IT資格道場

AWS AIP-C01 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

無料サンプル(第1章まで)

AWS 公式 Exam Guide(AIP-C01・2026年9月7日取得)準拠
2026年7月版

本書のご利用にあたって

- 本書は「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。
- 無料サンプルであっても著作物です。本書の転載・改変しての再配布・二次販売を禁じます。
- 本書を AI モデルの学習データとして利用すること(学習目的の入力・データセット化を含む)を禁じます。
- 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。

不正な配布を見つけた場合は info@shikaku-doujou.com までご連絡ください。

目次

3点セットの使い方

種類	役割
① 学習用テキスト(本書)	これだけで問題が解けるようになる本編
② 頻出度別ノート	テキストを読んだら次はこれ。頻出順に絞った問題と解説で「解ける」に橋渡し。試験直前の最終チェックにも
③ 暗記用ノート	覚えるべき要点だけを一問一答に凝縮。空き時間の反復と用語の再確認用

使い方: ① 学習用を読む → ② 頻出度別で解き方を掴む → ③ 問題演習で腕試し → ④ 頻出度別に戻って再確認(試験直前もここ)。暗記用は空き時間や用語の再確認に。

このテキストの位置づけ: 本書は①の学習用テキストです。まずはここを通読し、これだけで問題が解けるようになることを目標にしてください。

はじめに ── この教材の使い方

このテキストは、Amazon Web Services（AWS）が実施する **AWS Certified Generative AI Developer – Professional（試験コード AIP-C01）** の合格を目的に作られています。

この試験が問うのは、生成 AI の用語や個々のサービスの暗記ではありません。問われるのは、**業務要件から基盤モデル（foundation model、FM）と検索の設計を決め、エージェントやアプリケーションとして実装し、安全性・セキュリティ・ガバナンスで守り、コストと性能と監視で運用に載せ、評価とトラブルシューティングで良し悪しを判定できること**です。Professional の粒度なの

で、「知っている」ではなく「設計し、実装し、運用する」ところまで踏み込みます。

本書は AWS の公式教材ではなく、IT資格道場が独自に書き下ろしたオリジナルの教材です。実際に出題された問題を再現したものではありません。

試験の基本情報

項目	内容
試験名	AWS Certified Generative AI Developer – Professional（AIP-C01）
実施	Amazon Web Services
問題数	**75問**（うち採点対象 65問・非採点 10問。どれが非採点かは受験者には分かりません）
試験時間	**180分**
出題形式	**Multiple choice（4肢のうち正解1・誤答3）／Multiple response（5肢以上から正解2つ以上）** の2形式
合格ライン	**スケールスコア 100〜1,000 のうち 750**
採点方式	全体で合否を決める補償型。分野ごとの足切りはありません

受験料・試験時間・出題数・試験ガイドは改定されることがあります。 お申し込みの前に、AWS 公式サイトで最新の情報をご確認ください。本書は上に挙げた数字以外の制度情報（受験料・有効期限・再受験規定など）は扱いません。

出題範囲の全体像 —— 5つのドメインと本書の章

ドメイン	分野	採点対象に占める割合（公式）	本書
1	基盤モデルの統合、データ管理、コンプライアンス	**31%**	第1章
2	実装と統合	**26%**	第2章
3	AI の安全性、セキュリティ、ガバナンス	**20%**	第3章
4	GenAI アプリケーションの運用効率と最適化	**12%**	第4章
5	テスト、検証、トラブルシューティング	**11%**	第5章

本書の章の並びは、試験ガイドの並びとまったく同じです。節見出しの末尾にある `[1.1]` のような番号は、試験ガイドの Task statement の番号です。公式ガイドを手元に置いて対照しながら読めます。AWS はドメインごとの割合だけを公表しており、Task 単位の出題数は公表していません。本書もそれ以上の数字は書きません。

想定される受験者と、本書の読み進め方

公式ガイドは、この試験の対象を「生成 AI アプリケーションを AWS 上で設計・実装・運用する開発者」としています。一方で、**モデルの開発と学習そのもの・高度な機械学習の技法・データエンジニアリングと特徴量エンジニアリング**は対象外の職務として明示されています。つまり本試験は「モデルを作る人」ではなく「既存の基盤モデルを使ってプロダクトを作る人」の試験です。学習の重心も、モデル内部の理論ではなく、**Amazon Bedrock を中心とした AWS サービスの組み合わせ方**に置いてください。

とはいえ本書は、生成 AI の用語と AWS の基礎から順に積み上げる構成にしてあります。Bedrock を触ったことがなくても、章の順に読めば必要な知識がそろいます。

サービス名・機能名は英語のまま覚える

本文では、**サービス名・機能名・設定項目名を英語表記のまま**書いています（Amazon Bedrock、Knowledge Bases、Agents、AgentCore、Guardrails、Prompt Management、Flows、Model Evaluation、Converse API、Provisioned Throughput、cross-Region inference、OpenSearch Serverless、Strands Agents、MCP …）。本番の設問文と選択肢に出るのはこの表記であり、日本語訳だけを覚えても画面では出会いません。英語の名前を見た瞬間に「どのドメインの、どの場面の機能か」が反射で出る状態を目指してください。

全設問に効く判断軸 ——「4つの問い」

Professional の設問は、長い場面設定のうしろに必ず「何を最優先するか」が埋めてあります。選択肢が4つとも技術的に動く構成であることも珍しくありません。迷ったら、次の順で切ってください。

問い	何を切り分けるか	分かれ方
問い1: どの工程の話か	設計・データ／実装・統合／安全性・ガバナンス／運用・最適化／評価・切り分け	同じ Bedrock でも工程が違えば答えが変わります（例: Guardrails は「安全」の顔と「コンプライアンスの証跡」の顔で出ます）

問い	何を切り分けるか	分かれ方
問い2: 要件は何を最優先しているか	精度・遅延・コスト・データ所在・運用負荷	RAG かファインチューニングか、オンデマンドか Provisioned Throughput か、リアルタイムかバッチか、はこの軸で一意に決まります
問い3: 最小権限・最小露出で満たせているか	IAM のスコープ・ネットワーク経路（PrivateLink）・暗号化（KMS）・ログの残し方	要件を満たす案が複数あるとき、正解はより狭い権限・より閉じた経路のものです
問い4: マネージド機能で済むか	Bedrock の組み込み機能 対 自前実装	AWS の正解は「既にある機能を使う」側に寄ります。自前でベクトル検索やフィルタを書く選択肢は、要件がそれを強制していない限り誤答です

4ステップ学習法

購入者限定テキストは3冊で1セットです。

- **STEP 1（理解する）**: 本書「学習用テキスト」を通読し、5つのドメインの地図と4つの問いを頭に入れる
- **STEP 2（解き方を掴む）**: 「頻出度別ノート」で頻出度の高い論点を解いて、解き方を掴む
- **STEP 3（腕試し）**: 本サイトの問題演習で、本番と同じ2形式に慣れる
- **STEP 4（再確認）**: 「頻出度別ノート」に戻って総ざらい。試験直前もここへ戻る

「暗記用テキスト」は本線には入れません。通勤時間や休憩時間など、**空き時間の反復と用語の再確認**に並走させてください。

それでは、第1章から始めます。

第1章 基盤モデルの統合、データ管理、コンプライアンス（ドメイン1・採点対象の31%）

このドメインは採点対象の31%を占め、五つのドメインの中で最大です。問われるのは「Amazon Bedrock のボタンの位置」ではなく、業務要件（精度・遅延・コスト・データ所在・監査）から基盤モデル（foundation model、FM）と検索の設計を決め、それを実装し、運用に載せられるかどうかです。Professional の粒度なので、「知っている」ではなく「設計し実装し運用する」ところまで踏み込みます。

本章は Task statement 1.1〜1.6 の順に進みます。1.1 で要件分析と設計、1.2 で FM の選定と設定、1.3 でデータ検証と処理パイプライン、1.4 でベクトルストア、1.5 で検索メカニズム、1.6 でプロンプトエンジニアリングとガバナンスです。1.4 と 1.5 は合わせて RAG（Retrieval Augmented Generation、検索拡張生成）の設計そのものであり、この試験でもっとも配点の厚い部分です。

1-1 要件を分析し、GenAI ソリューションを設計する [1.1]

この Task statement は、白紙の状態から「何を作るか」を決める工程です。判断の材料は業務ニーズ（business needs）と技術的制約（technical constraints）の二つで、これを architectural design（アーキテクチャ設計）に落とし、PoC（proof-of-concept、技術実証）で検証し、standardized technical components（標準化された技術コンポーネント）として横展開できる形に固めます。

1-1-1 業務ニーズと技術的制約に沿ったアーキテクチャ設計 [1.1]

まず「生成 AI で解く価値があるか」を切り分けます。生成 AI が向くのは、入力も出力も自然言語・画像・音声のように定型化しにくく、正解が一意に決まらない仕事です。逆に、集計・照合・ルール判定のように決定的なロジックで書ける仕事に FM を当てるのは、コストと非決定性を無駄に持ち込むだけです。

業務の型	適した手段	FM を当てると起きること
売上の集計、在庫の突合	SQL・Lambda・Step Functions	コスト増、数値の揺らぎ、監査不能
定型帳票からの項目抽出	Amazon Textract、Bedrock Data Automation	単体の FM だけだと座標情報を失う
問い合わせ回答、要約、分類	Amazon Bedrock の FM	適合
社内文書に基づく質疑応答	RAG（Knowledge Bases + ベクトルストア）	素の FM だけでは社内情報を知らない
手順を持つ業務の自動遂行	エージェント（AgentCore、Strands Agents）	単発の推論では多段の手順を完遂できない

次に、要件を四つの軸で数値化します。この四つが、以降のすべての設計判断（モデル選定・推論方式・ベクトルストア・キャッシュ）の入力になります。

軸	具体的に決めること	設計に効く先
精度	何をもって正解とするか、許容できる誤答率、根拠提示の要否	モデル選定、RAG の有無、Guardrails の contextual grounding check

軸	具体的に決めること	設計に効く先
遅延	初回トークンまでの時間、全文完了までの時間、同期か非同期か	streaming の要否、batch inference の可否、Provisioned Throughput
コスト	1リクエストあたりの入力・出力トークン数、月間リクエスト数	モデルサイズ、プロンプト圧縮、キャッシュ、model cascading
データ所在と統制	データをどのリージョンで処理してよいか、誰がアクセスできるか、ログをどこまで残すか	cross-Region inference の型、KMS、PrivateLink、CloudTrail

要件を数値化する四つの軸と、設計に効く先

この四つが、以降のすべての設計判断（モデル選定・推論方式・キャッシュ）の入力になる

軸	具体的に決めること	設計に効く先
精度	何をもって正解とするか、許容できる誤答率、根拠提示の要否	モデル選定、RAG の有無、Guardrails の contextual grounding check
遅延	初回トークンまでの時間、全文完了までの時間、同期か非同期か	streaming の要否、batch inference の可否、Provisioned Throughput
コスト	1リクエストあたりの入力・出力トークン数、月間リクエスト数	モデルサイズ、プロンプト圧縮、キャッシュ、model cascading
データ所在と統制	どのリージョンで処理してよいか、誰がアクセスできるか、ログをどこまで残すか	cross-Region inference の型、KMS、PrivateLink、CloudTrail

要件の数値化が先で、サービス選定は後。四軸を決めないまま構成を選ぶことはできない。

図 1-1　要件を数値化する四つの軸と、設計に効く先

三つ目に、integration pattern（統合パターン）を選びます。GenAI アプリケーションが既存システムとどう繋がるかは、次の四つの型に整理できます。

統合パターン	形	向く場面	主なサービス
同期リクエスト・レスポンス	呼び出して待つ	チャット、その場の要約	API Gateway、Lambda、Bedrock Converse
ストリーミング	生成しながら少しずつ返す	対話 UI、長文生成	ConverseStream、WebSocket、server-sent events
非同期キュー	投げて後で受け取る	大量処理、重い生成	Amazon SQS、Lambda、Step Functions
バッチ	まとめて一括処理	夜間の一括分類・一括要約	Amazon Bedrock batch inference、Amazon S3

四つ目に、deployment strategy（デプロイ戦略）を決めます。ここでの選択肢は「FM をどこから呼ぶか」であり、Amazon Bedrock のサーバーレス呼び出し、Amazon Bedrock の Provisioned Throughput、Amazon SageMaker AI のエンドポイント（自前でモデルをホストする）の三つが軸です。詳細は 2-2 で扱いますが、設計段階では次の対比で当たりをつけます。

デプロイ戦略	課金の形	向く場面	注意点
Amazon Bedrock オンデマンド	トークン従量	トラフィックが読めない、まず作る	スロットリングの上限がある
Amazon Bedrock Provisioned Throughput	Model Unit 単位の時間課金	定常的に高スループットが要る、カスタムモデルを使う	削除するまで課金が続く。commitment の期間中は削除できない

デプロイ戦略	課金の形	向く場面	注意点
Amazon SageMaker AI エンドポイント	インスタンス時間課金	オープンウェイトモデル、独自の推論コード、細かいチューニング	GPU インスタンスの確保と運用が要る

図 1-2　FM をどこから呼ぶか —— デプロイ戦略の三択

誤答肢の切り方として、「まず小さく試したい」「月あたり数百件」という設定に Provisioned Throughput を当てる肢は、時間課金であることと commitment の存在から切れます。逆に「常時大量、遅延を安定させたい」にオンデマンドだけを当てる肢は、スロットリングを無視しているので切れます。

1-1-2 PoC で実現性・性能・ビジネス価値を検証する [1.1]

PoC（proof-of-concept）は、本番実装に進む前に「できるか・速いか・儲かるか」を確かめる工程です。ここでいう本番実装は full-scale deployment（全面展開）であり、PoC はその前段で feasibility（実現性）・performance characteristics（性能特性）・business value（ビジネス価値）の三つを検証す

る場です。Professional の観点では、PoC は「動いた／動かない」を見る場ではなく、あとで再現できる測定を残す場です。

Amazon Bedrock で PoC を組むときの最短経路は次のとおりです。

1. コンソールの Chat playground / Text playground で、対象モデルに実データに近いプロンプトを入れて出力の質を目視で確認します。
2. Converse API で同じプロンプトをコードから叩き、入力トークン数・出力トークン数・所要時間を記録します。
3. Amazon Bedrock evaluations で、複数モデルを同じデータセットに対して比較します。
4. RAG を含む構成なら、Amazon Bedrock Knowledge Bases を小さなデータで作り、検索結果の妥当性を確認します。
5. コストを、記録したトークン数と想定リクエスト数から月額に換算します。

PoC で必ず測る三つは、feasibility（実現性）、performance characteristics（性能特性）、business value（ビジネス価値）です。

検証項目	測り方	合格・不合格の決め方
feasibility（実現性）	代表的な入力 20〜50件に対する出力の正しさ	事前に定めた正答率のしきい値
performance characteristics（性能特性）	初回トークンまでの時間、全文完了時間、同時実行時のスロットリング発生	業務が許容する応答時間
business value（ビジネス価値）	処理1件あたりの人件費削減額 − 1件あたりの推論コスト	単価が黒字か

PoC でよくある失敗は、しきい値を先に決めずに始めることです。「良さそうです」で本番に進むと、後で品質が問題になったときに戻る基準がありません。PoC の開始前に、正答率のしきい値・許容遅延・許容単価を紙に書いておくのが Professional の作法です。

もう一つの失敗は、PoC を playground だけで終えることです。playground はスロットリングもリトライも隠してしまうため、同時実行時の性能特性が測れません。少なくとも API 経由で、想定同時実行数をかけた負荷をかけて測ります。

1-1-3 標準化された技術コンポーネントと AWS Well-Architected Framework [1.1]

同じ組織で複数の GenAI 案件が走ると、チームごとにプロンプトの持ち方・ログの取り方・ガードレールの掛け方がばらつきます。standardized technical components（標準化された技術コンポーネント）は、このばらつきを消すための共通部品です。

標準化すべき部品は次のとおりです。

部品	中身	実装先
推論の入口	認証、レート制御、モデル ID の解決、ログ出力	API Gateway + Lambda（GenAI gateway。2-3 で詳述）
プロンプト	テンプレートと変数、版管理、承認	Amazon Bedrock Prompt Management
安全装置	有害表現、拒否トピック、PII マスク、根拠チェック	Amazon Bedrock Guardrails

部品	中身	実装先
監査	誰がいつどのモデルを呼んだか、入出力の記録	AWS CloudTrail、Amazon CloudWatch Logs、model invocation logging
インフラ定義	ベクトルストア、IAM ロール、エンドポイント	AWS CloudFormation、AWS CDK、AWS Service Catalog

AWS Well-Architected Framework は、設計の抜けを機械的に潰すための問いの一覧です。六つの柱（pillar）は、運用上の優秀性（Operational Excellence）、セキュリティ（Security）、信頼性（Reliability）、パフォーマンス効率（Performance Efficiency）、コスト最適化（Cost Optimization）、持続可能性（Sustainability）です。

AWS Well-Architected Tool（AWS WA Tool）は、この問いをワークロードごとにチェックし、リスクを一覧化するマネージドサービスです。lens（レンズ）を適用すると、汎用の問いに加えて領域固有の問いが加わります。生成 AI のワークロードには Generative AI Lens を適用します。試験では「AWS WA Tool Generative AI Lens」という呼び方で問われます。標準化された技術コンポーネントを複数のデプロイシナリオに横展開する際は、この観点表をチェックリストとして使い回します。

柱	生成 AI 特有の問い（Generative AI Lens の観点）
運用上の優秀性	プロンプトとモデルのバージョンを追跡できるか、出力品質を継続的に評価しているか
セキュリティ	プロンプトインジェクションに備えているか、機微情報がプロンプトやログに漏れないか

柱	生成 AI 特有の問い（Generative AI Lens の観点）
信頼性	モデルが使えないときのフォールバック先があるか、スロットリング時に再試行するか
パフォーマンス効率	ユースケースに対してモデルが過大でないか、検索の再現率と遅延の釣り合いは取れているか
コスト最適化	入力トークンを減らす余地はないか、バッチにできる処理を同期で回していないか
持続可能性	必要以上に大きなモデルや過剰な再学習を避けているか

誤答肢の切り方として、「設計レビューの標準化」に対して SageMaker Model Monitor を当てる肢は、あれが本番モデルのデータ品質・データドリフト監視の道具であって設計レビューの道具ではないため切れます。「レンズ」を問う設問で AWS Trusted Advisor を当てる肢も、あちらはアカウント全体のコスト・上限・セキュリティのチェックであり、ワークロード単位の設計レビューではないので切れます。

1-2 FM を選定して設定する [1.2]

1-2-1 ユースケースに合う FM を評価して選ぶ [1.2]

FM の選定は、performance benchmarks（性能ベンチマーク）、capability analysis（能力の分析）、limitation evaluation（限界の評価）の三点で行います。感覚で「賢そうなモデル」を選ぶのではなく、自分のデータで測って決めます。

capability analysis では、次の観点でモデルを絞ります。

観点	見るところ
モダリティ	テキストのみか、画像入力に対応するか、音声・動画を扱えるか
コンテキスト長	1リクエストで渡せる入力トークンの上限
対応言語	日本語での品質。英語最適のモデルは日本語で落ちることがある
tool use（function calling）	外部ツールを呼ぶ構造化出力に対応するか
構造化出力	JSON スキーマに沿った出力を安定して返せるか
推論の深さ	多段の論理を要する課題に耐えるか
ファインチューニング対応	カスタマイズできるモデルか

limitation evaluation では、モデルの「できないこと」を先に洗います。学習データのカットオフ以降の事実を知らないこと、根拠のない内容を自信ありげに生成すること（hallucination、幻覚）、コンテキスト長を超える入力を扱えないこと、リージョンによっては提供されていないこと、が代表です。カットオフの問題は RAG で、幻覚は Guardrails の contextual grounding check と引用の提示で、コンテキスト長は分割と要約で、リージョンの問題は cross-Region inference で、それぞれ埋めます。

performance benchmarks は、Amazon Bedrock evaluations で自分のデータに対して測ります。評価ジョブには四つの型があります。

評価の型	誰が採点するか	向く場面
プログラムによるモデル評価（automatic）	あらかじめ定義された指標の計算	素早く定量比較したい、組み込みデータセットで足りる

評価の型	誰が採点するか	向く場面
判定モデルを使うモデル評価（LLM-as-a-judge）	別の LLM が採点し、理由も返す	正解文が一意でない生成タスクを大量に採点したい
人間のワーカーを使うモデル評価	自社の従業員や専門家	業務固有の微妙な良し悪し、法務・医療などの専門判断
RAG 評価（LLM ベース）	LLM が検索結果と生成結果を採点	Knowledge Bases や外部 RAG ソースの品質を測りたい

評価ジョブの対象にできるのは、foundation model、Amazon Bedrock Marketplace のモデル、カスタマイズ済みモデル、インポートしたモデル、prompt router、Provisioned Throughput を購入したモデルです。RAG 評価では、プロンプト（ユーザーの問い）に加えて ground truth（期待される検索結果や応答）をデータセットに含める必要があります。ground truth が無いと、検索が期待どおりかを機械が判定できないためです。

誤答肢の切り方として、「生成文の良し悪しを人手を掛けずに大量に採点したい」に人間ワーカー評価を当てる肢は、要件の「人手を掛けず」に反するので切れます。「法規制に関わる回答の妥当性を専門家に確かめたい」に LLM-as-a-judge を当てる肢は、判定者が LLM である点で要件を満たしません。

1-2-2 コードを変えずにモデルを切り替えられる構成 [1.2]

モデルは頻繁に更新されます。モデル ID をアプリケーションのコードに直書きすると、切り替えのたびにデプロイが要ります。dynamic model selection（動的なモデル選択）と provider switching（プロバイダの切り替え）をコード変更なしで行うには、モデル ID を設定として外に出します。

部品	役割
AWS AppConfig	モデル ID・推論パラメータ・ルーティング規則を設定として保持し、検証（validator）を通してから配信する。段階的にロールアウトでき、CloudWatch アラームで自動ロールバックできる
AWS Lambda	AppConfig から設定を読み、その ID で Amazon Bedrock を呼ぶ薄い層。AppConfig の Lambda 拡張機能を使えばローカルにキャッシュして毎回の取得を避けられる
Amazon API Gateway	呼び出し側に対する安定した契約（エンドポイントとスキーマ）を提供する。裏側のモデルが変わっても利用側は変わらない

この三点セットが AIP-C01 でくり返し出る型です。「モデルを切り替えたいがアプリの再デプロイをしたくない」という設問文が出たら、AWS AppConfig が答えの中心にあります。

誤答肢の切り方として、AWS Systems Manager Parameter Store や AWS Secrets Manager を当てる肢は、値を保持することはできても、検証・段階的デプロイ・自動ロールバックという設定配信の機能を持たない点で AppConfig に劣ります。設問が「安全に段階的に切り替えたい」と言っているなら AppConfig です。単に値を1つ保持するだけなら Parameter Store でも成立するので、設問文の要求語を読み分けます。

抽象化のもう一段は、呼び出し API 自体をモデル非依存にすることです。Amazon Bedrock の Converse API は、messages 形式の統一されたインターフェースを提供し、メッセージに対応するすべての Bedrock モデルで同じコードが使えます。モデル固有のパラメータが要る場合だけ、

additionalModelRequestFields に個別の構造を渡します。InvokeModel はモデルごとにリクエストボディの形が違うため、モデルを差し替えるとパース処理まで書き換えが要ります。「複数モデルを切り替えて使いたい」という要件では Converse API が正解になります。

1-2-3 障害に耐える設計（cross-Region inference・circuit breaker・graceful degradation） [1.2]

FM 呼び出しは、スロットリング、リージョン障害、モデルの提供終了で失敗します。継続運用（continuous operation）を担保する道具は四つです。

第一に、Amazon Bedrock の cross-Region inference（クロスリージョン推論）です。inference profile（推論プロファイル）を指定して呼ぶと、Amazon Bedrock が複数リージョンのコンピュートに自動でルーティングします。単一リージョンのキャパシティ不足によるスロットリングを吸収でき、リージョンによってはモデルが提供されていない問題も回避できます。

profile には二種類あります。

種別	ルーティング先	データ所在	コスト	選ぶ理由
Geographic cross-Region inference	指定した地理（US、EU、APAC など）内の商用リージョン	地理の境界内に留まる	標準価格	データ所在の規制がある
Global cross-Region inference	世界中のサポート対象商用リージョン	世界のいずれか	約10%の節約	地理の制約が無く、コストを優先する

cross-Region inference について押さえる事実は次のとおりです。ルーティング自体に追加料金はかからず、価格は inference profile を呼び出したリージョンを基準に計算されます。アカウントで手動有効化していないリージョンに

もルーティングされ得ますが、通信はすべて AWS ネットワーク内に留まり、リージョン間は転送時に暗号化されます。AWS CloudTrail は呼び出し元リージョンにログを記録し、実際に処理されたリージョンは additionalEventData.inferenceRegion フィールドで分かります。そして、inference profile は Provisioned Throughput をサポートしません。

この最後の一点は取り違えが起きやすい箇所です。「高スループットを確保しつつクロスリージョンで冗長化したい」という肢は、この制約により成立しません。

第二に、circuit breaker（サーキットブレーカー）パターンです。呼び出し先が連続して失敗しているときに、無駄な再試行をやめて即座に代替へ回す仕組みです。AWS Step Functions で実装する場合、失敗回数を Amazon DynamoDB に記録し、Choice ステートでしきい値を超えているかを判定して、超えていればフォールバック用の分岐へ送ります。しきい値以下なら Retry を設定した通常の呼び出しへ進みます。

状態	動き	遷移条件
closed（閉）	通常どおり呼ぶ	連続失敗がしきい値を超えたら open へ
open（開）	呼ばずに即フォールバック	一定時間が経過したら half-open へ
half-open（半開）	試しに数件だけ呼ぶ	成功したら closed、失敗したら open

circuit breaker が無いと、障害中のエンドポイントに再試行が殺到し、復旧を遅らせるうえに呼び出し側のタイムアウトが積み上がります。

第三に、cross-Region model deployment（リージョン横断のモデル配置）です。SageMaker AI に自前でホストしているモデルや、カスタマイズしたモデルは、複数リージョンに同じ構成を配置し、Amazon Route 53 のヘルスチェック付きフェイルオーバールーティングで切り替えます。cross-Region inference は Amazon Bedrock のマネージド機能であり、SageMaker のエンドポイントには適用されないことに注意します。

第四に、graceful degradation（縮退運転）です。FM が使えないときに全面停止するのではなく、機能を落として動き続けます。

段階	動き
通常	第一候補モデルで生成
縮退1	別リージョン、または小型・別プロバイダのモデルで生成
縮退2	生成をやめ、Knowledge Bases の検索結果（引用文）だけを返す
縮退3	キャッシュ済みの定型回答、または「現在混み合っています」と有人窓口の案内を返す

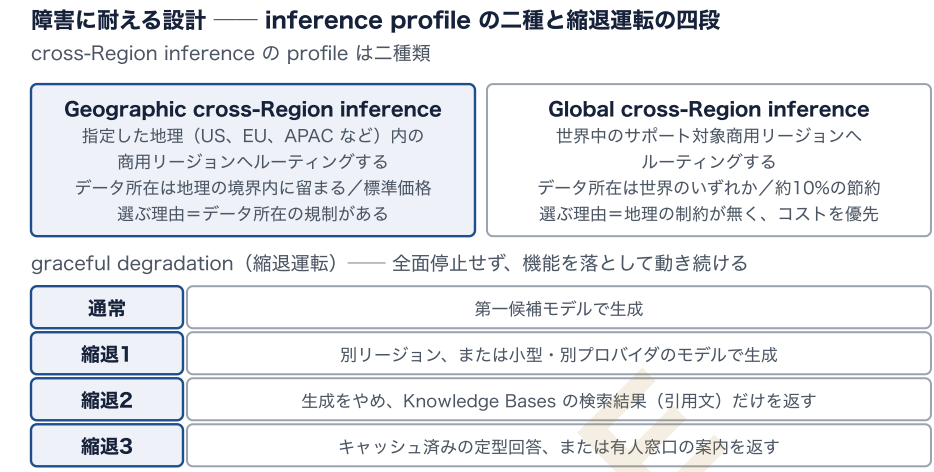

図 1-3　障害に耐える設計 —— inference profile の二種と縮退運転の四段

Amazon ElastiCache や Amazon DynamoDB に、よく来る問い合わせに対する生成結果をキャッシュしておくと、縮退時の受け皿になると同時に、平常時のコストと遅延も下がります。

1-2-4 FM カスタマイズのデプロイとライフサイクル管理 [1.2]

カスタマイズには段階があります。安いほうから順に試すのが原則で、いきなりファインチューニングに進むのは設計として誤りです。

手段	変えるもの	コスト	向く場面
プロンプトエンジニアリング	指示文だけ	ほぼゼロ	まず全部ここで試す
RAG	与える文脈	中	社内の事実・最新情報が要る

手段	変えるもの	コスト	向く場面
ファインチューニング（fine-tuning）	モデルの重み	高	文体・出力形式・専門語彙を定着させたい
継続事前学習・蒸留	モデルの重み	高	ドメイン全体の言語分布、またはコスト削減

Amazon Bedrock が提供するカスタマイズ手法は次のとおりです。

手法	与えるデータ	何が起きるか
Supervised fine-tuning（教師ありファインチューニング）	ラベル付きの入力・出力のペア	入力に対してどんな出力を返すべきかを学習し、重みが調整される
Reinforcement fine-tuning（強化ファインチューニング）	プロンプトのデータセット、または既存の Bedrock 呼び出しログ。加えて AWS Lambda で書いた報酬関数	報酬関数のスコアを手掛かりに反復的に学習する。ラベル付きの正解ペアを用意しなくてよい
Distillation（蒸留）	ユースケース固有のプロンプト（任意でプロンプト・応答のペア）	大きな teacher モデルの応答を Amazon Bedrock が生成し、それで小さい student モデルをファインチューニングする

図 1-4　FM カスタマイズは安いほうから順に試す

カスタマイズの課金は、学習では処理トークン数（学習データのトークン数 × エポック数）で、加えてモデルごとに月額のストレージ料金がかかります。そしてカスタマイズしたモデルで推論を行う設定は二つです。ひとつは Provisioned Throughput を購入して専用の処理能力を確保する方法、もうひとつは custom model deployment（カスタムモデルのデプロイ）を作成してオンデマンド推論で使う方法です。オンデマンドは使った分だけの課金で専用の計算資源を用意せずに済み、作成したデプロイの ARN を modelId に指定して呼びます。ただし利用条件があり、US East（N. Virginia）または US West（Oregon）であること、2025-07-16 以降にカスタマイズしたモデルであること、対応するベースモデルであることが要ります。保証されたスループットと安定した低遅延が要件なら Provisioned Throughput です。

Amazon SageMaker AI 側でカスタマイズする場合は、parameter-efficient fine-tuning（PEFT、パラメータ効率の良い適応）が中心になります。代表が

LoRA（low-rank adaptation、低ランク適応）と adapter（アダプター）です。

手法	更新するパラメータ	利点	運用上の効き方
フルファインチューニング	全パラメータ	適応度が最も高い	GPU メモリと時間を大量に消費し、成果物もモデル全体
LoRA	元の重みは凍結し、低ランクの差分行列だけを学習	学習が速く、成果物が小さい	同じベースモデルに複数の LoRA を切り替えて載せられる
adapter	層の間に小さな追加モジュールを挿入して学習	同上	タスクごとにモジュールを差し替えられる

LoRA の運用上の利点は、成果物が軽いため「顧客ごと・部門ごとに別の適応」を1つのベースモデルの上で切り替えられることです。フルファインチューニングだと、部門の数だけモデル全体を保管しホストすることになります。

ライフサイクル管理は SageMaker Model Registry を中心に組みます。ここで扱う対象は domain-specific（領域特化）な fine-tuned model（微調整済みモデル）であり、LoRA（low-rank adaptation）や adapters といった parameter-efficient adaptation techniques（パラメータ効率のよい適応技法）で作った成果物も同じ流れに載せます。

段階	やること	道具
登録	学習済みモデルを model package group に登録し、バージョンを採番する	SageMaker Model Registry
承認	評価指標がしきい値を超えたら Approved にする	Model Registry の approval status、Lambda

段階	やること	道具
デプロイ	Approved になったバージョンを自動でエンドポイントへ	EventBridge + CodePipeline、SageMaker Pipelines
監視	入力データの分布ずれ、品質劣化を検知する	SageMaker Model Monitor、CloudWatch
ロールバック	問題が出たら直前のバージョンへ戻す	エンドポイントの production variant の重み変更、blue/green
退役	使わなくなったバージョンを Archived にし、エンドポイントを削除する	Model Registry、SageMaker AI

自動デプロイパイプラインの安全弁は、デプロイ方式そのものに持たせます。SageMaker AI のエンドポイント更新では、blue/green（新旧を並べて一気に切り替える）と canary・linear（一部のトラフィックだけ新版に流し、段階的に増やす）の型があり、CloudWatch アラームを設定しておけば、しきい値を割ったときに自動でロールバックできます。

誤答肢の切り方として、「新モデルの品質が悪かったときに即座に戻したい」に対して「エンドポイントを削除して作り直す」肢は、切り替え中に無応答の時間が生まれるので不正解です。「モデルの精度劣化を継続的に見張りたい」に CloudTrail を当てる肢は、CloudTrail が API 呼び出しの監査ログであってモデル品質の指標ではないため切れます。そこは SageMaker Model Monitor です。

1-3 FM 消費のためのデータ検証と処理パイプラインを実装する [1.3]

FM に渡すデータの質が、出力の質の上限を決めます。この Task statement は、生のデータを FM が食べられる形にするまでの工程を扱います。四つの細目は、検証（validation）、複雑なデータ型の処理（text・image・audio・tabular）、モデル仕様に合わせた整形（formatting）、品質の向上（enhancement）です。

1-3-1 データ検証ワークフロー（AWS Glue Data Quality・SageMaker Data Wrangler） [1.3]

data validation（データ検証）は、FM に渡す前にデータが品質基準（quality standards）を満たしているかを機械で確かめる工程です。RAG のインデックスに壊れた文書が混じると、検索は「それらしく」壊れた根拠を返し、生成もそれに引きずられます。入口で止めるのが最も安いのです。

道具	何をするか	向く場面
AWS Glue Data Quality	Data Quality Definition Language（DQDL）でルールを書き、データセットに対して評価する。データをプロファイリングしてルールの推奨も出せる	S3 のデータレイク、Glue Data Catalog に載っているテーブルの定期チェック
Amazon SageMaker Data Wrangler	視覚的なインターフェースでデータの読み込み・可視化・変換を行い、欠損値や外れ値を把握して変換フローを組む	前処理の設計、探索的な確認

道具	何をするか	向く場面
カスタム Lambda 関数	独自のルール（文字数、文字コード、禁止語、スキーマ適合）を書いて判定する	標準的なルールで表せない業務固有の検証
Amazon CloudWatch メトリクス	検証で弾いた件数・合格率をカスタムメトリクスとして出し、しきい値でアラームを鳴らす	品質の継続監視、劣化の早期検知

FM 向けのデータで特に効く検証項目は次のとおりです。

検証項目	見るところ	落ちたときの影響
完全性	本文が空でないか、抽出に失敗した PDF が混じっていないか	空チャンクが検索結果を汚す
文字コード	UTF-8 か、文字化けしていないか	埋め込みベクトルが意味を持たなくなる
重複	同じ文書が複数回入っていないか	検索結果が同じ内容で埋まり、多様性が落ちる
鮮度	更新日が古すぎないか、失効した規程でないか	古い根拠で自信ありげに答える
機微情報	個人情報が含まれていないか	インデックス経由で漏えいする
長さ	チャンク化前の文書が極端に短くないか、極端に長くないか	短すぎると文脈を失い、長すぎると分割が粗くなる

機微情報の検出には、Amazon Macie（S3 上のデータをスキャンして機微情報を発見・分類する）と、Amazon Comprehend の PII 検出を使い分けます。

Macie は「バケットに何が置かれているか」を発見する側、Comprehend はパイプラインの中でテキスト1件ごとに検出する側です。実行時の入出力に対しては Amazon Bedrock Guardrails の sensitive information filters（機微情報フィルター）で遮断・マスクします。

検証で不合格になったデータの扱いは、次の三通りを設計時に決めておきます。破棄する、隔離用の S3 プレフィックスへ退避して人が見る、自動で補正して再検証する。黙って通すという選択肢だけは存在しません。

1-3-2 テキスト・画像・音声・表形式データの処理 [1.3]

FM に入れるデータは text（テキスト）だけではありません。image（画像）、audio（音声）、tabular（表形式）のそれぞれに専用の処理が要ります。

データ型	前処理でやること	主なサービス
text（テキスト）	抽出、正規化、分割、不要部分（ヘッダー・フッター・目次）の除去	Amazon Textract、Lambda、Amazon Bedrock
image（画像）	リサイズ、形式変換、base64 エンコード、テキストの抽出、説明文の生成	Amazon Rekognition、Amazon Textract、Amazon Bedrock のマルチモーダルモデル
audio（音声）	文字起こし、話者分離、タイムスタンプ付与	Amazon Transcribe
tabular（表形式）	列の意味を説明する見出しの付与、行の自然文化、集計値の事前計算	SageMaker Processing、AWS Glue、Lambda

Amazon Bedrock のマルチモーダルモデルは、画像とテキストを同じリクエストに含めて渡せます。Converse API では、messages の content 配列に

text ブロックと image ブロックを並べる形になります。画像から表を読み取らせる、図面の内容を説明させる、といった処理がこれで組めます。

Amazon Transcribe は音声を文字に起こすサービスです。生成 AI のパイプラインでは、会議録音・コールセンターの通話・動画の音声を文字に変換し、その文字列を FM に渡す、という繋ぎ方をします。話者分離（speaker diarization）を有効にすると「誰が話したか」が付くので、要約の質が上がります。カスタム語彙を登録すると、社内用語や製品名の書き起こし精度が上がります。

Amazon SageMaker Processing は、前処理・後処理・評価をマネージドなコンテナで実行するジョブです。数百万件の画像リサイズや、大規模なテキスト正規化のように、Lambda の実行時間やメモリでは収まらない処理をここに逃がします。Lambda との切り分けは明確で、Lambda はイベント駆動の軽い処理、SageMaker Processing はまとまった量のバッチ処理です。

表形式データを FM に渡すときの設計上の判断は、「表のまま渡すか、自然文に開くか」です。

渡し方	形	向く場面	弱点
CSV や Markdown 表のまま	行と列の構造を保つ	列同士の比較、集計値の読み取り	行数が多いとトークンを大量に消費する
1行1文の自然文に開く	「商品Aの2026年8月の売上は…」	RAG のチャンクとして埋め込む	行間の関係が失われる
事前に集計してから渡す	集計結果だけを渡す	傾向の説明、要約	細部の問いに答えられない

advanced multimodal pipeline architecture（高度なマルチモーダルパイプライン）は、これらを組み合わせた形になります。典型は、S3 にファイルが置

かれたら EventBridge が発火し、Step Functions が形式ごとに分岐して、画像なら Textract と Bedrock のマルチモーダルモデル、音声なら Transcribe、文書なら Bedrock Data Automation に流し、すべてをテキスト化してから Knowledge Bases に取り込む、という構成です。

Amazon Bedrock Data Automation（BDA）は、文書・画像・動画・音声といった非構造化コンテンツを、生成 AI で構造化データに変換するサービスです。分類・抽出・正規化・検証といった処理を自前でオーケストレーションせずに済みます。visual grounding（出力がどこから来たかを元の文書上で示す）と confidence score（信頼度スコア）が組み込まれており、抽出結果の確からしさを機械で判定できます。RAG の前処理としても使え、Knowledge Bases の parser として BDA を選ぶと、動画やコンテンツはまず文字（文字起こしやシーンの要約）に変換され、その変換後のテキストに対して通常のテキストチャンク戦略が適用されます。

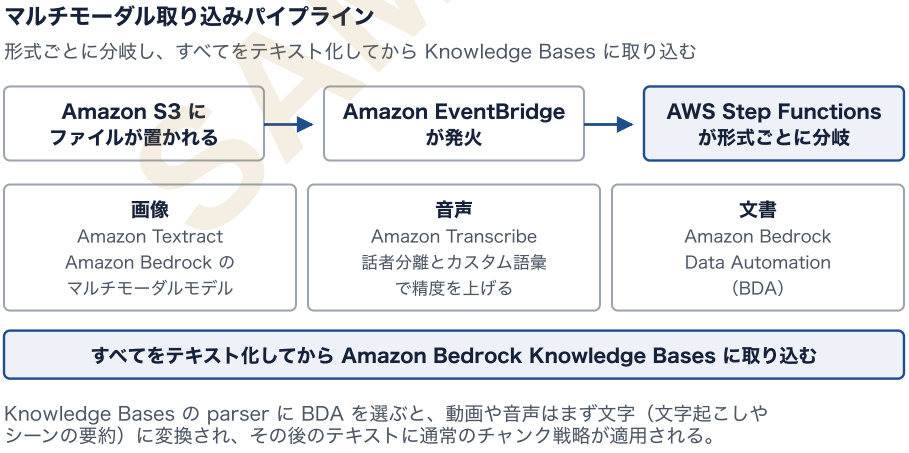

図 1-5　マルチモーダル取り込みパイプライン

1-3-3 モデル固有の要件に合わせた入力整形 [1.3]

ここで整形の基準になるのは model-specific requirements（モデル固有の要件）です。同じ入力でも、モデルごとに受け付ける形式・トークン上限・画像の解像度上限が違います。特に dialog-based applications（対話型アプリケーション）では、役割（system／user／assistant）を持つメッセージ配列として組み立てる必要があり、単なる文字列連結では意図した振る舞いになりません。

FM ごとに入力の形は違います。Amazon Bedrock API へのリクエストは JSON formatting（JSON 整形）が基本で、SageMaker AI のエンドポイントは自分が用意した推論コードが受け取れる形（JSON、CSV、JSON Lines など）、対話型アプリケーションは conversation formatting（会話形式の整形）です。

Amazon Bedrock の推論 API には二系統あります。

API	形	使いどころ
InvokeModel / InvokeModelWithResponseStream	body がモデル固有の JSON。モデルを変えるとボディの構造も変わる	そのモデル固有のパラメータをすべて使いたい
Converse / ConverseStream	messages 形式の統一インターフェース。すべてのメッセージ対応モデルで同じコードが動く	複数モデルを切り替える、標準的な対話を組む

Converse API のリクエストの骨格は次のとおりです。

フィールド	中身
modelId	モデル ID、または inference profile の ID

フィールド	中身
messages	role（user / assistant）と content の配列。content には text、image、document、toolUse、toolResult などのブロックを並べる
system	システムプロンプト。役割定義や制約をここに置く
inferenceConfig	maxTokens、temperature、topP、stopSequences
toolConfig	ツール（関数）の定義。tool use を使う場合
guardrailConfig	適用する Guardrails の ID とバージョン
additionalModelRequestFields	Converse の共通フィールドに無い、モデル固有のパラメータ

Converse API は bedrock-runtime エンドポイントでのみ利用できます。Mistral AI と Meta のモデルでは、Converse API が入力をモデル固有のプロンプトテンプレートに埋め込んで会話を成立させます。

推論パラメータの意味は取り違えが起きやすいので整理します。

パラメータ	何を変えるか	上げるとどうなるか
temperature	出力の確率分布の平坦さ	多様になるが、事実性が落ちる
topP	累積確率が topP に達するまでの候補から選ぶ	候補が広がり多様になる
maxTokens	生成する出力トークンの上限	長く書けるが、コストと遅延が増える

パラメータ	何を変えるか	上げるとどうなるか
stopSequences	この文字列が出たら生成を止める	出力の末尾を制御できる

事実の抽出や分類のように「一意の正解がある」タスクでは temperature を低く、キャッチコピー生成のように「多様性が価値」のタスクでは高くします。temperature と topP を同時に大きく動かすと制御が効かなくなるため、どちらか一方を主に動かすのが定石です。

conversation formatting（対話の整形）では、会話履歴をどこまで送るかが設計の勘所です。全履歴を毎回送るとトークンが線形に増え、コストと遅延が上がります。対処は三つです。直近 N ターンだけを送る、古いターンを FM で要約して1つのメッセージに畳む、そして必要な過去情報だけを検索して差し込む（会話履歴も RAG の対象にする）。履歴の保存先は Amazon DynamoDB が定番で、セッション ID をパーティションキー、タイムスタンプをソートキーにし、TTL（Time to Live）で自動失効させます。

1-3-4 入力データの品質向上（Amazon Bedrock・Amazon Comprehend・Lambda）[1.3]

入力の質を上げる打ち手は三つに整理できます。

打ち手	道具	具体例
整形し直す（reformat）	Amazon Bedrock の FM 自身	崩れた OCR テキストの復元、箇条書きへの整理、専門用語の統一
意味を取り出す（extract entities）	Amazon Comprehend	人名・組織名・日付・金額をエンティティとして抽出し、メタデータに付与する

打ち手	道具	具体例
正規化する（normalize）	Lambda 関数	全角・半角の統一、日付形式の統一、単位の統一、余分な空白と制御文字の除去

Amazon Comprehend は自然言語処理のマネージドサービスで、エンティティ認識、キーフレーズ抽出、感情分析、言語判定、トピックモデリング、PII 検出、そしてカスタム分類とカスタムエンティティ認識を提供します。GenAI パイプラインでの役割は二つで、前処理としてのメタデータ生成（抽出したエンティティを検索のフィルター条件にする）と、実行時の意図認識（intent recognition）です。

FM で前処理をする判断基準は、「決定的なルールで書けるか」です。全角・半角の変換や日付形式の統一は Lambda で決定的に書けるので FM は不要です。逆に、文脈を読まないと直せない崩れ（OCR の誤認識、口語の書き起こしの整理）は FM の出番です。ここを逆にすると、コストが上がり、しかも結果が毎回わずかに変わって再現性を失います。

誤答肢の切り方として、「日付表記の揺れを統一したい」に Amazon Bedrock の FM を当てる肢は、決定的処理に非決定的な道具を当てているので設計として劣ります。「大量の文書から組織名を抜き出してタグにしたい」に FM を当てる肢も、Amazon Comprehend のほうが安く安定するため、コスト要件がある設問では切れます。

1-4 ベクトルストアソリューションを設計して実装する [1.4]

ここからが RAG の中核です。1.4 は「ベクトルをどこにどう置くか」、1.5 は「どう取り出すか」を扱います。

1-4-1 FM 拡張のためのベクトルデータベース設計 [1.4]

vector database（ベクトルデータベース）は、テキストや画像を数値ベクトル（embedding、埋め込み）に変換して保存し、意味の近さで検索できるようにした保管庫です。従来のキーワード検索が「語が一致するか」を見るのに対し、semantic retrieval（意味検索）は「意味が近いか」を見ます。「解約したい」と書かれた問い合わせが「退会手続き」の文書に当たるのは、この性質によります。

Amazon Bedrock Knowledge Bases は、この一式（取り込み・チャンク化・埋め込み・インデックス・検索）をマネージドで提供します。データソース（S3 など）を指定すると、Amazon Bedrock がまず文書をチャンクに分割し、埋め込みモデルでベクトルに変換してベクトルインデックスに書き込み、元の文書へのマッピングを保持します。

選べるベクトルストアと、その選び分けは次のとおりです。

ベクトルストア	形	向く場面	注意点
Amazon OpenSearch Serverless	マネージドのベクトル検索。Knowledge Bases から自動作成もできる	標準的な RAG、ハイブリッド検索が要る	常時稼働のコストがかかる
Amazon OpenSearch Service（マネージドクラスター）	自分でクラスターを持つ。Neural plugin で Amazon Bedrock の埋め込みモデルと連携できる	シャード設計まで自分で握りたい、既存の検索基盤がある	運用の手間

ベクトルストア	形	向く場面	注意点
Amazon Aurora PostgreSQL（pgvector 拡張）	リレーショナル DB の中にベクトル列を持つ	業務データと同じトランザクションで扱いたい、SQL で絞り込みたい	大規模での検索性能はチューニング次第
Amazon S3 Vectors	ベクトル専用のバケットとインデックス。インフラのプロビジョニング不要	大量のベクトルを安く置きたい、クエリ頻度が低い	頻繁な高 QPS 用途には向かない
Amazon Neptune Analytics	グラフとベクトルを併せ持つ	関係構造を辿りながら検索したい	用途が限定的
Amazon DocumentDB、Amazon MemoryDB	それぞれのデータストアにベクトル検索を足す	既存のデータストアをそのまま使いたい	—

Amazon S3 Vectors は、AI エージェント・推論・RAG・意味検索向けに用途を絞った、コスト最適化のベクトルストレージです。構成要素は三つで、vector bucket（ベクトル専用のバケット種別）、vector index（バケット内でベクトルを整理する単位。類似度検索はこのインデックスに対して行う）、vector（インデックスに格納する個々のベクトル）です。

S3 Vectors の性質として押さえるのは次の点です。低頻度のクエリでサブ秒、高頻度のクエリで最短100ミリ秒程度の応答を狙う設計であり、クエリ頻度が低いワークロードに向きます。書き込みは強い整合性を持ち、書いた直後に読めます。ベクトルにはメタデータをキー・バリューで付けられ、既定ではすべてのメタデータがフィルター可能で、明示的に非フィルター対象に指定することもできます。メタデータの型は文字列・数値・真偽値・リストです。アクセス制御は IAM と Service Control Policies で行い、名前空間は s3vectors と

Amazon S3 本体（s3）とは別です。ベクトルバケットではすべての Block Public Access 設定が常に有効で、無効にできません。

S3 Vectors は他サービスと繋がります。Amazon OpenSearch Service へインデックスのスナップショットをエクスポートして、高 QPS・低遅延が要る場面だけ OpenSearch Serverless に載せる、という使い分けができます。Amazon Bedrock Knowledge Bases のベクトルストアとしても選べます。

「hierarchical organization（階層的な整理）」は、Knowledge Bases の hierarchical chunking（階層チャンク化）で実現します。詳細は 1-5-1 で扱いますが、ここで重要なのは、S3 Vectors をベクトルストアに使う場合には hierarchical chunking が推奨されないという点です。チャンクのトークン数が大きい（合計8,000トークンを超えるような）構成では、メタデータのサイズ上限に当たる可能性があるためです。

OpenSearch の Neural plugin は、OpenSearch の中で埋め込み生成と検索を繋ぐ仕組みです。Amazon Bedrock の埋め込みモデルをコネクタとして登録しておくと、文書の取り込み時に ingest pipeline が自動で埋め込みを作り、検索時にはクエリ文字列を自動でベクトル化して neural query を実行します。アプリケーション側で埋め込み生成のコードを書かずに済むのが利点です。topic-based segmentation（トピック別の切り分け）は、トピックごとにインデックスを分けるか、トピックをメタデータのフィルターとして持つ形で実装します。

Amazon RDS と Amazon S3 の組み合わせは、「文書の実体は S3、ベクトルとメタデータは DB」という分業です。ベクトルストアに文書全文を入れると容量とコストが膨らむので、チャンクのテキストと S3 の場所（バケット名とオブジェクトキー）だけを持ち、原本を提示するときは S3 の署名付き URL を返します。

Amazon DynamoDB をベクトルデータベースと組み合わせる形は、メタデータとベクトルの分離管理です。DynamoDB に文書 ID をキーとしたメタデータ（作成者、部門、更新日、アクセス権）を持ち、ベクトル検索で得た ID を使って DynamoDB を引き、権限を確認してから提示する、という流れになります。

1-4-2 メタデータフレームワークの設計 [1.4]

metadata framework（メタデータの枠組み）は、検索の精度（search precision）と文脈認識（context awareness）を上げるための設計です。ベクトルの近さだけで検索すると、「意味は近いが古い規程」「意味は近いが他部門の文書」が上位に来ます。メタデータで絞り込めば、この種の誤検索が消えます。

メタデータの種類	例	どう効くか
時間	作成日、更新日、有効期限	失効した文書を除外する、新しいものを優先する
出所	作成者、部門、システム名	権限に応じて見せる範囲を変える
分類	製品カテゴリ、業務領域、機密区分	問い合わせの領域に絞る
構造	文書内の章番号、ページ番号、見出し	引用箇所を正確に示す

Amazon S3 では、メタデータの持ち方が三通りあります。

持ち方	形	特徴
S3 オブジェクトメタデータ	オブジェクトに付随するシステム定義・ユーザー定義のメタデータ	更新日時などのシステム定義値が自動で付く

持ち方	形	特徴
S3 オブジェクトタグ	キー・バリューのタグ	IAM ポリシーの条件に使える。ライフサイクル規則の対象指定にも使える
サイドカーファイル	原本と同名の .metadata.json ファイルを隣に置く	Amazon Bedrock Knowledge Bases が読み取り、フィルター可能な属性として取り込む

Knowledge Bases でメタデータフィルターを効かせるには、S3 データソースの場合、対象ファイルと同じプレフィックスに `<ファイル名>.metadata.json` を置き、その中に metadataAttributes として属性を書きます。取り込み後、Retrieve や RetrieveAndGenerate のリクエストで filter を指定すると、その属性で絞った上で意味検索が走ります。

Knowledge Bases は取り込み時に、システム側のメタデータも自動で付けます。たとえば x-amz-bedrock-kb-document-page-number（ページ番号）です。ただし、チャンク戦略に「チャンク化しない（no chunking）」を選んだ場合は、引用でページ番号を見ることも、このフィールドでフィルターすることもできません。

図 1-6　メタデータの持ち方と、フィルターが効く経路

tagging system for domain classification（ドメイン分類のタグ体系）を設計するときの原則は三つです。第一に、値の語彙を固定します（自由記述にすると「営業部」「営業本部」「Sales」が乱立します）。第二に、階層を持たせるなら区切り文字を決めます（`legal/contract/nda` のような形）。第三に、付与を自動化します。Amazon Comprehend のカスタム分類器か、FM による分類で付け、人手のタグ付けに依存しない設計にします。

1-4-3 大規模でのベクトル検索性能（シャーディング・マルチインデックス・階層インデックス）[1.4]

ベクトル件数が増えると、検索の遅延と再現率（recall）が悪化します。high-performance vector database architecture（高性能なベクトルデータベースの構成）は、次の三つで組みます。

第一に、OpenSearch の sharding strategy（シャーディング戦略）です。インデックスは複数の primary shard に分割され、各 shard に replica を持てます。ベクトル検索では、検索リクエストがすべての shard に配られ、各 shard の結果がマージされます。

設計項目	考え方
primary shard 数	データ量とノード数から決める。多すぎると shard あたりの検索オーバーヘッドが積み上がり、少なすぎると1 shard が大きくなりすぎて遅くなる
replica 数	読み取りスループットと可用性を上げる。書き込みコストは増える
shard の配置	1ノードに偏らせない。ベクトルインデックスはメモリを多く使うため、ノードのメモリが律速になる

ベクトル検索で特に効くのは、インデックスがメモリに載るかどうかです。近似最近傍探索（approximate nearest neighbor、ANN）のグラフ構造はメモリ上に保持されるため、ノードのメモリ量に対してベクトル数と次元数が大きすぎると、性能が急に落ちます。次元数を落とす（後述の Titan Text Embeddings V2 の 512 や 256 次元）ことは、精度をわずかに犠牲にしてメモリとコストを大きく下げる打ち手です。

第二に、multi-index approach（マルチインデックス方式）です。ドメインごとにインデックスを分けます。

方式	形	利点	欠点
単一インデックス＋メタデータフィルター	全部を1つに入れ、検索時に絞る	運用が単純、横断検索が容易	インデックスが巨大化する

方式	形	利点	欠点
ドメイン別マルチインデックス	法務・人事・製品で別インデックス	各インデックスが小さく速い、権限分離が明快	横断検索は複数インデックスへの問い合わせが要る

判断基準は、横断検索の必要性と権限分離の厳しさです。「部門をまたいで見せてはいけない」が要件なら、マルチインデックスにしてインデックス単位で IAM を効かせるほうが安全です。

第三に、hierarchical indexing technique（階層インデックス）です。まず粗い単位（文書やセクションの要約ベクトル）で候補を絞り、次に細かい単位（チャンク）で精査する二段構えにします。全チャンクを毎回総当たりするより速く、文脈も保てます。Knowledge Bases の hierarchical chunking は、この考え方をチャンク側で実現したものです。

1-4-4 既存システムとの統合コンポーネント [1.4]

社内の知識は、S3 だけにあるわけではありません。document management system（文書管理システム）、knowledge base（ナレッジベース製品）、internal wiki（社内 wiki）に散っています。これらを GenAI アプリケーションに繋ぐ方法は三つです。

方法	形	向く場面
Knowledge Bases のデータソースコネクタ	Amazon Bedrock Knowledge Bases が対応するデータソースを直接指定する	対応済みのソース。同期の仕組みまで込みで済む
Amazon Kendra	多数のコネクタを持つインテリジェント検索サービス。検索結果を FM に渡す	全社横断の検索、権限を反映した検索が要る

方法	形	向く場面
独自の取り込みパイプライン	各システムの API を Lambda で叩き、S3 に正規化して置いてから Knowledge Bases に取り込む	対応コネクタが無い社内独自システム

Amazon Kendra と Amazon Bedrock Knowledge Bases の使い分けは頻出です。

観点	Amazon Kendra	Amazon Bedrock Knowledge Bases
本質	検索サービス。多数のコネクタと、ソース側の権限を反映するアクセス制御を持つ	RAG の基盤。チャンク化・埋め込み・検索・生成までを一体で提供する
出力	検索結果（文書・抜粋・回答候補）	検索結果、または引用付きの生成応答
権限	ユーザー・グループ単位のアクセス制御でソース側の権限を反映できる	ベクトルストアのメタデータフィルターとアプリ側の制御で実装する
選ぶ理由	多様なソースを繋ぎ、権限を厳密に反映したい	Amazon Bedrock の FM と組み合わせて回答生成まで一気に作りたい

独自パイプラインを作る場合の正規化の型は、「原本を S3 に置き、テキストとメタデータを別に持つ」です。wiki の HTML をそのまま入れると、ナビゲーションや広告的な要素がチャンクに混ざります。本文だけを抜き、見出し構造をメタデータに残してから取り込みます。

1-4-5 ベクトルストアの鮮度を保つデータ保守 [1.4]

ベクトルストアは作って終わりではありません。元の文書が更新されたのにベクトルが古いままだと、RAG は古い情報を根拠として提示します。data maintenance system（データ保守の仕組み）は四つの部品でできています。

部品	中身	実装
incremental update mechanism（増分更新）	変わったものだけを再取り込みする	Knowledge Bases の同期は、前回から変更のあった文書だけを処理する。Lambda で自作する場合はハッシュや更新日時を記録して比較する
real-time change detection（リアルタイム変更検知）	変更を即座に捉える	S3 のイベント通知、Amazon EventBridge、DynamoDB Streams、AWS DMS の CDC
automated synchronization workflow（自動同期）	検知から取り込みまでを自動で回す	EventBridge → Lambda → StartIngestionJob、または Step Functions
scheduled refresh pipeline（定期再構築）	定期的に全体を作り直す	EventBridge Scheduler で日次・週次に起動

増分更新と定期再構築は、どちらか一方ではなく併用します。増分更新は速いが、削除の取りこぼしや失敗の蓄積で少しずつずれます。定期的に全体を作り直す仕組みを併せ持ち、ずれをリセットします。

削除の扱いは設計上の落とし穴です。元の文書が消えたときにベクトルを消し忘れると、存在しない文書を根拠に回答が生成されます。S3 の削除イベントを拾って、対応するチャンクをインデックスから削除する経路を必ず作ります。

再構築中に検索が止まらないようにするには、blue/green の考え方をインデックスに適用します。新しいインデックスを別名で作り、取り込みが完了して検証に通ったら、エイリアス（OpenSearch の index alias）を切り替えます。切り替えは一瞬で、失敗したら戻すだけです。

誤答肢の切り方として、「文書が更新されたら即座に検索結果へ反映したい」に日次のバッチ再構築を当てる肢は、要件の「即座に」に反します。逆に「1日1回の反映で十分、コストを抑えたい」にリアルタイム検知の構成を当てる肢は、過剰設計としてコスト要件に負けます。設問の時間要件を読むのが分かれ目です。

1-5 FM 拡張のための検索メカニズムを設計する [1.5]

1.4 で「ベクトルをどこに置くか」を決めました。1.5 は「そこから何をどう取り出すか」です。RAG の回答品質は、生成モデルの賢さよりも検索（retrieval）の設計で決まります。取れていない情報は、どんな FM でも書けません。

この Task statement は六つの工程に分かれます。document segmentation（文書分割）、embedding（埋め込み）の選定、vector search solution の配備、advanced search architecture（高度な検索構成）、query handling（クエリ処理）、そして FM から呼ぶための consistent access mechanism（一貫したアクセス手段）です。上流から順に見ていきます。

1-5-1 文書分割（chunking）の設計 [1.5]

document segmentation は、長い文書を検索単位に切り分ける工程です。切り方が粗いと 1 チャンクに無関係な話題が混ざって検索精度が落ち、細かすぎると文脈が切れて回答が断片的になります。

Amazon Bedrock Knowledge Bases が提供する chunking capabilities（分割機能）は次のとおりです。既定値は公式ドキュメントの記述に基づきます。

分割方式	設定する値	動き	向く文書
Default chunking（既定分割）	なし	およそ 300 トークンに分割し、文の境界を保つ	まず動かす、素性の分からない文書
Fixed-size chunking（固定サイズ分割）	1 チャンクの最大トークン数、overlap percentage（重なりの割合）	指定トークン数で機械的に切り、隣接チャンク間を指定割合だけ重ねる	形式の揃った文書、分割条件を厳密に制御したい場合
Hierarchical chunking（階層分割）	parent chunk のトークン数、child chunk のトークン数、overlap tokens（重なりトークン数）	child で検索し、ヒットした child を parent に置き換えて FM に渡す	章立てのある規程・マニュアル
Semantic chunking（意味分割）	maximum tokens、buffer size、breakpoint percentile threshold	文の意味的な距離を測り、話題が変わる位置で切る	見出しの無い議事録・報告書
No chunking（分割しない）	なし	1 ファイル＝1 チャンク	すでに小さく分割済みのファイル群

Knowledge Bases の分割方式は五つ

分割方式	動き	向く文書
Default chunking	およそ 300 トークンに分割し、文の境界を保つ	まず動かす、素性の分からない文書
Fixed-size chunking	指定トークン数で機械的に切り、隣接チャンク間を overlap percentage だけ重ねる	形式の揃った文書、分割条件を厳密に制御したい場合
Hierarchical chunking	child で検索し、ヒットした child を parent に置き換えて FM に渡す	章立てのある規程・マニュアル
Semantic chunking	文の意味的な距離を測り、話題が変わる位置で切る	見出しの無い議事録・報告書
No chunking	1 ファイル＝1 チャンク	すでに小さく分割済みのファイル群

hierarchical chunking は S3 vector bucket をベクトルストアにする構成では推奨されない。
semantic chunking は判定に基盤モデルを使うため追加コストが発生する。

図 1-7　Knowledge Bases の分割方式は五つ

fixed-size chunking は Lambda functions で自前実装することもできます。Knowledge Bases の管理下で済ませるか、Lambda で custom processing（カスタム処理）を書くかの判断軸は、切り分けが content structure（コンテンツ構造）に依存するかどうかです。HTML の見出しタグ、Markdown の階層、社内規程の「第〇条」といった構造に沿って hierarchical chunking を行いたい場合は、custom transformation を Lambda で実装します。

四つの注意点を押さえてください。第一に、hierarchical chunking では child が parent に置き換えられるため、返るチャンク数が要求した件数より少なくなることがあります。第二に、hierarchical chunking は S3 vector bucket をベクトルストアにする構成では推奨されておらず、合計 8000 トークンを超えるような大きな分割ではメタデータサイズの上限に当たることがあります。第三に、semantic chunking は判定に基盤モデルを使うため追加コストが発生します。第四に、no chunking を選ぶと引用にページ番号を出せ

ず、x-amz-bedrock-kb-document-page-number のメタデータでの絞り込みもできなくなります。

overlap（重なり）の役割も問われます。チャンクの境界に答えがまたがると、どちらのチャンクにも答えが半分しか入らず検索に引っかかりません。重なりを持たせることでこれを防ぎますが、重なりを大きくするほどチャンク数と保存コストと埋め込みコストが増えます。

誤答肢の切り方として、「章ごとの文脈を保ったまま、検索は細かい単位で当てたい」という要件に fixed-size chunking を当てる肢は、構造を無視する点で hierarchical chunking に劣ります。逆に「フォーマットが揃った短いレコードが数百万件」に semantic chunking を当てる肢は、FM 呼び出しのコストだけが増えて効果が薄く、切れます。

音声・動画・画像を含む multimodal content（マルチモーダルコンテンツ）は扱いが別です。Nova multimodal embeddings を使う場合、分割は埋め込みモデル側で行われ、音声・動画のチャンク長を 1〜30 秒（既定 5 秒）で設定します。Bedrock Data Automation（BDA）parser を使う場合は、いったんテキスト（文字起こしやシーン要約）に変換してから、上の text chunking 方式が適用されます。テキスト向けの分割設定は、Nova multimodal embeddings 利用時には音声・動画・画像には効きません。

1-5-2 embedding solution の選定と設定 [1.5]

embedding（埋め込み）は、テキストを固定長の数値ベクトルに変換する処理です。semantic search（意味検索）はこのベクトル同士の近さで「意味が近い」を判定します。選定の軸は dimensionality（次元数）と domain fit（対象領域との相性）の二つです。

検討軸	見るところ	判断
dimensionality（次元数）	256／512／1024 など。Amazon Titan embeddings は次元数を選べる	次元が大きいほど表現力は上がるが、保存容量と検索コストと遅延が増える
domain fit（領域適合）	医療・法務・社内固有語をどれだけ区別できるか	自社データで実測する。汎用ベンチマークの順位だけで決めない
対応言語	日本語を含む多言語対応か	日本語文書に英語専用モデルを当てない
入力トークン上限	1 回の埋め込みに入れられる長さ	chunking の最大トークン数はこの上限を超えられない
マルチモーダル対応	画像・音声を同じ空間に埋め込めるか	画像検索を含むなら multimodal embeddings

Amazon Bedrock embedding models の performance characteristics（性能特性）を評価する手順は、モデル選定と同じ型です。自社の代表クエリと正解チャンクの組を 50〜200 件用意し、各モデルで Recall@k と MRR（Mean Reciprocal Rank）を測って比べます。ここでの測定は 1.1 の PoC の一部として実施し、記録を残します。

大量文書の初期取り込みでは、Lambda functions を使って batch generate embeddings（埋め込みの一括生成）を行う構成が定石です。S3 に置かれた文書を SQS 経由で Lambda に配り、埋め込み API を呼んでベクトルストアへ書き込みます。埋め込みモデル側のスロットリングに当たるため、SQS のバッチサイズと Lambda の同時実行数で流量を絞り、失敗はデッドレターキューへ退避させます。

重要な制約を一つ。検索時のクエリは、インデックス作成時と同じ埋め込みモデル・同じ次元数で埋め込まなければなりません。埋め込みモデルを変更したら、インデックス全体を作り直します。「埋め込みモデルを新しいものに差し替えたら精度が落ちた」という設問で、原因として「既存ベクトルを再生成していない」を選ばせる形はよく出ます。

1-5-3 vector search solution の配備 [1.5]

vector search solutions（ベクトル検索基盤）の選択肢は三つに整理できます。1.4 のベクトルストア設計と対応しますが、ここでは「semantic search capabilities をどう有効化するか」の運用面を押さえます。

選択肢	有効化の形	向く場面	運用の重さ
Amazon OpenSearch Service（vector search capabilities）	k-NN のインデックスを作り、エンジン（Faiss など）とアルゴリズム（HNSW など）を選ぶ	既に OpenSearch を運用している、hybrid search と高度な絞り込みが要る	自分でクラスタとインデックスを管理する
Amazon Aurora（pgvector extension）	PostgreSQL に pgvector 拡張を入れ、vector 型の列とインデックスを作る	業務データと同じトランザクションで一貫して扱いたい、SQL で結合したい	既存 RDB 運用の延長で済む
Amazon Bedrock Knowledge Bases（managed vector store functionality）	Knowledge Base 作成時にベクトルストアを Bedrock 側に用意させる	早く作りたい、ベクトルストアの運用を持ちたくない	もっとも軽い

Amazon OpenSearch Serverless を使えば、OpenSearch のインデックス機能を保ったままクラスタ運用を外せます。Amazon S3 Vectors は保存コスト

を最優先する選択肢で、検索頻度が低く容量が大きい用途に向きます。ただし前述のとおり hierarchical chunking との併用は推奨されていません。

誤答肢の切り方として、「既存の PostgreSQL の顧客テーブルと結合して、権限のある顧客の文書だけを検索したい」に Knowledge Bases のマネージドストアだけを当てる肢は、結合ができない点で Aurora pgvector に劣ります。逆に「ベクトル検索の運用要員がいない、まず 2 週間で出したい」に自己管理の OpenSearch クラスタを当てる肢は、運用負荷の要件に負けます。

1-5-4 高度な検索構成（hybrid search・reranker）[1.5]

素の semantic search には弱点があります。型番、法令番号、社内の略号のように「文字列としての一致」が重要な語は、意味ベクトルでは近い他の語と混ざります。これを埋めるのが hybrid search（ハイブリッド検索）です。

検索方式	仕組み	強い場面	弱い場面
keyword search（キーワード検索）	語の出現に基づくスコア（BM25 など）	型番・固有名詞・略号の完全一致	言い換え、同義語、質問文での検索
semantic search（意味検索）	埋め込みベクトルの近さ	言い換え、意図の近さ、多言語	表記ゆれのない固有識別子、否定表現
hybrid search（両者の組み合わせ）	keyword と vector のスコアを結合して並べ替える	実務の大半。固有名詞と自然文が混ざるクエリ	設定と検証の手間が増える

Amazon OpenSearch は semantic search と keyword search の両方を持つため、hybrid search を組みやすい選択肢です。Amazon Bedrock Knowledge Bases の検索設定でも、検索タイプとして semantic と hybrid を選べます。

その上に置くのが reranking（再ランク付け）です。Amazon Bedrock reranker models は、検索で取れた候補（例えば上位 50 件）をクエリとの関連度で採点し直し、上位数件だけを FM に渡します。二段構えにする理由は計算量です。全文書に対して精緻な採点をするのは重すぎるので、まずベクトル検索で粗く広く集め、次に reranker で精密に絞ります。

段	目的	件数の目安	使うもの
一次検索	取りこぼさない（再現率を優先）	上位 20〜100 件	hybrid search
再ランク付け	精度を上げる（適合率を優先）	上位 3〜10 件に絞る	Amazon Bedrock reranker models
生成	根拠つきで答える	絞った件数	Converse API、Knowledge Bases の RetrieveAndGenerate

図 1-8　一次検索で広く、reranker で絞る —— 二段構えの検索

metadata filtering（メタデータによる絞り込み）も併用します。1.4 で設計したメタデータフレームワークの部署・機密区分・有効期限を検索条件に入れることで、権限の無い文書を検索の時点で除きます。これは精度対策であると同時にアクセス制御の一部です。

誤答肢の切り方として、「検索結果に関係のない文書が混ざる。取れてはいるが順位が悪い」という症状に、chunk サイズの変更や埋め込みモデルの変更を当てる肢よりも、reranker の追加が直接的です。逆に「そもそも該当文書が候補に上がってこない」という症状に reranker を当てる肢は、候補に無いものは並べ替えられないので切れます。症状が再現率の問題か適合率の問題かを見分けるのが分かれ目です。

1-5-5 query handling（クエリ処理）の高度化 [1.5]

利用者の入力はそのまま検索に適した形とは限りません。曖昧、複数の問いが混ざる、指示語で前の発言を指す、といったずれを整えるのが query handling systems（クエリ処理）です。

手法	何をするか	実装の型
query expansion（クエリ拡張）	同義語・略称の展開、想定される言い換えを追加して検索の網を広げる	Amazon Bedrock の FM に拡張語を生成させる
query decomposition（クエリ分解）	「A と B の違いは」を A の検索と B の検索に分ける	Lambda functions で分解し、それぞれ検索して結果を統合する
query transformation（クエリ変換）	会話履歴を踏まえて指示語を実体に置き換える、検索用の文に書き換える	Step Functions で変換・検索・統合を段として並べる

三つの使い分けを場面で押さえます。社内用語が多く、利用者が正式名称を知らない問い合わせでは query expansion が効きます。比較・複合条件の質問では query decomposition が効きます。マルチターンのチャットで「それの手続きは？」のような入力が来る場面では query transformation が必須です。

コストと遅延の対価があることも押さえてください。いずれも FM 呼び出しを 1 回以上増やします。すべてのクエリに三つとも適用するのは過剰です。実務では、まず素の検索を行い、結果のスコアが閾値を下回ったときだけ拡張や分解に進む、という条件分岐にします。この分岐は Step Functions の Choice で表現できます。

1-5-6 FM から検索を呼ぶための一貫したアクセス手段 [1.5]

最後に、作った検索基盤を FM 側からどう呼ぶかです。consistent access mechanisms（一貫したアクセス手段）を用意することで、モデルやアプリを差し替えても検索の呼び出し方が変わらないようにします。

方式	形	向く場面
function calling interfaces（関数呼び出しインターフェイス）	検索を「ツール」として FM に宣言し、FM が必要と判断したときだけ呼ぶ	FM 自身に検索の要否を判断させたい
Model Context Protocol（MCP）clients	MCP サーバーとして検索を公開し、MCP クライアントから呼ぶ	複数のエージェント・複数のフレームワークから同じ検索を使い回す
standardized API patterns（標準化された API パターン）	API Gateway に固定の検索エンドポイントを置き、すべてのアプリがそこだけを叩く	社内の複数アプリで検索を共通化し、統制と監査を一点に集める

function calling は Amazon Bedrock の Converse API の tool use として実装します。検索関数の名前、説明文、パラメータのスキーマを宣言すると、FM は必要に応じて「この関数をこの引数で呼びたい」という応答を返します。アプリケーションが実際に検索を実行し、結果を会話に差し戻すと、FM がそれを根拠に回答を作ります。関数の説明文が曖昧だと FM が呼ばない、あるいは誤った引数で呼ぶため、説明文の記述は精度に直結します。

MCP（Model Context Protocol）は、ツールの公開方法をプロトコルとして標準化したものです。検索を MCP サーバーとして一度公開すれば、MCP に対応したどのエージェント基盤からも同じ形で呼べます。Amazon Bedrock AgentCore Gateway は、既存の API や Lambda functions を MCP 互換のツールに変換して公開する仕組みで、この型を最短で実現します。MCP の実装形態は 2.1 で詳しく扱います。

standardized API patterns は統制の観点で選ばれます。すべての検索を API Gateway 経由に集約すれば、認証、レート制限、ログ、課金の按分を一か所で行えます。「部署ごとにベクトルストアへ直接アクセスしていて、誰が何を検索したか分からない」という課題に対する答えは、この集約です。

誤答肢の切り方として、「複数のエージェントフレームワークから同じ検索を使いたい」に、フレームワーク固有の SDK での実装を当てる肢は、標準化の要件に反します。逆に「単一のチャットアプリから 1 つの検索を呼ぶだけ」に MCP サーバーの構築を当てる肢は、過剰設計としてコスト・工期の要件に負けます。

1-6 FM インタラクションのためのプロンプトエンジニアリング戦略とガバナンスを実施する [1.6]

1.5 までで「何を FM に渡すか」が決まりました。1.6 は「どう指示するか」と「その指示をどう統制するか」です。Professional の粒度では、良いプロンプトを書く技術だけでなく、組織として prompt management and governance（プロンプトの管理と統制）を成立させる設計が問われます。

1-6-1 model instruction framework（モデル指示の枠組み）[1.6]

FM の振る舞いを制御する手段は三層あります。層ごとに「何を止められるか」が違うため、対応関係で覚えます。

層	手段	制御できること	限界
指示	system prompt での role definitions（役割定義）、template configurations（テンプレート設定）による出力形式の指定	口調、役割、出力フォーマット、扱う話題の範囲	指示は上書きされうる。悪意ある入力には弱い
管理	Amazon Bedrock Prompt Management	指示の版管理、変数化、承認、再利用	内容そのものの妥当性は保証しない
強制	Amazon Bedrock Guardrails による responsible AI guidelines（責任ある AI の指針）の強制	禁止話題、有害表現、個人情報、根拠の無い出力の遮断	業務ロジックの正しさは見ない

FM の振る舞いを制御する三層 —— 指示・管理・強制

層	手段	制御できること／限界
指示	system prompt での role definitions、template configurations による出力形式の指定	口調、役割、出力フォーマット、扱う話題の範囲 限界＝指示は上書きされうる。悪意ある入力には弱い
管理	Amazon Bedrock Prompt Management	指示の版管理、変数化、承認、再利用 限界＝内容そのものの妥当性は保証しない
強制	Amazon Bedrock Guardrails による responsible AI guidelines の強制	禁止話題、有害表現、個人情報、根拠の無い出力の遮断 限界＝業務ロジックの正しさは見ない

Guardrails は指示とは独立に働き、入力と出力の双方を検査して遮断する。
「プロンプトに書いたのに守られない」という設問の答えは、ほぼ Guardrails。

図 1-9　FM の振る舞いを制御する三層 —— 指示・管理・強制

role definitions は「あなたは当社の人事規程に基づいてのみ回答する社内アシスタントです。規程に記載が無い場合は『規程に記載がありません』と答えてください」のように、役割と禁止事項を具体的に書きます。曖昧な役割定義は、そのまま曖昧な出力になります。

template configurations は出力形式の固定です。後続の処理がプログラムなら、JSON スキーマを提示して「このスキーマに厳密に従い、前後に説明文を付けない」と指示します。形式の揺れは、後段の Lambda のパース失敗として現れます。

Amazon Bedrock Guardrails は指示とは独立に働きます。プロンプトで「暴力的な内容は出さないでください」と書くのは指示であり、迂回されえます。Guardrails は入力と出力の双方を検査して遮断するため、指示が破られても最後の砦になります。「プロンプトに書いたのに守られない」という設問の答えは、ほぼ Guardrails です。

1-6-2 対話を成立させる仕組み（文脈の保持と意図の把握）[1.6]

一問一答で終わらない interactive AI systems（対話型システム）では、文脈の保持と意図の把握が要ります。

要素	実装	設計の要点
conversation history storage（会話履歴の保存）	Amazon DynamoDB にセッション ID をパーティションキー、発話の連番をソートキーとして保存	保持期間を TTL で自動削除。全履歴を毎回送るとトークン費用が膨らむため、直近 N ターン＋要約に圧縮する
intent recognition（意図認識）	Amazon Comprehend で分類・エンティティ抽出を行い、問い合わせの種類を判定する	FM に毎回判定させるより速く安い。定型の振り分けは Comprehend に寄せる
clarification workflows（聞き返しのワークフロー）	AWS Step Functions で「情報が足りない → 聞き返す → 揃ったら実行」の分岐を組む	何が揃えば実行してよいかを状態として持つ。FM の気分で決めさせない

会話履歴には二つの相反する要求があります。文脈を保つには長く持ちたい、コストと遅延を抑えるには短くしたい。実務では、直近数ターンは原文のまま、それより前は FM に要約させた 1 段落として保持する構成を取ります。要約は DynamoDB の同じ項目に書き戻します。

Amazon Comprehend を使う理由も押さえます。「返品したい」「営業時間を知りたい」といった意図の分類は、FM に投げても解けますが、Comprehend の分類のほうが遅延もコストも小さく、結果も安定します。FM は自由記述の生成に使い、決まった分類は専用サービスに任せる、という切り分けが Professional の判断です。

1-6-3 prompt management と governance [1.6]

組織で FM を使うと、同じようなプロンプトが各チームに散らばり、誰がどの版を使っているか分からなくなります。これを防ぐのが prompt management and governance（プロンプトの管理と統制）です。

Amazon Bedrock Prompt Management は次の単位で構成されます。

用語	意味
prompt（プロンプト）	モデルに与える入力そのもの
variable（変数）	プロンプト中のプレースホルダー。テスト時や実行時に値を差し込む
prompt variant（バリアント）	メッセージ・モデル・推論設定を変えた代替構成。比較して良いものを残す
prompt builder（プロンプトビルダー）	コンソール上でプロンプトとバリアントを作成・編集・テストする画面
version（バージョン）	保存した時点の内容を固定した版。アプリケーションからはこの版を指定して呼ぶ

parameterized templates（パラメータ化テンプレート）は variable によって実現します。「{{顧客名}} 様の {{商品名}} に関する問い合わせに回答してください」のように書き、実行時に値を差し込みます。テンプレートを共有しつつ用途ごとに使い分けられます。

approval workflows（承認フロー）は、Prompt Management のバージョンを軸に組みます。開発者が新しい版を作り、レビュー担当が評価結果を確認し、承認されたバージョン番号だけをアプリケーションの設定値（AWS AppConfig など）に反映します。アプリのコードを変えずに使用する版を切り替えられるため、問題があれば前の版に戻すだけで済みます。

統制の残りの三つは記録です。

目的	サービス	記録される内容
template repositories（テンプレートの保管）	Amazon S3	プロンプト定義ファイル、評価データセット、変更履歴（バージョニング有効化）
使用の追跡	AWS CloudTrail	どの IAM プリンシパルがいつどの API を呼んだか（管理操作の監査証跡）
アクセスの記録	Amazon CloudWatch Logs	アプリケーションが出力した呼び出しログ、入出力のメタデータ

CloudTrail と CloudWatch Logs の役割の違いは頻出です。CloudTrail は「誰が API を呼んだか」の監査証跡、CloudWatch Logs は「アプリが何を記録したか」のログです。「プロンプトテンプレートを誰が変更したかを追跡したい」は CloudTrail、「実際の入出力の内容を分析したい」は Bedrock の model invocation logging と CloudWatch Logs です。

1-6-4 プロンプトの品質保証（テストと回帰検知） [1.6]

プロンプトはコードと同じく変更すると壊れます。quality assurance systems（品質保証の仕組み）を、コードのテストと同じ型で組みます。

段階	内容	実装
期待出力の検証	出力が指定形式か、必須項目が入っているか、禁止語が無いかを機械判定する	Lambda functions で JSON スキーマ検証・正規表現・数値範囲チェック

段階	内容	実装
edge cases（境界事例）のテスト	空入力、極端に長い入力、多言語、意図的な逸脱の誘導などを一括で流す	Step Functions の Map で試験ケースを並列実行し、結果を集計する
prompt regression（回帰）の検知	版を上げたあとに、以前は通っていたケースが落ちていないかを継続的に見る	合格率をカスタムメトリクスとして CloudWatch に送り、閾値を下回ったらアラーム

回帰の検知を CloudWatch で行う理由は、モデル側が更新される可能性があるためです。自分のプロンプトを変えていなくても、参照するモデルのバージョンが変われば出力は変わりえます。定期実行の評価ジョブを回し、合格率の推移を見るのが Professional の運用です。

LLM-as-a-Judge（別の FM に採点させる方式）は、正解が一意に決まらない出力の評価に使います。機械判定で測れるのは形式の正しさまでで、要約の妥当性や回答の丁寧さは判定モデルに採点基準を渡して点数化します。Amazon Bedrock のモデル評価では、人手評価と自動評価に加えてこの方式を利用できます。詳細は第4章で扱います。

1-6-5 出力品質を上げる高度なプロンプト技法 [1.6]

基本的なプロンプトを超えて FM performance（性能）を高める技法は四つです。

技法	内容	効く場面
structured input components（構造化された入力）	指示・文脈・入力データ・出力形式を見出しや XML 風タグで区切る	長いプロンプト。どこが指示でどこがデータか FM に取り違えさせない

技法	内容	効く場面
output format specifications（出力形式の指定）	JSON スキーマ・列名・許容値の列挙まで明示する	後続処理がプログラムのとき
chain-of-thought instruction patterns（思考の連鎖の指示）	結論の前に手順を順に書かせる	計算、条件分岐のある判断、多段の推論
feedback loops（フィードバックループ）	出力を評価して、不合格ならプロンプトを補って再実行する	品質のばらつきが許されない業務

structured input components には実務上の効用がもう一つあります。入力データを明確に区切ると、データ中に紛れ込んだ指示文（prompt injection）を「これはデータであって指示ではない」と扱いやすくなります。区切りだけで防ぎ切れるわけではないので、Guardrails と併用します。

chain-of-thought は遅延とトークン費用を増やします。単純な分類にまで適用するのは無駄です。「複数の条件を突き合わせて可否を判断する」「金額を計算して根拠を示す」といった多段の判断にだけ当てます。

feedback loops は Step Functions で表現します。生成 → 検証 → 不合格なら不足点を添えて再生成、というループに、最大試行回数と全体のタイムアウトを必ず設けます。上限の無いループは、コストとレイテンシの事故になります。

誤答肢の切り方として、「出力の JSON が時々壊れて後続が失敗する」に chain-of-thought を当てる肢は、原因（形式の不安定さ）に対する手当てになっていません。出力形式の明示と、Lambda での検証・再実行が正解です。逆に「計算結果がしばしば誤る」に出力形式の指定を当てる肢も、形式と正確さを取り違えています。

1-6-6 複雑なプロンプト連鎖（Amazon Bedrock Prompt Flows）[1.6]

単一のプロンプトで解けない仕事は、複数のプロンプトを繋いで解きます。Amazon Bedrock Flows（Prompt Flows）は、この連鎖をノードとして視覚的に組む機能です。Prompt Management で保存したプロンプトを prompt node として組み込めます。

構成要素	役割
sequential prompt chains（逐次のプロンプト連鎖）	抽出 → 分類 → 要約のように段を並べ、前段の出力を次段の入力にする
conditional branching（条件分岐）	モデルの応答内容に応じて経路を分ける。例：機密判定が真なら人手レビューへ
reusable prompt components（再利用可能な部品）	Prompt Management のプロンプトを複数のフローから参照する
pre-processing / post-processing steps（前処理・後処理）	Lambda ノードで入力の正規化や出力の整形・検証を挟む

Prompt Flows と AWS Step Functions の使い分けは頻出です。対応関係で押さえます。

要件	選ぶもの	理由
プロンプトの連鎖と分岐が中心、ノーコードで業務側も触りたい	Amazon Bedrock Prompt Flows	生成 AI の流れに特化し、Prompt Management と直結する
複数の AWS サービスをまたぐ長時間の処理、リトライ・人手承認・並列処理が要る	AWS Step Functions	汎用のワークフロー基盤で、待機・再試行・Map などの制御が豊富

要件	選ぶもの	理由
単純な 1 段の呼び出し	Lambda から Converse API を直接呼ぶ	フローの管理単位を増やす必要が無い

誤答肢の切り方として、「業務部門が自分でプロンプトの流れを組み替えたい」に Step Functions のステートマシン定義を当てる肢は、業務部門が触れる形かという点で Prompt Flows に劣ります。逆に「数日かかる文書処理を、外部システムの応答を待ちながら進めたい」に Prompt Flows を当てる肢は、長時間の待機と外部連携という点で Step Functions に負けます。

1-7 場面別に判断を固める（ドメイン1の総合演習）[1.1/1.2/1.3/1.4/1.5/1.6]

ドメイン1の部品を、設問に近い場面設定で組み合わせます。要件の言葉から設計を引き出す練習です。

1-7-1 場面：社内規程に基づく人事問い合わせ対応 [1.1/1.4/1.5/1.6]

人事部門が、就業規則・給与規程・福利厚生の資料に基づいて社員の問い合わせに答える仕組みを作ります。要件は次のとおりです。資料は Word と PDF で章立てがある。規程に無いことは答えてはならない。回答には根拠の条文を示す。改定された規程は翌営業日までに反映する。人事評価に関する資料は評価者だけが検索できる。

要件	設計	理由
章立てを保った検索	hierarchical chunking（child で検索し parent で文脈を渡す）	条文単位で当てつつ、条文の属する章の文脈を失わない

要件	設計	理由
規程に無いことは答えない	system prompt の role definitions と Amazon Bedrock Guardrails の contextual grounding check	指示と強制の二層。指示だけでは保証にならない
根拠の条文を示す	Knowledge Bases の引用機能。メタデータに規程名・条番号・改定日を持たせる	1.4 のメタデータフレームワークがそのまま効く
翌営業日までの反映	S3 の更新イベントを EventBridge で受け、取り込みジョブを起動する増分更新	日次バッチでも要件は満たすが、増分のほうが速く安い
評価者だけの検索	メタデータに機密区分を持たせ、検索時のフィルタで絞る。IAM と二段で効かせる	検索の時点で除かないと、根拠として出てしまう

誤答肢の切り方として、「規程に無いことを答えてしまう」への対処に「fine-tuning で規程を学習させる」を当てる肢は方向が違います。fine-tuning は口調や形式の学習に向き、事実の最新性と根拠提示には RAG が適します。改定のたびに再学習するのは、コストと反映速度の両面で不利です。

1-7-2 場面：コールセンターの通話ログ分析基盤 [1.1/1.3/1.5]

コールセンターの通話録音から、問い合わせ傾向と対応品質を分析する基盤を作ります。要件は次のとおりです。1日あたり数千件の音声。話者を分けたい。個人情報は分析前に落とす。分析は翌朝までに完了すればよい。過去2年分を検索できるようにする。

要件	設計
音声のテキスト化と話者分離	Amazon Transcribe。speaker diarization を有効化し、カスタム語彙で製品名の精度を上げる
個人情報の除去	Amazon Comprehend の PII 検出、または Guardrails の機密情報フィルタでマスクしてから保存する
数千件を翌朝までに	Amazon Bedrock の batch inference。急がない大量処理はバッチが最も安い
傾向の抽出	定型の分類は Amazon Comprehend、自由記述の要約と論点抽出は FM に分担させる
過去2年分の検索	ベクトルストアに蓄積。検索頻度が低く容量が大きいなら Amazon S3 Vectors が候補

時間要件が「翌朝までに」であることが、この場面の分かれ目です。リアルタイム推論やストリーミングを当てる肢は、要件を満たしはしますがコスト面で劣ります。逆に「オペレーターの通話中に支援したい」という要件なら、バッチは論外でストリーミングが正解になります。同じ素材でも、時間要件で設計が反転します。

1-7-3 対比で覚える：ドメイン1で取り違えやすい組み合わせ [1.1/1.2/1.3/1.4/1.5/1.6]

比較	決め手
RAG ↔ fine-tuning	事実の最新性と根拠提示か、口調・形式・専門的な振る舞いの習得か
fine-tuning ↔ distillation（蒸留）	精度を上げるか、大きなモデルの挙動を小さなモデルに写して安くするか

比較	決め手
default chunking ↔ hierarchical chunking	まず動かすか、章立ての文脈を保つか
fixed-size chunking ↔ semantic chunking	形式が揃っているか、見出しの無い文章で話題の切れ目を見つけたいか
semantic search ↔ hybrid search	言い換えに強くしたいか、型番・固有識別子の完全一致も要るか
reranker の追加 ↔ chunk 設計の見直し	候補には入るが順位が悪いか、そもそも候補に入らないか
query expansion ↔ query decomposition	用語のゆれを吸収したいか、複合的な問いを分けたいか
Prompt Management ↔ Guardrails	指示の版を管理するか、内容を強制的に遮断するか
CloudTrail ↔ CloudWatch Logs	誰が API を呼んだかの監査証跡か、アプリと入出力の記録か
Amazon Bedrock Prompt Flows ↔ AWS Step Functions	プロンプト連鎖中心でノーコードか、サービス横断の汎用ワークフローか
OpenSearch（vector search） ↔ Aurora pgvector	検索機能の豊富さと hybrid search か、業務データとの SQL 結合か
Knowledge Bases のマネージドストア ↔ 自己管理のベクトルストア	早さと運用の軽さか、細かい制御と既存資産の活用か
Amazon Textract ↔ Amazon Bedrock のマルチモーダルモデル	帳票の構造と座標を取るか、画像の内容を理解して記述するか
Amazon Comprehend ↔ FM	定型の分類・抽出を安く速くか、自由記述の生成か

ドメイン1を貫く原則を三つ挙げます。第一に、要件の数値化が先で、サービス選定は後です。精度・遅延・コスト・データ所在の四軸を決めないまま構成を選ぶことはできません。第二に、検索で取れていない情報は生成できません。品質問題は、まず検索側を疑います。第三に、プロンプトによる指示と、Guardrails・IAM による強制を区別します。指示は破られうるという前提で設計するのが Professional の粒度です。

サンプルはここまでです

続き(第2章〜巻末)は、AWS AIP-C01 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。

AWS AIP-C01 学習用テキスト(無料サンプル)

版

2026年7月版(AWS 公式 Exam Guide（AIP-C01・2026年9月7日取得）準拠)

発行

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

お問い合わせ

info@shikaku-doujou.com