減らすことです。この方法には、いくつかあります。カスタマーは Amazon CodeGuru Reviewer などのツールを使用して、一般的なエラーを見つけて修正できます。また、ソフトウェアを本番環境にデプロイする前に、包括的なピアコードレビュー、ユニットテスト、統合テスト、リグレッションテスト、および負荷テストを実行する必要があります。テストのコードカバレッジを増やすと、一般的でないコード実行パスでも確実にテストできるようになります。

小さな変更をデプロイすることは、変更の複雑さが軽減され、予期しない結果を防ぐのにも役立ちます。それぞれのアクティビティは、障害が発生する前に欠陥を特定して修正する機会を提供します。

障害を防ぐもう 1 つの方法は、定期的なテストです。カオスエンジニアリングプログラムを実装すると、ワークロードがどのように障害を起こすかをテストしたり、復旧手順を検証したり、本番環境で障害が発生する前に障害モードを見つけて修正するのに役立ちます。カスタマーは、カオスエンジニアリング実験ツールセットの一部として AWS Fault Injection Simulator を使用できます。

分散システムで障害を防ぐもう 1 つの方法は、耐障害性です。フェイルファストモジュール、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、トランザクション、冪等性はすべて、ワークロードの耐障害性に役立つ手法です。

トランザクションは、ACID プロパティに準拠したオペレーションのグループです。その内容は次のとおりです。

- 不可分性 — すべてのアクションが実行されるか、まったく実行されないかのどちらかです。
- 整合性 — 各トランザクションは、ワークロードを有効な状態のままにします。
- 分離性 — トランザクションを並行して実行すると、ワークロードは連続して実行された場合と同じ状態になります。
- 耐久性 — トランザクションがコミットされると、ワークロードに障害が発生した場合でもその影響はすべて維持されます。

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を行うと、ハイゼンバグ、過負荷、またはその他の条件によって引き起こされる一時的な障害を克服できるようになります。トランザクションが冪等性の場合は、複数回再試行でき、それによる悪影響はありません。

耐障害性のあるハードウェア構成にハイゼンバグが及ぼす影響については、ハイゼンバグがプライマリサブシステムと冗長サブシステムの両方に発生する確率はごくわずかであるため、ほとん

ど心配する必要はないでしょう。(Jim Gray の「Why Do Computers Stop and What Can Be Done About It? (コンピューターはなぜ停止するのか、そしてそれに対して何ができるのか？)」(1985 年 6 月、Tandem Technical Report 85.7 を参照)。分散システムでは、ソフトウェアで同じ結果を達成することが必要です。

ハイゼンバグが発生したとき、ソフトウェアが不正な操作やエラーをすばやく検出して、再試行できるようにすることが不可欠です。これは、防御的なプログラミングと、入力、中間結果、および出力の検証によって実現されます。また、プロセスは分離されており、他のプロセスと状態を共有しません。

このモジュラー型アプローチを取ることで、障害発生時の影響範囲が限定されます。プロセスの失敗は独立して発生します。プロセスが失敗したとき、ソフトウェアは「プロセスペア」を使用して作業を再試行する必要があります。これにより、失敗したプロセスの処理を新しいプロセスに引き継ぐことができます。ワークロードの信頼性と完全性を維持するには、各オペレーションを ACID トランザクションとして扱う必要があります。

これにより、トランザクションを中止し、加えられた変更をロールバックすることで、ワークロードの状態を損なうことなくプロセスを失敗させることができます。これにより、復旧プロセスは既知の正常な状態からトランザクションを再試行し、正常に再起動できます。このようにして、ソフトウェアはハイゼンバグに対して耐障害性を持たせることができます。

ただし、ボーアバグに対してソフトウェアに耐障害性を持たせることは、現実的ではありません。どのレベルの冗長性でも正しい結果を達成することはできないため、これらの欠陥はワークロードが本番環境に入る前に発見して取り除く必要があります。(Jim Gray の「Why Do Computers Stop and What Can Be Done About It? (コンピューターはなぜ停止するのか、そしてそれに対して何ができるのか？)」(1985 年 6 月、Tandem Technical Report 85.7 を参照)。

MTBF を増やす最後の方法は、障害による影響の範囲を狭めることです。そのための主な方法は、「耐障害性と障害分離」で前述したように、モジュール化による障害分離を使用して障害コンテナを作成することです。障害発生率を下げると、可用性が向上します。AWS はサービスのコントロールプレーンとデータプレーンへの分割、アベイラビリティーゾーン独立性 (AZI)、リージョナル分離、セルベースアーキテクチャ、障害を分離するためのシャッフルシャーディングなどの手法を使用します。これらは AWS カスタマーも使用できるパターンです。

例えば、ワークロードによってカスタマーがインフラストラクチャの異なる障害コンテナに配置され、全カスタマーの 5% にサービスが提供されているシナリオについて検討してみましょう。これらの障害コンテナの 1 つで、リクエストの 10% でクライアントのタイムアウトを超えるレイテンシーが増加するイベントが発生しました。このイベントでは、95% のカスタマーがサービスを 100% 利用できました。残りの 5% については、サービスは 90% 利用可能だったようです。こ

の結果、可用性は、100% の顧客に対してリクエストの 10% が失敗するのではなく (90% の可用性)、1 − (5% of customers×10% of their requests) = 99.5% となります。

> ルール 11
>
> 障害を分離すると、全体的な障害発生率が低下するため、影響範囲が狭まり、ワークロードの MTBF が増加します。

依存関係の MTBF の増加

AWS 依存関係の MTBF を増やす最初の方法は、障害分離を使用することです。多くの AWS サービスは、AZ であるレベルの分離を提供しています。つまり、ある AZ で障害が発生しても、別の AZ のサービスには影響しません。

複数の AZ で冗長 EC2 インスタンスを使用すると、サブシステムの可用性が向上します。AZI は 1 つのリージョン内でスペアリング機能を提供するため、AZI サービスの可用性を高めることができます。

ただし、すべての AWS サービスが AZ レベルで動作するわけではありません。他の多くはリージョンでの分離を提供します。このようなケースでは、リージョンサービスの可用性がワークロードに必要な全体的な可用性をサポートしていない場合、マルチリージョンのアプローチを検討してみましょう。各リージョンでは、スペアリングと同等のサービスの分離インスタンス化を提供しています

マルチリージョンサービスの構築を容易にするさまざまなサービスがあります。例:

- Amazon Aurora Global Database
- Amazon DynamoDB グローバルテーブル
- Amazon ElastiCache (Redis OSS) — グローバルデータストア
- AWS Global Accelerator
- Amazon S3 クロスリージョンレプリケーション
- Amazon Route 53 Application Recovery Controller

このドキュメントでは、マルチリージョンワークロードを構築する戦略について詳しく説明しませんが、マルチリージョンアーキテクチャの可用性のメリットと、希望する可用性目標を達成するために必要な追加コスト、複雑さ、運用慣行について比較検討する必要があります。

依存関係の MTBF を増やす次の方法は、ワークロードを静的に安定させるように設計することです。例えば、製品情報を提供するワークロードがあるとします。カスタマーが製品をリクエストすると、サービスは外部のメタデータサービスに製品詳細の取得をリクエストします。その後、ワークロードはその情報をすべてユーザーに返します。

ただし、メタデータサービスが利用できない場合、カスタマーからのリクエストは失敗します。代わりに、メタデータをサービスのローカルに非同期でプルまたはプッシュして、リクエストへのレスポンスに使用できます。これにより、重要なパスからメタデータサービスへの同期呼び出しが不要になります。

さらに、メタデータサービスが利用できなくてもサービスは利用できるため、可用性計算の依存関係としてそれを削除できます。この例は、メタデータは頻繁に変更されず、古いメタデータを提供する方がリクエストが失敗するよりも良いという前提に基づいています。別の同様の例として、DNS の serve-stale があります。これは、TTL の有効期限が過ぎてもデータをキャッシュに保持し、更新された回答がすぐに利用できない場合のレスポンスに使用できます。

MTBF を増やす最後の方法は、依存関係の MTBF を増やして障害による影響の範囲を狭めることです。前述のとおり、失敗は二者択一のイベントではありません。障害には程度があります。これがモジュール化の効果です。障害は、そのコンテナによって処理されているリクエストまたはユーザーだけに抑えられます。

その結果、イベント中の障害が減り、影響の範囲が制限されるため、最終的にはワークロード全体の可用性が向上します。

一般的な影響の原因の削減

1985年、Jim Gray は、Tandem Computers での研究中に、障害は主にソフトウェアとオペレーションの 2 つの要因によって引き起こされていることを発見しました。(Jim Gray の「Why Do Computers Stop and What Can Be Done About It? (コンピューターはなぜ停止するのか、そしてそれに対して何ができるのか？)」(1985 年 6 月、Tandem Technical Report 85.7 を参照)。それから 36 年経った今でも、これは真実です。テクノロジーの進歩にもかかわらず、これらの問題に対する簡単な解決策はなく、主な障害の原因は変わっていません。このセクションの冒頭では、ソフトウェアの障害への対処について説明しました。ここではオペレーションと障害の発生頻度の削減に焦点を当てます。

機能と比較した安定性

the section called “分散システムの可用性” セクションのソフトウェアとハードウェアの障害発生率のグラフをもう一度見てみると、各ソフトウェアリリースに不具合が追加されていることがわかりま

す。つまり、ワークロードに何らかの変更を加えると、障害のリスクが高まります。これらの変更は通常、当然の結果をもたらす新機能のようなものです。可用性の高いワークロードでは、新機能よりも安定性が優先されます。したがって、可用性を向上させる最も簡単な方法の 1 つは、デプロイの頻度を減らすか、提供する機能を減らすことです。デプロイの頻度が高いワークロードは、そうではないワークロードよりも本質的に可用性が低くなります。ただし、機能を追加できないワークロードはカスタマーの需要に追いついておらず、時間が経つにつれて有用性が低下する可能性があります。

では、どうすればイノベーションを続け、安全に機能をリリースできるのでしょうか。その答えは標準化です。正しいデプロイの方法は何ですか。どのようにデプロイを発注しますか。テストの基準は何ですか。ある段階から別の段階までの待機時間はどれくらいですか。ユニットテストはソフトウェアコードを十分にカバーしていますか。これらの疑問は標準化によって解消されます。また、負荷テストを実施しなかったり、デプロイ段階をスキップしたり、あまりにも多くのホストに早期に展開しすぎることによって引き起こされる問題を防止できます。

標準化を実装する方法は自動化です。これにより、人為的なミスの可能性が減り、コンピューターは得意なこと、つまり毎回同じことを同じ方法で繰り返すことが可能になります。標準化と自動化を両立させるには、目標を設定します。目標は、手動による変更なし、条件付き認証システムのみによるホストアクセス、すべての API の負荷テストの記述などです。運用上の優秀性は文化的規範であり、これには大規模な変更が必要になる場合があります。目標に対するパフォーマンスを確立して追跡することは、ワークロードの可用性に幅広い影響を与える文化的な変化を促進します。AWS Well-Architected 運用上の優秀性の柱は、運用上の優秀性を達成するための包括的なベストプラクティスを提供します。

オペレーターの安全

障害を引き起こす運用上のイベントのその他の主な原因は人です。人間は間違いを犯します。間違った認証情報を使用したり、間違ったコマンドを入力したり、Enter キーを押すのが早すぎたり、重要な手順を見逃したりする可能性があります。手動によるアクションを継続的に実行すると、エラーが発生し、障害につながります。

オペレーターのエラーの主な原因の 1 つは、わかりにくい、直感的でない、または整合性のないユーザーインターフェイスです。Jim Gray は 1985 年の研究で、「オペレーターに情報を求めたり、何らかの機能の実行を求めたりするインターフェースは、シンプルで整合性があり、オペレーターに対する耐障害性がなければならない」とも述べています。(Jim Gray の「Why Do Computers Stop and What Can Be Done About It? (コンピューターはなぜ停止するのか、そしてそれに対して何ができるのか？)」(1985 年 6 月、Tandem Technical Report 85.7 を参照)。この洞察は今でも真実です。わかりづらい、または複雑なユーザーインターフェイス、確認や指示の欠如、あるいは単に不親切な人による言葉が原因で、オペレーターが間違ったことをする原因となった例は、過去 30 年間にわたり業界全体で数多く存在します。

ルール 12

オペレーターが正しいことを簡単に行えるようにします。

過負荷の防止

最後に影響に寄与する一般的な因子は、カスタマー、つまりワークロードの実際のユーザーです。正常なワークロードは使用されるようになりますが、その使用量がワークロードのスケールする能力を上回ることもあります。ディスクがいっぱいになり、スレッドプールが使い果たされ、ネットワーク帯域幅が飽和状態になり、データベース接続の制限に達するなど、さまざまなことが起こり得ます。

これらを解消する確実な方法はありませんが、運用健全性メトリクスを通じて容量と使用率を積極的に監視することで、障害が発生する可能性がある場合、早期に警告を発することができます。負荷分散、回路ブレーカー、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などの手法は、影響を最小限に抑えて成功率を高めるのに役立ちますが、このような状況では障害はまだ解消されていません。運用健全性メトリクスに基づく自動スケーリングは、過負荷による障害の頻度を減らすのに役立ちますが、使用率の変化に対して十分迅速に対応できない場合があります。

カスタマーが継続的に利用可能な容量を確保する必要がある場合は、可用性とコストのバランスを取る必要があります。容量不足が可用性の低下につながらないようにする 1 つの方法として、各カスタマーにクォータを設定し、割り当てられたクォータの 100% を提供するようにワークロードの容量をスケーリングすることです。カスタマーがクォータを超えるとスロットリングされますが、これは障害ではなく、可用性にも影響しません。また、十分な容量を確保するためには、カスタマーベースを綿密に追跡し、将来の使用率を予測する必要があります。これにより、カスタマーによる過剰消費によって、ワークロードで障害が発生するような状況を回避できます。

- Amazon Builders' Library — 負荷分散による過負荷の回避
- Amazon Builders' Library — マルチテナントシステムの公平性

例えば、ストレージサービスを提供するワークロードについて検証してみましょう。ワークロード内の各サーバーは 1 秒あたり 100 回のダウンロードをサポートでき、カスタマーには 1 秒あたり 200 回のダウンロードというクォータが提供され、500 人のカスタマーがいます。この数のカスタマーをサポートするには、サービスは 1 秒あたり 100,000 ダウンロードの容量を提供する必要があり、それには 1,000 台のサーバーが必要です。いずれかのカスタマーがクォータを超えると、そのカスタマーはスロットリングされ、他のすべてのカスタマーに十分な容量が確保されます。これは、作業単位を拒否せずに過負荷を回避する 1 つの方法の簡単な例です。

結論

このドキュメントでは、高可用性のための 12 のルールを定めています。

- ルール 1 — 分散システムの可用性を向上させるには、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 つの要素があります。
- ルール2 — ワークロード内のソフトウェアの可用性は、ワークロード全体の可用性にとって重要な要素であり、他のコンポーネントと同様に重視する必要があります。
- ルール 3 — 依存関係を減らすことは、可用性にプラスの影響を与える可能性があります。
- ルール 4 — 一般的には、可用性の目標がワークロードの目標と同等かそれ以上の依存関係を選択します。
- ルール5 — スペアリングを使用すると、ワークロード内の依存関係の可用性が向上します。
- ルール6 — スペアリングのコスト効率には上限があります。必要な可用性を実現するには、必要最小限のスペアのみを使用します。
- ルール 7 — 特にリカバリ中は、データプレーンのコントロールプレーンに依存関係を作成しません。
- ルール 8 — 可能な限り、依存関係が損なわれてもワークロードが正しく動作できるように、依存関係を疎結合します。
- ルール9 — MTTD と MTTR を減らすには、可観測性と計測が不可欠です。
- ルール 10 — 問題解決ではなく、影響の軽減に焦点を当てます。最速の方法で通常のオペレーションに戻す。
- ルール11 — 障害を分離すると、全体的な障害発生率が低下するため、影響範囲が狭まり、ワークロードの MTBF が増加します。
- ルール 12 — オペレーターが正しいことを簡単に行えるようにします。

ワークロードの可用性を向上させるには、MTTD と MTTR を減らし、MTBF を増やす必要があります。つまり、テクノロジー、人材、プロセスを対象として、可用性を向上させる次の方法について説明しました。

- MTTD
  - カスタマーエクスペリエンスのメトリクスを積極的に監視することで、MTTD を削減します。
  - きめ細かなヘルスチェックを活用して、迅速なフェイルオーバーを実現します。
- MTTR

- 影響範囲と運用健全性メトリクスを監視します。
- 1/再起動、2/リブート、3/再イメージ化/再デプロイ、4/交換の順に MTTR を短縮します。
- 影響範囲を把握して、障害を回避します。
- 仮想マシンや物理ホストを介したコンテナやサーバーレス機能など、再起動時間が短いサービスを利用します。
- 可能であれば、障害の発生したデプロイを自動的にロールバックします。
- 診断オペレーションと再起動手順のためのランブックと運用ツールを確立します。

- MTBF
  - 本番環境にリリースする前に厳密なテストを実施し、ソフトウェアのバグや欠陥を排除します。
  - カオスエンジニアリングと障害挿入を実装します。
  - 障害を許容できるように、依存関係には適切な数のスペアリングを持たせます。
  - 障害コンテナによって障害時の影響範囲を最小限に抑えます。
  - デプロイと変更に関する標準を実装しまう。
  - シンプルで直感的かつ一貫性があり、十分に文書化されたオペレーターインターフェイスを設計します。
  - オペレーショナルエクセレンスの目標を設定します。
  - 可用性がワークロードの重要な側面である場合は、新機能のリリースよりも安定性を優先します。
  - 過負荷にならないように、スロットリングまたは負荷分散、あるいはその両方を使用して使用量クォータを実装します。

障害を防止を完全に成功させることはできないことに留意してください。影響の範囲と大きさを制限し、理想的にはその影響を「ダウンタイム」のしきい値以下に保ち、非常に迅速で信頼性の高い検出と軽減に投資し、可能な限り最良の障害隔離機能を備えたソフトウェア設計に注力します。今日の分散システムでは、依然として障害を不可避なものとして受け入れ、高可用性を実現するためにあらゆるレベルで設計する必要があります。

付録 1 — MTTD と MTTR の重要なメトリクス

以下は、イベント中の MTTD と MTTR の削減に役立つインストルメンテーションと可観測性の標準化のフレームワークです。

カスタマーエクスペリエンスメトリクス。これらのメトリクスは、サービスが応答性が高く、カスタマーのリクエストに対応できることを反映しています。例えば、コントロールプレーンのレイテンシーです。これらのメトリクスでは、エラー率、可用性、レイテンシー、ボリューム、スロットルレートを測定します。

影響評価メトリクス。これらのメトリクスは、イベント発生時の影響範囲についての分析情報を提供します。例えば、データプレーンイベントの影響を受けたカスタマーの数や割合などです。影響を受けたものの数または割合を測定します。

運用健全性メトリクス これらのメトリクスは、サービスが応答性が高く、カスタマーのリクエストに応えられることを反映していますが、共通のインフラストラクチャサブシステムとリソースに焦点を当てています。例えば、EC2 フリートの CPU 使用率の CPU 使用率の割合です。これらのメトリクスでは、使用率、容量、スループット、エラー率、可用性、レイテンシーを測定する必要があります。

寄稿者

このドキュメントの寄稿者は以下を含みます。

- Amazon Web Services、Principal Solutions Architect、Michael Haken

詳細情報

詳細については、次を参照してください。

- Well-Architected の信頼性の柱
- Well-Architected 運用上の優秀性の柱
- Amazon Builders' Library — デプロイ中のロールバックの安全性の確保
- Amazon Builders' Library — 5 つの 9s とそれ以上:最も可用性の高いデータプレーンからの教訓
- Amazon Builders' Library — 安全で手間のかからないデプロイの自動化
- Amazon Builders' Library — 回復力のあるサーバーレスシステムの大規模な設計および運用
- Amazon Builders' Library — 高可用性デプロイに対する Amazon のアプローチ
- Amazon Builders' Library — 回復力のあるサービスを構築するための Amazon のアプローチ
- Amazon Builders' Library — 失敗を成功に導くための Amazon のアプローチ
- AWS アーキテクチャセンター

ドキュメント履歴

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
初版発行	ホワイトペーパーの初回発行。	2021 年 11 月 12 日

> **Note**
>
> RSS の更新を購読するには、使用しているブラウザで RSS プラグインを有効にする必要があります。

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。