IT資格道場

Databricks Certified Generative AI Engineer Associate 学習用テキスト

予備知識ゼロから合格ラインへ —— 読んで理解する1冊目

SAMPLE

無料サンプル(第1章まで)

Databricks Certified Generative AI Engineer Associate 出題範囲(Exam Guide・2026 年 3 月 18 日版)準拠
2026年11月版

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

SAMPLE

目次

この試験の範囲 [overview]

第1章 Design Applications [1.1/1.2/1.3/1.4/1.5/1.6]

- 出力の形を指定するプロンプト [1.1]
- 構造化出力（response_format）とプロンプトだけの縛りの違い [1.1]
- システムプロンプトと few-shot の例 [1.1]
- 業務の要件を NLP のタスクに当てはめる [1.2]
- 要件を複数のタスクに割る場面と、LLM 以外で足りる場面 [1.2]
- 埋め込み・再ランキング・チャットなどモデルの種類と用途 [1.2]
- チェーンの部品と並び [1.3]
- 入力と出力の形から部品を選ぶ [1.3]
- 業務の目標を入力と出力に書き下す [1.4]
- RAG・エージェント・単純なプロンプトのどれで解くか [1.4]
- 待ち時間・費用・正確さ・説明責任を設計に写す [1.4]
- ツール呼び出し（関数呼び出し）の仕組みとツールの定義 [1.5]
- 多段の推論でツールを並べる順と ReAct の型 [1.5]
- Unity Catalog の関数をエージェントのツールにする [1.5]
- 読むだけのツールと、書き込む・外へ送るツールの分け方 [1.5]
- Agent Bricks の全体像と、名前の変遷 [1.6]
- Knowledge Assistant の使い所と材料 [1.6]
- Multiagent Supervisor の使い所 [1.6]
- Information Extraction の使い所 [1.6]
- Agent Bricks と自前のコードの選び分け、質の上げ方 [1.6]

サンプルはここまでです

SAMPLE

3 冊の使い方は、ダウンロード欄の案内を参照してください。

SAMPLE

この試験の範囲 [overview]

Databricks Certified Generative AI Engineer Associate は、Databricks の上で、生成 AI アプリを設計し、作り、配り、見張る作業を問う試験です。

- 試験コード: 無し(名前で呼ぶ)
- 出題: 採点される 45 問の選択式(択一と複数選択)
- 試験時間: 90 分
- 言語: 日本語でも受験可

出題範囲は 6 つの分野に分かれます。この教材も、同じ順に章を立てます(下の表)。配点は、Exam Guide(2026 年 3 月 18 日公開版)の割合です。

分野	配点
1 アプリケーションの設計(Design Applications)	14%
2 データの準備(Data Preparation)	14%
3 アプリケーションの開発(Application Development)	30%
4 アプリケーションの組み立てと配備(Assembling and Deploying Applications)	22%
5 ガバナンス(Governance)	8%
6 評価とモニタリング(Evaluation and Monitoring)	12%

- Exam Guide の細目: 56 個

SAMPLE

- 章と節の見出しの後ろにある [2.1] のような印: その細目の番号
- 名前の変更: Vector Search は 2026 年 6 月に AI Search に改称。AI Gateway は、今の文書では Unity Gateway の名前(本文は Exam Guide の名前を主に、今の名前を括弧で併記)

SAMPLE

第1章 Design Applications [1.1/1.2/1.3/1.4/1.5/1.6]

アプリケーションの設計は、コードを書く前に決める章です。何を入れて何を返すか、どのモデルのどの機能を使うか、ツールをどの順に呼ぶか、Agent Bricks に任せるか自前で書くか。この 4 つの判断が、後ろの章の実装と評価をすべて決めます。試験では、業務の場面が 3〜4 行で示され、並んだ設計案から合うものを選ぶ形（1 つを選ぶ問と、複数を選ぶ問）で出ます。用語を当てはめるだけでは解けません。場面の条件（データの種類・待ち時間・費用・権限）を読んで、設計を決める力が問われます。

出力の形を指定するプロンプト [1.1]

LLM は何も指定しなければ、自由な文章を返します。後続の処理が JSON を読む、画面が表を描く、といった使い道があるなら、形をプロンプトで縛る必要があります。形を縛る手は 4 つあります。

- 指示: 「JSON だけを返す」「Markdown の表で返す」「3 行の箇条書きで答える」と、形を文で言い切る
- 出力例: 入力と期待する出力を 1 組以上見せる（few-shot）
- 区切り記号: ユーザーの文章や取得した文書を `<document>` `</document>` や `"""` で囲み、指示とデータを混ぜない
- キーの一覧: JSON のキー名と型、取りうる値（enum）を全部書く

実際のプロンプトは次のようになります。区切り記号で入力を囲み、出力のキーを固定し、余計な文を禁じます。

SAMPLE

```
あなたは商品レビューの分類係です。
<review> と </review> の間の文章から、次の 2 つのキーだけを持つ JSON を返してください。
- product: 商品名（文字列）
- sentiment: "positive" "neutral" "negative" のどれか
JSON 以外の文字（前置きや説明）は出力しないでください。

<review>
SmartHome Mini は小さくて使いやすい。黒を買いました。
</review>
```

この書き方で崩れやすい場所は 3 つあります。モデルが「はい、JSON は次のとおりです。」と前置きを付ける場合。JSON を ```json の囲みで返す場合。キー名を勝手に変える場合です。いずれも、受け取る側で JSON として読むとエラーになります。LangChain の `JsonOutputParser` で試すと、囲み付きの JSON は読めますが、前置き付きの文字列は `OutputParserException` になります。プロンプトで縛る方法は「頼む」だけなので、守られる保証はありません。温度（temperature）を下げると揺れは減りますが、ゼロにはなりません。

LLM の出力を受けた後に、形式・文体・内容が要件に合っているかを確かめる後処理やガードレールを置く設計も使われます。JSON、HTML、Markdown などの決まったファイル形式が要る場合は、後処理が出番です。構造化出力が必要なら、Instructor や Outlines のようなライブラリが、検証の段を作る出発点になります。[1]

SAMPLE

構造化出力（response_format）とプロンプトだけの縛りの違い [1.1]

構造化出力は、JSON のスキーマをリクエストに渡して、モデルの出力をそのスキーマに合わせさせる機能です。Databricks では、チャットのリクエストの `response_format` に指定します。提供元のモデルごとに、提供元固有の構造化出力の形式を使う必要はありません。使いどころは 3 つです。大量の文書からの情報抽出（レビューを肯定・中立・否定に分類するなど）、出力形式が決まったバッチ推論、非構造データから構造データへの変換です。[2]

SAMPLE

```
response_format = {
    "type": "json_schema",
    "json_schema": {
        "name": "research_paper_extraction",
        "schema": {
            "type": "object",
            "properties": {
                "title": {"type": "string"},
                "authors": {"type": "array", "items": {"type":
"string"}},
                "keywords": {"type": "array", "items": {"type":
"string"}},
            },
        },
        "strict": True,
    },
}
response = client.chat.completions.create(
    model="databricks-gpt-oss-20b",
    messages=messages,
    response_format=response_format,
)
```

`type` には `json_schema`（スキーマに従う JSON）のほか、スキーマを渡さない `json_object` と、普通の文章があります。スキーマが事前に決まらない抽出では `json_object` を使います。ただし後述のとおり、Claude のモデルは `json_schema` しか受け付けません。

SAMPLE

スキーマに書ける内容には制限があります。Foundation Model APIs が受け付けるのは JSON スキーマの一部だけで、単純なスキーマほど生成の質が上がります。

制限	内容
使えないキー	`pattern`（正規表現）、`anyOf` `oneOf` `allOf` `prefixItems` `$ref`（入れ子や合成）
型の並び	`[型, "null"]` の 1 通りだけ。それ以外の型の並びは使えない
キー数の上限	スキーマに書くキーは最大 64 個
長さの制約	`maxProperties` `minProperties` `maxLength` などは強制されない
深い入れ子	質が下がる。できるだけ平らにする

トークンの消費にも注意が必要です。構造化出力の質を上げるためにプロンプトへの追記などが行われるので、入力と出力のトークンが増え、料金に反映されます。[2]

Claude のモデルには追加の制約があります。

- 使える `type` は `json_schema` だけ。`json_object` は使えない。制約なしの出力が要るなら `response_format` を省く
- ストリーミングと併用できない。`response_format` を付けるときは `stream` を false にする
- `tools` や `tool_choice` と組み合わせられない
- Claude Fable 5.2 は、Chat Completions と Responses のどちらでも構造化出力に対応しない。`response_format`（Responses なら `text.format`

SAMPLE

の JSON スキーマ）を付けると HTTP 400 になる。代わりに関数呼び出しを自動選択で使う

- databricks-claude-opus-5-5、sonnet-5-5、haiku-5-5、fable-5-1 は、Chat Completions の `response_format` を付けると HTTP 400 になる。Opus 5.5・Sonnet 5.5・Haiku 5.5 は、Anthropic Messages API の `output_config` の `format` を使う
- databricks-claude-opus-5 は、モデルが構造化出力の要求を断ると `finish_reason` が `content_filter` の応答を返すことがある。その出力はスキーマに合わない場合があるので、読む前に `finish_reason` を確かめる

プロンプトだけで形を縛る方法と比べると、違いは保証の強さです。プロンプトの指示は文章での頼みごとで、モデルが前置きを付けたりキーを変えたりしても、リクエストは成功します。構造化出力はスキーマを渡すので、出力の形がスキーマに沿うように作られます。それでも、文字数の上限や正規表現のような細かい制約までは強制されません。受け取る側で値を検証する処理は省けません。たとえば Python の `jsonschema` の `validate` を使うと、必須キーが無い JSON は「`'title' is a required property`」のようなエラーで見つかります。

場面で選ぶなら、下流のコードが JSON を直接読む、バッチで何万件も処理して崩れた行を許せない、という条件なら構造化出力です。人が読む文章の口調と構成を整えたいだけなら、システムプロンプトと出力例で足ります。[2]

場面で考えます。10 万件のレビューから `product` と `sentiment` を JSON で取り出し、後続の表へ取り込みます。1 件でも崩れた行があると取り込みが止まるので、プロンプトの指示だけでは足りません。`response_format` に `json_schema` を渡し、キーを少なく平らにします。同じ呼び出しで `tools` も使

SAMPLE

いたいときに Claude のモデルを選ぶと、`response_format` と `tools` は併用できません。抽出の呼び出しと、ツールを使う呼び出しを分けます。

システムプロンプトと few-shot の例 [1.1]

メッセージには役割（role）があります。`system` は、役割・口調・守るルール・出力形式のような、全ターンに効かせたい指示を置く場所です。`user` は、その回の質問や取得した文脈を置く場所です。形式の指示を `system` に入れておけば、ユーザーの文面が変わっても形は崩れません。

few-shot は、プロンプトの中に「入力と理想の出力」の組を数個並べる方法です。モデルは例から形式・文体・内容の水準を読み取ります。例は想定する質問の種類を広く覆うように選びます。チェーンが苦手とする質問の種類を見つけて、その理想の答えを作り、例として入れる方法が使われます。例が実際の質問と似ていないと、効果は出ません。

プロンプトのテンプレートは、取得した情報に加えて日付やユーザーの属性のような変数を差し込める形に作ります。複雑な質問には、「順を追って考えて」と促す Chain-of-Thought（CoT）で、手順を分けた推論を引き出せます。あるモデルで良かったプロンプトは別のモデルで同じ効果が出るとは限りません。モデルを替えたら、書式と長さを試し直します。[3]

業務の要件を NLP のタスクに当てはめる [1.2]

業務の依頼を、NLP のタスク（モデルがやる仕事の種類）に翻訳するところから設計が始まります。

要件の言い方	タスク	見分けの手がかり
段落のメモを 1 文の要点にする	要約（summarization）	長い入力から短い出力。意味は保つ

SAMPLE

要件の言い方	タスク	見分けの手がかり
問い合わせをカテゴリに振り分ける	分類（classification）	出力が固定のラベルのどれか
契約書から日付と金額を取り出す	抽出（extraction）	入力にある値を決まった項目に写す
日本語の FAQ を英語にする	翻訳（translation）	言語を変えて意味を保つ
商品の紹介文を書く	生成（generation）	入力に無い文章を新しく作る
社内規程について質問に答える	質問応答（question answering）	質問に対して根拠のある答えを返す

試験の見本問題には、段落ほどのメモ欄を 1 文の要旨に縮める要件が出ます。評価の対象にするタスクのカテゴリは、要約（summarization）です。分類や text2text の言葉に引きずられず、「長い文を短く縮めて意味を残す」を要約と読み取ります。[4]

Databricks には、これらのタスクを SQL から呼べる AI Functions があります。タスク専用の関数と、汎用の `ai_query` の 2 種類です。[5]

関数	できること
`ai_summarize`	要約
`ai_classify`	与えたラベルへの分類
`ai_extract`	定義したスキーマでの項目の抽出
`ai_translate`	翻訳
`ai_analyze_sentiment`	感情分析

SAMPLE

関数	できること
`ai_mask`	指定した種類の情報の伏せ字
`ai_fix_grammar`	文法の修正
`ai_similarity`	2 つの文字列の意味の近さ
`ai_gen`	プロンプトへの回答
`ai_parse_document`	文書から文章・表・図の説明とレイアウトを取り出す
`ai_forecast`	時系列のデータを、指定した先の期間まで予測する（表を返す関数）
`ai_query`（汎用）	プロンプトとモデルを自分で選べる

Databricks は、目的に合うタスク専用の関数があるならそこから始めることを勧めています。カスタマイズが要らず、Databricks が保守する研究由来の手法が使われるからです。`ai_query` は、プロンプト・モデルのパラメータ・出力形式をもっと細かく制御したいとき、カスタムモデル・ファインチューニング済みモデル・外部モデルを呼びたいとき、スループットや質をさらに詰めたいときに選びます。[5]

AI Functions を使う前提条件が 2 つあります。クラシックの SQL ウェアハウスでは使えません。Databricks Runtime 15.4 LTS 以上が必要で、今後 17.3 LTS 以上へ引き上げられる予定があります（18.2 以上が推奨）。コストの面では、タスク専用の関数は Model Serving を通して Databricks が管理するサーバーレス GPU で推論し、クエリを動かす計算資源とは別に課金されます。`ai_query` は指定した Model Serving のエンドポイントに対して課金されます。`ai_forecast`・`ai_predict_class` などは、実行元の計算資源の中だけで動き、別の推論費用は出ません。[5]

SAMPLE

`ai_parse_document`、`ai_extract`、`ai_classify` を SQL を書かずに定期実行するなら、ジョブに AI Function タスク（ベータ）を足します。実行のたびに、新しい入力か変更のあった入力だけを処理します。[6]

大量のデータにかける（バッチ推論）ときの勧めは 4 つです。[7]

1. データ全体を 1 つのクエリに出す。並列化・再試行・スケールは AI Functions が自動で行うので、自分で小分けにしない
2. Databricks がホストするモデル（`system.ai` のモデルサービス、名前が `databricks-` で始まるエンドポイント）を使う。プロビジョンドスループットより、手間のかからないこちらが向く
3. バッチ推論に最適化されたモデルを選ぶ。それ以外は処理量が下がり、完了が遅れる
4. 大きなジョブでは `failOnError` を false にする。失敗した行はエラーメッセージで返り、成功した結果をやり直さずに済む

要件を複数のタスクに割る場面と、LLM 以外で足りる場面 [1.2]

1 つのプロンプトに複雑な処理を詰めると、質が下がりがちです。目安は、複雑な論理が複数あるなら LLM の呼び出しを分けることです。サポートボットの例では、次の 3 呼び出しに分けます。[3]

1. 意図の分類: 問い合わせを「製品情報」「トラブルシューティング」「アカウント管理」のように分類する
2. 抽出: 意図に応じて、製品名・エラー内容・アカウント番号を取り出す
3. 書き換え（生成）: 抽出した意図と内容で、元の質問を具体的な検索用の質問に直す

SAMPLE

分けると遅延は増えますが、各段を細かく制御でき、検索される文書の質が上がる可能性があります。段ごとに別のモデルを使えます。意図の判定のような軽い仕事に高価な汎用モデルは過剰で、小さいモデルで間に合います。最初から最大のモデルを選ぶ必要はありません。質・遅延・コストを見ながらモデルを替えます。サブタスクごとに小さいモデルをファインチューニングする案もあります。[3]

抽出→分類→生成の型もあります。顧客のメールから注文番号と商品名を抽出し、苦情・質問・依頼に分類し、分類ごとの返信文を生成する流れです。各段の出力が次の段の入力になるので、段ごとに評価できます。

LLM を使わない方が良い場面もあります。入力の形が固定で規則で書ける（郵便番号の抜き出しに正規表現）、完全一致のキーワード検索で足りる、数値の予測で表形式データから学習できる（古典的な機械学習。`ai_predict_class` や `ai_forecast` のような AI Functions もある）、といった場合です。出力が毎回変わってよい文章を作るときだけ、LLM の柔軟さが生きます。LLM は呼び出しごとに費用と待ち時間が出るので、決まりきった処理を LLM に任せるのは設計として重すぎます。[3]

埋め込み・再ランキング・チャットなどモデルの種類と用途 [1.2]

モデルの種類と用途の対応は、次の表のとおりです。

種類	入力 → 出力	主な用途
チャット / 生成モデル	メッセージ → 文章	回答生成・要約・分類・抽出・ツール呼び出し
埋め込みモデル	文章 → ベクトル	意味での検索・類似度・クラスタリング・RAG の索引と検索

SAMPLE

種類	入力 → 出力	主な用途
再ランキングモデル	質問と候補の組 → 並べ替えた順位	検索で取れた候補の精度を上げる

埋め込みモデルは、文章を、意味を表す数値の列（ベクトル）に変えます。Databricks がホストする代表は 3 つです。

モデル	ベクトルの次元	入力できる長さ
GTE Large (En)（`databricks-gte-large-en`）	1024	8192 トークン
BGE Large (En)（`databricks-bge-large-en`）	1024	512 トークン
Qwen3-Embedding-0.6B（`databricks-qwen3-embedding-0-6b`）	最大 1024（設定できる）	約 32K トークン

BGE は入力が 512 トークンなので、それより長いチャンクは入りきりません。Qwen3-Embedding-0.6B は 100 以上の言語とコードに対応し、公開プレビューです。[8]

再ランキングは、最初の検索で取れた候補の並びを、質問との関連の強さで取り直す段です。mxbai-rerank や ColBERTv2 のようなクロスエンコーダのモデルで、検索の質の向上が見込めます。検索をやり直すのではなく、候補の順を入れ替える点が要点です。[3]

モデルの提供形態は 3 つに分かれます。

SAMPLE

提供形態	向く用途	課金
pay-per-token	使い方が変動する、または時々しか使わない	使ったトークンの分
オンデマンドのプロビジョンドスループット	専用の容量が要り、あとから容量を調整したい	割り当てた容量の分。期間の約束は無い
予約型のプロビジョンドスループット	使い方が読める、業務の中核のアプリやエージェント	固定の期間、予約した全容量。使わなくても払う

Databricks は `system.ai` に、pay-per-token のモデルサービス（例 `system.ai.claude-opus-5`）を用意しています。LLM アプリの試作は、まず汎用の LLM に当てて動かし、それからカスタムモデルの学習と配備に投資する順が勧められています。[9]

チェーンの部品と並び [1.3]

チェーンは、LLM を呼ぶ処理を部品に分けて順につなげたものです。RAG のチェーンは、推論時に次の流れで動きます。[10]

1. （任意）ユーザーの質問の前処理: 質問をテンプレートに入れる、別のモデルで書き換える、検索用のキーワードを抜き出す。出力は検索用のクエリ
2. 検索（retrieval）: 検索用のクエリを、文書のチャンクを埋め込んだときと同じ埋め込みモデルでベクトルにして、似たチャンクを上位から取る
3. プロンプトの拡張（augmentation）: 質問と検索で取れた文脈を、使い方を指示するテンプレートに入れて 1 つのプロンプトにする
4. 生成（generation）: 拡張されたプロンプトを LLM に渡し、文脈に根拠を置いた回答を作る
5. （任意）後処理: ビジネスのルールを適用する、出典を付ける、文を整える

SAMPLE

試験で押さえるのは、部品の役割と並びです。

部品	役割	入力 → 出力
プロンプトテンプレート	変数を差し込み、指示の枠を作る	変数の辞書 → プロンプト
リトリーバー	索引から関連する文書を取る	クエリ → 文書の一覧
LLM	回答を生成する	プロンプト → 文章
出力パーサー	LLM の出力を後段が使える型に直す	文章 → 文字列・辞書・オブジェクト

並びの原則は、リトリーバーがテンプレートの前、パーサーが LLM の後ろです。検索結果が無いまま、プロンプトは組み立てられません。クエリは、文書のチャンクと同じ埋め込みモデルで変換します。モデルが違うとベクトルの空間が合わず、類似度を比べても意味がありません。

検索で取る件数と、その件数をどう質問に組み合わせるかは、生成の質に大きく効く判断です。チェーン全体では、フィルタリングやアクセス権の確認、生成物のコンテンツのモデレーションといったガードレールを途中に挟めます。

LangChain では、| で部品をつなぎます。次のコードは、プロンプトテンプレート、LLM、出力パーサーをこの順に並べた最小のチェーンです。LLM の部分は、動作確認用の偽の関数で置き換えました。

SAMPLE

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.runnables import RunnableLambda

prompt = ChatPromptTemplate.from_messages([
    ("system", "次の文章から JSON だけを返す。キーは product と
sentiment。"),
    ("human", "{review}"),
])
fake_llm = RunnableLambda(
    lambda p: '{"product": "SmartHome Mini", "sentiment": "positive"}'
)
chain = prompt | fake_llm | JsonOutputParser()
chain.invoke({"review": "小さくて使いやすい"})
# {'product': 'SmartHome Mini', 'sentiment': 'positive'}  (dict)
```

`StrOutputParser` は文字列のまま返し、`JsonOutputParser` は辞書にします。本物の LLM を使うときは、Databricks 上のモデルなら `databricks_langchain` の `ChatDatabricks(endpoint="...")` が LLM の部品になります。[10]

入力と出力の形から部品を選ぶ [1.3]

部品は、入力と出力の形から逆算して選びます。

要件	選ぶ部品
下流のコードが JSON を読む	出力パーサー（`JsonOutputParser`）、または構造化出力の `response_format`

SAMPLE

要件	選ぶ部品
画面にそのまま文章を出す	`StrOutputParser`
質問に社内文書の根拠が要る	リトリーバーを追加し、検索結果をプロンプトに差し込む
会話の履歴を踏まえた回答	メッセージの履歴を渡す部品（履歴を含む質問の書き換えを検索の前に置く）
外部のシステムを呼びたい	ツール呼び出し（関数呼び出し）の部品と、ツールの結果を LLM に戻す経路

会話の履歴では、検索の前に「履歴を言い換えて 1 つの独立した質問にする」処理を挟みます。「それはいくら?」のような質問は、そのままでは検索に使えません。複数ターンの会話の言い換えは、質問の書き換え（query rewriting）の代表例です。誤字の修正や同義語への置き換えも、同じ段で行えます。ただし、書き換えを入れるなら、検索側の部品もそれに合わせて変える必要があります。[3]

質問に「2023 年の報告書」「ノート PC」のような条件が含まれるとき、その条件を抜き出して検索のフィルタに渡す設計もあります（フィルタ抽出）。これを動かすには、文書のメタデータに年や製品の分類を持たせるデータパイプライン側の準備と、フィルタを受け取れるリトリーバーの両方が要ります。チェーンの部品だけを足しても動きません。[3]

場面で考えます。社内の Q&A に、会話の履歴の反映、社内文書の検索、JSON での回答を入れます。履歴を使って質問を 1 つの独立した文に書き換える段が最初で、次に検索、続いて検索結果を差し込むプロンプトテンプレート、LLM、最後に JSON の出力パーサーが並びます。履歴を踏まえない質問のまま検索すると、「それはいくら」のような質問で関係のない文書を取ってきます。[3]

SAMPLE

出力を JSON に固定したいとき、構造化出力（`response_format`）を使うのか、出力パーサーで読むのかが分かれ目になります。前者は生成の段階で形を縛り、後者は生成された文字列を読んで直す方法です。Claude のモデルは `response_format` を `tools` と組み合わせられないので、ツール呼び出しと JSON の形の指定を 1 回の呼び出しで同時にしたい場合は、構造化出力ではなくパーサーなどで受ける設計になります。

業務の目標を入力と出力に書き下す [1.4]

「社員の問い合わせを減らしたい」のような業務の目標は、そのままでは設計できません。AI パイプラインの入力と出力に書き直します。入力には、質問の文、取り込む文書、メタデータ（部署・日付・権限・ユーザーの識別）があります。出力には、答えの文、出典（根拠にした文書への参照）、形式（JSON・表・文章）があります。

項目	例: 営業担当が顧客の契約条件を素早く確認したい
入力	担当者の質問、契約書の PDF 群、契約の相手先の ID
出力	条件の要約、該当する契約書の名前と条項
必要になる部品	文書から探すので検索。条項の参照を返すので出典。担当者が見られる契約だけを返すのでアクセス制御

この書き下しが済むまで、部品は選べません。

出力に出典を求める設計は、回答の検証可能性（説明責任）に直結します。RAG には、質問に出典を付けて返し、ユーザーが事実を確かめられる利点が

SAMPLE

あります。検索の段で、ユーザーの権限に基づき取り出す文書を絞れるので、個人情報や社外秘を含むデータのアクセス制御も組み込めます。[11]

RAG・エージェント・単純なプロンプトのどれで解くか [1.4]

入力と出力が書けたら、解き方を選びます。

解き方	向く条件	向かない条件
単純なプロンプト	入力だけで答えが出る（要約・分類・書式の変換）。外のデータが要らない	社内の最新の情報や、そのモデルが学習していない知識が要る
RAG	自社の文書・最新の情報・出典付きの回答が要る。事前に知識の入れ先が決まっている	回答の前に外部のシステムを呼んで状態を変える、手順が質問ごとに変わる
エージェント	質問ごとに手順が変わる。複数の知識源やツールを使い分ける。外部の操作（更新・送信）が要る	手順が固定で、1 回の検索と生成で終わる

RAG は、検索（retrieval）、プロンプトの拡張（augmentation）、生成（generation）の 3 段で、外部の知識源から取った裏付けを LLM に渡す方法です。得意なのは、メモ・メール・文書のようにモデルが学習していない独自の知識、更新された知識、出典の引用、アクセス権の反映です。RAG に使うデータは、PDF・Office 文書・Wiki のような非構造化データと、データベースのテーブルや API のような構造化データの両方があります。[11]

エージェントは、モデルが質問を受けて何をするかを決め、ツールを呼び、結果を見て、答えられるまで繰り返す仕組みです。単一のモデル呼び出しとは違

SAMPLE

い、ループで動きます。Databricks は最初の設計を、動く範囲でいちばん単純にするよう勧めています。そこから評価と改善を回して複雑にします。[12]

構造化データの扱いも設計判断です。試験の見本問題には、自動車部品の販売アプリが出ます。注文と請求の質問には答えられ、出荷と到着予定の質問には答えられません。顧客は `account_id` と `transaction_id` を入力する条件です。この場合の改善は、`transaction_id` を主キーにして、請求データと到着予定日を載せたテーブルを作り、その行を引いて回答に使うことです。出荷方針の文書だけを索引にしても、取引ごとの到着予定日は出てきません。ファインチューニングも、日々変わる到着日の対策としては合いません。プロンプトへ固定の「14 日足す」を書く案は、実際の日付から外れます。取引ごとに値が決まっている事実は、表の検索（キーでの引き当て）で渡すのが正しい設計です。[13]

表のデータを使う道具は、状況で選びます。

目的	使う道具
ワークスペースをまたぐビジネスの質問に自然言語で答える	Genie One の MCP
特定の Genie Agent（旧称 Genie スペース）に質問する	Python から AI Bridge のツール
SQL を自分で書いて実行させる	Databricks SQL の MCP
決まったクエリに、エージェントが引数だけ渡す	Unity Catalog の関数（ツール）

場面で考えます。経理部が、経費規程（PDF 120 本）について社員の質問に答えるボットを作ります。規程は毎月改定され、回答には条文の参照が要り、手順は質問ごとに変わりません。解き方は RAG です。単純なプロンプトでは、規

SAMPLE

程の中身をモデルが知りません。エージェントにすると、ツールを選ぶ段が増えて遅く、費用も上がります。同じ部署が「出張の申請も代わりに出して」と求めたら、申請システムへ書き込む操作が加わるので、ツール呼び出しを持つエージェントに広げます。[14]

待ち時間・費用・正確さ・説明責任を設計に写す [1.4]

要件に制約があれば、設計の部品に翻訳します。品質、費用、待ち時間は、RAG アプリが業務の要件を満たすかどうかを測る 3 つの軸です。評価は開発中、モニタリングは本番稼働後に行います。[11]

制約	設計での手
待ち時間が厳しい	LLM の呼び出し回数を減らす。軽いモデルを使う。取る件数を絞る。再ランキングは検索の質を上げる反面で段が増えるので、必要性を見て入れる
費用を抑えたい	軽い段（意図の判定）には小さなモデルを使い、重い生成だけ大きなモデルにする。バッチなら AI Functions に任せる
正確さを上げたい	検索の質（取れた文書が足りているか）と、生成の質（根拠に沿って答えているか）を分けて見る。文書の分割・メタデータ・埋め込みモデル・取る件数・再ランキング・プロンプトを調整する
説明責任が要る	出典を返す。ユーザーの権限で取れる文書を絞る。ガードレールを挟む

場面で考えます。顧客対応のチャットに、応答は 3 秒以内、誤答の理由を説明できること、月の費用に上限、という条件が付きました。意図の判定に大きなモデルを使っているなら、そこを小さなモデルに替えて待ち時間と費用を下

SAMPLE

げ、生成だけ大きなモデルに残します。説明責任には、出典の返却と、ユーザーの権限での文書の絞り込みを足します。再ランキングは検索の質を上げますが段が増えるので、3 秒の枠の中で測ってから入れるかを決めます。

正解の答え（ground truth）が手元に無くても Agent Evaluation はチェーンの質を測れ、正解があれば answer correctness のような指標も加わります。[15]

検索の質と生成の質は、きれいに分かれません。検索の質には、データパイプライン側（解析・分割・メタデータ・埋め込みモデル）とチェーン側（質問の変換・取る件数・再ランキング）が両方効きます。生成の質は、検索が悪いと必ず下がります。そのため、片側だけを直すのでなく、どの部品を替えると全体にどう効くかを考えます。取る文書に重要な情報が欠けている、不要な情報が混じっている、というときに高品質の回答は作りにくくなります。[16]

ツール呼び出し（関数呼び出し）の仕組みとツールの定義 [1.5]

関数呼び出し（function calling）では、LLM が関数を実行するのではありません。LLM は、呼びたい関数の名前と引数を JSON で作ります。その関数は、自分のコードが実行します。順序は 4 段です。[17]

1. `tools` に関数の定義を入れて、質問と一緒にモデルを呼ぶ
2. モデルが、関数を呼ぶかどうかを決める。呼ぶなら、スキーマに沿った引数の JSON が返る
3. 引数の文字列を JSON として読み、その引数で関数を実行する
4. 関数の結果を新しいメッセージに足して、もう一度モデルを呼ぶ。モデルが結果を要約して、ユーザーに返す

SAMPLE

ツールの定義は、名前、説明、引数のスキーマの 3 つです。次の例は Databricks の文書にある天気の関数です。

```
tools = [{
    "type": "function",
    "function": {
        "name": "get_current_weather",
        "description": "Get the current weather in a given location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {"type": "string",
                             "description": "The city and state, e.g.
San Francisco, CA"},
                "unit": {"type": "string", "enum": ["celsius",
"fahrenheit"]},
            },
        },
    },
}]
response = client.chat.completions.create(
    model="databricks-meta-llama-3-3-70b-instruct",
    messages=messages, tools=tools, tool_choice="auto",
)
```

関数の説明と引数の説明が、モデルが関数を選ぶ手がかりです。関数の定義には、パラメータと説明を必ず付けなければなりません。

SAMPLE

`tool_choice` は、モデルが関数を使う方法を制御します。

値	動き
`"auto"`（既定）	呼ぶかどうか、どの関数を呼ぶかをモデルが決める
`"required"`	必ず 1 つ以上の関数を呼ぶ。どれを呼ぶかはモデルが選ぶ
`{"type": "function", "function": {"name": "my_function"}}`	その特定の関数だけを呼ぶ
`"none"`	関数呼び出しを無効にして、ユーザー向けの文章だけを生成する

関数のスキーマの書き方は、構造化出力と同じく JSON スキーマの一部だけが使えます。`pattern`、`anyOf` `oneOf` `allOf` `prefixItems` `$ref`、`[型, "null"]` 以外の型の並びは使えません。スキーマのキー数の上限は 16 個で、構造化出力の 64 個とは別の数字です。1 回に定義できる関数は最大 32 個です。並列の関数呼び出しは使えません。入れ子が深いと質が下がります。

そのほか、Databricks の関数呼び出しは OpenAI 互換で、Foundation Model APIs と、外部モデルを配信するエンドポイントでだけ使えます。関数呼び出しはプレビューの間は単一ターンに最適化されていて、複数ターンの関数呼び出しには、対応する Claude のモデルが勧められています。Llama 4 Maverick は単一ターン向けで、複数ターンは開発中です。Claude Fable 5.2 は、強制的なツールの指定（特定の関数を指名する `tool_choice`）に対応しません。ツールの数が増えるほど入力のトークンが増えます。[17]

コードでは、モデルの返事に `tool_calls` があれば、`function.arguments` の文字列を `json.loads` で読み、関数を実行し、結果を `role` が `tool` のメッセー

SAMPLE

ジとして返します。次のコードは、モデルの返事を固定の値で置き換えて、この流れを動かしたものです。

SAMPLE

```
import json

def get_weather(location, unit="celsius"):
    return {"location": location, "temp": 21, "unit": unit}

TOOLS = {"get_weather": get_weather}
msgs = [{"role": "user", "content": "シカゴの気温を教えて"}]
fake = [
    {"role": "assistant", "content": None, "tool_calls": [{
        "id": "call_1", "type": "function",
        "function": {"name": "get_weather",
                     "arguments": "{\"location\": \"Chicago,
IL\"}"}}]},
    {"role": "assistant", "content": "21度です。"},
]
i = 0
for _ in range(5):  # 上限を決めてループする
    r = fake[i]; i += 1
    msgs.append(r)
    if not r.get("tool_calls"):
        print("final:", r["content"]); break
    for c in r["tool_calls"]:
        args = json.loads(c["function"]["arguments"])
        res = TOOLS[c["function"]["name"]](**args)
        msgs.append({"role": "tool", "tool_call_id": c["id"],
```

SAMPLE

```
                    "content": json.dumps(res)})
# 並び: ['user', 'assistant', 'tool', 'assistant']
```

メッセージの並びは、ユーザーの質問、ツール呼び出しを含むアシスタントの返事、ツールの結果、最終回答のアシスタントの返事の順です。Databricks の文書の例は、ツールを呼ぶループに回数の上限（10 回）を置き、超えたらエラーにするので、無限に回ることはありません。上のコードでは、上限を 5 回にしています。

多段の推論でツールを並べる順と ReAct の型 [1.5]

複数のツールを使う質問は、順序で成否が決まります。基本の順は、知識を集める、判断する、行動する、です。たとえば「顧客 A の未払いを確認して、あれば督促メールを出す」なら、顧客情報の検索、未払いの判定、メールの送信の順になります。前の結果が次の引数になる依存があるので、順序を入れ替えられません。督促メールを送る前に検索を挟まないと、誰に何を送るかが決まりません。

この型を実装した代表が ReAct（Reasoning and Acting）です。推論（何をするか考える）、行動（ツールを呼ぶ）、観察（結果を受け取る）を、答えを出せるまで繰り返します。エージェントの標準の構成も同じです。エージェントはモデルで状況を推論し、ツールを呼んで情報を集めたり、他のシステムの状態を変えたりし、結果を観察して、答えられるまで繰り返します。[12]

設計での注意は 3 つあります。

- 依存のあるツールは順に呼ぶ。1 回の関数呼び出しで複数の関数を並べて呼ぶ並列の呼び出しには、Databricks の関数呼び出しは対応していない

SAMPLE

- ループに回数の上限を置く。上限に届いたらエラーを返し、終わらない呼び出しを防ぐ
- ツールの説明を具体的に書く。モデルは説明を見て、どのツールをいつ使うかを決める。特にエージェントの調整役（スーパーバイザー）には、各ツールの説明を詳しく書くほど、仕事の振り分けが良くなる

Databricks の AI Playground では、コードを書かずにツール呼び出しのエージェントを試作できます。「Tools enabled」と表示されたモデルを選び、`Tools` の `+ Add tool` から、組み込みの Unity Catalog 関数 `system.ai.python_exec`（任意の Python を実行する）を足します。UC 関数、関数の定義（自分で書く関数）、AI Search（Vector Search の現名）の索引、MCP サーバーも足せます。足したツールを毎回すべて使うとは限りません。LLM が質問に合わせて選びます。AI Search の索引を使った回答には、出典が付きます。Playground でのエージェントの組み立ては、いまは legacy の扱いで、コードのエージェントには Agent Bricks CLI が勧められています。[18]

Unity Catalog の関数をエージェントのツールにする [1.5]

Unity Catalog の関数（UC 関数）は、エージェントに決まったロジック（SQL のクエリに引数を渡すなど）を実行させたいときの道具です。引数はエージェントが決めて、クエリは事前に作っておきます。クエリが事前に分かっているときの構造化データの取得に向きます。[14]

SQL で作る場合、関数と引数の COMMENT が、モデルが読む説明になります。

SAMPLE

```
CREATE FUNCTION my_catalog.my_schema.lookup_customer_info(
  customer_name_input STRING COMMENT 'Name of the customer whose info
to look up'
)
RETURNS STRING
COMMENT 'Returns metadata about a particular customer, given the
customer''s name, including the customer''s email and ID.'
RETURN SELECT CONCAT('Customer ID: ', customer_id, ', ',
                     'Customer Email: ', customer_email)
  FROM my_catalog.my_schema.customer_data
  WHERE customer_name = customer_name_input
  LIMIT 1;
```

Python の関数も登録でき、型ヒントと docstring（`Args` と `Returns` の説明）が、そのままツールの定義になります。

SAMPLE

```
from unitycatalog.ai.core.databricks import DatabricksFunctionClient
client = DatabricksFunctionClient()

def add_numbers(number_1: float, number_2: float) -> float:
    """Add two numbers.

    Args:
        number_1: The first number to add.
        number_2: The second number to add.
    Returns:
        The sum of the two numbers.
    """
    return number_1 + number_2

client.create_python_function(
    func=add_numbers, catalog="my_catalog", schema="my_schema",
replace=True)
```

エージェントへ渡すときは、`UCFunctionToolkit` に関数のフルネームを渡し、`toolkit.tools` をエージェントの `tools` にします。LangGraph なら、次の形です。[19]

SAMPLE

```
from databricks_langchain import ChatDatabricks, UCFunctionToolkit
from langchain.agents import create_agent

toolkit = UCFunctionToolkit(function_names=
["my_catalog.my_schema.add_numbers"],
                            client=client)
agent = create_agent(
    model=ChatDatabricks(endpoint="databricks-claude-sonnet-4-5"),
    tools=toolkit.tools,
)
```

権限は 3 層で押さえます。関数の親のカタログとスキーマに `USE CATALOG` と `USE SCHEMA` が要ります。関数を作るにはスキーマの `CREATE FUNCTION`、既存の関数を呼ぶには、その関数の `EXECUTE` が要ります。

そのほかの要点は次のとおりです。

- Python 3.10 以上のノートブックが要る。クラシック計算ならランタイム 15.0 以上
- `DatabricksFunctionClient` は、関数の作成と既定の実行にサーバーレスの Spark Connect を使う。サーバーレスを使えない ID では `PERMISSION_DENIED: Cannot access Spark Connect` が出る。SQL ウェアハウスだけでは足りない
- `execution_mode="local"` は、信頼できる Python 関数のテストに使う。関数を取得してローカルのサブプロセスで動かし、SQL 関数には対応しない。本番ではサーバーレスの既定を使い、信頼できるコードだけを実行する

SAMPLE

- 新しい Python のエージェントでは、登録した関数を `UCFunctionToolkit` で直接呼ぶ方法が勧められている。既存の連携向けには、UC 関数の MCP サーバーがある
- 外部 API を呼ぶ関数（`http_request` を使うもの）は、legacy の REST API ツールの扱い。新しく作るツールは MCP から始める

UC 関数の MCP サーバー（Public Preview）は、Unity Catalog に登録したカスタムの Python と SQL の関数を、あらかじめ決めたツールとして公開する Databricks のワークスペースの MCP エンドポイントです。アクセスは Unity Catalog の権限で管理されます。エンドポイントの URL は `https://<workspace-hostname>/api/2.0/mcp/functions/{catalog}/{schema}/{function_name}` で、OAuth のスコープは `unity-catalog` です。`system.ai` スキーマの関数もここから使え、代表が `system.ai.python_exec`（コードインタープリタ）です。[20]

読むだけのツールと、書き込む・外へ送るツールの分け方 [1.5]

ツールは、世界を変えないものと変えるものに分けて設計します。

種類	例	設計
読むだけ	顧客の検索（SQL の SELECT）、文書の検索、天気の取得	モデルの判断で自由に呼ばせてよい。権限は読み取りだけを与える
書き込む・外へ送る	メール送信、レコード更新、課金、外部 API への POST	呼ぶ前に人の承認を挟む。権限を最小に絞る。入力を検査する

任意のコードを実行するツール（`system.ai.python_exec` や、スーパーバイザーのコード実行）は、エージェントがアクセスできる機密情報を漏らす恐れが

SAMPLE

あります。信頼できるコードだけを実行し、ガードレールと適切な権限で意図しないアクセスを防ぐ責任は、使う側にあります。[21]

人の承認は、エージェントの実行を一時停止して人の返事を待つ形で作れます。Databricks のエージェントサーバー（DurableAgentServer）では、エージェントが人の入力のために止まると、呼び出し（invocation）の状態が `interrupted` になります。承認の結果は、同じ `session_id` の新しい呼び出しで `resume` として送ると続きから動きます。Supervisor Agent のウェブ検索は、公開のウェブへクエリを送る前に、毎回エンドユーザーの承認を求めます。[22]

ユーザーの識別も大切です。エージェントが長期の記憶を読み書きする相手を決める `actor` は、アプリでサインインしたユーザーから設定し、ユーザーが入力した文字列からは設定しません。入力文字列を信じると、他人の記憶を読めてしまいます。`actor` を渡さないと、エージェントはセッション ID を使うので、記憶は新しいセッションに引き継がれません。[22]

場面で考えます。経理のエージェントに、請求書テーブルを検索するツールと、取引先へ督促メールを送るツールを持たせます。検索は SELECT だけの UC 関数にして `EXECUTE` を渡し、モデルの判断で呼ばせます。メール送信は呼ぶ前に止め、担当者が内容を見てから `resume` で続きを動かします。全部の呼び出しに承認を挟むと、業務が回りません。世界を変える呼び出しだけに絞ります。[22]

Agent Bricks の全体像と、名前の変遷 [1.6]

試験ガイドが挙げる Agent Bricks の 3 つは、Knowledge Assistant、Multiagent Supervisor、Information Extraction です。どれも、コードを書かずに作る、特定の用途のエージェントです。画面で用途を書いて、知識源や

SAMPLE

ツールを選ぶと、サービングのエンドポイントができます。作り手は用途と材料を宣言し、作った後の質の改善は、集めた質問とガイドラインを足して回します（Supervisor は足したデータから調整役を再学習し、Information Extraction は自然言語のフィードバックで項目の説明を自動で調整します）。作った後の改善で、コードを直す必要はありません。ドメインの専門家が書いた質問と自然言語の指示（ガイドライン）を入れて回します。[23]

名前	用途	入力の材料
Knowledge Assistant	文書に基づく質問応答。出典を付けて答える	ボリュームのファイル、テーブルのファイル列、Vector Search（現 AI Search）の索引
Multiagent Supervisor（文書上の名は Supervisor Agent）	複数のエージェントとツールを束ねて、複雑な依頼をこなす	Genie Agent、エージェントの端点、UC 関数、MCP サーバーなど
Information Extraction	文書から決まった項目を JSON で取り出す	ボリュームのファイル、テーブルの列

名前と位置づけは、公式の文書で変わっています。Databricks の「Agent Bricks」という名前は、いまはコードを書く開発者向けのエージェント開発基盤（Agent Runtime での配備、Databricks Sandbox、メモリとセッション、MLflow のトレース）を指します。そのうえで、Knowledge Assistant と Supervisor Agent は legacy（新しいエージェントには勧めない旧来の製品）に分類され、代わりにコード不要なら Genie Agents、コードで組むなら Agent Bricks CLI が案内されています。[24] Information Extraction は新版が提供され、旧版が legacy です。試験ガイドの名前（Multiagent Supervisor）と、いまの文書の名前（Supervisor Agent）が違っても、別の製品として読み

SAMPLE

分ける必要はありません（文書の頁の URL は今も multi-agent-supervisor です）。[25]

Knowledge Assistant の使い所と材料 [1.6]

Knowledge Assistant は、文書についての質問に、引用付きで高品質に答えるチャットボットを作る製品です。従来の RAG の限界を補うため、Instructed Retriever という手法を使います。文書の言語は問いません。使いどころは、製品ドキュメントについてのユーザーの質問、人事規程についての社員の質問、サポートのナレッジベースに基づく顧客への回答です。作ると、下流のアプリから使えるエージェントのエンドポイントができます。[23]

要件は、サーバーレス計算、Unity Catalog、Model Serving へのアクセス、予算が 0 でないサーバーレスの使用ポリシー、対応リージョンのワークスペースです。知識源は 3 種類から選びます。

種類	条件
ボリューム（またはそのフォルダ）のファイル	txt, pdf, md, ppt/pptx, doc/docx。100 MB を超えるファイルと、500 ページを超える PDF・DOC・DOCX・PPT・PPTX は取り込みで飛ばされる。PowerPoint は 1 スライドを 1 ページと数える。TXT と MD にページ数の上限は無い。名前が `_` か `.` で始まるファイルも飛ばされる

SAMPLE

種類	条件
テーブルのファイル	ストリーミングテーブル、または Change Data Feed が有効なテーブル。BINARY か STRING の `content` 列と、`file_path`・`file_name`・`file_size`・`file_modification_time` を持つ STRUCT の `metadata`（または `_metadata`）列が要る。取り込まれるのは選んだ内容の列だけ
Vector Search（現 AI Search）の索引	埋め込みモデルが `databricks-gte-large-en`、`databricks-bge-large-en`、`databricks-qwen3-embedding-0-6b` のどれか。引用に使う Doc URI 列と、取り出す Text 列を指定する

取り込みは `ai_parse_document` を使うので、1 文書 500 ページの上限もそこから来ます。知識源は 1 つのエージェントに最大 10 個で、最後の 1 つは外せません。

運用での決まりが 4 つあります。[23]

- 作成から知識源の同期までに、数時間かかることがある
- ファイルの知識源は、更新や追加をしたら `Sync` を押す。同期は差分で、追加した新しいファイルだけが処理される。索引の知識源は自動で更新されるので、手動の同期は要らない
- 知識源の追加と削除ができるのは、エージェントの作成者だけ。名前と説明の更新と同期は `CAN MANAGE` を持つ人ができる
- 権限は `Can Manage`（設定・権限・品質の改善）と `Can Query`（エンドポイントへの問い合わせだけ。エージェントのページは見られない）の 2 つ

SAMPLE

知識源の「内容の説明」は、エージェントがどの知識源をいつ使うかを決める手がかりです。作ったら、画面のテストで `View thoughts`（考え方）、`View trace`（トレース全体）、`View sources`（引用元）を見て、引用が付くかを確かめます。エージェントを削除すると、既定のストレージに置かれた一時データなどはすべて消えます。[23]

Multiagent Supervisor の使い所 [1.6]

Multiagent Supervisor（Supervisor Agent）は、Genie Agent、エージェントの端点、Unity Catalog の関数、MCP サーバー、カスタムのエージェントを束ねて、領域の違う複雑な依頼をこなす仕組みです。仕事の分担、タスクの委任、結果の統合は Supervisor が行います。調べものと実績データの両方を使う市場分析、社内手続きの質問とチケット処理の自動化、方針・FAQ・アカウントなどの問い合わせへの素早い対応が使いどころです。[25]

使える部品と、エンドユーザーに必要な権限は次のとおりです。権限がなければ、Supervisor はその部品から有用な答えを返せません。[25]

部品	エンドユーザーに必要な権限
Genie Agent	その Genie Agent と、基になる Unity Catalog オブジェクトへのアクセス
公開したダッシュボード	`CAN VIEW`
Knowledge Assistant のエンドポイント	エンドポイントの `CAN QUERY`
Model Serving のエンドポイント	`CAN QUERY`
Unity Catalog の関数	`EXECUTE`（呼ぶには、親の `USE CATALOG`・`USE SCHEMA` も要る）

SAMPLE

部品	エンドユーザーに必要な権限
Unity Catalog のテーブル	`SELECT` と、親の `USE CATALOG`・`USE SCHEMA`
ボリューム	`READ VOLUME` と、親の `USE CATALOG`・`USE SCHEMA`
AI Search の索引	`SELECT` と、親の `USE CATALOG`・`USE SCHEMA`
Unity Catalog の MCP サービス	`EXECUTE` と、親の `USE CATALOG`・`USE SCHEMA`
カスタムの MCP サーバー・カスタムのエージェント	Databricks アプリの `CAN_USE`

必ず 1 つ以上のサブエージェント（部品）が要り、1 つの Supervisor に入れられるエージェントとツールは最大 50 個です。ユーザーがどの部品にもアクセスできなければ、Supervisor は会話を終えます。一部にだけアクセスできるなら、アクセスできない部品から会話をそらします。AI Search の索引は、Delta Sync の索引以外は使えません。各部品の説明を詳しく書くほど、仕事の振り分けが良くなります。

どの Supervisor にもコード実行のツールが最初から入っています。計算が要る依頼で、モデルが書いたコードを、サンドボックスのサーバーレス計算で実行します。言語は Python（既定）、SQL、シェルです。インターネットに出られません。アクセスできるのは、Supervisor にツールとして追加した Unity Catalog のテーブルとボリュームの読み取りだけで、エンドユーザーの権限が効きます。ウェブ検索は、Databricks がホストする `databricks-gpt-5` で動き、Supervisor のモデルが何でも同じです。強化セキュリティとコンプライアンスのアドオンを有効にしたワークスペースでは使えません。[25]

SAMPLE

場面で考えます。社内の IT ヘルプデスクで、手順書の質問（文書）、障害チケットの件数の集計（テーブル）、アカウントの再発行（関数）を 1 つの窓口で受けたいとします。文書の質問だけなら、Knowledge Assistant 1 つで足り、Supervisor は要りません。3 種類の仕事が混ざるので、Multiagent Supervisor に、Knowledge Assistant のエンドポイント、Genie Agent、UC 関数を並べ、各部品に説明を付けます。エンドユーザーには、`CAN QUERY`、Genie Agent へのアクセス、`EXECUTE` が部品ごとに別々に要ります。[25]

Information Extraction の使い所 [1.6]

Information Extraction は、構造化されていない文書やテキスト（PDF、画像、表を含む）から、決めたスキーマに沿って項目を取り出します。契約書から当事者と条件、請求書から明細と支払条件、診療記録と所見から重要事項、が使いどころです。新版は、Agent Bricks の画面、SQL、REST API、非同期 API（ベータ）から使えます。[26]

作り方は次のとおりです。

1. データを選ぶ。Unity Catalog のボリュームか、テーブルの STRING や VARIANT の列。ファイルが 1 つ、行が 1 つあれば作れる
2. 抽出のスキーマを決める。自然言語で説明して `Generate Schema` で JSON スキーマを作らせるか、フィールドの名前・型・説明を手で足す
3. 精度を見る。オプションに、`Precision mode`（微妙な項目や長い文書の精度を上げる）、`Citations`（抽出値の根拠）、`Confidence scores`（確信度）がある。自然言語のフィードバックか、スキーマの説明の手直しで直す
4. 評価データで測る。Unity Catalog のテーブルに、入力列（STRING か `ai_parse_document` の VARIANT）と、正解の JSON を入れた列を置く。項目ごとの正解率・適合率・再現率・F1 が出る

SAMPLE

5. 使う。SQL の `ai_extract` で全データに適用するか、Lakeflow のパイプラインを定期実行して、新しいデータが来るたびにストリーミングテーブルを更新する

`ai_extract` の呼び出しは、次のように、文章とスキーマを渡します。

```
SELECT ai_extract(
  'Invoice #12345 from Acme Corp for $1,250.00 dated 2024-01-15',
  '{"invoice_id": {"type": "string", "description": "Unique invoice
identifier"},
    "vendor_name": {"type": "string", "description": "Legal business
name"},
    "total_amount": {"type": "number", "description": "Total invoice
amount"},
    "invoice_date": {"type": "string", "description": "Date in YYYY-MM-
DD format"}}',
  options => map('version', '2.1'));
```

制限は 3 つです。REST API の `ai_extract` の既定の上限は、ワークスペースあたり 1 分に 120 リクエストです。エージェントの最大コンテキスト長は 128k トークンです。ユニオン型のスキーマには対応しません。非同期 API の入力は、ボリュームの JSONL ファイルで、1 行が `custom_id` と `body`（`{"text": "..."}`）を持つ JSON です。1 ファイル 1〜50,000 行、200 MiB までです。大量の処理には、ジョブの AI Function タスクが勧められています。旧版（legacy）の Information Extraction は、PDF を先にテーブルへ変える必要があり、強化セキュリティとコンプライアンスのワークスペースには対応しません。[26] [27]

SAMPLE

Agent Bricks と自前のコードの選び分け、質の上げ方 [1.6]

3 つの用途のどれかに当てはまり、コード無しで素早く作りたいときは、Agent Bricks が向きます。当てはまらないときや、ループと部品を細かく決めたいときは、自前のコード（Agent Framework）で書きます。

要件	選ぶもの
社内文書に引用付きで答える Q&A を、コード無しで作る	Knowledge Assistant
文書から、決まった項目を表にする	Information Extraction
Genie Agent・UC 関数・別のエージェントを役割ごとに束ねる	Multiagent Supervisor
独自のループ・独自の前後処理・任意のフレームワーク（LangGraph、OpenAI Agents SDK）で動かしたい	自前のコード（Agent Framework）。MLflow の `ResponsesAgent` で包んで配備する
ツールの呼び出し順・状態・承認を自分で制御したい	自前のコード

Agent Bricks の質の上げ方は、コードを直さず、材料と指示を足す方法です。

- 質問（ユーザーが実際に聞く質問、エージェントが間違えた質問）を `Examples` に足す
- 専門家が、質問ごとに自然言語のガイドライン（答え方の指示）を書く。保存した直後から効く
- Unity Catalog のテーブルから質問とガイドラインを取り込める。必要な列は `eval_id` 、`request` 、`guidelines` 、`metadata` 、`tags`
- 知識源の名前と説明、全体の `Instructions` を直す

SAMPLE

- Supervisor は、足した例とガイドラインから、調整役を再学習して最適化する
- Information Extraction は、自然言語のフィードバックで説明が自動で調整され、評価データで項目ごとの正解率が測れる

ラベル付きのデータを集めると品質が上がります。専門家に頼むには、`CAN_MANAGE` の権限を渡して設定ページのリンクを共有します。Supervisor の場合は、各サブエージェントへのアクセスも専門家に必要です。[23] [25]

Agent Bricks で足りず自前のコードへ移る判断は、「コードの外から指示しても直らない」ときです。たとえば独自の分岐が要る、使いたいツールが Agent Bricks の対応部品に無い、人の承認を挟む、のような場合です。[28]

場面で考えます。契約書から決まった 8 項目を表にする仕事は、Information Extraction でスキーマを書けば足ります。一方、送信の前後に独自の検査を挟み、既存の LangGraph のコードを動かし続けたいときは、画面の設定だけでは決めきれません。自前のコードで書き、`ResponsesAgent` で包んで配備します。

SAMPLE

サンプルはここまでです

続き(第2章〜巻末)は、Databricks Generative AI Engineer Associate 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

Footnotes

1. https://docs.databricks.com/aws/en/agents/tutorials/ai-cookbook/quality-rag-chain#post-processing-and-guardrails ↩

2. https://docs.databricks.com/aws/en/machine-learning/model-serving/structured-outputs#use-structured-outputs
   - https://docs.databricks.com/aws/en/machine-learning/model-serving/structured-outputs#limitations
   - https://docs.databricks.com/aws/en/machine-learning/model-serving/structured-outputs#structured-outputs-with-anthropic-claude-models ↩ ↩2 ↩3

3. https://docs.databricks.com/aws/en/agents/tutorials/ai-cookbook/quality-rag-chain#query-understanding
   - https://docs.databricks.com/aws/en/agents/tutorials/ai-cookbook/quality-rag-chain#prompt-augmentation
   - https://docs.databricks.com/aws/en/agents/tutorials/ai-cookbook/quality-rag-chain#llm ↩ ↩2 ↩3 ↩4 ↩5 ↩6 ↩7 ↩8

4. https://www.databricks.com/sites/default/files/2026-10/databricks-certified-gen-ai-associate-exam-guide-august-

SAMPLE

2026.pdf ↩

5. https://docs.databricks.com/aws/en/large-language-models/ai-functions#task-specific-and-general-purpose
   - https://docs.databricks.com/aws/en/large-language-models/ai-query#when-to-use-ai_query
   - https://docs.databricks.com/aws/en/large-language-models/ai-functions#-view-costs-for-ai-function-workloads ↩ ↩2 ↩3
6. https://docs.databricks.com/aws/en/large-language-models/ai-functions#use-ai-functions-in-production-workflows ↩
7. https://docs.databricks.com/aws/en/large-language-models/ai-query#best-practices ↩
8. https://docs.databricks.com/aws/en/machine-learning/foundation-model-apis/supported-models ↩
9. https://docs.databricks.com/aws/en/machine-learning/model-serving/foundation-model-overview ↩
10. https://docs.databricks.com/aws/en/agents/tutorials/ai-cookbook/fundamentals-inference-chain-rag ↩ ↩2
11. https://docs.databricks.com/aws/en/agents/retrieval-augmented-generation#what-is-retrieval-augmented-generation ↩ ↩2 ↩3
12. https://docs.databricks.com/aws/en/agents/concepts/#components-of-an-agent ↩ ↩2
13. https://www.databricks.com/sites/default/files/2026-10/databricks-certified-gen-ai-associate-exam-guide-august-

SAMPLE

2026.pdf ↩

14. https://docs.databricks.com/aws/en/agents/custom-agents/structured-retrieval-tools#run-a-predefined-query-with-a-unity-catalog-function ↩ ↩2

15. https://docs.databricks.com/aws/en/agents/tutorials/ai-cookbook/evaluate-define-quality ↩

16. https://docs.databricks.com/aws/en/agents/tutorials/ai-cookbook/quality-overview ↩

17. https://docs.databricks.com/aws/en/machine-learning/model-serving/function-calling#what-is-function-calling
    - https://docs.databricks.com/aws/en/machine-learning/model-serving/function-calling#-use-function-calling
    - https://docs.databricks.com/aws/en/machine-learning/model-serving/function-calling#limitations ↩ ↩2

18. https://docs.databricks.com/aws/en/large-language-models/ai-playground ↩

19. https://docs.databricks.com/aws/en/agents/custom-agents/create-custom-tool#step-3-use-the-function-in-your-agent ↩

20. https://docs.databricks.com/aws/en/agents/mcp-tools/uc-functions ↩

21. https://docs.databricks.com/aws/en/agents/agent-bricks/multi-agent-supervisor#supported-subagents-and-tools ↩

SAMPLE

22. https://docs.databricks.com/aws/en/agents/query-llms#request-body ↩ ↩2 ↩3

23. https://docs.databricks.com/aws/en/agents/agent-bricks/knowledge-assistant#step-1-configure-your-agent ↩ ↩2 ↩3 ↩4 ↩5

24. https://docs.databricks.com/aws/en/agents/legacy/ ↩

25. https://docs.databricks.com/aws/en/agents/agent-bricks/multi-agent-supervisor#supported-subagents-and-tools • https://docs.databricks.com/aws/en/agents/custom-agents/create-custom-tool ↩ ↩2 ↩3 ↩4 ↩5 ↩6

26. https://docs.databricks.com/aws/en/agents/agent-bricks/info-extraction#step-2-configure-and-refine-your-extraction-schema ↩ ↩2

27. https://docs.databricks.com/aws/en/agents/agent-bricks/key-info-extraction#limitations ↩

28. https://docs.databricks.com/aws/en/agents/custom-agents/author-agent ↩

SAMPLE

Databricks Certified Generative AI Engineer Associate 学習用テキスト(無料サンプル)

版

2026年11月版(Databricks Certified Generative AI Engineer Associate 出題範囲(Exam Guide・2026 年 3 月 18 日版)準拠)

発行

IT資格道場(https://www.shikaku-doujou.com)

お問い合わせ

info@shikaku-doujou.com

Databricks、Unity Catalog、Mosaic AI、MLflow および Databricks の製品名・サービス名は、Databricks, Inc. の米国およびその他の国における商標または登録商標です。Apache、Apache Spark、Spark は Apache Software Foundation の商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式シラバスの内容を基に作成したもので、Databricks, Inc. との提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点は info@shikaku-doujou.com までご連絡ください。

SAMPLE