もう並列実行は怖くない
コネクション枯渇解消のための実践的アプローチ

@katakyo

Kaigi on Rails 2025

★mybest

Partner with RubyStackNews

Independent Ruby & Rails publication for senior developers

Why RubyStackNews?

- Focused on Ruby and Ruby on Rails
- Long-form articles based on real conference talks
- Audience of senior developers and tech leads
- Readers from the US, Europe, and Asia

RubyStackNews turns conference talks and real-world experience into practical, production-focused technical articles.

Partnerships & Sponsorships

- Article sponsorships
- Inline placements inside articles
- Sidebar visibility

View partnership details

rubystacknews.com

Partnership info

01

自己紹介

自己紹介

片田 恭平（katakyo)

プロダクト開発部 バックエンドエンジニア

𝕏@katakyo_51

mybest

●自己紹介

23 卒でマイベストに新卒入社

現在は社内システムの AI ワークフローの開発、社内の AI 活用改善などして

います

kashiwa.rb によく出没します

●趣味

自作 PC/ ラーメン / サウナ

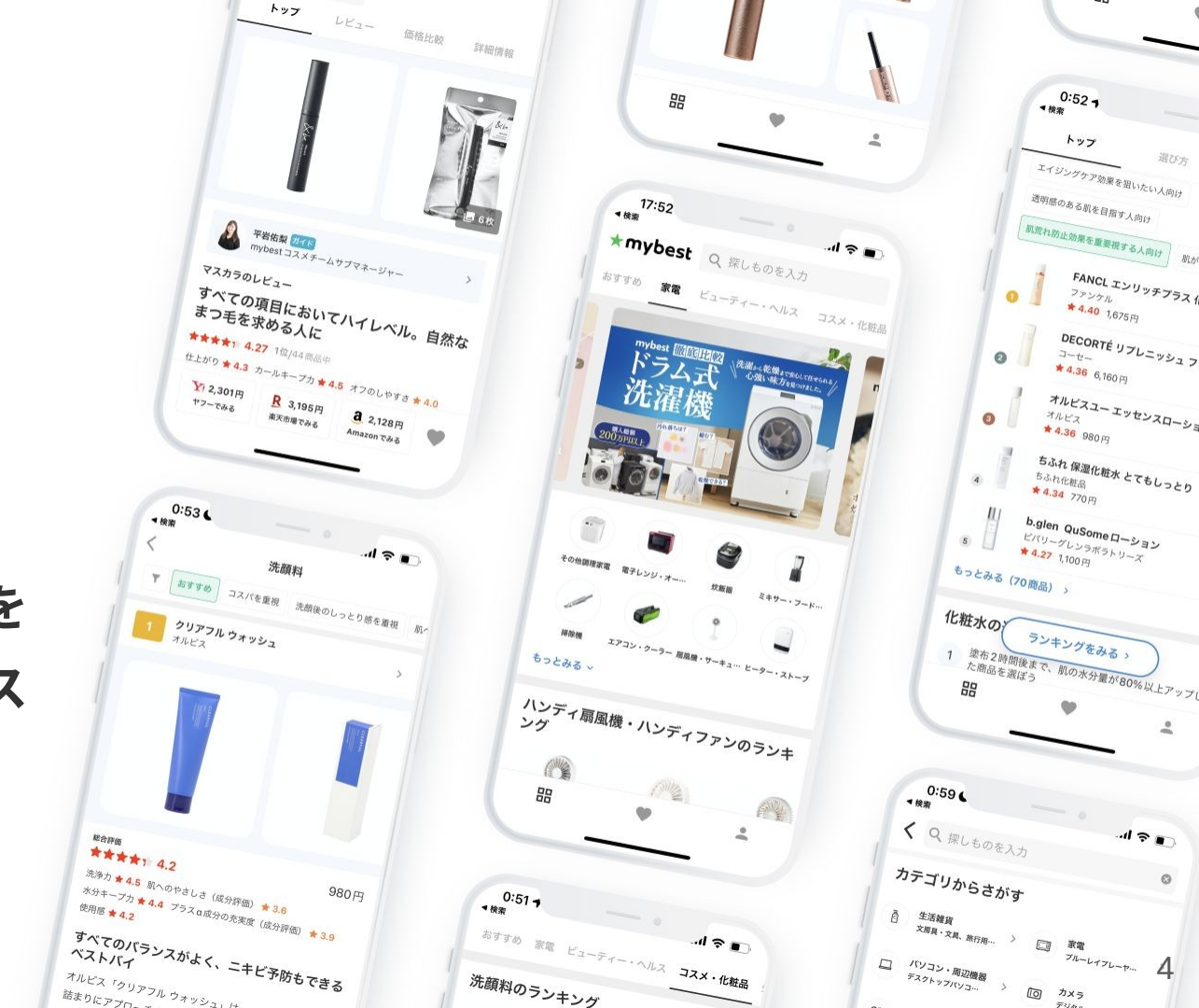
★mybest

ユーザーの“選択”をサポートするサービス

月間利用者数

3,000 万人以上

（2025年8月時点）



商品を自社で購入し、専門性を持ったメンバーが徹底検証

選択に資するデータベースを作るために、ユーザーニーズや利用シーンに合わせて 各領域で専門性を持ったメンバーが徹底検証を行い、唯一無二のデータベースを制作しています。

雨傘の検証。送風機を使って「耐風性」を比較

防水カメラの検証。プールを貸し切り、実際に潜って撮影

縦型洗濯機の検証。主要メーカー 7 社の中から人気 18 商品を購入して検証

ヘアアイロンの検証。全商品を実際に使用して違いを検証

チェーンソーの検証。片道 2 時間ほどかけて、山奥にこもって検証

電子レンジの検証。実際の使い勝手などをリアルに検証

今日持ち帰っていただけるもの

- Active Record のコネクションプールの仕組みと、それがなぜ枯渇するのかについて

- RDS の max_connections の数 , Active Record のコネクションプールのサイズ , Sideki
  レッド数 といった複数プロセスと複数スレッドの並列処理のパラメータを見積もるための
  方や手順
- コネクションの枯渇を未然に防ぐための方法

想定する聞き手

- バッチの並列処理の設計に困っている方
- Active Record のコネクションや並列実行の適切なパラメータ設定の考え方を知りたい方
- これから並列実行を行おうとしている方

想定環境

バックエンド　Ruby on Rails(8.0)

バッチ処理　Rake タスク

非同期 Job　Sidekiq

インフラ　AWS ECS, RDS

使用した gem Parallel

アジェンダ

1. 自己紹介
2. 今回並列実行を行った背景
3. 並列実行時に起きたエラー
4. 安全に並列処理を行うための方法
5. まとめ

02

今回並列実行を行った背景

マイベストでの事業課題

マイベストは、オールジャンルで商品 DB を
作成しています

扱う商品数は約 1000 万点で
今後も増え続けていきます

1 つの商品に複数のスペック情報を登録していきます

重量	164g	開閉方法	手動開閉
晴雨兼用	✓	PU加工	✓
収納時の長さ	15.5cm	UVカット率	99.99%以上
遮光率	99.99%以上	遮熱率	58.9%

マイベストでの事業課題

商品データやスペック情報の入力・チェック業務をほぼ全て手動で行っていた

ネットリサーチ
データ入力

ダブルチェック

時間と労力がとてもかかる .....

そこでゲームチェンジャー
として現れたのが

AI

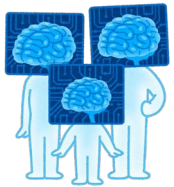
マイベストでの事業課題

AI によって商品情報の入力ができないかというプロジェクトが始動

AsIs

人によるリサーチとデータ入力

ToBe



AI によるリサーチとデータ入力

人間のチェック

AI ワークフロー

OpenAI や Gemini といった LLM の API を複数回実行し、仮説検証を進めてい

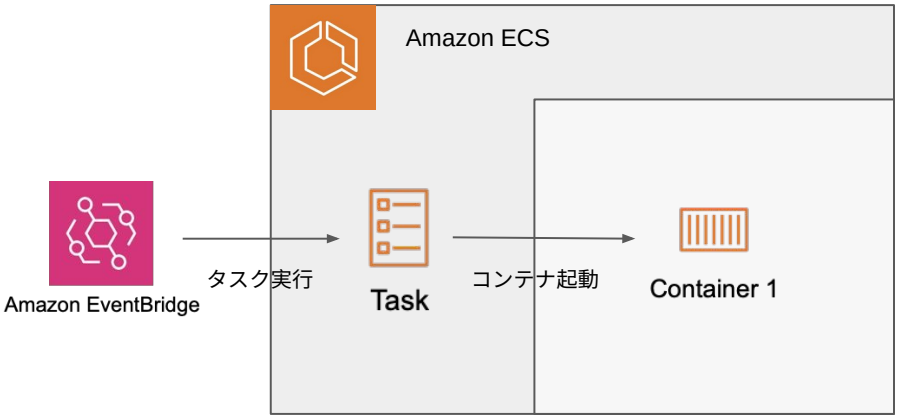
OpenAI や Gemini などの LLM を使って自動

ある程度ワークフローの
仮説検証もできたので
プロダクトに組み込もうとなりました

AI ワークフローのシステム概要

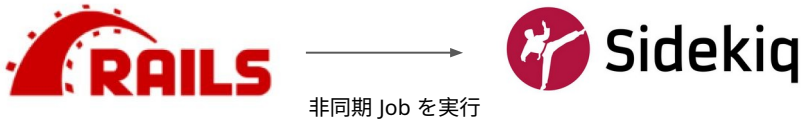
1 日次バッチで AI ワークフローの実行

未入力の商品データを埋めるために
Rake タスクを ECS タスクとして定期実行



2 管理画面から AI ワークフローを実行

管理画面から特定のボタンを押したときに
Sidekiq で非同期 Job を実行

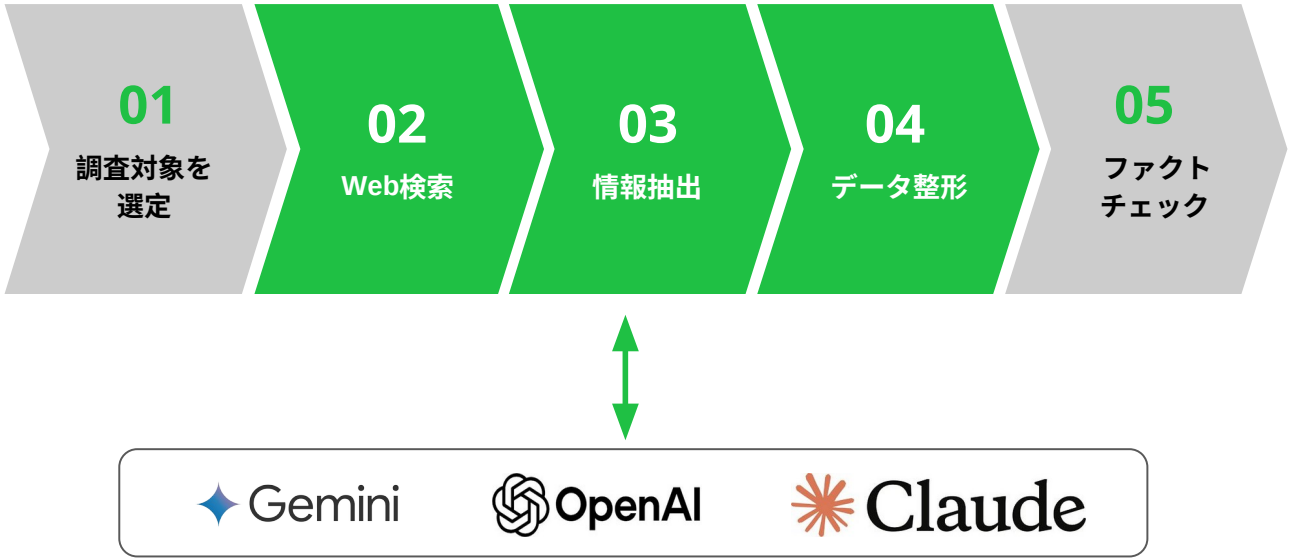


ただここで問題が ....

AI のワークフローは時間がかかる

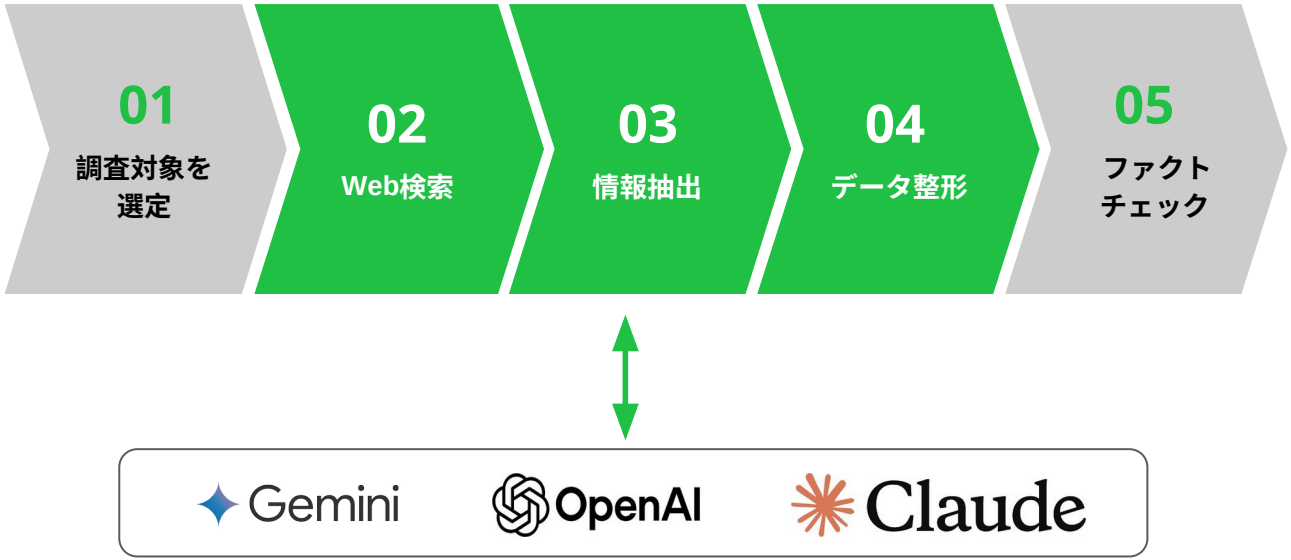
AI ワークフローの課題

精度改善のため、 AI のステップを複数に増やしたため 1 商品あたりのリサーチ
約 2 分近くかかるようになってしまった



AI ワークフローの課題

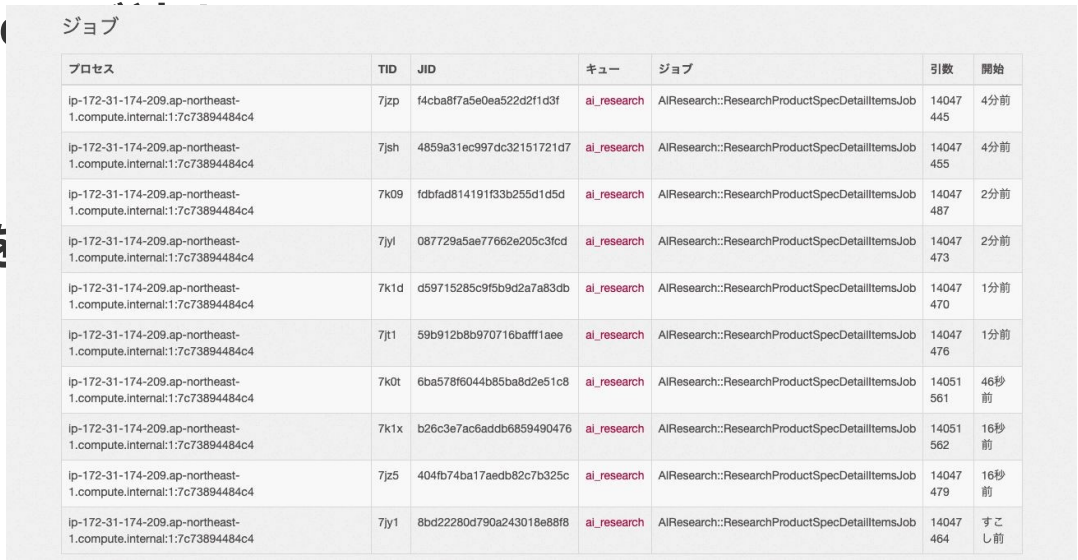
リサーチ対象商品が月に 120 万商品ほどあり、 1 日 3500 商品ほど捌けていた
1 ヶ月でも周り切らない



非同期 Job での AI ワークフローの課題

スレッド数が少ないため、 J
しまった

当時の運用だと、 Sidekiq を
行っていた



ジョブ

プロセス	TID	JID	キュー	ジョブ	引数	開始
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7jzp	f4cba8f7a5e0ea522d2f1d3f	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14047445	4分前
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7jsh	4859a31ec997dc32151721d7	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14047455	4分前
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7k09	fdbfad814191f33b255d1d5d	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14047487	2分前
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7jyl	087729a5ae77662e205c3fcd	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14047473	2分前
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7k1d	d59715285c9f5b9d2a7a83db	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14047470	1分前
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7jt1	59b912b8b970716bafff1aee	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14047476	1分前
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7k0t	6ba578f6044b85ba8d2e51c8	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14051561	46秒前
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7k1x	b26c3e7ac6addb6859490476	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14051562	16秒前
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7jz5	404fb74ba17aedb82c7b325c	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14047479	16秒前
ip-172-31-174-209.ap-northeast-1.compute.internal:1:7c73894484c4	7jy1	8bd22280d790a243018e88f8	ai_research	AIResearch::ResearchProductSpecDetailItemsJob	14047464	すこし前

外部 API の I/O がボトルネックだから、パフォーマンスチューニングしづらい …

並列実行すれば、
なんとかなるのでは?

本当に大丈夫?

03

並列実行時に起きたエラー

Parallel gem について

Parallel gem を使うことで、 Ruby で並列化を手軽に実装することができる

3 種類の並列化 (マルチプロセス、マルチスレッド、 Ractor) を引数で切り替えること可能

```ruby
def exec(product_ids, threads: 8)
  Product.where(id: product_ids).find_in_batches do |batch|
    Parallel.each(batch.to_a, in_threads: threads) do |product|
      research_spec_detail_items(product)
    end
  end
end
```

AI ワークフローのマルチスレッドを実装

今回の AI ワークフローは I/O バウンドがボトルネックなので、 1 プロセスの ECS タ
8 マルチスレッドで動かし高速化を目指した

```ruby
def exec(product_ids, threads: 8)
  Product.where(id: product_ids).find_in_batches do |batch|
    Parallel.each(batch.to_a, in_threads: threads) do |product|
      research_spec_detail_items(product)
    end
  end
end
```

バッチを実行してみる

コネクションプールが枯渇していることがあった

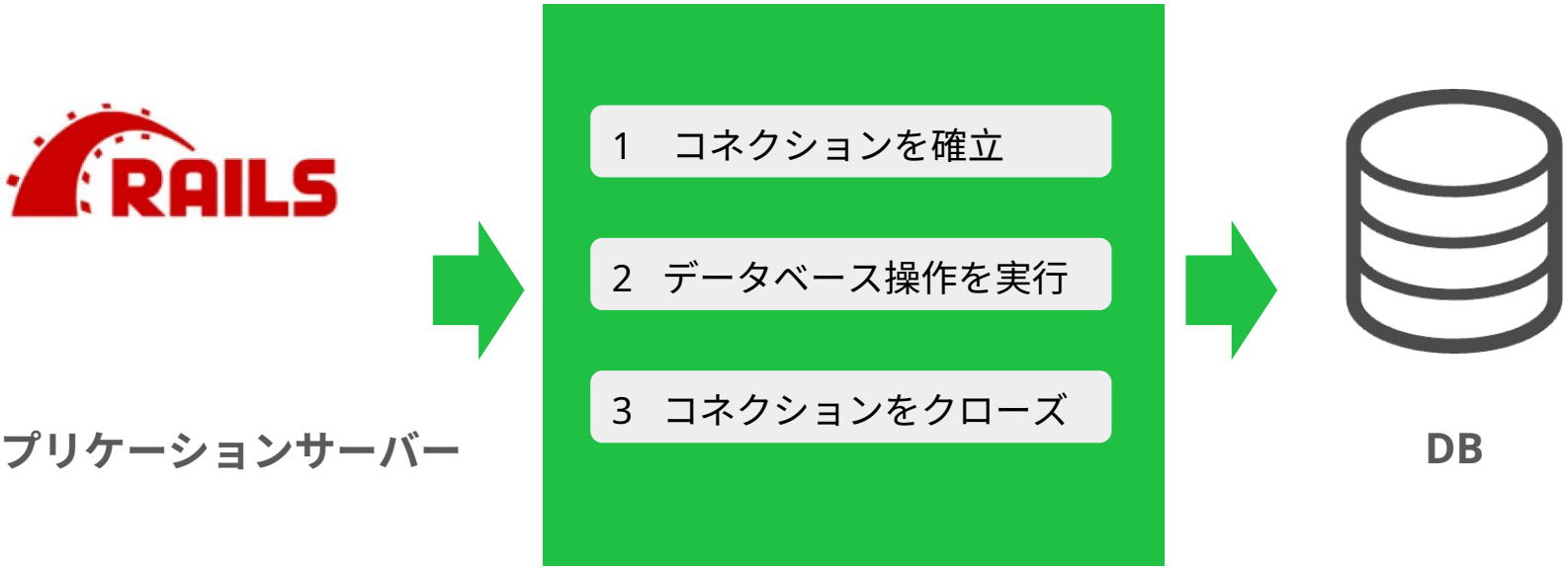
Bugsnag アプリ
New handled error in production from my-best.com
Handled error
ActiveRecord::ConnectionTimeoutError: could not obtain a connection from the pool within 5.000 seconds (waited 5.003 seconds); all pooled connections were in use

Bugsnag アプリ
New handled error in production from my-best.com
Handled error
ActiveRecord::ConnectionTimeoutError: could not obtain a connection from the pool within 5.000 seconds (waited 5.003 seconds); all pooled connections were in use

Bugsnag アプリ
New handled error in production from my-best.com
Handled error
ActiveRecord::ConnectionTimeoutError: could not obtain a connection from the pool within 5.000 seconds (waited 5.003 seconds); all pooled connections were in use

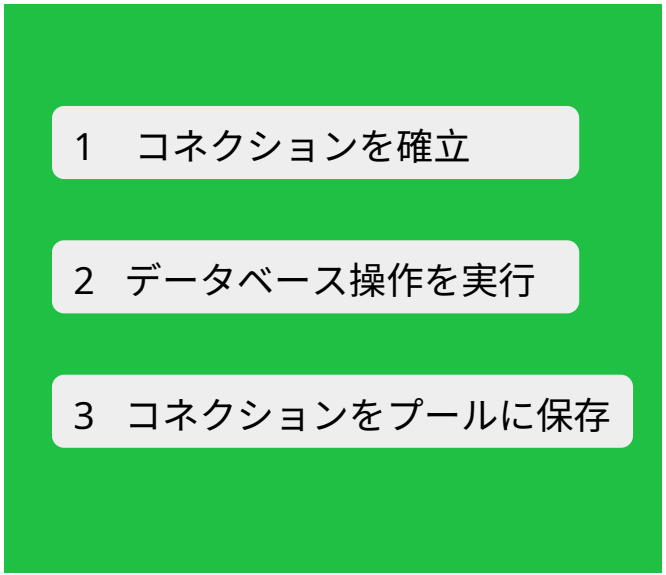
データベースのコネクションとは?

アプリケーションサーバーは、データベースとやりとりする必要が生じるとアプリケーションサーバーとデータベースサーバー間の専用通信チャネルである「コネクション」を確立します

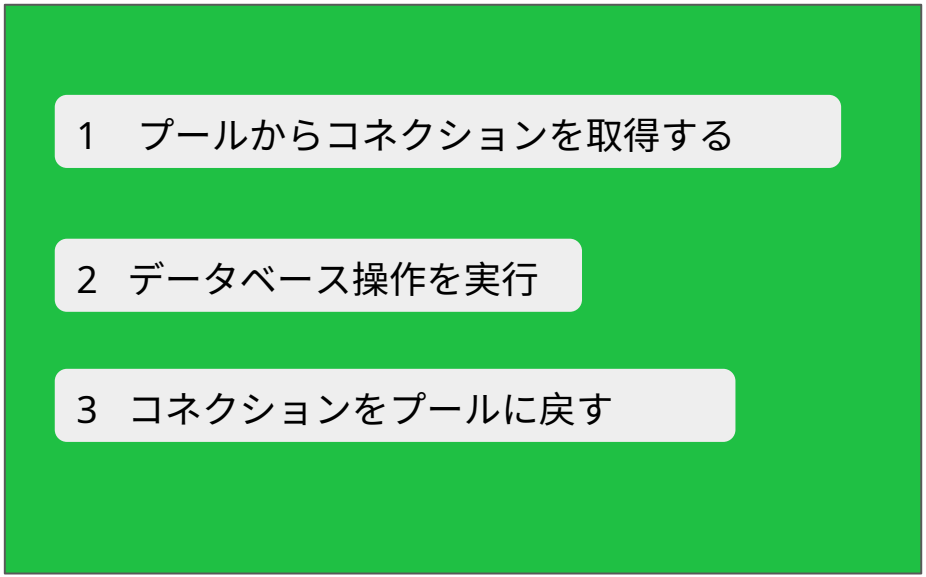


データベースのコネクションプールとは?

DB との接続・切断はコストが高い処理のため、 Active Record はあらかじめ一定数の接続を確保しておき、必要に応じて使い回す「コネクションプール」という仕組みを持っています。



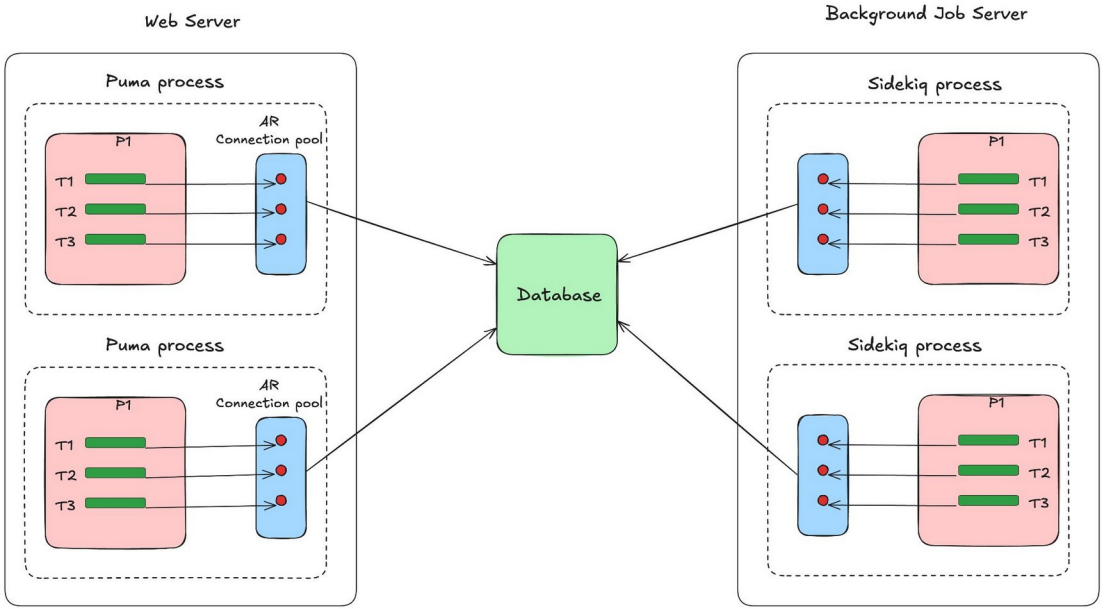
コネクション確立前



すでにコネクションが確立済み

ActiveRecord におけるコネクションプールについて

**Active Record は、データベースコネクションプールを Web プロセスやバックグラウンドプ
ごとに管理します**



https://www.bigbinary.com/blog/understanding-active-record-connection-pooling

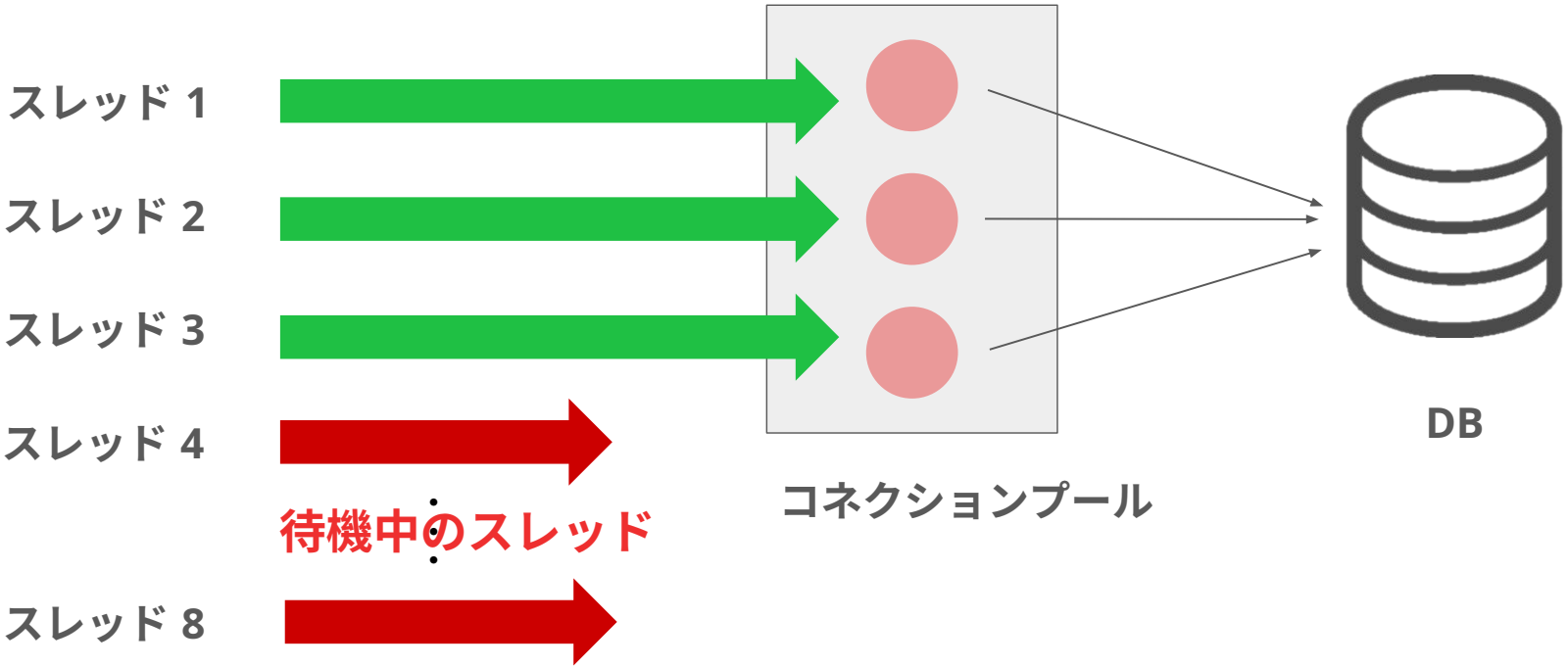
コネクションプールが枯渇した理由

Parallel ブロック内の関数で DB データの操作があったが、スレッド数に対してコネクションプールの数が適切に設定されていないことが原因でした

```ruby
def exec(product_ids, threads: 8)
  Product.where(id: product_ids).find_in_batches do |batch|
    Parallel.each(batch.to_a, in_threads: threads) do |product|
      research_spec_detail_items(product)
    end
  end
end
```

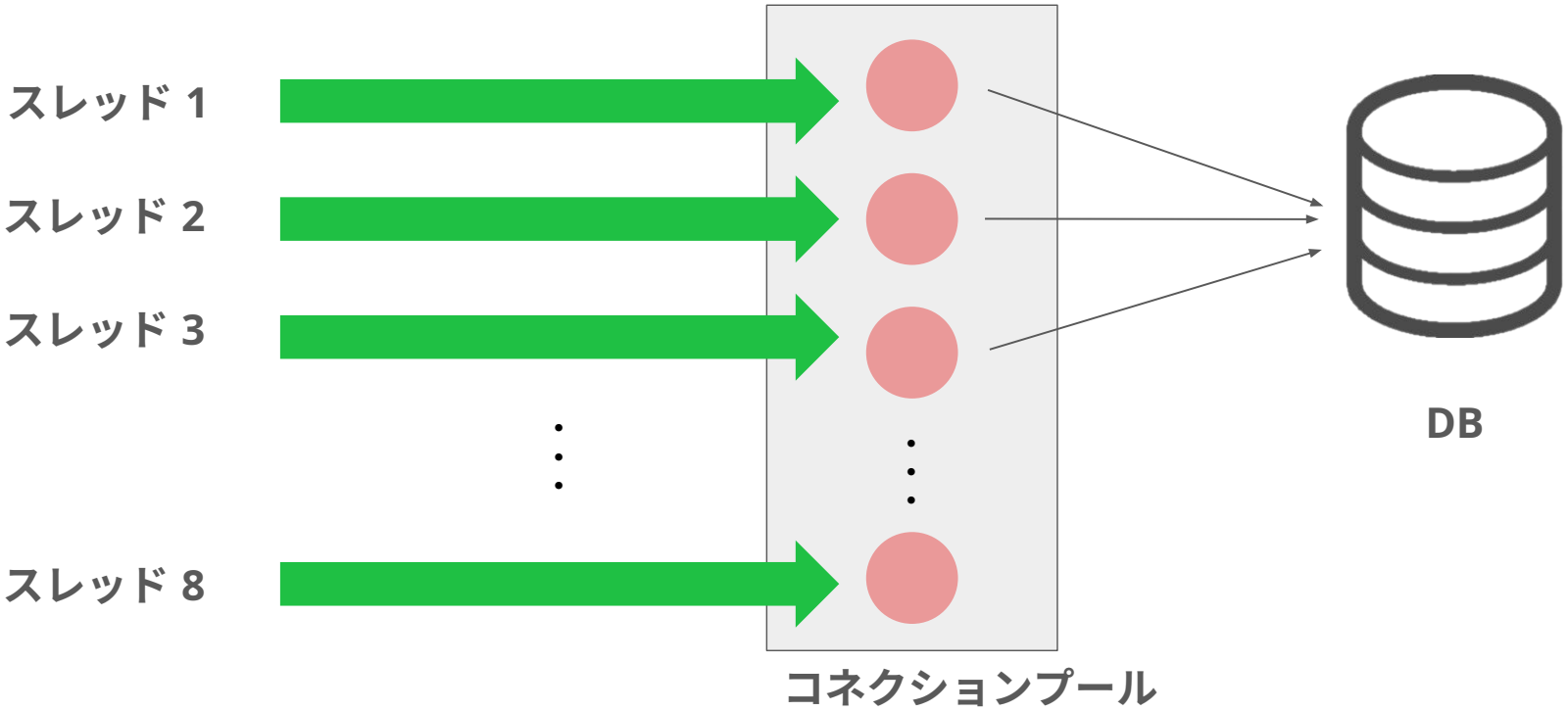
失敗したケース

スレッド数に対してコネクションプールのサイズが足りていないため、待機する秒数が checkout_timeout を超えたため発生



適切なケース

1 つのスレッドで複数のデータベースコネクションは発生しないので、
DB のやり取りが発生する場合は、コネクションプールのサイズをスレッドと同数に合わせる



データベースのプールサイズの設定方法

ActiveRecord では、 database.yml でプールで保持するコネクションの最大数を設定できます

Rails ではプロセスごと (Puma) のプール数を連動するために RAILS_MAX_THREADS というデフォルト値
が用意されているが、こちらを 8 スレッドの場合は 8 にする

```
pool: <%%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
```

設定が甘かった ....
Sidekiq の時はちゃんと設定しよう

Sidekiq の並列処理と　concurrency

concurrency　の設定値が、　1 プロセス内で同時に
スレッドの数を決定します

AI ワークフローを Sidekiq で動かす

ワークフローの共通化クラスにおいて Sidekiq のスレッドを使いたい

**先ほど Rake タスクでは 8 スレッドで設定した値を Sidekiq の Job ではスレッドのネ
避けるため、この値を 1 に設定して実装しました。**

```
module AIResearch
  class ProductAiResearchJob < ApplicationJob
    queue_as :ai_research

    def perform(product_id)
      products_researcher = AIResearch::ProductsResearcher.new
      products_researcher.exec([product_id], 1, threads_num:
1)  end
  end
end
```

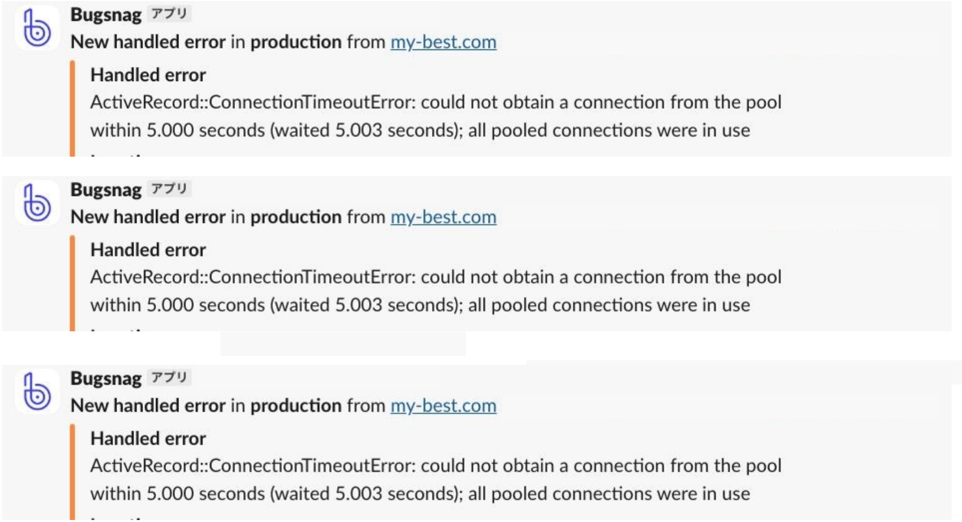
新しいプロセスで Sidekiq を立ち上げてみる

sidekiq-ai-research という名前の新しいプロセスを立ち上げて既存の Job に影響が出ないように試みた

試験的にプロセスを作成し、 Sidekiq のスレッド数を 10 でこのプロセスのコネクションプールのサイズを 10 と設定した

```
# config/sidekiq-ai-research.yml
:concurrency: 10
:queues:
  - hogehoge
  - fugafuga

:limits:
  fugafuga: 5
```

またコネクションプールが
枯渇してしまうケースが発生



Sidekiq でのコネクションプールが枯渇した原因

Sidekiq は起動時にプロセス自身がコネクションを 1 つ確保
ます。そのため、実際にスレッドが使えるコネクションは設
値より 1 つ少なくなります。

結果として、 sidekiq.yml の concurrency と database.y
pool の数を同じにすると、コネクションが 1 つ不足する事
陥ります。

```
# config/sidekiq-ai-research.yml
:concurrency: 10
:queues:
  - hogehoge
  - fugafuga

:limits:
  fugafuga: 5
```

並列処理を行うためのコネクションプールのサイズ

Batch の時

コネクションプールのサイズ　= Batch 内でのスレッド数

Sidekiq の時

コネクションプールのサイズ　= Sidekiq のスレッド数　+1

05

安全に並列処理を行うための方法

安全に並列処理を行うための方法

1. パラメータ設定の考え方
2. コネクションプールを枯渇させないようにする工夫
3. パフォーマンス改善

パラメータ設定の考え方

DB と可能性のあるサービスを考えて悲観的に見積もる

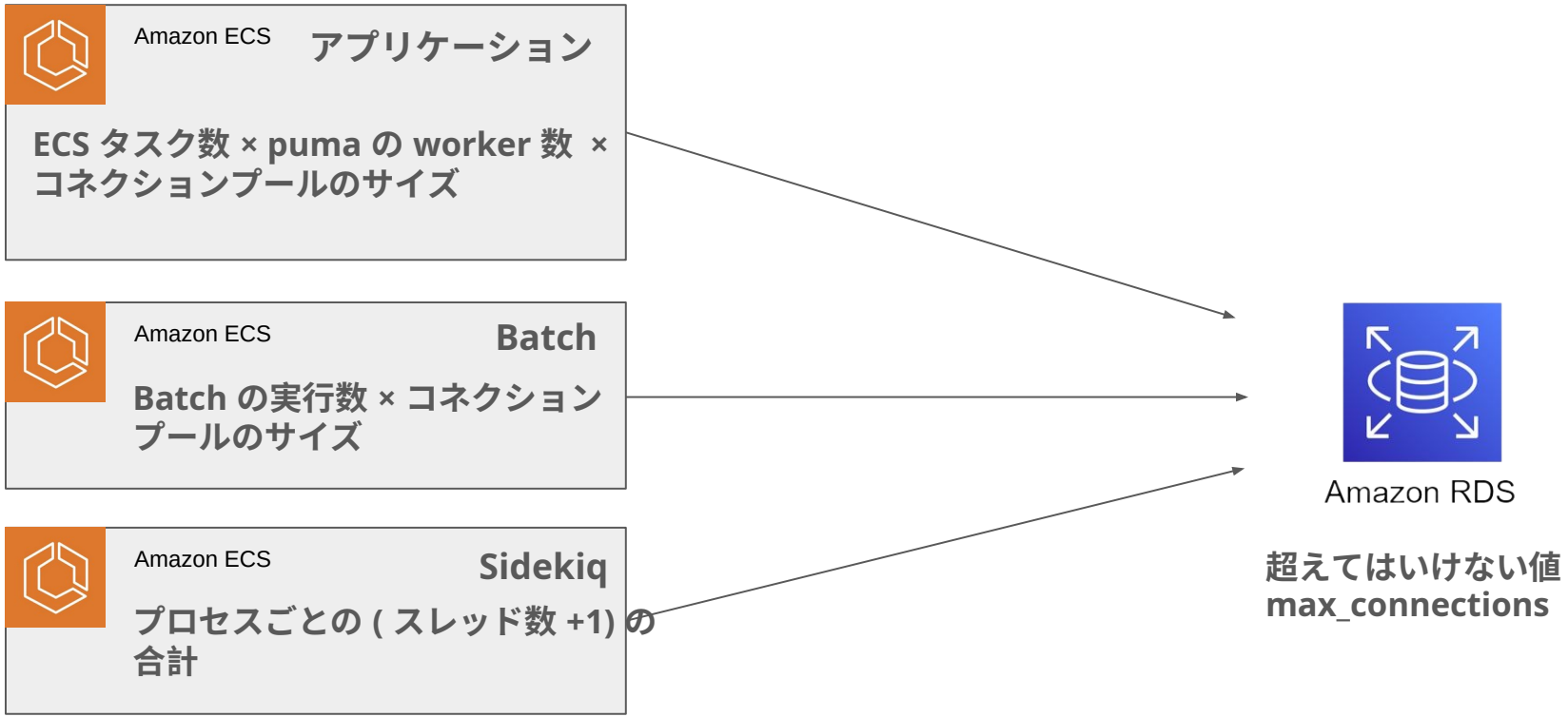
max_connections の数を超えると Too Many Connections エラーによりアプリケーションのダウンタイムが発生する

DB の最大コネクション数は以下のように見積もる

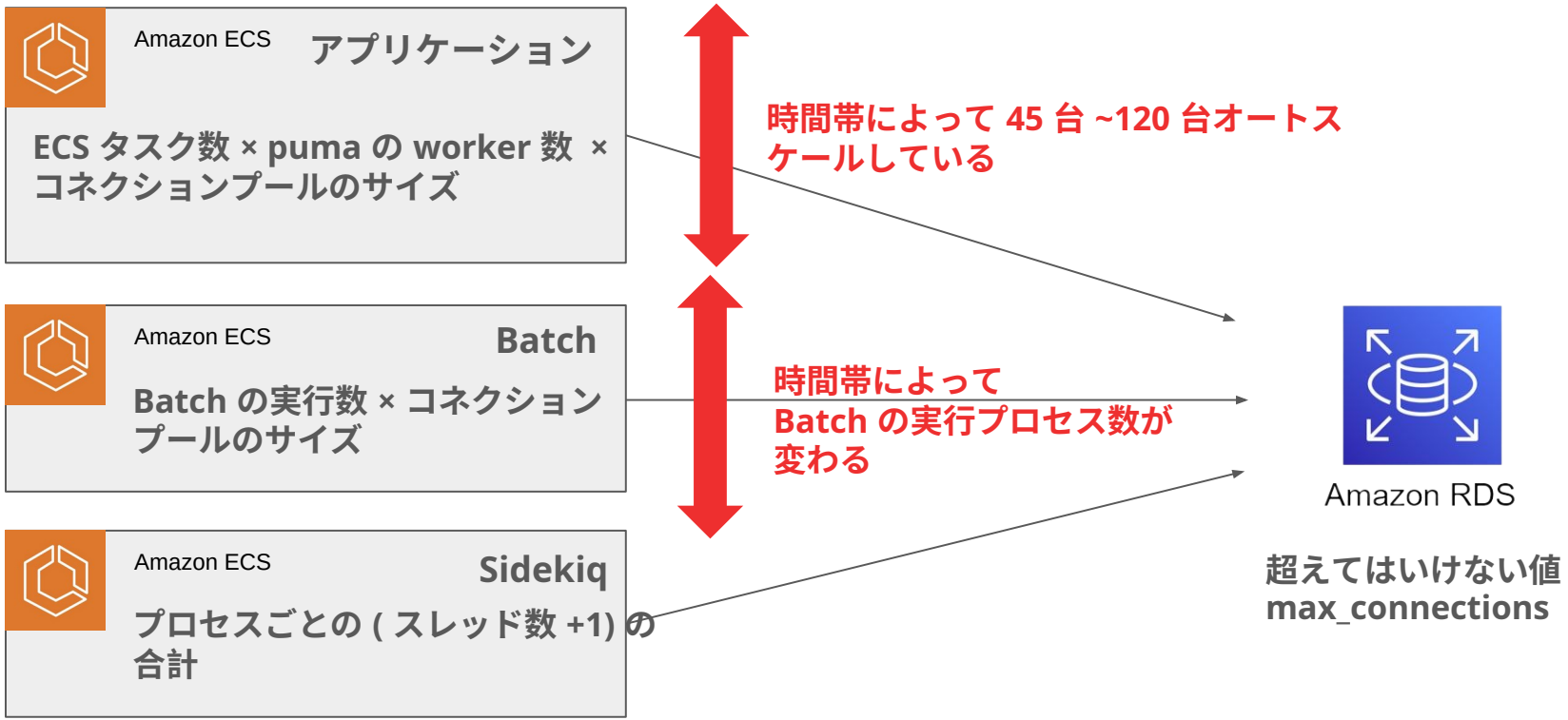
- Web コネクション数 = ECS タスク数 × Puma のワーカー数 × コネクションプールのサイズ
- バッチのコネクション数 = ECS タスク数 × コネクションプールのサイズ
- バックグラウンドコネクション数 = ECS タスク数 × Sidekiq プロセス数 × プロセスあたりのスレッド数

- DB の最大コネクション数 =
  Web コネクション数 + バッチのコネクション数 + バックグラウンドコネクション数

インフラ全体でのパラメータ見積もり



インフラ全体でのパラメータ見積もり



max_connections の値を確認する

使用している RDS のスペックごとに DB コネクションが可能な max_connections 数が異なる

db.r5.4xlarge (Writer) を使っていたため 4000 という値が max_connectios になります

Aurora Serverless v2 インスタンスによるこのパラメータの処理方法については、「Aurora Serverless v2 の最大接続数」を参照してください。

インスタンスクラス	max_connections のデフォルト値
db.t2.small	45
db.t2.medium	90
db.t3.small	45
db.t3.medium	90
db.t3.large	135
db.t4g.medium	90
db.t4g.large	135
db.r3.large	1,000
db.r3.xlarge	2000
db.r3.2xlarge	3000
db.r3.4xlarge	4000

https://docs.aws.amazon.com/ja_jp/AmazonRDS/latest/AuroraUserGuide/AuroraMySQL.Managing.Performance.html

全体のコネクション数は RDS の max_connections の半分にする

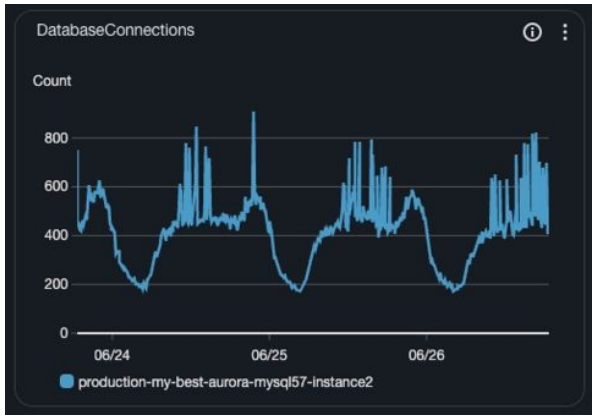
- デプロイ時には、瞬間的に古いサービスと新しいサービスが一時的に共存するため、ECS のタスク量が 2 倍になる可能性がある

- DB に接続している、全体のコネクション数の合計が、 RDS の max_connections の半分を超えないように設定します。

データベースのコネクションを確認

サービスによって ECS タスクのサーバー台数が変動するため、ピーク時のデータベースコネクションを計測

追加可能なデータベースコネクション数
= (max_connections / 2) - ピーク時のデータベースコネクション数



ワークフローの I/O の比率を測定

アプリケーションの I/O 待ちの割合を各ワークフローの測定により 90% 以上あると算出しました。

$$Speedup = \frac{1}{(1-p) + \frac{p}{N}}$$

p: 並列化できる処理の割合
N: 並列実行に利用するプロセッサ（スレッド）の数
(1 - p): 逐次実行（並列化できない）が必要な処理の割合

アムダールの法則などから、現状のスレッド数を増やしても性能向上が鈍化する点を見極め、
CPU リソース効率が最も良い 8 スレッドを 1 プロセスあたりの最大値として採用し、それ以上実行
たい場合はサーバー台数を増やして並行処理を行った

Sidekiq の公式推奨の concurrency の上限

Sidekiq の 1 プロセスでの concurrency は 50 以下で推奨されています
50 を超える場合はプロセスを増やす

Concurrency

You can tune the amount of concurrency in your Sidekiq process. By default, one sidekiq process creates **5 threads**. If that's crushing your machine with I/O, you can adjust it down:

```
bundle exec sidekiq -c 4
RAILS_MAX_THREADS=3 bundle exec sidekiq
```

I don't recommend setting concurrency higher than 50. Starting in Rails 5, `RAILS_MAX_THREADS` can be used to configure Rails and Sidekiq concurrency. Note that ActiveRecord has a connection pool which needs to be properly configured in `config/database.yml` to work well with heavy concurrency. Set `pool` equal to the number of threads:

```
production:
  adapter: mysql2
  database: foo_production
  pool: <%= ENV['RAILS_MAX_THREADS'] || 10 %>
```

https://github.com/sidekiq/sidekiq/wiki/Advanced-Options

正しくコネクションプールが設定されているかログを仕込む

ActiveRecord::Base.connection_pool.stat などでコネクションプールのサイズや、現在のコネクション数などを可視化してデプロイ時に設定が意図通りか確認

```
def exec(product_ids, threads: 8)
  Product.where(id: product_ids).find_in_batches do |batch|
    Rails.logger.info('===== display connection pool stat =====')
    Rails.logger.info("===== #{ActiveRecord::Base.connection_pool.stat.inspect} =====")
    Parallel.each(batch.to_a, in_threads: threads) do |product|
      research_spec_detail_items(product)
    end
  end
end
```

```
===== {:size=>8, :connections=>3, :busy=>3, :dead=>0, :idle=>0, :waiting=>0, :checkout_timeout=>5.0} =====
```

コネクションプールを
枯渇させないようにする工夫

アプリケーションのコードでもコネクションを意識

バッチの中で DB のコネクションが発生する箇所と外部 API との通信を意識

```ruby
def process(product)
  attrs = ActiveRecord::Base.connection_pool.with_connection do
    { id: product.id, name: product.name }
  end

  spec_text = @llm.fetch_specs_from_name(attrs[:name])
  specs     = parse_specs(spec_text)

  ActiveRecord::Base.connection_pool.with_connection do
    product = Product.find(attrs[:id])
    transaction = Transactions::UpdateProductSpecJson.new(product)
    transaction.exec(specs)
  end
end
```

DB の読み取り

外部 API の通信

DB の書き込み

```ruby
class ProductSpecDetailItemsResearcher
  def initialize(llm_client:)
    @llm = llm_client
  end

  def exec(product_ids, threads: 8)
    Product.where(id: product_ids).find_in_batches do |batch|
      Parallel.each(batch.to_a, in_threads: threads) do |product|
        process(product)
      rescue => e
        Rails.logger.error(e)
      end
    end
  end

  private

  def process(product)
    attrs = ActiveRecord::Base.connection_pool.with_connection do
      { id: product.id, name: product.name }
    end

    spec_text = @llm.fetch_specs_from_name(attrs[:name])
    specs     = parse_specs(spec_text)

    ActiveRecord::Base.connection_pool.with_connection do
      product = Product.find(attrs[:id])
      transaction = Transactions::UpdateProductSpecJson.new(product)
      transaction.exec(specs)
    end
  end

  def parse_specs(text)
    JSON.parse(text)
  rescue
    {}
  end
end
```

ActiveRecord::Base.connection_pool.with_connection とは

DB コネクションプールから接続を「一時的に借りて、ブロック終了時に必ず返却」するための安全なラッパー

メリット

スレッドや並列処理下でも接続リークを防ぎ、ConnectionTimeout の発生確率を下げる。

デメリット

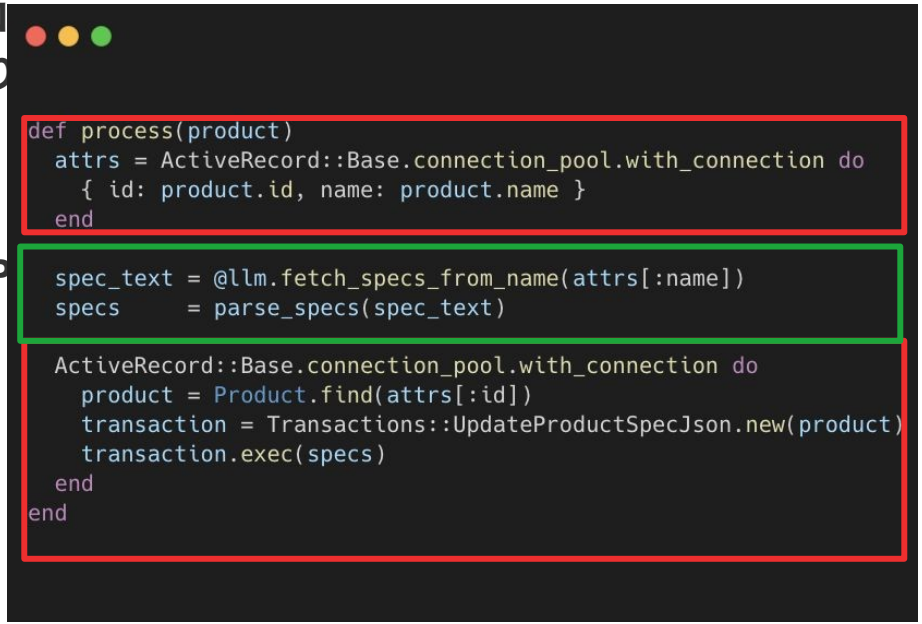
コネクションプールの checkout/checkin の頻発でプットが下がることがある

```ruby
def process(product)
  attrs = ActiveRecord::Base.connection_pool.with_connection do
    { id: product.id, name: product.name }
  end

  spec_text = @llm.fetch_specs_from_name(attrs[:name])
  specs     = parse_specs(spec_text)

  ActiveRecord::Base.connection_pool.with_connection do
    product = Product.find(attrs[:id])
    transaction = Transactions::UpdateProductSpecJson.new(product)
    transaction.exec(specs)
  end
end
```

ActiveRecord::Base.connection_pool.with_connection とは

外部 API の通信時に with_connection ブロックを閉じてコネクションを開放することにより、コネクションを占有し続けないようにする

コネクションが発生するブロックと、外部 API の通信などの I/O を明確に分ける工夫をする



パフォーマンス改善

長時間のトランザクションに気をつける

Web アプリケーションでは、 データベースへのコネクションが多数 発生するため
このような状況で、データの整合性を保つためにトランザクションは不可欠

トランザクションを長時間張るとテーブルロックの可能性が発生し、
ActiveRecord::LockWaitTimeout エラーやコネクションを占有し続けてパフォーマンスが悪化する原因となります。

トランザクションに気をつける

- **範囲を最小限に**

  本当に一貫性が求められる、必要最低限の DB 操作のみをトランザクションで囲みます。

- **重い処理は外に出す**

  外部 API の呼び出しや、時間のかかる計算などはトランザクションの外で実行させる

- **ロックの粒度を意識する**

  不必要にテーブル全体をロックしないよう、更新範囲を限定するなどの工夫をする。

パフォーマンスを意識するために

ワークフロー全体の処理を rack-lineprof の gem を使った調査を開発にテストとして行った

例として、各ワークフローのトランザクションを貼っているメソッドやクラスを行単位でプロ
ングするようにした

```
local-app  |                |   1  def process(product)
local-app  |   0.9ms     1  |   2    attrs = ActiveRecord::Base.connection_pool.with_connection do
local-app  |   0.5ms     1  |   3      { id: product.id, name: product.name }
local-app  |                |   4    end
local-app  |  85.3ms     1  |   6    spec_text = @llm.fetch_specs_from_name(attrs[:name])
local-app  |   1.1ms     1  |   7    specs     = parse_specs(spec_text)
local-app  |   2.6ms     1  |   9    ActiveRecord::Base.connection_pool.with_connection do
local-app  |   1.7ms     1  |  10      product = Product.find(attrs[:id])
local-app  |   8.9ms     1  |  11      transaction = Transactions::UpdateProductSpecJson.new(product)
local-app  |  45.4ms     1  |  12      transaction.exec(specs)
local-app  |                |  13    end
local-app  |                |  14  end
```

https://nishinatoshiharu.com/usage-rack-lineprof/

プロファイリングによって検知できた例

- N+1 や非効率なクエリ処理の検知

- トランザクション時に S3 への画像の upload の同期処理

プロファイリングによって検知できた例

- N+1 や非効率なクエリ処理の検知

  →パフォーマンスチューニング
- トランザクション時に S3 への画像の upload の同期処理

  →S3 への画像の upload 処理とレコードの保存処理を分離し、 S3 への画像の upload を非同期処理に修正

安全に並列処理を行うためやったこと

- 環境変数の設定の見直しと適切なパラメータ見積もり
- ワークフロー内での DB コネクションが発生するブロックと I/O の完全分
  よるコネクション管理
- トランザクション内などで不要な I/O や N+1 などが起きていないか、行単
  でプロファイリングして確認

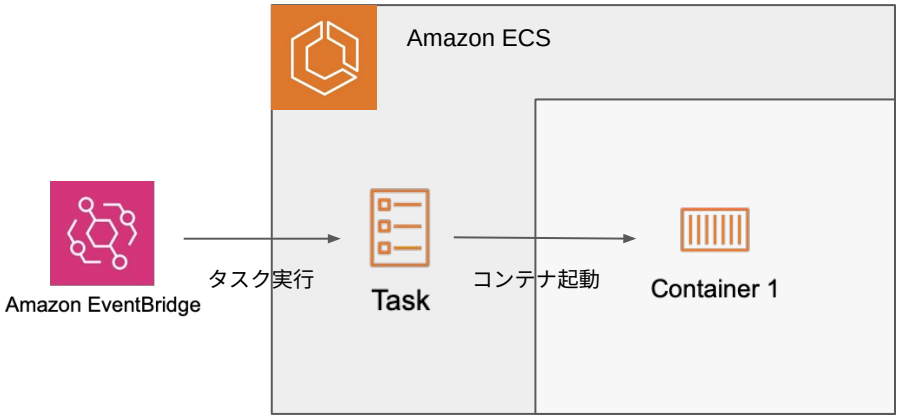
実際に追加したプロセス

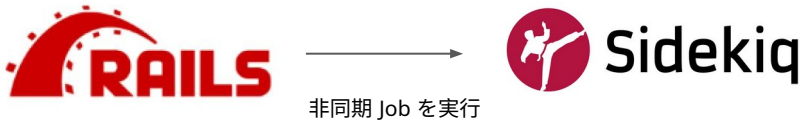
1 日次バッチで AI ワークフローの実行

2 管理画面から AI ワークフローを実行

ECS タスクを 15 台で 8 スレッドの並列処理を組み合わせた並行処理

別プロセスを立ち上げて非同期の 50 並列処理を実行





結果

- 120 並列で問題なく、バッチを完了
!5 台のサーバーで 8 スレッドの処理で
リサーチ件数が 3500 商品 / 日→ 45 万商品 / 日に!

- Sidekiq も 50 並列で他の Job に影響を与えることなく、高速
実現

06

まとめ

まとめ

- 並列実行するときはコネクションプールを意識してパラメータを設定しましょう
- コネクション枯渇を防ぐために、 DB コネクションが発生する処理と I/O の処理を分けてコネクションプールに余裕が出るようにしましょう
- LLM の API など I/O がボトルネックの時は並列化を検討してみましょう

After イベントを 3 社合同でやります!

https://connpass.com/event/370180/

LT 枠もぜひ募集していますのでよかったらご参加ください!

https://connpass.com/event/370180/

ご清聴ありがとうございました

参考文献

- https://www.bigbinary.com/blog/tuning-puma-max-threads-configuration-with-gvl-instrumentation
- https://techracho.bpsinc.jp/hachi8833/2025_07_01/151299
- https://www.bigbinary.com/blog/understanding-active-record-connection-pooling
- https://techracho.bpsinc.jp/hachi8833/2025_07_16/151486
- https://qiita.com/HrsUed/items/6a103322bf4e67e9054c
- https://nishinatoshiharu.com/usage-rack-lineprof/
- https://speakerdeck.com/andpad/yasasiiactiverecordnodbjie-sok-nosikumi