
AWSでISRO実現!
その謎を解明すべく
Amazonの奥地へと
足を踏み入れる!!
Singular Perturbations Inc
Keisuke Nishitani

Keisuke Nishitani

CTO@ Singular Perturbations Inc

🎨 Programming is a creative work.

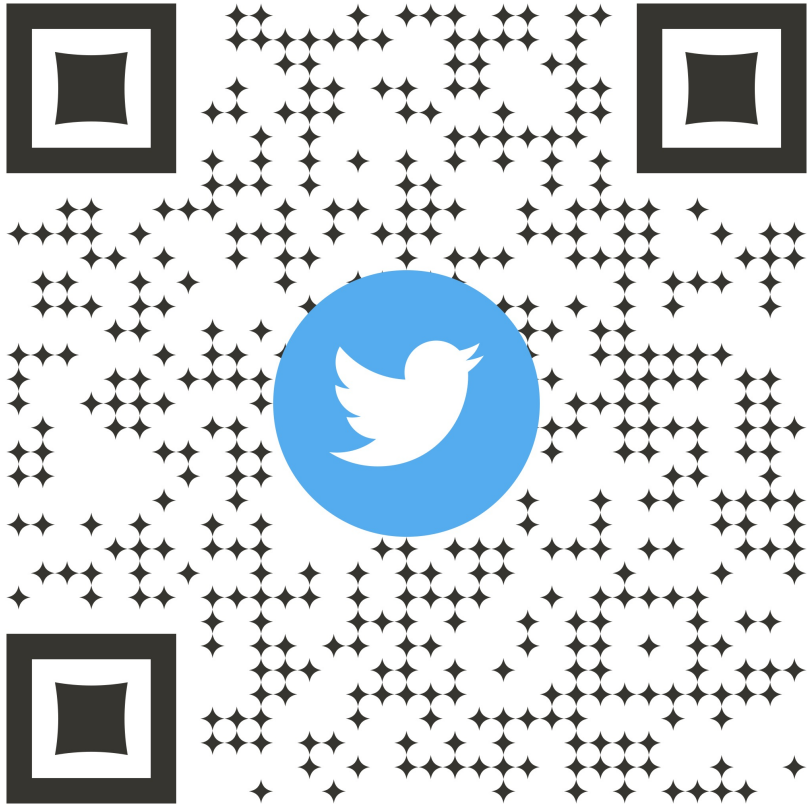
⚡ Everything will be serverless.

♫ Love Music

⛺ Love Camping

@Keisuke69

💻 Blog: https://www.keisuke69.net/



👨‍💻 カジュアル面談:https://meety.net/matches/kBtcRNFYzKEI

Singular Perturbations


VISION
世界の悲しい経験を減らす。

MISSION

コンピューターサイエンスがもたらす知能を安全に関わる全ての人へ

CRIME NABI

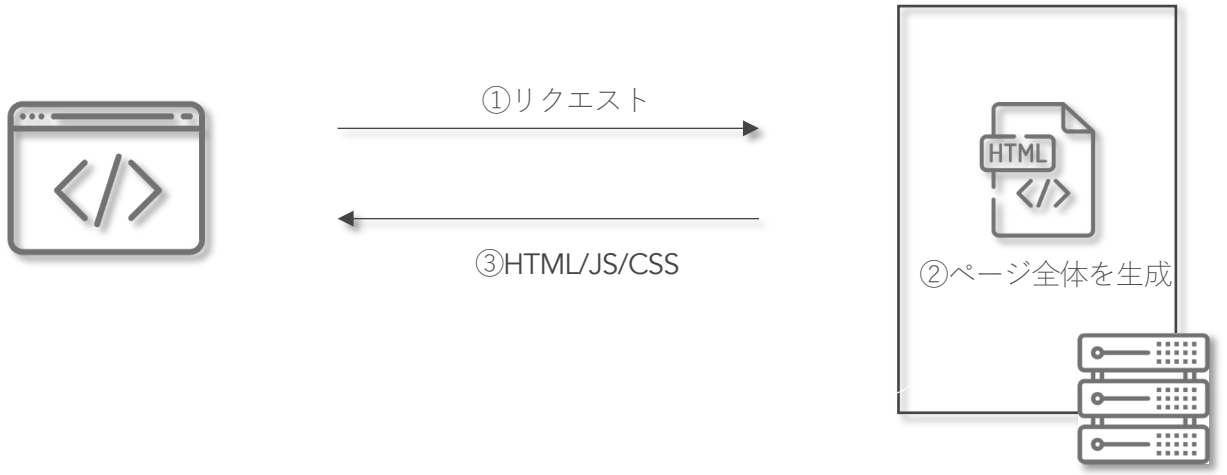
Patrol Community


今日のテーマ


Incremental Static Regenerationが実行可能な環境を
AWSで構築する方法
※実際にはAPIがいたり、GraphQLがあったりもするけどそこは話しません

フロントエンドWebのおさらい

トラディショナルWebアプリ



モダンなフロントエンドWebアプリケーション

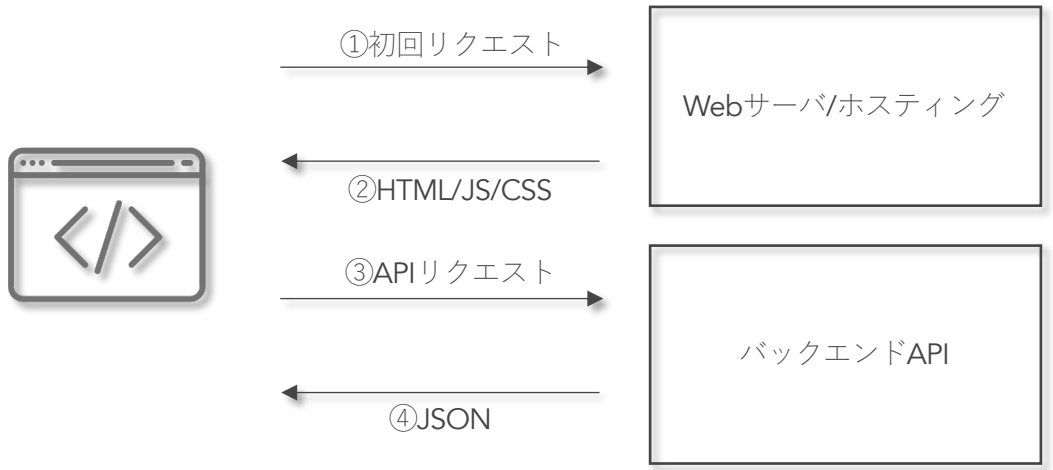
Single Page Application

Single Page Application (SPA)

- 単一のHTMLページで構成されるWebアプリケーション
  - 1枚のHTMLに置かれたJavaScript (およびTypeScript)もしくはアプリケーションですべてが成立

- HTMLのレンダリングおよびリクエストのルーティングがクライアントサイド（≒ブラウザ）で行われる
  - 同一のページ内で行われる
  - ページ遷移がサクサク動く

- React / Vue.js /Angularあたりが有名



SPAの課題

- SEO周りが課題になりがち
  - HTMLがリクエストされてもJSがロードされて処理が実行されるまでは真っ白。ここ大事。
  - 検索エンジンのクローラーが仮にJavaScriptを解釈しないものだと真っ白な一枚の何もないものを受け取るだけ
  - 最近ではGoogleのクローラー（Googlebot）はJavaScriptの解釈と実行も可能
  - GooglebotはレンダリングエンジンとしてHeadless Chromiumを用いてJavaScriptを実行するが、一昨年まではChrome41相当の古いものであったためそもそも正しく動いてくれない、つまり正しいHTMLを認識してくれないという問題があった
  - 現在は最新のChromeに常に対応していくことが発表されている
- ファーストビューが遅い
  - UXの問題に加えて対クローラーの観点でも問題になりうる
    - 例えば非同期に他のAPIをコールしてデータを取得してコンテンツを生成する場合にローディング中の表示やスピナを表示することが多いがクローラーは最後まで待ってくれないケースが多い
- 動的なOGP対応が難しい
- Google以外のbot等ではJavaScriptを解釈して実行してくれないものもある

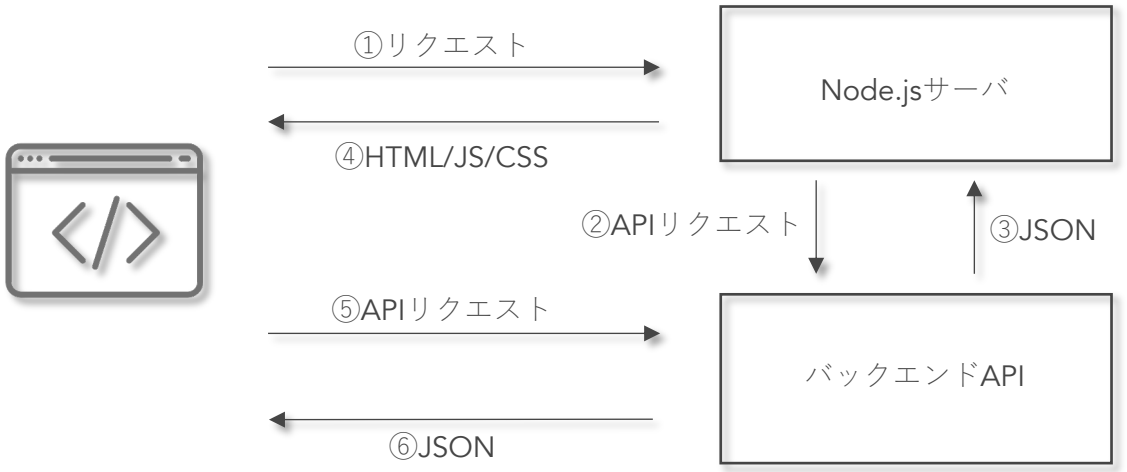
Single Page Application

Server Side Rendering

Server Side Rendering (SSR)

- 端的に言うと初回のHTMLをサーバ側でレンダリング
  - 言葉とおりレンダリングのためのサーバを用意して、必要に応じてAPIリクエストを実行し、HTMLをレンダリング
  - 2回目以降はSPAと同様
  - ReactだとNext.js、VueだとNuxt.jsあたりが著名

- SPAで課題になったクローラーによるアクセスであっても正しいコンテンツを返せるようになる
  - 動的なOGPの問題について同様



SSRの課題

- やはりサーバが必要になってくること

- サーバが必要になるため、それに伴うインフラ的な課題が生まれてくる
  - CPU負荷高くなりがち
  - さばけるリクエスト量が少なくなりがち
  - キャパシティ不足になってしまうとレスポンスが返せなくなることがあり、そうなるとブラウザ上では画面が真っ白に

Single Page Application

Server Side Rendering

Static Site Generation

Static Site Generation (SSG)

- JSで構築されたページを事前にすべて静的なページとして生成
  - 静的サイトとして配信可能
  - 動的な処理がないためSSRと比べると軽量であり高速
  - 動的な部分がないためセキュアになる
  - 配信もシンプルになりWebサーバだけで可能。スケールも容易
  - 各種ホスティングサービスも利用可能。Amazon Simple Storage Service (S3)も
- Gatsby、HugoなどのStatic Site Generatorを利用するだけでなく、Next.jsやNuxt.jsでも可能
  - JamstackもSSGが前提

SSGの課題

- ビルドプロセスの整備が重要

- コンテンツの更新のたびに全体のビルドが必要となる
  - コンテンツの数が増えると時間かかる
  - CMSなんかで入稿されたらまるっと再ビルドが必要になる

- 当然ながら動的なコンテンツは処理できないためリアルタイム性の高いコンテンツには利用できない

Single Page Application

Server Side Rendering

Static Site Generation

Incremental Static Regeneration

Incremental Static Regeneration (ISR)

- リクエストに対して静的にビルドされたページを返しつつ、有効期限が過ぎたら非同期で静的ページの再生成を行う

- Cache Controlにおけるstale-while-revalidateと同様の考え方が適用されたもの
  - Next.js 9.4から追加された機能

- SSRとSSGそれぞれの辛みをある程度解消してくれる
  - ISRでは静的ページの事前ビルドをしつつ、キャッシュを有効活用しつつ、静的ページの更新を自動的に行う
  - 動的に近いこともできるようになった
  - SSGのようにサイト全体をリビルドする必要がない

- 次世代のStatic Site Generationと言っても過言ではない（西谷談）

シンプルなサンプル

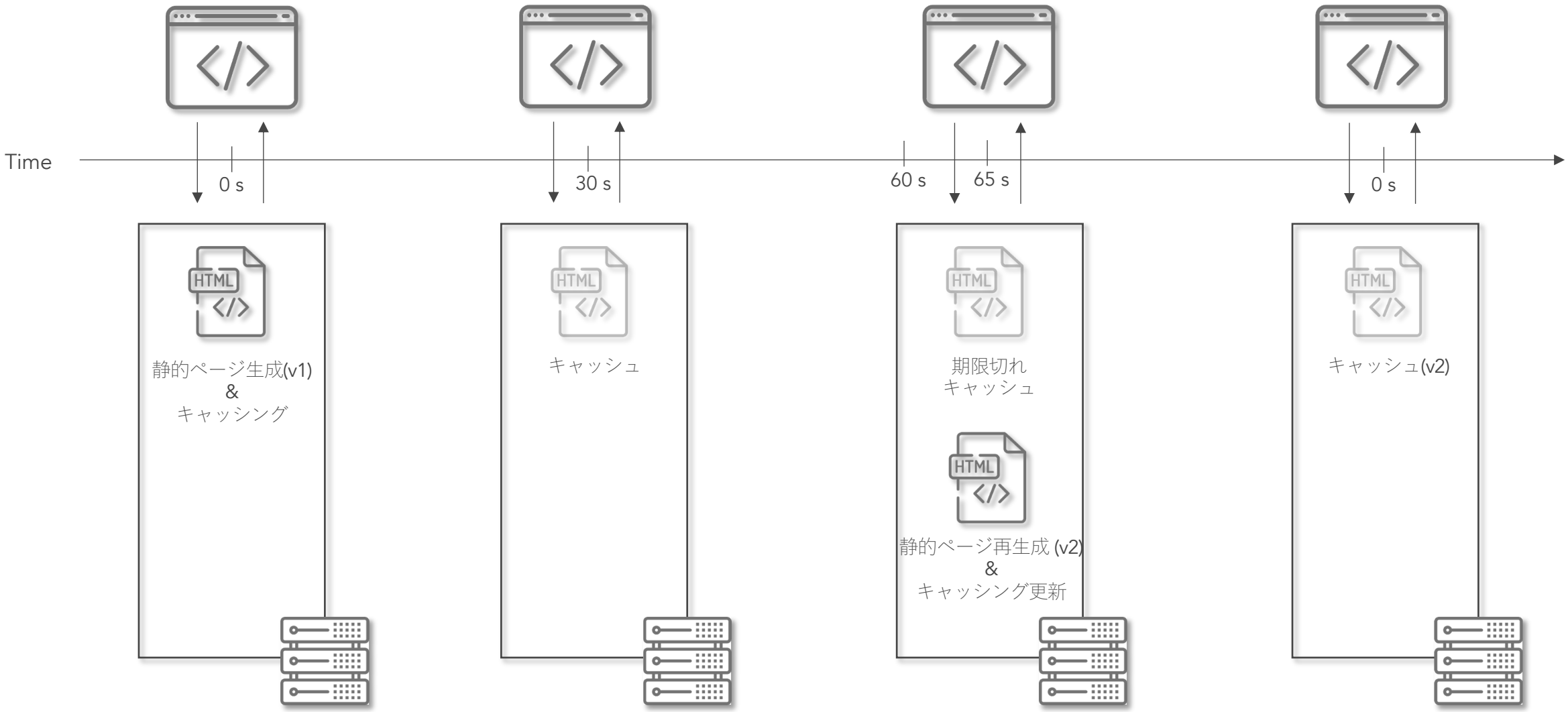
```
export default function Index({current}) {
  return (
    <div>
        現在時刻は{current}です。
    </div>
  );
}

export async function getStaticProps() {
    const date = new Date();
    const current = date.toLocaleString()
  return {
    props: {
      current,
    },
    revalidate: 10,
  };
}
```

シンプルなサンプル

- SSGと同様にgetStaticProps()を利用
  - Revalidateの指定により、指定したインターバルでビルドし直される

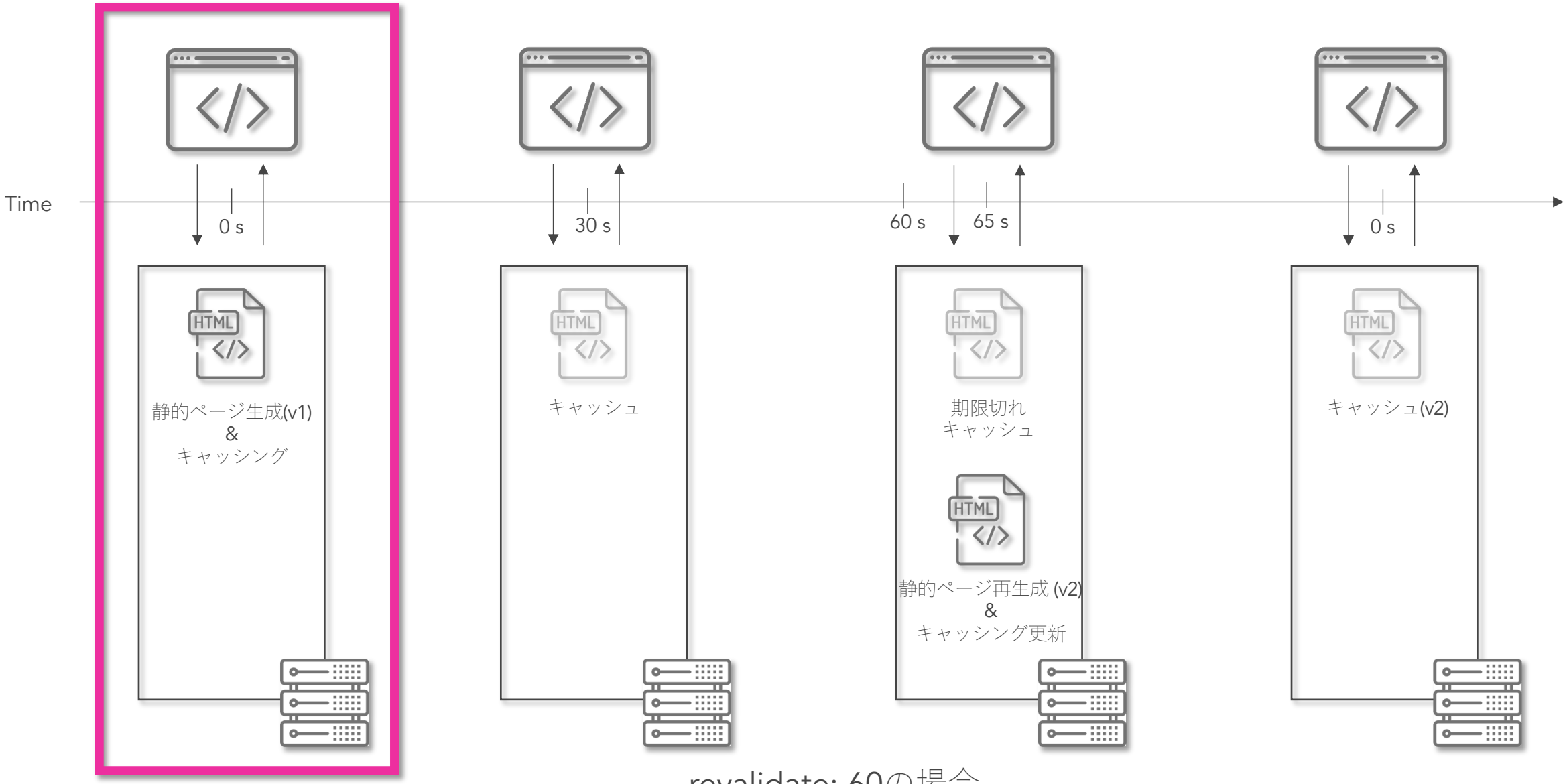
- 動的ルーティングする場合はこれに加えてgetStaticPaths()で fallback: trueを指定する必要がある

Incremental Static Regeneration



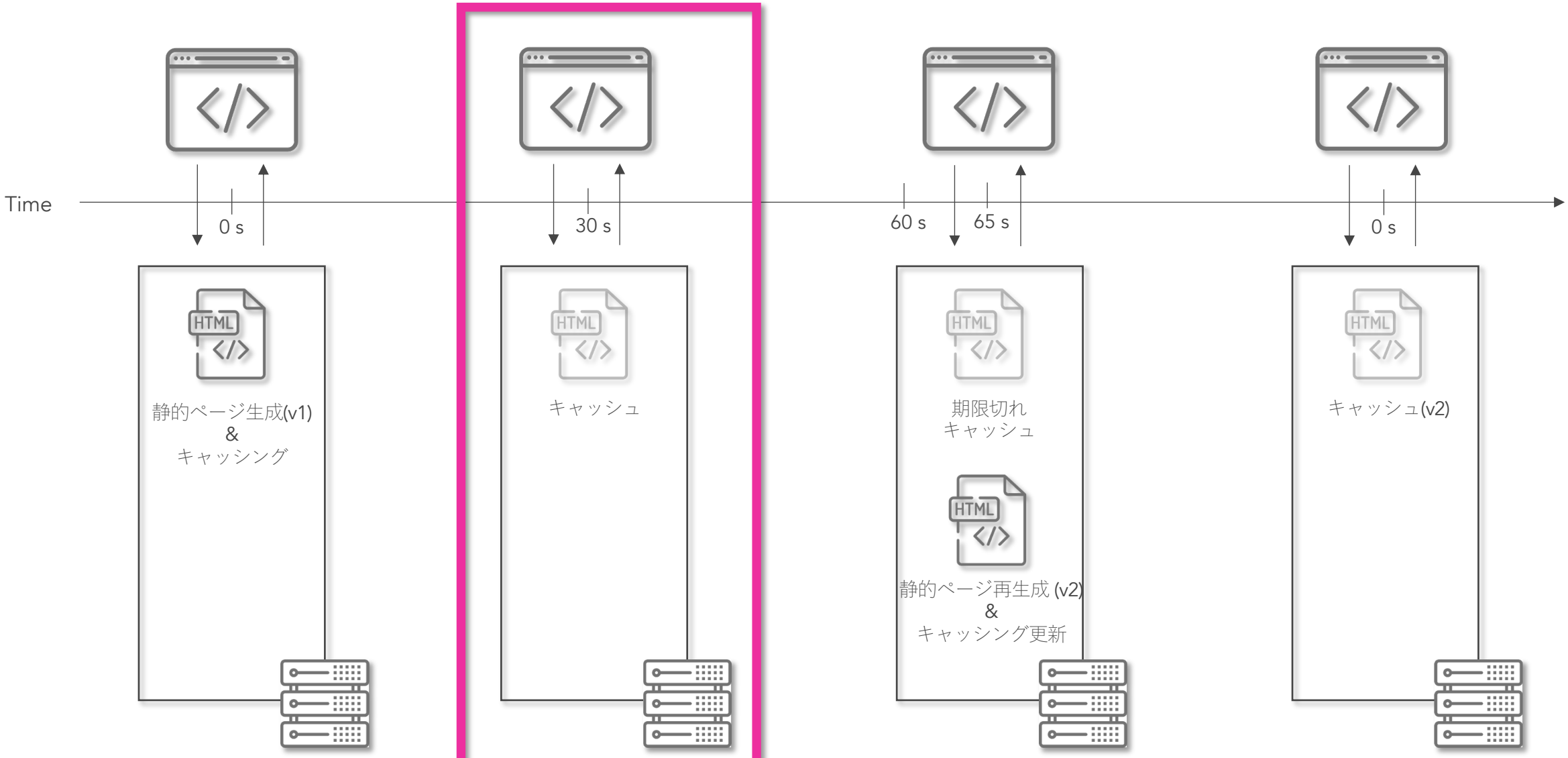
revalidate: 60の場合

Incremental Static Regeneration



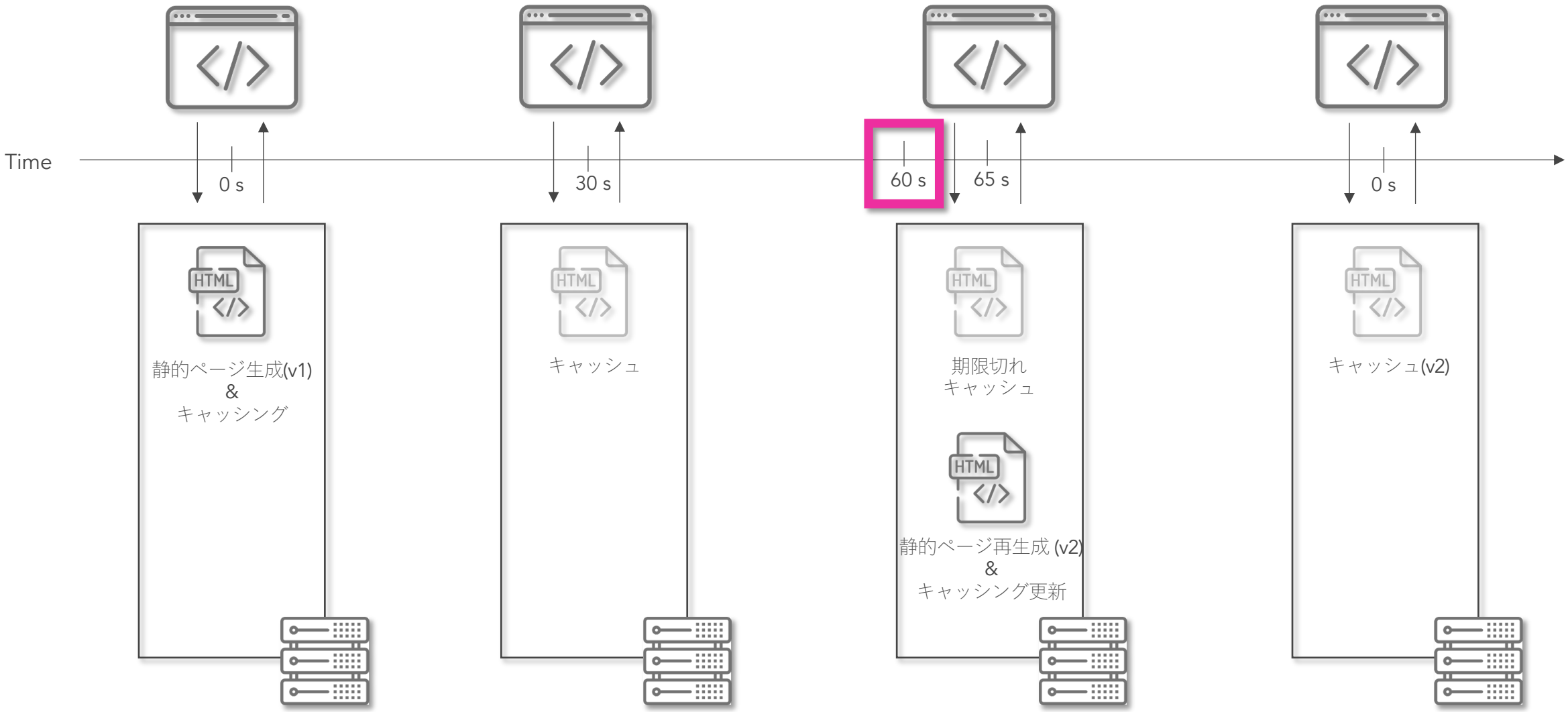
revalidate: 60の場合

Incremental Static Regeneration



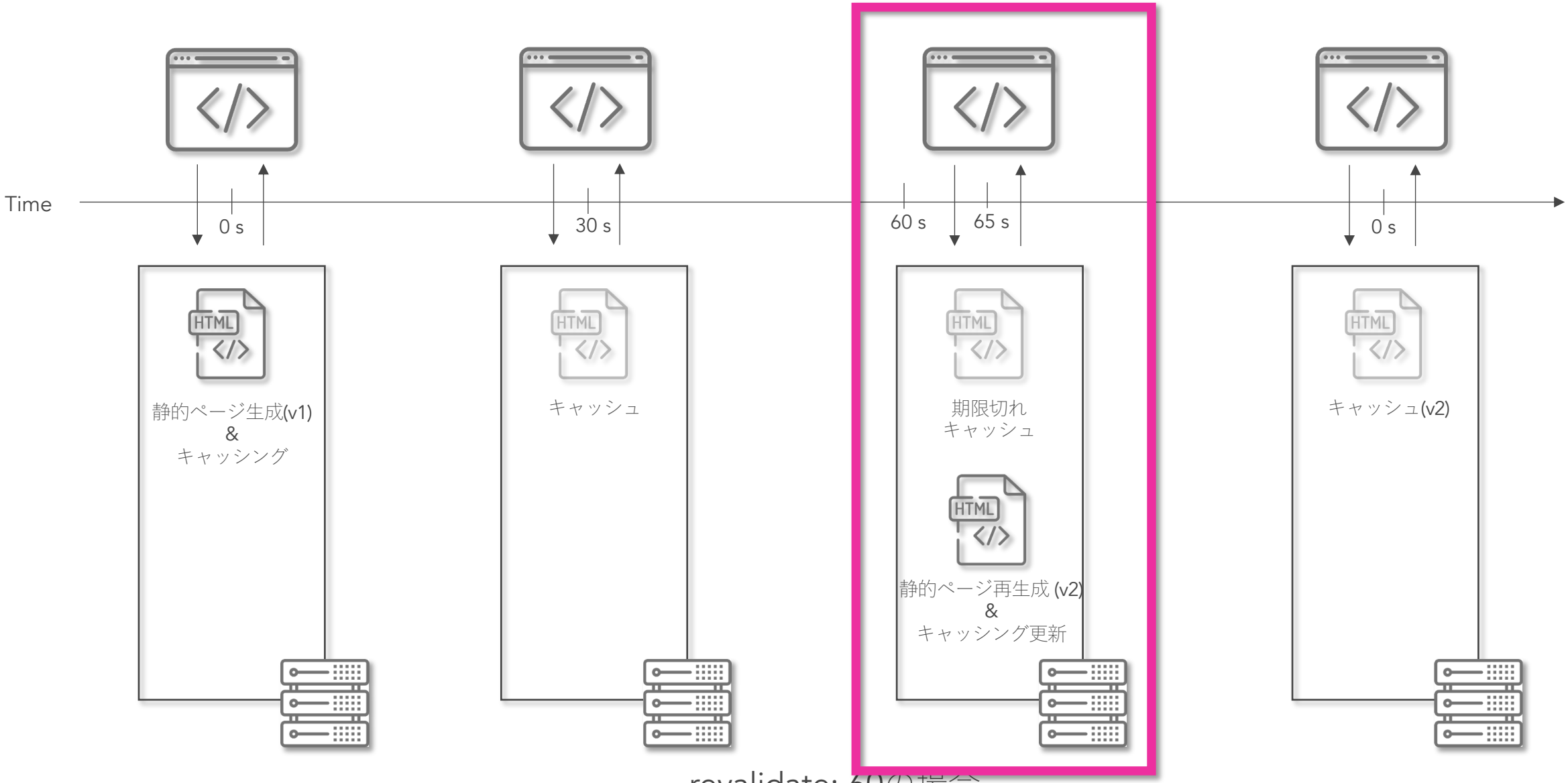
revalidate: 60の場合

Incremental Static Regeneration



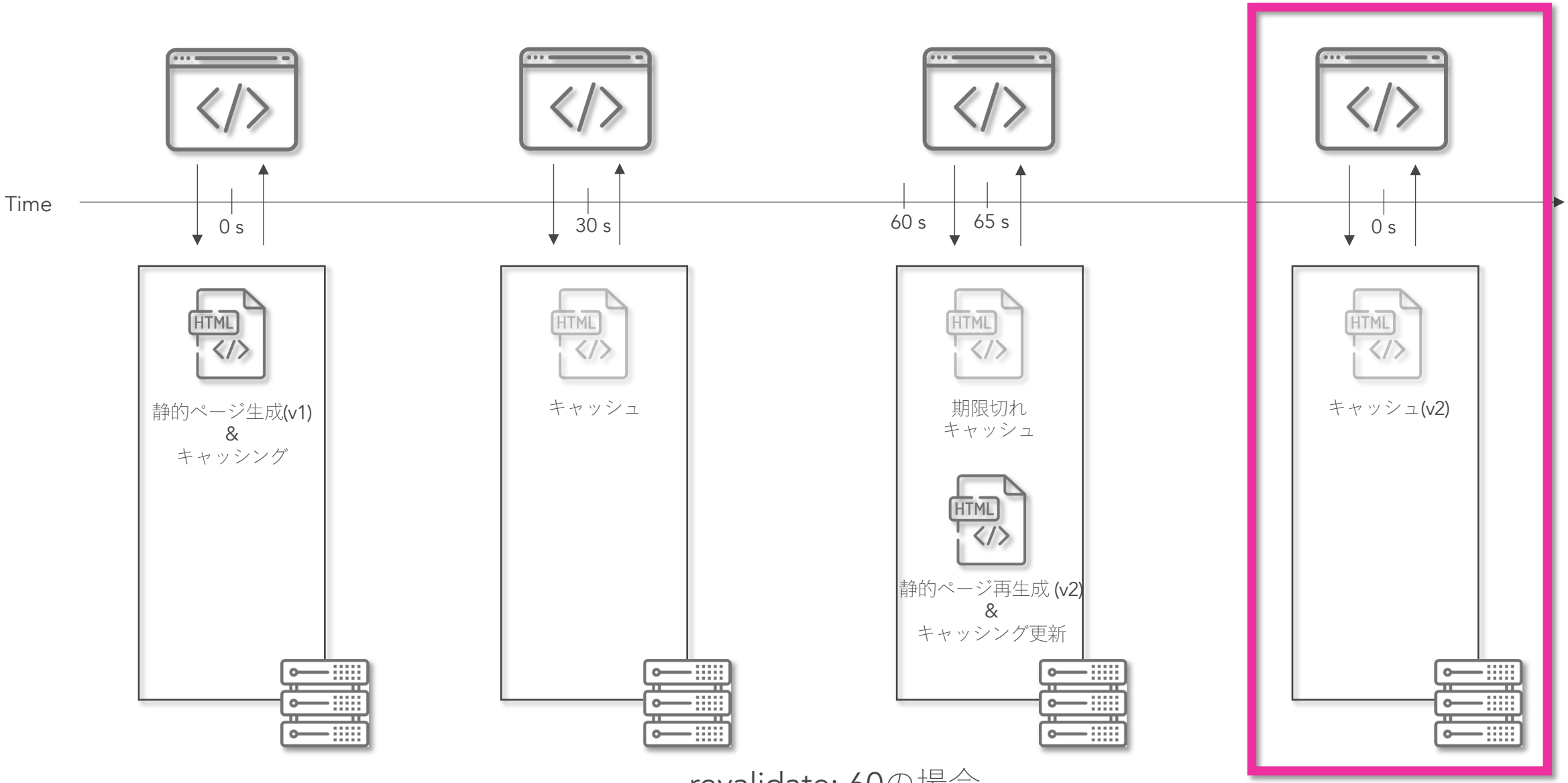
revalidate: 60の場合

Incremental Static Regeneration



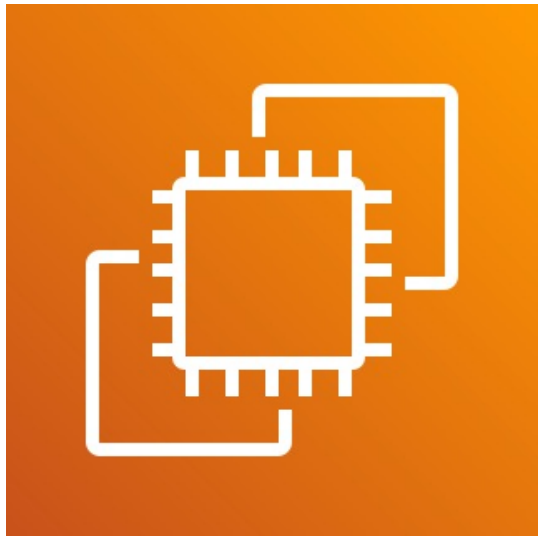
revalidate: 60の場合

Incremental Static Regeneration



revalidate: 60の場合


そんなISRなフロントエンドアプリをAWSでホストするには?



Amazon Elastic Compute Cloud
(Amazon EC2)

一応こんなのもある

AWS Elastic Beanstalk

Amazon Lightsail

Containers

Amazon Elastic Container Service
(Amazon ECS)

AWS Fargate

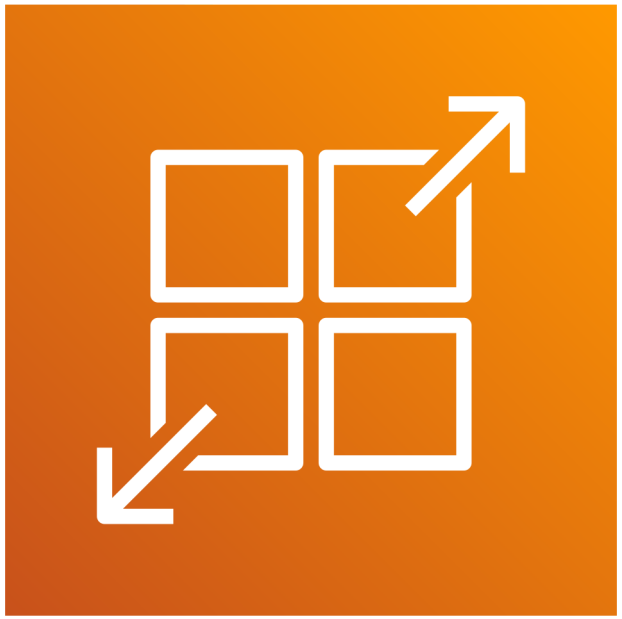
Amazon Elastic Kubernetes Service
(Amazon EKS)

コンテナの利用

- FargateでもECSでもEKSでもk8s on EC2でもOK

- ISRはnext startをサポートしているのでNode.jsをセットアップしたコンテナとWebサーバとなるコンテナがあれば動く
  - つまり、通常のWebアプリケーションと同じ
- Pros
  - 一番わかりやすい（≠ 楽）
    - 従来型の構成を再現しやすい
  - コンテナならではのポータビリティ、柔軟性、軽量なことによる起動と開発プロセスの高速化

- Cons
  - EC2よりはマシとはいえ、DockerfileやTask Difinition、Podといったコンテナ環境の定義を運用管理していくのは手間
  - ランタイムの管理が必要
  - AWSやインフラよりの知識がアプリケーション開発者に求められる
  - キャッシュが分散し、反映にタイムラグなどが発生する



AWS App Runner

AWS App Runner

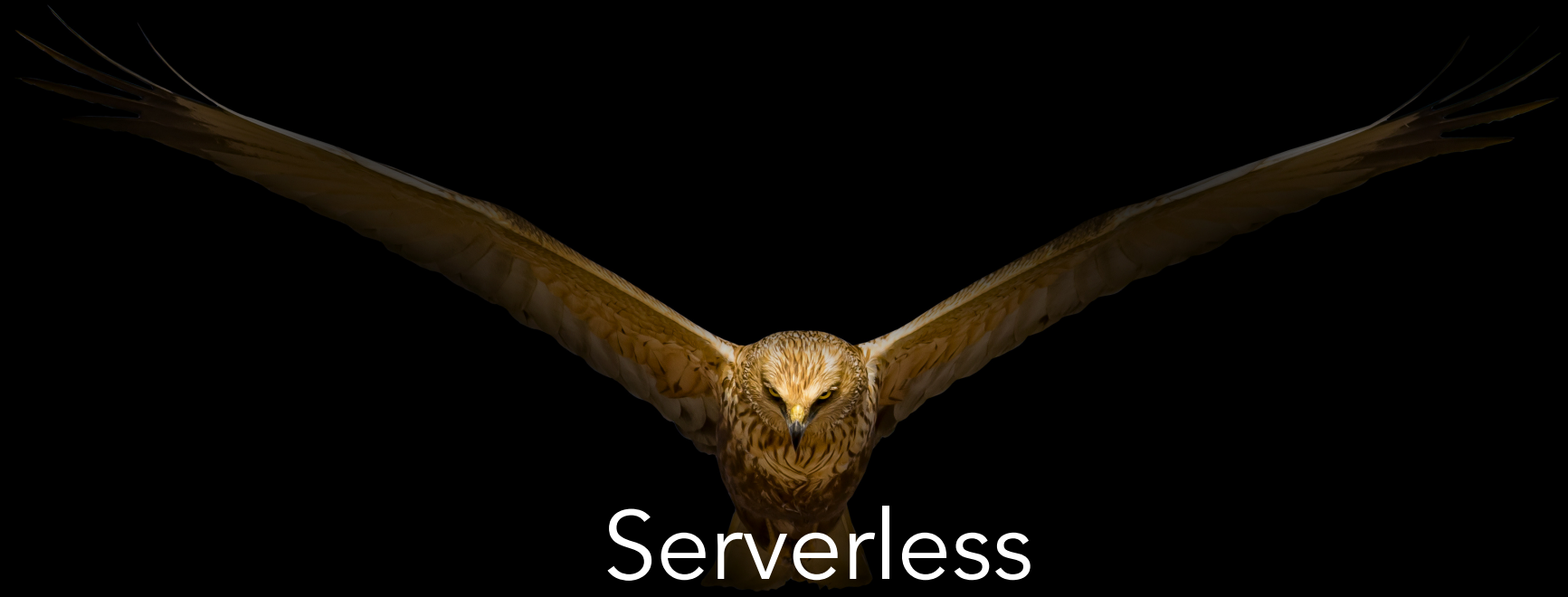
- コンテナベースで作られたWebアプリのフルマネージドサービス
- 簡単、手軽に、大規模なWebシステムのビルドとデプロイが可能
- Pros
  - コンテナサービス各種で必要となるDockerfile以外の定義ファイルたちが不要
  - デプロイパイプラインまで用意してくれる
  - AWS Lambdaと異なりコールドスタートがない
    - 最小インスタンス数が1
  - インフラ知識不要
    - VPCとかLBはもちろんオートスケーリングとかも意識しなくていい、一応
- Cons
  - 最小インスタンス数が1なので完全にゼロスタートはできない?
    - ただ、課金は考慮される
  - VPC内のリソースにアクセスできない (2021年9月30日時点)
    - Amazon Relational Database Service (Amazon RDS)とかAmazon Elastic File System (Amazon EFS)が使えない
  - 当然ながら各AWSサービス単位での細かいチューニングは難しい
  - キャッシュの問題などコンテナで運用する場合と同じ問題がある


キャッシュの問題とは？

キャッシュの問題とは

- ISRでは仕組み上、サーバ内部でキャッシュが保持される
- つまり、コンテナごと、インスタンスごとに別のファイルになる
- ロードバランサからはどのインスタンスにいくかわからない
  - 最初の生成が行われるタイミングがバラバラ
  - revalidateもコンテナごとにバラバラ
- ロードバランサによるリクエストの転送先とrevalidateのタイミング次第ではなかなか更新されないといった状況が発生しうる
  - 極端な例として、revalidateの時刻に達していないコンテナにばかりリクエストが転送されてしまった場合


DANGER



Serverless

サーバーレスを利用することの意味

- AWS Lambda等のサーバーレスなサービスを使うことによるメリット
  - コンテナとかインフラそのものの管理をしなくていい
  - CPU負荷が高くなりがちな点とスケーラビリティ上の難点を解決してくれる
  - AWS Lambdaの場合、1リクエスト（＝イベント）に対して1インスタンス（＝コンテナ）が対応
  - 同時にリクエストが来た場合もそれぞれ別のインスタンスが対応するため、リクエストが集中してサーバに処理が集中してCPU負荷が高まった結果クライアントにレスポンスが返せなくなるみたいなことが発生しない

- しかし、AWS Lambdaで実装を頑張るのは現実的ではない
  - ISRは仕組み上内部でキャッシュが保持されるため、AWS Lambdaの関数モデルで実現するのは難しい

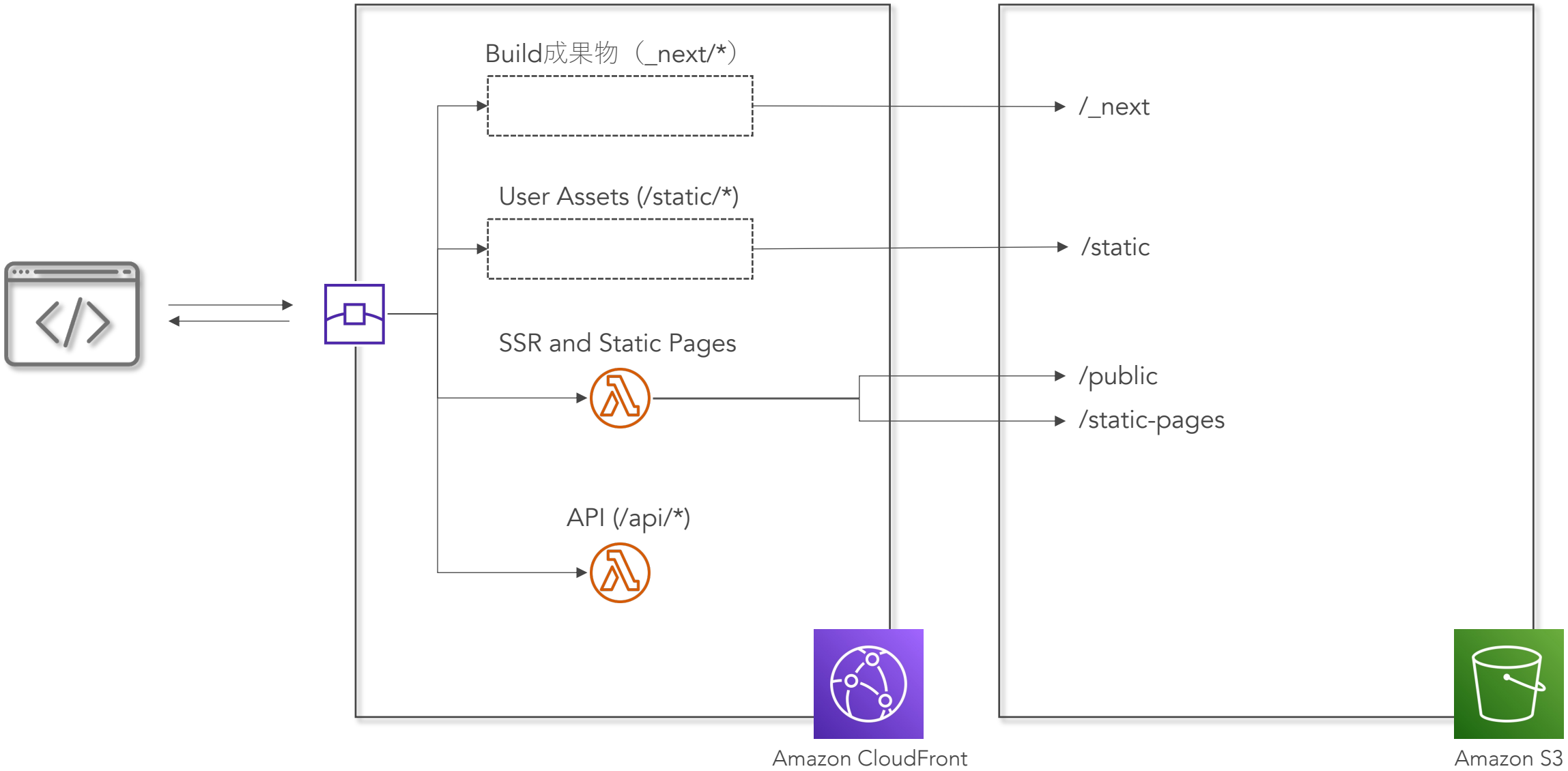
Serverless Next.js Component

- AWSを始めとするサーバーレス環境のデプロイやプロビジョニングツールであるServerless Frameworkの関連機能であるServerless Componentの一つ

- Next.jsをサーバーレスで実行可能にする
  - Lambda@Edgeで実現
  - 静的なアセットや静的ファイルとして生成されたものはS3でホスティングするようにデプロイした上でCloudFrontを使って配信される
  - Lambda@Edgeでルーティングや SSG、SSRのハンドリングといったことを自前で実装するのはコストが高い
  - Zero configurationを謳っており、SSGやSSRはAPIで自動判別して対応
  - Image Optimizationにも対応
  - ISRにも対応

- Lambda@Edgeで静的ページへのリクエストをハンドリングしてSSRしないリクエストはS3へとフォワード
  - Dynamic Routingもサポート

Serverless Next.js Component



Serverless Next.js Component

- Origin ResponseでISRのページかどうかをチェック
  - Expiresヘッダがあるか、revalidateプロパティがあるか
- Expiresヘッダーをチェック
  - 将来の値であればcache-controlヘッダを適用しエッジに保存
  - 過去の値であればページの再生成をトリガー。同時に既存ページをno-cacheで返す
- 再生成のためにSQSのFIFO キューにメッセージをpublishし、Lambda関数で非同期に生成


もっと楽に...


AWS Amplify

AWS Amplify Console

- Library、CLI等からなるAWS Amplify のサービス群の1つ
  - 元来は静的サイトのデプロイパイプラインとホスティングが守備範囲
  - GitにPushしたらビルドが実行されてデプロイされるというフローを簡単に実装できる
  - SPAやSSG/Jamstackに向いている

- Next.jsをサポート
  - 2021年5月にNext.js 9のSSRをサポート、7月に10/11に対応
  - SSRだけでなくIncremental Static Regenerationに対応
  - Image Optimization(next/image)とScript Optimization (next/script)も対応
  - Serverless Next.js Componentをベースにしている模様

- 現状では一番簡単だと思う（西谷談）

Wrap up

- AWSでISRを実現する手段は複数ある
  - EC2からサーバーレスまで
- EC2やコンテナなどで実現する場合はキャッシュの問題がある
  - 特にスケールする場合に問題になる
- 実装の簡単さではAWS Amplify ConsoleとAWS App Runner
  - Serverless Next.js Component
- 結論として、AWSでISRを実現するには現状ではAWS Amplify Console or Serverless Next.js Component
  - 個人的にはAWS Amplify Console一択だがAWS WAFを使いたい場合などはServerless Next.js Componentもあり
  - コンテナでの実装はキャッシュの問題があって現実的に使い物にならないのでは


OMAKE

stale-while-revalidate

- キャッシュコントロールの一種
  - ブラウザやCDNにおけるキャッシュにおいて、一定期間はキャッシュからレスポンスするが指定時間を経過したら非同期でオリジナルをfetchしてキャッシュをレスポンス/更新する
  - ISRだけでなく、データ取得にフォーカスしたSWRというライブラリもある
- Cache Controlをサポートするcdnであれば実は同じようなことができる
  - Next.js側ではISRをせずとも通常のSSRでよくなる
  - そもそもNext.jsとかSSRとか関係なくなるという期待
- しかし、AWSのCDNであるCloudFrontは現状ではstale-while-revalidateには対応していない（stale-if-errorには対応している）


終


TEAM
仲間募集中！
興味ある方はDMください！