生成AIの確率性と
マルチエージェントの限界を考える
― PromptからTrajectoryへ ―

矢印キー（左右）でスライドを移動できます

Speaker notes

表紙

この資料では、生成AIを「高性能だが少し不安定な道具」として眺めるところから始め、なぜその不安定さが生じるのかを生成原理から整理します。そのうえで、Multi-Agent化によって問題が自動的に解消されるわけではないことを確認し、最終的に Prompt だけでなく生成系列全体、すなわち Trajectory を設計するという考え方へ進みます。

副題を「PromptからTrajectoryへ」としているのは、Prompt Engineering を否定するためではありません。Prompt は重要です。ただし Agent が長時間・多段階に動くようになると、最初の入力条件だけでなく、その後に続く判断・ツール実行・成果物生成・再入力まで含めた系列全体が設計対象になります。

終盤では Claude Code / Cursor の Plan・Rewind/Revert・Worktree(s) を、まず製品が実装している事実として確認します。その後、それらを「切る・戻す・分ける」という Trajectory 制御の観点から読み直し、さらに複数 Harness 間を制御する multi-ai-cli の位置づけへ接続します。

この資料の読み方

会場で表示するスライドは、できるだけ図と命題だけに絞っています。一方、この Speaker Notes は「登壇者が読む台本」ではなく、資料を持ち帰った人が後から単独で読み直しても論理を追えるようにした解説です。そのため、スライド上では省略した数式、用語の限定、技術的な注意、図の読み方を意図的に詳しく書いています。

特に 5〜7 は生成AIの最小限の生成原理、10〜12 は Crowding / Drift の説明モデル、17〜24 は Prompt から Trajectory、Harness、Orchestration へ議論を広げる部分です。すでに自己回帰生成を知っている読者は 8 以降へ進んでも構いません。

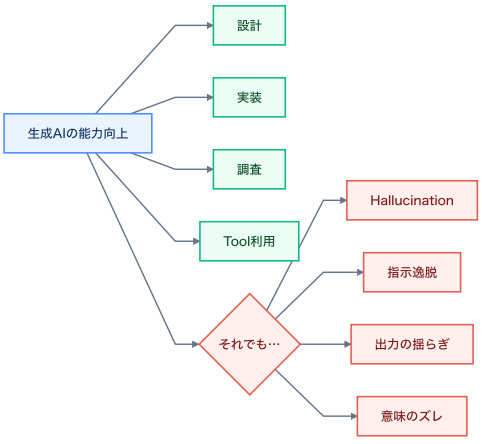
本稿で使う記号は次で固定します。

- x_0: 生成開始時点の条件（Prompt、System Prompt、Skills、Context、権限などをまとめたもの）
- y_t: その段の生成結果
- s_t: ファイル、テスト結果、権限など、モデルの外側の状態
- τ: 初期条件から最終成果物までの系列全体（Trajectory）

なお、本資料に出てくる数式には、一般的なTransformerの式と、本資料の設計意図を説明するための模式式が混在します。後者はLLM一般を厳密に記述する定理ではなく、設計上の構造を分けて考えるためのモデルです。

1. 便利だけれど、どこか不安定な生成AI

生成AIは大きく進化したが、安定したソフトウェアと同じ感覚ではまだ扱えない



- 設計・実装・調査・Tool利用まで能力は拡大している
- 一方でHallucination、指示逸脱、出力の揺らぎ、意味のズレは残る

Hallucination | Context依存 | 出力の揺らぎ | 高性能 ≠ 決定的

Speaker notes

1. 便利だけれど、どこか不安定な生成AI

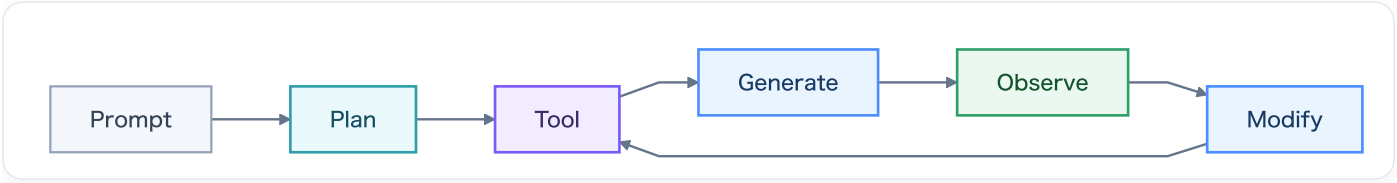
生成AIはここ数年で急速に能力を広げました。文章生成だけではなく、調査、設計、コード生成、テスト、外部ツールの利用まで、一連の作業をかなり高い水準で実行できます。一方で、同じような依頼でも出力が揺れる、条件を一部取りこぼす、存在しない事実をもっともらしく述べる、といった現象は現在も残っています。

ここで重要なのは、「能力が上がった」ことと「決定的なソフトウェアになった」ことを分けることです。従来型ソフトウェアでは、同じ状態・同じ入力・同じ実装なら同じ結果を返すことを基本的に期待します。しかし生成AIは、入力文脈に対して候補のスコアや分布を形成し、その上で生成します。デコード設定によって再現性を高めることはできますが、モデルの内部計算そのものが「唯一の正解を格納して取り出す」構造になるわけではありません。

この資料では、この不安定さを単なる欠陥や成熟不足として扱いません。むしろ「確率的な生成器をどう実務システムへ組み込むか」という設計問題として考えます。この見方を採ると、モデルをさらに賢くすることとは別に、モデルの外側へ制御・検証・状態管理を置く意味が見えてきます。

2. そして、自律型Agentへ

生成AIは「答えるAI」から「連続して仕事をするAI」へ進んでいる



- 計画・ツール実行・観測・修正をループする
- 一回の応答ではなく長時間の生成系列が実務単位になる

Agentic AI | Tool Calling | Long-running Task | Autonomy

Speaker notes

2. そして、自律型Agentへ

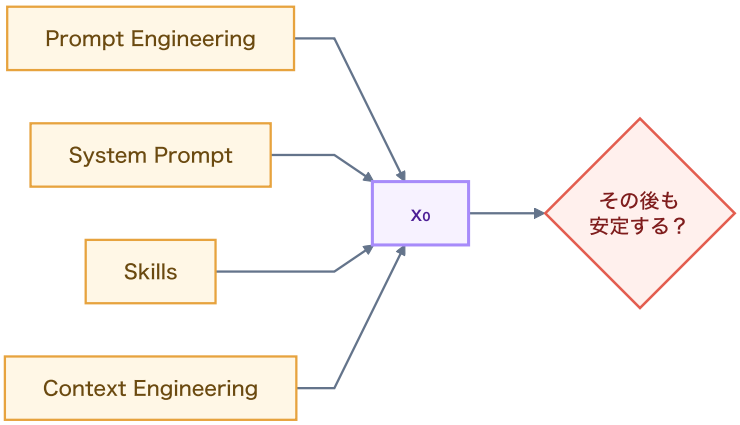
現在の生成AIは、一問一答のChatから、複数ステップを自分で進めるAgentへ移っています。典型的には、まず問題を解釈し、計画を作り、ファイルや検索・シェルなどのツールを使い、結果を観測し、必要なら計画や実装を修正して再びツールを使います。

この変化は非常に大きいものです。一回の生成が多少揺れても、人間がその場で確認するなら影響は限定できます。しかし Agent が十数回、数十回と生成とツール実行を連続させる場合、途中の判断が次の判断条件になります。つまり評価対象は「一回の回答品質」だけではなく、「長い系列の中でどの状態を通って最終結果に到達したか」に広がります。

自律性そのものには大きな価値があります。人間が逐一指示しなくても調査や実装を進められるためです。ただし、自律能力が高いことと、全工程を無制限に自律実行させることが最適であることは同じではありません。後半では、この差を Trajectory と権限設計の問題として整理します。

3. Promptを良くすれば安定するのか？

現在の多くの改善策は、まず生成開始時点の条件を良くする方向にある



- Prompt、System Prompt、Skills、Context設計は有効である
- ただし主に生成開始時点 x_0 を整える操作として理解できる

Prompt Engineering | System Prompt | Skills | Context Engineering | x_0

Speaker notes

3. Promptを良くすれば安定するのか？

生成AIを安定させるために、最も身近な方法は入力を改善することです。曖昧さを減らし、役割、前提、制約、出力形式、参照資料を明示する。System Prompt や Skills に再利用可能な判断基準を置く。必要なContextだけを渡す。これらはすべて有効です。

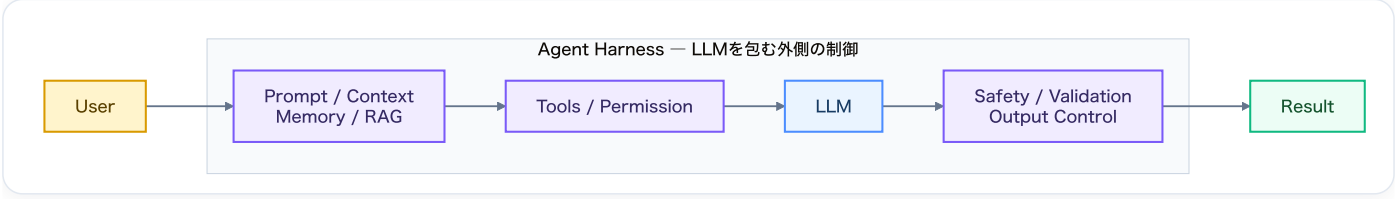
この資料では、これらを「生成開始時点の条件を整える」として一度まとめます。記号では初期状態を x_0 と置きます。もちろん実際の製品では System Prompt、会話履歴、検索結果、メモリ、権限、ツール定義など複数の要素から条件が構成されます。ここで x_0 は、それらを含む「生成を開始するときの条件」の簡略表現です。

重要な問いは、その後です。Agentでは、x_0 から一回だけ出力して終了するとは限りません。最初の出力が次のContextになり、ツールの結果が追加され、ファイルが変更され、その状態を読み直して次の判断を行います。したがって x_0 の品質を上げることは必要条件に近い一方、それだけで最終状態までの系列全体を保証することはできません。

この問いを残したまま、まず生成AIの仕組みを最小限だけ確認します。

4. 私たちが使っている生成AIは、LLMそのものではない

商用生成AIは、LLMと、それを包んで入力・実行・出力を制御するHarnessからなるシステムである



- 本資料では、LLMの入力・実行・出力を外側から制御する仕組みの集合を Agent Harness と呼ぶ
- Prompt / Context、Memory、RAG、Tools / Permission、Safety / Validation、Output Control は主にHarness側の機能である
- モデル性能と、ユーザーが観測する製品全体の性能は分けて考える

LLM | Agent Harness | Context | Tools / Permission | RAG | Validation

Speaker notes

4. 私たちが使っている生成AIは、LLMそのものではない

普段利用しているChatGPT、Claude Code、Cursorのような生成AI製品は、ニューラルネットワークとしてのLLMそのものではありません。ユーザー入力がそのままLLMへ入り、LLMが返した文字列がそのまま画面へ表示される、という単純な構造ではなく、モデルの周囲に多くの制御機構があります。

本資料では、この **LLMの入力・実行・出力を外側から制御する仕組みの集合** を、説明上 **Agent Harness** と呼びます。

なお、Harnessという言葉がすべての製品・論文で完全に同じ範囲を指すわけではありません。本資料では、後半で扱うOrchestrationとの階層を明確にするため、LLMを包み、その使い方を制御する外側の層をHarnessとして整理します。

図の読み方

図では、UserからResultまでの間に、一つのAgent Harnessを置いています。LLMはそのHarnessの内部で利用される生成器です。

Harness側には、例えば次のような機能があります。

Prompt / Context

ユーザーの入力だけでなく、System Prompt、会話履歴、検索結果、参照資料、Tool定義などを、どの順序・形式・量でLLMへ渡すかを決めます。

Contextを短くする、重要な制約を前方へ配置する、過去履歴を圧縮する、といった処理もモデルそのものではなく、外側のContext設計として実装されます。

Memory

会話中の状態や過去の情報を保存し、必要に応じて次のContextへ戻します。

ここで重要なのは、Memoryに保存された情報と、モデルの学習済みパラメータ θ は別物だということです。Memoryへ情報を書き込んだからといって、通常はLLMの重みそのものがその場で学習し直されるわけではありません。

RAG / Search

RAGでは、外部のDocument、Database、検索Indexなどから情報を取得し、その中から何をLLMへ渡すかを決めます。

厳密には、Vector DatabaseやSearch EngineそのものはHarnessとは独立したコンポーネントとして実装される場合があります。しかし、**検索をいつ実行し、何を取得し、その結果のどこをContextへ載せるかを決める制御** はHarness側にあります。

したがって本資料では、RAGをLLMの外側にあるContext供給機構としてHarness側へ含めて考えます。

Tools / Permission

現在のAgentは、LLMが文章を生成するだけではなく、Search、File操作、Shell、API、Database、GitなどのToolを利用します。

しかし、LLM自身がOSの権限を直接持っているわけではありません。どのToolを利用可能にするか、Readだけ許可するか、Write時には人間の承認を必要とするか、Sandbox内だけで実行するか、といった権限制御はHarness側の責務です。

Coding Agentでは、この部分が特に重要です。ファイル探索、差分適用、Shell実行、テスト、Git操作などは、LLMそのものの能力ではなく、HarnessがLLMと外部Softwareを接続することで実現されています。

Safety / Validation

LLMが生成した結果を、そのまま最終結果として採用する必要はありません。

Safety Policy、Schema Validation、Compiler、Test、Database、外部仕様などと照合し、条件を満たさなければ停止・修正・再生成することができます。

ここでも、正しさをモデル内部だけに期待するのではなく、外側から検証する構造が重要になります。

Output Control

生成結果をJSONなど特定形式へ制約する、不要部分を除去する、エラー時に再試行する、拒否する、別の処理へ分岐するといった出力後の制御もHarness側に置かれます。

モデル性能と製品性能は同じではない

この構造を見ると、ユーザーが観測している生成AI製品の性能は、LLM単体の性能だけでは決まらないことが分かります。

同じ、あるいは近い基盤モデルを利用していても、

- どのContextを渡すか
- Memoryをどう使うか
- どのToolを持たせるか

- Search / RAGをどう接続するか
- Write権限をどこまで許可するか
- どのタイミングでValidationするか
- 失敗時にRetryするか
- Stateをどこまで保持するか

によって、最終的な製品の振る舞いは大きく変わります。

したがって、**モデル性能と製品性能を混同しないこと** が重要です。

概念的には、LLMを M、Harnessを H とすれば、ユーザーが観測する振る舞いは単純な

$$M(x)$$

ではなく、Harnessによって入力・実行・出力を制御された系として考える方が適切です。

ただし、HarnessとLLMはAgent実行中に何度も相互作用するため、単純な関数合成だけで実装を表現できるわけではありません。ここでの記号は、**LLMの外側にも振る舞いを決める層がある**ことを示すための模式表現です。

後半への接続

このページでは、LLMから見た『外側』としてAgent Harnessを導入しました。

後半では、Claude CodeやCursorのPlan、Rewind/Revert、Worktreeなどが、このHarnessの中で生成過程をどのように制御しているかを見ます。

さらにMulti-Agentになると、外側はHarnessだけでは終わりません。

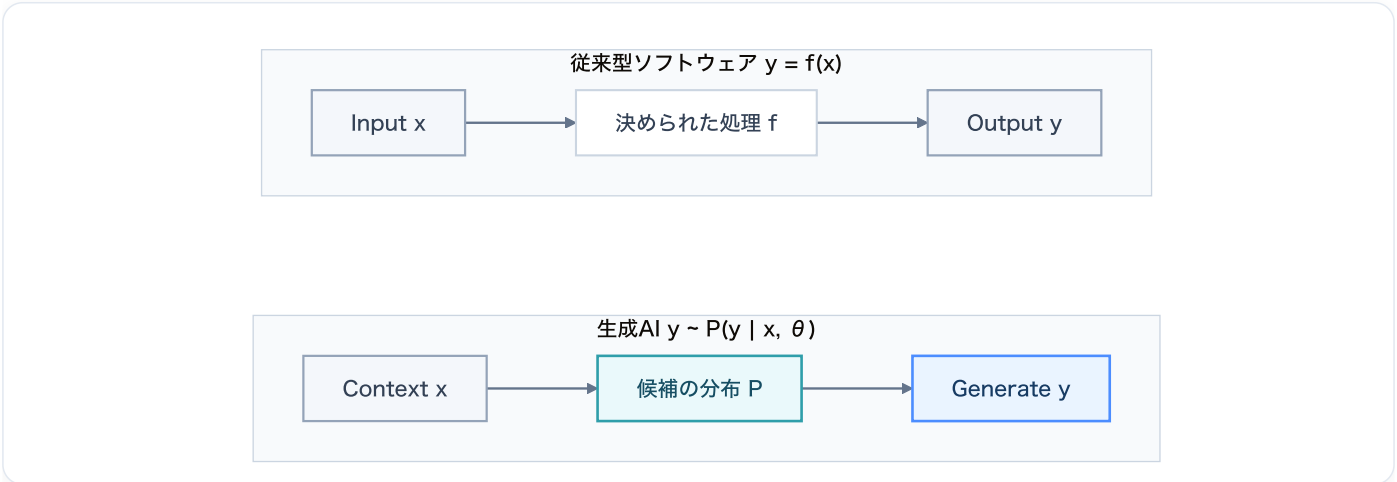
$$\text{LLM} \rightarrow \text{Agent Harness} \rightarrow \text{Orchestration}$$

というように、複数Harness間の順序、Artifact、Context、Permission、Gateを制御する、さらに一段外側の層が必要になります。

つまり、このページは後半の『外は一つではない』という議論の出発点です。

5. 従来型ソフトウェアと生成AIは、少し違う

生成AIを「入力に対して一つの答えを返す関数」と考えると、振る舞いを誤解しやすい



- 従来型の直感は y=f(x)
- 生成AIは条件付き分布から出力を生成する系として見る

Deterministic | Probabilistic | Conditional Generation | Sampling

Speaker notes

5. 従来型ソフトウェアと生成AIは、少し違う

このページの要点

このプレゼンでスライド上に大きく出す数式は、基本的に二つです。従来型ソフトウェアの直感を

$$y = f(x)$$

と置きます。入力 x に対して、人間が実装した処理規則 f を適用し、出力 y を得るという見方です。現実のソフトウェアにも時刻、乱数、外部状態などは存在しますが、設計上は状態と規則を明示し、再現可能性を高める方向へ作ります。

生成AIは、理解のためには

$$y \sim P(y \mid x, \theta)$$

と置く方が適切です。θ は学習済みパラメータ、x は現在与えられている文脈、P はその条件の下で形成される出力分布です。

「確率的」とは何か

ここで重要なのは、モデル内部に「正解文が一つ格納されていて、それを検索して返す」と考えないことです。モデルは現在の文脈から、次に続き得るTokenの候補へスコアを付け、分布を形成しながら逐次的に文章を作ります。

文章全体 $y = (y_1, \ldots, y_T)$ の生成は、概念的には

$$P(y \mid x, \theta) = \prod_{t=1}^{T} P(y_t \mid x, y_{<t}, \theta)$$

と分解して考えられます。つまり「文章全体を一度に一つ選ぶ」のではなく、「ここまでに何が生成されたか」を条件に、次Tokenの分布を何度も作り直していきます。

技術的な注意

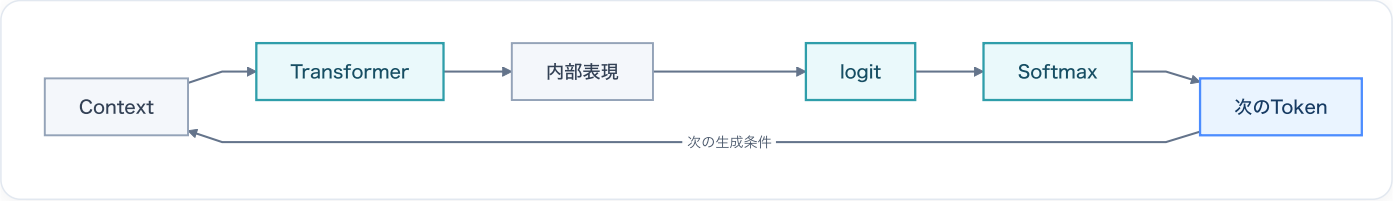
ここでいう確率は「その回答が正しい確率」ではありません。現在の文脈のもとで、そのTokenや系列がどの程度続きやすいかを表す分布です。高確率だから事実として正しい、という意味ではありません。この違いが次のHallucinationの説明へ直結します。

また、temperatureを低くしたりgreedy decodingを使ったりすれば、同じ条件で同じTokenを選びやすくすることはできます。しかしそれは、モデル内部が決定論的な業務ロジックへ変わるという意味ではありません。候補scoreと分布を計算する生成機構は残ります。

後半で扱うHallucination、Crowding、Drift、Multi-Agentの連鎖は、すべてこの「分布を形成しながら逐次生成する」という性質から見ていきます。

6. 生成AIは、次の言葉を繰り返し生成する

現在の文脈から候補を作り、
生成した結果を次の文脈へ加えていく



- Transformerで文脈表現を更新しlogitを作る
- Softmax等を通して次Tokenを選び、そのTokenが次のContextになる

Transformer | logit | Softmax | Token | Context

Speaker notes

6. 生成AIは、次の言葉を繰り返し生成する

このページの要点

このスライドで重要なのは、「現在のContextから次Tokenの候補を作り、選んだTokenをContextへ追加して、また次Tokenを作る」という自己回帰的な循環です。Transformerの詳細をすべて覚える必要はありません。

図の最後にある

```
次のToken → Context
```

という戻り矢印が最も重要です。生成AIは、自分が直前に生成したものを次の生成条件の一部として再び利用します。

Transformer内部では何が起きているか

Transformer内部では、Attentionを含む複数の層がTokenごとの文脈表現を更新します。代表的なAttentionは

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

と書けます。これはモデル全体の一部であり、実際にはMLP、残差接続、正規化なども含まれます。

最終的な内部表現から、語彙中の各Token候補へlogit z_i が計算されます。Temperature T を使う場合の典型的なSoftmaxは

$$P(i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

です。Softmaxはlogitを候補間の相対分布へ変換します。その後、greedy、sampling、top-p等のデコード規則で実際の次Tokenを決めます。

「文章全体の確率」と「次Tokenの確率」

前ページの

$$y \sim P(y \mid x, \theta)$$

は、この逐次生成をまとめて書いたものです。より細かく見れば、

$$P(y \mid x, \theta) = \prod_t P(y_t \mid x, y_{<t}, \theta)$$

です。

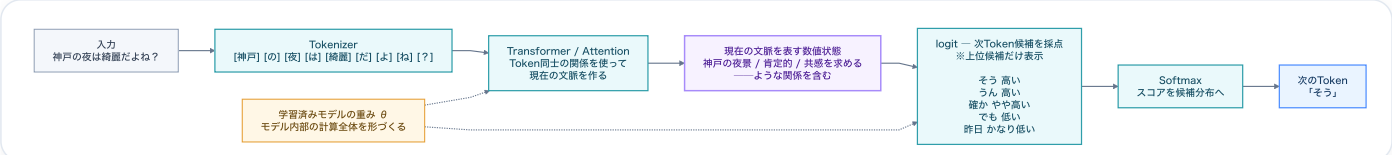
つまり「あー、だから確率なのか」という理解はここにあります。各時点で唯一の文章を取り出しているのではなく、候補分布を作り、その中から次Tokenを決めています。

次ページへの接続

次ページでは、この抽象図を「神戸の夜は綺麗だよね？」という文章で具体化します。そこでTokenizer、学習済みの重み、内部表現、logit、Softmaxがどうつながって見えるかを追います。

7. 具体例：文章はどう次のTokenへ変わるのか

文章はToken列へ変換され、Token間の関係を文脈依存の内部表現へ更新しながら、次Tokenの候補分布を作る



- 完成した返答を保存して検索しているのではなく、学習済みの重みと現在の文脈から、次のToken候補をその場で計算している
- Softmaxで得るのは「正しい確率」ではなく、現在の文脈で「次に続く候補分布」である
- Tokenizerは文章をモデルが扱うToken列へ変換する。実際の分割単位はモデルごとに異なる

Tokenizer | Embedding | Attention | Contextual Representation | Next Token

Speaker notes

7. 具体例：文章はどう次のTokenへ変わるのか

このページの要点

前ページでは、

```
Context → Transformer → 内部表現 → logit → Softmax → 次Token
```

という抽象的な生成ループを示しました。このページでは、その内部で文章がどのような形式へ変換されていくのかを、説明用の具体例

```
神戸の夜は綺麗だよね？
```

で追います。

このページの結論を先に言えば、

> **完成した返答を検索しているのではなく、現在の文脈と学習済みパラメータから、次に続くToken候補をその場で計算している**

ということです。

1. Tokenizer — まず文字列をToken列へ変える

生成AIが最初に「神戸＝場所」「夜＝時間」「綺麗＝形容詞」と人間のような品詞解析をしてから考えるわけではありません。まずTokenizerが文字列をモデルの語彙に対応するToken ID列へ写像します。

スライドでは説明のため、

```
［神戸］［の］［夜］［は］［綺麗］［だ］［よ］［ね］［？］
```

と表示しています。ただし、これは模式例です。実際のToken境界はTokenizerの語彙や方式によって異なり、「神戸」が一Tokenになる保証もありません。日本語では単語、サブワード、文字に近い単位が混在し得ます。

2. Embedding — Tokenを計算可能なベクトルへ変える

Token IDを t_i とすると、概念的には

$$e_i = E(t_i) + p_i$$

のようにToken embeddingと位置情報を組み合わせ、各Tokenを高次元ベクトルとして扱います。モデル内部で「神戸」「夜」「綺麗」という文字列がそのまま流れているわけではありません。

3. Transformer — 周囲との関係で表現を更新する

Transformerの各層では、各Tokenの表現が他のTokenとの関係を受けながら何度も更新されます。ここで大事なのは、文法規則をコードとして逐一適用しているわけではない一方、学習を通じて文法的・意味的・会話的な関係を内部表現として利用できるようになっていることです。

例えば「夜」単独なら時間帯を表すTokenですが、

```
神戸 + 夜 + 綺麗
```

という文脈では「神戸の夜景」に関係する方向が強くなり得ます。また、

```
綺麗 + だよね？
```

という並びは、肯定・同意・共感を求める会話的な文脈に影響します。

図中の

```
神戸の夜景 / 肯定的 / 共感を求める
```

は人間向けの説明ラベルです。モデル内部に「場所：神戸」「意図：共感」という明示項目が保存されているわけではなく、実体は高次元の分散表現です。

4. 学習済みモデルの重み θ はどこで効くのか

ここは特に重要です。

学習済みモデルの重みは、logitを作る最後だけに突然登場するものではありません。Embedding、Attention、MLP、出力射影など、モデル内部の計算全体を形づくっています。大量の学習データから獲得された「どのような文脈で、どのような表現が続きやすいか」という統計的・構造的な情報が、重み θ に分散して反映されています。

したがって、

```
現在の文脈
      ＋
学習済みモデルの重み
      ↓
次Token候補のscore
```

という見方が初心者向けには分かりやすいでしょう。

5. logit — 語彙全体のToken候補へscoreを付ける

生成位置の最終的な内部状態を h とすると、語彙中の各Token候補へのlogitは概念的に

$$z = W_{\text{out}} h + b$$

のような線形射影で得られます。

実際には語彙中の膨大なTokenすべてにscoreがあります。スライドの

```
そう      高い
うん      高い
確か      やや高い
でも      低い
昨日      かなり低い
```

は、その中の一部を説明用に抜き出した模式例です。特定モデルから取得した実測logitではありません。

また、モデル内部に

```
返答候補A：「そうだね」
返答候補B：「うん、綺麗だね」
返答候補C：「確かにそうだね」
```

という完成済み返答集があり、そこから一つを検索しているわけではありません。「そう」を選んだ後は、その「そう」もContextへ入り、次はまた語彙全体に対して次Tokenのscoreを計算します。

6. Softmax — scoreを候補分布へ変える

logitをSoftmaxへ入れると、

$$P(i) = \frac{\exp(z_i / T)}{\sum_j \exp(z_j / T)}$$

のように候補間の相対分布が得られます。

ここで最も誤解してはいけないのは、

> **この確率は「その回答が正しい確率」ではない**

という点です。例えば「そう」が45%だったとしても、「そう」が45%の確率で事実として正しいという意味ではありません。「現在のContextのもとで、次Tokenとして『そう』が相対的に続きやすい」という意味です。

この区別が、そのまま次ページのHallucinationにつながります。

7. 「理解」「感情」とどう考えるか

Transformerは意味的・文法的な構造を内部表現として扱うことができます。そのため「何も意味を扱っていない」と言い切るのも正確ではありません。

一方、この生成過程を説明するために、人間と同じ主観的な意味理解や感情体験を仮定する必要もありません。「そうだね」と出力したからといって、神戸の夜景を思い浮かべ、「綺麗だ」と感じ、共感した結果として返答した、と説明する必要はありません。

このページで確認できるのは、少なくとも

```
Tokenizer
→ 文脈依存の内部表現
→ 学習済みパラメータ
→ logit
→ Softmax
→ 次Token
```

という計算機構だけで、その返答生成を説明できるということです。

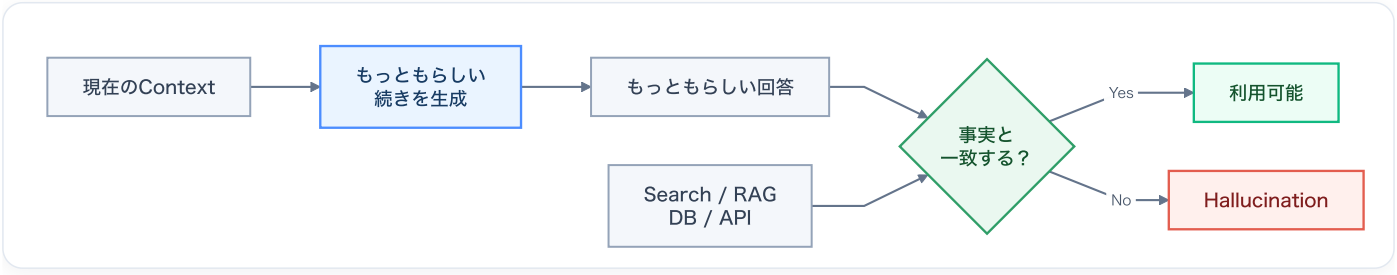
なお、大規模モデルが学習データ中の文やパターンを記憶し、近い形で再現する場合はあります。したがって「一切記憶しない」という意味ではありません。ここで否定しているのは、基本アーキテクチャが「完成済みの返答データベースを検索する仕組み」である、という見方です。

次ページへの接続

ここまで理解すると、「生成AIはなぜ確率的なのか」が具体的になります。同時に、「続きやすさ」と「現実に正しいこと」は別だと分かります。次ページでは、その差がHallucinationとしてどう現れるかを見ます。

8. Hallucinationは「故障」だけではない

もっともらしい答えでも、事実とは限らない



- モデルは「次に続きやすいToken」を生成しており、それ自体が事実性を保証するわけではない
- 事実性が必要な箇所はSearch/RAG/API/DBなどの外部根拠へ接続し、Validationする

Hallucination Grounding RAG Search Validation

Speaker notes

8. Hallucinationは「故障」だけではない

このページの要点

前ページでは、生成AIが完成した回答を検索しているのではなく、現在のContextと学習済みモデルの重みから、次に続きやすいTokenの候補をその場で計算していることを見ました。

ここで重要なことがあります。

> **「次に続きやすい」ことと、「事実として正しい」ことは別です。**

これがHallucinationを理解するための重要な出発点です。

生成AIは、現在の文脈に対してもっともらしい続きを作ることが非常に得意です。しかし、その文章がどれだけ自然でも、内容そのものが現実の事実と一致しているとは限りません。

例えば、実在しない論文名、存在しないAPI、誤った日付、架空の人物情報などでも、周囲の文章として自然につながるToken列を作ることはできます。

つまり、

> **もっともらしい ≠ 正しい**

です。

なぜ自然なのに間違えるのか

前ページまでの生成過程を思い出します。

```
現在のContext
      ↓
Transformer
      ↓
内部表現
      ↓
logit
      ↓
Softmax
      ↓
次に続きやすいToken
```

この処理の目的は、現在の文脈に対して次にどのTokenが続きやすいかを計算することです。

モデル内部に、

```
この文章は事実である
この文章は事実ではない
```

という万能な外部世界の事実表があり、それを毎回照合してからTokenを選んでいるわけではありません。

もちろん学習済みモデルの重みには、学習データから得た膨大な知識や統計的関係が反映されています。そのため、多くの場合は事実に合った回答を生成できます。

しかし、**学習した知識を使ってもっともらしい文章を生成できることと、生成時点で外部世界の事実を検証していることは別です。**

この違いが、Hallucinationが発生し得る理由の一つです。

「高確率」と「正しい」は別

5章では生成AIを、理解のために

$$y \sim P(y \mid x, \theta)$$

と表しました。

ここで、

- x は現在のContext
- θ は学習済みモデルのパラメータ
- $P(y \mid x, \theta)$ は、その条件のもとでどの出力が生成されやすいか

を表しています。

しかし、私たちが本当に知りたいのが、現実世界 W に対して

> 「この回答は正しいのか」

であるなら、別の条件が必要です。

重要なのは、

$$P(y \mid x, \theta)$$

が高いことではなく、生成された y が外部の根拠と一致していることです。

重要なので、ここは明確に分けます。

> **生成確率は、事実の正しさの確率ではありません。**

例えば、あるTokenが70%で選ばれたとしても、

> 「その内容が70%の確率で真実である」

という意味ではありません。

それは、

> **現在のContextでは、そのTokenが他の候補より続きやすかった**

という意味です。

この違いを理解すると、Hallucinationは単純な『故障』ではなく、生成AIの基本的な生成原理と外部事実の間にあるギャップとして理解できます。

Hallucinationは「何も知らないから」だけ起きるわけではない

Hallucinationを、単純に

> 「モデルがその事実を知らなかったから嘘を作った」

と考えるのも少し単純すぎます。

モデルが関連知識を持っていても、

- Contextの解釈を誤る
- 似た知識を混同する
- 日付や数値を取り違える
- 不十分な情報から自然な続きを補完する
- 古い学習内容を現在の情報として使う

といった理由で、もっともらしいが誤った回答を生成することがあります。

したがって、Hallucination対策を『モデルをもっと賢くする』だけに求めるのではなく、必要な箇所を外部根拠へ接続するという設計が重要になります。

Groundingの役割

そこで使われるのがGroundingです。

Groundingとは、この文脈では生成AIの回答を、外部の根拠へ接続する考え方です。

例えば、

- Search
- RAG
- Database
- API
- 業務マスタ
- 正式な仕様書

などから情報を取得し、それをContextへ加えます。

```
外部の根拠
    ↓
Search / RAG / DB / API
    ↓
Context
    ↓
生成AI
```

これにより、モデルの学習済み知識だけに依存するのではなく、現在の外部情報を使って生成できるようになります。

第4章で説明したように、このSearchやRAGも、実務ではLLMそのものではなく、主としてHarness側が提供する外部制御です。

つまりHallucination対策の一部は、すでに

> **モデル内部を変えるのではなく、モデルの外側を設計する**

という方向で行われています。

これは後半のTrajectoryやOrchestrationの議論にもつながります。

Groundingすれば必ず正しくなるわけではない

ただし、SearchやRAGを付ければHallucinationが完全に消えるわけではありません。

例えば、

- 検索結果そのものが誤っている
- 古い情報を取得する
- 必要な文書を検索できない
- 関係のない文書を取得する
- モデルが取得文書を読み違える
- 根拠にはない内容を補完する
- 引用した根拠と結論が一致していない

といった失敗は残ります。

したがって、高い事実性が必要な処理では、

```
根拠を取得する
      ↓
生成する
      ↓
検証する
```

という三つを分けて考える方が安全です。

特に業務システムでは、生成AI自身に

> 『自分の回答が正しいか確認してください』

と再度尋ねるだけでは、同じ生成系による自己評価になってしまいます。

必要に応じて、Database、API、Schema、Compiler、Test、業務ルール、人間など、生成系列の外側にある固定的な根拠と比較する必要があります。

図の読み方

このページの図は非常に単純です。

```
現在のContext
      ↓
もっともらしい続きを生成
      ↓
もっともらしい回答
      ↓
事実と一致する？
```

ここまでは生成AIの仕事です。

しかし、

事実と一致する？

という判定には、モデル内部の生成確率とは別の根拠が必要になります。

そこで横から、

```
Search / RAG
DB / API
```

を接続しています。

この図で伝えたいことは、

> **生成と事実確認は、同じ仕事ではない**

ということです。

生成AIには、もっともらしい回答を生成させる。

事実性が必要なら、外部根拠へGroundingし、必要に応じてValidationする。

この役割分担が重要です。

次ページへの接続

ここまで見ると、

> 『では、SearchやRAGでたくさん情報をContextへ入れれば安全になるのでは？』

と思うかもしれません。

これは非常に自然な発想です。

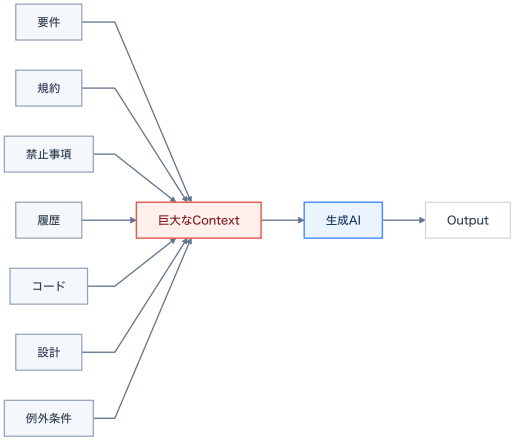
しかし、**情報を増やすことと、重要な情報を強く効かせることは同じではありません。**

Contextが大きくなると、モデルが同時に扱う意味要素も増えます。

次ページでは、Long Contextになると何が起きるのかを見ていきます。

9. Contextを増やせば安全になるわけでもない

情報量を増やすことと、重要な条件を強く効かせることは同じではない



- 長いContextでは参照・競合する意味要素も増える
- 必要情報を選択し、固定点や制約を見えやすくする設計が必要

Long Context | Context Engineering | 情報選択 | 制約

Speaker notes

9. Contextを増やせば安全になるわけでもない

このページの要点

生成AIの性能を上げようとすると、「必要そうなものを全部Contextへ入れればよい」と考えがちです。要件、規約、会話履歴、コード、設計書、例外条件、過去のレビューをすべて入れれば、情報欠落は減るように見えます。

しかし情報量が増えることは、モデルが参照し得る意味要素や候補関係も増えることです。TransformerはContext内の全情報を、単純に同じ重みで保持・利用するわけではありません。現在のqueryや各層の内部表現に応じて、どの要素がどの程度効くかが変わります。

Context Engineeringは「詰め込み」ではない

実務的には、Context Engineeringを

> **最大量を詰め込む技術ではなく、その時点の判断に必要な情報を選び、識別しやすい形で渡す技術**

と考える方が分かりやすいでしょう。

重要な禁止事項や元要件を、長い会話履歴の途中に埋め込むより、短い固定ファイル、明示的な制約、スキーマ、テストOracleとして外部化した方が強い場合があります。

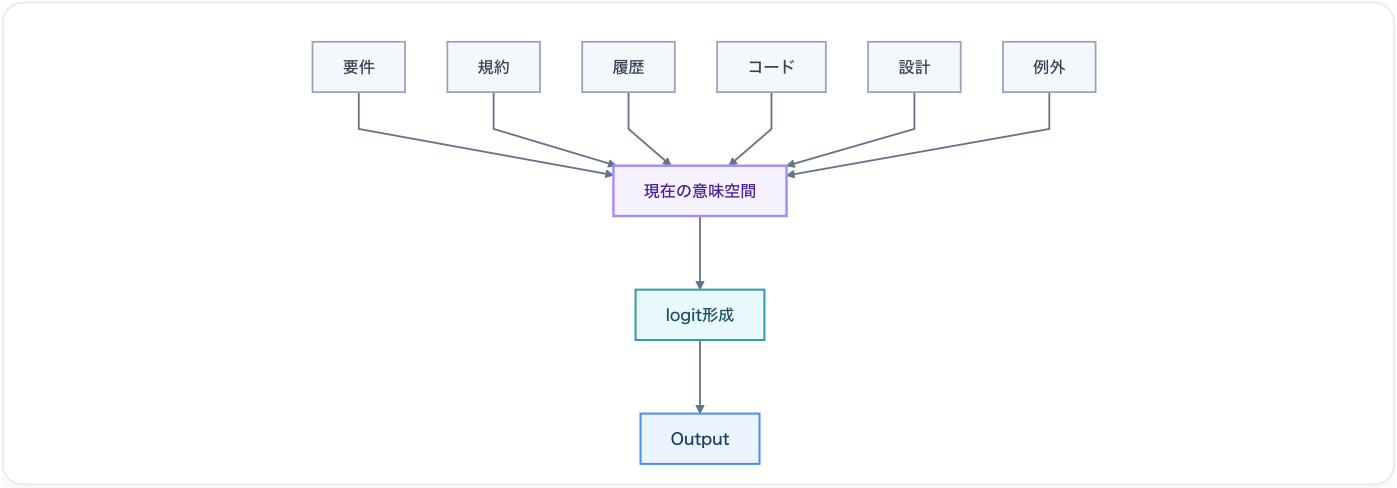
また、Long Contextの問題は「長いほど必ず悪い」という単純な話ではありません。長いContextによって必要情報へアクセスできる利点もあります。問題は、必要情報・不要情報・競合情報をどう構成し、どの時点で何を渡すかです。

次ページへの接続

次ページでは、同一生成時点のContext内に多数の意味要素が共存したときの相対的な競合を、本資料では説明上「空間方向」の問題として整理します。

10. Softmax Crowding — 「空間」の問題

同一時点の意味空間に多数の意味要素が存在すると、重要条件の相対的な影響が弱くなり得る



- 同じ生成時点に存在する意味要素の競合を見る
- 「空間」も Softmax Crowding も説明用の軸・呼称であり、物理空間や標準用語ではない

Softmax Crowding | 意味空間 | 競合 | SPACE

Speaker notes

10. Softmax Crowding — 「空間」の問題

用語の位置づけ

ここでは、同じ生成時点のContextの中に多数の意味要素が存在し、重要な情報の相対的な寄与が埋没し得る問題を、直感的に「空間方向」と呼びます。これは物理的な空間や厳密な幾何空間を意味しません。「一つの時点に横に並んで存在する情報同士の競合」を、後で扱う時間方向のDriftと区別するための説明上の軸です。

また、**Softmax Crowding** も一般的なTransformer教科書に標準定義された用語として使っているのではありません。本資料で、正規化された競合によって重要要素の相対的な寄与が薄まり得る構造を説明するために用いている呼称です。

数理的な最小モデル

よりTransformerに近い形で見るなら、単一のAttention head、単一query q に対するkey k_i の重みを

$$\alpha_i = \frac{\exp(q^\top k_i / \sqrt{d})}{\sum_j \exp(q^\top k_j / \sqrt{d})}$$

と考えます。

ある重要情報に対応するkeyのscoreが高くても、他にも無視できないscoreを持つkeyが多数あれば、Softmaxで正規化された相対weightはそれらの間で配分されます。ここから、

> **重要情報が存在することと、その情報がその時点の生成へ十分強く効くことは同じではない**

という設計上の直感を取り出しています。

重要な限定

実際のTransformerは多層・Multi-Headであり、残差接続やMLPも含みます。Contextを増やせば特定の制約へのAttentionが必ず単調に低下する、という定理ではありません。また、最終出力TokenのSoftmaxと、Attention内部のSoftmaxは役割が異なります。

したがって、このページは「Context要素が一つの巨大Softmaxで直接競争している」と主張しているのではありません。多数の候補・関係を正規化しながら処理する構造の中で、重要な情報の相対的寄与が埋没し得る、という実務上の説明モデルです。

次ページへの接続

ここで見ているのは一回の生成時点です。次ページでは、一回の生成結果が次の条件へ入り、その意味変化が系列に沿って累積する「時間方向」を扱います。

11. Semantic Drift — 「時間」の問題

前段の小さな意味変化が、次段では入力条件になる

- 一回の出力が次の生成条件へ取り込まれる
- 生成ステップが進むにつれて元の意味からの距離が累積し得る

Semantic Drift | Generation Step | 累積 | TIME

Speaker notes

11. Semantic Drift — 「時間」の問題

このページの要点

Semantic Driftは、一回の生成内部で情報が競合する問題ではなく、生成や成果物変換を繰り返したときに、意味が系列方向へずれていく問題です。本資料では説明上これを「時間方向」と呼びます。

例えば元要件 x_0 が

```
20歳未満は購入不可
```

だったとします。次の仕様 y_1 で

```
成人のみ購入可能
```

と言い換えられ、さらに実装 y_2 で

```
18歳以上
```

と解釈されれば、各段階単体では自然でも、元の制約からは離れています。

実際の生成条件は直前出力だけではない

単純な説明では

$$y_{t+1} \sim P(y_{t+1} \mid y_t, p_t, \theta_t)$$

と書けますが、実際のAgentでは次の生成条件は直前出力 y_t だけではありません。会話履歴、Tool結果、ファイル、System Prompt、権限、検索結果などを含む現在のContext C_t を考える方が実態に近いです。

概念的には、

$$y_{t+1} \sim P(y_{t+1} \mid C_t, \theta_t)$$

そして外部状態や新しい出力によって、

$$C_{t+1} = U(C_t, y_{t+1}, \mathrm{ToolResult}_t, s_t)$$

のように次のContextが更新される、と考えられます。

この式は実装仕様ではなく、「前段の生成だけでなく、Toolや外部状態も次の条件になる」ことを示す模式式です。Semantic Driftを単純な一次Markov連鎖として扱う意図はありません。

Driftの模式モデル

ズレを状態量 D_t として抽象化すると、本資料で使う説明モデルは

$$D_{t+1} = D_t + g_t(D_t) + \eta_t - \delta_t$$

です。

- D_t: すでに蓄積しているずれ
- $g_t(D_t)$: 既存のずれが次の変換へ影響する項
- η_t: 新しい揺らぎや解釈差
- δ_t: 外部から挿入される修正

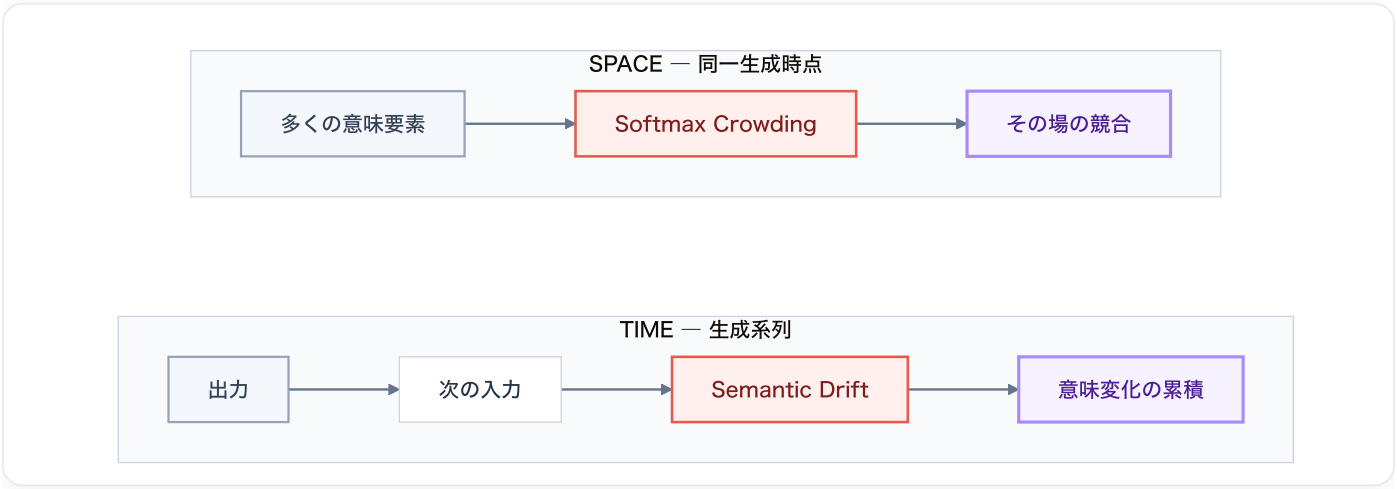
これはLLM一般を厳密に記述する定理ではありません。設計上、「既存のずれ」「新しい揺らぎ」「外部修正」を分けて考えるための模式モデルです。

なぜ外部修正が重要か

途中成果物の意味が少し変わると、その変化は次段では「入力」として扱われます。つまり、前段で生じた誤差は自然に消えるとは限りません。そこでValidation、Human Gate、Checkpoint、Rewindのような δ_t を外部から挿入する仕組みが重要になります。

12. 「空間」と「時間」で見る生成AIのズレ

一回の生成内部と、生成系列全体を分けて考える



- Crowdingは同一時点における意味要素の競合
- Driftは生成系列に沿った意味変化の累積

SPACE TIME Crowding Drift

Speaker notes

12. 「空間」と「時間」で見る生成AIのズレ

ここまでを一度整理します。Softmax CrowdingとSemantic Driftは同じ現象ではありません。前者は、一つの生成時点で複数の意味要素が同時に存在し、重要な条件が相対的に埋没し得る問題です。後者は、ある段階の出力が次段の入力になり、その変化が系列に沿って累積し得る問題です。

この違いを、説明上「空間」と「時間」で整理します。

空間方向：

$$\{c_1, c_2, \ldots, c_N\} \longrightarrow P(y_t \mid c_1, \ldots, c_N)$$

同一時点 t のContextに、多数の意味要素が共存します。

時間方向：

$$x_0 \to y_1 \to y_2 \to \cdots \to y_T$$

前段の結果が後段の条件となります。

実務では両方が同時に起きます。長いAgent loopでは、時間が進むほど履歴が増え、その結果として空間方向のContext競合も増えることがあります。したがって「Contextを短くする」「途中成果物を外部へ固定する」「必要な時点で系列を切る」といった対策は、空間と時間の両方へ効きます。

この整理を持ったまま、次に「AIを複数にすればこの問題は消えるのか」を考えます。

補足：空間と時間は独立ではない

実務では両者は相互作用します。長いAgent loopでは、時間が進むほど履歴や成果物が増え、その結果として同一時点のContext競合も増えることがあります。逆に、一回のContext内で重要制約が埋没すると、その出力が次段の入力になり、時間方向のDriftへ接続します。

したがって「Contextを短くする」「元要件を外部へ固定する」「途中成果物で境界を作る」「必要な時点で系列を切る」といった対策は、空間と時間の両方へ効く場合があります。

13. Multi-Agentへの期待

複数のAIに役割を分ければ、互いの弱点を補えるように見える

- Architect / Implementer / Reviewer / Testerの分業には実務上の利点がある
- 専門化、並列化、観点分離はMulti-Agentの価値である

Multi-Agent Specialization Review Parallelism

Speaker notes

13. Multi-Agentへの期待

生成AIの不安定さが問題なら、複数Agentに役割を分けて相互チェックすればよい、という発想は非常に自然です。Architectが設計し、Implementerが実装し、Reviewerが検査し、Testerがテストを作る。人間の開発チームに似た分業構造です。

この発想自体を否定する必要はありません。役割を分けることでPromptやContextをタスクごとに短くできますし、異なるモデルを使えば得意分野を組み合わせられます。並列化によって速度を上げることもできます。Reviewerを読み取り専用にしてWrite権限を与えない、といった権限分離も有効です。

問題は「Multi-Agentにしただけで正しさが自動的に保証される」と考えることです。各Agentは依然として生成AIであり、それぞれの工程で意味解釈と生成を行います。

ここからは、Multi-Agentを否定するのではなく、「何が解決され、何が解決されないのか」を分けます。そのうえで、Multi-Agentの価値をAgent数そのものではなく、Orchestrationの設計へ置き直します。

14. しかし、Agentを増やしても生成原理は変わらない

Multi-Agent化しても、確率的な意味変換の連鎖であることは変わらない

- 各Agentが成果物を読み直して再解釈する
- Agent数の増加は決定性の増加ではなく、意味変換点の増加にもなる

Agent Chain | Probabilistic Transformation | Meaning Transformation

Speaker notes

14. しかし、Agentを増やしても生成原理は変わらない

Agent Aが要件を設計書へ変換し、その設計書をAgent Bがコードへ変換し、さらにAgent Cがレビュー結果へ変換する。各段階は、前ページまでと同じ条件付き生成です。

単純化すれば、

$$y_{t+1} \sim P_A(y_{t+1} \mid y_t), \qquad y_{t+2} \sim P_B(y_{t+2} \mid y_{t+1})$$

となります。AgentをAからBへ切り替えても、y_{t+1} に含まれた意味変化が自動的に元へ戻るわけではありません。Bは渡された成果物を現在の条件として解釈します。

したがって、Agent数を増やすことは決定性を上げることと同義ではありません。むしろ工程を細かく分けることで意味変換点が増えます。ただし、工程を短くして各段階を検証可能にするなら、その増えた境界を利用して制御できます。ここが重要です。

つまり、Multi-Agentの評価は「何体いるか」ではなく、「各境界で何を固定し、何を検証し、何を次へ渡すか」で考える必要があります。

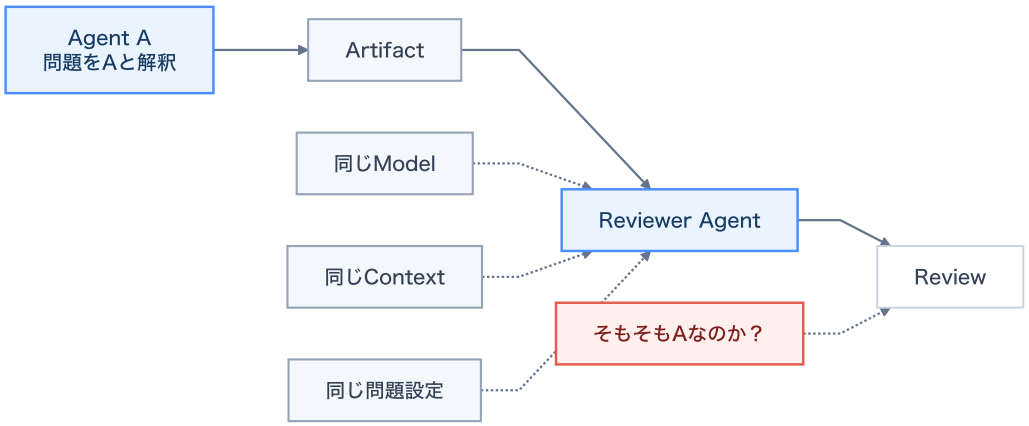
技術的な補足

ここで「確率的変換」と呼んでいるのは、各Agentが前段成果物を再解釈し、その解釈に基づいて新しい成果物を生成するという意味です。Agentを分けること自体が悪いのではありません。むしろ境界を増やすことで、各段階を短くし、検証点を置けるという利点もあります。

問題は、その境界を単なる受け渡し点にするか、意味保存を確認する制御点として使うかです。

15. Reviewer Agentも「独立した専門家」とは限らない

AIを増やすことと、独立した検証能力を増やすことは同じではない



- 同じ問題設定・Context・基盤モデルを共有すれば誤りも相関し得る
- 独立性を高めるには根拠、権限、Context、Engineを意識的に分離する

Reviewer | Correlation | Shared Context | Independent Verification

15. Reviewer Agentも「独立した専門家」とは限らない

人間のレビューでは、別の経験や知識を持つ専門家が異なる視点から問題を発見することを期待します。しかしAI Reviewerを追加しただけで、同じ意味の独立性が得られるとは限りません。

Agent Aの誤りを事象 E_A、Reviewerの誤りを E_R とします。もし完全に独立なら

$$P(E_A \cap E_R) = P(E_A)P(E_R)$$

ですが、同じ基盤モデル、同じ要件解釈、同じContext、類似したSystem Promptを共有すれば、誤りには相関が生じ得ます。その場合は一般に

$$P(E_A \cap E_R) \neq P(E_A)P(E_R)$$

です。別Engineを使うことは独立性を高める手段の一つですが、学習データや選好が近ければ、それでも失敗は相関し得ます。

特に危険なのは、Reviewerが前段の問題設定そのものを受け入れてしまう場合です。「この設計はAという要件に対して整合しているか」は確認しても、「そもそも元要件はAだったのか」まで遡らない可能性があります。

独立性を高めたいなら、Reviewerへ元要件を直接渡す、前段の長い会話履歴を渡さない、別Engineを使う、読み取り専用にする、外部テストやDBなど非生成系の根拠を使う、といった構造上の工夫が必要です。生成系同士の相互チェックより、生成系列の外側にある固定点との比較の方が効きます。

「独立性」はモデル名だけでは決まらない

別Engineを使えば独立性が自動的に保証されるわけでもありません。学習データ、問題設定、参照資料、評価基準、前段Artifactが共通なら、異なるモデルでも同じ誤りへ収束することがあります。

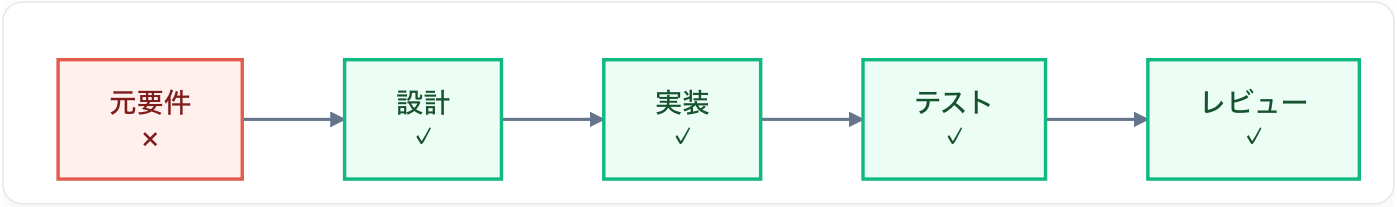
独立性を高めるには、モデルを変えること以上に、

- 元要件をReviewerへ直接渡す
- 前段の推論履歴を必要以上に共有しない
- Write権限を分離する
- テスト、型、スキーマ、DB制約など非生成系のOracleを使う

といった構造上の分離が重要です。

16. 一番怖いのは「間違ったまま綺麗に完成する」こと

内部整合性が高くても、元の意味が保存されているとは限らない



- 設計・実装・テスト・レビューが互いに整合していても元要件が逸脱している場合がある
- 局所整合性と意味保存を分けて評価する

Local Consistency | Meaning Preservation | Requirement Drift

Speaker notes

16. 一番怖いのは「間違ったまま綺麗に完成する」こと

Multi-Agentで最も見つけにくい失敗は、明らかに壊れたコードではありません。途中で生じた誤った前提を全Agentが共有し、その前提の内部では設計・実装・テスト・レビューが高い整合性を持ってしまう状態です。

例えば元要件が「20歳未満は購入不可」なのに、途中で「18歳以上なら購入可」と変わったとします。設計書も18歳、コードも18歳、テストも18歳、Reviewerも設計との一致を確認してOKを出せば、成果物群は非常に綺麗です。しかし元要件への忠実度は0です。

ここで区別したいのが局所整合性と意味保存です。局所整合性を $C(y_i, y_{i+1})$、元要件からの保存度を $S(x_0, y_i)$ と書けば、

$$C(y_i, y_{i+1}) \approx 1$$

であっても、

$$S(x_0, y_i) \ll 1$$

という状態はあり得ます。

したがって、後段成果物同士を比較するだけでは不十分です。元要件、外部仕様、テストOracleなど、生成系列の外側にある固定点と比較する必要があります。この固定点こそ、後半でいう外部制御の一部です。

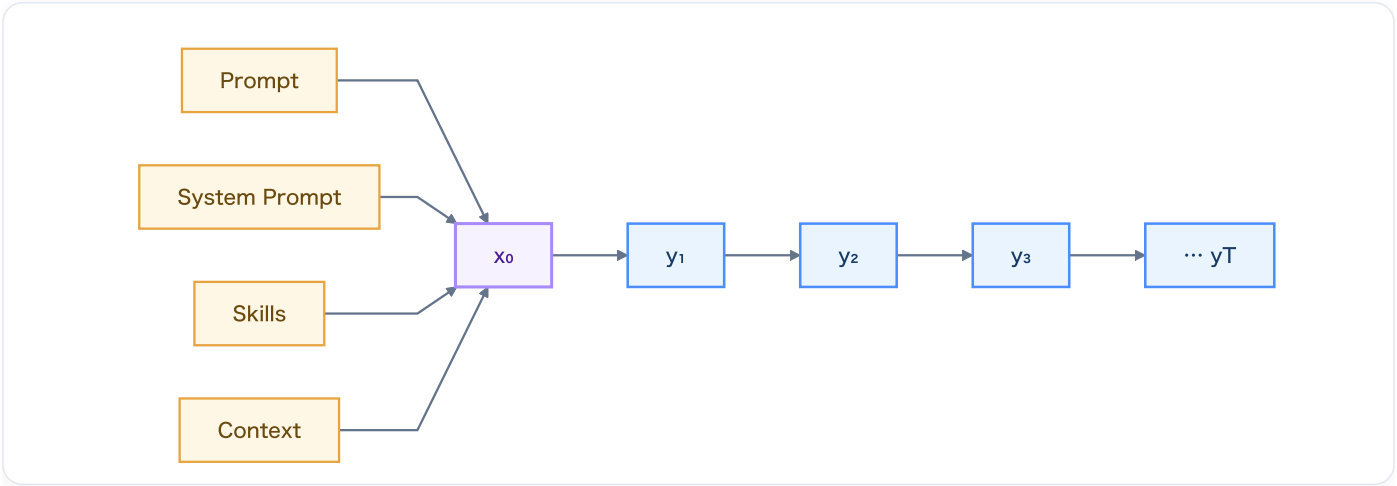
数式の位置づけ

$C(y_i, y_{i+1})$ や $S(x_0, y_i)$ は、このページの関係を説明するための抽象記号です。特定の標準指標を意味しているわけではありません。

重要なのは、隣接成果物同士がよく一致していることと、元要件が最終成果物まで保存されていることを別々に評価する、という考え方です。

17. Prompt Engineering が磨いているのは、主に x₀

Promptを改善することは重要だが、それはTrajectoryの出発点を改善している



- Prompt/System Prompt/Skills/Contextは初期条件 x₀ の品質を上げる
- Agentでは x₀ の後に判断・生成・Tool実行が続く

Prompt Engineering | Initial Condition | x₀ | Starting Point

Speaker notes

17. Prompt Engineering が磨いているのは、主に x_0

このページの要点

ここで最初の問いへ戻ります。Prompt Engineering、System Prompt、Skills、Context Engineeringはすべて有効です。ただし、本資料ではそれらの主な作用を

> **生成開始時点 x_0 の条件を良くする**

と整理します。

なお、機械学習分野で **Prompt Tuning** はsoft prompt等を学習する特定手法を指すことがあるため、このページではより広い意味を表す **Prompt Engineering** と表記します。

Promptが効く場所

Promptを明確にすれば、最初の条件付き分布

$$P(y_1 \mid x_0, \theta)$$

を目的に近い領域へ寄せやすくなります。曖昧さを減らす、制約を明記する、出力形式を固定する、参照情報を整理する、といった入力設計は、最初の生成自由度を実務上扱いやすい範囲へ狭める操作です。

しかしAgentは y_1 で終わらない

Agentでは概念的に、

$$x_0 \to y_1 \to y_2 \to \cdots \to y_T$$

と生成・判断・Tool実行が続きます。

実際には各段の間にTool結果、ファイル状態、テスト結果、権限状態などが入るため、より正確には状態 s_t を含めて

$$y_{t+1} \sim P(y_{t+1} \mid x_0, y_{1:t}, s_t, \theta)$$

のような依存関係になります。

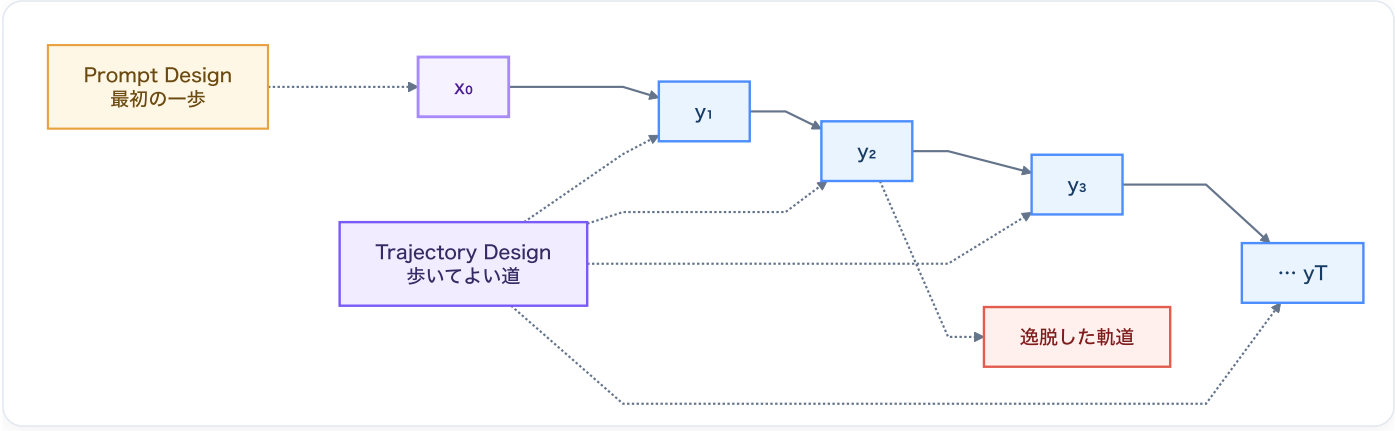
Promptを否定しているわけではない

ここでの主張は「Prompt Engineeringは不要」ということではありません。むしろ x_0 を良くする重要な設計です。

ただし、長いAgent実行では「出発点を良くする」だけでなく、「その後どこを通り、どこで止まり、何を次へ渡すか」という別の設計問題が残ります。これが次のTrajectory Designへの橋になります。

18. Agent時代には、Trajectory全体が設計対象になる

x₀だけではなく、確率的な生成過程が最終成果物へ到達するまでの経路全体を設計する



- PromptはTrajectoryの初期条件として残る
- 停止点・検証点・受け渡し情報・権限・復帰点まで設計対象になる
- Trajectoryを設計せず自律実行を続けるのは、確率的な生成過程を外部制御なしに歩かせる、いわば「バイアス付きランダムウォーク」に近い

Trajectory | Prompt Design | Generation Process | Agent Loop | Biased Random Walk | Control

Speaker notes

18. Agent時代には、Trajectory全体が設計対象になる

Trajectoryとは、ここでは生成AIが初期条件から最終成果物へ到達するまでに通る系列全体を指します。

単純化すれば、

$$\tau = (x_0, y_1, y_2, \ldots, y_T)$$

と置けます。

しかし実際のAgentでは、生成結果だけが状態ではありません。Tool実行結果、ファイル差分、テスト結果、Permission、Checkpoint、人間による判断なども次の生成条件へ影響します。

そこで状態列 s_t まで含めれば、

$$\tau = (x_0, s_0, y_1, s_1, y_2, s_2, \ldots, y_T, s_T)$$

のように考える方が実務に近くなります。

PromptはTrajectoryの一部である

Prompt Designは不要になるわけではありません。

むしろPrompt、System Prompt、Skills、Contextなどによって決まる x_0 は、Trajectoryの非常に重要な初期条件です。

したがって本資料では、

$$\text{Prompt Design} \subset \text{Trajectory Design}$$

と整理します。

Prompt Engineeringは『最初の一歩をどの方向へ踏み出すか』を強く決めます。

Trajectory Designは、その後についても、

- どこまで連続して生成させるか
- どこで一度止めるか
- 何をArtifactとして固定するか
- 次のAgentへ何を渡すか
- 何を渡さないか
- どこでValidationするか
- どこで人間が判断するか
- どの操作にWrite権限を与えるか
- 失敗したらどこまで戻すか
- どの状態から再開するか

まで含めて設計します。

つまりAgent時代の設計対象は、**『良いPromptを書く』から『良い経路を作る』へ拡張される**ということです。

Trajectoryを設計しない自律実行は、何に近いのか

ここで直感的な比喩を使います。

Trajectoryを設計せず、自律型Agentへ

> 『目的だけ渡すので、あとは自分で判断して最後まで進めてください』

と任せたとします。

Agentは各ステップで、その時点のContext、学習済みモデルの重み、Tool結果、現在の状態などをもとに次の判断を行います。

概念的には、現在状態を s_t とすれば、

$$s_{t+1} \sim P(s_{t+1} \mid s_t, x_0, \theta, \text{Tools})$$

のような、現在状態に条件付けられた遷移として見ることができます。

もちろん、これは数学でいう純粋なランダムウォークではありません。

AgentにはPromptという目的があり、学習済みパラメータによる強い偏りがあり、現在のContextにも依存しています。したがって各方向へ無作為に同じ確率で移動しているわけではありません。

むしろイメージとしては、**目的方向への強いバイアスを持った確率的な歩行**です。

そこで本資料では直感的に、

> **Trajectoryを設計せず、自律型Agentに処理を連続させるのは、確率的な生成過程を外部から制御せずに歩かせる――いわば『バイアス付きランダムウォーク』に近い。**

と表現します。

ここで重要なのは『ランダム』という単語そのものではありません。

重要なのは、**各ステップではもっともらしい局所判断をしていても、その判断系列全体が最終目的に対して正しい経路になる保証はない**という点です。

例えば、

```
x₀  元要件
 ↓
y₁  小さな解釈変化
 ↓
y₂  その解釈を前提に設計
 ↓
y₃  設計に整合する実装
 ↓
y₄  実装に整合するテスト
```

という系列を考えます。

y_1 の小さな意味変化は、その場では問題に見えないかもしれません。

しかし y_2 では、それが『入力された正しい前提』として扱われます。さらに y_3、y_4 と進むにつれて、その前提の上に局所的に整合した成果物が積み上がります。

その結果、前章で見た

> **『間違ったまま綺麗に完成する』**

という状態に到達することがあります。

局所的に正しいことと、経路全体が正しいことは別

Agentは通常、各時点では合理的に見える判断をします。

```
現在の状態
    ↓
次にもっともらしい行動
    ↓
新しい状態
    ↓
次にもっともらしい行動
```

という処理を繰り返します。

しかし、

> **各ステップの局所最適性 ≠ Trajectory全体の意味保存**

です。

途中で少し方向を間違えても、その後のAgentは『いま存在する状態』を正しいものとして次の判断を行う可能性があります。

この意味で、長時間Agentを動かすときに問題になるのは、一回一回の回答精度だけではありません。

どの状態を経由して最終結果へ到達したのかそのものが品質になります。

Trajectory Designは、確率性を消すことではない

ここも重要です。

Trajectoryを設計する目的は、生成AIを決定論的Softwareへ変えることではありません。

生成AIには、探索、設計、実装、推論といった確率的生成能力を大きく使わせます。

その一方で、

```
S₀
 ↓
S₁
 ↓
[Validation]
 ↓
S₂
 ↓
[Human Gate]
 ↓
S₃
 ↓
[Checkpoint]
 ↓
S₄
```

のように、外側から

- Stop
- Validation
- Gate
- Permission
- Checkpoint
- Rewind
- Artifact

を配置します。

つまりTrajectory Designとは、

> **確率過程そのものを消すのではなく、確率過程が通ってよい経路と、その途中の境界を設計すること**

です。

この考え方は、次ページの『Trajectoryを外から制御する』へそのままつながります。

Prompt DesignとTrajectory Designの違いを一言で言うと

直感的には、次のように整理できます。

> **Prompt Engineeringは『最初の一歩』を良くする。**
>
> **Trajectory Engineeringは『歩いてよい道』を設計する。**

もちろんPromptはTrajectoryの重要な一部です。

しかしAgentが長時間・多段階に動く時代には、初期条件だけでなく、その後の経路全体が工学的な設計対象になります。

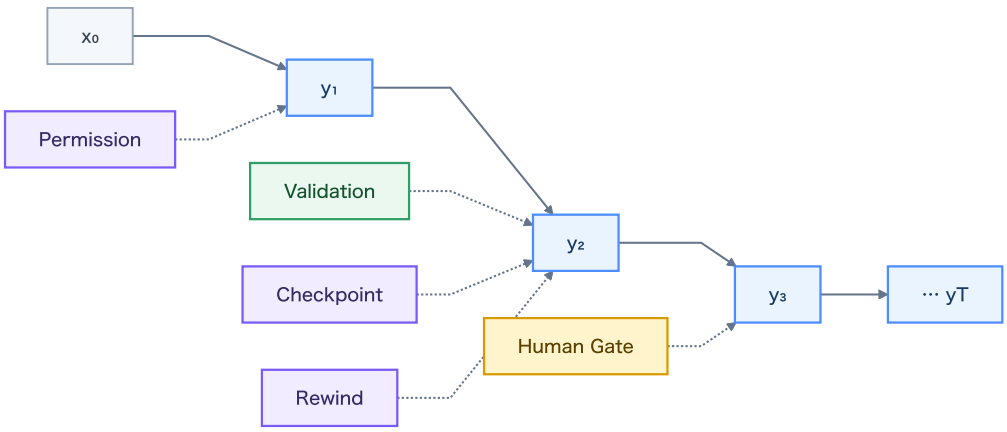
用語の位置づけ

Trajectoryという語は広く『経路・軌跡』を意味します。本資料では、初期条件から最終成果物までに通る生成、Tool結果、状態変化、Human Gate等を含む系列全体を、設計対象としてまとめて扱うために使っています。

したがって、単に会話Token列だけを指しているわけではありません。Coding Agentでは、ファイル差分やテスト結果、権限状態、CheckpointなどもTrajectoryの一部として扱う方が実務的です。

19. Trajectoryを「外」から制御する

生成系列全体の管理を、生成AI自身の自律ループだけに任せない



- Context、State、Permission、Validation、Checkpoint、Human Gateを外部制御として置く
- 生成AIの能力を弱めるのではなく、能力を使う範囲と復帰可能性を設計する

External Control | Checkpoint | Validation | Human Gate | Permission

Speaker notes

19. Trajectoryを「外」から制御する

Trajectoryを設計する際の中心は、生成AI自身に系列全体の管理を完全委任しないことです。AIに「自分で適切に止まり、自分で誤りを検知し、自分で元状態へ戻り、自分で権限を抑制してほしい」と頼むだけでは、制御規則そのものが再び確率的生成へ入ってしまいます。

そこで外部制御 u_t を導入するイメージを持ちます。状態 s_t の更新を

$$s_{t+1} = F(s_t, y_t, u_t)$$

と書いたとき、u_t はPermission、Validation、Human Gate、Checkpoint、Stop条件など、Harness側が決定的または明示的に与える制御です。

Semantic Driftの模式式

$$D_{t+1} = D_t + g_t(D_t) + \eta_t - \delta_t$$

で見れば、Validationや人間の修正は δ_t を意図的に挿入する仕組みと解釈できます。

ここで目的はAIの能力を下げることではありません。AIには設計・実装・探索を大きく任せます。ただし、不可逆な操作、長い連続生成、重要な意味決定の境界に、外部から観測・停止・復帰できる構造を置きます。

この「外」は一種類ではありません。次のページで、LLMの外とAgent Harnessの外を分けます。

Hard control と Soft control

外部制御にも強さの違いがあります。例えばOSやGitによる権限制約、Schema validation、テスト失敗、Read-only mountのような制御は、比較的決定的に止められます。一方、「AI Reviewerに確認させる」のようなValidationは、その検証自体が再び生成AIなら確率性を含みます。

したがって、重要な境界ほど非生成系の制約や人間判断と組み合わせることが重要です。「外に置けばすべて決定的になる」という意味ではありません。

20. 「外」は一つではない

商用生成AI製品の中でLLMを制御するHarnessと、そのさらに外側でHarness同士を制御するOrchestrationは、異なる制御層である

- LLMから見た外側はAgent Harnessであり、Context、Tools、Permission、Stateなどを制御する
- Harnessから見た外側はOrchestrationであり、複数Harness間のI/O、順番、Artifact、Gateを制御する
- Harnessは主に製品側が提供する制御層、Orchestrationは利用者・システム設計者が自ら構成しやすい制御領域である

LLM | Agent Harness | Orchestration | Control Layer | User-controlled

Speaker notes

20. 「外」は一つではない

ここで、第4章で導入した **Agent Harness** という考え方を回収します。

第4章では、私たちが普段利用している生成AI製品はLLMそのものではなく、LLMの周囲にPrompt / Context、Memory、RAG、Tools / Permission、Safety / Validation、Output Controlなどの制御機構が存在すると説明しました。

本資料では、この **LLMを包み、その入力・実行・出力を制御する外側の仕組み** をAgent Harnessと呼んでいます。

しかし、「外を設計する」と言ったときの外側は、Harnessで終わりではありません。

このページでは、外側を二つの階層に分けます。

---

1. LLMから見た外：Agent Harness

一つ目は、LLMから見た外側です。

```
LLM
 ↑
Agent Harness
```

Agent Harnessは、一つのLLMまたはCoding Agentの実行を包み、例えば次のようなものを制御します。

- Prompt / Context
- Memory
- RAG / Search
- Tools
- Permission
- File / Shell操作
- State
- Stop条件
- Validation
- Output Control

例えばClaude CodeやCursorでは、LLMが自分でOS権限を持ってファイルを書き換えているわけではありません。ファイル探索、編集、Shell実行、Git操作、Permission確認などは、LLMの外側にあるCoding Agent Harnessによって提供されています。

したがって、LLMを M、Harnessを H とすれば、一つのAgentを概念的に

$$A = H(M)$$

のように考えることができます。

以前は

$$A = H \circ M$$

のようにも表しましたが、実際のCoding AgentではHarnessとLLMが何度も相互作用するため、これは厳密な一回の関数合成を意味するものではありません。重要なのは、**LLM単体と、それを動かすHarnessを分けて考えること**です。

商用生成AI製品では、Harnessの多くは製品側にある

ここで制御主体にも注目します。

ChatGPT、Claude Code、Cursorなどを利用するとき、Harnessそのものの実装の多くは製品側から提供されます。

利用者は、Prompt、設定、Permission、Plan Modeなど、製品が公開している範囲を操作できます。しかし、Harness全体の実装を自由に変更できるわけではありません。

したがって概念的には、

```
┌──── 商用生成AI / Coding Agent製品 ────────┐
│                                          │
│   Agent Harness                          │
│      ↓                                   │
│    LLM                                   │
```

```
└──────────────────────────────────────┘
```

という一つの製品として私たちは利用しています。

後で見るClaude CodeやCursorのPlan、Rewind/Revert、Worktree(s)も、主としてこの **Coding Agent Harness内部でTrajectoryを制御する機能** として理解できます。

---

2. Harnessから見た外：Orchestration

しかし、複数のAgent Harnessを組み合わせると、別の問題が出てきます。

例えば、

```
Claude Code
    ↓
design.md
    ↓
Codex
    ↓
code
    ↓
Reviewer
```

という処理を行いたいとします。

Claude Code自身のHarnessは、Claude Code内部の実行を制御できます。Codex側のHarnessも、そのAgent内部の実行を制御できます。

しかし、

- Claude Codeをいつ呼ぶのか
- Claude Codeの何をCodexへ渡すのか
- design.mdを固定成果物にするのか
- Codexへ元要件も渡すのか

- どこで人間が確認するのか
- Reviewerは何を基準に検証するのか
- 失敗したらどこへ戻すのか

という **HarnessとHarnessの間の関係** は、個々のHarnessだけでは決まりません。

そこで必要になるのが、そのさらに外側にある **Orchestration** です。

```
Agent Harness A
       ↓
Artifact / Gate / I/O
       ↓
Agent Harness B
```

Orchestrationは、複数Harness間について、

- 呼び出し順序
- Artifact
- Contextの受け渡し
- Permission
- Validation
- Human Gate
- Stop / Resume
- Parallel / Sequential実行

などを制御します。

Orchestratorを O、各Agent Harnessを A_i とすれば、概念的には

$$O(A_1, A_2, \ldots, A_n)$$

と表せます。

ここでもこれは実装式ではなく、**一つのAgent内部ではなく、複数Agentの関係を一段外から制御する層が存在する**ことを表す模式です。

重要なのは「誰が制御できるか」

この二階層には、制御主体にも違いがあります。

LLMから見た外：Agent Harness

```
LLM
↑
Agent Harness
```

これは多くの場合、商用生成AI・Coding Agent製品の内部にあります。

もちろんPermission、Plan、Context、Tool設定など、利用者へ公開されている制御点はあります。しかしHarness全体の実装主体は基本的には製品側です。

Harnessから見た外：Orchestration

```
Agent Harness A
       ↓
      Gate
       ↓
Agent Harness B
```

こちらは、**私たち利用者・システム設計者自身が設計できる領域が大きい**という点が重要です。

どのAIを使うか、どの順番で呼ぶか、何をArtifactとして固定するか、どこで人間を挟むか、何を次のAgentへ渡さないか、といった構造を自分たちで決められます。

本資料で『中を賢くするな。外を設計しろ。』と言っているとき、特に実務で私たちが直接設計できるのは、このHarness間のOrchestration領域です。

ただし、これはHarness内部を制御できないという意味ではありません。製品が公開するPlan、Permission、Checkpointなどを利用することも重要です。違いは、**Harness内部は製品が提供する制御機構を利用することが中心であり、Harness間はシステム全体の構造そのものを自分たちで設計できる**という点です。

図の読み方

第4章の図では、次の構造を示しました。

```
User
 ↓
Agent Harness
 ↓
LLM
 ↓
Result
```

つまり、**LLMから見た外側**を説明しました。

このページでは、その図をさらに一段外へ広げています。

```
                ┌──── Orchestration ────┐
                │                       │
Agent Harness A ─┼→ Artifact / Gate ─→ Agent Harness B
                │                       │
                └───────────────────────┘

       Agent Harness内部では

       Plan / Tools / Permission / State
                      ↓
                     LLM
```

上側は **Harnessから見た外**、下側は **LLMから見た外** を表しています。

この二つを混同しないことが、これからのスライドを理解するポイントです。

この後のスライドへの接続

次にClaude CodeとCursorを見ます。

Claude Code / CursorのPlan、Rewind/Revert、Worktree(s)は、主として

> **一つのCoding Agent Harnessの中で、生成Trajectoryをどう制御するか**

という問題への機能です。

その後Multi-Agentへ進むと、問題は一段外へ移ります。

> **複数のHarnessのTrajectoryを、どう接続・停止・検証するか**

が設計対象になります。

multi-ai-cliは、この後者、つまり **Harnessから見た外側のOrchestrationを自分たちで設計する一例** として位置づけます。

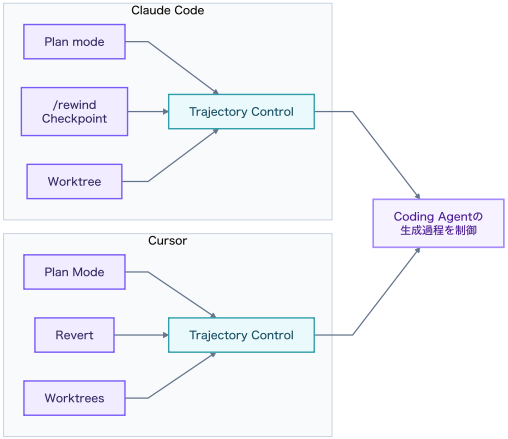
したがって後半の流れは、

```
LLM
 ↓
Agent Harness
 ↓
Coding AgentのTrajectory Control
 ↓
Harness間のOrchestration
 ↓
multi-ai-cli
```

というように、制御対象を一段ずつ外側へ広げていく構成になっています。

21. Claude Code / Cursor が実装している制御機能

現在のCoding Agentは、自律性を高めるだけでなく、生成過程を止め・戻し・隔離する機能を製品として持っている



- Claude Code: Plan mode / /rewind / native worktree
- Cursor: Plan Mode / Revert / Worktrees

Claude Code | Cursor | Plan | Rewind / Revert | Worktree

Speaker notes

21. Claude Code / Cursor が実装している制御機能

このページでは「事実」と「解釈」を分ける

ここではまず、Trajectoryという本資料の解釈を入れる前に、Coding Agent製品が実際に備えている制御機能を確認します。

Claude CodeにはPlan mode、過去状態へ戻るためのrewind/checkpoint系の機能、分離されたworktreeで作業する仕組みがあります。CursorにもPlan Mode、Revert、Worktreesがあります。

このページで言いたいのは、

> **強いCoding Agent製品は、自律性を高めるだけでなく、「編集前に止める」「過去状態へ戻る」「作業空間を隔離する」という制御機構も製品として持っている**

という事実です。

重要な限定

AnthropicやCursorが「Semantic Drift理論を理由にこれらの機能を実装した」と主張している、という意味ではありません。その因果関係は本資料からは言えません。

次ページで行うのは、

> 既に存在する製品機能を、Trajectoryという共通の設計視点で読むと、どのような制御原理として理解できるか

という筆者側の整理です。

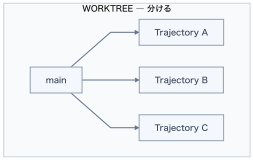
製品仕様について

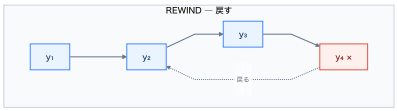
Coding Agent製品は更新頻度が高く、機能名、コマンド、UI、挙動は将来変わる可能性があります。この資料を後から読む場合は、実際に利用する時点の公式Documentationで最新仕様を確認してください。

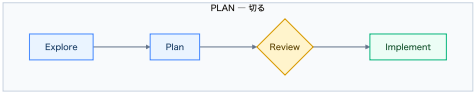
この資料のURL欄には関連する筆者記事を掲載しており、その記事では製品機能とTrajectory解釈の関係を詳しく整理しています。

22. なぜ Plan・Rewind・Worktree が重要なのか

異なる機能をTrajectoryの視点から見ると「切る・戻す・分ける」という共通した制御原理が見える







- Planは探索/計画と実装を切り、Writeの前に境界を置く
- Rewind/Revertは誤った軌道から既知状態へ戻し、Worktreeは並列軌道の状態を分ける

Plan Rewind Worktree 切る・戻す・分ける

Speaker notes

22. なぜ Plan・Rewind・Worktree が重要なのか

前ページは製品機能としての事実確認でした。このページからは、これらをTrajectoryという視点で解釈します。公式がこの理論で機能を実装した、という意味ではありません。前半で見たずれに対して、既存機能がどの制御になっているかを読む、という立場です。

Planは「切る」です。探索・要件解釈・設計方針の形成と、実際のファイル編集を同じ連続系列にしません。概念的には

$$\tau = \tau_{\text{plan}} \mid \tau_{\text{build}}$$

のように境界を置き、その境界で人間や明示的な承認を入れます。計画に誤りがあれば、Write副作用が起きる前に修正できます。

Rewind/Revertは「戻す」です。誤った状態 s_k から修正を積み重ね続けるのではなく、既知の状態 s_j $(j < k)$ へ復帰して別の分岐を試します。これは

$$s_k \to s_j \to s'_{j+1} \to \cdots$$

というTrajectoryの再分岐です。

Worktreeは「分ける」です。複数Agentや複数アプローチが同じWorking Treeへ同時に副作用を与えるのではなく、状態空間を分離します。Gitの観点では履歴を共有しながらWorking DirectoryとBranchを分けるため、並列Trajectory同士の直接干渉を減らせます。Worktreeは並列作業の隔離機能でもあります。同時に、ずれた軌道を本線へ混ぜない装置としても読めます。

前半のずれとの対応は次のとおりです。

- 計画と実装が同じ系列で進み、前提がずれたまま Write される → Plan で切る
- ずれた状態の上に修正を積み、Drift が残る → Rewind / Revert で戻す
- 複数の試みが同じ作業ツリーで混ざる → Worktree で分ける
- 成果物同士は整合しているが、元要件から離れている → 元要件を外に固定し、Gate で止める

この三つをまとめると、「切る・戻す・分ける」です。どれもモデル内部のAttentionや重みを変更する機能ではありません。生成系列の外側に構造を置くことで、強いAgentを扱いやすくしています。

三つは「正しさの保証」ではない

Plan、Rewind/Revert、Worktreeを使えば正しい結果が保証されるわけではありません。これらは、誤りを起こさない装置というより、

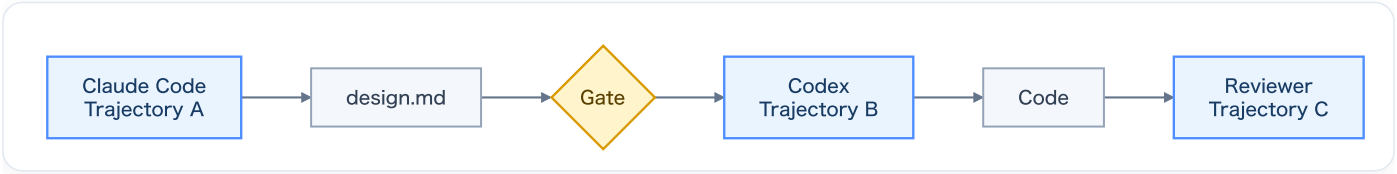
- 誤りがWriteへ進む前に止める
- 誤った状態から戻れるようにする
- 複数の試行を混ぜない

ための可逆性・隔離性・観測可能性を高める装置です。

この違いは重要です。生成AIそのものの確率性を消すのではなく、失敗したときに修正可能な構造へ置く、というのがTrajectory制御の中心です。

23. Multi-Agentでは、Harness間のTrajectoryを設計する

複数Agentでは、一段上からTrajectory同士の接続を制御する必要がある



- 各Harness内部の制御だけではAgent間のI/Oや順序は決まらない
- Artifact、Context、Permission、Gate、Stop/ResumeをHarness間で設計する

Harness-to-Harness | Artifact | I/O | Gate | Orchestration

Speaker notes

23. Multi-Agentでは、Harness間のTrajectoryを設計する

Claude CodeやCursorのような一つのCoding Agent Harnessの内部でPlan・Rewind・Worktreeを使っても、複数Harnessを組み合わせたときの「全体の順番」は別に設計する必要があります。

例えばClaude側で設計を作り、その design.md をCodexへ渡して実装し、そのコードをReviewerへ渡す場合、重要なのは各AIのモデル性能だけではありません。設計成果物の形式、どのContextを渡すか、Reviewerへ元要件を直接渡すか、実装Agentにどこまで Write権限を与えるか、どの地点で止めるかが全体品質を決めます。

Harness A_i のTrajectoryを τ_i とすると、Multi-Agent全体は単に

$$\tau_1 \to \tau_2 \to \tau_3$$

ではなく、その間に変換・検証・人間判断を含む接続写像 G_i を置いて

$$\tau_1 \xrightarrow{G_1} \tau_2 \xrightarrow{G_2} \tau_3$$

と考えると分かりやすくなります。G_i がArtifact形式、Validation、Human Gate、Context選択などです。

ここで主役はAgent数ではなくOrchestrationになります。どの順番で、何を渡し、何を許可し、どこで止めるかをHarnessの外側から決める層です。

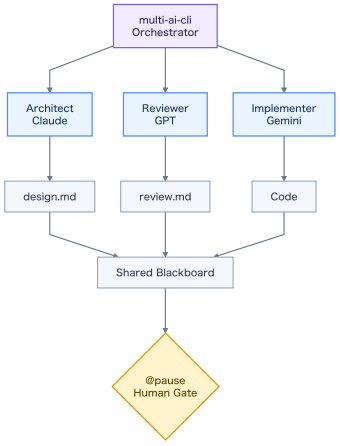
G_i は実装関数そのものではない

式中の G_i は、本資料の説明上、Harness間に置かれる接続規則をまとめて表した記号です。実装では一つの関数とは限らず、Artifact形式、Schema validation、Context選択、Permission、Human Gate、Stop/Resumeなどの組み合わせになります。

また、図のClaude Code / Codex / Reviewerという並びは、制御層を説明するための模式例です。製品分類や唯一の正しい組み合わせを示しているわけではありません。

24. multi-ai-cli は、Harnessのさらに外側を設計する

複数AIの生成軌道を、外側から観測・接続・停止する



- Agent/Engine分離、Shared Blackboard、Artifact relay、Human GateでHarness間を接続する
- Interactive REPLとstateless Filter modeを分け、状態遷移の範囲も制御する

multi-ai-cli | Shared Blackboard | @pause | Agent / Engine | Orchestration

Speaker notes

24. multi-ai-cli は、Harnessのさらに外側を設計する

ここでmulti-ai-cliの位置づけが明確になります。これは本資料の主張を実装した唯一の解ではなく、前ページの Orchestration 層の一例です。multi-ai-cliは「Claude CodeやCursorと同じ機能をもう一つ作る」ことを目的にしていません。Claude、GPT/Codex、Gemini、Grok、Localなど複数のEngineやAgentを、さらに外側から接続・観測する軽量なOrchestration層です。

現在の実装ではAgentとEngineを分離しています。論理的な役割を a、物理的な実行バックエンドを e とすると、

$$a \mapsto e$$

という対応を設定で変えられます。Architectという役割をClaudeへ割り当ててもGPTへ割り当てても、Workflow側の役割名を保てます。これは役割設計とモデル選択を分離するための構造です。

Agent間は長い共通会話履歴だけでつなぐのではなく、ローカルファイルをShared Blackboardとして利用します。

$$AI_A \rightarrow Artifact \rightarrow AI_B$$

という形にすることで、受け渡すContextを明示できます。依存がゼロになるわけではありませんが、どの成果物がどのAgentへ渡ったかを観測しやすくなります。

@sequence では逐次・並列WorkflowとArtifact relayを記述でき、@pause でHuman Gateを正常系として挿入できます。またFilter modeは stdin -> AI -> stdout のstatelessな一回実行に限定し、REPL専用の状態fulな操作を意図的に使えなくしています。これは機能を減らすことによって許容する状態遷移を狭くする設計でもあります。

したがってmulti-ai-cliの本質は、AIをさらに自由にすることではなく、複数HarnessのTrajectoryを外側から構造化し、人間が観測・修正可能な範囲へ置くことです。

multi-ai-cliの位置づけ

multi-ai-cliは本資料の主張を実装する唯一の方法ではありません。重要なのは個別ツールではなく、

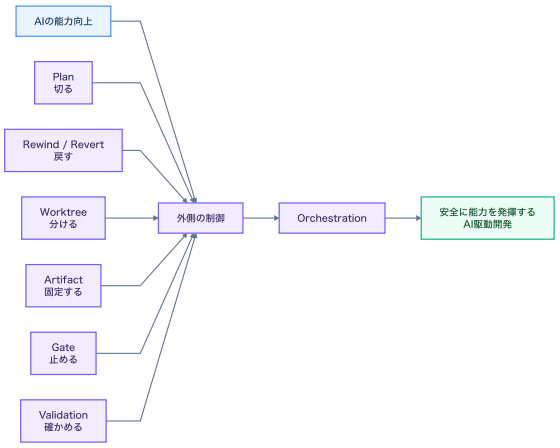
- 役割とEngineを分離する
- 長い共通履歴ではなくArtifactで接続する
- Human Gateを正常系として置く
- 状態を持つモードとstatelessなモードを分ける
- Harness間の順序とI/Oを外から見えるようにする

という設計原理です。

したがって、multi-ai-cliを使わなくても、同じ原理をCI/CD、Workflow Engine、GitHub Actions、独自Agent Harness等で実装することはできます。

まとめ. AIが強くなるほど、「外」が重要になる

能力が上がることと、自律権限を最大化することは別問題である



- Plan=切る、Rewind/Revert=戻す、Worktree=分ける
- Artifact=固定、Gate=停止、Validation=検証としてTrajectoryを外から扱う

Autonomy | Permission | Governance | Trajectory Control

Speaker notes

まとめ. AIが強くなるほど、「外」が重要になる

全体のまとめ

今回の主張は、AIを弱く使うことではありません。むしろ逆です。AIが長時間実行でき、Toolを使い、自分で計画し、並列作業までできるようになったからこそ、その能力をどの状態空間・どの権限・どのTrajectoryの中で使うかが重要になります。

最初に見た生成原理は

$$y \sim P(y \mid x, \theta)$$

でした。生成AIは、現在の文脈と学習済みパラメータから候補分布を形成し、次Tokenを逐次生成します。

この確率的な生成能力を、無理に

$$y = f(x)$$

のような決定的業務ロジックへ変える必要はありません。必要なのは、決定性が必要な部分をSoftware、Validation、Permission、State管理、Schema、Test、人間判断などへ分離することです。

前半から後半までの一本の流れ

この資料の論理は次のようにつながっています。

```
生成AIは候補分布から逐次生成する
        ↓
「続きやすい」と「正しい」は別
        ↓
Hallucination
        ↓
Contextを増やすだけでも解決しない
        ↓
同一時点の競合 = Crowding（空間）
        ↓
```

```
出力が次の条件になる = Drift（時間）
        ↓
Multi-Agentにしても生成原理は消えない
        ↓
Promptで x₀ を整える
        ↓
Trajectory全体を設計する
        ↓
切る・戻す・分ける・固定する・止める・確かめる
        ↓
Harnessの外をOrchestrationする
```

Claude Code / Cursorとの接続

Claude Code / Cursorが実装しているPlan、Rewind/Revert、Worktree(s)は、本資料のTrajectoryという視点から見ると、

- Plan = 切る
- Rewind / Revert = 戻す
- Worktree = 分ける

という制御として読むことができます。

これは公式が同じ理論を掲げているという意味ではなく、既存の製品機能を共通の設計原理で整理した筆者の解釈です。

複数Harnessを組み合わせる場合は、その一段外側でArtifact、I/O、Gate、Permission、順序、Stop/Resumeを設計する必要があります。multi-ai-cliはそのOrchestration層の一例です。

実務への持ち帰り

1. 元要件を生成系列の外に固定し、後段Agentに勝手に上書きさせない
2. 長い会話履歴だけで接続せず、短いArtifactで受け渡す
3. Reviewerには前段の解釈だけでなく元要件を直接渡す
4. Writeの前にPlan / Gateを置く
5. ずれたら修正を積み増すだけでなく、既知状態へRewindする
6. 並列作業はWorktree等で状態を分離する
7. 検証は生成AI同士だけでなく、Test / Schema / DB制約 / 人間も使う

最後の一文

「中を賢くするな。外を設計しろ。」という言葉は、モデル研究や能力向上を否定するものではありません。

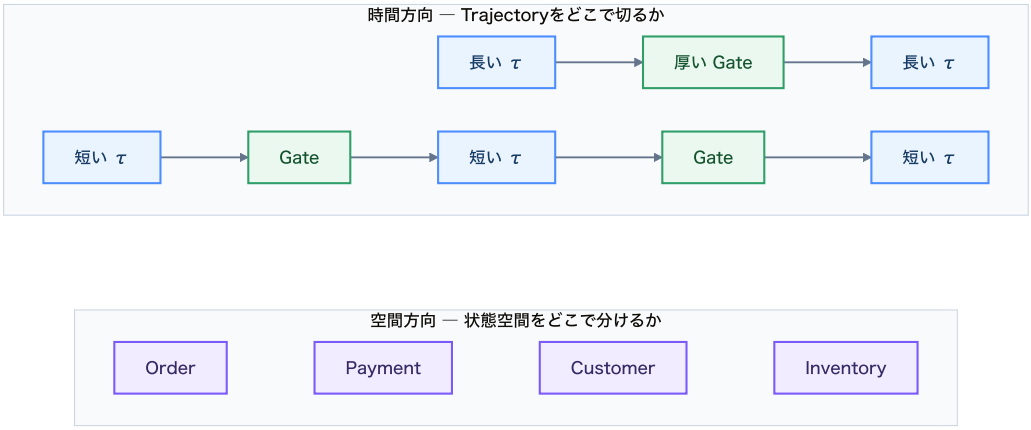
モデルを賢くすることと、その能力を安全・再現可能・観測可能な業務構造へ組み込むことは、別の工学問題です。

確率的なものは、確率的なものとして使う。

- 生成AIは生成する
- 人間は目的と意味を決める
- 従来型ソフトウェアは状態と制約を守る
- Orchestrationは、その境界と経路を設計する

中を賢くするな。外を設計しろ。

WaterfallかAgileかより前に、生成AIをどの単位で走らせ、どこでTrajectoryを切るかを決める



- 時間方向では、一つのTrajectoryをどこまで連続させてGateを置くかを設計する
- 空間方向では、Domain / Subdomainごとに状態空間とWrite範囲をどこまで分けるかを設計する
- 大きく切ればGateは少なく厚く、小さく切ればGateは増えるがRewind範囲を小さくできる

Trajectory Granularity | Waterfall | Agile | Domain Boundary | Gate | Rewind

Speaker notes

付録1. 大規模開発では、Trajectoryの粒度を設計する

ここからは付録です。

本編では、一つのAgent実行だけでなく、複数Agent / Harnessを含む生成系列全体をTrajectoryとして設計する、という考え方を説明しました。

この考え方を大規模ソフトウェア開発へ持ち込むと、次の問いが出てきます。

> **大規模開発では、Trajectoryをどの粒度で切るべきか。**

一見すると、これはWaterfallかAgileかという開発方法論の話に見えます。

しかし生成AIの側から見ると、どちらの場合も生成原理そのものは変わりません。

生成AIは各地点で、現在のContextと状態から次の判断・生成を行います。

変わるのは主に、

- どこまでを一つのTrajectoryとして連続生成させるか
- どこでGateを置くか
- 何をArtifactとして外部へ固定するか
- どこまでを同じ状態空間として扱うか

です。

時間方向の粒度

Waterfall型の大きな工程を単純化すると、例えば、

```
要件定義
    ↓
設計
```

```
    ↓
詳細設計
    ↓
実装
    ↓
テスト
    ↓
結合
```

となります。

Trajectoryの観点では、それぞれの工程内部に比較的長い τ_i があり、工程境界にGate G_i が置かれる構造として見ることができます。

```
τ₁ — G₁ — τ₂ — G₂ — τ₃ — G₃ — τ₄
```

ここで τ_i は各工程内部の生成・判断・Tool実行の系列、G_i はReview、Approval、Validation、Artifact固定などの工程境界です。

この構造の利点は、意味の確定やWrite権限の切り替えを、明確な工程境界へ寄せやすいことです。

一方で、Gateへ到達するまでのTrajectoryが長くなるほど、その内部でSemantic Driftが進む余地も大きくなります。

特に危険なのは、本編で説明した

> **『間違ったまま綺麗に完成する』**

という状態です。

設計、実装、テストが互いには整合していても、元要件から離れている可能性があります。

そのため工程レビューでは、直前成果物だけを見るのではなく、必要に応じて生成系列の外側に固定した元要件へ直接戻って比較する必要があります。

小さな単位で切る場合

一方、Feature、Use Case、Subdomainなどの小さな単位でTrajectoryを回す構造では、

```
Order       τ₁ → Gate
Payment     τ₂ → Gate
Customer    τ₃ → Gate
Inventory   τ₄ → Gate
```

のようになります。

一つ一つの τ_i は短くなります。

そのため、ずれた場合に戻る範囲を小さくしやすく、変更のBlast Radiusも限定しやすくなります。

これはWorktreeを使って並列Trajectoryを分離する考え方の、より大きな組織・Architecture版と見ることもできます。

しかし、小さく分ければ自動的に安全になるわけではありません。

分割数が増えると、

```
τ₁ → G₁ → τ₂ → G₂ → τ₃ → G₃ → ...
```

のように、接続点 G_i の数が増えます。

つまり、

> **Trajectoryを短くするとDrift距離は短くできるが、意味の接続点は増える。**

というトレードオフがあります。

これはMulti-AgentでAgent数を増やしたときと同じ構造です。

Agentを増やすと一つのAgentが扱うContextを小さくできますが、Artifactの受け渡しと意味変換点が増えます。

Domainを細かく分ける場合も同様で、各Domain内部が正しくても、境界の契約や接続写像が曖昧なら、

> **局所的には正しいが、システム全体では違う**

という状態が発生します。

WaterfallとAgileをTrajectoryとして見る

ここではWaterfallとAgileの方法論全体を同一視しているわけではありません。

Trajectoryという観点だけを取り出して単純化すると、典型的には、

- Waterfall：比較的長い τ と、少数の厚いGate
- Agile：比較的短い τ と、頻度の高い小さなGate

として見ることができます。

したがって生成AI導入の本丸は、単純に

> 『WaterfallとAgileのどちらがAIに向いているか』

ではありません。

むしろ、

> **どの長さのTrajectoryを許し、どこで外部固定点と照合するか**

という粒度設計です。

二つの粒度

大規模AI開発では、少なくとも二つの粒度を考える必要があります。

時間方向

一つのTrajectoryをどこまで連続させるか。

```
短い τ  ←――――――→  長い τ
```

空間方向

一つのTrajectoryへどこまでの意味・State・Write権限を持たせるか。

```
Component
   ↓
Feature
```

```
   ↓
Subdomain
   ↓
Domain
   ↓
System
```

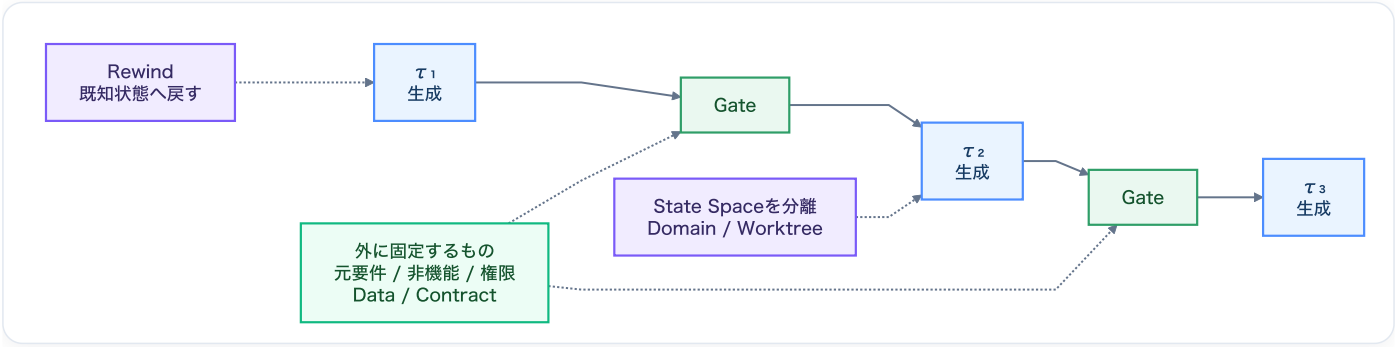
したがって大規模開発における生成AI利用は、

> **時間方向にTrajectoryをどこで切るかと、空間方向に状態空間をどこで分けるかを同時に設計する問題**

として見ることができます。

付録2. WFかAgileかより先に決めること

開発方法論を決める前に、何を生成系列の外へ固定し、AIにどこまで生成させるかを決める



- 元要件・業務ルール・非機能要件・権限・データ制約を生成系列の外に固定する
- Gateは直前成果物だけでなく、必要に応じて外部固定点と比較する
- Rewind単位と並列可能な状態空間をあらかじめ設計する

Invariant Artifact Gate Rewind State Space Orchestration

Speaker notes

付録2. WFかAgileかより先に決めること

大規模開発へ生成AIを導入するとき、最初に決めるべきことを『WaterfallかAgileか』だけに置くと、本質を見失う可能性があります。

生成AIから見れば、どちらの工程モデルでも各地点では確率的な生成と状態遷移が起きています。

したがって、開発方法論を選ぶ前に、AIが動く状態空間とTrajectoryの境界を設計する必要があります。

本資料の考え方から整理すると、少なくとも次の六点を先に決めておくと分かりやすくなります。

---

1. 何を生成系列の外に固定するか

最も重要な問いです。

例えば、

- 元要件
- 業務ルール
- Domain Boundary
- 非機能要件
- Security Policy
- Permission
- Data Constraint
- API Contract
- Schema

などです。

これらをAgentが生成する成果物の一つとしてだけ扱うと、Trajectory内部で書き換えられる可能性があります。

重要な不変条件は、生成系列の外側に置き、後段Agentが勝手に変更できない固定点として扱う方が安全です。

---

2. AIに生成させてよい単位はどこまでか

一つのAgentへ、

> 『要件から実装とテストまで全部やって』

と任せるのか、

```
Feature
Use Case
Domain Slice
Component
```

などの小さな単位へ分割するのかを決めます。

これは一つのTrajectory τ の長さを決める問題です。

長くすれば受け渡し回数は減りますが、内部Driftを観測する地点も減ります。

短くすれば途中で確認しやすくなりますが、Agent間・工程間の意味変換点が増えます。

---

3. どこでTrajectoryを切るか

切る場所には、例えば、

- Artifact完成時
- Design Review
- Commit
- Test成功時
- Pull Request
- Human Approval
- Release Boundary

などがあります。

本編で見たPlanと同じで、重要な副作用が発生する前に境界を置くことが有効です。

特にWrite権限、Database変更、外部API実行など、戻すコストが高い操作ほど、手前にGateを置く意味が大きくなります。

---

4. Gateは何を見るのか

Gateを置くだけでは不十分です。

重要なのは、Gateが何と何を比較するかです。

例えば、

```
設計 → 実装 → テスト
```

という系列で、実装を設計とだけ比較し、テストを実装とだけ比較すると、全体が局所的には整合していても元要件から離れている可能性があります。

そこで、

```
元要件 ──────────────────┐
                         ↓
設計 → 実装 → テスト → Gate
```

のように、必要に応じてGateから生成系列の外側に固定された元要件へ直接戻ります。

これは本編で説明した

> **内部整合性 ≠ 意味保存**

を大規模工程へ適用したものです。

---

5. Rewindの単位は何か

失敗したときに、どこへ戻るのかも先に決めます。

例えば、

- 一つの生成
- Commit
- Pull Request
- Artifact
- Feature
- Sprint
- 工程
- Subdomain

などです。

戻る単位が大きすぎると損失が大きくなり、小さすぎると誤った前提そのものが残ることがあります。

したがってRewindは単なるUndoではなく、

> **どの状態までを『既知の正しい状態』として扱うか**

という設計です。

---

6. 何を並列化してよいか

AIを並列に動かす場合は、同じ状態へ複数Agentが同時に副作用を与えないようにします。

検討すべき境界は、

- Worktree
- Repository
- Module
- Domain
- Database
- API Contract
- Write Permission

などです。

本編のWorktreeで説明した『分ける』を、大規模開発ではArchitectureやTeam Boundaryへ拡張して考えます。

大規模AI開発とは何をすることか

ここまでをまとめると、大規模開発で生成AIを利用することは、単に

> 『全工程をAIに任せる』

ことではありません。

むしろ、

1. Domainごとに状態空間を分ける
2. 各Trajectoryの入口条件を固定する
3. AIが生成してよい範囲を限定する
4. 出口でArtifactを固定する
5. Gateで外部不変条件と比較する
6. 必要ならRewindする
7. 次のTrajectoryへ必要なContextだけを渡す

というOrchestrationの問題です。

本編で使った言葉へ戻せば、

> **切る・戻す・分ける・固定する・止める・確かめる**

を、今度は一つのCoding Agentではなく、工程、Domain、Team、Repositoryといった大規模開発の単位へ載せ替えることになります。

WaterfallかAgileか、その前に

したがって、先に決めるべきなのは開発方法論そのものではありません。

まず、

> **何を生成系列の外へ固定するのか。**

そして、

> **どこまでを一つのTrajectoryとしてAIへ許すのか。**

を決めます。

この二つが決まれば、その構造をWaterfall型の工程へ載せることも、Agile型の短いIterationへ載せることもできます。

生成AI時代の大規模開発では、工程モデルそのものより一段下にある、**Trajectoryの粒度と境界の設計**が重要になります。

これは本編の

> **『中を賢くするな。外を設計しろ。』**

を、大規模開発へ拡張したものです。

あとがき

今回の考え方は、突然できたものではない

- 生成AIを確率的な生成器として捉える視点は、2025年3月頃から検討していた
- Hallucination、Semantic Drift、Multi-Agent、Harness、Orchestrationを整理する中で現在の形へ収束した

Afterword Trajectory Semantic Drift Harness Orchestration

Speaker notes

あとがき

生成AIを単なるコード生成器としてではなく、**確率的な振る舞いを持つ生成器として捉え、その外側に人間、Software、検証構造を置く**という方向性自体は、2025年3月頃から検討していました。

当時はまだ、現在のように **Trajectory** という言葉で整理していたわけではありません。

しかし、その後、

- Hallucination
- Context
- Semantic Drift
- Multi-Agent
- Meaning Preservation
- Agent Harness
- Orchestration

と個別の問題を整理していく中で、少しずつ共通した構造が見えてきました。

それは、生成AI内部の確率性そのものを消そうとするのではなく、

> **生成AIがどこまで自由に生成してよいのか、その外側の構造を設計する**

という考え方です。

振り返ると、2025年3月の時点では、まだ『Hallucinationをどう抑えるか』『AIをどこまで任せるか』という問題意識として現れていたものが、その後Semantic DriftやMulti-Agentの連鎖を考える中で、

```
Prompt
  ↓
Generation
  ↓
Trajectory
  ↓
Harness
  ↓
Orchestration
```

という階層へ整理されていったとも言えます。

今回の資料は、その約一年半の考察を一度まとめたものです。

そして、ここで完成というわけでもありません。

付録で触れたように、大規模開発へ適用すると、次には

> **Trajectoryをどの粒度で切るべきか**

という問題が出てきます。

WaterfallかAgileかという工程モデルの選択より前に、何を生成系列の外へ固定し、どこまでを一つのTrajectoryとしてAIへ許すのかを設計する必要があります。

つまり議論は、PromptからTrajectoryへ、TrajectoryからHarnessへ、HarnessからOrchestrationへ、さらに大規模開発のArchitectureへと続いていきます。

最後に、本資料の中心に置いた言葉へ戻ります。

> **確率的なものは、確率的なものとして使う。**
>
> **中を賢くするな。外を設計しろ。**