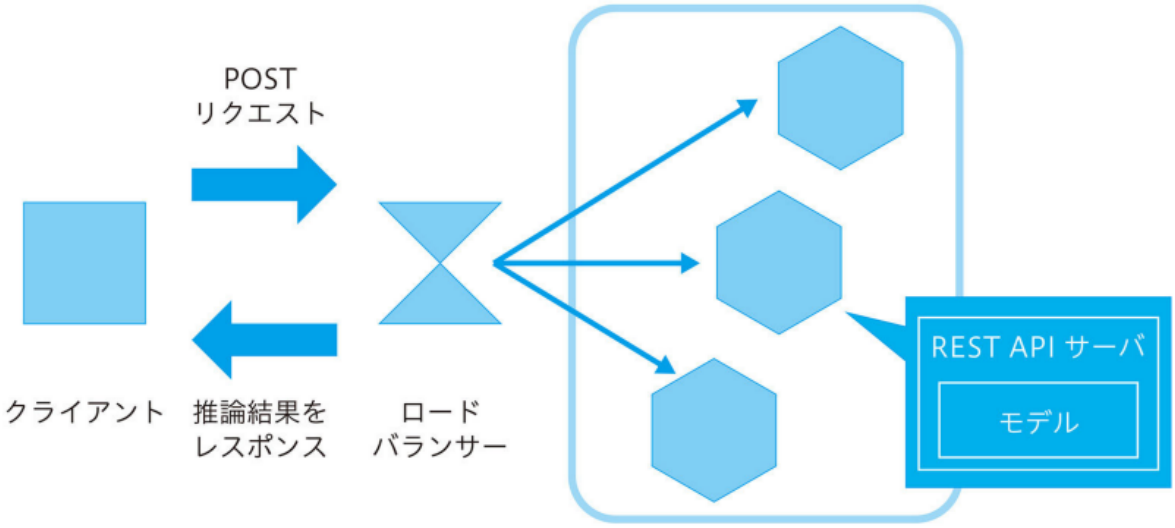
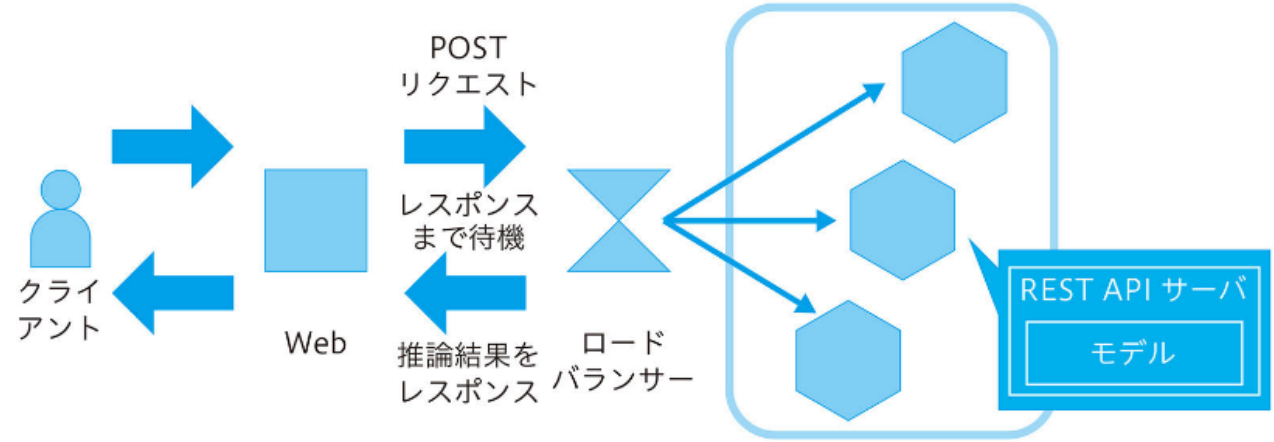
システムパターン分類

- 基本的なパターン
    - Webシングルパターン
      Webアプリケーションサーバにモデルを同梱させるパターン。同一のサーバに学習済みの機械学習モデルの推論器を実装するため、DBやストレージなどの構成を考慮する必要がなく、複数台のwebサーバを運用する場合にはロードバランサーを導入して不可を分散する仕組みにすることができる。概要については以下の図を参考のこと。

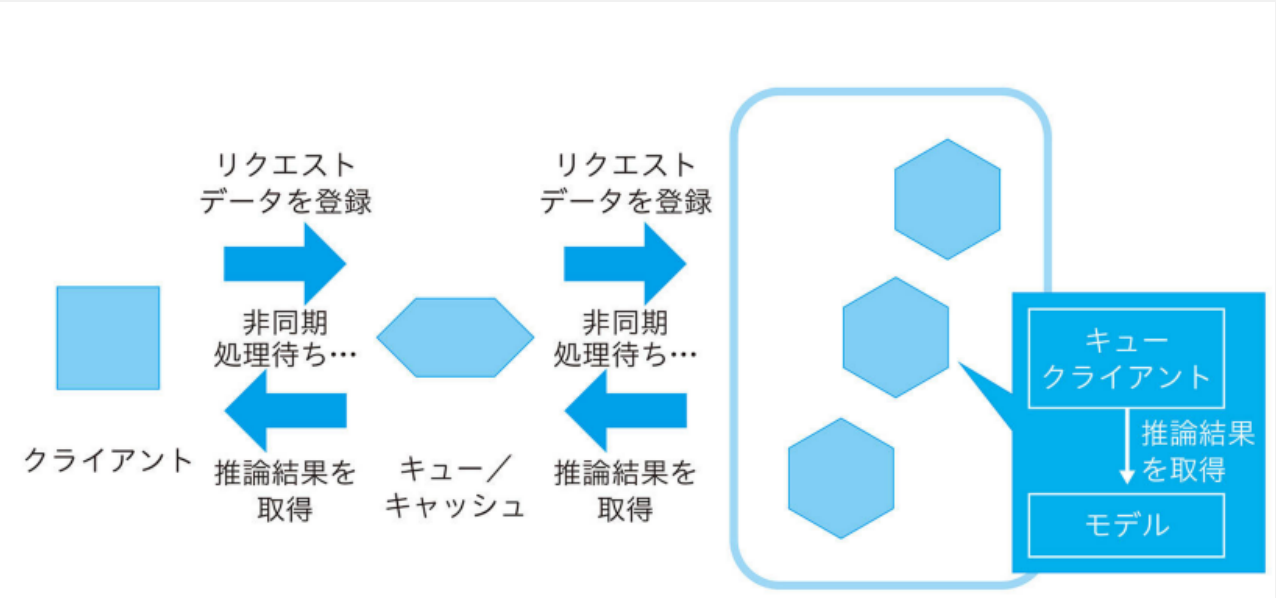


- メリット
    - 推論器を素早く稼働させることが可能
    - 汎用的な構成で、運用負荷も低い
    - 障害対応なども容易
- デメリット
    - 複数の推論器やフレームワークを組み合わせるのが難しい
- リクエスト時の処理パターン
    - 同期推論パターン
      クライアントは推論のリクエストを送信し、そのレスポンスが返ってくるまで次のリクエストを送らない。
        - メリット
            - シンプルな構成
            - 順序立ててワークフローを作ることが可能
        - デメリット
            - 推論器がレスポンスまでにクライアントの待機時間が増加する可能性がある。

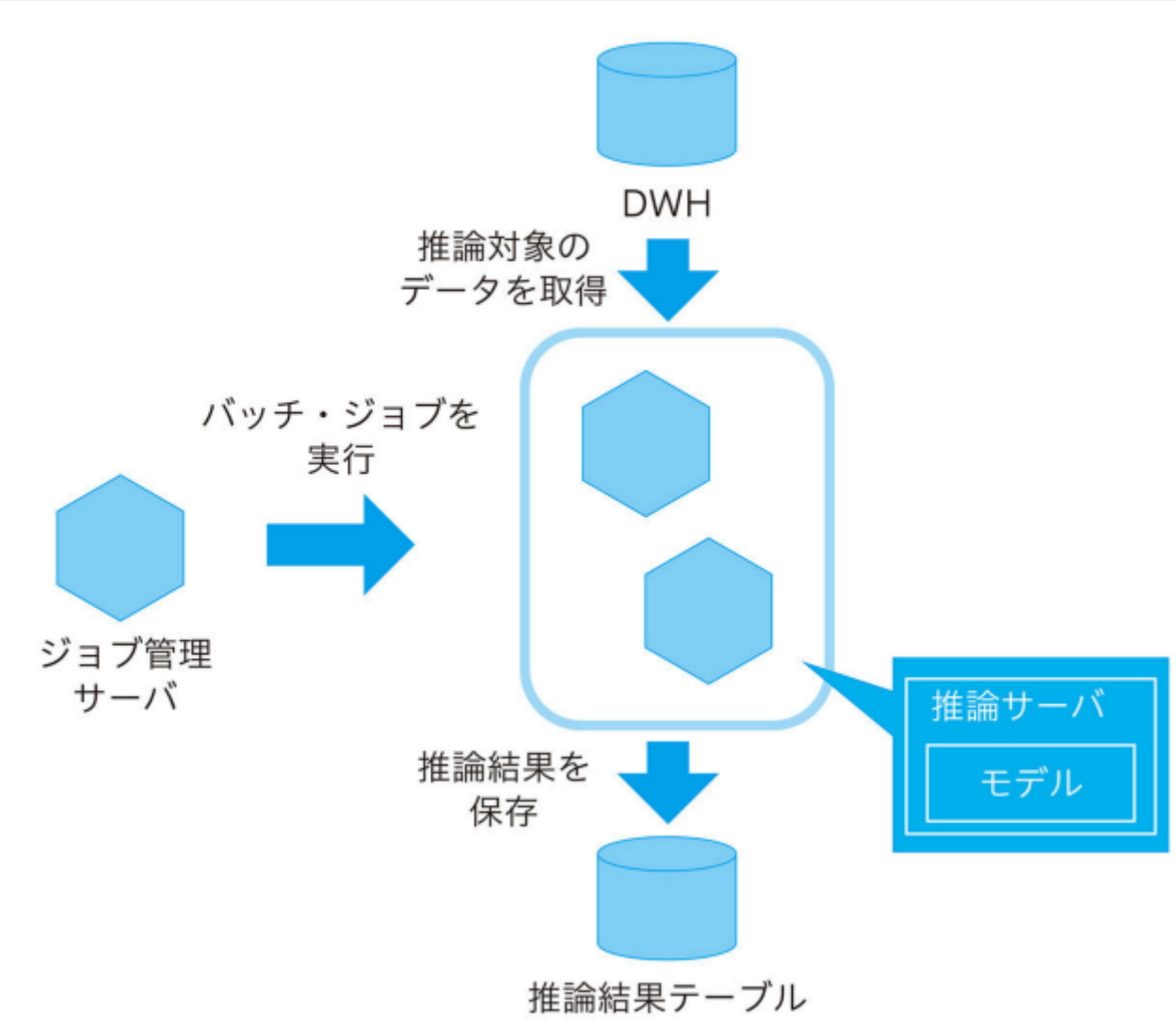
- 非同期推論パターン
  クライアントのリクエスト処理と推論器側のリクエスト処理を分離する。クライアント側からリクエストが送られた場合にも直接推論器にデータを送信するのではなく、キューやキャッシュに一度データを格納する。（詳細は図を参照）このような構成にすることでワークフロー側の処理は推論器側の処理が完了するのを待つことなく実行できる。
    - メリット
        - クライアント側のワークフローと推論側のワークフローを疎結合にすることが可能
        - 推論がある程度遅い場合でもクライアント側の悪影響を回避可能
    - デメリット
        - Queue方式で処理することを検討する場合、障害などが発生した場合には一部の処理が行われないなどのエラーにつながる可能性がある
- バッチ推論パターン
  リアルタイムでクライアント側のリクエストを処理する必要がない場合にはジョブをまとめてバッチ処理の形で推論を実行する構成。
    - メリット
        - サーバのリソース管理を柔軟に行うことでコスト削減が可能
    - デメリット
        - バッチ毎に処理するデータ量や実行頻度の調整が必要

> 同期推論パターン
>
> 
>

非同期推論パターン



バッチ推論パターン



- 処理の分割のパターン
  複雑な処理や複数の推論器を必要とするシステムを構築するにあたって、

- 前処理・推論パターン
  一つのモデルを推論器として使用する場合であっても機能ごとにサーバを用意して構成する場合がある。

    - メリット
        - 前処理と推論器でサーバごと分離することにより、リソースの効率化や障害の切り分けが可能
        - 推論器側と前処理側のサーバでそれぞれ独立する形でバージョン管理が可能になる。

- 直列マイクロパターン
  複数の推論器を利用し、かつその実行順序や依存関係が決まっている際に、複数の推論器を組み合わせて一つのワークフローとする構成。プロキシをクライアントと全推論器との仲介として配置し、クライアント側や各推論器側のリクエストを順番に処理するような構成。

    - メリット

        - 各推論器モデルを順番に実行することが可能
        - 前の推論モデルの結果次第で次モデルへのリクエスト先を選択する構成も可能
        - 推論器側のリソースの効率化や障害の切り分けが容易に
        - 推論器毎のスケールアウトや更新が容易

    - デメリット

        - 各々の推論器の処理が早くてもサービス全体として遅くなる場合がある。
        - サーバ間の障害が考えられる。

- 並列マイクロパターン
  依存関係のない複数の推論器を並行で実行する際に、推論器を並列に並べ、個別に推論リクエストを送る構成。基本的には、仲介するプロキシを置くことで、クライアント側にデータ取得や集約などのタスク負荷を書けないような構成にするのが一般的。また、実装においては同期、非同期のいずれかの処理方法を検討する必要がある。

    - メリット

        - 推論サーバの分割やリソース調整が容易
        - 推論のワークフロー間で依存関係を持たせず構築が可能

    - デメリット

        - 同期的に処理する場合は最も遅いレスポンスに待機時間が引っ張られる。
        - 非同期的に処理してタイムアウトや優劣を設ける場合、推論に時間がかかるが、有益な結果をもたらす推論器について結果を反映できなくなる可能性がある。
        - 運用が難しい

- 時間差推論パターン
  インタラクティブなアプリケーションやレスポンスの速度が違う推論器を活用する場合に利用される。並列マイクロパターンと似たような構成ではあるが、複数の推論結果を集約する必要がない場合などに利用される。

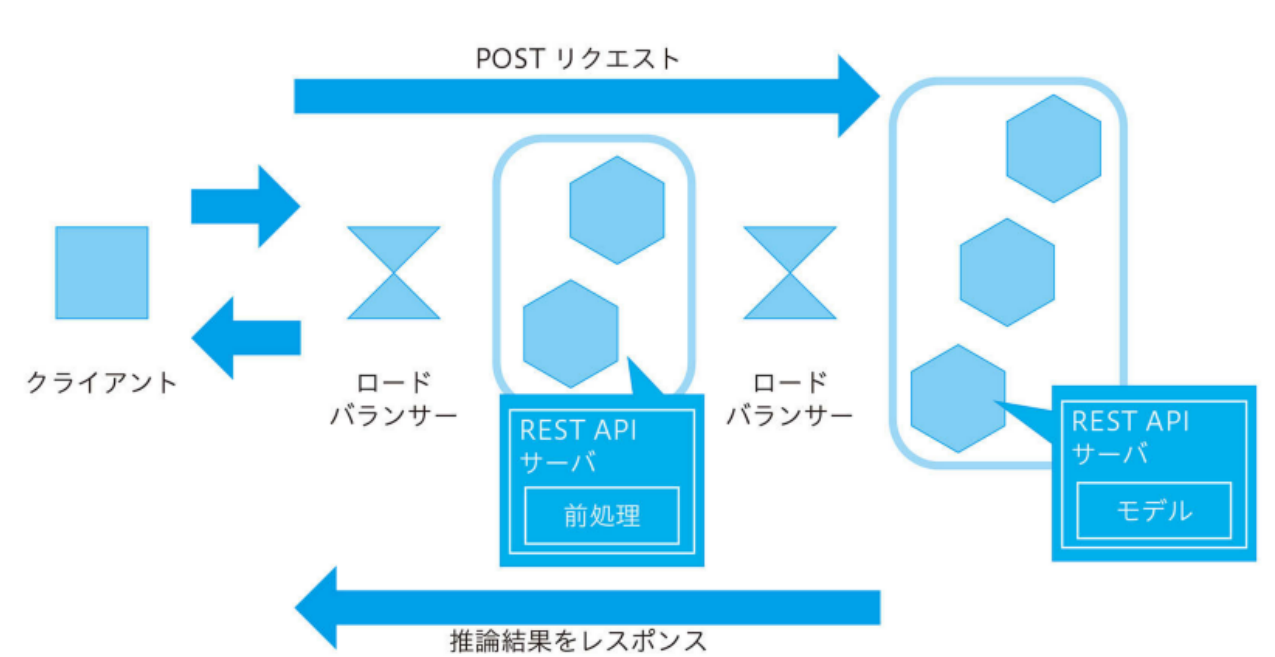
    - メリット

    - 柔軟かつ素早いレスポンスを確保できる一方、可能な限り良い推論結果を提供することが可能。

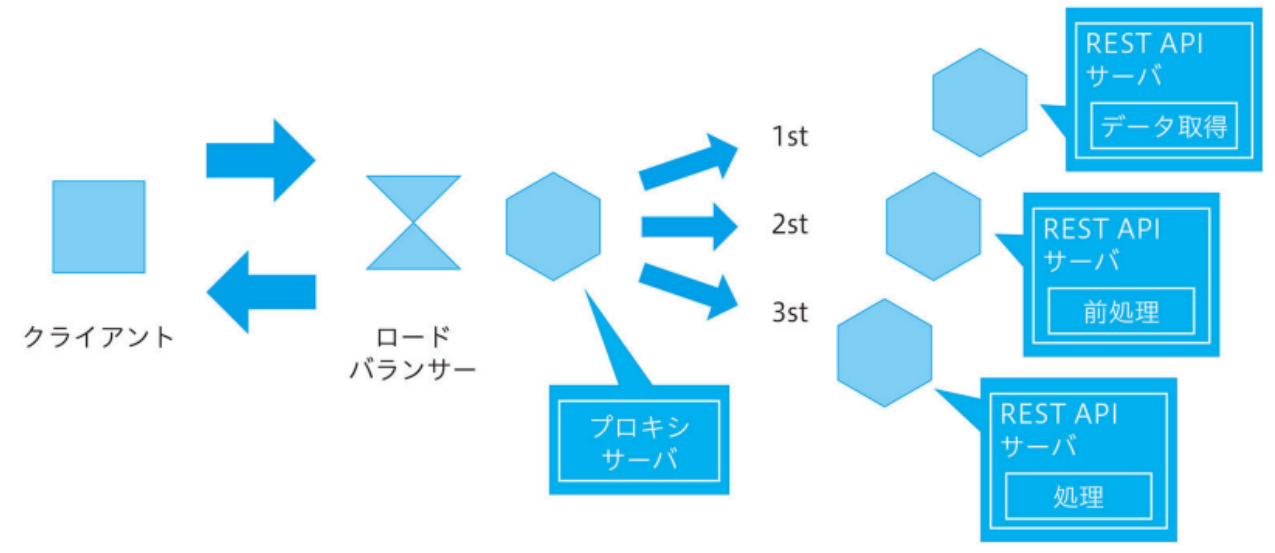
- デメリット

    - ユーザー体験の向上を検討する必要がある場合、待機時間の早い推論器と遅い推論器の時間差を埋める、アプリケーション側の工夫などが求められる。

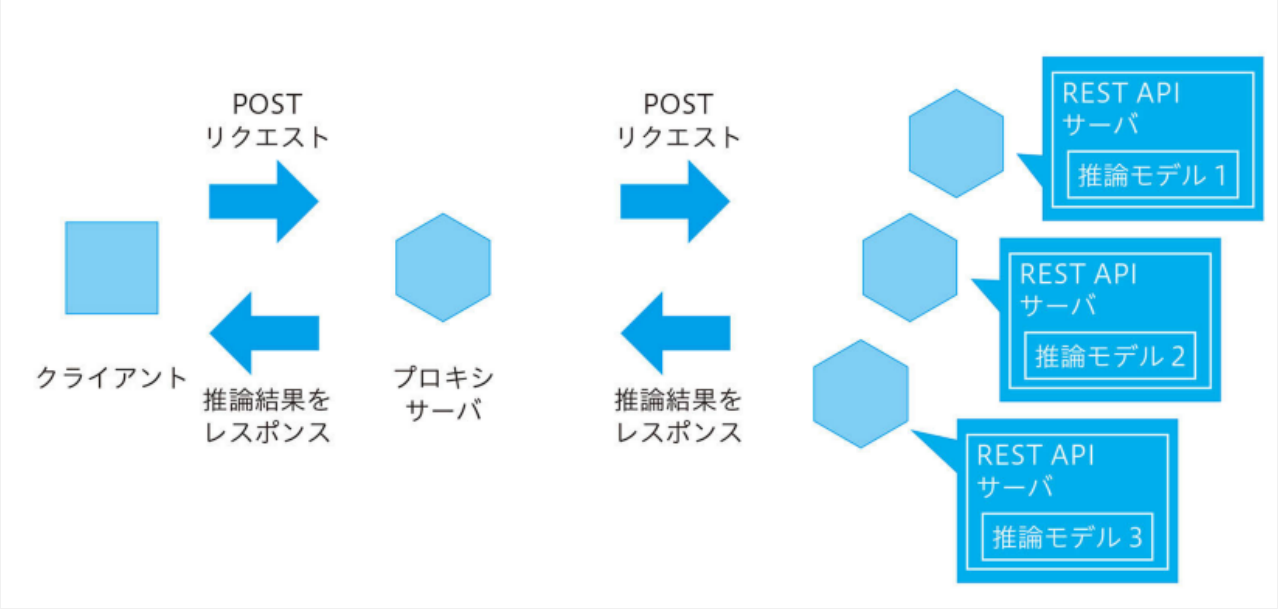
前処理・推論パターン



直列マイクロサービスパターン



並列マイクロパターン



- キャッシュパターン
  状況によっては同じデータを複数回繰り返し使うようなケースが存在する。同定できるデータであればキャッシュを用いることで推論負荷を避けることができる。
    - 推論キャッシュパターン 同一データのリクエストに対して同じ推論結果を返し、かつ同一のデータが複数回送信されてくる場合のシステム構成。
        - メリット
            - 素早いレスポンス
            - 推論器のコスト削減
        - デメリット
            - キャッシュサーバのコストの増加
    - データキャッシュパターン
      同一データの推論リクエストが発生し、かつそのデータを同定可能なときに、データそのものや前処理後のデータをキャッシュに残し、高速なデータ処理を実現するシステム構成。
        - メリット *
        - デメリット
- 開発パターン
    - 推論器テンプレートパターン
    - Edge AI パターン
- アンチパターン
    - オンラインビックサイズパターン
    - オールインワンパターン

リクエスト処理パターン	処理分割パターン	キャッシュパターン	開発パターン	アンチパターン
同期推論パターン	前処理・推論パターン	推論キャッシュパターン	推論器テンプレートパターン	オンラインビックサイズパターン
非同期推論パターン	直列マイクロパターン	データキャッシュパターン	Edge AI パターン	オールインワンパターン
バッチ処理パターン	並列マイクロパターン			
	時間差推論パターン			