

AI-901 学習ノート

Microsoft Azure AI Fundamentals

目次

AI-901 学習ノート | Microsoft Azure AI Fundamentals

 試験範囲のバージョンについて (必読)

 試験の基本情報

 出題比率と時間配分

 学習ロードマップ (目安: 15~20時間)

第1章 | AI の概念と機能を特定する (40~45%)

1-1. 責任ある AI の原則

1-2. AI モデルのコンポーネントと構成

1-3. AI ワークロード

 第1章 チェック問題

第2章 | Microsoft Foundry を使用して AI ソリューションを実装する (55~60%)

2-1. Foundry の基本 (第2章の土台)

2-2. 生成 AI アプリとエージェントを実装する

2-3. テキストと音声の AI ソリューションを実装する

2-4. コンピュータービジョンと画像生成の AI ソリューションを実装する

2-5. 情報抽出の AI ソリューションを実装する

 第2章 チェック問題

第3章 | 直前まとめ

3-1. 早見表

3-2. 逆引き「〜したいときは何を使う？」

3-3. 紛らわしいペア比較

3-4. 数字・固有名詞の暗記リスト

3-5. コードの早見表

第4章 | 学習チェックリストと試験当日

4-1. 理解度チェックリスト

4-2. 試験当日のポイント

4-3. 追加で活用したい公式リソース

AI-901 学習ノート | Microsoft Azure AI Fundamentals

公式の出題範囲 (2026年4月15日版) の各項目に沿って、概念と実装を整理した学習ノートである。上から順に読み、章末のチェック問題、問題集 (200問。<https://ai-901-study.pages.dev/quiz>)、公式の練習用評価、Foundry での操作とコードの実行で理解を確かめる。

⚠ 試験範囲のバージョンについて (必読)

AI-901 の出題範囲は 2026年4月15日 版 (Skills measured as of April 15, 2026) である。

項目	状況
英語版	2026年4月15日版の範囲
日本語版	学習ガイドの日本語ページ (機械翻訳) も、2026年4月15日版を載せている。公式には、ローカライズ版の試験は英語版の更新から約8週間後に更新され、予定どおりにならないこともあるとされている

変更点

学習ガイドに変更ログ (Change log) は無い (2026年10月10日に確認)。AI-900 は2026年6月30日に廃止され、AI-901 に置き換えられた。同じ認定 (Microsoft Certified: Azure AI Fundamentals) を取るには、今は AI-901 に合格する。

✔ **結論**: このノートの章立ては2026年4月15日版の各項目に対応している。受験の前に、英語版の学習ガイドで日付が変わっていないかを確認する。項目への対応だけで、理解の定着や合格が保証されるわけではない。

- 💡 **公式の学習パスは2本** 試験ページの「Two ways to prepare」には、自習用の学習パスが2本ある。このノートは、両方の内容を出題範囲の項目の順に並べ直している。
- **AI concepts for developers and technology professionals**: 概念 (責任ある AI、生成 AI とエージェント、自然言語処理、音声、コンピュータービジョン、情報抽出、RAG) の7モジュール
 - **Get started with AI applications and agents on Azure**: Foundry での実装の7モジュール。最後の「Get started with Microsoft Foundry IQ」は、出題範囲に項目としては無い (Foundry IQ は一部の機能が一般提供、ポータルの agentic retrieval はプレビュー)

📄 試験の基本情報

項目	内容
正式名称	Exam AI-901: Microsoft Azure AI Fundamentals (日本語ページの表記は「Microsoft Azure AI の基礎」)
認定資格	Microsoft Certified: Azure AI Fundamentals
合格ライン	700点 (1~1000 のスケールドスコア。正答率の70%と同じとは限らない)
問題数	公表されていない。Microsoft の認定試験は、一般に40~60問程度で、変わることがある
試験時間	AI-901 個別の記載は無い。Fundamentals の試験は、試験時間45分 (着席時間65分) とされている
出題形式	事前には公表されない。試験サンドボックスで画面の操作を体験できる
受験方法	Pearson VUE で予約し、オンライン (監督付き) かテストセンターで受ける
言語	日本語を含む13言語
有効期限	Fundamentals の認定に期限は無い (更新は不要)
前提知識	Azure の AI ソリューションの概念的な知識と、それを扱う基本的な技術力。Python のコーディングの構文とプログラミングの手法の知識。Azure のリソースに慣れていること

⚠ 問題数と試験時間は、AI-901 個別の公表値ではない。受験の前に試験ページで確かめる。

学習ガイドの注記には、ほかに次のことが書かれている。

- ほとんどの問題は一般提供 (GA) の機能を扱う。よく使われているプレビューの機能は出題されることがある
- REST (Representational State Transfer) の API、SDK、CLI に慣れている必要がある
- 各項目の箇条書きは、そのスキルをどう評価するかの例示で、関連するトピックも出題されうる

📊 出題比率と時間配分

領域	比率	優先度
1. AI の概念と機能を特定する	40~45%	★★
2. Microsoft Foundry を使用して AI ソリューションを実装する	55~60%	★★★

戦略メモ

- 第2章が半分以上を占める。項目の多くは「Foundry SDK などて軽量アプリを作る」なので、コード例はクライアントの作り方、呼ぶメソッド、結果の取り出し方の3点で読めるようにする
- 第1章は第2章の土台になる。トークン、埋め込み、デプロイの種類、ワークロードの区別を先に固める

📖 学習ロードマップ(目安:15~20時間)

ステップ	内容	目安時間
1	このノートを通読(用語の全体像をつかむ)	3h
2	第1章を精読+章末チェック	4h
3	第2章を精読+章末チェック。コード例を1行ずつ読む	7h
4	直前まとめを反復	2h
5	問題集と公式の練習用評価 → 間違えた分野をノートに戻って復習	3h

第1章 | AI の概念と機能を特定する (40～45%)

この章では、責任ある AI の6原則を場面から見分けること、生成 AI モデルのしくみ・選び方・デプロイの設定、そして AI ワークロードの種類と手法を区別できるようにする。

1-1. 責任ある AI の原則

責任ある AI とは、有害・違法・不快なコンテンツの生成や、自動化された行動のリスクを減らすガードレールを含めて、AI システムを作るための考え方である。AI システムを作る側は、構想から設計・実装・運用までの各段階で、これを考える。Microsoft は、その原則として次の6つを挙げている。学習ガイドでは、6つそれぞれが「～に関する考慮事項を説明する」という個別の項目になっている。

原則	英語	一言で	典型的な場面
公平性	Fairness	誰に対しても公平に扱う	融資の審査 AI が、性別や人種で結果を変えないようにする
信頼性と安全性	Reliability and safety	想定どおりに安全に動く	医療の診断を助ける AI を厳密にテストし、想定外の入力でも危険な結果を出さないようにする
プライバシーとセキュリティ	Privacy and security	個人のデータを守る	学習データや入力に含まれる個人情報を保護し、不正なアクセスを防ぐ
包括性	Inclusiveness	あらゆる人が使えるようにする	手が不自由な人でも操作できるよう、音声でも入力できるようにする
透明性	Transparency	しくみと限界を理解できるようにする	求人を推薦する AI について、推薦に効いている要素と、精度が落ちる条件を利用者に説明する
説明責任	Accountability	人間が責任を持つ	AI の判断に責任を負う担当者と、問題が起きたときの対応の手順を決めておく

各原則の考え方

- **公平性**: AI は学習データの偏りを学んでしまうことがある。過去の採用の実績に偏りがあれば、採用を支援する AI もその偏りを再現するおそれがある。データそのものだけでなく、データを選ぶ基準にも無意識の偏りが入りうる。作る側は、特定の属性の人だけが不利な結果を受けていないかを評価する
- **信頼性と安全性**: AI は確率的に動くため、テストと違う入力 came ときに予想外の出力をすることがある。作る側は、本番の前に十分にテストし、想定外の入力でも危険な結果を出さない設計にする。たとえば、予測の信頼度がしきい値を下回るときは、AI に行動させない
- **プライバシーとセキュリティ**: AI は大量のデータを扱うため、個人情報の扱いに特に注意が要る。作る側は、学習データを安全に保つだけでなく、学習済みのモデルから個人や組織の非公開の情報を引き出せないようにする。不要になった個人データは速やかに消し、見る必要の無い人にはアクセスさせない
- **包括性**: 身体能力、性別、民族、年齢などにかかわらず、あらゆる人が AI の恩恵を受けられるようにする。公平性が「結果が偏らないこと」なのに対し、包括性は「使える人の範囲を広げること」である
- **透明性**: 利用者が AI システムの目的、しくみ、限界を理解できるようにする。AI を使っていることを明示する、学習に使ったデータの特徴（機密を明かさな範囲で）や誤りやすい場面を開示する、といったことが該当する
- **説明責任**: AI システムを設計・運用する人と組織が、その結果に責任を持つ。AI が判断したからといって責任は消えない。社内のガバナンスの体制、承認の流れ、問題が起きたときの対応の手順を整える

生成 AI では、有害なコンテンツの生成を抑える方法の1つとして **コンテンツフィルター** が使われる。

★ **試験ポイント**: 迷いやすい組み合わせは、何が問題になっているかで分ける。

- 公平性 vs 包括性: **結果の偏り** なら公平性、**使える人の範囲** なら包括性
- 透明性 vs 説明責任: **利用者が理解できる** ようにする話なら透明性、**人や組織が責任を負う** 話なら説明責任
- 信頼性と安全性 vs プライバシーとセキュリティ: **AI の動作が危険か** なら信頼性と安全性、**データの保護** ならプライバシーとセキュリティ

1-2. AI モデルのコンポーネントと構成

生成 AI モデルのしくみ

この節では、生成 AI が文章を作るしくみを説明する。テキストを生成する AI の中心にあるのは **言語モデル** で、その働きは **学習** と **生成** の2段階に分けると整理しやすい。学習は、モデルを作るときに、大量のテキストから言葉どうしの関係を覚える段階である。生成 (**推論** とも呼ぶ) は、学習済みのモデルを使うときに、**プロンプト** (応答を得るためにモデルに与える入力) に続く文章を作る段階である。

言語モデルのうち、大規模なものを **大規模言語モデル (LLM: Large Language Model)**、よりコンパクトなものを **小規模言語モデル (SLM: Small Language Model)** と呼ぶ。言語モデルは、語や句の言語的・意味的な関係を学んでいて、プロンプトに続く **補完** (completion) を生成するよう学習されている。

LLM と SLM の違いは、学習データの量と、モデルの中の変数の数による。LLM は強力で汎化しやすいが、学習と利用のコストが高くなりやすい。SLM は、特定の分野に絞った用途や、デバイス上で動くローカルのアプリやエージェントのように、小さなモデルを手軽にデプロイしたい場面に向く。

多くの言語モデルは **Transformer** というしくみを使う。学習パスは、学習と生成の両方を、Transformer の簡略化した説明 (エンコーダーとデコーダー) で説明している。

学習: 言葉どうしの関係を覚える

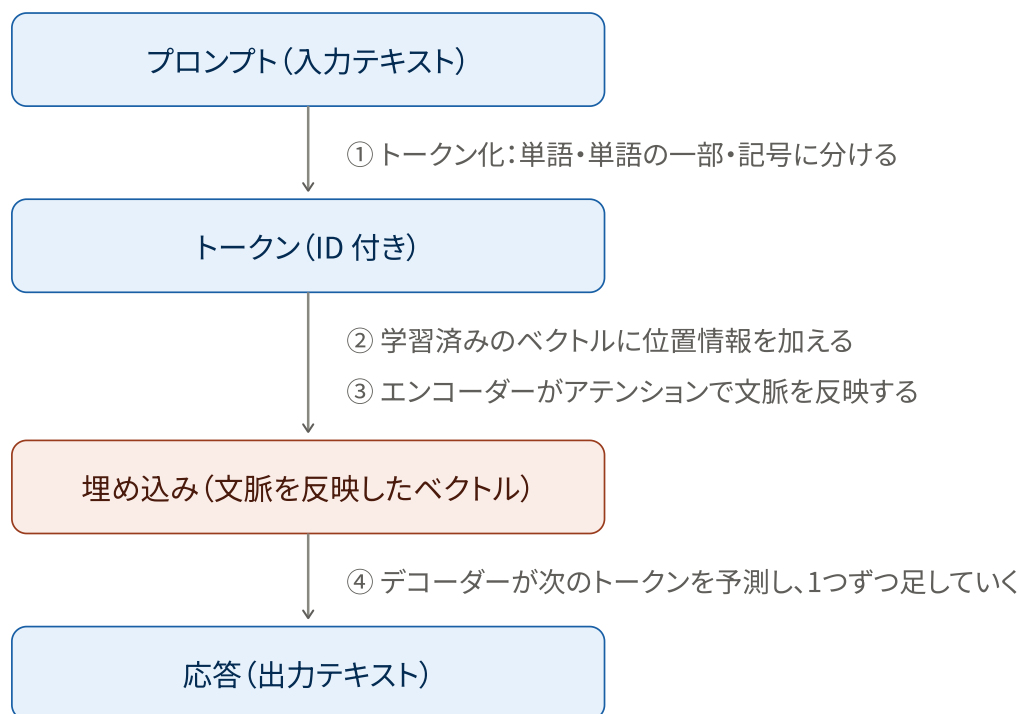
- **トークン化**: 学習では、まず学習用のテキストを **トークン** という単位に分け、それぞれに一意の整数の ID を付けて、語彙を作る。これが学習の最初のステップである。トークンには、単語のほか、単語の一部、句読点、よく使われる文字の並びが含まれる。最新の LLM の語彙は、数十万のトークンからなる
- **ベクトルと位置エンコーディング**: 次に、各トークンに、数値の並び (**ベクトル**) を割り当てる。ベクトルの値は最初はランダムで、トークンが文の中のどこにあるかを示す **位置エンコーディング** と一緒に、Transformer に入力する

- **アテンションと埋め込み**:Transformer の **エンコーダー** は、**アテンション**(あるトークンがほかのどのトークンにどれだけ影響されるかを重み付けするしくみ)で、学習データの中での使われ方を反映したベクトルを作る。大量のテキストに触れるほど、一緒に使われやすいトークンのパターンを学んでいく。こうして意味や文脈を数値で表したベクトルを **埋め込み (Embedding)** と呼ぶ。意味が近い内容ほど、埋め込みのベクトルの向きが近くなる傾向がある(近さはコサイン類似度で測る)。たとえば「医師」と「看護師」は近く、「医師」と「自転車」は遠い。この性質は、意味の近い文書を探す検索にも使われる
- **デコーダーの学習**:Transformer の **デコーダー** は、続きがわかっている文を使い、それより前のトークンだけから次のトークンを予測する練習をする(後ろのトークンを隠すアテンションを使う)。学習の処理は、デコーダーの予測と正解を比べ、外れが減るように重みを調整していく

こうして大量のデータで学習を済ませ、汎用的な言語・推論・マルチモーダル能力を持つ大規模なモデルを **基盤モデル(Foundation model)** と呼ぶ。GPT、Claude、Mistral などがこれにあたる。基盤モデルはそのままデプロイして使うことも、追加の学習(**ファインチューニング**)で特定の用途向けに調整することもできる。

生成:プロンプトに続く文章を作る

学習済みの言語モデルが、プロンプトを受け取ってから応答を返すまでの流れは、次の図のように整理できる。図は、学習パスの簡略化した説明に沿って、トークン・埋め込み・アテンション・予測の関係を示す概念図であり、特定のモデルの内部構成をそのまま表したものではない。



- ①～③ **プロンプトを埋め込みにする**:モデルは、受け取ったプロンプトも学習のときと同じようにトークンに分け、学習済みのベクトルに位置情報を加える。そのうえで、エンコーダーがアテンションで文脈を反映する。入力と出力の長さはトークンの数で数え、Standard のデプロイでは、料金やレート制限もトークンの数で計算される
- ④ **次のトークンの予測**:デコーダーは、プロンプトと生成済みのトークンから、次のトークンの確率を予測し、temperature などの設定に従って1つを選ぶ。選んだトークンを足し、また次を予測する処理を、終了のトークンや出力の上限まで繰り返す。毎回、最も確率の高いトークンだけを選ぶとは限らない

★ **試験ポイント**:テキストを最初に分ける単位 → トークン(学習の最初のステップもトークン化)。意味の近さを表すベクトル → 埋め込み。文脈を捉えるしくみ → アテンション(Transformer)。埋め込みを作るのはエンコーダー、次のトークンを1つずつ予測して文章を作るのはデコーダー(学習パスの簡略化した説明)。

言語モデルの限界と RAG

言語モデルには、主な限界として次の3つがある。

限界	内容
知識の範囲	公開されていない情報(社内の文書など)を知らない
知識の鮮度	学習した情報が古くなる
検証可能性	信頼できる根拠に支えられていない回答を、自信ありげに出すことがある(ハルシネーション)

対策の中心は **グラウンディング**、つまりモデルの回答を信頼できる情報源に基づかせることである。その代表的な手法が **RAG** (Retrieval-Augmented Generation: 検索拡張生成) である。RAG を使うアプリは、関係する情報を探す **取得** (retrieval) と、取り出した情報を文脈としてプロンプトに加え、モデルに答えさせる **拡張生成** (augmented generation) を組み合わせる。RAG はモデルの学習済みのパラメーターを変えない。利用者が質問するたびに、アプリが関係する情報を探してプロンプトに加え、モデルに渡す。そのため、元の文書と検索用のインデックスを更新すれば、モデルを学び直さなくても、モデルは新しい情報で答えられる。ただし、RAG は根拠の無い回答を減らすのが、なくすわけではない。

データの準備(インデックスの作成):RAG のアプリが検索できるのは、前もって検索できる形に準備した情報だけである。この準備では、元の文書からテキストを取り出し、**チャンク** (小さな断片) に分け、埋め込みのベクトルに変え、検索用のインデックスを作る。チャンクが大きすぎると関係の無い情報が混じり、モデルのコンテキストウィンドウ(入力と出力

を合わせて、一度に扱えるトークンの量)を多く使う。小さすぎると、理解に要する見出しや定義から切り離される。境界で文脈を失わないよう、チャンクを少し重ねることが多い。準備は1回では終わらない。元の文書の追加・変更・削除を、インデックスに反映し続ける。

検索の種類	向いている場面
キーワード検索	正確な語、識別子、製品コードが大事なとき
ベクトル検索	意味の近い内容を探すとき。利用者の質問も、インデックスを作ったときと同じ埋め込みモデルでベクトルにし、チャンクのベクトルと比べる
ハイブリッド検索	キーワード検索とベクトル検索を組み合わせる

拡張プロンプト: アプリがモデルに送るプロンプトには、システム指示、ユーザーの質問、根拠として取り出したチャンク、「与えた文脈から答え、足りなければそう伝え、できれば引用を付ける」という指示を入れる。取り出した文書には、モデルの動きを変えようとする文が混じるおそれがあるので、アプリは取り出した文書を、命令ではなくデータとして扱う。RAGの評価は、取得(関連性、網羅性)と生成(根拠性、関連性、引用の質)を分けて行う。正しい情報が取得できていなければ、生成のプロンプトを変えるだけでは直らない。

このほかの弱点に **出力のばらつき** があり、同じ入力でも出力が変わることがある。temperature を下げるとばらつきを減らせるが、なくせるわけではない。同じ入力に同じ値が要するなら、用途に特化したツールを使う。

★ **試験ポイント:** 社内の情報を知らない、情報が古い、根拠が無い → RAG でグラウンディングする。RAG はモデルを学び直させない。意味の近さで探す → ベクトル検索、製品コードなどの完全一致 → キーワード検索、両方 → ハイブリッド検索。

機能に基づいて適切なモデルを選ぶ

Microsoft Foundry (旧称 Azure AI Foundry。その前は Azure AI Studio) の **モデルカタログ** は、数千のモデルを探して比較できる場所である。提供元、機能、推論のタスクなどで絞り定める。モデルカタログのモデルは、提供のされ方で2種類に分かれる。

種類	内容
Azure が直接販売するモデル (Models sold directly by Azure)	Microsoft がホストし、Microsoft の製品条件で提供する。すべての Azure OpenAI のモデルと、主要な提供元の一部のモデルが含まれる。Azure の SLA (Service Level Agreement: サービスレベル契約) とサポートの対象で、セキュリティとコンプライアンスの備えも整っている
パートナーとコミュニティのモデル (Models from partners and community)	パートナーやコミュニティが提供する、それ以外のモデル

各モデルの詳細ページには通常、機能の説明、ベンチマークの結果、対応する推論のタスク、ファインチューニングの可否、責任ある AI に関する文書 (**モデルカード**) が載っている。関連するモデル (同じ系統で、サイズ、能力、バージョンが違うもの) のまとまりを **モデルファミリー** と呼ぶ。学習パスで「よく使われるモデルファミリー」として挙がっているのは次の3つ。

モデルファミリー	特徴
GPT-5.x (OpenAI)	多段階の推論、計画、エージェントの処理に強い。推論の深さを調整できる
Claude Opus (Anthropic)	高度なエージェント、複雑なコードの推論に強い
Mistral Large 3 (Mistral AI)	汎用モデル。品質とコスト・速度のバランスが良い

学習パスは、GPT-5 ファミリーは登録が必要で使える人が限られるとし、全 Foundry ユーザーが使える、速度と低遅延に最適化された GPT-4.1 も紹介している (どちらも学習パスの時点の説明)。

タスクから、どのタイプのモデルを選ぶかを判断できるようにしておく。

タスク	選ぶモデルのタイプ	例
チャット	会話向けの汎用モデル。軽量でよければ SLM	GPT-5.x chat, GPT-4.1, Phi-4
コーディング	コードに特化したモデル	GPT-5.1-codex, Claude Sonnet
要約	長い文脈を扱える推論モデル	GPT-5.x, Claude Opus / Sonnet
埋め込み	埋め込み専用のモデル	text-embedding-3-small
マルチモーダル	テキストに加えて、画像や音声などを入力できるモデル	Phi-4-multimodal-instruct, GPT-5.x chat, Mistral Large 3
画像生成	テキストから画像を作るモデル	GPT-Image-1.5, GPT-Image-1
動画生成	テキストや画像から動画を作るモデル	Sora
業界特化	特定の分野向けに調整されたモデル	金融・医療・法務向けのモデル

用途がはっきり決まっている場合 (言語検出、音声の文字起こし、請求書からの項目の抽出など) は、汎用モデルの代わりに **Foundry Tools** (旧称 Azure AI services。その前は Azure Cognitive Services) を使う選択肢もある。Azure Language、Azure Speech、Azure Content Understanding などが含まれる。Foundry Tools は構築済みのモデルで動き、結果が予測しやすく、すぐ使える。

💡 **モデルの顔ぶれは入れ替わりが速い** 2026年10月時点の製品ドキュメントでは、GPT-4.1 系は新しく使い始める利用者がデプロイできない状態 (Deprecated) で、GPT-5.x の chat モデルは提供を終えている。登録が必要なモデルとして GPT-5 ファミリーは挙がっていない。このノートのモデル名は学習パスの例に合わせているので、今も使えるかは製品ドキュメントで確かめる。

★ **試験ポイント**:「埋め込み」は文章をベクトルに変えるモデルで、文章を生成しない。検索や類似度の計算 → 埋め込みモデル。同じ入力に同じ値が要る定型の処理 → Foundry Tools。

モデルを評価する

Foundry ポータルでは、次の方法でモデルを比べて評価できる。

- **ベンチマーク**:標準的なデータセットでの性能のスコア。共通の基準で比べられる
- **リーダーボード**:品質、安全性、スループットなどでモデルを順位付けしたもの
- **自分のデータで試す**:モデルのページのベンチマークから「Try with your own data」を選び、実際のプロンプトで試す

評価の指標には2つの系統がある。

系統	指標の例	使いどころ
従来の NLP (Natural Language Processing: 自然言語処理) の指標	正確度 (accuracy)、適合率 (precision)、再現率 (recall)、F1	正解がはっきり決まるタスク
AI 支援の品質の指標	根拠性 (groundedness) 、関連性 (relevance)、一貫性 (coherence)、流暢性 (fluency)、GPT 類似度 (GPT similarity)	生成した文章の質を、従来の指標では測れない観点で評価したいとき

評価器 (Evaluator) は、モデルやエージェントの出力の品質・安全性を測る部品である。たとえば安全性の評価器は、有害なコンテンツ、偏り、暴力、自傷などを見つける。Foundry には、再利用できる評価器をまとめた **Evaluator Library** がある。**評価器そのものは、問題を検出・採点するだけで、直しはしない。**

モデルのデプロイと構成パラメーター

デプロイ とは、モデルをアプリから API で呼び出せる状態にすることである。本番だけでなく、開発や検証にも使う。デプロイすると、次のことが起きる。

- 計算資源 (CPU・GPU・メモリなど) が割り当てられる
- API のエンドポイントが作られ、アプリから呼び出せるようになる
- 選択したモデルのバージョンや安全性の設定などの構成が適用される
- 使用量、性能、遅延、エラー、コストの監視とログの記録が始まる

デプロイのときに設定する主な項目は、**デプロイの種類、モデルのバージョン、TPM (Tokens Per Minute:1分あたりのトークン数) のレート制限** の3つ。

デプロイ名 はデプロイのときに付ける名前。コードからモデルを呼び出すときに指定するのはこのデプロイ名である。たとえば gpt-4.1 を「helpdesk-chat」という名前でデプロイしたら、コードでは「helpdesk-chat」を指定する。既定では、デプロイ名はモデル名と同じになる。

デプロイの種類

デプロイの種類は、「課金・処理の方式」と「データを処理する場所」の2つの軸の組み合わせで決まる。

方式	課金	向いている場面
Standard	使ったトークン数に応じた従量課金	利用量が変わる、ばらつきの大きい処理
Provisioned	処理能力を PTU (Provisioned Throughput Unit) の単位で確保し、PTU の数と時間で支払う。使ったトークン数には関係しない	大量で安定した処理量があり、予測できる性能が要る本番の環境
Batch	Global Standard より50%安い。24時間以内の完了が目標 (それより長くなることもある)	急がない大量の非同期の処理

処理する場所	内容
Global	Azure のどのリージョンでも処理されうる。Global Standard は多くのワークロードの出発点で、新しいモデルが最初に提供され、Standard 系でいちばん安く、使えるリージョンもいちばん広い
Data Zone	Microsoft が定めたデータゾーン (US、EU、アジア太平洋) の中で処理される。EU データゾーンは EU Data Boundary に従い、EU の加盟国のほか、ノルウェーやスイスなどの EFTA (European Free Trade Association: 欧州自由貿易連合) の国を含むことがある
Azure の地域 (Standard、Regional Provisioned)	指定した Azure の地域 (geography) の中で処理される。運用上の理由で、同じ地域の別のリージョンで処理されることがある

主な組み合わせは次の8つ。使える種類は、モデルとリージョンによって異なる。

- Global: Global Standard、Global Provisioned、Global Batch
- Data Zone: Data Zone Standard、Data Zone Provisioned、Data Zone Batch
- Azure の地域: Standard、Regional Provisioned

このほか、ファインチューニングしたモデルの評価用の Developer がある (SLA もデータの所在の保証も無い)。Developer 以外では、保存されるデータ (保存時のデータ) は指定した Azure の地域 (geography) に留まる。Global や Data Zone で変わるのは、推論の処理が行われる場所である。

要件	選ぶデプロイの種類
特に制約が無い (既定)	Global Standard
EU データゾーンの中でデータを処理したい	Data Zone Standard (EU データゾーンのリージョンで)
データの処理場所の制約 + 確保した処理能力	Data Zone Provisioned

要件	選ぶデプロイの種類
急がない大量の処理を安く	Global Batch または Data Zone Batch

★ **試験ポイント**: 条件が2つ並んだら、場所 (Global / Data Zone / Azure の地域) と方式 (Standard / Provisioned / Batch) を別々に決める。「急がない・大量・安く」→ Batch、「大量で安定・予測できる性能」→ Provisioned、「EU データゾーンの中で処理」→ Data Zone。

TPM とスロットリング

TPM は、そのデプロイに割り当てる1分あたりのトークン数のレート制限である。制限を超えたかどうかは、入力トークン数や最大出力トークン数から見積もった値で判定されるので、課金される実際のトークン数とは一致しないことがある。TPM に連動して、1分あたりのリクエスト数 (**RPM**: Requests Per Minute) の上限も決まり、比率はモデルによって異なる。画像のモデルなど、TPM ではなく容量の単位で動くモデルもある。

スロットリング とは、システムが安定を保つために処理量を制限することである。アプリが送るプロンプトが長かったり、最大出力トークン数を大きく設定していたりすると、1回のリクエストが TPM の見積もりを多く使う。上限を超えると、サービスは **429 Too Many Requests** のエラーを返す。短時間に集中したリクエストでも RPM の制限にかかる。アプリ側の基本の対処は、**最大出力トークン数を必要な量に下げる** ことと、**同時に送るリクエストの数を減らす** ことである。

💡 **実務での再試行** アプリは、429 が返ったら、応答のヘッダー **retry-after-ms** が示す待ち時間を守って再試行する。指定が無い場合は、待ち時間を段階的に延ばす **指数バックオフ** を使い、再試行の回数に上限を設ける。失敗したリクエストもレート制限に数えられるので、待たずに送り直すと制限が続く。リクエストは一度に集中させず、分散して送る。

実行時のパラメーター

アプリは、デプロイしたモデルを呼び出すときに、出力の性質を調整するパラメーターを指定できる。Foundry ポータルのプレイグラウンドでも、コードでも同じ設定ができる。

パラメーター	役割	調整の目安
temperature	出力のランダムさ (創造性と決定性のバランス)	低い (0 付近) : 焦点が絞られた、毎回似た回答。高い: 多様で創造的な回答
max output tokens (最大出力トークン数)	回答の長さの上限。推論モデルでは、推論のトークンも含めた上限になる	大きくするとトークンの消費が増え、スロットリングの原因にもなる
system instructions (システム指示)	モデルの役割、口調、制約を決める	「あなたは〇〇の担当者です」「契約の解釈は答えないこと」など
top_p	候補にするトークンを、確率の高いものから累積の確率で絞り込む	temperature と同じく多様性を調整する。ふつうは temperature と top_p のどちらか一方だけを変える

推論モデルの多くは、temperature や top_p に対応していない (GPT-6 Astra のように対応するものもあるので、使うモデルのドキュメントで確かめる。2026年10月時点)。

★ **試験ポイント**: 回答のばらつきを減らしたい → temperature を下げる。レート制限のエラー → 最大出力トークン数を下げる、同時のリクエストを減らす。

1-3. AI ワークロード

一般的な AI ワークロードのシナリオ

学習ガイドは、一般的な AI ワークロードとして、生成 AI とエージェント型 AI、テキスト分析、音声、コンピュータービジョン、情報抽出を挙げている。

ワークロード	何をするか	シナリオの例	学習パスでの主な実装手段
生成 AI	プロンプトに応じて新しいコンテンツ (文章・画像・コードなど) を作る	質問に答えるチャットボット、文書のたたき台の作成、複雑な文書の要約や解説、翻訳	Foundry のモデル (GPT など)
エージェント型 AI	目的に向けて自分で手順を考え、ツールを呼び出して行動する	社内システムで空き状況を調べ、会議を予約するアシスタント	Foundry Agent Service
テキスト分析	テキストから意味や情報を読み取る	口コミの感情分析、共有する前のデータからの個人情報の伏せ字化、よくある質問に答える定型のチャットボット	汎用モデル、または Azure Language (言語検出・PII 検出など)
音声	音声をテキストにする、テキストを音声にする	話しかけて作業させるエージェント、通話や会議の文字起こし、動画の音声解説、音声翻訳	Azure Speech、音声を入力できるマルチモーダルモデル
コンピュータービジョン	画像や動画の内容を理解する、画像を作る	写真の自動キャプションやタグ付け、ビジュアル検索、店頭在庫の監視、顔認識による認証	ビジョン対応のマルチモーダルモデル、画像生成モデル
情報抽出	文書・画像・音声・動画から、決まった項目を構造化データとして取り出す	申込書の自動処理、紙の書類のデジタル化、検索のための文書のインデックス作成、会議の記録からの要点の抽出	Azure Content Understanding

生成 AI は、「プロンプトを受け取って出力を返す」という1回のやり取りが基本である。**エージェント** は、生成 AI モデルの上に作られた「仕事をこなすアプリケーション」で、Foundry では次の3つの組み合わせとして定義される。

- **モデル**: 推論に使う生成 AI モデル (たとえば GPT-4.1)
- **指示 (Instructions)**: エージェントの役割、振る舞い、制約、出力のルールを決めるシステムプロンプト
- **ツール**: エージェントが使える機能。情報を取りに行く **ナレッジツール** (検索エンジンやデータベースなど) と、作業を実行する **アクションツール** (システムの更新、コードの実行など) がある

モデルは素の知能、エージェントはその知能の上に作られたタスク志向の働き手、と整理できる。エージェントは、外部のツールを自動で呼び出す、目標を手順に分ける、会話中の作業の記憶を保つ、入力を処理して次の行動を決める、といったことができる。それぞれ得意分野を持つ複数のエージェントが、プロンプトでやり取りしながら協力する構成を **マルチエージェントシステム** と呼ぶ(学習ガイドの実装の項目は、単一のエージェント)。

観点	生成 AI モデル単体	エージェント
基本の動き	プロンプト → 出力	目的 → 手順を考える → ツールで行動 → 結果
外部への作用	なし(テキストなどを返すだけ)	ツールでファイルの読み書き、検索、システムの更新などを行う
再利用	毎回プロンプトを組み立てる	モデル・指示・ツールをまとめて保存し、繰り返し使える

★ **試験ポイント**: 外部のシステムに作用する(調べて予約する、発注する)、複数の手順をこなす → エージェント型 AI。1回の入力に1回の出力 → 生成 AI。

一般的なテキスト分析の手法

自然言語処理(NLP) は、人の言葉をコンピューターに理解させる技術の分野で、LLM の土台にもなっている。今は多くの処理を生成 AI モデルがこなすが、予測できる結果が要るときや、独自のルールを当てはめたいときは、専用の NLP のツールが使われる。学習ガイドに挙がっている手法(キーワード抽出、エンティティ検出、センチメント分析、要約)とその仲間を押さえる。

手法	何をするか	例
キーワード抽出 (キーワード抽出)	文章の主な概念や論点を抜き出す	ホテルの口コミ「朝食のビュッフェが充実していて、駅から近い」→「朝食のビュッフェ」「駅」
エンティティ認識 (エンティティ検出、NER: Named Entity Recognition)	人名、地名、組織名、日時、数量などを見つけ、種類を判定する	「3月10日に名古屋の本社で」→「3月10日」は DateTime、「名古屋」は Location
エンティティリンク	見つけたエンティティを、既知の項目(Wikipedia の記事など)と結び付ける	「Mercury」が惑星の水星か元素の水銀かを文脈で判断し、該当する記事に結び付ける
テキスト分類	文書の内容に基づいてカテゴリに分ける	問い合わせを「請求」「配送」「故障」に分ける
感情分析 (センチメント分析)	テキスト分類の一種。文書や文が、肯定・否定・中立のどれかを判定する	SNS(ソーシャルメディア)の投稿、商品の口コミの評価
オピニオンマイニング	感情分析をさらに細かくし、何に対して肯定・否定なのかを特定する	「画面はきれいだが、電池の持ちが悪い」→ 画面は肯定、電池は否定
要約	長い文章の重要な情報をまとめる。元の文をそのまま選ぶ 抽出型 と、新しい文章で書き起こす 抽象型 がある	会議の記録を数行にまとめる
言語検出	どの言語で書かれているかを判定し、信頼度のスコアを付ける。多段階の処理の最初の手順になることが多い	フランス語の問い合わせ → fr と信頼度
PII 検出	エンティティ検出の特に専門的な形。個人を特定できる情報(氏名、電話番号、住所など)を見つけ、必要に応じて伏せ字にする	問い合わせの本文の氏名と電話番号を「***」に置き換える

PII(Personally Identifiable Information) は個人を特定できる情報のことで、Azure Language の PII 検出は、健康に関する個人情報(**PHI**: personal health information)も対象にできる。

分析する前の処理: 分析するテキストの集まりを **コーパス** と呼ぶ。統計的な分析の前には、次のような処理でテキストを整える。

処理	内容
トークン化	コーパスをトークンに分ける
正規化	句読点を除き、小文字にそろえる。頻度だけを見る分析では役立つが、意味の一部が失われることがある
ストップワードの除去	読みやすさには役立つが意味の薄い語を、分析から外す
n-gram	よく続けて現れる複数の語を、1つのまとまりとして扱う(1語はユニグラム、2語はバイグラム、3語はトライグラム)
ステミング	語尾を機械的に削って、同じ語根にまとめる
レンマ化	言語の規則と語彙を使い、実在する辞書の形(レンマ)に戻す
品詞タグ付け	各トークンに、名詞・動詞・形容詞・副詞などの文法上の種類を付ける

統計的な手法:

手法	内容
頻度分析	正規化したトークンの出現回数を数え、よく出る語から文書の主題を推測する。複数の文書が同じ語を多く含むと、文書どうしを区別しにくい
TF-IDF(Term Frequency-Inverse Document Frequency)	ある文書には多く、ほかの文書には少ない語ほど高いスコアにする。その文書での出現頻度(TF)と、その語を含む文書の少なさ(IDF)を掛け合わせる
Bag-of-words	文法と語順を無視し、テキストを語の出現頻度(または有無)のベクトルで表す。Naive Bayes などの分類器に入れ、文書の分類や感情分析に使う

手法	内容
TextRank	文をノード、文どうしの類似度を辺の重みにしたグラフで、「重要な文の多くと似ている文ほど重要」としてスコアを計算する。上位の文を選ぶ抽出型の要約に使う。語に当てはめればキーワード抽出にもなる

意味モデル:埋め込みは Word2Vec や GloVe などの手法で広まり、その後、アテンションで周りのトークンの影響を加えた **文脈化された埋め込み** (GPT のモデルなど) が、今の生成 AI の土台になった。意味モデルを使えば、要約、キーワード抽出、NER、分類も埋め込みで行える。

★ **試験ポイント**:**キーフレーズ** は文章の主な論点で、種類の分類はしない。**エンティティ** は人・場所・日時などの決まった種類に当てはまるもので、種類 (カテゴリ) が付く。対象ごとの評価まで取り出す → オピニオンマイニング。

音声認識と音声合成

機能	別名	方向	用途
音声認識	音声テキスト変換、 STT (Speech to Text)	音声 → テキスト	字幕、通話の文字起こし、口述筆記
音声合成	テキスト音声変換、 TTS (Text to Speech)	テキスト → 音声	応答の読み上げ、メッセージの読み上げ、館内放送

音声認識の流れ:代表的な処理を、次の6つの段階に分けて考える。中心になるのは **音響モデル** と **言語モデル** である。実際には複数の段階を統合したモデルもあり、すべての実装が独立した6部品を持つわけではない。

段階	内容
音声の取り込み	アナログの音声を1秒に数千回標本化して、デジタルにする
前処理	波形を、MFCC (Mel-Frequency Cepstral Coefficients:メル周波数ケプストラム係数) などの特徴のベクトルにまとめる
音響モデル	特徴から、各時点での 音素 (特定の音を表す単位) である確率が高いかを予測する。今の音響モデルの多くは Transformer を使う。音素は言語ごとに違う
言語モデル	語彙、文法、よくある語の並びの知識で、聞き分けにくい部分のあいまいさを解消する。音素の並びを単語に結び付ける役割を持つ。専門分野の用語で学習したカスタム言語モデルで、専門的な場面の精度を上げられる
デコード	音響モデルと言語モデルの両方に最もよく合う語の並びを選ぶ (ビームサーチ)
後処理	句読点や数字の書式を整え、読みやすくする。Azure Speech は、単語ごとのタイムスタンプと信頼度のスコアも返す

音声合成の流れ:代表的な処理を、次の4つの段階に分けて考える。

段階	内容
テキストの正規化	略語、数字、日付、記号を、読み上げる形に展開する。同じつづりで読みの違う語は、文脈で読みを決める
言語解析	語と音節に分け、発音の辞書などで音素に変え、強く読む音節を決める
韻律の生成	音の高さの変化、長さ、強さ、間を予測する。 韻律 (リズム、強勢、抑揚) が平板だと、機械的な声に聞こえる
音声の生成	音響モデルが音の特徴 (メルスペクトログラム) を作り、 ニューラルボコーダー が波形にする

音声合成には通常、**読み上げるテキスト** と **使う声** の2つが要る。声の種類、話す速さ、声の高さ (ピッチ)、音量も指定できる。ニューラルネットワークを使った **ニューラル音声** は、従来の音声合成で不自然になりがちだった抑揚を改善し、より自然な声を出せる。

音声認識と音声合成を組み合わせると、IVR (Interactive Voice Response:自動音声応答) のような双方向の対話になる。音声の技術は、背景の雑音を無視する、割り込みを検出する、より表現豊かで人に近い声を作る、といった課題に向けて進化している。音声のソリューションを作る側は、音声の品質、対応する言語と方言、音声データの扱い、遅延、アクセシビリティを事前に確かめ、テキストなど代替りの入出力の手段も用意する。

★ **試験ポイント**: 音声 → テキストは STT (認識)、テキスト → 音声は TTS (合成)。音声の特徴から音素の確率を出すのは音響モデル、語の知識であいまいさを解消し単語に結び付けるのは言語モデル。抑揚や間 → 韻律。

コンピュータービジョンと画像生成

コンピュータービジョン は、画像、動画、カメラの映像などの視覚的な入力を AI で処理する技術の分野である。古くからある代表的なタスクは次のとおり。

タスク	何をするか	出力のイメージ
画像分類	主な被写体のラベルを付けた大量の画像で学習したモデルで、画像の内容からラベルを予測する	「この画像はりんご」
物体検出	画像の中の複数の物体を見つけ、それぞれの位置を示す	物体ごとのラベルと、四角い枠 (バウンディングボックス) の座標
セマンティックセグメンテーション	画素 (ピクセル) ごとに、どの物体に属するかを分類する	物体の形に沿った塗り分け。物体の位置がより正確にわかる

コンピュータービジョンのモデルは、統計に基づく分類器から CNN (畳み込みニューラルネットワーク) を経て、今の Transformer に基づくマルチモーダルモデルへと進んできた。しくみの要点は次のとおり。

項目	内容
画像の表し方	コンピュータにとって画像は、ピクセルの値の配列である。グレースケールなら各ピクセルが0(黒)～255(白)の値を持ち、カラー画像の多くは赤・緑・青(RGB)の3つのチャネルの層でできている
フィルター	カーネル という小さな重みの配列を画像の上でずらし、周りのピクセルとの重み付きの和を計算して新しい画像を作る(畳み込み)。重みしだいで、エッジの強調、ぼかし、シャープ化などになる
CNN(Convolutional Neural Network: 畳み込みニューラルネットワーク)	フィルターで画像から特徴マップを取り出し、その値をニューラルネットワークに入れてラベルを予測する。物体検出のモデルは、CNNの特徴抽出と、画像の中の関心領域の特定を組み合わせる
ViT(Vision Transformer: ビジョントランスフォーマー)	画像をピクセルの小さな区画(パッチ)に分けてベクトルにし、言語モデルと同じアテンションで、パッチどうしの関係を捉える。埋め込みに入るのは、色、形、コントラスト、質感などの視覚的な特徴
マルチモーダルモデル	画像と説明文の組で学習し、言語のエンコーダーとViTのエンコーダーの埋め込みを、1つの共有の空間にまとめる。初めて見る画像でも、視覚的な特徴に近い言葉を探して説明文を作れる
拡散モデル(画像生成)	同じマルチモーダルのしくみを逆向きに使う。ランダムなピクセルの値(ノイズ)から始め、プロンプトと比べながらノイズを少しずつ取り除いて画像を作る
動画生成	視覚的な特徴に加えて、現実の物体の物理的なふるまいと、時間の流れに沿った一貫性も考慮する

マルチモーダルのビジョンモデルは、画像に何が写り、何が起きているかを意味から解釈し、説明文を作ったり、タグを提案したりできる。テキスト・画像・音声・動画など複数の種類のデータを扱えるので、画像を説明する、写真について質問に答える、グラフや図を読み解く、といったことを自然言語のやり取りでこなす。画像を理解して自然言語で答える GPT のモデルは「**ビジョン対応 GPT モデル**(GPT with vision)」とも呼ばれる。

方向	モデルの種類	できること
画像 → テキスト(理解)	マルチモーダルモデル(ビジョン対応)	画像の説明、画像についての質問応答、図や文書の読み取り
テキスト → 画像(生成)	画像生成モデル	テキストの説明から新しい画像を作る、画像を編集する
テキスト・画像 → 動画(生成)	動画生成モデル	テキストや画像から短い動画を作る

★ **試験ポイント**: 画像全体にラベル → 画像分類。物体ごとの位置(枠) → 物体検出。画素ごとの塗り分け → セマンティックセグメンテーション。フィルターで特徴マップを取り出す → CNN、パッチに分けてアテンション → ViT。ノイズから少しずつ画像を作る → 拡散モデル。

情報を抽出する手法

情報抽出 は、構造化されていないコンテンツ(文書、画像、音声、動画)から、必要な項目を構造化データとして取り出すワークロードである。文書から情報を取り出す処理は、2段階で進む。まずコンピュータビジョンの **OCR**(Optical Character Recognition: 光学文字認識)が、画像の中の文字を検出して取り出す。次に、機械学習(最近は生成 AI も)のモデルが、そのテキストを意味に基づいてデータの項目(フィールド)に対応付ける。OCR でわかるのは「どんな文字があるか」、フィールド抽出でわかるのは「その文字が何を意味し、どこに入るか」である。

段階	OCR	フィールド抽出
1	画像の取り込み(画質が精度を大きく左右する)	OCR の出力(テキスト、座標などの位置、信頼度、レイアウト)を受け取る
2	前処理(ノイズの除去、コントラストの調整、傾きの補正)	項目の候補を見つける
3	文字の領域の検出(レイアウトの解析、読む順序の決定)	項目に対応付ける(キーと値の組、表)
4	文字の認識(前後の文字や辞書を使い、文字ごとの信頼度を付ける)	値を正規化する(日付・通貨・数値の形式をそろえ、妥当性を確かめる)
5	出力の生成(文章への組み立て、元の画像の上の座標の記録)	業務システムに連携する

フィールド抽出は、書かれている内容だけでなく、文書の **どこに** 書かれているかにも大きく頼る。項目を見つける方法には、次の3つがある。

方法	しくみ	特徴
テンプレート型	既知のレイアウト、ラベルと値の組、正規表現で探す	既知の書式なら精度が高く、速く、結果を説明しやすい。テンプレートを手で作る必要があり、レイアウトのばらつきに弱い
機械学習型	例の文書で学習し、学んだ関係から抽出する	複数の書式を扱える。Transformer のモデルは、文脈の手がかりを使うのが得意
生成 AI 型	文書のテキストと スキーマ の定義を LLM に渡し、項目に対応付けさせる	プロンプトで抽出できる。フューショットの例なども使う

スキーマ とは、「何の情報を取り出したいか」と「それをどういう構造で返してほしいか」を定義したものである。スキーマに基づく抽出は、「注文番号」「合計金額」など定義した項目に、意味で値を当てはめて取り出すので、ラベルの表記が違ってても、ラベルが無くても扱える。これに対し、**基本的な OCR** は、文字の認識に前後の文字や辞書を使うものの、文字の意味、文脈、項目どうしの関係までは理解しない。

フィールド抽出では、取り出した値に信頼度を付け、関連する項目どうしの整合(小計と合計など)でも値を確かめる。精度が大事な用途では、作る側は、人が確かめる手順(human-in-the-loop)を入れることを考える。Azure Document Intelligence in Foundry Tools や Azure Content Understanding in Foundry Tools のようなサービスを使うと、開発の手間を減らせる。学習ガイドの第2領域で名前が挙がっているのは Content Understanding で、文書・画像・音声・動画のスキーマに基づく抽出を行う。

★ **試験ポイント**: 文字を読むだけ → 基本的な OCR。決まった項目を意味で取り出して構造化する → スキーマに基づく抽出。定型の書式 → テンプレート型、さまざまな書式 → 機械学習型、テキストとスキーマを LLM に渡す → 生成 AI 型。

✓ 第1章 チェック問題

- 書類選考を助ける AI が、特定の年齢層の応募者を不当に低く評価していた。主にどの原則の問題か。
- 画面の文字を読み上げる機能をアプリに加え、目の不自由な人も使えるようにした。どの原則に基づく対応か。
- AI の判断で問題が起きたときに、誰が対応し、どう報告するかを組織として決めておいた。どの原則か。
- チャットの画面に、回答は AI が生成したもので誤りを含みうることを表示した。どの原則か。
- 言語モデルがテキストを扱うとき、最初にテキストを分ける単位を何と呼ぶか。
- 社内の FAQ から、質問と意味の近い項目を探す機能を作りたい。どのタイプのモデルを使うか。
- 数十万件のログをまとめて要約する。完了を急がず、厳密な期限の保証よりコストの削減を優先したい。どのデプロイの種類が向くか。
- 同じ質問への回答のばらつきを減らしたい。temperature は上げるか、下げるか。
- レート制限のエラーが続いている。リクエストの中身と送り方を見直す対処を2つ挙げよ。
- Foundry の評価器で、出力に有害な内容が見つかった。評価器はそれを自動で直すか。
- 「画面はきれいだが、電池の持ちが悪い」という口コミから、項目ごとの評価を取り出したい。どの手法か。
- 音声認識で、音声を音素に変えるのはどのモデルか。
- 語尾を機械的に削って同じ語根にまとめる処理と、言語の規則で実在する辞書の形に戻す処理を、それぞれ何と呼ぶか。
- 「ある文書には多く出るが、ほかの文書にはあまり出ない語」ほど高いスコアを付ける手法を何と呼ぶか。
- 写真の中のどの画素が、どの物体に属するかを分類するタスクを何と呼ぶか。
- ランダムなノイズから始め、プロンプトと比べながら少しずつ画像を作る手法を何と呼ぶか。
- 在庫の減った商品を社内システムで調べ、発注の依頼まで自動で作る AI を作りたい。どのワークロードか。
- 取引先ごとに様式の違う注文書から、注文番号と合計金額を取り出したい。基本的な OCR だけで足りるか。

▼ 解答

- 公平性 (特定の属性の人に不利な結果が出ている)
- 包括性 (使える人の範囲を広げている)
- 説明責任 (人と組織が結果に責任を持つ体制)
- 透明性 (利用者が AI のしくみと限界を理解できるようにしている)
- トークン
- 埋め込みモデル (文章をベクトルに変え、意味の近さを測る)
- Global Batch (データの処理場所に制約があれば Data Zone Batch)。Global Standard より50%安く、24時間以内の完了が目標だが、保証ではない。翌朝などの厳密な期限がある場合は、この目標だけで Batch を選ばない
- 下げる (top_p を下げても近い効果がある。ふつうは temperature と top_p のどちらか一方だけを変える)
- 最大出力トークン数を下げる、同時に送るリクエストの数を減らす (429 が返ったときは、`retry-after-ms` を守って再試行する)
- 直さない。評価器そのものは検出・採点するだけ
- オピニオンマイニング
- 音響モデル (音素を単語に対応付けるのは言語モデル)
- ステミングとレンマ化
- TF-IDF
- セマンティックセグメンテーション
- 拡散 (拡散モデル)
- エージェント型 AI (ツールで外部のシステムに作用し、複数の手順をこなす)
- 足りない。基本的な OCR は文字を読むだけで、意味や項目の関係を理解しない。スキーマに基づく抽出 (Azure Content Understanding) を使う

第2章 | Microsoft Foundry を使用して AI ソリューションを実装する (55～60%)

この章では、Foundry の基本と、4つの実装の項目（生成 AI アプリとエージェント、テキストと音声、ビジョンと画像生成、情報抽出）を、ポータルでの操作と Python のコードの両方で押さえる。

💡 **コードの読み方** 学習ガイドの注記には「REST API、SDK、CLI に慣れていること」とある（英語版は "You should be familiar with REST APIs, SDKs, and CLIs."）。この章のコードは、どの SDK も「クライアントを作る → メソッドを呼ぶ → 結果を読む」の3段で読む。

2-1. Foundry の基本(第2章の土台)

この節は学習ガイドの個別の項目ではないが、このあとの実装の前提になる。

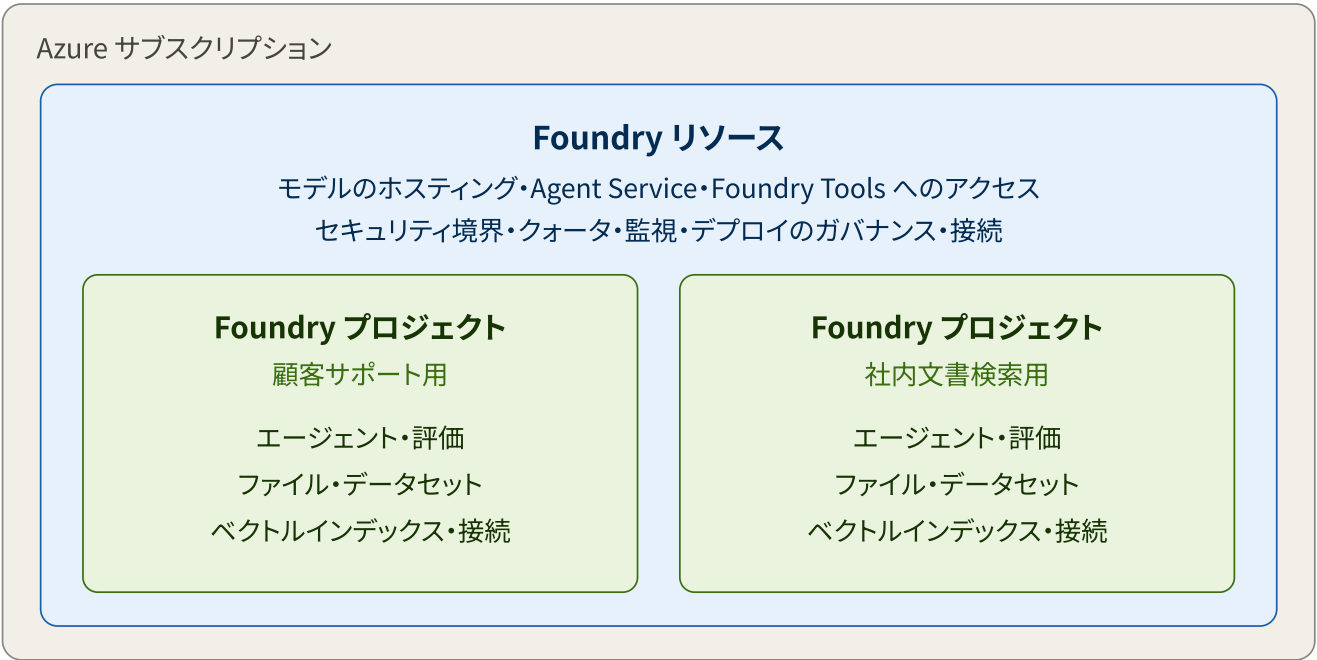
Foundry の構成要素

Microsoft Foundry は、AI アプリとエージェントを構築・デプロイ・管理するための、エンタープライズ向けの統合された PaaS (Platform as a Service) である。モデル、エージェントの実行、監視、ガバナンスの機能を1か所にまとめている。中心になる要素は次の4つ。

要素	内容
モデル	モデルカタログから選んでデプロイする。OpenAI、Anthropic、Mistral、Meta、DeepSeek などのモデルがある
エージェント	モデル・指示・ツールを組み合わせた、タスク志向の AI。 Foundry Agent Service で構築・実行する
ツール	エージェントやアプリに組み込む機能。Azure Language、Azure Speech などの Foundry Tools もここに入る
ナレッジ	エージェントに組織のデータを参照させるしくみ。 Foundry IQ （一部の機能はプレビュー）が、複数のデータソースをまとめた知識ベースを提供する

Foundry リソースと Foundry プロジェクト

Foundry を使い始めるには、Azure サブスクリプションの中にもまず **Foundry リソース** を作り、その中に **Foundry プロジェクト** を作る。たとえば、1つの Foundry リソースの下に、顧客サポート用と社内文書検索用のプロジェクトを分けて作る。



観点	Foundry リソース	Foundry プロジェクト
正体	Azure のリソース	リソースの中の作業スペース
持つもの	モデルのホスティング、Agent Service、Foundry Tools へのアクセス、セキュリティ境界、クォータ、監視、デプロイのガバナンス、すべてのプロジェクトで使える接続	エージェント、評価、ファイル、データセット、ベクトルインデックス、そのプロジェクトの接続
典型的な単位	チームや部門に1つ	AI のユースケースごとに1つ

このノートで扱う言語検出・PII 検出 (Azure Language)、音声認識・音声合成 (Azure Speech)、情報抽出 (Azure Content Understanding) は、1つの Foundry リソースから使える。これらの機能のために、サービスごとの専用のリソースを別に作る必要はない。使う機能が、そのリージョンで提供されているかは確かめる。

★ **試験ポイント**：ユースケースごとに分ける単位は **プロジェクト**。セキュリティ境界、クォータ、モデルのホスティングを持つのは **リソース**。このノートで扱う Azure Language や Azure Speech の機能を使うために、サービスごとの別のリソースは要らない。

Foundry ポータル

Foundry ポータル は、モデルやエージェントを開発・テスト・運用する Web の画面である。クラシック と 新しい ポータルの2種類があり、学習パスは新しいポータルを前提にしている。新しいポータルでは、Build → Models → AI services タブと進むと、Azure Language や Azure Speech などの Foundry Tools をブレイグラウンドで試せる。

ポータルでの典型的な流れは次のとおり。

- 1. Foundry ポータルにサインインし、Foundry プロジェクトを作る
- 2. モデルカタログからモデルを選んでデプロイする
- 3. ブレイグラウンド で、プロンプトやパラメーターを試す
- 4. 調整したモデルを、自分のクライアントアプリから呼び出す

クライアントとサーバー

AI アプリは クライアント・サーバー 型で動く。クライアントアプリ は、ユーザーが操作するプログラム（スマホアプリ、ブラウザー、コマンドラインのツールなど）で、サーバーにリクエストを送って結果を表示する。Foundry では、サーバーにあたるのは モデルのデプロイ である。

クライアントがすること	サーバー（モデルのデプロイ）がすること
画面やコマンドラインを用意し、ユーザーの入力（テキスト、音声、画像）を受け取る	プロンプトを受け取り、モデルで推論する
入力をプロンプトや API のリクエストの形に整えて送る	システム指示、安全性の設定、文脈を当てはめる
返ってきた結果を表示する	生成した出力（テキスト、画像、音声、JSON）を返す

軽量クライアントアプリ とは、ユーザーの入力を集め、リモートのサービスを呼び出し、結果を表示することに絞った小さなアプリである。重い計算はサーバー側（モデル）が行う。学習ガイドの第2領域には「軽量アプリを作る」という項目が6つある（チャット、エージェント、テキスト分析、音声、ビジョン、情報抽出）。

エンドポイントと認証

アプリは、Foundry の機能を、ネットワーク経由の API として呼び出す。エンドポイント は、サービスの入口になる URL で、人がブラウザーで開くのではなく、アプリのコードから呼び出す。

種類	用途
プロジェクトのエンドポイント	Foundry プロジェクトと、その中のエージェントなどを操作する
モデルのエンドポイント	デプロイしたモデルにプロンプトを送る

学習パスのコード例に出てくるエンドポイントの形は次のとおり。

URL の形	接続先
https://<リソース名>.services.ai.azure.com/api/projects/<プロジェクト名>	Foundry プロジェクト（エージェントの呼び出しなど）
https://<リソース名>.openai.azure.com/openai/v1/	OpenAI 互換の API（モデルの呼び出し）
https://<リソース名>.cognitiveservices.azure.com/	Foundry Tools（Azure Language など）
https://<リソース名>.services.ai.azure.com/	Foundry リソース（Content Understanding など）

 **ドメインだけで決めつけない**：同じ学習パスの中に、モデルの呼び出しを `https://<リソース名>.cognitiveservices.azure.com/openai/deployments/...` の形で示す例もある。接続先は、ドメインとパス（`/api/projects/`、`/openai/` など）を合わせて読む。

エンドポイントは保護されていて、アプリが呼び出すには、次のどちらかの認証が要る。

認証の方法	しくみ	特徴
API キー	ランダムな長い文字列をリクエストに含める	手軽。漏れると誰でも使ってしまうので管理に注意
Microsoft Entra ID	Entra ID の資格情報で取ったトークンを、 <code>Authorization: Bearer <アクセス トークン></code> のヘッダーで送る	キーを扱わずに済む。ロールベースのアクセス制御（RBAC：Role-Based Access Control）で権限を細かく管理できる

Python の SDK での指定の仕方は、SDK によって違う。

- Azure の SDK（Azure Language、Content Understanding など）：キーは `AzureKeyCredential(key)`、Entra ID は `DefaultAzureCredential()`
- OpenAI の SDK：API キーを使う場合はキーの文字列を `api_key=` に渡す。Entra ID を使う場合も、`get_bearer_token_provider()` で作ったトークンプロバイダーを `api_key=` に渡せるので、引数名だけでは認証の方法を決められない
- Speech SDK：キーの文字列を `subscription=` に渡す

`DefaultAzureCredential` は、環境に合った資格情報を自動で探して使う。開発中なら Azure CLI のサインイン情報、Azure 上で動かすならマネージド ID、という具合である。

API キーのように、システムへのアクセス権を与える秘密の値を **シークレット** と呼ぶ。Microsoft は、まず Entra ID によるキーレスの認証（たとえばマネージド ID）を推奨している。API キーを使う場合は、コードや GitHub に書かず、**Azure Key Vault** に保存し、アプリは実行時に **マネージド ID** で取り出すのが推奨される方法である。

★ **試験ポイント**: Entra ID のアクセストークンや `DefaultAzureCredential` → Entra ID。 `AzureKeyCredential`、キーを渡す `api_key=`・`subscription=` → API キー。
`api_key=` だけで判断せず、渡している値を見る。キーを使う場合の安全管理 → Azure Key Vault + マネージド ID (キーを使わずに済むなら Entra ID のキーレスの認証)。

REST と SDK と CLI

アプリは、エンドポイントに **REST** (Representational State Transfer) の形で HTTP リクエストを送る。リクエストでは、操作を表すメソッド (`GET`、`POST` など)、対象の URL、メタデータ (認証情報やデータの形式) を入れる **ヘッダー** を指定し、必要な操作では **本文 (ボディ)** も送る。AI の API では JSON の本文をよく使うが、本文を送らない操作や、画像・音声などのバイナリーを返す操作もある。

```
curl -X POST "https://<リソース名>.services.ai.azure.com/api/projects/<プロジェクト名>/openai/responses?api-version=2025-11-15-preview" \
-H "Content-Type: application/json" \
-H "Authorization: Bearer <アクセストークン>" \
-d '{
  "model": "<デプロイ名>",
  "input": "Foundry プロジェクトの役割を一文で説明して"
}'
```

`curl` は、コマンドラインから HTTP のリクエストを送るツールである。上の例は学習パスの例と同じ形で、プレビュー版の `api-version` を指定して REST を直接呼んでいる。2026 年 10 月時点の製品ドキュメントのクイックスタートは、`api-version` を付けない `.../api/projects/<プロジェクト名>/openai/v1/responses` の形を使っている。多くの開発者は REST を直接書かず、**SDK** (ソフトウェア開発キット) を使う。SDK は、Python や C# などの言語から REST を呼ぶコードを代わりに組み立ててくれるライブラリである。

CLI (コマンドラインインターフェイス) は、コマンドでサービスを操作するツールで、Azure なら **Azure CLI** がある。Azure CLI でサインイン (`az login`) しておくと、`DefaultAzureCredential` がその資格情報を使う。

Python 開発の基本作法

パッケージのインストール: Visual Studio Code などのコードエディターの **ターミナル** (エディターの中のコマンドライン) で `pip install <パッケージ名>` を実行する。

環境変数と .env ファイル: エンドポイント、キー、デプロイ名は、コードに直接書かず `.env` ファイルに保存する。`.env` は秘密の値を含むので、Git などのバージョン管理に入れない。

```
OPENAI_ENDPOINT=https://<リソース名>.openai.azure.com/openai/v1/
FOUNDRY_API_KEY=<Foundry リソースのキー>
CHAT_DEPLOYMENT=<チャットモデルのデプロイ名>
```

```
import os
from dotenv import load_dotenv

load_dotenv() # .env の内容を環境変数に読み込む
endpoint = os.getenv("OPENAI_ENDPOINT") # 名前を指定して値を取り出す
```

クライアントオブジェクト: コードでは、SDK が用意した **クラス** (設計図) に、エンドポイントと資格情報を渡して **オブジェクト** (実体) を作る。これを **クライアントオブジェクト** と呼び、コードはそのメソッドを呼んでサービスを使う。

このノートのコード例は、必要な環境変数が設定済みであることを前提にしている。`.env` だけに値を書いた場合は、各プログラムで `load_dotenv()` を呼ぶ。`os.getenv()` 自体には、`.env` を読む機能は無い。`example.com` のファイル URL は説明用であり、実行するにはサービスからアクセスできる実際の URL に置き換える。

★ **試験ポイント**: `.env` に書いた名前と `os.getenv()` に渡す名前は完全に一致させる。一致しないと `os.getenv()` は `None` を返し、クライアントの作成や呼び出しの段階でエラーになる。

2-2. 生成 AI アプリとエージェントを実装する

システムプロンプトとユーザープロンプト

生成 AI モデルに送るプロンプトには、役割の違う 2 種類がある。

種類	誰が書くか	役割	例
システムプロンプト (システム指示)	アプリの開発者	アシスタントの振る舞い、口調、使うツール、守るべき制約 (ガードレール) を決める	「あなたは社内の経費精算の窓口です。規程に書かれていないことは推測せず、担当部署を案内してください」
ユーザープロンプト	エンドユーザー (アプリがユーザーに代わって作ることもある)	実際の質問や依頼	「出張の新幹線代は、いつまでに申請すればいい？」

システムプロンプトは会話全体に効く前提条件、ユーザープロンプトは毎回の依頼内容と考える。会話の一貫性を保つため、生成 AI のアプリは会話の履歴を記録し、要約したものを次のプロンプトに含めることが多い。ユーザーが追加の質問を送るときも、アプリが前のプロンプトと応答と一緒にモデルに渡すので、モデルは前の流れを踏まえて答えられる。効果的なプロンプトの書き方は次のとおり。

コツ	内容	例
具体的に書く	あいまいな依頼を避け、目的と条件を書く	×「まとめて」○「この議事録から決定事項だけを3つの箇条書きで」
役割を与える	システムプロンプトで立場を決める	「あなたは新入社員向けの IT ヘルプデスクです」
出力の形式を指定する	箇条書き、表、JSON など、欲しい形を伝える	「結果を JSON で返して。キーは category と reason」
例を示す	入力と出力の例を見せる。例が1つなら ワンショット 、複数なら フューショット	「例: 入力『最高!』→ 出力『肯定』」
文脈を与える	話題、対象の読者、欲しい形式、判断に必要な背景や参照するデータを伝える	「新入社員向けに」と読者を書く、社内規程の該当箇所をプロンプトに入れる
区切りを明確にする	指示と、処理する対象のテキストを分ける	--- などの記号で対象のテキストを囲む
逃げ道を用意する	答えられないときの振る舞いを決める	「わからない場合は推測せず『わかりません』と答えて」

汎用モデルは自然言語の指示で動くので、**指示が具体的なほど、構造化された詳しい結果が返る**。指示があいまいだと、出力の形式も内容もばらつく。

★ **試験ポイント**: 振る舞い・口調・制約 → システムプロンプト。毎回の質問や依頼 → ユーザープロンプト。例を1つ見せる → ワンショット、複数見せる → フューショット。

Foundry ポータルでモデルをデプロイして試す

ポータルでは、モデルカタログからモデルを選んでデプロイし、そのまま **プレイグラウンド** (Foundry Playgrounds) で試せる。プレイグラウンドでできることは次のとおり。

- ・ システム指示、temperature、最大出力トークン数などを設定する
- ・ チャットの画面でプロンプトを送り、応答を確かめる
- ・ 複数のモデルを比べる
- ・ うまくいった設定を、そのまま動く **コードとして表示** する

プレイグラウンドは、ポータルでの試行とコードでの実装の橋渡しになる。表示されるコードは、OpenAI 互換の **Responses API** でデプロイを呼び出すもので、試したシステムプロンプト、ユーザープロンプト、パラメーターの値をそのままコードに持ち込める。

★ **試験ポイント**: プレイグラウンドで調整した設定をアプリに移すいちばん手早い方法 → プレイグラウンドのコード表示で、Responses API を使うサンプルコードを取り出す。

Foundry SDK で軽量チャットクライアントを作る

Responses API は、Azure OpenAI でモデルとやり取りするための、新しい統一された API である。チャット補完 (Chat Completions) と Assistants API の機能を1つにまとめたもので、テキスト生成だけでなく、画像の入力やツールの呼び出しにも対応している。

Python からは **OpenAI Python ライブラリ** (`openai` パッケージ) で呼び出す。生の HTTP リクエストを書かずに、Python のコードでモデルを呼べる。

Foundry SDK は、2種類のクライアントを提供している。多くのアプリでは両方を使う。

クライアント	パッケージ	用途
プロジェクトクライアント (<code>AIProjectClient</code>)	<code>azure-ai-projects</code>	Foundry 固有の操作 (エージェントの取得など)
OpenAI 互換クライアント	<code>openai</code>	Responses API で、モデルやエージェントを呼び出す

デプロイしたモデルに Responses API でプロンプトを送る、最小限のチャットクライアントの例を示す。

```
# pip install openai python-dotenv
import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()

# ① クライアントオブジェクトを作る（エンドポイント+キー）
client = OpenAI(
    base_url=os.getenv("OPENAI_ENDPOINT"), # https://<リソース名>.openai.azure.com/openai/v1/
    api_key=os.getenv("FOUNDRY_API_KEY"),
)

# ② Responses API でプロンプトを送る
response = client.responses.create(
    model=os.getenv("CHAT_DEPLOYMENT"), # モデル名ではなく「デプロイ名」
    input=[
        {"role": "system", "content": "あなたは社内の IT ヘルプデスクです。手順は番号付きで短く答えてください。"},
        {"role": "user", "content": "VPN につながらないとき、最初に確かめることは?"},
    ],
    max_output_tokens=400, # 回答の長さの上限
    temperature=0.2, # 低くして回答をそろえる
)

# ③ 生成されたテキストを表示する
print(response.output_text)
```

コードを読むときのポイントは次のとおり。

- `client.responses.create()` が Responses API の呼び出し
- `model=` に渡すのは **デプロイ名**
- `input=` には、文字列をそのまま渡すことも、`role` (system/user) 付きのメッセージのリストを渡すこともできる
- 結果のテキストは `response.output_text` で取り出す
- `max_output_tokens` と `temperature` は、プレイグラウンドで調整したパラメーターと同じもの。GPT-5 などの推論モデルの多くは `temperature` に対応していないので、そのデプロイを使うときは `temperature` を外す
- この例は `OpenAI` クラスを直接作っている。`AIProjectClient` の `get_openai_client()` で取り出した OpenAI 互換クライアントからも、同じように `model=` にデプロイ名を渡してモデルを呼べる

Foundry ポータルで単一エージェントを作る

ポータルでエージェントを作る手順は、プレイグラウンドでモデルを試すのと同様である。

1. エージェントが使う **モデル** を選ぶ
2. **システム指示** を書く(例:「あなたは社内の備品の貸し出しを案内するアシスタントです。答えは短い箇条書きで返してください」)
3. **ツール** と **ナレッジ** を追加する
4. 名前を付けて、エージェントとして保存する
5. プレイグラウンドでテストし、指示やツールを調整する

モデル単体の利用とエージェントを分けるのは、3つ目の **ツール** と **ナレッジ** である。ポータルでは、**ツール=アクション**、**ナレッジ=文脈** と整理して、別々に追加する(概念のうえでは、ツールを、情報を取りに行くナレッジツールと、作業を行うアクションツールに分けることもある)。

ツール (Tools)

ツールは、モデルが外部のシステムを呼び出して **行動する** ための機能である。モデルは使えるツールの一覧を確かめ、ユーザーの依頼に関係があると判断したときにツールを呼び出す。

- **コードインタープリター**: コードを実行して、データ分析やファイルの処理を行う
- **ナレッジソースの利用**: 文書を検索する
- **カスタム関数や API**: 自社システムの API など呼び出す
- **MCP サーバー**: MCP (Model Context Protocol) で公開された外部の機能を呼び出す

ツールは **Foundry Tool Catalog** で一元的に探して管理できる。

MCP (Model Context Protocol) は、AI エージェントが外部のツールやデータソースにつながるための **オープンな標準** である。万能のアダプターのようなもので、サービスごとに個別の連携コードを書かなくても、MCP サーバーにつなげば、そのサーバーが公開している機能を決まった方法で使える。

役割	担当	動き
MCP クライアント	AI エージェント(またはエージェントを動かすアプリ)	リクエストを送り、結果を受け取る

役割	担当	動き
MCP サーバー	ツール・データ・アクションを公開するサービス	リクエストを受けて機能を実行し、構造化された結果を返す

エージェントは MCP サーバーにつなぐと、そのサーバーがどんなツールを持っているかを自動で調べ、必要に応じて呼び出す。

ナレッジ (Knowledge)

ナレッジは、社内の PDF、SharePoint、Azure Storage のファイルなどの **外部のコンテンツをエージェントに参照させる** しくみで、RAG で実現される。Foundry は次の流れで処理する。

1. コンテンツを取り込み、インデックスを作る
2. 質問に関係する内容を検索し、回答の根拠 (グラウンディング) にする
3. より正確で、根拠をたどれる、その分野に即した回答を返す

ナレッジを使った回答には、参照した情報源の **引用 (出典)** が付く。

ナレッジを実装する方法の1つが **Foundry IQ** である (2026年10月時点で、一部の機能は一般提供、Foundry ポータルの agentic retrieval はプレビュー)。Foundry IQ は、Azure Blob Storage、SharePoint、OneLake、Web などの複数のソースをまとめた知識ベースを作る。インデックスを作るタイプのソースでは、Foundry IQ がチャンク化やベクトルの埋め込みを自動で行う。対応するソースと構成では、Foundry IQ はユーザーの権限や Microsoft Purview の秘密度ラベルを守り、そのユーザーが見てよい情報だけを返す。ただし、RAG に Foundry IQ が必須なわけではない。

★ **試験ポイント**: 外部のシステムを呼び出して行動させる → ツール。社内文書を参照させ、出典付きで答えさせる → ナレッジ (RAG)。MCP では、エージェントがクライアント、ツールを公開する側がサーバー。

エージェント用の軽量クライアントアプリを作る

ポータルで作ったエージェントは、**Foundry Projects SDK** (`azure-ai-projects`) でアプリから呼び出せる。アプリは、プロジェクトのエンドポイントに接続し、**Project API** を通してエージェントを使う。これで、エージェントを Web アプリ、ボット、バックエンドの処理に組み込める。

```
# pip install "azure-ai-projects>=2.3.0" azure-identity
import os
from azure.identity import DefaultAzureCredential
from azure.ai.projects import AIProjectClient

# ① プロジェクトのエンドポイントに接続する (認証は Entra ID)
project_client = AIProjectClient(
    endpoint=os.getenv("PROJECT_ENDPOINT"), # https://<リソース名>.services.ai.azure.com/api/projects/<プロジェクト名>
    credential=DefaultAzureCredential(),
)

# ② ポータルで作ったエージェントを名前で取得する
agent = project_client.agents.get(agent_name="equipment-helper")

# ③ プロジェクトクライアントから、OpenAI 互換のクライアントを取り出す
openai_client = project_client.get_openai_client()

# ④ Responses API で、エージェントを指定して呼び出す
response = openai_client.responses.create(
    input=[{"role": "user", "content": "ノートパソコンを1週間借りたい。手続きは?"}],
    extra_body={"agent_reference": {"name": agent.name, "type": "agent_reference"}},
)

print(response.output_text)
```

コードを読むときのポイントは次のとおり。

- エージェントを呼ぶときは、まず `AIProjectClient` を作り、`get_openai_client()` で OpenAI 互換のクライアントを取り出す
- エンドポイントは **プロジェクトのエンドポイント** (`.../api/projects/<プロジェクト名>`)
- 認証は `DefaultAzureCredential()` (Entra ID)。`AIProjectClient` が対応している認証は、現時点では Entra ID だけである
- エージェントは、`extra_body` の `agent_reference` に、`"type": "agent_reference"` 付きで名前を渡して指定する。製品ドキュメントのクイックスタートのように、`project_client.get_openai_client(agent_name="...")` で、はじめからエージェントに結び付いたクライアントを取り出す書き方もある (この場合、`extra_body` は要らない)
- エージェントの識別情報は、ポータルのプレイグラウンドでコード表示に切り替え、環境変数 (`.env`) の欄で確かめられる

⚠ **学習パスのコードは古い書き方**: 学習パスのコード例は、`extra_body={"agent": {"name": "...", "type": "agent_reference"}}` と、キーを `agent` にした形で書かれている。これは `azure-ai-projects` 2.0 のベータ版の書き方で、2.0.0b4 以降はキーが `agent_reference` に変わった。どちらの形でも、`"type": "agent_reference"` でエージェントを指していることは同じ。

観点	モデルを直接呼ぶ(チャットクライアントの例)	エージェントを呼ぶ
接続先	OpenAI 互換のエンドポイント	プロジェクトのエンドポイント
作るクライアント	<code>OpenAI(...)</code>	<code>AIProjectClient(...)</code> → <code>get_openai_client()</code>
指定するもの	<code>model=</code> にデプロイ名	<code>extra_body</code> の <code>agent_reference</code> (または <code>get_openai_client(agent_name=...)</code>) にエージェント名
システム指示やツール	必要に応じてコードの中で指定する	エージェントの定義に含まれているので、コードでは指定しない

★ **試験ポイント**: `AIProjectClient` → `get_openai_client()` → `agent_reference` の流れはエージェントの呼び出し。`OpenAI(...)` で作って `model=` にデプロイ名を渡すのはモデルの呼び出し。

2-3. テキストと音声の AI ソリューションを実装する

テキスト分析を含む軽量アプリを作る

Foundry では、テキスト分析を2つの方法で実装できる。どちらを選ぶかが、この節の大事な判断になる。

観点	汎用 AI モデル	Azure Language in Foundry Tools
正体	大量のテキストで学習した言語モデル(GPT など)	特定のテキスト分析タスク専用の分析器を持つサービス
使い方	自然言語のプロンプトで指示する	タスクごとに用意されたメソッドを呼ぶ
出力	自由な文章。プロンプトしだいで形が変わり、同じ入力でも呼び出しごとに表現が変わることがある	構造化された結果(言語コード、信頼度、抽出したエンティティなど)。 決定的 (同じ入力なら同じ結果)
得意なこと	柔軟な分析、複数のタスクの組み合わせ(翻訳してから要約、など)、追加の質問	言語コード、信頼度スコア、伏せ字にしたテキストなど、決まった形の値が要る処理
向いている場面	会話的・探索的な分析	自動化されたパイプライン など、一貫した結果が大事な処理
Python での呼び出し	<code>openai</code> パッケージ (Responses API)	<code>azure-ai-textanalytics</code> パッケージ

判断の目安は「**同じ入力なら同じ値(決定的な結果)が要るか**」。言語コードや伏せ字のような決まった値を、自動化パイプラインで一貫して得たいなら Azure Language、柔軟な分析や追加の質問なら汎用モデルを選ぶ。

汎用モデルの構造化出力: 汎用モデルにも、JSON Schema で出力の形を指定する **構造化出力**(structured outputs) がある。ただし、これは出力の形式を制約する機能で、内容の正しさや、同じ入力に同じ回答を返すことを保証するものではない。

汎用モデルでの分析: 汎用モデルは、追加の設定や学習なしに、プロンプトの指示だけでキーフレーズ抽出、エンティティ認識、感情分析、要約、翻訳、分類などをこなす。コードはチャットクライアントと同じで、Responses API にプロンプトを送るだけである。欲しい粒度(全体の感情だけか、文ごとの内訳か)はプロンプトで指定する。

```
# client と os は、チャットクライアントの例で作ったものを使う
response = client.responses.create(
    model=os.getenv("CHAT_DEPLOYMENT"),
    input=[
        {"role": "system", "content": "ロコミの感情を、文ごとに肯定・否定・中立で判定してください。"},
        {"role": "user", "content": "部屋は清潔だった。チェックインの待ち時間は長かった。"},
    ],
)
print(response.output_text)
```

Azure Language SDK での分析: Azure Language in Foundry Tools は、クライアントライブラリから使う。Python 版は **Azure Language Python SDK** で、パッケージは `azure-ai-textanalytics` である。学習パスのコード例は、言語検出と PII 検出を扱っている。

```
# pip install azure-ai-textanalytics python-dotenv
import os
from dotenv import load_dotenv
from azure.core.credentials import AzureKeyCredential
from azure.ai.textanalytics import TextAnalyticsClient

load_dotenv()

# クライアントを作る（エンドポイントは https://<リソース名>.cognitiveservices.azure.com/）
client = TextAnalyticsClient(
    endpoint=os.getenv("LANGUAGE_ENDPOINT"),
    credential=AzureKeyCredential(os.getenv("FOUNDRY_API_KEY")),
)

# --- 言語検出 ---
text = "Bonjour, je voudrais changer la date de ma réservation."
result = client.detect_language([text])[0]          # リストで渡し、結果もリストで返る
if result.is_error:
    raise RuntimeError(f"言語検出に失敗: {result.error.code}: {result.error.message}")
print(result.primary_language.name)                # 言語名
print(result.primary_language.iso6391_name)        # ISO 639-1 の言語コード
print(result.primary_language.confidence_score)     # 0～1 の信頼度

# --- PII 検出 ---
text = "Please contact Ken Tanaka at 555-0142 about order 8821."
result = client.recognize_pii_entities([text])[0]
if result.is_error:
    raise RuntimeError(f"PII 検出に失敗: {result.error.code}: {result.error.message}")
print(result.redacted_text)                        # 個人情報を伏せ字にしたテキスト
for entity in result.entities:
    print(entity.text, entity.category, entity.confidence_score)
```

コードを読むときのポイントは次のとおり。

- ・クライアントのクラスは `TextAnalyticsClient`、認証は `AzureKeyCredential(key)`
- ・メソッドは **テキストのリスト** を受け取り、結果もリストで返す。1件だけのときも `[text]` で渡し、`[0]` で結果を取り出す
- ・文書ごとに失敗することもある。`is_error` を確認してから成功時のプロパティを読み、失敗した文書では `error` のコードとメッセージを確認する
- ・言語検出の結果は `primary_language` の中に、言語名 (`name`)、言語コード (`iso6391_name`)、信頼度 (`confidence_score`) が入っている
- ・PII 検出の結果には、伏せ字にしたテキスト (`redacted_text`) と、見つかった個人情報の一覧 (`entities`) が入る。一覧の各要素には、テキスト、カテゴリ (`Person`、`PhoneNumber`、`Address` など)、信頼度が付く

同じ `TextAnalyticsClient` には、ほかのテキスト分析のメソッドもある (学習パスのコードに出てくるのは最初の2つ)。

メソッド	機能
<code>detect_language()</code>	言語検出
<code>recognize_pii_entities()</code>	PII 検出と伏せ字化
<code>extract_key_phrases()</code>	キーフレーズ抽出
<code>recognize_entities()</code>	エンティティ認識 (NER)
<code>recognize_linked_entities()</code>	エンティティリンク
<code>analyze_sentiment()</code>	感情分析 (<code>show_opinion_mining=True</code> でオピニオンマイニングも)

🔦 **Azure Language の機能の提供終了の予定** Azure Language のキーフレーズ抽出、感情分析とオピニオンマイニング、要約、エンティティリンクは、提供を終える予定が発表されている (2026年10月時点。エンティティリンクは2028年9月1日、ほかの3つは2029年3月31日)。言語検出、PII 検出、NER は続く。学習パスでも、キーフレーズ抽出や感情分析は汎用モデルで行う例として、Azure Language は言語検出と PII 検出の例として説明している。

エージェントに Azure Language を組み込む: エージェントは生成 AI モデルで言葉を理解・生成できるが、モデルだけでは決定的で構造化されたテキスト分析にならない。そこで、**Azure Language in Foundry Tools** をツールとして追加すると、一貫した結果を返すテキスト分析をエージェントに持たせられる。その入口が **Azure Language MCP サーバー** (プレビュー) である。このサーバーは、エンティティ認識、感情分析、言語検出などの Azure Language の機能を MCP の形で公開するマネージドサービスで、自分の環境で動かすローカル版もある。

Foundry ポータルでの手順は次のとおり。

1. モデルをデプロイし、エージェントとして保存する

2. プレイグラウンドのツールの検索で「Azure Language in Foundry Tools」を探して追加する
3. 接続の設定で **Foundry リソース名** を指定する
4. プロンプトで、ツールを使ってテキストを分析するよう指示する

エージェントのコードで Azure Language の REST API を直接呼んだり、認証のトークンを管理したりする必要はない。

★ **試験ポイント**: 「同じ入力なら同じ値(決定的)」「自動化パイプライン」「言語コード」「伏せ字」→ Azure Language。「柔軟に」「会話しながら深掘り」→ 汎用モデル。エージェントに Azure Language の機能を持たせる → Azure Language MCP サーバー (ツール名は Azure Language in Foundry Tools)。

マルチモーダルモデルで音声のプロンプトに応答する

音声を入力として受け取れる **マルチモーダルモデル** をデプロイすると、アプリは音声をそのままプロンプトとして渡せる。モデルは音声の内容を理解し、それに応じた回答を返す。

方式	流れ
音声認識してからモデルに渡す	Azure Speech で音声をテキストにし、そのテキストを言語モデルに送る。処理は2段階になる
マルチモーダルモデルに直接渡す	音声のデータとテキストの指示を、1つのリクエストで送る。音声認識の処理を別に組まなくてよい

アプリは、音声を **base64** (画像や音声のようなバイナリーデータを、文字だけで表す方式) でテキストにして、メッセージの中に入れる。製品ドキュメントの例では、Chat Completions API のメッセージに `input_audio` 型で音声 (base64 のデータと `wav` などの形式) を入れて送る (2026年10月時点)。

★ **試験ポイント**: 「デプロイしたマルチモーダルモデルで、音声のプロンプトに応答する」→ 音声を入力できるモデルに、音声のデータをテキストの指示と一緒に直接渡す。

Azure Speech で軽量アプリを作る

音声認識 (STT)

Azure Speech in Foundry Tools には、マイクや音声ファイルの音声をテキストにする **音声テキスト変換 API** (Speech to Text) がある。新しい Foundry ポータルでは、Build → Models → AI services タブの「Azure Speech - Speech to Text」のプレイグラウンドで、音声ファイルをアップロードするか、その場で録音して文字起こしを試せる。

アプリに組み込むときは **Speech SDK** を使う。SDK が、ネットワーク通信、認証、音声のストリーミング、結果の解析を代わりに行う。処理の流れは次のとおり。

1. アプリが Speech SDK を初期化する (エンドポイントと認証情報を渡す。認証はキーか Entra ID)
2. アプリが音声を取り込む (マイク、音声ファイル、ストリーム)
3. SDK が音声を Azure Speech に安全に送る
4. Azure Speech が、クラウドで音声を認識する
5. 認識されたテキストがアプリに戻る

```
# pip install azure-cognitiveservices-speech
import os
import azure.cognitiveservices.speech as speechsdk

# ① 接続設定 (Foundry リソースのキーとエンドポイント)
speech_config = speechsdk.SpeechConfig(subscription=os.getenv("FOUNDRY_API_KEY"),
                                       endpoint=os.getenv("SPEECH_ENDPOINT"))
speech_config.speech_recognition_language = "ja-JP"

# ② 音声の入力元 (既定のマイク)。ファイルなら AudioConfig(filename="meeting.wav")
audio_config = speechsdk.audio.AudioConfig(use_default_microphone=True)

# ③ 音声認識のオブジェクトを作る
recognizer = speechsdk.SpeechRecognizer(speech_config=speech_config, audio_config=audio_config)

# 1回分の発話を認識する場合
result = recognizer.recognize_once_async().get()
if result.reason == speechsdk.ResultReason.RecognizedSpeech:
    print(result.text)
elif result.reason == speechsdk.ResultReason.NoMatch:
    print("音声を認識できませんでした:", result.no_match_details)
elif result.reason == speechsdk.ResultReason.Canceled:
    details = result.cancellation_details
    raise RuntimeError(f"音声認識の中止: {details.reason}: {details.error_details}")

# 話し続ける音声を認識し続ける場合 (イベントで結果を受け取る)
recognizer.recognizing.connect(lambda evt: print("途中:", evt.result.text)) # 認識の途中経過
recognizer.recognized.connect(lambda evt: print("確定:", evt.result.text)) # 確定した結果
recognizer.canceled.connect(lambda evt: print("音声認識の中止:", evt))
recognizer.start_continuous_recognition()
input("Enter キーで終了します")
recognizer.stop_continuous_recognition()
```

コードを読むときのポイントは次のとおり。

- パッケージは `azure-cognitiveservices-speech`。`speechsdk` という名前で読み込むのが慣例
- `SpeechConfig` (接続設定) → `AudioConfig` (音声の入力元) → `SpeechRecognizer` (認識の処理) の順に作る
- 認識する言語は `speech_recognition_language` で指定する。`reason` が `RecognizedSpeech` が確かめ、認識できなかった場合と中止された場合も扱う
- 1回の発話なら `recognize_once_async()`、続けて話す音声なら `start_continuous_recognition()` とイベントの処理
- `recognizing` は認識の途中の暫定の結果、`recognized` は確定した結果

リアルタイム文字起こしとバッチ文字起こし

バッチ文字起こしでは、アプリが、文字起こしするファイルを **SAS URI** (SAS: Shared Access Signature、URI: Uniform Resource Identifier) などで指定する。SAS URI は、Azure Storage のファイルに、期限や権限を限ったアクセス権を付けたアドレスである。

観点	リアルタイム文字起こし	バッチ文字起こし
対象	マイクなどからの音声のストリーム (音声ファイルも可)	保存済みの大量の録音ファイル
動き	音声をストリーミングで送り、テキストを順に受け取る	Azure Storage などのファイルを SAS URI などで指定し、結果を 非同期 で受け取る
向いている場面	プレゼン、デモ、話している最中の字幕	過去の通話の録音をまとめて処理する
注意点	アプリが音声の入力を待ち受けている必要がある	ジョブは ベストエフォート で処理され、いつ始まるかは保証されない

音声合成 (TTS)

Azure Speech には、テキストを音声にする **テキスト音声変換 API** (Text to Speech) もある。音声はスピーカーでその場で再生することも、音声ファイルに保存することもできる。多くの言語と地域の発音に対応した定義済みの声があり、ニューラルネットワークを使った **ニューラル音声** で自然な抑揚を出せる。自社専用の **カスタム音声** を作ることもできるが、利用には申請が要り、使える対象が限られている。

Foundry ポータルの「Azure Speech - Text to Speech」のプレイグラウンドでは、声を選び、話す速さや声の高さを調整して、生成される音声を確認られる。TTS の SDK は、主に次の2つの用途で使う。

- **クライアントアプリ**: テキストを音声にして、その場で再生する (デスクトップアプリ、スマホアプリ)
- **バックエンドのサービス**: あとで再生する音声ファイルを作る

```

import os
import azure.cognitiveservices.speech as speechsdk

# ① 接続設定
speech_config = speechsdk.SpeechConfig(subscription=os.getenv("FOUNDRY_API_KEY"),
                                       endpoint=os.getenv("SPEECH_ENDPOINT"))

# ② 音声の出力先（既定のスピーカー）
audio_config = speechsdk.audio.AudioOutputConfig(use_default_speaker=True)

# ③ 使う声を指定する（日本語のニューラル音声）
speech_config.speech_synthesis_voice_name = "ja-JP-NanamiNeural"

# ④ 音声合成のオブジェクトを作り、テキストを読み上げる
synthesizer = speechsdk.SpeechSynthesizer(speech_config=speech_config, audio_config=audio_config)
result = synthesizer.speak_text_async("ご注文の商品は明日発送します。").get()

# ⑤ 結果を確認する
if result.reason == speechsdk.ResultReason.SynthesizingAudioCompleted:
    print("読み上げ完了")
elif result.reason == speechsdk.ResultReason.Canceled:
    details = result.cancellation_details
    raise RuntimeError(f"音声合成の中止: {details.reason}: {details.error_details}")

```

コードを読むときのポイントは次のとおり。

- 音声認識と同じ `azure-cognitiveservices-speech` パッケージを使う
- 出力先は `AudioOutputConfig`（認識のときの `AudioConfig` は入力元）
- 声は `speech_synthesis_voice_name` で指定する
- 合成は `SpeechSynthesizer` の `speak_text_async()`。結果の `reason` が `SynthesizingAudioCompleted` なら成功

観点	音声認識 (STT)	音声合成 (TTS)
処理のクラス	<code>SpeechRecognizer</code>	<code>SpeechSynthesizer</code>
音声の設定	<code>AudioConfig</code> （入力元：マイク・ファイル）	<code>AudioOutputConfig</code> （出力先：スピーカー・ファイル）
主なメソッド	<code>recognize_once_async()</code> 、 <code>start_continuous_recognition()</code>	<code>speak_text_async()</code> 、 <code>speak_ssml_async()</code>
共通	<code>SpeechConfig</code> で接続設定	<code>SpeechConfig</code> で接続設定 + 声の指定

読み上げ方を細かく制御したいときは、**SSML** (Speech Synthesis Markup Language: 音声合成マークアップ言語) という XML 形式で、話す速さ、声の高さ、間（ポーズ）、強調などを指定し、`speak_ssml_async()` で読み上げる。ただし、高品質の HD 音声では、使える SSML の要素が声のモデルによって違う。DragonHD の声（名前が `DragonHDLatestNeural` で終わる）では、速さ・高さの `<prosody>` と強調の `<emphasis>` が使えず、間の `<break>` は使える。Dragon HD Omni の声（`DragonHDOmniLatestNeural`）では、`<break>` も使えない。

Voice Live で音声対話エージェントを作る

音声から音声へ (Speech to Speech) は、話しかけた音声を受け取り、音声で応答する機能である。ユーザーは文字を読んだり打ったりせず、人と話すように AI とやり取りできる。テキストを介する構成では、次の3段階を組み合わせる。音声を直接扱うモデルもあるので、すべての音声対話がこの構成とは限らない。

1. **音声認識**: ユーザーの音声をテキストにする
2. **処理・推論**: テキストを分析・翻訳・要約したり、エージェントが次に何を言うかを決めたりする
3. **音声合成**: 応答のテキストを音声にする

主な用途は、音声アシスタントや AI エージェント、音声の翻訳、ハンズフリーのアプリ（カーナビ、キオスク端末）、アクセシビリティ、電話の自動応答などである。

Voice Live API (Azure Speech の Voice Live) は、リアルタイムの音声対話を **1つのサービスにまとめたもの** である。音声認識、AI の推論、音声合成を自分でつなぎ合わせる必要がなく、裏側のモデルとインフラは Azure が管理する（標準で用意されたモデルの場合。自分の Foundry リソースにデプロイしたモデルをつなぐ BYOM (Bring Your Own Model) もある）。音声を送ると音声の応答が返るほか、アバターなどの映像を返したり、必要に応じてアクションを実行したりもできる。

- 開発者は、ソリューションを作るときに、エージェントが使う **生成 AI モデルを選ぶ** 必要がある
- プレイグラウンドでは、エージェントのほうから話しかける「プロアクティブな関与」などを設定できる
- Foundry のエージェントで **Voice モード** を有効にすると、Voice Live がエージェントの定義に組み込まれる。音声の設定がエージェント側にまとまるので、クライアント側のコードが少なくて済む
- Python では `azure-ai-voicelive` パッケージを使う。ポータルにあるサンプルコードには、セッションの開始、マイクとスピーカーの接続、音声の送受信、ユーザーの割り込みへの対応などが含まれている

★ **試験ポイント**: `AudioConfig` + `SpeechRecognizer` → 音声認識、`AudioOutputConfig` + `SpeechSynthesizer` → 音声合成。保存済みの大量の録音 → バッチ文字起こし。抑揚・間・強調の細かい制御 → SSML。リアルタイムの音声対話を1つのサービスで → Voice Live。

2-4. コンピュータービジョンと画像生成の AI ソリューションを実装する

マルチモーダルモデルでプロンプトの画像を解釈する

Foundry のビジョン対応モデルでできることは次のとおり。テキストと画像を1つのモデルで扱えるので、画像専用の処理を別に組む必要が減る。

- 画像の内容を自然言語で説明する
- 画像の中の物、文字、場面について質問に答える
- グラフ、スクリーンショット、文書、写真から意味を読み取る
- 画像の理解とテキストの指示を、1つのプロンプトで組み合わせる

学習パスで紹介されている、Foundry のモデルカタログの代表的なマルチモーダルモデルは次のとおり。

モデル	特徴
GPT-4.1 / GPT-4.1-mini / GPT-4.1-nano	テキストと画像を一緒に処理できる汎用モデル。画像の説明、画像についての質問応答、文書やスクリーンショットの分析、グラフや図の解釈に使える
GPT-5 シリーズ (GPT-5.1、GPT-5.2 など)	エンタープライズやエージェント向けの高度なマルチモーダルモデル。構造化された出力、ツールの利用、長い文脈での推論に対応
パートナーのモデル	Anthropic などのモデルも、テキストと画像の理解に対応している

主な使われ方は、**AI アプリ** (画像の理解でユーザーの作業を助ける) と **AI エージェント** (画像の情報を使ってよりよい判断をする) の2つ。たとえば、工事現場の写真から安全装備の着け忘れを見つけるアプリや、ホワイトボードの写真を読み取って議事録にまとめるエージェントがこれにあたる。新しい Foundry ポータルのプレイグラウンドでは、ビジョン対応モデルを選んで画像をアップロードし、モデルが画像をどう解釈するかを確かめられる。

生成モデルで新しい画像や動画を作る

ビジョン対応モデルは画像を見てテキストを返すが、**画像生成モデル** はその逆で、テキストの説明から画像を作る。学習パスで紹介されている、Foundry の主な画像生成モデルは次のとおり。

モデル	特徴
GPT-Image-1.5	GPT-Image-1 ファミリーでいちばん新しく高性能なモデル。プロンプトへの忠実さ、繰り返し生成したときの一貫性が高い。テキストから画像、画像から画像、精密な画像の編集に対応。ブランディング、マーケティング、デザイン向け
GPT-Image-1	以前の DALL-E を発展させた汎用の画像生成モデル。テキストから画像、画像のバリエーション、画像の編集に対応。Responses API やエージェントのツールなど、幅広く対応している
GPT-Image-1-Mini	GPT-Image-1 の軽量・低コスト版。品質より速さやコストを重視する場面、試作、社内ツール、大量の生成向け
FLUX (Black Forest Labs)	Black Forest Labs の画像生成モデルのファミリー

学習パスは、多くの新しいプロジェクトでは **GPT-Image-1 ファミリー**、特に **GPT-Image-1.5** から始めることを推奨している。画像生成の活用場面として、メディア、出版、コンテンツ制作が挙げられている。

⚠ **画像・動画のモデルは入れ替わりが速い**: このノートのモデル名は学習パスに合わせている。2026年10月時点の製品ドキュメントの提供終了の予定では、Sora 2 (2025-12-08 版、プレビュー) は2026年10月15日、GPT-Image-1 は2026年10月23日、GPT-Image-1.5 は2026年12月16日に提供を終える。画像生成には GPT-Image-2 系などの新しいモデルが載っている。今も使えるかは、製品ドキュメントで確かめる。

コードで画像を生成する: アプリが Responses API に `image_generation` ツールを指定して呼び出すと、画像が生成され、base64 のデータとしてアプリに返る。製品ドキュメントでは、`model=` に画像生成ツールを呼び出せるチャットモデル (または推論モデル) のデプロイを指定し、画像生成モデルのデプロイはヘッダー `x-ms-oi-image-generation-deployment` で指定する。

```

import os
import base64
from openai import OpenAI

client = OpenAI(
    base_url=os.getenv("OPENAI_ENDPOINT"),
    api_key=os.getenv("FOUNDRY_API_KEY"),
    default_headers={
        "x-ms-oai-image-generation-deployment": os.getenv("IMAGE_DEPLOYMENT"), # 画像生成モデルのデプロイ名
        "api_version": "preview",
    },
)

response = client.responses.create(
    model=os.getenv("CHAT_DEPLOYMENT"), # 画像生成ツールを呼び出すチャットモデルのデプロイ名
    input="雨上がりの商店街を、水彩画のタッチで描いたポスター用のイラスト",
    tools=[{"type": "image_generation"}], # 画像生成ツールを使う
)

# 出力の中から、画像生成の結果 (base64) を取り出す
image_base64 = next(
    item.result for item in response.output
    if item.type == "image_generation_call"
)

# base64 を元のバイナリーに戻し、PNG ファイルとして保存する
with open("poster.png", "wb") as f:
    f.write(base64.b64decode(image_base64))

```

コードを読むときのポイントは次のとおり。

- `tools=[{"type": "image_generation"}]` で画像の生成を指示する
- 結果は `response.output` の中の、`type` が `image_generation_call` の要素の `result` に入っている
- 画像は **base64 のテキスト** で返るので、`base64.b64decode()` でバイナリーに戻してから保存する
- 学習パスのコード例は、ヘッダーを使わず、`model=` に画像生成モデルのデプロイ名を渡す形で書かれている。`tools` の指定と結果の取り出し方は同じ

OpenAI Python SDK には、画像の生成と編集のための **Images API** もあり、こちらで画像を生成することもできる。

動画生成モデル: 学習パスの時点で、Foundry が直接提供しているネイティブの動画生成モデルは **Sora** だけである。ほかの Foundry のモデルはマルチモーダル (テキスト、画像、音声) であっても、動画を出力しない。学習パスでは、次の2つを紹介している。

モデル	入力と出力	特徴
Sora 1	テキスト → 動画。画像を入力に使うこともできる	OpenAI の最初のテキストから動画への生成モデル。短い動画クリップを作る
Sora 2 (パブリックプレビュー)	テキスト → 動画、画像 → 動画、動画 → 動画 (リミックス)	音声の生成にも対応。動画全体を作り直さずに、部分的に直せる (リミックス)

どちらにも、責任ある AI の制限 (実在の人物、著作権のあるキャラクター、特定の種類のコンテンツの制限など) が組み込まれている。動画の生成は計算量が多いので、**非同期のジョブ** として実行される。REST API での流れは次のとおり。

1. **ジョブを作る**: アプリが `POST /openai/v1/videos` にプロンプト、サイズ、長さを送ると、動画の ID が返る
2. **状態を確かめ続ける (ポーリング)**: `GET /openai/v1/videos/{video_id}` で、状態が `completed` か `failed` になるまで確かめる (途中は `queued`、`in_progress`)
3. **動画をダウンロードする**: 完了したら `GET /openai/v1/videos/{video_id}/content` で MP4 のファイルを取り出す

生成には、解像度や長さによって、通常1〜5分かかる。

★ **試験ポイント**: 画像を生成する → `tools=[{"type": "image_generation"}]`、結果は base64 → `b64decode()` して保存。動画を出力できるのは Sora (学習パスの時点)。動画の生成は非同期のジョブ (作成 → ポーリング → ダウンロード)。

ビジョン機能を含む軽量アプリを作る

コードでは、**Responses API** で、テキストの指示と画像を **1つのリクエストにまとめて** 送る。画像は URL か、base64 にしたデータの URL で渡す。

```
# pip install openai
import os
from openai import OpenAI

client = OpenAI(
    base_url=os.getenv("OPENAI_ENDPOINT"), # https://<リソース名>.openai.azure.com/openai/v1/
    api_key=os.getenv("FOUNDRY_API_KEY"),
)

response = client.responses.create(
    model=os.getenv("VISION_DEPLOYMENT"), # ビジョン対応モデルのデプロイ名
    input=[
        {
            "role": "user",
            "content": [
                {"type": "input_text", "text": "この倉庫の写真から、通路をふさいでいる物を挙げてください。"},
                {"type": "input_image", "image_url": "https://example.com/warehouse.jpg"},
            ],
        },
    ],
)

print(response.output_text)
```

コードを読むときのポイントは次のとおり。

- 1つのメッセージの `content` に、`input_text` (テキストの指示) と `input_image` (画像) を並べる。これを **マルチパートメッセージ** と呼ぶ
- 画像は `image_url` に、URL か base64 のデータの URL (`data:image/jpeg;base64,...`) を渡す
- 使うのはテキストのチャットと同じ `client.responses.create()` 。モデルがビジョン対応であればよい

★ **試験ポイント**: 画像とテキストを1つのメッセージで送る → `content` に `input_text` と `input_image` を並べる (マルチパートメッセージ)。呼ぶのはチャットと同じ `responses.create()` 。

2-5. 情報抽出の AI ソリューションを実装する

Content Understanding とは

Azure Content Understanding in Foundry Tools は、文書、画像、音声、動画などの構造化されていないコンテンツから、必要な情報を **構造化データ (JSON)** として取り出すサービスである。処理は次の3段階で進む。

1. **取り込み**: アプリが、コンテンツを Content Understanding に送る
2. **AI による分析**: Content Understanding が、OCR、音声認識、自然言語の理解、マルチモーダル AI モデルを組み合わせて分析する
3. **構造化された出力**: Content Understanding が、定義した構造に沿った結果を JSON でアプリに返す。保存、検索、ほかのシステムへの連携がしやすい

JSON (JavaScript Object Notation) は、構造化データを保存・交換するためのテキストの形式で、人にも読みやすく、プログラムでも扱いやすい。

文書とフォームから情報を抽出する

Content Understanding が、基本的な OCR や文字起こしのサービスと違うのは、**スキーマに基づいて抽出する** 点である。たとえば納品書のスキーマは、次のように定義できる。

- 発行元の会社名
- 納品書番号
- 納品日
- 納品先
- 明細 (コレクション: 複数の行)
 - 品名
 - 数量
 - 単価
 - 金額
- 合計金額

スキーマのポイントは2つある。

- **入れ子の構造を持てる**: 「明細」のように複数の行 (コレクション) があり、それぞれの行が品名・数量・単価・金額を持つ、という関係まで表せる。値どうしの関係を扱えるのは、基本的な OCR には無い点である
- **意味で当てはめる (セマンティック)**: 「伝票番号」「No.」と書かれていても、ラベルの無い番号でも、同じ意味だと判断できれば「納品書番号」として取り出せる

アナライザー は、入力を受け取り、AI で分析し、構造化された結果を出す Content Understanding の処理の単位である。アナライザーにはスキーマが含まれていて、一度設定すれば、送られてきたコンテンツに同じ抽出のルールを一貫して当てはめる。結果も毎回予測できる JSON の形で返るので、その後の保存や自動処理がしやすい。

種類	内容
構築済みアナライザー (prebuilt)	よくある用途向けに、あらかじめ用意されたもの
カスタムアナライザー	自分でスキーマを定義して作るもの

学習パスで紹介されている構築済みアナライザーの ID は次のとおり。ほかにも、`prebuilt-documentSearch`、`prebuilt-receipt`、`prebuilt-callCenter` など多数ある。

アナライザー ID	対象
<code>prebuilt-invoice</code>	請求書
<code>prebuilt-imageSearch</code>	画像
<code>prebuilt-audioSearch</code>	音声
<code>prebuilt-videoSearch</code>	動画

使い方の流れは、アナライザーを選ぶか作る → コンテンツを送って分析を依頼する → サービスがスキーマを当てはめる → スキーマに沿った構造化 JSON が返る、である。

Foundry リソースを作れば、新しい Foundry ポータルで Content Understanding を試せる。サンプルのコンテンツも用意されていて、自分のファイルをアップロードして分析することもできる。文書の画像を分析すると、テキストとレイアウトの情報が返り、請求書なら「販売元の住所」などの項目と、その値が対応付けられて返る。結果は JSON でも確かめられる。ポータルは、コードで自動化する前に、アナライザーが期待どおりの項目を返すかを手早く確かめる方法になる。

★ **試験ポイント**: 文字を読むだけ → 基本的な OCR。決まった項目を意味で取り出し、入れ子の構造で返す → スキーマに基づく抽出 (Content Understanding)。スキーマを含み、同じルールを繰り返し当てはめる単位 → アナライザー。自分でスキーマを決める → カスタムアナライザー。

画像から情報を抽出する

画像にも、同じようにスキーマを当てはめられる。たとえば店の棚の写真に「棚の段数」「空いている段の数」「値札が見えない商品の数」というスキーマを当てはめると、写真ごとに各項目の値が入った結果が返る。構築済みの `prebuilt-imageSearch` は、画像の内容を説明する文を返す。Content Understanding の SDK の説明では、文字 (手書きを含む) が写った画像には、`prebuilt-documentSearch` を使うよう案内されている。

音声と動画から情報を抽出する

構築済みの音声アナライザー (`prebuilt-audioSearch`) は、話者を分けた **文字起こし** と、会話の要約を返す。自分で決めた項目を取り出すには、スキーマを定義したカスタムアナライザーを使う。たとえば社内研修の録音に「講師名」「取り上げたテーマ」「受講者からの質問」というスキーマを当てはめると、各項目が埋まった結果が返る。動画では映像と音声の両方を分析するので、場面ごとの内容 (例: 紹介された製品の名前と価格) も項目にできる。

Content Understanding で軽量アプリを作る

Content Understanding API を使うと、情報抽出をコードから自動化できる。分析は **非同期** で行われる。アプリが分析を依頼した時点では結果は返らず、サービス側の処理が終わってから、アプリが結果を受け取る。REST で直接扱う場合は、アプリが、返ってきた **Operation-Location** の URL を、処理が終わるまでポーリングする。Python SDK を使えば、このポーリングを SDK が代わりに行う。

下のコードは、Content Understanding が使うモデル (構築済みアナライザーに要る gpt-4.1、gpt-4.1-mini、text-embedding-3-large など) のデプロイと、Foundry リソースごとに1回の既定のモデルの割り当てが済んでいる前提である。

```
# pip install azure-ai-contentunderstanding
import os
from azure.ai.contentunderstanding import ContentUnderstandingClient
from azure.ai.contentunderstanding.models import AnalysisInput
from azure.core.credentials import AzureKeyCredential

# ① クライアントを作る（エンドポイントは https://<リソース名>.services.ai.azure.com/）
client = ContentUnderstandingClient(
    endpoint=os.getenv("CU_ENDPOINT"),
    credential=AzureKeyCredential(os.getenv("FOUNDRY_API_KEY")),
)

# ② アナライザーの ID と入力指定して、分析を始める（時間のかかる操作）
poller = client.begin_analyze(
    analyzer_id="prebuilt-invoice",
    inputs=[AnalysisInput(url="https://example.com/sample-invoice.pdf")],
)

# ③ 完了を待って結果を受け取る（ポーリングは SDK が行う）
result = poller.result()

# ④ 抽出した結果を読む
for content in result.contents:
    print(content.markdown)    # 内容を Markdown にしたもの（音声なら文字起こし）
    print(content.fields)      # スキーマに沿って抽出した項目と値
```

コードを読むときのポイントは次のとおり。

- パッケージは `azure-ai-contentunderstanding`、クラスは `ContentUnderstandingClient`
- 分析を始めるのは `begin_analyze()`。時間のかかる非同期の処理 (**LRO**: Long Running Operation) を始めるメソッドで、Azure の SDK では名前が `begin_` で始まる
- `begin_analyze()` が返すのは結果そのものではなく **ポラー**（処理の状態を追いかけるオブジェクト）。サービス側の処理は非同期だが、この同期クライアントの `poller.result()` は完了まで待ってから結果を返す。メソッド名が `begin_` だから Python の `await` が要る、とは限らない
- `analyzer_id` を `prebuilt-audioSearch` に変えれば音声、`prebuilt-videoSearch` に変えれば動画の分析になる。コードの形は同じ
- 結果の `contents` の各要素に、`markdown`（内容や文字起こし）と `fields`（抽出した項目）が入っている
- 学習バスのコード例は、入力を `inputs=[{"url": ...}]` の辞書で渡している。SDK のリファレンスの型は `AnalysisInput` のリストである

返ってくる JSON は次のような形になる。各項目には型と値が入り、信頼度 (`confidence`) が付くこともある。

```
{
  "status": "Succeeded",
  "result": {
    "analyzerId": "prebuilt-invoice",
    "contents": [
      {
        "markdown": "# 請求書 ... サンプル商事株式会社 ...",
        "fields": {
          "CustomerName": { "type": "string", "valueString": "見本工業株式会社", "confidence": 0.93 },
          "InvoiceDate": { "type": "date", "valueDate": "2026-09-30", "confidence": 0.98 }
        }
      }
    ]
  }
}
```

★ **試験ポイント**: `begin_` で始まるメソッド → 非同期 (LRO)。結果は `poller.result()` で完了を待って受け取る。REST なら Operation-Location をポーリングする。

✓ 第2章 チェック問題

- 1つの部門で、問い合わせ対応と社内文書の検索という2つの AI を並行して作る。Foundry リソースとプロジェクトはどう分けるのが典型的か。
- Foundry の REST のリクエストで、キーの代わりに **Authorization: Bearer <トークン>** のヘッダーを送っている。このトークンを発行するのは何か。
- `.env` に `FOUNDRY_API_KEY=...` と書いたが、コードでは `os.getenv("API_KEY")` を読んでいる。どうなるか。
- 「規程に無いことは推測で答えず、担当部署を案内する」という指示は、システムプロンプトとユーザープロンプトのどちらに書くか。
- プロンプトに入力と出力の例を3組入れて、期待する形を示した。この書き方を何と呼ぶか。

6. `client.responses.create(model=..., input=...)` の `model=` に渡すのは、モデル名とデプロイ名のどちらか。
7. エージェントに社内マニュアルを参照させ、出典付きで答えさせたい。ポータルでエージェントに追加するのは、ツールとナレッジのどちらか。
8. MCP で、ツールやデータを公開する側の役割を何と呼ぶか。
9. ポータルで作ったエージェントをアプリから呼ぶ。最初に作るクライアントのクラスと、OpenAI 互換のクライアントを取り出すメソッドは何か。
10. アンケートの自由記述から氏名や電話番号を伏せ、同じ入力なら同じ結果になるようにしてデータベースに保存したい。汎用モデルと Azure Language のどちらが向くか。
11. PII 検出の結果から、個人情報を伏せた文章を取り出すプロパティは何か。
12. `AudioOutputConfig` と `SpeechSynthesizer` を使っているコードは、音声認識と音声合成のどちらか。
13. Azure Storage に保存した1年分の通話の録音を、まとめて文字起こししたい。どの方式が向くか。
14. 音声認識・推論・音声合成を自分でつなぐに、リアルタイムの音声対話エージェントを作りたい。何を使うか。
15. 画像とテキストの指示を1つのメッセージで送るとき、それぞれの `type` は何か。
16. Responses API で画像を生成するとき、`tools` に指定する値と、返ってくる画像のデータの形式は何か。
17. Sora で動画を作るとき、REST で呼ぶ3つの操作を順に挙げよ。
18. Content Understanding で、自社で決めた点検報告書の項目(点検者、異常の有無など)を取り出したい。構築済みとカスタムのどちらのアナライザーを使うか。
19. `poller = client.begin_analyze(...)` の直後に `poller.fields` を読もうとして失敗した。何が足りないか。
20. 録音した音声の質問に、音声認識の処理を別に組まずに答えさせたい。どんなモデルをデプロイし、音声をどう渡すか。
21. Content Understanding で、手書きの文字が写った画像から内容を取り出したい。案内されている構築済みアナライザーはどれか。
22. アプリのコードと設定から Foundry の API キーをなくしたい。推奨される認証の方法は何か。キーを使い続ける場合は、どこに保管するか。

▼ 解答

1. Foundry リソースを1つ作り、ユースケースごとに Foundry プロジェクトを作る
2. Microsoft Entra ID (Entra ID のトークンによる認証。Python では `DefaultAzureCredential` など取得する)
3. `os.getenv()` が `None` を返し、クライアントの作成や呼び出しでエラーになる。名前を一致させる
4. システムプロンプト(振る舞いと制約を決める内容)
5. フューショット(例が1つならワンショット)
6. デプロイ名
7. ナレッジ(RAG で文脈を与える。ツールは行動。概念のうえでは、情報を取りに行くナレッジツールと呼ぶこともある)
8. MCP サーバー(エージェントは MCP クライアント)
9. `AIProjectClient` を作り、`get_openai_client()` で取り出す
10. Azure Language (PII 検出。決定的で、同じ入力なら同じ結果が返る)
11. `redacted_text`
12. 音声合成(TTS)
13. バッチ文字起こし(SAS URIなどでファイルを指定し、非同期で結果を受け取る)
14. Voice Live (Voice Live API)
15. テキストは `input_text`、画像は `input_image`
16. `{"type": "image_generation"}`。画像は base64 のテキストで返るので、`b64decode()` してから保存する
17. ジョブの作成 → 状態のポーリング → 完了後のダウンロード。順に `POST /openai/v1/videos`、`GET .../videos/{video_id}`、`GET .../videos/{video_id}/content`
18. カスタムアナライザー(取り出す項目をスキーマで定義する)
19. `poller.result()` で完了を待って結果を受け取っていない。`begin_analyze()` が返すのは結果ではなくポーラー。抽出した項目は、結果の `contents[i].fields` にある
20. 音声を入力できるマルチモーダルモデル。音声のデータ(base64)を、テキストの指示と一緒に1つのリクエストで送る
21. `prebuilt-documentSearch` (`prebuilt-imageSearch` は画像の内容を説明する文を返す)
22. Microsoft Entra ID によるキーレスの認証(マネージド ID など)。キーを使うなら Azure Key Vault に保存し、マネージド ID で取り出す

3-1. 早見表

サービス／用語	一言でいうと	カテゴリ
Microsoft Foundry	AI アプリとエージェントを作る、エンタープライズ向けの統合 PaaS	プラットフォーム
Foundry リソース	Azure のリソース。モデルのホスティング、Foundry Tools へのアクセス、セキュリティ境界、クォータ	構成
Foundry プロジェクト	リソースの中の作業スペース。ユースケースごとに作る	構成
Foundry Agent Service	エージェントの構築・実行の基盤	エージェント
Foundry IQ	エージェント向けの知識ベース (一部の機能はプレビュー)	エージェント
Foundry Tools	用途に特化した構築済みの AI サービスの総称	ツール
Azure Language	テキスト分析 (言語検出、PII 検出など)。決定的な結果を返す	ツール
Azure Speech	音声認識と音声合成	ツール
Voice Live	音声対話 (音声認識・推論・音声合成) を1つのサービスで	ツール
Azure Content Understanding	文書・画像・音声・動画から、スキーマに沿って項目を抽出	ツール
Responses API	モデルとエージェントを呼び出す、統一された API	API
MCP	エージェントが外部のツールにつながるオープンな標準	規格
埋め込みモデル	文章をベクトルに変える。検索や類似度の計算に使う	モデル
GPT-Image-1.5	画像生成 (学習パスが新しいプロジェクトに推奨)	モデル
Sora	動画生成 (学習パスの時点で、Foundry のネイティブの動画生成モデル)	モデル

3-2. 逆引き「～したいときは何を使う？」

やりたいこと	答え
柔軟に分析したい、会話しながら深掘りしたい	汎用 AI モデル (Responses API)
同じ入力なら同じ値 (言語コード、PII の伏せ字など) が要る、自動化パイプラインに組み込む	Azure Language
個人情報を伏せ字にしたい	Azure Language の PII 検出 (<code>redacted_text</code>)
エージェントに決定的なテキスト分析を持たせたい	Azure Language MCP サーバー
音声を文字にしたい、字幕を付けたい	Azure Speech の音声認識 (STT)
保存済みの大量の録音をまとめて文字にしたい	バッチ文字起こし
テキストを読み上げたい	Azure Speech の音声合成 (TTS)
抑揚、間、強調を細かく制御したい	SSML
リアルタイムの音声対話を1つのサービスで作りたい	Voice Live
音声のプロンプトにそのまま応答させたい	音声を入力できるマルチモーダルモデル
画像を説明させたい、写真について質問したい	ビジョン対応のマルチモーダルモデル
テキストから画像を作りたい	画像生成モデル (GPT-Image-1 ファミリー)
テキストから動画を作りたい	Sora
様式がばらばらな文書から、決まった項目を JSON で取り出したい	Azure Content Understanding
意味の近い文書を探したい	埋め込みモデル
社内文書に基づいて、出典付きで答えさせたい	RAG (エージェントのナレッジ、Foundry IQ)
急がない大量の処理を安くしたい	Batch (Global Batch / Data Zone Batch)
EU データゾーンの中でデータを処理したい	Data Zone (Data Zone Standard など)
回答のばらつきを減らしたい	temperature を下げる
レート制限のエラーを減らしたい	最大出力トークン数を下げる、同時のリクエストを減らす
キーをコードに書かずに管理したい	Azure Key Vault + マネージド ID (キーを使わずに済むなら Entra ID のキーレスの認証)

3-3. 紛らわしいペア比較

比較	違いの核心
公平性 vs 包括性	結果の偏り vs 使える人の範囲
透明性 vs 説明責任	利用者が理解できる vs 人と組織が責任を負う
信頼性と安全性 vs プライバシーとセキュリティ	AI の動作が危険か vs データの保護
LLM vs SLM	強力で汎化しやすいがコストが高い vs 特定の分野やデバイス上のローカルのアプリ向け
ナレッジツール vs アクションツール	情報を取りに行く(検索など) vs 作業を実行する(システムの更新など)
キーワード検索 vs ベクトル検索	正確な語やコードの一致 vs 意味の近さ(ハイブリッド検索は両方)
キーフレーズ vs エンティティ	主な論点(種類なし) vs 決まった種類に分類(カテゴリあり)
ステミング vs レンマ化	語尾を機械的に削る vs 規則と語彙で辞書の形に戻す
抽出型 vs 抽象型の要約	元の文を選ぶ vs 新しい文章で書き起こす
音響モデル vs 言語モデル	音声の特徴から音素の確率 vs 語の知識であいまいさを解消
CNN vs ViT	フィルターで特徴マップ vs パッチに分けてアテンション
汎用モデル vs Azure Language	柔軟で、結果の表現が変わりうる(構造化出力で形は指定できる) vs 決定的で、タスク別に構造化された結果
ツール vs ナレッジ	行動(アクション) vs 文脈(RAG)
MCP クライアント vs MCP サーバー	エージェント側 vs ツールを公開する側
モデルを直接呼ぶ vs エージェントを呼ぶ	<code>OpenAI(...)</code> で <code>model=</code> にデプロイ名 vs <code>AIProjectClient</code> → <code>get_openai_client()</code> で <code>agent_reference</code>
<code>AudioConfig</code> vs <code>AudioOutputConfig</code>	音声の入力元(認識) vs 音声の出力先(合成)
リアルタイム vs バッチ文字起こし	ストリームを順に受け取る vs 保存済みのファイルを非同期で
基本的な OCR vs スキーマに基づく抽出	文字を読むだけ vs 意味で項目に当てはめ、入れ子の構造で返す
構築済み vs カスタムアナライザー	用意されたもの(<code>prebuilt-...</code>) vs 自分でスキーマを定義
Standard vs Provisioned vs Batch	トークンの従量課金 vs PTU で確保した処理能力 vs 50%安い非同期
API キー vs Entra ID	キーを渡す <code>AzureKeyCredential</code> ・ <code>api_key=</code> ・ <code>subscription=</code> vs Entra ID のトークン。 <code>api_key=</code> にはトークンプロバイダーも渡せる

3-4. 数字・固有名詞の暗記リスト

- **合格ライン**: 700点(スケールドスコア)
- **出題比率**: 第1領域 40〜45%、第2領域 55〜60%
- **出題範囲の版**: 2026年4月15日(ローカライズ版の試験は、英語版の約8週間後に更新)
- **Batch**: Global Standard より50%安い。24時間以内の完了が目標
- **データゾーン**: US、EU (EU Data Boundary)、アジア太平洋
- **Provisioned の単位**: PTU。PTU の数と時間で支払う
- **TPM/RPM**: 1分あたりのトークン数/リクエスト数の上限
- **動画生成の時間**: 通常1〜5分(解像度や長さによる)
- **動画のジョブの状態**: `queued`、`in_progress`、`completed`、`failed`
- **構築済みアナライザーの ID**: `prebuilt-invoice`、`prebuilt-imageSearch`、`prebuilt-audioSearch`、`prebuilt-videoSearch`
- **ピクセルの値**: グレースケールで 0 (黒)〜255 (白)。カラーは RGB の3チャンネル
- **n-gram**: 1語=ユニグラム、2語=バイグラム、3語=トライグラム

3-5. コードの早見表

パッケージ・クラス・主なメソッド

用途	pip のパッケージ	クライアント/主なクラス	主なメソッド・書き方	結果の取り出し
モデルの呼び出し (チャット、画像分析、 画像生成)	<code>openai</code>	<code>OpenAI(base_url=..., api_key=...)</code>	<code>client.responses.create(model=</code> デプロイ名, <code>input=...)</code>	チャット・画像分析は <code>response.output_text</code> 。画像生成は <code>response.output</code> 内の <code>image_generation_call</code> の <code>result</code>

用途	pip のパッケージ	クライアント／主なクラス	主なメソッド・書き方	結果の取り出し
エージェントの呼び出し	azure-ai-projects 、 azure-identity	AIProjectClient(endpoint=..., credential=DefaultAzureCredential())	agents.get(agent_name=...) 、 get_openai_client() → responses.create(..., extra_body={"agent_reference": {...}})	response.output_text
テキスト分析	azure-ai- textanalytics	TextAnalyticsClient(endpoint=..., credential=AzureKeyCredential(key))	detect_language(...)、 recognize_pii_entities(...) など	result.primary_language 、 result.redacted_text 、result.entities
音声認識	azure- cognitiveservices- speech	SpeechConfig → AudioConfig → SpeechRecognizer	recognize_once_async().get() 、 start_continuous_recognition()	result.text 、イベントの evt.result.text
音声合成	azure- cognitiveservices- speech	SpeechConfig → AudioOutputConfig → SpeechSynthesizer	speak_text_async(text).get() 、 speak_ssmml_async(ssml)	result.reason == ResultReason.SynthesizingAudioCompleted
音声対話エージェント	azure-ai-voicelive	(ポータルのサンプルコードを使う)	—	—
情報抽出	azure-ai- contentunderstanding	ContentUnderstandingClient(endpoint=..., credential=AzureKeyCredential(key))	begin_analyze(analyzer_id=..., inputs=[...]) → poller.result()	result.contents[i].markdown 、.fields
動画生成	(REST を直接呼ぶ)	—	POST /openai/v1/videos → GET .../videos/{id} → GET .../videos/{id}/content	MP4 のファイル

コードの見分け方

コードに出てくるもの	判断
client.responses.create()	Responses API でモデル(またはエージェント)を呼んでいる
"type": "input_image"	マルチモーダルモデルに画像を渡している(画像分析)
tools=[{"type": "image_generation"}]	画像生成
"type": "agent_reference"	エージェントの呼び出し
AIProjectClient	Foundry プロジェクトへの接続(エージェント関連)
TextAnalyticsClient	Azure Language (テキスト分析)
SpeechRecognizer 、AudioConfig	音声認識 (STT)
SpeechSynthesizer 、AudioOutputConfig	音声合成 (TTS)
ContentUnderstandingClient 、begin_analyze	Content Understanding (情報抽出)
prebuilt-invoice 、prebuilt-audioSearch など	Content Understanding の構築済みアナライザー
AzureKeyCredential(key)	API キーによる認証
DefaultAzureCredential()	Microsoft Entra ID による認証
load_dotenv() 、os.getenv(...)	load_dotenv() は .env から環境変数に読み込む。os.getenv() は設定済みの環境変数の値を取り出す

同期と非同期

処理	方式	結果の受け取り方
チャット、画像分析、画像生成	その場で結果が返る	レスポンスをそのまま読む
音声認識 (リアルタイム)	ストリーミング	イベントで順に受け取る
音声のバッチ文字起こし	非同期 (ベストエフォート)	あとで結果を受け取る
Content Understanding の分析	非同期 (LRO)	poller.result() で完了を待つ (REST なら Operation-Location をポーリング)
Sora の動画生成	非同期のジョブ	状態をポーリングし、完了後にダウンロード

第4章 | 学習チェックリストと試験当日

4-1. 理解度チェックリスト

第1章: AI の概念と機能を特定する

- ☐ 責任ある AI の6原則を、英語名とあわせて言え、場面からどれかを選べる
- ☐ 公平性と包括性、透明性と説明責任の違いを説明できる
- ☐ 言語モデルの学習と生成(推論)の違いと、トークン化 → 埋め込み → 次のトークンの予測の流れを説明できる
- ☐ LLM と SLM、基盤モデル、ファインチューニングの違いを説明できる
- ☐ グラウンディングと RAG の関係を説明できる
- ☐ タスク(チャット、埋め込み、マルチモーダル、画像生成、動画生成)から、モデルのタイプを選べる
- ☐ Azure が直接販売するモデルと、パートナーとコミュニティのモデルの違いを説明できる
- ☐ ベンチマーク、リーダーボード、評価器の役割と、評価器が直さないことを説明できる
- ☐ デプロイの種類(方式と処理する場所)を、要件から選べる
- ☐ TPM、RPM、スロットリングと、その対処を説明できる
- ☐ temperature、最大出力トークン数、システム指示、top_p の役割を説明できる
- ☐ 6つのワークロードを、シナリオから見分けられる
- ☐ 生成 AI とエージェントの違い(モデル・指示・ツール)を説明できる
- ☐ テキスト分析の手法(キーフレーズ、エンティティ、感情、要約、言語検出、PII)を区別できる
- ☐ 前処理(正規化、ストップワード、n-gram、ステミング、レンマ化、品詞タグ付け)と TF-IDF、TextRank を説明できる
- ☐ 音声認識の6段階と、音声合成の4段階(代表的な処理の流れ)を説明できる
- ☐ 画像分類、物体検出、セマンティックセグメンテーションを区別できる
- ☐ フィルター、CNN、ViT、拡散モデルの要点を説明できる
- ☐ OCR とフィールド抽出の違い、テンプレート型・機械学習型・生成 AI 型の違いを説明できる

第2章: Microsoft Foundry を使用して AI ソリューションを実装する

- ☐ Foundry リソースと Foundry プロジェクトの関係を説明できる
- ☐ エンドポイントの形から、接続先を読み取れる
- ☐ API キーと Entra ID の認証を、コードの書き方から見分けられる(キーレスの推奨、Key Vault)
- ☐ `.env`、`load_dotenv()`、`os.getenv()` の使い方と、名前が合わないときに起きることを説明できる
- ☐ システムプロンプトとユーザープロンプトを書き分けられる
- ☐ プレイグラウンドでできることと、コード表示の使い道を説明できる
- ☐ Responses API でチャットクライアントを書ける(`model=` にデプロイ名、`output_text`)
- ☐ エージェントのツールとナレッジ、MCP のクライアントとサーバーを区別できる
- ☐ `AIProjectClient` からエージェントを呼ぶコードを読める
- ☐ 汎用モデルと Azure Language の使い分け(決定的な結果が要るか)と、構造化出力との違いを説明できる
- ☐ `TextAnalyticsClient` で言語検出と PII 検出を行うコードを読める
- ☐ 音声のプロンプトをマルチモーダルモデルに渡す方法を説明できる
- ☐ 音声認識と音声合成のコード(`AudioConfig` / `AudioOutputConfig`)を見分けられる
- ☐ リアルタイムとバッチの文字起こし、SSML、Voice Live を使い分けられる
- ☐ 画像分析のマルチパートメッセージ(`input_text` + `input_image`)を書ける
- ☐ 画像生成のコード(`image_generation`、base64)を読める
- ☐ 動画生成の非同期のジョブの流れを説明できる
- ☐ Content Understanding のスキーマ、アナライザー(構築済み/カスタム)と、画像に使う構築済みアナライザーを説明できる
- ☐ `begin_analyze()` と `poller.result()` の関係を説明できる

4-2. 試験当日のポイント

- 合格は700点。スケールドスコアなので、正答率の70%と同じとは限らない
- Fundamentals の試験では、試験中に Microsoft Learn を参照できない
- 事前に試験サンドボックス(<https://aka.ms/examdemo>)で画面の操作に慣れておく
- コード片が出てきたら、クライアントのクラス、呼ぶメソッド、結果の取り出し方の3点で読む
- 迷った問題は後で見直せるように印を付け、先に進む。休憩を取ると、それより前の問題には戻れない

4-3. 追加で活用したい公式リソース

リソース	URL
学習ガイド (英語・こちらが最新)	https://learn.microsoft.com/en-us/credentials/certifications/resources/study-guides/ai-901
学習ガイド (日本語・機械翻訳)	https://learn.microsoft.com/ja-jp/credentials/certifications/resources/study-guides/ai-901
試験ページ	https://learn.microsoft.com/ja-jp/credentials/certifications/exams/ai-901/
認定資格ページ	https://learn.microsoft.com/ja-jp/credentials/certifications/azure-ai-fundamentals/
公式の練習用評価 (AI Skills Navigator。 サインインが必要)	https://aiskillsnavigator.microsoft.com/credentials/cert-83587e0a0754cfee561ade3e27d9fa1cdaf15ae03be52d2413b2b858d1b4eda4
学習パス: AI concepts for developers and technology professionals	https://learn.microsoft.com/training/paths/ai-concepts/
学習パス: Get started with AI applications and agents on Azure	https://learn.microsoft.com/training/paths/get-started-ai-apps-agents/
Foundry のデプロイの種類	https://learn.microsoft.com/azure/foundry/foundry-models/concepts/deployment-types
Foundry のモデルの提供終了の予定	https://learn.microsoft.com/azure/foundry/openai/concepts/model-retirement-schedule
汎用モデルの構造化出力 (JSON Schema)	https://learn.microsoft.com/azure/foundry/openai/how-to/structured-outputs
OpenAI と Azure OpenAI のエンドポイントの切り替え (認証の節)	https://learn.microsoft.com/azure/developer/ai/how-to/switching-endpoints
Azure OpenAI のクォータとレート制限 (429 と再試行)	https://learn.microsoft.com/azure/foundry/openai/how-to/quota
Content Understanding の構築済みアナライザーの一覧	https://learn.microsoft.com/azure/ai-services/content-understanding/concepts/prebuilt-analyzers
試験サンドボックス	https://aka.ms/examdemo