生成AIコスト削減のための工夫

2026.3.24

エムオーテックス株式会社　Cho Hyeonguk

※本スライドは個人の見解であり、所属する組織の意見ではありません。

自己紹介

- 名前：Cho Hyeonguk（チョウ　ヒョンウク）
- 所属：エムオーテックス株式会社　AI戦略G
- やっていること：生成AIエージェント開発・機械学習モデル開発
- 最近好きなもの：白ワイン

なぜこれを話そうと思ったか

生成AIを使い始めると、気づかないうちにコストが積み上がっていきます。最初は小さな金額でも、機能が増え、使う頻度が上がるにつれて、ある日突然「え、こんなにかかってたの？」となる。

そういう経験、ありませんか？

自分もそういう経験がありました。使いながらコストを意識するようになり、どうすれば無駄なく動かせるかをずっと考えてきました。トークンの減らし方、モデルの使い分け、キャッシュの活用、計測の仕方。今日はその中から、実践的なものを厳選してお伝えします。

なぜコストが問題になるのか

- 機能が増える・ユーザーが増える → コストが**比例以上**に膨らむ
- コストの正体は**トークン数**（入力＋出力）
- 入力・出力それぞれのトークン数×単価で課金される仕組み
- APIコールのたびに課金 → 油断するとすぐ積み上がる

つまり：**「無駄なトークンを減らし、賢くモデルを使う」**ことがすべてです

4つの考え方

この4つの軸で考えると、どこから手をつければいいかが見えてきます。全部一度にやる必要はありません。まず計測して、ボトルネックから攻めましょう。

アプローチ	そもそも何か	一言で言うと
Token Density	AIに送るトークン量を減らす考え方	送る前に削る
Architectural Routing	タスクに合ったモデルを選ぶ考え方	モデルを使い分ける
Stateful Efficiency	会話の状態・履歴を効率よく扱う考え方	状態を賢く管理する
Observability	コストを可視化して改善し続ける考え方	計測して改善し続ける

① Token Density ：送る前に削る

AIに渡す前に入力テキストを整理することが、コスト削減の第一歩です。
AIに意味が伝わればフォーマットは何でも構いません。トークン効率を優先しましょう。

Whitespace正規化

```
def normalize_whitespace(text: str) -> str:
  “””
  余分な空白・改行・インデントを除去します
  “””

   # 連続改行を最大2つに
   text = re.sub(r'¥n{3,}', '¥n¥n', text)
   # 連続スペース・タブを1つに
   text = re.sub(r'[ ¥t]+', ' ', text)
   return text.strip()
```

フォーマット変換

# JSON ：トークン数多	# YAML ：トークン数小

{
 "name": "田中",
 "age": 30,
 "role": "engineer"
}
	name: 田中
	age: 30
	role: engineer

② Token Density：Few-shot vs Zero-shot

Few-shotは精度を上げますが、その分トークンが増えます。どちらが経済的かはタスク次第です。

- 判断基準
  - *Zero-shotで十分な品質が出るなら、Few-shotは不要*
  - *Few-shotが必要な場合も、例示は最小限に*
  - *例示部分は**Prompt Caching**と組み合わせてコストを相殺*

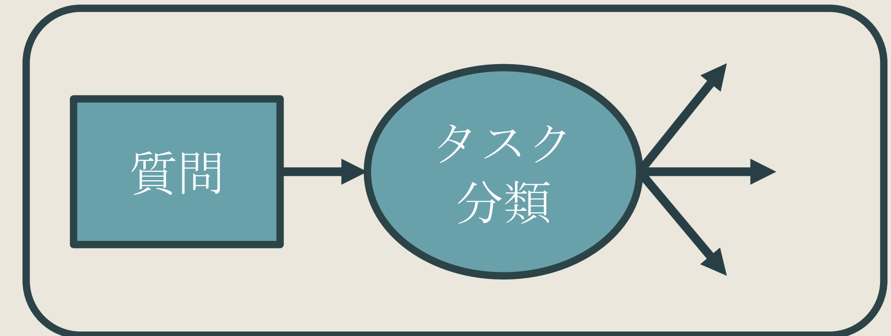
- ファインチューニングへの移行も検討できます
  - *同じFew-shot例示を数千〜数万回/月送り続けているなら移行の価値あり*
  - *ただし学習コスト・データ準備コストとのトレードオフを試算してから判断*

観点	Zero-shot	Few-shot
トークン数	少ない	例示数に比例して増加
精度	タスクによる	複雑なタスクで有効
推奨用途	単純な分類・抽出	出力形式の制御・複雑な推論

③ Architectural Routing：モデルを使い分ける（Intent-based Routing）

タクシーで近所のコンビニに行かないのと同じで、すべてのリクエストに高性能モデルを使う必要はありません。リクエストの意図を事前に分類し、タスクの複雑さに合ったモデルに振り分けるのが Intent-based Routing です。

タスク	推奨モデル
単純なFAQ・定型応答	SLM・軽量モデル
要約・分類・抽出	中規模モデル
複雑な推論・コード生成	高性能モデル



ルーティング判定自体も**軽量モデル or ルールベース**で行うことでオーバーヘッドを最小化できます。SLMは単純タスクであれば、大型モデルと遜色ない品質を大幅低コストで実現できます。

④ Architectural Routing：バッチ処理（Batch Inference）

リアルタイムの応答が不要なユースケースでは、バッチ処理が最もコスト効率の高い選択肢の一つです。

- 向いているユースケース
  - *大量ドキュメントの一括要約・分類*
  - *データセットへのラベリング*
  - *定期的なレポート生成・コンテンツ評価*

項目	オンデマンド	バッチ処理
レスポンス	リアルタイム	非同期（数分〜数時間）
コスト	標準単価	**最大50%削減**（各モデル・サービスで異なります）
向いているユースケース	チャット・対話	一括分類・要約・評価

⑤ Stateful Efficiency：Context Window制御

会話が長くなるほど、毎回送るトークン数は増え続けます。AIに何を見せて何を見せないか——その取捨選択がコンテキスト管理の本質であり、コスト直結の問題です。主に3つのアプローチがあります。

- **Token Threshold Management**：コンテキストが一定トークンを超えたら古い履歴を自動削除・圧縮
- **Semantic Summary（意味的要約）**：逐語保持をやめて、意味的に重要な情報だけを要約して保持
- **Selective Retention**：何を残すかを選ぶアプローチ。重要度でフィルタリングする方法と、直近のN件だけを保持する方法がある

⑤ Stateful Efficiency：Context Window制御

すべての履歴が常に必要なわけではありません。何を渡すかを意識するだけで、コンテキストの質とコストは大きく変わります。

原本

田中：このPCの見積を依頼したい。予算は20万円程度考えているんだよね。

AI：わかりました。見積結果は以下のURLからダウンロードできます。

田中：ありがとう！

制御後

Semantic Summary → 要約して保持
[要約] 担当者：田中、予算：20万円、PC見積依頼

Selective Retention → 重要度で選ぶ or 直近N件だけ残す
[保持] 「田中：このPCの見積を依頼したい。予算は20万円程度考えているんだよね。AI：見積結果は以下のURLからダウンロードできます。」
[削除] 「わかりました」「ありがとう」

⑥ Stateful Efficiency：Caching

キャッシュには2種類あります。モデル層で入力コストを削減する Prompt Caching と、アプリケーション層で出力結果を再利用する Response Caching です。

- **Response Cachingの応用**
  - *Semantic Cachingと組み合わせると意味的に近い入力にも適用可能*
  - *ただし類似度計算のオーバーヘッドとのトレードオフに注意*
- 使い方のヒント
  - *「営業時間は？」→「9時~18時です」や「ありがとう！」→「どういたしまして！」のような定型応答は、毎回生成する必要はありません。よく使う定型応答はそもそもAIで生成しないという選択肢もあります。Response Cachingはその中間地点として有効です。*

	Prompt Caching	Response Caching
キャッシュ対象	入力プロンプト	出力結果
実装層	モデル層	アプリケーション層
効果	トークン単価の削減（最大90%）	APIコスト自体を**ゼロ**に
向いている場面	共通プロンプトが多い	同一入力が繰り返される

⑦ Stateful Efficiency：出力とToolのスリム化

出力トークン・Tool定義・Tool設計——どれも意外と見落とされがちなコスト発生源です。

- **Output Token制限：**
  - *用途に応じた上限を設定し、不必要に長い出力を防ぎます。用途別に設定するのが基本です。*
  - *「箇条書きで3点にまとめて」のようにプロンプトで出力形式を明示的に指定するのも有効です。出力トークンは入力より単価が高いモデルも多いため、効果が大きい施策です。*

タスクの種類	max_tokens
ラベルだけ返す	10
要約	200
本文生成	1000

※このmax_tokensに加え、各ユーザーが使える累積max_tokensに制限がある場合は、利用限度に近くなったら事前に通知するとUXが良くなります

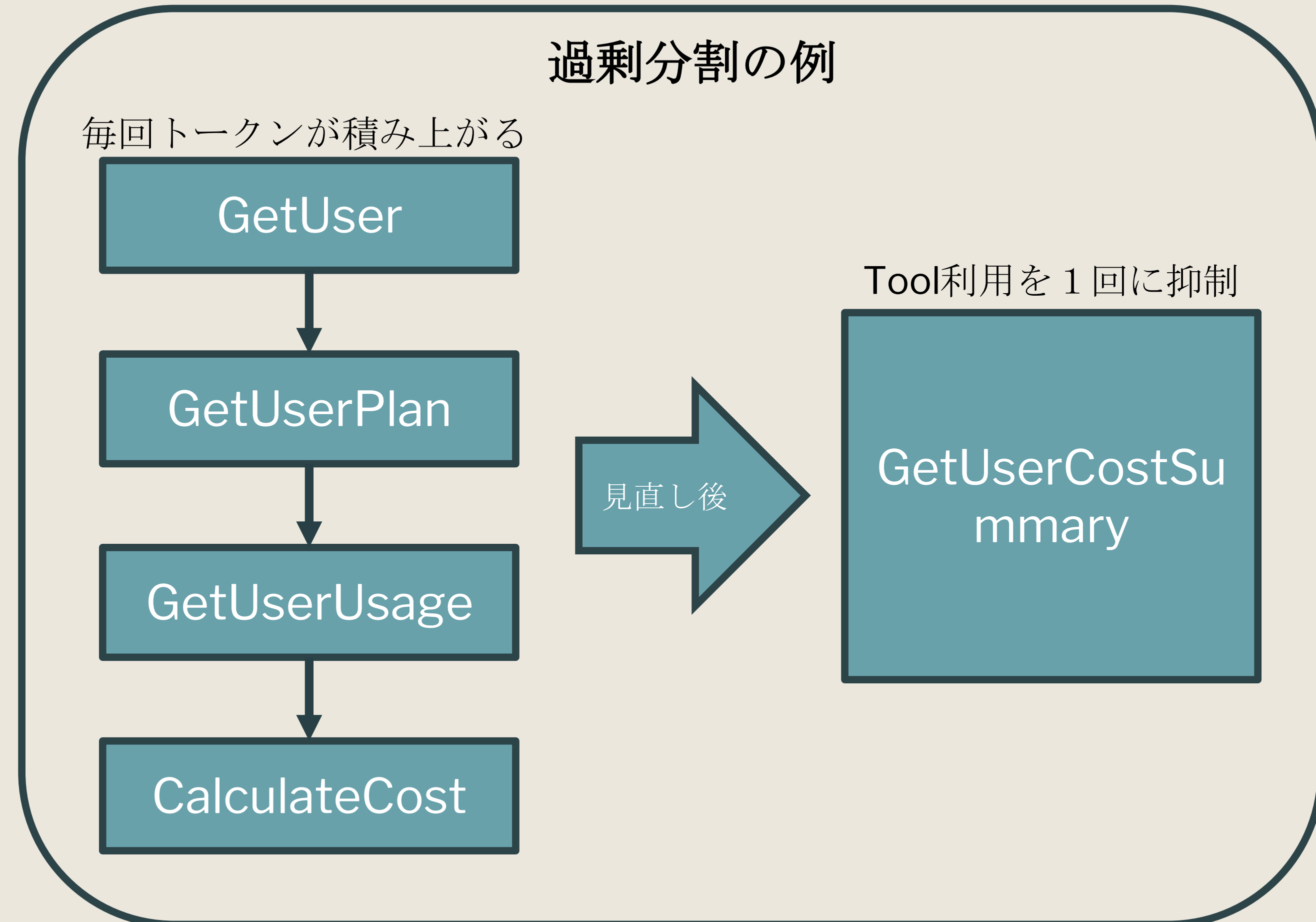
⑦ Stateful Efficiency：出力とToolのスリム化

- **Tool Schema最適化：**
  - *Tool定義のスキーマ自体がトークンを消費します。不要なフィールドや冗長な説明は削りましょう。*
  - *使わないTool定義がある場合は、設定から除外するのも有効です。*

Before（冗長）	After（必要最小限）
{ "description": "ユーザー情報を取得します。このツールはユーザーIDを受け取り...", "parameters": { "user_id": {...}, "include_history": {...}, "format": {...} } }	{ "description": "ユーザー情報を取得", "parameters": { "user_id": { "type": "string" } } }

⑦ Stateful Efficiency：出力とToolのスリム化

- **Tool設計の見直し：**
  - *Toolを細かく分割しすぎると、1つのタスクを完了するために複数のToolを何度も呼び出すことになります。APIコールのたびにトークンが積み上がるため、コストは指数的に膨らみやすくなります。*
  - *単機能に分割する前に、本当にその粒度が必要かを見直しましょう。*



⑧ Observability：最適化は計測から

「何にいくらかかっているか」が見えない状態では、どこを改善すべきかわかりません。まず計測の仕組みを整えることが、すべての最適化の出発点です。

「計測すべきメトリクス」

メトリクス	目的
入力・出力トークン数	コストの内訳把握
モデル別・エンドポイント別コスト	ルーティング最適化の判断
キャッシュヒット率	Prompt Cachingの効果測定
レイテンシ vs コスト	モデル選定の根拠
エラー率・リトライ数	無駄なAPIコストの検出

「ツールのメトリクス」

ツール名	メトリクス
Amazon Bedrock	CloudWatchでネイティブに可視化
LangSmith / LangFuse	リクエスト単位でトレース・コストを追跡
カスタムロギング	トークン数・モデル・レイテンシを記録して独自集計

改善サイクル：計測 → ボトルネック特定 → 改善 → 再計測
定期的にコストが高いエンドポイント・ユーザーのトップNを確認し、施策の前後でメトリクスを比較して効果を定量的に評価しましょう。

まとめ

今日話した内容、全部一度にやる必要はないです。

まず「自分のサービスが何にいくら使っているか」を把握するところから始めてみてください。それだけで見えてくるものがあります。

よければ、まず3つだけ試してみてください。

1. **モニタリングを入れる**（何にいくらかかっているかを把握することが第一歩です）
2. **Output Token制限を設定する**（すぐにできて効果が高いです）
3. **Prompt Cachingを有効にする**（共通プロンプトがあれば即効性があります）

ご清聴いただきありがとうございました。