2026年3月版

SOFTWARE ENGINEERING KNOWLEDGE BASE

ソフトウェアエンジニアリング知識体系

2025-2026

実践的技術選定ハンドブック

開発環境、アプリケーション設計、インフラ、品質、AI
駆動開発までを横断し、各技術の位置づけと選定の判断軸を整理した書籍版資料。

発行

haya株式会社

SOFTWARE ENGINEERING
KNOWLEDGE BASE
2025-2026

目次

第1章 開発環境の整え方

> この章を読む前に: プログラミングの基本概念（変数、関数、ファイル操作）を理解していれば十分。特定の言語の経験は不要。

ソフトウェア開発は、コードを書く前の「環境構築」で最初の壁にぶつかる。

何時間もかけてツールをインストールし、エラーを検索し、

動くようになった頃には疲弊している——これは多くのエンジニアが通る道だ。

この章では、なぜ特定のツールが必要とされるのか、

どのツールを選ぶべきかの判断基準を示す。

1.1 OS選択

> このセクションで答える問い: 開発用のOSは何を選ぶべきか？

なぜ必要か

プログラミング自体はどのOSでもできる。しかし、本番環境のサーバーはほぼ Linux で動いている。

開発環境と本番環境のOSが異なると、「自分のPCでは動くのに本番では動かない」という問題が起きやすい。

また、開発ツールの多くは Unix 系OS（macOS / Linux）を前提に作られているため、

Windows ではひと手間かかるケースがある。

全体像

OS	開発での位置づけ	本番環境との相性	エコシステム
macOS	Web/モバイル開発の標準。iOS開発には必須	Unix系なので高い	Homebrewで大半のツールが入る
Linux（Ubuntu等）	サーバーと同じOSで開発できる最大の利点	最も高い	apt/dnf でパッケージ管理
Windows + WSL2	WSL2 により Linux環境を内包可能に	WSL2 内は高い	2026年時点で実用的な選択肢

選定の判断軸

```
graph TD
A[開発を始めたい] --> B{iOS アプリを作る?}
B -->|はい| C[macOS 一択]
```

```
B -->|いいえ| D{既にPCを持っている?}
D -->|Windows| E[WSL2 を導入]
D -->|Mac| F[そのまま macOS]
D -->|持っていない| G{予算は?}
G -->|20万円以上| H[MacBook がおすすめ]
G -->|10万円以下| I[Linux ノートPC or Windows + WSL2]
```

実務での注意点

> **WARNING**
>
> Windows をWSLなしで使う落とし穴: パス区切り文字（\ vs /）、改行コード（CRLF vs LF）、ファイル名の大文字小文字の扱いなど、 些細な差異が積み重なりトラブルの原因になる。 Windows で開発する場合、WSL2 の導入は事実上必須と考えてよい。

> **TIP**
>
> macOS ユーザーへ: Homebrew のインストール（/bin/bash -c "$(curl -fsSL ...)" ）は 開発環境構築の最初の一歩。ここから他のすべてのツールを入れていく。

まとめ

迷ったら macOS が最もスムーズ。Windows ユーザーは WSL2 を必ず入れる。

いずれの場合も、最終的にはターミナル上で Linux コマンドを使って開発する、

という点は共通している。OSの違いは「Linux にたどり着くまでの道のり」の違いでしかない。

1.2 ターミナルとシェル

このセクションで答える問い: ターミナルとは何か？なぜGUIではなくコマンド操作が必要なのか？

なぜ必要か

日常のPC操作は、アイコンをクリックしてファイルを開くGUI（グラフィカルユーザーインターフェース）で事足りる。

しかしソフトウェア開発では、数百のファイルを一括操作したり、

サーバー上でプログラムを起動したり、自動化スクリプトを実行する場面が頻繁にある。

これらの作業は、テキストでコマンドを打つ ターミナル（端末）の方が圧倒的に効率がよい。

料理に例えるなら、GUIは「完成された料理キット」、

ターミナルは「包丁とまな板」だ。

キットは手軽だが、応用が利かない。

包丁の使い方を覚えれば、どんな料理でも作れるようになる。

全体像

ターミナル（Terminal）はコマンドを入力する画面のこと。

シェル（Shell）はターミナルの裏で動いている、コマンドを解釈するプログラムのこと。

シェル	特徴	おすすめ度（2026年時点）
zsh	macOS のデフォルト。bash互換で拡張性が高い	☒ 最もおすすめ
bash	Linux のデフォルト。情報量が最も多い	☒ 安定した選択
fish	補完が賢くデフォルトで使いやすい。bash 非互換	好みで選択

選定の判断軸

初心者は zsh を選んでおけば間違いない。macOS では最初から入っている。

Linux（Ubuntu）では bash がデフォルトだが、zsh に切り替えるのも簡単だ。

fish は「設定なしで便利」という利点があるが、

bash スクリプトとの互換性がないため、ネット上のコマンド例をコピペすると動かないことがある。

この落とし穴を許容できるなら快適な選択肢だ。

実務での注意点

> IMPORTANT
>
> シェルの設定ファイル（~/.zshrc や ~/.bashrc）は 開発環境の「設計図」にあたる。何を書いたかわからなくなると、環境が壊れた時に復旧できない。 変更するときは必ずコメントを残す習慣 をつけよう。

まとめ

ターミナル操作は開発者の基本スキル。シェルは zsh がバランスが良い。

最初は怖く感じるが、cd、ls、mkdir、cat の4コマンドから始めれば十分だ。

使い慣れるにつれ、GUIより速く正確に作業できるようになる。

1.3 エディタ・IDE

このセクションで答える問い:
コードを書くツールは何を使うべきか？2026年時点で、IDE選定の基準は何が変わったのか？

なぜ必要か

コードはただのテキストファイルなので、極論すればメモ帳でも書ける。

しかし実際の開発では、数千行のコードの中からエラー箇所を見つけたり、

関数の定義元にジャンプしたり、変数名を一括で変更したりする作業が日常的に発生する。

これらを手作業でやるのは、地図なしで知らない街を歩くようなものだ。

2026年時点の変化は、AI支援が「付加機能」ではなく主要な選定軸になったことにある。

ただし、AI だけで選ぶのは危険だ。言語サポート、リファクタリング能力、

チームでの共有しやすさ、リモート開発との相性も同じくらい重要である。

全体像

ツール	位置づけ	強み	注意点
VS Code + AI拡張	汎用エディタの基準点	拡張機能が豊富。GitHub Copilot などを後付けしやすい	設定の自由度が高く、チームごとの差が出やすい
Cursor	AI統合型コードエディタ	コードベース理解、自然言語編集、VS Code 由来の操作感	AI 機能を強く使う前提なので、ルール整備が必要
Windsurf	エージェント寄りのエディタ	Cascade による計画・編集・ターミナル連携	自動編集の範囲が広く、権限設計とレビュー手順が重要
JetBrains IDE	言語特化の本格 IDE	静的解析、補完、リファクタリングが強い。AI Assistant / Junie も統合可能	動作は比較的重く、製品ごとの学習コストもある
Claude Code	ターミナル型コーディングエージェント	計画、複数ファイル編集、コマンド実行、CI 連携	IDE の代替ではなく、既存エディタと併用する道具

AI統合による選定基準の変化（2026年）

2020年代前半 と 2026年 で、見るべきポイントは次のように変わった。

観点	旧基準（2020-2024）	新基準（2026）
第一選定軸	言語サポートと拡張機能	AI がコードベースをどこまで理解できるか
第二選定軸	動作速度、UI の好み	複数ファイル編集や計画実行ができるか
第三選定軸	プラグインの多さ	チームのルールを AI に共有しやすいか
補助的観点	AI は任意機能	AI は重要。ただしレビュー可能性が前提

選定の判断軸

まず迷ったとき

最初の一手は VS Code + AI拡張 か Cursor の二択でよい。

無料または低コストで情報量の多さを優先するなら VS Code、

最初から AI と対話しながら書く体験を重視するなら Cursor が向いている。

言語特化で深く開発する人

JetBrains IDE（IntelliJ IDEA、PyCharm、WebStorm など）が強い。

Java、Kotlin、Python のように IDE の解析精度が生産性へ直結する領域では、

汎用エディタよりも価値が出やすい。

エージェント的に任せたい人

Windsurf や Claude Code は、「質問に答える」だけではなく

計画を立て、編集し、場合によってはコマンド実行まで進める。

複数ファイルにまたがる変更や調査タスクで特に効果が高い。

チーム開発を重視する人

どのエディタを選ぶか以上に、AI に守らせるルールを共有できるか が重要だ。

.editorconfig、フォーマッタ、リンタ、テストに加え、

リポジトリ固有の指示ファイルを置いておくと、AI の出力が安定しやすい。

実務での注意点

> WARNING
>
> AI 出力は必ずレビューし、テストする GitHub Copilot の責任ある利用ガイドでも、
> 生成コードは人間がレビューし、十分にテストすべきだと明記されている。AI
> が速く書けることと、正しいことは別問題である。

> IMPORTANT
>
> クラウド送信の前提を確認する AI エディタは、コード断片やプロンプトをクラウドに送る場合がある。
> 業務コード、個人情報、秘密情報を扱う場合は、
> 利用規約、保存ポリシー、プライバシーモードの有無を先に確認する。

> TIP
>
> エディタの設定はチームで統一するのがおすすめ。
> .editorconfig、フォーマッタ設定、リンタ設定をリポジトリに置けば、IDE が違っても最低限の一貫性を保ちやすい。

> NOTE
> 1つのツールですべてを解決しなくてよい 普段は VS Code や Cursor を使い、重い変更だけ Claude Code や Copilot coding agent に任せる、 という使い分けはごく自然である。

まとめ

2026年の IDE 選定では、AI 支援の有無は無視できない。

ただし最終的に見るべきは、AI が強いか ではなく

その AI をチームのルールと品質管理の中で安全に使えるか である。

最初の選択としては VS Code + AI拡張 または Cursor が扱いやすい。

1.3.1 クラウド開発環境（リモート開発 / ブラウザベース IDE）

このセクションで答える問い: ローカルマシンに IDE をインストールしないという選択肢はあるのか？

なぜ必要か

ある。しかも近年は「例外的な方法」ではなく、普通の選択肢になった。

特に次のような状況では、クラウド開発環境の価値が高い。

- 新メンバーの環境構築を短時間で揃えたい

- 手元の PC が低スペックで、大きなビルドや依存関係の展開が重い

- GitHub 上のリポジトリを、全員同じ Linux 環境で扱いたい

- 一時的な検証環境やレビュー環境をすぐ作りたい

全体像

ツール	位置づけ	強み	向いている場面
GitHub Codespaces	GitHub ホスト型の開発環境	devcontainer.json を使った再現性、ブラウザ / VS Code / CLI 接続	GitHub 中心のチーム
DevPod	インフラ非依存のワークスペース管理	devcontainer.json を使い、Docker / Kubernetes / SSH / 各種クラウドに展開できる	ベンダーロックインを避けたいチーム
Replit	ブラウザ中心の開発・試作環境	Agent を使った高速な試作、公開までの導線が短い	学習、デモ、プロトタイプ

選定の判断軸

GitHub Codespaces（推奨：GitHub 主体のチーム）

GitHub Codespaces は GitHub がホストする Linux 開発環境で、

ブラウザ、VS Code、GitHub CLI から接続できる。

devcontainer.json をリポジトリに置けば、チーム全員が同じ開発環境を再利用しやすい。

DevPod（推奨：環境定義を使い回したいチーム）

DevPod はクライアント中心のツールで、devcontainer.json を起点に

ローカル、クラウド VM、Kubernetes、SSH 先などへ開発環境を展開できる。

「Codespaces 的な体験はほしいが、実行基盤は自分で選びたい」という場面に向く。

Replit（推奨：学習・試作）

Replit はブラウザから始めやすく、Agent を使ったアプリ試作が速い。

ただし、複雑な社内システムや既存モノレポ開発では、

Codespaces や DevPod の方が制御しやすいことが多い。

クラウド開発環境 vs ローカル開発環境

観点	ローカル	クラウド
初期セットアップ	個人差が大きい	設定をリポジトリに寄せやすい
チーム間の再現性	崩れやすい	高い
マシンスペック依存	強い	弱い
オフライン耐性	高い	低い
コスト管理	端末費用が中心	実行時間・ストレージ課金に注意

実務での注意点

> WARNING
>
> ネットワーク遅延とコストを軽視しない クラウド開発環境は便利だが、遅延と課金は必ず発生する。
> 常時接続が前提のため、回線品質が悪い環境では体験が大きく落ちる。

> IMPORTANT
>
> 環境定義はリポジトリに置く クラウド開発環境の価値は、「誰が開いても同じ環境になる」ことにある。
> devcontainer.json や関連設定をリポジトリにコミットして、個人設定ではなくプロジェクト設定として管理しよう。

> TIP
> ローカルとクラウドを対立させる必要はない。日常開発はローカル、オンボーディングや一時的な検証はクラウド、という使い分けが現実的である。

まとめ

クラウド開発環境は、再現性とオンボーディング速度 を大きく改善する。

GitHub 中心なら Codespaces、実行基盤を選びたいなら DevPod、

学習や試作なら Replit がわかりやすい。

ただし、遅延・課金・秘密情報の扱いは必ず設計に含めること。

1.4 バージョン管理（Git）

> このセクションで答える問い: なぜファイルを「名前を変えて保存」するだけではダメなのか？

なぜ必要か

レポートを書くとき、report_v1.docx、report_v2_final.docx、

report_v2_final_本当にfinal.docx と増殖した経験はないだろうか。

1人でもこの状態なのに、チームで同じファイルを同時に編集したらどうなるか。

誰がいつ何を変えたかわからなくなり、上書き事故が起き、

昨日まで動いていたコードが突然壊れる。

Git（ギット）はこの問題を解決する バージョン管理システム だ。

すべての変更履歴を記録し、いつでも過去の状態に戻せる。

複数人が同時に作業しても、変更を安全に統合（マージ）できる。

2026年現在、Git は事実上の業界標準であり、Git を使わない開発現場はほぼ存在しない。

全体像

Git の基本概念を4つだけ押さえればよい:

概念	身近な例え	説明
リポジトリ（Repository）	プロジェクトのフォルダ	コードと全変更履歴を格納する場所
コミット（Commit）	ゲームのセーブポイント	ある時点の状態を記録する操作
ブランチ（Branch）	パラレルワールド	本線に影響を与えず新機能を試せる分岐

概念	身近な例え	説明
マージ（Merge）	パラレルワールドの合流	ブランチの変更を本線に取り込む操作

GitHub（ギットハブ）は Git のリポジトリをインターネット上にホスティングするサービス。

Git そのものとは別物だが、チーム開発ではほぼセットで使われる。

競合サービスに GitLab、Bitbucket がある。

選定の判断軸

Git 自体に選択肢はない（業界標準が Git 一択）。

選ぶのは ホスティングサービス の方だ:

サービス	特徴	向いている場面
GitHub	最大のユーザー数。OSS の中心地。GitHub Actions（CI/CD）が強力	ほとんどの場面でおすすめ
GitLab	CI/CD が組み込み。セルフホスト可能	社内に閉じた開発。CI/CDを一元管理したい
Bitbucket	Atlassian製品（Jira等）との連携	Jira をすでに使っているチーム

実務での注意点

> WARNING
>
> パスワードやAPIキーを絶対にコミットしない。一度 Git に記録された情報は、コミットを消しても履歴に残り続ける。誤ってコミットした場合、履歴の書き換えという面倒な作業が必要になる。.gitignore ファイルで機密情報を含むファイルを除外する習慣をつけよう。

> TIP
>
> コミットメッセージは「何をしたか」ではなく「なぜしたか」を書く。Fix bug ではなく ユーザーがログインできない問題を修正（セッションの有効期限チェックが抜けていた） のように書くと、後から読んだときに文脈がわかる。

まとめ

Git はソフトウェア開発の「読み書きそろばん」に相当する必須スキル。

リポジトリ、コミット、ブランチ、マージの4概念を理解すれば基本は十分。

ホスティングは GitHub が最も無難な選択肢だ。

1.5 パッケージマネージャとランタイム管理

このセクションで答える問い: なぜ公式サイトからインストーラーをダウンロードするだけではダメなのか？

なぜ必要か

開発では多くのツールやライブラリを使う。

Node.js、Python、データベース、各種コマンドラインツール……。

これらを1つずつ公式サイトからダウンロードしてインストールすると、

更新の管理が大変になり、バージョンの不整合で問題が起きる。

さらに厄介なのが、プロジェクトAでは Node.js 18 を、プロジェクトBでは Node.js 22 を

使いたいという状況だ。PCに1つのバージョンしか入っていないと切り替えが面倒になる。

パッケージマネージャ はツールのインストール・更新・削除を自動化する仕組み、

ランタイムバージョン管理ツール は複数バージョンの切り替えを可能にする仕組みだ。

全体像

パッケージマネージャ（OSレベル）:

ツール	対応OS	使い方の例
Homebrew	macOS / Linux	brew install node
apt	Ubuntu / Debian	sudo apt install nodejs
winget	Windows	winget install OpenJS.NodeJS

ランタイムバージョン管理:

ツール	特徴
mise（旧rtx）	多言語対応（Node, Python, Go 等）。.mise.toml も使える
asdf	多言語対応。プラグイン方式で広く使われている
nvm	Node.js 専用。シンプル
pyenv	Python 専用

選定の判断軸

複数言語を扱うなら mise か asdf のような多言語対応ツールが扱いやすい。

mise は asdf の .tool-versions も読めるため、既存プロジェクトとの互換性を保ちやすい。

1つの言語しか使わないなら、その言語専用のツール（nvm, pyenv）でも十分だ。

実務での注意点

> IMPORTANT
> sudo をつけてグローバルにインストールするのは避ける。 権限の問題で後々トラブルになる。Homebrew や mise を使えば ユーザー権限の範囲内でツールを管理できる。

まとめ

公式サイトから手動インストールするのは初回だけにしよう。

以降は Homebrew（macOS）や apt（Linux）でツールを管理し、

言語のバージョン管理には mise や asdf のようなツールを使う。

これにより「あのプロジェクトでは Node.js いくつだっけ？」問題から解放される。

1.6 コンテナによるローカル開発

> このセクションで答える問い: Docker って何？なぜ開発に必要なのか？

なぜ必要か

「自分のPCでは動くのに、同僚のPCでは動かない」——

この問題はOSの違いだけでなく、ライブラリのバージョン差、設定ファイルの違い、

データベースの有無など、無数の原因で発生する。

コンテナ技術は「アプリケーションとその動作に必要なものすべてを箱に詰める」技術だ。

物流のコンテナが、中身に関係なく同じ規格の箱で輸送できるように、

ソフトウェアのコンテナも「箱ごと渡す」ことで環境の差異を吸収する。

全体像

用語	説明
Docker（ドッカー）	コンテナを作って動かすための主要ツール。入門資料も多い
イメージ（Image）	コンテナの「設計図」。Dockerfile に記述して作る
コンテナ（Container）	イメージから起動した実体。使い捨て可能
Compose	複数のコンテナをまとめて管理する仕組み。Web + DB のような構成に使う

```
graph LR
A[Dockerfile] -->|ビルド| B[イメージ]
B -->|起動| C[コンテナ1]
B -->|起動| D[コンテナ2]
E[compose.yaml] -->|管理| C
```

```
E -->|管理| D
```

選定の判断軸

macOS / Windows のローカル開発では Docker Desktop を基準に考えるのがわかりやすい。

Docker Desktop には Docker Engine、Docker CLI、Compose が含まれており、

Docker 公式も導入経路の1つとして案内している。

Linux では Docker Engine と Compose プラグインを直接入れる構成も一般的だ。

代替として OrbStack（macOS専用）がある。軽量な選択肢として、

macOS ユーザーの候補に入る。

ツール	OS	特徴
Docker Desktop	macOS / Windows / Linux	公式。GUI あり。Compose もまとめて使える
OrbStack	macOS のみ	軽量な代替。チームでの採用可否は事前確認が必要
Podman	Linux / macOS / Windows	デーモンレスな代替。Docker 互換だが完全一致ではない

実務での注意点

> WARNING
>
> Docker Desktop は macOS/Windows でメモリを多く消費する。
> メモリ8GBのPCでは開発が厳しくなるため、最低16GBを推奨する。

> NOTE
>
> Docker は「本番環境でも使う」技術であり、ローカル開発だけの道具ではない。
> ここで基礎を学んでおくと、第7章（クラウドとホスティング）と 第10章（CI/CD）の理解がスムーズになる。

まとめ

Docker は「環境ごと箱に詰めて配る」技術。チーム開発の環境差異問題を大きく減らせる。

まずは Docker Desktop またはチーム標準の実装を使い、

Dockerfile と Compose の設定ファイル（compose.yaml や docker-compose.yml）を読めるようになれば十分だ。

1.7 dotfiles と開発環境の再現性

> このセクションで答える問い: PCを買い替えたとき、開発環境を素早く復元するには？

なぜ必要か

開発環境のセットアップには、シェルの設定、エディタのプラグイン、

Git の設定、各種ツールのインストールなど、数十の手順がある。

これを毎回手作業でやるのは時間の無駄であり、漏れやミスの原因にもなる。

PCの故障、買い替え、新しいチームメンバーのオンボーディング――

開発環境を「ゼロから再現」する場面は必ず訪れる。

全体像

dotfiles（ドットファイル）は、Unix系OSで . から始まる設定ファイルの総称だ。

.zshrc、.gitconfig、.vimrc などがこれにあたる。

これらを Git リポジトリで管理し、セットアップスクリプトを用意しておけば、

新しいPCでもコマンド1つで同じ環境を再現できる。

dotfiles リポジトリの典型的な構成:

```
dotfiles/
├── .zshrc # シェル設定
├── .gitconfig # Git 設定
├── .editorconfig # エディタ設定
├── Brewfile # Homebrew でインストールするツール一覧
└── setup.sh # セットアップスクリプト
```

選定の判断軸

dotfiles の管理方法はシンプルな方がよい:

方法	複雑さ	説明
Git + シンボリックリンク	低	最も一般的。ln -s でホームディレクトリにリンクを張る
GNU Stow	低	シンボリックリンクの作成を自動化するツール
chezmoi	中	テンプレート機能あり。マシンごとに設定を微調整できる
Nix / Home Manager	高	完全に宣言的に環境を定義。学習コスト高

初心者は Git + シンボリックリンク で十分。

複数マシンで微妙に設定を変えたい場合は chezmoi を検討する。

実務での注意点

> WARNING
>
> dotfiles リポジトリを パブリック（公開） にする場合、 APIキーやトークンが含まれていないか必ず確認すること。
> .gitconfig にGitHubのトークンを書いていた、という事故は珍しくない。

> TIP
>
> Homebrew ユーザーは brew bundle dump で現在インストール済みのツール一覧を Brewfile
> に出力できる。新しいPCでは brew bundle install で一括復元できる。

まとめ

dotfiles を Git で管理し、セットアップスクリプトを用意しておけば、

PCの買い替えや新メンバーのオンボーディングが劇的に楽になる。

最初は .zshrc と .gitconfig の2ファイルから始めれば十分だ。

「環境構築の再現性」は、チーム開発の生産性を左右する地味だが重要な要素だ。

1.8 AI時代の現在地：開発環境の2026年スナップショット

このセクションで答える問い: 2026年時点で、AIツールは開発環境の中でどう位置づけるべきか？

なぜ必要か

AI を無視して開発環境を語るのは、現在では不自然になった。

主要なエディタ、IDE、ターミナルツール、GitHub 上の開発フローまで、

AI を前提とした機能が組み込まれているからだ。

ただし、「AI を使うかどうか」はもう論点ではない。

本当に重要なのは、どこまで任せるか、どう品質を守るか、

チームのルールをどう共有するか である。

全体像

2026年の AI ツールは、だいたい次の3層に分かれる。

レイヤー	代表例	役割
エディタ内 AI	Cursor、Windsurf、GitHub Copilot、JetBrains AI Assistant	補完、説明、局所編集、コードベース質問

レイヤー	代表例	役割
ターミナル型エージェント	Claude Code、Copilot CLI	調査、計画、複数ファイル編集、コマンド実行
PR / GitHub 上のエージェント	Copilot coding agent など	Issue からの実装、テスト実行、PR 作成

以前は「エディタの横にあるチャットツール」という位置づけだったが、

現在は エディタ・ターミナル・レビュー工程そのものに入り込んでいる。

選定の判断軸

1つのツールですべてを解決しようとしない方がよい。

現実的には、次の組み合わせが扱いやすい。

1. 日常作業用のエディタ AI を1つ選ぶ

VS Code + Copilot、Cursor、JetBrains など。

2. 重い変更用のエージェントを1つ持つ

Claude Code や Copilot coding agent のような、自律度の高い道具を補助線として使う。

3. 品質管理は従来どおり人間と自動化で担保する

テスト、リンタ、型チェック、レビューは依然として必須である。

実務での注意点

> WARNING
>
> AI レビューは人間レビューの代替ではない GitHub も Copilot code review
> を「人間のレビューを補完するもの」と位置づけている。
> 生成コードもレビューコメントも、そのまま鵜呑みにしてはいけない。

> IMPORTANT
>
> リポジトリ固有ルールを文章化する 2026年の開発環境では、AI に何を守らせるかが生産性を左右する。
> AGENTS.md、エディタ固有のルールファイル、レビュー指針を置いておくと、AI の出力のばらつきを大きく減らせる。

> TIP
>
> ジュニアほど「生成させる」より「説明させる」使い方から始めるとよい。
> まずはコードの説明、テストケース案、リファクタリング案に使い、
> 差分を読めるようになってから自動編集の比率を上げる方が安全だ。

まとめ

2026年の開発環境において、AI は独立した別ツールではなく、

エディタ・ターミナル・レビュー工程に組み込まれた一部 になっている。

ただし、AI に任せるほど、ルール・テスト・レビューの重要性はむしろ増す。

「AI を使う」よりも「AI を制御できる環境を作る」ことが重要である。

この章のチェックリスト

- [ ] 自分のOSで開発を始める手順（macOS → Homebrew / Windows → WSL2）を説明できるか
- [ ] 「ターミナル」と「シェル」の違いを人に説明できるか
- [ ] zsh / bash / fish の違いと、初心者にどれを勧めるか根拠を持って答えられるか
- [ ] VS Code / Cursor / JetBrains / Claude Code の役割の違いを説明できるか
- [ ] ローカル開発環境とクラウド開発環境をどう使い分けるか説明できるか
- [ ] 開発環境の「再現性」がなぜ重要かを説明できるか

第2章 プログラミング言語の選び方

> この章を読む前に: 第1章で開発環境を整えていることを前提とする。
> プログラミングの基本（変数、条件分岐、ループ、関数）を学習済みであると理解が深まる。

プログラミング言語は道具だ。日曜大工で「ドライバーとハンマー、どちらが優れているか」を

議論しても意味がないように、言語にも「万能の最強言語」は存在しない。

ネジを回すならドライバー、釘を打つならハンマーが正解であるのと同じように、

作りたいものと状況に応じて適切な言語を選ぶ能力が重要だ。

この章では、2026年時点で主要なプログラミング言語を

「何を作りたいか」という軸で整理し、選定の判断基準を提供する。

2.1 言語選定の基本判断軸

> このセクションで答える問い: プログラミング言語を選ぶとき、何を基準にすればよいか？

なぜ必要か

言語選定を間違えると、プロジェクトの途中で「この言語ではやりたいことが難しい」

「保守できる人がいない」といった問題に直面する。

特にチームやプロダクトの規模が大きくなるほど、言語の変更コストは膨大になる。

最初の選定が重要なのはそのためだ。

全体像

言語選定で考慮すべき主要な軸は次の6つだ。

軸	問い	見るべきポイント
型システム	実行前にどこまで誤りを防げるか？	静的型付けか、型ヒントで補強できるか
エコシステム	必要なライブラリやフレームワークが揃っているか？	Web、AI、モバイル、運用など領域ごとの成熟度
採用市場	その言語を扱える人を採用・育成しやすいか？	既存チームの経験、学習資料、求人の多さ
学習コスト	チームが戦力化するまでの時間はどれくらいか？	文法の複雑さ、周辺ツール、開発体験
運用特性	ビルド、デプロイ、性能、メモリ効率は要件に合うか？	単一バイナリ化、起動速度、実行環境

軸	問い	見るべきポイント
AI支援のしやすさ	AI 補助を使ったときに品質を保ちやすいか？	公式ドキュメント、サンプル、型・テストの充実度

> IMPORTANT
>
> 「好きな言語」と「プロジェクトに適した言語」は異なる。
> 個人開発なら好みで選んでよいが、チーム開発では上の6軸を総合的に判断する。

まとめ

言語選定は「何を作るか」「誰と作るか」「どう運用するか」で決まる。

技術的な美しさだけでなく、採用・保守・AI支援の使いやすさまで含めて判断しよう。

2.2 主要プログラミング言語の詳細分析

> このセクションで答える問い: 各言語はどんな特徴を持ち、どんな場面で強いのか？

なぜ必要か

言語の名前は知っていても、それぞれの得意領域や文化を理解していなければ、

適切な選定はできない。ここでは 2026年時点で実務でよく使われる言語を、

「何に向いているか」という観点で整理する。

全体像

TypeScript / JavaScript

項目	内容
ひとこと	Web 開発の中心言語。TypeScript は JavaScript に型を加える
型システム	TypeScript は静的型付け、JavaScript は動的型付け
得意領域	Webフロントエンド、Node.js バックエンド、フルスタック開発、サーバーレス
強み	ブラウザで動く唯一の標準言語。フロントとバックで言語を揃えやすい
弱点	ツールチェーンが多く、設定が複雑になりやすい。大規模ではビルドも重くなりやすい

> NOTE
>
> 新規の Web チーム開発では、JavaScript より TypeScript の方が扱いやすいことが多い。
> 型情報があると、リファクタリングとレビューの負担を減らしやすい。

Python

項目	内容
ひとこと	読みやすさ重視。AI・データ・自動化に強い
型システム	動的型付け。型ヒントで補強できる
得意領域	データ分析、機械学習、Web API、スクリプティング、業務自動化
強み	AI/ML ライブラリが非常に強い。学習資料も多く、試作が速い
弱点	実行性能は言語要件によっては不足する。パッケージ管理の選択肢も複数ある

> TIP
>
> Python では従来の venv + pip に加え、uv のような高速なツールも有力になっている。
> ただし、チームで使うなら「何を標準にするか」を最初に決めておく方が重要だ。

Go

項目	内容
ひとこと	シンプルさと運用しやすさを両立したコンパイル言語
型システム	静的型付け
得意領域	Web API、マイクロサービス、CLI、インフラツール
強み	単一バイナリで配布しやすい。標準ライブラリが強く、運用が単純になりやすい
弱点	表現力は控えめで、抽象化を多用する設計には向きにくい

> NOTE
>
> Go は「誰が書いても似た形になりやすい」ことを重視した言語である。 そのため、長期運用する API やインフラ系のコードで強みが出やすい。

Rust

項目	内容
ひとこと	安全性と性能を強く両立したいときの選択肢
型システム	静的型付け。所有権によるメモリ安全性
得意領域	システムプログラミング、CLI、WebAssembly、性能重視の部品
強み	実行性能が高く、メモリ安全性をコンパイル時に担保しやすい
弱点	学習コストが高い。所有権と借用の理解に時間がかかる

> IMPORTANT
>
> Rust は「何にでも使う第一言語」というより、 性能、安全性、低レベル制御が本当に必要か で選ぶ言語である。

その他の確立された言語

言語	位置づけ
Java / Kotlin	エンタープライズ開発の定番。Kotlin は Android 開発でも中心的
Ruby	Rails による高速開発と読みやすさに強みがある
PHP	Web の既存資産が大きい。Laravel による現代的な開発も強い
C#	.NET の業務開発、Windows 開発、Unity のゲーム開発で強い
Swift	iOS / macOS アプリ開発の基本言語

新興言語の現在地

言語	現在地	どう見るべきか
Zig	低レベル制御と C 互換性で注目	学習や検証対象としては有望だが、第一候補にするには慎重さが必要
Mojo	AI 計算系で注目される新顔	将来性はあるが、実務採用はまだ限定的

言語別のざっくり比較

観点	TypeScript	Python	Go	Rust	Java / Kotlin
主戦場	Web 全般	AI / データ / 自動化 / Web API	API / インフラ / CLI	システム / ツール / 高性能部品	業務システム / Android
学習のしやすさ	中	高	中	低	中
デプロイのしやすさ	ランタイム依存	仮想環境管理が必要	単一バイナリ化しやすい	単一バイナリ化しやすい	JVM / Android 前提
型による保護	強い	任意	強い	非常に強い	強い
AI支援との相性	強い	強い	良好	良好だが学習コストは高い	良好

まとめ

Web 開発なら TypeScript、AI / データなら Python、

運用しやすい API やインフラなら Go、性能と安全性を最優先するなら Rust が基本線になる。

大事なのは、「流行っているか」より その言語の強みが要件に合っているか である。

2.3 用途別の推奨マッピング

> このセクションで答える問い:「〇〇を作りたい」とき、どの言語を選ぶべきか？

なぜ必要か

言語の特徴を知っていても、実際に案件へ当てはめる段階で迷うことは多い。

ここでは用途から逆算して、最初の候補を絞る。

全体像

作りたいもの	第一候補	第二候補	理由
Webフロントエンド	TypeScript	-	ブラウザで動く標準言語は JavaScript 系
Web API（小〜中規模）	TypeScript	Python	フロントとの統一や開発速度を取りやすい
Web API（高スループット / 運用重視）	Go	Java / Kotlin	配布と運用が比較的単純で、並行処理にも向く
データ分析・機械学習	Python	R	ライブラリと教材が豊富
iOS アプリ	Swift	-	Apple プラットフォームで自然な選択
Android アプリ	Kotlin	-	Android での定番
クロスプラットフォームアプリ	Flutter	React Native / Kotlin Multiplatform	1つのコードベースで複数 OS を狙える
CLI ツール	Go	Rust / Python	配布形態と性能要件で選ぶ
インフラツール	Go	Rust / Python	クラウド運用や自動化との相性がよい
ゲーム	C#	C++	エンジン選定と一体で決まることが多い

選定の判断軸

```
graph TD
A[何を作りたい?] --> B{Webアプリ?}
B -->|はい| C{フロントかバックか}
C -->|フロント| D[TypeScript]
C -->|バック| E{重視するのは?}
E -->|開発速度| F[TypeScript / Python]
E -->|運用の単純さ| G[Go]
E -->|既存JVM資産| H[Java / Kotlin]
B -->|いいえ| I{モバイルアプリ?}
I -->|はい| J{ネイティブか共通化か}
J -->|ネイティブ| K[Swift / Kotlin]
J -->|共通化| L[Flutter / React Native]
I -->|いいえ| M{AI / データ処理?}
M -->|はい| N[Python]
M -->|いいえ| O{性能最優先か}
O -->|はい| P[Rust]
```

```
O -->|いいえ| Q[Go / Python]
```

> NOTE
> この表で TypeScript、Python、Go が繰り返し出てくるのは、
> 言語自体が優れているからというより、用途の広さと保守のしやすさ が大きいからだ。

実務での注意点

> WARNING
> 用途別の推奨よりも、次の条件が優先されることがある。
> 既存チームの経験、既存システムとの接続、法規制、性能要件、採用可能性である。
> 「一般論ではこちらが良い」より「このチームで今動かせるか」の方が重要な場面は多い。

まとめ

迷ったら、Web なら TypeScript、AI / データなら Python、

インフラや運用を重視する API なら Go を出発点にすれば大きく外しにくい。

ただし、最終判断は既存資産とチーム能力を含めて行う。

2.4 AI時代の現在地

> このセクションで答える問い: 言語選定の際に、AI時代特有の新しい考慮軸は何か？

なぜ必要か

AI 補助ツールが日常化したことで、言語選定にも新しい観点が増えた。

同じ機能を実装する場合でも、サンプルコード、公式ドキュメント、

型情報、テスト基盤が豊富な言語の方が AI を安全に使いやすい。

ただし、AI が得意そうだからその言語を選ぶ のは順序が逆である。

まず要件に合う言語を選び、その上で AI 支援の受けやすさを比較するのが正しい。

全体像

AI時代に見るべき言語の条件は次のとおりだ。

観点	AI支援で有利になりやすい条件	例
公式情報の豊富さ	公式ドキュメントや定番ライブラリが整っている	TypeScript, Python, Java
型と静的解析	型チェックやリンタで AI の誤りを早く炙り出せる	TypeScript, Go, Rust, Kotlin
サンプルコードの多さ	類似実装やベストプラクティスが見つけやすい	TypeScript, Python
テストしやすさ	自動テストで AI 出力をすぐ検証できる	どの言語でも重要

選定の判断軸

AI 時代でも、判断順序は次のままでよい。

1. プロダクト要件に合うか

ブラウザ、モバイル、AI推論、低レベル制御など、用途が最優先。

2. チームで保守できるか

AI が生成しても、最終的に読むのは人間である。

3. AI支援で品質を保ちやすいか

型、テスト、公式ドキュメント、サンプルの多さが効く。

実務での注意点

> WARNING
>
> AI が書きやすい言語 = 最適な言語 ではない。 たとえば AI が TypeScript をよく書けても、数値計算や ML パイプラインなら Python の方が自然である。 ツール都合で言語を選ぶのではなく、要件都合で選ぶ。

> IMPORTANT
>
> AI 出力は公式ドキュメントで裏取りする 言語仕様やライブラリ API は変わる。AI はもっともらしいが古いコードを出すことがあるため、
> 特に新しい機能や標準ライブラリの話は必ず公式情報で確認する。

まとめ

AI時代の言語選定では、エコシステムの大きさそのものよりも、

型・テスト・公式情報が整っていて、AI の出力を検証しやすいか が重要である。

AI は加速装置であり、言語選定の主語ではない。

2.5 2025-2026年の言語トレンド

> このセクションで答える問い: 今、言語の世界では何が起きているのか？

なぜ必要か

技術選定は流行だけで決めてはいけないが、流れを無視するのも危険だ。

どの言語に新しいツールが集まり、どの領域で需要が強いかを知っておくと、

学習投資の優先順位を決めやすい。

全体像

トレンド	何が起きているか	実務上の意味
TypeScript の定着	Web フロントだけでなく、フルスタック開発でも中心的	Web チームの共通言語として扱いやすい
Python ツールチェーンの改善	uv などでパッケージ管理・実行体験が改善	Python を採用しやすくなっている
Go の安定感	クラウドネイティブ、API、CLI で継続的に強い	長期運用するサービスで選びやすい
Rust の影響力拡大	アプリ本体だけでなく開発ツールや基盤部品でも存在感がある	性能と安全性が必要な局面で候補に上がりやすい
既存言語の粘り強さ	Java / Kotlin、C#、Swift、PHP、Ruby も依然重要	既存資産の大きい領域では主役が変わっていない
新興言語は観測対象	Zig、Mojo などは注目されるが、標準選択肢にはまだ距離がある	学習や検証は有益だが、本番採用は慎重に見る

選定の判断軸

トレンドを見るときは、次の3点を分けて考えるとよい。

1. 学習対象として面白いか

新しい考え方に触れられるか。

2. 業務で採用しやすいか

チーム人数、保守性、採用市場に合うか。

3. 今のプロダクト課題を解決するか

性能、開発速度、運用性の改善に本当に効くか。

実務での注意点

WARNING

新しい言語やツールに飛びつく前に、採用市場と保守体制を確認しよう。
個人学習では新技術を追う価値が高いが、チームのプロダクトでは
「詳しい人が1人しかいない技術」は大きな運用リスクになる。

まとめ

2025-2026年の大きな流れは、**Web では TypeScript、AI / データでは Python、

クラウド運用では Go、性能と安全性では Rust** という住み分けがさらに明確になったことだ。

一方で、既存の業務システムやモバイルでは Java / Kotlin、C#、Swift などの重要性は続いている。

この章のチェックリスト

- [ ] 言語選定の6つの判断軸（型、エコシステム、採用市場、学習コスト、運用特性、AI支援のしやすさ）を説明できるか
- [ ] TypeScript, Python, Go, Rust の得意領域をそれぞれ説明できるか
- [ ] Web API を作るときに、TypeScript / Python / Go をどう使い分けるか説明できるか
- [ ] 静的型付けと動的型付けの違いを非エンジニアに説明できるか
- [ ] AI時代でも「要件に合う言語を先に選ぶべき理由」を説明できるか
- [ ] uv の位置づけを、既存の venv + pip と対比して説明できるか
- [ ] Zig や Mojo を「学習対象」と「本番採用候補」で分けて評価できるか

第3章 フロントエンド開発

> この章を読む前に: HTML/CSS の基本構造（タグ、セレクタ）と、 JavaScript の基礎（変数、関数、イベント）を理解していると読みやすい。
> 不安がある場合は第2章のTypeScript/JavaScriptの項を先に読むことを推奨する。

Webアプリケーションにおいて、ユーザーが直接目にし、操作する部分を

フロントエンド と呼ぶ。レストランでいえば「客席と接客」にあたる部分だ。

料理がどれほど美味しくても（バックエンドが優秀でも）、

メニューが読めなかったり注文の仕方がわからなかったりすれば（UIが悪ければ）、

客は帰ってしまう。

この章では、フロントエンド開発の全体像と、フレームワーク選定の判断基準を整理する。

3.1 フロントエンドの全体像

> このセクションで答える問い: フロントエンド開発にはどんなアプローチがあるのか？

なぜ必要か

2000年代のWebサイトは、サーバーがHTMLを生成してブラウザに返すシンプルな仕組みだった。

しかしGmail（2004年）やGoogle Maps（2005年）の登場により、

「ページ全体を再読み込みせず、部分的に更新する」リッチなWebアプリが求められるようになった。

この需要に応えるために、フロントエンドの技術は急速に進化し、複雑化した。

2026年時点では、複数のアーキテクチャパターンが並存しており、

プロジェクトの性質に応じて適切なものを選ぶ必要がある。

全体像

パターン	正式名称	仕組み	向いている用途
SSR	Server-Side Rendering	サーバーでHTMLを生成してブラウザに返す	SEO重視のサイト、初回表示速度が重要なページ
SPA	Single Page Application	初回にJSを一括読み込み、以降はブラウザ内で画面遷移	管理画面、ダッシュボード等のログイン後アプリ
SSG	Static Site Generation	ビルド時にHTMLを事前生成	ブログ、ドキュメントサイト、LP

パターン	正式名称	仕組み	向いている用途
ISR	Incremental Static Regeneration	SSGの発展形。一定間隔でページを再生成	ECサイトの商品ページ等、更新頻度が中程度
Islands	Islands Architecture	ページの大部分は静的HTML、インタラクティブ部分だけJSで動かす	コンテンツ中心のサイトに部分的な動的機能を追加

```
graph LR
subgraph "サーバー側で処理"
SSR
SSG
end
subgraph "ブラウザ側で処理"
SPA
end
subgraph "ハイブリッド"
ISR
Islands
end
SSG -->|更新頻度UP| ISR
SPA -->|SEO必要| SSR
SSR -->|静的部分多| Islands
```

選定の判断軸

```
graph TD
A[どんなサイト?] --> B{SEOが重要?}
B -->|はい| C{更新頻度は?}
C -->|高い| D[SSR: Next.js / Nuxt]
C -->|低い| E[SSG: Astro / Next.js]
B -->|いいえ| F{どんなアプリ?}
F -->|管理画面/SaaS| G[SPA: React + Vite]
F -->|コンテンツ+少しの動的機能| H[Islands: Astro]
```

実務での注意点

> WARNING
>
> 「とりあえずSPA」は危険。SEOが不要な管理画面ならSPAで問題ないが、
> 検索エンジンからの流入が重要なサービスでは、SSR や SSG を検討すべきだ。

まとめ

フロントエンドのアーキテクチャは一択ではない。

SEOの必要性、更新頻度、インタラクティブ性の3つを軸に判断する。

2026年時点の主流は、サーバー側で先に描画しつつ、必要な部分だけクライアントで動かす

ハイブリッド型だ。

3.2 フレームワーク選定

このセクションで答える問い: React, Vue, Svelte…結局どれを選べばいいのか？

なぜ必要か

フロントエンドフレームワークなしで複雑なWebアプリを作るのは、

設計図なしでビルを建てるようなものだ。

状態管理、コンポーネントの再利用、ルーティング、データの取得……

これらを毎回ゼロから実装するのは非現実的であり、バグの温床になる。

フレームワークは、こうした共通課題のベストプラクティスをパッケージ化したものだ。

全体像

選択肢	主な組み合わせ	強み	向いている場面
React	Next.js, Remix	情報量が多く、採用市場も大きい。コンポーネント資産が豊富	中長期で人を集めやすい構成を取りたいとき
Vue	Nuxt	テンプレートが読みやすく、学習コストが比較的低い	フロントエンド経験が浅いチーム、Vue 資産がある組織
Svelte	SvelteKit	記述量が少なく、状態とテンプレートの結びつきがわかりやすい	小〜中規模で軽快に作りたいとき
Astro	Astro + 任意UIフレームワーク	コンテンツ中心で、必要な部分だけ対話的にしやすい	ドキュメント、ブログ、メディア、LP
Angular	Angular 単体	フレームワーク内の規約が強く、大規模チームで揃えやすい	エンタープライズ色の強い大規模開発

選定の判断軸

観点	React / Next.js	Vue / Nuxt	Svelte / SvelteKit	Astro
情報量	非常に多い	多い	比較的少ない	多い
学習のしやすさ	中	高い	高い	高い
対話的 UI	強い	強い	強い	必要部分に絞る設計が得意
コンテンツ中心サイト	可能	可能	可能	特に得意
採用・引き継ぎ	しやすい	しやすい	チーム依存	用途次第

迷ったときの考え方

- 採用市場、情報量、既存ライブラリ資産を重視するなら React / Next.js

- 読みやすさと学習速度を重視するなら Vue / Nuxt

- 小さく始めて軽快に作りたいなら Svelte / SvelteKit

- コンテンツ中心で JavaScript を送りすぎたくないなら Astro

実務での注意点

> WARNING
>
> フレームワークの人気だけで選ばないこと。
> SEO、認証、管理画面、リアルタイム性、デザインシステム、チーム経験の方が 実務でははるかに重要である。

> IMPORTANT
>
> Next.js を選ぶ場合は、App Router と Server Components を前提に考えるとよい。 一方で、すべての画面を Next.js で統一する必要はない。 単純な管理画面や埋め込みアプリでは、Vite ベースの SPA の方が単純なこともある。

まとめ

2026年時点でも、一番人気の技術 = すべての案件の正解 ではない。

React / Next.js は無難な選択肢だが、Vue / Nuxt、Svelte / SvelteKit、Astro には

それぞれ明確な強みがある。大事なのは「この案件で何を最適化したいか」を先に決めることだ。

3.2.1 Next.js App Router と Server Components

このセクションで答える問い: Server Components とはなにか？Next.js の設計はどう変わったのか？

なぜ必要か

従来の React フロントエンドでは、データ取得も画面表示もクライアント側へ寄りがちだった。

その結果、初回表示が重くなりやすく、SEO や低速回線で不利になることがあった。

Next.js の App Router は、どこまでをサーバーで描画し、どこからをクライアントで動かすか

を明示的に分けやすくした。これが現在の Next.js を理解するうえで重要な変化である。

全体像

React の Server Components は、React 公式では「バンドル前に別環境で先にレンダリングされる新しい種類のコンポーネント」と説明されている。

Next.js App Router では、この考え方が既定の設計に組み込まれている。

概念	役割	典型例
Server Components	サーバーで実行され、結果をクライアントへ渡す	データ取得、認可後の読み取り、重い整形処理
Client Components	ブラウザで実行され、対話を担当する	useState、イベント処理、ブラウザ API 利用
'use client'	クライアント境界を宣言する	ボタン、フォーム、モーダル、ドラッグ操作

```
// app/products/page.tsx
import ProductGrid from './product-grid';

export default async function Page() {
const products = await fetchProducts();
return <ProductGrid products={products} />;
}

// app/products/product-grid.tsx
'use client';

import { useState } from 'react';

export default function ProductGrid({ products }) {
const [query, setQuery] = useState('');
return <>{/* 検索入力や並び替えなどの対話処理 */}</>;
}
```

選定の判断軸

Server Components が特に有効なのは次の場面だ。

- 初回表示を軽くしたい

- データ取得をサーバー側へ寄せたい

- クライアントへ送る JavaScript を減らしたい

逆に、次の場面では Client Components や従来型のクライアント状態管理が重要になる。

- フォーム入力、モーダル、ドラッグ、キーボード操作など対話中心の UI

- リアルタイム更新、楽観的更新、オフライン対応

- ブラウザ API へ強く依存する機能

> TIP
>
> Next.js の公式ドキュメントでも、Layouts と Pages は App Router で Server Components が既定になっている。
> まずサーバーで済む部分を Server Components に置き、必要な箇所だけ Client Components
> に切り出すと整理しやすい。

実務での注意点

> WARNING
>
> Server Components は「何でも安全」という意味ではない。 Next.js は server-only による保護やビルド時チェックを提供しているが、 2025年12月には App Router / RSC に関わる重大なセキュリティアドバイザリ CVE-2025-66478 も公開されている。App Router を使うなら、パッチ適用を継続すること。

> IMPORTANT
>
> Next.js 16 では Turbopack が新規プロジェクトの既定バンドラとなり、 Cache Components が導入された。これらは便利だが、まず理解すべき基本は Server Components と Client Components の境界 である。

まとめ

Next.js の現在地は、「React をブラウザで全部動かす」ではなく

サーバーとクライアントの責務を分ける 設計にある。

Server Components は強力だが、対話的 UI は依然として Client Components が担う。

この境界を正しく設計できるかが、Next.js を使いこなせるかどうかを決める。

3.3 状態管理の考え方

> このセクションで答える問い: 「状態」とは何か？なぜ管理が必要なのか？

なぜ必要か

Webアプリケーションには「状態（state）」がある。

ログイン中のユーザー名、ショッピングカートの中身、フォームの入力値、

ダークモードのON/OFF——これらはすべて状態だ。

小さなアプリでは状態管理に困らないが、アプリが成長すると

「この画面で変更した値が別の画面に反映されない」

「同じデータを複数箇所で持っていて不整合が起きる」といった問題が発生する。

全体像

状態は次の3種類に分けると整理しやすい。

レイヤー	説明	例	主な扱い方
ローカル状態	1つのコンポーネント内で閉じる状態	フォーム入力、タブ切り替え、モーダル開閉	useState、useReducer

レイヤー	説明	例	主な扱い方
サーバー状態	サーバー由来で再取得や同期が必要な状態	商品一覧、通知一覧、ユーザー情報	Server Components、TanStack Query、SWR
共有 UI 状態	画面をまたいで共有したい UI 状態	テーマ、サイドバー開閉、一時的な選択状態	Context、Zustand、Jotai、Pinia

選定の判断軸

> IMPORTANT
>
> まず ローカル状態で閉じられないか を考える。 次に サーバー状態として扱えないか を考える。
> グローバル状態ライブラリは最後に検討する。

React 系でよくある選択肢は次のとおり。

手法	向いている場面	注意点
useState / useReducer	コンポーネント内の状態	最初の選択肢
Context	深い階層へ少量の共有状態を渡す	更新頻度が高い値を大量に入れすぎない
TanStack Query / SWR	再取得、キャッシュ、ミューテーションがあるサーバー状態	UI 状態とは分けて考える
Zustand / Jotai	共有 UI 状態を軽量に持ちたい	何でも入れると肥大化する
Redux Toolkit	厳密な更新フローが必要な大規模アプリ	小規模では過剰になりやすい

> TIP
>
> Next.js App Router では、読み取り中心のデータ取得を Server Components へ寄せやすい。
> ただし、リアルタイム更新、クライアントキャッシュ、楽観的更新が必要なら TanStack Query のような道具は今も有効である。

まとめ

状態管理は「ローカル → サーバー → 共有 UI」の順で小さく考える。

最初から大きな状態管理ライブラリを入れるより、どの種類の状態かを見分けること の方が重要だ。

3.4 スタイリング手法

> このセクションで答える問い: CSSの書き方にはどんな選択肢があり、何を選ぶべきか？

なぜ必要か

Webアプリの見た目を整えるのがCSS（Cascading Style Sheets）だ。

小規模なら素のCSSで問題ないが、アプリが大きくなると「クラス名の衝突」

「使われていないスタイルの蓄積」「デザインの一貫性の崩壊」が起きる。

これらの問題を解決するために、さまざまなスタイリング手法が生まれた。

全体像

手法	仕組み	強み	注意点
Tailwind CSS	ユーティリティクラスを組み合わせる	実装速度が高く、デザインルールを揃えやすい	HTML が長くなりやすい
CSS Modules	ファイル単位でスタイルを閉じる	素の CSS に近く、衝突を避けやすい	共通トークン設計は別途必要
vanilla-extract	TypeScript から CSS を生成する	型とトークン設計を合わせやすい	初期学習コストがある
CSS-in-JS	JS 内でスタイルを記述する	コンポーネントとの凝集度が高い	ランタイムコストや RSC との相性を確認する必要がある

選定の判断軸

- Tailwind CSS: 速度と一貫性を優先する新規開発向け

- CSS Modules: 素の CSS に近い考え方で進めたいチーム向け

- vanilla-extract: デザイントークンを型安全に管理したいとき

- CSS-in-JS: 既存資産がある場合や、特定の設計と密接に結びついている場合

> IMPORTANT
>
> Tailwind CSS は便利だが、CSS の基礎知識を置き換えるものではない。
> display、flex、grid、余白、レイアウト、セレクタの理解は依然必要である。

> TIP
>
> AI 生成の UI は Tailwind で出てくることが多いが、 それだけを理由に Tailwind を選ぶ必要はない。
> チーム標準へ変換する前提で使ってもよい。

まとめ

新規案件では Tailwind CSS が有力候補だが、唯一の正解ではない。

長期保守まで考えるなら、チームが読み続けられる書き方か を優先して選ぶべきだ。

3.5 ビルドツールチェーン

このセクションで答える問い: なぜ書いたコードをそのままブラウザで動かさないのか？

なぜ必要か

現代のフロントエンド開発では、TypeScript、JSX、最新のJavaScript構文、

CSS のプリプロセッサなど、ブラウザが直接理解できない形式でコードを書く。

これらを「ブラウザが実行できる形」に変換する工程が ビルド だ。

工場に例えるなら、原材料（TypeScript、JSX）を加工して

製品（最適化されたJS/CSS/HTML）を出荷する生産ラインがビルドツールチェーンだ。

全体像

ツール	役割	現在地
Vite	開発サーバー + ビルド	SPA や軽量なフロントエンド開発の基準点
Turbopack	Next.js 向けバンドラ	Next.js に組み込まれ、ローカル開発体験を改善する中心技術
esbuild	高速トランスパイル / バンドル基盤	多くのツールの内部で使われる
Biome	リント / フォーマット	1つのツールに寄せたいチームの候補
ESLint + Prettier	リント / フォーマット	依然として実務の標準構成の1つ

選定の判断軸

Next.js を使う場合

Next.js 16 では、Turbopack が新規プロジェクトの既定バンドラになった。

まずはフレームワーク標準をそのまま使い、問題が出てから個別設定を考える方がよい。

Next.js 以外の SPA / SSG

Vite を基準に考えるのがわかりやすい。

React、Vue、Svelte いずれでも導入しやすく、公式ガイドも充実している。

リント / フォーマット

ESLint プラグイン資産に強く依存するなら ESLint + Prettier。

道具を減らしたいなら Biome を検討する。

実務での注意点

> WARNING
>
> ビルド速度の比較は、プロジェクト構成で大きく変わる。
> ベンチマーク記事だけで判断せず、実際のリポジトリで計測すること。

> TIP
> ツールを入れ替える前に、依存関係の削減、コード分割、不要なプラグイン除去の方が 効くことも多い。

まとめ

2026年の実務では、Next.js なら Turbopack、その他は Vite という整理がわかりやすい。

ただし、重要なのは「最速の道具を選ぶこと」より、

チームが理解できる構成を保つこと である。

3.6 テスト戦略

このセクションで答える問い: フロントエンドでは何をどうテストすればよいのか？

なぜ必要か

「画面をクリックして目で確認する」は最も原始的なテストだ。

コードが変わるたびに全画面を手動チェックするのは現実的でない。

自動テストを書くことで、変更のたびに「壊れていないか」を機械的に検証できる。

全体像

テスト種別	何を確認するか	主な道具	優先度
ユニットテスト	計算、変換、バリデーション	Vitest	高い
コンポーネントテスト	UI の描画、イベント、状態変化	Testing Library + Vitest	高い
E2Eテスト	重要なユーザーフロー	Playwright	高い
ビジュアル確認	見た目の崩れ	Playwright の screenshot assertions、Storybook 系ツール	中
アクセシビリティ確認	キーボード操作、ラベル、コントラスト	axe、Lighthouse、手動確認	高い

選定の判断軸

> TIP
> すべてを完璧にテストしようとしないこと。 まずは 壊れると困るロジック と 主要フロー に絞る。

実務では、次の組み合わせが現実的だ。

1. Vitest でロジックを守る

2. Testing Library で主要コンポーネントを守る

3. Playwright でログイン、購入、送信などの重要フローを守る

実務での注意点

> WARNING
> AI が生成したコードも、通常コードと同じ品質ゲートに通す。「AI
> が書いたから別枠で扱う」のではなく、テスト、静的解析、アクセシビリティ確認、
> レビューへ通常どおり流す方が安全である。

まとめ

フロントエンドのテストは、Vitest + Playwright を軸に考えると整理しやすい。

そこへコンポーネントテストとアクセシビリティ確認を必要に応じて足していくのが実践的だ。

3.6.1 AI を活用した UI 試作

> このセクションで答える問い: プロトタイピングから本実装までの時間をどう短縮するか？

なぜ必要か

デザイン案からコンポーネント実装までを、毎回ゼロから手で書く必要はなくなってきた。

近年は、自然言語や画像から UI のたたき台を作るツールが増えており、

試作の速度を上げやすい。

全体像

ツール	主な性格	向いている場面
v0	Vercel の UI / アプリ生成ツール	Next.js / React 前提で部品を試作したいとき
Bolt	ブラウザ上でアプリやサイトを構築する AI builder	試作から公開までを一気に触りたいとき
Lovable	自然言語から Web アプリを組み立てるフルスタック系ツール	要件を文章で素早く形にしたいとき

選定の判断軸

状況	推奨ツール	理由
既存の Next.js / Vercel 前提	v0	既存スタックへ寄せやすい
まず動くものを早く見たい	Bolt	ブラウザ中心で試作しやすい
要件を文章からアプリ化したい	Lovable	フルスタック試作との相性がよい

> IMPORTANT
> これらのツールは 試作を速くする道具 と考えるとよい。 本番品質をそのまま保証する道具ではない。

実務での注意点

> WARNING
> AI 生成 UI は「ドラフト」であって「完成品」ではない。
> アクセシビリティ、状態管理、データ取得、認証、エラーハンドリング、デザイン整合性は
> 人間が最終責任を持って確認する必要がある。

> TIP
> もっとも効果が高い使い方は、最初の叩き台を AI で出し、
> そこからチームのデザインシステムと実装規約へ寄せていく方法である。

まとめ

AI ツールは UI 試作を大きく速くする。

ただし、実務で価値が出るのは 生成速度 そのものではなく、

生成物を設計・品質基準へ合わせて仕上げられることだ。

3.7 パフォーマンス最適化の基本

> このセクションで答える問い: Webアプリの「速さ」をどう測り、どう改善するのか？

なぜ必要か

パフォーマンスは UX に直結し、検索流入やコンバージョンにも影響する。

また、近年は「読み込みの速さ」だけでなく、操作にどれだけ素早く応答するか も重要になっている。

全体像

Google が公開している Core Web Vitals が代表的な基準である。

指標	正式名称	何を測るか	目標値
LCP	Largest Contentful Paint	主要コンテンツの表示完了まで	2.5秒以下
INP	Interaction to Next Paint	操作への応答の滑らかさ	200ms以下
CLS	Cumulative Layout Shift	レイアウトのずれ	0.1以下

選定の判断軸

主な改善ポイントは次のとおりだ。

手法	効く指標	例
画像最適化	LCP, CLS	適切なサイズ、width/height、遅延読み込みの使い分け
JavaScript の削減	INP, LCP	不要な依存削減、コード分割、Client Components の絞り込み
レイアウト固定	CLS	画像、広告、埋め込み要素の領域確保
フォント最適化	LCP, CLS	サブセット化、font-display、自己ホスト検討
実測の導入	全体	Lighthouse だけでなく、実ユーザー計測も見る

> TIP
>
> web.dev でも、Core Web Vitals は実ユーザー計測を重視している。 Lighthouse は出発点として有効だが、最終判断はフィールドデータで行うべきだ。

まとめ

パフォーマンスは「測ってから改善する」が鉄則。

Chrome DevTools の Lighthouse や PageSpeed Insights で現状を測定し、

Core Web Vitals の目標値を基準に優先度をつけて改善する。

まずは 画像最適化 と コード分割 から始めるのが効果的。

3.8 AI 時代のフロントエンド開発哲学（2026年）

このセクションで答える問い: AI ツール時代に、フロントエンドエンジニアに求められるスキルは何か？

全体像

AI の普及で、フロントエンド開発の作業配分は変わった。

AI が得意な作業	人間が責任を持つべき作業
たたき台の UI 作成	情報設計、仕様の優先順位づけ
定型的なコンポーネント生成	アクセシビリティ、法令対応
テストケース案、リファクタリング案	データ境界、認証、エラー設計
ドキュメントの下書き	品質判断、最終レビュー

選定の判断軸

AI を活用するときは、次の順で考えると破綻しにくい。

1. 何を自動化してよいか決める

たとえば UI たたき台、Story 作成、テストの下書きなど。

2. 何は人間が必ず見るか決める

アクセシビリティ、状態管理、設計境界、セキュリティなど。

3. 品質ゲートを通常どおり通す

リント、型チェック、テスト、レビューを省略しない。

実務での注意点

> WARNING
>
> AI によって実装速度が上がるほど、未整理の UI や場当たり的な状態管理も増えやすい。
> 速度の向上は、そのまま品質の向上を意味しない。

> IMPORTANT
>
> フロントエンドで本当に価値があるのは、コードを速く書くことより 複雑さを制御すること である。 AI を使っても、責務分離、設計一貫性、アクセシビリティの重要性は下がらない。

まとめ

AI 時代のフロントエンドでは、実装そのものよりも

生成物の品質を見抜き、設計へ戻せる力 が重要になる。

AI は実装速度を上げるが、何を作るかを決める責任までは引き取ってくれない。

この章のチェックリスト

- [ ] SSR, SPA, SSG, Islands の違いを説明でき、使い分けの基準を言えるか
- [ ] React, Vue, Svelte の特徴と選定理由を述べられるか
- [ ] Server Components と Client Components の違いを説明できるか
- [ ] 「状態管理」の3つのレイヤーを説明できるか
- [ ] Core Web Vitals の3つの指標を言えるか
- [ ] Tailwind CSS を選ぶ理由と注意点を説明できるか
- [ ] AI UI ビルダー（v0, Bolt, Lovable）の違いと使い分けを述べられるか
- [ ] Next.js と Vite の使い分けを説明できるか
- [ ] 「AI 生成 UI をそのまま本番投入してはいけない理由」を説明できるか

第4章 バックエンド開発

> この章を読む前に: HTTPの基本（リクエスト/レスポンス、URLの構造）と、
> 第2章で紹介した言語の概要を把握していると理解がスムーズ。

フロントエンドが「客席と接客」なら、バックエンドは「厨房」だ。

注文（リクエスト）を受け取り、材料（データ）を調理（処理）し、

料理（レスポンス）を返す。お客さんからは見えないが、

レストランの品質を左右する最も重要な部分だ。

4.1 バックエンドの役割と責務

> このセクションで答える問い: バックエンドとは具体的に何をする部分なのか？

なぜ必要か

「Webサイトくらいフロントエンドだけで作れないのか？」と思うかもしれない。

静的なブログなら確かにそうだ。しかし、ユーザー登録、決済処理、データの永続化、

他サービスとの連携が必要になった瞬間、バックエンドが必要になる。

ブラウザ上のコードはユーザーが自由に読み書きできるため、

秘密の情報（APIキー、データベースの接続先）をフロントエンドに置くことはできない。

バックエンドはこうした「信頼できる環境」での処理を担う。

全体像

バックエンドが担う主な責務:

責務	説明	例
API提供	フロントエンドやモバイルアプリにデータを返す	ユーザー情報の取得、商品一覧の返却
認証・認可	「誰か」を確認し「何ができるか」を制御する	ログイン、権限チェック
ビジネスロジック	アプリケーション固有の処理	料金計算、在庫チェック、予約のバリデーション
データ永続化	データベースへの読み書き	ユーザーの作成、注文の保存
外部連携	他サービスのAPIを呼ぶ	決済（Stripe）、メール送信、SMS通知

責務	説明	例
バックグラウンド処理	時間のかかる処理を非同期で実行	レポート生成、画像のリサイズ、メール一斉送信

まとめ

バックエンドは「データを安全に管理し、ビジネスロジックを実行する信頼できる環境」。

フロントエンドが「見た目」を担うのに対し、バックエンドは「中身」を担う。

4.2 フレームワーク選定

> このセクションで答える問い: バックエンドのフレームワークは何を基準に選ぶのか？

なぜ必要か

HTTPリクエストの受付、ルーティング、リクエストの検証、レスポンスの生成……

これらを毎回ゼロから書くのは非効率だ。

バックエンドフレームワークはこれらの共通処理を抽象化し、

開発者がビジネスロジックに集中できるようにする。

全体像

バックエンドでは、まず 言語と実行環境、次に フレームワーク を選ぶ。

数値ベンチマークだけで決めるのではなく、運用・採用・周辺ツールまで含めて考える。

JavaScript / TypeScript 系の実行環境

実行環境	強み	向いている場面
Node.js	エコシステムが圧倒的。学習資料も多い	長期運用、既存資産、ライブラリ互換性を重視するとき
Bun	TypeScript 実行や開発体験をまとめやすい	新規開発で高速なローカル体験を重視するとき
Deno	TypeScript ファーストで権限モデルが明確	実行権限を厳密に管理したいとき

JavaScript / TypeScript 系フレームワーク

フレームワーク	強み	向いている場面
Hono	Web 標準寄りで、複数ランタイムへ載せやすい	エッジ、サーバーレス、軽量 API

フレームワーク	強み	向いている場面
Fastify	低オーバーヘッドで、プラグイン構造が明確	API サーバー、ストリーミング、性能重視
Express	情報量が多く、既存資産が非常に多い	既存保守、学習用、小さな API
NestJS	構造化しやすく、DI やモジュール分割がしやすい	大規模チーム、規約を強めたい組織

Python 系

フレームワーク	強み	向いている場面
FastAPI	型ヒントからバリデーションや API ドキュメントを生成しやすい	API、非同期 I/O、AI/ML サービスの接続
Django	管理画面、ORM、認証などが一体で揃う	管理機能の多い Web アプリ、業務システム、MVP

Go 系

選択肢	強み	向いている場面
標準ライブラリ (net/http)	依存を最小化しやすい	シンプルな API、長期保守
Echo / Gin	ルーティングやミドルウェアを手早く整えられる	標準ライブラリだけでは面倒な箇所が増えたとき

その他

フレームワーク	言語	位置づけ
Ruby on Rails	Ruby	規約が強く、試作と管理画面を速く作りやすい
Spring Boot	Java / Kotlin	大規模な企業システムで定番
Laravel	PHP	現代的な PHP 開発の中心選択肢

選定の判断軸

```
graph TD
A[バックエンドFWを選ぶ] --> B{チームの主力言語は?}
B -->|TypeScript| C{規模は?}
C -->|小〜中規模 / エッジ| D[Hono]
C -->|中〜大規模| E[NestJS or Fastify]
B -->|Python| F{用途は?}
F -->|API中心| G[FastAPI]
F -->|管理画面が必要| H[Django]
B -->|Go| I[標準ライブラリ or Echo]
B -->|未定| J{プロトタイプ重視?}
J -->|はい| K[Rails or Django]
J -->|いいえ| L[Hono or FastAPI]
```

実務での注意点

> WARNING
> ランタイムの速度比較だけで選ばないこと。 実際のボトルネックは、データベース、外部API、キュー、キャッシュ、デプロイ構成にあることが多い。

> IMPORTANT
> フレームワークより先に、どこで動かすか を決める方が重要な場合がある。 たとえば、Cloudflare Workersのようなエッジ前提なら Hono が自然だが、 長時間ジョブや重い SDKを多用するなら通常のサーバー環境の方が扱いやすい。

まとめ

TypeScript なら Hono / Fastify / NestJS、Python なら FastAPI / Django、

Go なら 標準ライブラリから必要に応じて補う という整理がわかりやすい。

結局は、性能の数値より チームが運用し続けられるか で選ぶべきだ。

4.3 API設計パラダイム

このセクションで答える問い: REST, GraphQL, gRPC…APIの設計方法はどう選ぶ？

なぜ必要か

API（Application Programming Interface）は、

フロントエンドとバックエンドの間の「会話のルール」だ。

レストランの注文システムに例えるなら、「メニュー番号で注文する」のか

「好きな食材を伝えてシェフにお任せする」のか、という違いだ。

ルールが明確でないと、注文が通らない（バグ）、思った料理が来ない（仕様の齟齬）が起きる。

全体像

パラダイム	仕組み	向いている場面	学習コスト
REST	URL + HTTPメソッドでリソースを操作	汎用的。ほとんどのWebアプリ	低
GraphQL	クライアントが欲しいデータの形を指定	複雑なデータ構造。モバイルでの通信量削減	中
gRPC	バイナリプロトコル。スキーマ定義必須	マイクロサービス間通信。高パフォーマンス	高
tRPC	TypeScriptの型をクライアント-サーバー間で共有	TypeScriptフルスタック。型安全なAPI	低（TS経験者）

選定の判断軸

> TIP
>
> 迷ったら REST から始める。世界で最も広く使われており、 ほぼすべての HTTP クライアントが対応している。
> TypeScript フルスタックなら tRPC が魅力的だが、TypeScript 境界の内側で閉じる設計に向いている。

実務での注意点

> WARNING
>
> GraphQL は「クライアントが自由にクエリを組める」反面、
> 不注意なクエリでサーバーに過大な負荷をかけるリスクがある。 導入する場合は、N+1
> 対策、深さ制限、クエリコスト制御を考える必要がある。

> IMPORTANT
>
> gRPC は内部サービス間通信では強いが、ブラウザから直接使う前提では設計が少し重い。 公開 API か内部 API
> かで、向いている方式は変わる。

まとめ

ほとんどのプロジェクトは REST で十分。TypeScript統一なら tRPC が魅力的。

GraphQL は「複数クライアントが異なるデータ構造を必要とする」場合に検討する。

4.4 LLM API統合パターン（AI時代の新しい責務）

> このセクションで答える問い:
> バックエンドがLLMをオーケストレーターとして使う場合、何に気をつけるべきか？

なぜ必要か

LLM を使う機能では、フロントエンドから直接モデル API を叩くより、

バックエンドで呼び出しを中継・制御する方が安全なことが多い。

理由は、API キー保護、入力検証、コスト制御、監査ログ、ツール実行の制御が必要になるからだ。

バックエンドは、単なるプロキシではなく 信頼境界 として振る舞う必要がある。

全体像

パターン	何をするか	向いている場面
同期レスポンス	入力を受けて 1 回の応答を返す	要約、分類、短い補助回答
ストリーミング	生成途中を段階的に返す	チャット、長文生成、進捗体験が重要な UI
ツール呼び出し	モデルが必要な操作を提案し、バックエンドが許可して実行する	予約、検索、社内 API 利用、エージェント
構造化出力	JSON スキーマなどで出力形を制約する	抽出、分類、ワークフロー連携
RAG	検索結果や文書断片を添えて生成する	ナレッジ検索、社内文書 QA

典型的な責務分割

```
クライアント
↓
[バックエンド]
├── 入力検証
├── プロンプト / メッセージ組み立て
├── 権限制御
├── モデル呼び出し
├── コスト・タイムアウト管理
├── ログ / 監査
└── ツール実行の許可判定
↓
[モデル API]
```

実装イメージ

1. ストリーミング

```
app.post('/chat', async (c) => {
const { text } = await c.req.json();

return c.streamText(async (stream) => {
const result = await llm.stream({
messages: [{ role: 'user', content: text }],
});

for await (const chunk of result) {
await stream.write(chunk);
}
});
});
```

2. ツール呼び出し

```
const completion = await llm.generate({
messages,
tools: [fetchUserDataTool],
});

for (const call of completion.toolCalls) {
if (!isAllowed(call, currentUser)) {
throw new Error('許可されていないツール呼び出し');
}

const toolResult = await runTool(call);
messages.push({ role: 'tool', content: JSON.stringify(toolResult) });
}
```

3. 構造化出力

```
const result = await llm.generateObject({
schema: articleExtractionSchema,
prompt: '記事からタイトル、著者、要約を抽出する',
});
```

実務での注意点

> WARNING
> プロンプトインジェクション対策は必須
> ユーザー入力、外部文書、検索結果、ツール出力はすべて未信頼入力として扱う。
> モデルに「従ってよい指示」と「ただの参考情報」を混同させない設計が必要である。
> 少なくとも、入力分離、ツール権限の最小化、危険操作の人手承認を考えるべきだ。

> WARNING
> コストと待ち時間を設計に含める モデル呼び出しは、通常の DB クエリより高価で遅く、失敗形も多い。
> タイムアウト、レート制限、上限金額、再試行条件、キャッシュ方針を先に決めること。

> IMPORTANT
> プロンプトテンプレートやツール定義は、アプリケーションコードと同じく バージョン管理とレビュー対象にする。

まとめ

LLM 統合では、ストリーミング、ツール呼び出し、構造化出力 が基本パターンになる。

もっとも重要なのは、モデルの賢さではなく 境界制御、コスト制御、監査可能性 を

バックエンド側で担保することだ。

4.5 認証・認可の設計

> このセクションで答える問い: 「ログイン」の裏側では何が起きているのか？

なぜ必要か

認証（Authentication）は「あなたは誰か」を確認すること。

認可（Authorization）は「あなたは何ができるか」を制御すること。

この2つはWebアプリケーションの安全性の根幹であり、設計を間違えると

個人情報の漏洩やなりすましといった深刻なセキュリティ事故につながる。

全体像

方式	仕組み	適した場面
セッション + Cookie	サーバー側でセッションを持ち、Cookie で識別する	Web アプリ、BFF、SSR 主体の構成
トークンベース認証	トークンを API で送る	API、モバイル、サービス間通信
OAuth 2.0 / OIDC	外部 ID プロバイダへ認証を委任する	ソーシャルログイン、企業 SSO
Passkeys / WebAuthn	デバイス鍵で認証する	パスワード依存を減らしたい新規ログイン体験

選定の判断軸

> IMPORTANT
>
> パスワード保存、パスワードリセット、MFA、アカウント回復、Passkeys まで含めて
> すべて自前実装するのは避けたい。
> まずは実績あるライブラリや認証サービスを優先し、独自実装は要件が強い場合に限る。

サービス	特徴	コスト
Clerk	開発者体験が良い。UIコンポーネント付き	無料枠あり。有料は従量制
Auth.js	OSS。Next.js との統合が強い	無料（セルフホスト）
Supabase Auth	Supabase（BaaS）の一部。PostgreSQL連携	無料枠あり
Firebase Auth	Google製。モバイル対応が強い	無料枠あり

まとめ

Web アプリでは セッション + Cookie が扱いやすいことが多い。

外部ログインは OIDC / OAuth 2.0、パスワードレスを強めたい場合は

Passkeys / WebAuthn を検討する。重要なのは、認証方式そのものより

回復フローまで含めて安全に設計すること である。

4.6 バックグラウンドジョブとキュー

このセクションで答える問い: 時間のかかる処理をどう扱うのか？

なぜ必要か

メールの一斉送信、大量データのCSVエクスポート、画像のリサイズ――

これらをAPIリクエストの中で実行すると、レスポンスが数十秒待ちになり、

ユーザー体験が著しく悪化する。タイムアウトでエラーになることもある。

バックグラウンドジョブは「後でやっておきます」という仕組みだ。

レストランで「持ち帰り用の箱を準備しておきますので、お食事を続けてください」

と言われるのに似ている。

全体像

ツール	言語/環境	特徴
BullMQ	Node.js	Redis ベース。ジョブ、リトライ、遅延実行を扱いやすい
Celery	Python	Python 系で広く使われるジョブキュー
SQS + Lambda	AWS	マネージドなキューと実行環境で組みやすい
Inngest	多言語	イベント起点でワークフローを記述しやすい

選定の判断軸

観点	まず考えること
再実行	同じジョブが複数回動いても問題ないか
失敗処理	何回まで再試行するか、失敗後にどこへ送るか
可観測性	いま何件詰まっているか、どこで失敗したか見えるか
実行時間	数秒か、数分か、長時間か

> WARNING
>
> ジョブキューを入れると「いつか実行される」世界になる。 API
> 直列実行の感覚のまま書くと、重複実行や順序ずれで事故が起きる。 冪等性、再試行、デッドレターキュー
> を設計に含めること。

まとめ

時間のかかる処理はジョブに逃がす。

ただし、キューを入れれば終わりではなく、冪等性と失敗時の扱い まで作って初めて実務で使える。

4.7 リアルタイム通信

このセクションで答える問い: チャットや通知のような「リアルタイム」はどう実現するのか？

なぜ必要か

通常のHTTPは「クライアントが聞いたときだけサーバーが答える」一方通行モデルだ。

しかし、チャットアプリ、株価のリアルタイム表示、共同編集機能では、

サーバー側からクライアントに「変化があったよ」と能動的に通知する必要がある。

全体像

技術	仕組み	向いている場面
WebSocket	双方向の常時接続	チャット、ゲーム、共同編集
SSE（Server-Sent Events）	サーバー→クライアントの一方向ストリーム	通知、ログのリアルタイム表示
ポーリング（Polling）	定期的にサーバーに問い合わせ	更新頻度が低い場面（数分間隔）
WebTransport	HTTP/3 ベースの双方向通信	より低遅延な通信が必要で、対応状況を確認できる場面

選定の判断軸

> TIP
>
> 「とりあえずWebSocket」ではなく、まず SSE で十分か を検討する。
> 通知やフィードの更新だけならSSEの方がシンプルで、HTTPの仕組みに乗るため扱いやすい。
> 双方向通信が本当に必要な場合（チャット、ゲーム）のみ WebSocket を使う。

> IMPORTANT
>
> WebTransport は新しい選択肢だが、対応ブラウザ、プロキシ、サーバー構成まで含めて確認が必要。 まずは SSE または WebSocket で足りるかを考える方が現実的である。

まとめ

サーバーからの通知は SSE、双方向通信は WebSocket が基本。

更新頻度が低ければ単純なポーリングで十分なことも多い。「最もシンプルな手段」を選ぼう。

4.8 AI機能を持つバックエンド設計（2026年）

このセクションで答える問い: AI機能を持つバックエンドでは、何が追加で重要になるのか？

なぜ必要か

AI 機能を持たないバックエンドなら、この節の多くは不要である。

しかし、モデル呼び出し、RAG、ツール実行、エージェント実行を含むと、

通常の API 設計だけでは足りなくなる。

問題は「モデルが賢いか」ではなく、未信頼入力をどう扱うか、

どの操作を許可するか、失敗や暴走をどう止めるか にある。

全体像

観点	通常の API	AI 機能を持つ API で増える論点
入力	フォーム値、JSON	ユーザー入力、文書断片、検索結果、ツール出力も未信頼入力
実行時間	短い同期処理	長い生成、非同期ジョブ、エージェント実行
外部依存	決済、メール、SaaS API	モデル API、ベクタ検索、ツール実行
コスト	ほぼ固定	トークン量や実行回数で変動
監査	リクエストログ	モデル名、入力種別、ツール呼び出し、コスト、結果の追跡

追加で設計すべきこと

1. 入力境界の明確化

ユーザー入力とシステム指示を分離する。

2. ツール権限の最小化

何をモデルに実行させてよいかを明示する。

3. 長時間処理への移行

数秒で返らない処理はジョブ化する。

4. 観測性の追加

モデル呼び出し単位でログとコストを取る。

実務での注意点

> WARNING
>
> 「プロンプト = ただの文章」と考えない 実務では、プロンプト、ツール定義、ガードレールはコードに近い。
> バージョン管理、レビュー、変更履歴の把握が必要である。

> WARNING
>
> AIエージェント時代の「複数エージェント問題」 1つのシステムに複数のAIエージェントが同時に動作する場合、
> データベース競合、リソース枯渇、無限ループの検出が従来とは比較にならないほど複雑に。 例:
> エージェントAが「ユーザー削除」を指示 → エージェントBが「ユーザー情報取得」を試みる → 矛盾 対策: -
> エージェント間の通信ログを完全記録 - トランザクション一貫性の強化（楽観的ロックではなく悲観的ロック検討） -
> エージェント実行の「チェックポイント」設計（途中で停止・検証・再開可能に）

まとめ

AI 機能を持つバックエンドでは、通常の API 設計に加えて

入力境界、権限制御、長時間処理、観測性 を強く意識する必要がある。

すべてのバックエンドが AI 前提になるわけではないが、

AI を扱うなら通常の延長線ではなく、別の信頼境界として設計すべきである。

この章のチェックリスト

- [ ] バックエンドの6つの責務を列挙できるか
- [ ] Node.js / Bun / Deno を、数値ではなく用途で比較できるか
- [ ] Hono がエッジランタイム対応である意味を説明できるか
- [ ] REST, GraphQL, tRPC の違いと使い分けを説明できるか
- [ ] LLM API 統合における「ストリーミング」「ツール呼び出し」「構造化出力」の違いを説明できるか
- [ ] 「プロンプトインジェクション対策」が 2026年のバックエンド設計に必須である理由を言えるか
- [ ] 認証と認可の違いを説明できるか
- [ ] セッション認証とトークンベース認証の使い分けを説明できるか
- [ ] バックグラウンドジョブが必要な場面を3つ挙げられるか
- [ ] AI機能を持つバックエンドで追加設計すべき論点を説明できるか

第5章 データベースとデータ管理

> この章を読む前に: バックエンドの役割（第4章4.1節）を理解していることが前提。
> SQLの文法を知らなくても読めるように書いているが、概念は登場する。 2026年注記: 本章はAI機能を持つアプリも視野に入れているが、すべてのシステムでベクトル検索が必須という意味ではない。

アプリケーションは「データ」で成り立っている。ユーザーの情報、投稿、注文、設定……

これらのデータを安全に保存し、必要なときに素早く取り出す仕組みがデータベースだ。

図書館に例えるなら、本（データ）をどう分類し、どの棚に並べ、

どうやって探し出すかを決める仕組みがデータベースの設計にあたる。

5.1 データベースの種類と使い分け

> このセクションで答える問い: RDB, NoSQL, NewSQL…何が違い、いつどれを使うのか？

なぜ必要か

データベースは1種類ではない。「どんなデータを、どのように使うか」によって

最適なデータベースは変わる。間違った選定をすると、

アプリが成長した後にパフォーマンス問題や設計の限界にぶつかる。

データベースの移行は最もコストが高い技術的変更の1つだ。

全体像

RDBMS（リレーショナルデータベース）

データを「表（テーブル）」の形で整理し、表と表の「関係（リレーション）」でデータを結びつける。

Excelの表をイメージするとわかりやすい。

DB	特徴	向いている場面
PostgreSQL	機能が広く、JSON、全文検索、拡張機能も使いやすい	新規開発、複雑なクエリ、拡張性を重視する場面
MySQL	実績が長く、既存資産も多い	既存システムとの互換、MySQL前提の運用基盤がある場面

> TIP
> 新規の一般的な Web アプリなら、PostgreSQL は有力な出発点である。 ただし、既存システムや運用知識が MySQL に寄っているなら、MySQL も十分に現実的な選択肢だ。

NoSQL

テーブル構造に縛られない柔軟なデータ格納方式。用途によって種類が異なる。

DB	種類	得意なこと
Redis	Key-Value	超高速な一時データ保存。キャッシュ、セッション管理
MongoDB	ドキュメント	スキーマが流動的なデータ。プロトタイピング
DynamoDB	Key-Value / ドキュメント	AWS と強く統合した大規模ワークロード

マネージドDB / クラウドDB

RDBMS をそのまま運用するのではなく、クラウドサービスとして使う選択肢もある。

DB	特徴	向いている場面
Neon	PostgreSQL をサーバーレスに近い形で使いやすい	開発環境の分岐やスケールの柔軟性を重視するとき
PlanetScale	MySQL 系の運用をマネージドで進めたい	MySQL 系の運用負荷を下げたいとき
Supabase	PostgreSQL を中心に認証やストレージもまとめて扱える	BaaS 寄りに素早く組み立てたいとき
Turso	SQLite ベースで軽量に配りやすい	エッジや軽量アプリ、ローカルファースト寄りの構成

選定の判断軸

```
graph TD
A[データベースを選ぶ] --> B{データの構造は明確?}
B -->|はい| C[RDBMS: PostgreSQL]
B -->|流動的/未定| D{規模は?}
D -->|小〜中| E[PostgreSQL のJSON型で対応]
D -->|超大規模| F[MongoDB or DynamoDB]
C --> G{クラウドネイティブが必要?}
G -->|はい| H[Neon or Supabase]
G -->|いいえ| I[PostgreSQL セルフホスト]
```

実務での注意点

> WARNING
> 「MongoDBはスキーマレスだから楽」という理由だけで選ぶと後悔する。
> スキーマレスは「スキーマが不要」ではなく「スキーマの管理がアプリ側の責任になる」ことを意味する。
> 多くのWebアプリでは、最初から PostgreSQL を選ぶ方が結果的に楽だ。

まとめ

一般的な Web アプリでは RDBMS を先に選ぶ のが自然で、その起点として PostgreSQL は強い。

ただし、DB 選定は一発で完結しない。キャッシュ、検索、分析、必要に応じてベクトル検索を

後から組み合わせる前提で考える方が実務的だ。

5.2 ORM・クエリビルダの選定

このセクションで答える問い: SQLを直接書くべきか？それとも抽象化ツールを使うべきか？

なぜ必要か

アプリケーションからデータベースを操作するには、通常SQLという言語を使う。

しかしSQLを文字列としてコードに直接書くと、型安全性がなく、

SQLインジェクション（悪意あるSQLを注入される攻撃）のリスクがある。

ORM（Object-Relational Mapping）やクエリビルダは、

プログラミング言語のコードからSQLを安全に生成する仕組みだ。

全体像

ツール	言語	種別	特徴
Prisma	TypeScript	ORM	スキーマから型生成しやすい。入門しやすい
Drizzle	TypeScript	クエリビルダ寄り ORM	SQL に近い感覚で扱いやすい
SQLAlchemy	Python	ORM + クエリビルダ	柔軟で長く使われている
GORM	Go	ORM	Go での生産性を上げやすい
生 SQL	どの言語でも可	SQL	複雑な集計やチューニングで必要になる

選定の判断軸

TypeScript では、開発速度と型生成を重視するなら Prisma、

SQL に近い制御を保ちたいなら Drizzle がわかりやすい。

Python では SQLAlchemy が定番で、Go では ORM を使うか生 SQL を使うかを

チーム文化で選ぶことが多い。

IMPORTANT

ORM を入れても SQL の理解は不要にならない。インデックス、JOIN、実行計画、N+1 問題は、ORM の上でもそのまま発生する。

WARNING

「ORM を使えば SQL インジェクションが完全に防げる」とは言えない。文字列結合で生 SQL を組み立てれば危険は残る。パラメータ化クエリ を徹底すること。

まとめ

ORM やクエリビルダは、生産性と型安全性を上げるための道具である。

ただし、複雑な問い合わせでは生 SQL を使うのも普通 であり、

重要なのは ORM を使うことではなく、安全にクエリを書くことだ。

5.3 スキーマ設計の基本原則

このセクションで答える問い: データベースの「表の設計」はどう考えるのか？

なぜ必要か

スキーマ設計はアプリケーションの土台だ。後から変更するのが最も困難な部分でもある。

建物の基礎工事に相当するため、最初にしっかり考える価値がある。

全体像

RDBMSのスキーマ設計で守るべき基本原則:

原則	意味	例
正規化	データの重複を排除する	ユーザー名を注文テーブルに直接書かず、ユーザーIDで参照する
適切な型選択	データに合った型を使う	日時は文字列ではなく TIMESTAMP 型を使う
外部キー制約	テーブル間の関係を明示する	注文テーブルのユーザーIDが、ユーザーテーブルに存在することを保証
インデックス	よく検索するカラムに索引を張る	図書館のカード目録のようなもの

実務での注意点

> WARNING
>
> 過度な正規化は逆効果。テーブルが増えすぎると JOIN が複雑になり、
> パフォーマンスが悪化する。実務では「ある程度の非正規化」が必要な場面もある。

まとめ

正規化を基本としつつ、パフォーマンスとのバランスで非正規化も検討する。

インデックスは「検索条件に使うカラム」に張る。設計に迷ったらシンプルに始めて後で調整する。

5.4 マイグレーション戦略

このセクションで答える問い: データベースの構造を安全に変更するにはどうするのか？

なぜ必要か

アプリが進化すれば、テーブルの追加やカラムの変更が必要になる。

しかしデータベースにはすでにユーザーのデータが入っている。

「テーブルを作り直す」わけにはいかない。

マイグレーション（Migration）は、データを保持したままスキーマを段階的に変更する仕組みだ。

Git がコードの変更履歴を管理するように、マイグレーションはDBスキーマの変更履歴を管理する。

全体像

ツール	言語	特徴
Prisma Migrate	TypeScript	Prisma スキーマからマイグレーションファイルを自動生成
Drizzle Kit	TypeScript	Drizzle スキーマとDBの差分を検出
Alembic	Python	SQLAlchemy と連携。Python 標準
golang-migrate	Go	SQL ファイルベース。シンプル

実務での注意点

> WARNING
>
> 本番で破壊的変更を一度に入れないこと。 列追加 → アプリ側対応 → 古い列削除、のように Expand / Migrate / Contract の段階で進める方が安全である。

まとめ

マイグレーションは「DBスキーマのGit」である。

変更を履歴として残し、段階的に本番へ適用できる形にすることが重要だ。

5.5 キャッシュ戦略

> このセクションで答える問い: なぜ毎回データベースに問い合わせないのか？

なぜ必要か

データベースへの問い合わせには時間がかかる。頻繁にアクセスされるデータを

毎回DBから取得するのは、辞書を引くたびに図書館まで行くようなものだ。

よく使う言葉はメモしておけば早い——これがキャッシュの発想だ。

全体像

レイヤー	仕組み	ツール
CDN	静的ファイルを世界各地のサーバーに分散配置	CloudFlare, Fastly, CloudFront
アプリケーションキャッシュ	アプリ内で計算結果やDB結果をメモリに保持	Redis, Memcached
ブラウザキャッシュ	ブラウザがレスポンスを保存して再利用	HTTP Cache-Control ヘッダ

実務での注意点

> WARNING
> キャッシュの最大の問題は「古いデータが返される」こと。
> 「キャッシュの無効化（invalidation）」はコンピュータサイエンスで 最も難しい問題の1つと言われている。
> キャッシュする前に「古いデータが返されても許容できるか」を必ず確認しよう。

まとめ

キャッシュは「速度と鮮度のトレードオフ」。Redis によるアプリケーションキャッシュと、

CDN による静的ファイルの配信高速化が一般的な組み合わせだ。

5.6 検索エンジン

このセクションで答える問い: データベースの LIKE 検索では足りないのはどんなとき？

なぜ必要か

ECサイトで「赤い ワンピース 夏」と検索したとき、商品名にこの文字列が完全一致する

商品だけ表示されても困る。「紅い」「ドレス」「サマー」も含めてほしいし、

人気順で並べてほしい。こうした「賢い検索」はデータベースの基本機能では難しい。

全体像

ツール	特徴	向いている場面
PostgreSQL 全文検索	DB組み込み。追加インフラ不要	簡易的な検索機能。日本語は工夫が必要
Meilisearch	高速。設定が簡単。タイポ耐性	小〜中規模の製品検索
Typesense	Meilisearch の競合。ホスト版あり	同上
Elasticsearch	最も高機能。分散検索エンジン	大規模。ログ分析との兼用

まとめ

まずは PostgreSQL の全文検索で十分か検討し、要件が超えたら Meilisearch を導入する。

Elasticsearch は大規模で高機能が必要な場合の選択肢。

5.7 ベクトル検索とベクトルデータベース

このセクションで答える問い: ベクトル検索とは何か？生成AIアプリケーション開発で何ができるのか？

なぜ必要か

キーワード検索は、文字列が近いものを探すのは得意だが、

意味が近いものを探すのは苦手だ。

生成AIアプリやナレッジ検索では、「単語が同じ」より「意味が近い」を探したいことがある。

例えば、ドキュメント検索で「太陽」と「恒星」は別の単語だが、意味的には関連している。

このような「意味的な近さ」を数値化して検索する仕組みがベクトル検索であり、

その専門基盤がベクトルデータベースだ。

全体像

ベクトル検索の流れ:

```
graph LR
A[ドキュメント/テキスト] -->|エンベディング| B[数値ベクトル<br/>768次元など]
B -->|ベクトルDB保存| C[ベクトルDB<br/>pgvector等]
D[ユーザークエリ] -->|エンベディング| E[クエリベクトル]
E -->|コサイン類似度計算| F[類似ベクトル検索]
F -->|上位K件取得| G[検索結果]
G -->|LLMへ提供| H[回答生成]
```

主な選択肢

DB	タイプ	強み	向いている場面
pgvector	PostgreSQL 拡張	既存 PostgreSQL と同居できる	ベクトル検索がアプリの一機能である場合
Pinecone	マネージド専門サービス	運用を委譲しやすい	専門サービスとして素早く始めたい場合
Qdrant	専門ベクトルDB	フィルタ付き検索や自前運用との相性がよい	メタデータ条件が重要な場合
Weaviate	ベクトル + ハイブリッド検索	キーワードとベクトルを併用しやすい	検索体験を重視する場合
Milvus	大規模専門型	大量データ向けの構成を取りやすい	ベクトル検索自体が中心業務になる場合

RAGアーキテクチャのデータフロー

```
graph TD
A[ドキュメント群<br/>PDF/Web/DB] -->|テキスト抽出| B[テキスト処理]
B -->|チャンク分割<br/>500-1000文字単位| C[チャンク]
C -->|エンベディングモデル| D[ベクトル変換<br/>OpenAI等]
D -->|保存| E[ベクトルDB<br/>pgvector/Pinecone等]

F[ユーザー質問] -->|エンベディング| G[クエリベクトル]
G -->|ベクトル検索| E
E -->|類似チャンク取得<br/>上位K件| H[検索結果]
H -->|コンテキスト構築| I["プロンプト<br/>(システム+検索結果+質問)"]
I -->|LLM推論| J[回答生成<br/>ChatGPT等]
J -->|ユーザーへ返却| K[最終回答]
```

選定の判断軸

PostgreSQL + pgvector を選ぶ場合

- 既存 PostgreSQL を活用したい

- トランザクションや通常データと一緒に扱いたい

- まずは 1 つの DB に寄せて単純に運用したい

専門ベクトルDBを選ぶ場合

- ベクトル検索がプロダクトの中核である

- メタデータ条件や検索の柔軟性が重要

- 通常のトランザクション DB と分けて独立運用したい

実務での注意点

> WARNING
>
> DB の種類より埋め込みと評価の方が効くことが多い
> ベクトルDBのブランドより、エンベディングモデル、チャンク分割、再ランキング、
> 評価方法の方が検索品質へ強く効くことが多い。

> WARNING
>
> チャンク分割戦略はRAG成功の鍵 テキストを固定長で分割する単純な方法では、文脈を失う。
> まずは単純な分割で始めつつ、必要なら意味段落ベースの分割も検討する。

まとめ

ベクトル検索は、AI 機能を持つアプリでは有力な選択肢になる。

ただし、最初から専門ベクトルDBを入れる必要はない。

既存 PostgreSQL があるなら pgvector から始め、検索品質や運用要件が伸びたら分離を考える方が現実的だ。

5.8 AI時代のデータベース設計

このセクションで答える問い: なぜ2026年のデータベース選定はかつてと異なるのか？

なぜ必要か

以前のデータベース選定は「RDBMS か NoSQL か」で語られがちだった。

しかし現在は、1つの DB ですべてを賄うより、ワークロードごとに

役割を分ける設計が一般的になっている。

全体像

役割	代表例	何を置くか
トランザクションDB	PostgreSQL, MySQL	ユーザー、注文、設定などの正規データ
キャッシュ	Redis	セッション、一時結果、レート制限
検索	PostgreSQL 全文検索, Meilisearch, Elasticsearch	キーワード検索、絞り込み
ベクトル検索	pgvector, Pinecone, Qdrant	類似文書、RAG 用の埋め込み
分析基盤	DWH やログ基盤	集計、BI、履歴分析

選定の判断軸

```
graph TD
A[プロジェクト開始] --> B{通常の業務データを保存する?}
B -->|はい| C[まずRDBMS]
C --> D{検索が必要?}
D -->|はい| E[全文検索 or 検索エンジン追加]
D -->|いいえ| F{AI検索が必要?}
E --> F
F -->|はい| G[pgvector か専門ベクトルDBを検討]
F -->|いいえ| H[キャッシュだけ追加]
```

> IMPORTANT
>
> AI 機能があるからといって、必ず専門ベクトルDBが必要とは限らない。 多くのケースでは、RDBMS + キャッシュを先に整え、 その後に検索やベクトル検索を追加する方が安全である。

実務での注意点

> WARNING
>
> ポリグロット化は強力だが、構成が増えるほど運用負荷も増える。
> 「分けられるから分ける」のではなく、障害分離、性能、チーム体制の観点で必要性を確認すること。

まとめ

2026年のデータベース設計は、単一製品選定というより 役割分担の設計 に近い。

まずは RDBMS を軸に据え、必要になったときにキャッシュ、検索、ベクトル検索を

追加していく考え方が崩れにくい。

この章のチェックリスト

- [ ] RDBMS と NoSQL の違いを説明でき、使い分けの基準を言えるか
- [ ] PostgreSQL を第一候補とする理由を3つ挙げられるか
- [ ] ORM やクエリビルダを使う理由と、生 SQL が必要な場面を説明できるか
- [ ] マイグレーションとは何か、なぜ必要かを説明できるか
- [ ] キャッシュの「メリットとリスク」を両方説明できるか
- [ ] ベクトル検索とは何か、RAGアーキテクチャで何ができるかを説明できるか
- [ ] pgvector と Pinecone を使い分ける基準を言えるか
- [ ] 2026年のデータベース設計がなぜ「ポリグロット」へシフトしているのか説明できるか

第6章 モバイル開発

> この章を読む前に: フロントエンド開発の基本概念（第3章）を把握していると理解しやすい。
> ただし、モバイル開発の経験がなくても読める構成にしている。

モバイル開発で最初に決めるべきことは、言語やフレームワークではない。

まず決めるべきなのは、どの配布経路を使うか、どこまで端末機能を使うか、既存チームの強みをどこに置くか、の3点だ。

Web サービスの延長として始めるなら PWA が十分なこともある。

一方で、通知、課金、カメラ、バックグラウンド処理、ストア配信、端末固有の操作感まで求めるなら

ネイティブアプリかクロスプラットフォームアプリを前提に考えた方がよい。

6.1 ネイティブ vs クロスプラットフォームの判断基準

> このセクションで答える問い: iOS と Android を別々に作るべきか、1つのアプローチで両方を扱うべきか？

なぜ最初にここを決めるのか

モバイル開発は、技術選定を誤ると後から修正しにくい。

特に次の3つは初期判断の影響が大きい。

1. 配布経路
2. ハードウェア機能への依存度
3. チームの既存スキル

アプリストア配信が不要で、通知やオフラインも限定的なら PWA で十分な場合がある。

逆に、ストア課金、重いアニメーション、AR、Bluetooth、バックグラウンド実行などが重要なら、

最初からネイティブまたはネイティブ寄りの構成を選んだ方が手戻りが少ない。

全体像

アプローチ	共有範囲	強み	注意点	向いている場面
ネイティブ	ほぼ共有しない	各OSの機能とUXを最大限に使える	iOS/Android を別々に育てる必要がある	カメラ、AR、決済、音声、端末統合が重要
クロスプラットフォーム	UI とロジックの大部分を共有できる	少人数でも両OSを同時に出しやすい	端末固有要件が増えるほど調整が必要	MVP、B2B、業務アプリ、標準的な消費者向けアプリ

アプローチ	共有範囲	強み	注意点	向いている場面
PWA	Web とほぼ同じ	配布と更新が速い	ブラウザ差分と端末API制約がある	社内ツール、予約、承認、閲覧、軽い入力中心

判断フロー

```
graph TD
A[モバイル対応が必要] --> B{ストア配信が必要か}
B -->|不要| C{通知や端末機能は限定的か}
C -->|はい| D[PWA を優先検討]
C -->|いいえ| E[ネイティブアプリを検討]
B -->|必要| F{既存チームは何に強いか}
F -->|TypeScript/React| G[Expo / React Native から検討]
F -->|モバイル専任あり| H[ネイティブ or KMP を検討]
F -->|デザイン制御重視| I[Flutter を検討]
H --> J{端末固有機能が多いか}
J -->|はい| K[ネイティブ中心]
J -->|いいえ| L[KMP や共有構成も有力]
```

> TIP
>
> 小規模チームは、まず 配布経路 と 既存スキル で絞ると判断が速い。
> そのうえで端末固有要件が強ければネイティブへ寄せる、という順番にすると失敗しにくい。

ビジネス観点の比較

観点	ネイティブ	クロスプラットフォーム	PWA
初速	遅め	速い	最速
UX の作り込み	強い	十分高いが方式依存	ブラウザ制約あり
保守の単純さ	低い	中〜高	高い
ストア審査依存	高い	高い	低い
既存Web資産の再利用	低い	中	高い
端末機能への深いアクセス	強い	中〜高	低〜中

まとめ

正解は1つではない。

ただし、次の整理はかなり安定している。

- PWA: 配布速度とWeb再利用を最優先するとき

- クロスプラットフォーム: 少人数で iOS / Android を同時に育てたいとき

- ネイティブ: UX と端末統合が競争力そのものになるとき

6.2 クロスプラットフォームフレームワーク

このセクションで答える問い: Expo / React Native、Flutter、Kotlin Multiplatform はどう違うのか？

全体像

選択肢	主な共有範囲	向いているチーム	強み	注意点
Expo / React Native	UI とロジック	TypeScript / React チーム	参入障壁が低い。Webチームと連携しやすい	ネイティブ依存が増えると設計整理が必要
Flutter	UI とロジック	モバイル専任または新規学習を許容できるチーム	見た目を揃えやすく、描画制御がしやすい	Dart 学習が必要。各OSらしさは自分で作る場面がある
Kotlin Multiplatform	主にビジネスロジック	Android / バックエンド Kotlin チーム	ドメインロジック共有とネイティブUIを両立しやすい	UI 共有を前提にしすぎると難しくなる

Expo / React Native

Expo は「ネイティブ機能が使えない簡易版」という理解ではもう不十分だ。

現在は prebuild、config plugin、Expo Modules により、多くのネイティブ連携を段階的に扱える。

そのため、React / TypeScript チームがモバイル参入するなら、

まず Expo から始める判断はかなり合理的だ。

ただし、次の要件が強くなるほど、早めにネイティブ境界を明示した方がよい。

- 独自ネイティブモジュール

- 低レベルな Bluetooth / NFC / センサー処理

- ストア配布前提での細かなビルド差分管理

- React Native ライブラリの互換性検証

> NOTE
>
> Expo で始めて、必要になったらネイティブ側へ降りる という段階的な進め方は妥当だ。 ただし「いつでも無痛で降りられる」と考えるのは危険で、早い段階で依存ライブラリの対応状況を確認する必要がある。

Flutter

Flutter は UI を強く制御したい案件と相性がよい。

独自レンダリングにより、OS差を吸収しつつ一貫した見た目を作りやすい。

向いている場面は次の通り。

- デザイン再現性を重視する

- Android と iOS で見た目を揃えたい

- アニメーションやカスタムUIが多い

- Web チーム資産よりモバイル体験を優先する

一方で、各OSらしい細部を完全に自然に見せたい場合は、

標準コンポーネントの雰囲気やアクセシビリティ動作まで含めて確認が必要になる。

Kotlin Multiplatform

Kotlin Multiplatform は、UI を完全共有する技術というより、

ビジネスロジック共有の技術 と理解した方が実務では安全だ。

特に次の共有に向いている。

- API クライアント

- 認証や権限のルール

- バリデーション

- キャッシュ戦略

- ドメインロジック

- ViewModel 相当の状態管理

そのうえで UI は SwiftUI と Jetpack Compose で個別に作る。

この形が最も手堅い。

性能を見るときの観点

フレームワーク比較で、雑なベンチマーク表だけを見て決めるのは危険だ。

実務で見るべきなのは次の観点である。

観点	何を見るか
初期起動	認証、設定読込、初回描画までの体感
長いリスト	スクロール、画像読込、再描画の安定性
端末機能	カメラ、音声、位置情報、Bluetooth の統合難度
アプリサイズ	SDK、モデル、画像資産を含めた配布サイズ
開発速度	ホットリロードだけでなく、ネイティブ障害時の調査速度

まとめ

迷ったら次の整理で十分に実用的だ。

- Expo / React Native: Web チームが最短で参入したい

- Flutter: UI 制御を強く持ちたい

- KMP: ネイティブUIを維持しながらロジック共有したい

6.3 ネイティブ開発

> このセクションで答える問い: ネイティブ開発を選ぶなら何を使うのか？

全体像

プラットフォーム	中心言語	中心UI	補助的に知るべきもの	実務上の要点
iOS	Swift	SwiftUI	UIKit, App Store Connect, Xcode	macOS が必要。SwiftUI と UIKit の相互運用を理解する
Android	Kotlin	Jetpack Compose	View 系UI, Play Console, Gradle	Compose 中心だが既存 View 資産との共存も多い

2026年時点の実務ポイント

iOS

SwiftUI は中心的なUIフレームワークになっているが、

既存資産や一部の高度な部品では UIKit の理解がまだ必要だ。

また、最新の Swift 系列では並行性チェックが強化されており、

古いコードをそのまま移植すると警告や修正が増えることがある。

Android

Jetpack Compose が新規開発の中心だ。

ただし Android は端末差分、権限、バックグラウンド制約、ライフサイクルの癖が強いため、

UI だけでなくプラットフォーム知識そのものが依然として重要である。

ネイティブ開発環境の比較

項目	iOS	Android
開発マシン	macOS 必須	Windows / macOS / Linux で可能
IDE	Xcode	Android Studio
プレビュー	SwiftUI Preview	Compose Preview
テスト	XCTest, XCUITest	JUnit, instrumented test, Compose test

項目	iOS	Android
配布前確認	TestFlight	Internal / Closed testing

まとめ

ネイティブ開発は、フレームワーク比較以前に OSそのものを学ぶ 必要がある。

その代わり、ストア審査、アクセシビリティ、通知、端末統合まで含めて最も素直に作れる。

6.3.1 Kotlin Multiplatform（KMP）の詳細

> このセクションで答える問い: Kotlin Multiplatform は本番投入に耐えるのか？

結論

KMP は本番投入できる。

ただし、成功しやすいのは 共有ロジック中心 の使い方だ。

逆に、最初からすべてを共有しようとすると難度が上がる。

特に Swift export は公式ドキュメントでも制約付きで、実験的な要素が残る。

したがって、iOS 側との境界設計は慎重に行う必要がある。

典型構成

```
graph TD
A[KMP プロジェクト] --> B[shared]
B --> C[API / 認証 / キャッシュ]
B --> D[バリデーション / ドメインロジック]
B --> E[状態管理]

A --> F[iOS App]
A --> G[Android App]

F --> H[SwiftUI]
G --> I[Jetpack Compose]
```

KMP が向いている場面

向いている	理由
iOS / Android を両方出す前提	共有するほど効果が出る
認証や業務ルールが複雑	ロジック重複を減らしやすい
Kotlin に強いチームがいる	学習コストが抑えやすい
UI は各OSで自然に作りたい	ネイティブUIと相性がよい

向いていない	理由
Webチームだけでモバイルを始める	React Native / Expo の方が初速が出やすい
UI 完全共有を強く求める	KMP 単体の主戦場ではない
iOS 側の外部SDK依存が重い	相互運用の検証コストが上がる

実務での注意点

> WARNING
>
> KMP で重要なのは「どこまで共有しないか」を先に決めることだ。 共有対象を広げすぎると、iOS 側の自然さやデバッグ効率を損ないやすい。

次の点は早めに確認する。

1. 使用したい iOS / Android SDK が共有層に置けるか
2. 例外処理と非同期境界をどう見せるか
3. 生成コードや相互運用のビルド時間を許容できるか
4. iOS 側チームが Kotlin 境界を受け入れられるか

推奨構成

```
my-app/
├── shared/
│   ├── src/commonMain/kotlin/
│   │   ├── data/
│   │   ├── domain/
│   │   ├── presentation/
│   │   └── util/
│   ├── src/androidMain/kotlin/
│   ├── src/iosMain/kotlin/
│   └── build.gradle.kts
├── iosApp/
│   └── SwiftUI 画面
└── androidApp/
    └── Compose 画面
```

まとめ

KMP は「Flutter の代替」ではなく、

ネイティブUIを維持したまま共有したい ときの有力な選択肢だ。

6.4 PWA という選択肢

このセクションで答える問い: アプリを作らずにモバイル対応する方法はあるか？

PWA で何ができるか

PWA は、Web アプリをアプリらしく配布・利用しやすくするための設計パターン群だ。

重要なのは「PWA は単一APIではない」という点である。

中核になる要素は次の通り。

要素	役割	実務メモ
Web App Manifest	名前、アイコン、起動方法を定義	インストール体験の土台
Service Worker	キャッシュ、オフライン、更新制御	オフラインや高速化で重要
Push / Notifications	再訪を促す通知	ブラウザとOSで制約差がある

> NOTE
>
> 近年は「インストール可能であること」と「Service Worker を使うこと」が必ずしも同義ではない。 ただし、実務で PWA と呼ぶものの多くは、結局 Service Worker を使っている。

PWA の強みと限界

項目	強み	限界
配布	URL で共有でき、更新が即時反映される	ストアの発見性は基本的に得にくい
開発	既存Web資産を使いやすい	モバイル専用UXは作り込みが必要
通知	再訪導線を作れる	iOS ではホーム画面追加済みなど条件がある
端末機能	カメラ、位置情報などは扱える	Bluetooth、NFC、バックグラウンド処理は差分が大きい
オフライン	Service Worker で実現できる	同期競合や更新戦略の設計が必要

PWA 実装の最小構成

```
{
"name": "My PWA App",
"short_name": "MyApp",
"start_url": "/",
"display": "standalone",
"theme_color": "#ffffff",
"background_color": "#ffffff",
"icons": [
{
"src": "/icon-192.png",
"sizes": "192x192",
"type": "image/png"
}
]
}
```

```
const CACHE_NAME = "app-v1";
const ASSETS = ["/", "/index.html", "/app.css", "/app.js"];

self.addEventListener("install", (event) => {
event.waitUntil(
caches.open(CACHE_NAME).then((cache) => cache.addAll(ASSETS))
);
});

self.addEventListener("fetch", (event) => {
event.respondWith(
caches.match(event.request).then((cached) => cached || fetch(event.request))
);
});
```

6.4.1 PWA と AI 時代

> このセクションで答える問い: AI 時代でも PWA は有力か？

有力な場面

AI 機能を持つアプリでも、次の条件なら PWA は十分有力だ。

- 主機能が 検索・要約・承認・入力補助 である

- サーバー側の AI を呼ぶ構成が中心である

- ストア申請より改善速度を優先したい

- 社内利用や限定配布が主である

つまり、AI = ネイティブ必須 ではない。

AI 機能の多くはネットワーク越しの推論と相性がよく、PWA でも成立する。

PWA では厳しい場面

一方で次はネイティブ寄りの方が安定しやすい。

- 重いカメラパイプライン

- 音声の常時待受

- 端末内モデルを使った継続的な推論

- バックグラウンド実行への強い依存

- ストア課金やOS連携通知を中心とした体験

まとめ

PWA は「ネイティブの劣化版」ではなく、

配布速度とWeb再利用を最大化する別解 だと捉える方が実務に合う。

6.5 モバイル向けオンデバイス AI

このセクションで答える問い: モバイルデバイス上で AI を実行する場合、何に注意すべきか？

なぜオンデバイスを検討するのか

オンデバイス推論の価値は、精度より先に 制約の回避 にある。

観点	オンデバイスが効く理由
レイテンシ	往復通信を減らせる
プライバシー	端末外へ出さない設計を取りやすい
オフライン	通信断でも最低限の体験を残せる
コスト	呼び出し回数に比例するAPI費用を抑えやすい

主な選択肢

プラットフォーム	主な手段	向いている処理
iOS	Core ML, Vision, Natural Language, SoundAnalysis	画像分類、検出、テキスト処理、音分析
Android	ML Kit, 端末向け推論ランタイム	OCR、翻訳、顔検出、軽量推論
共通化したい場合	ONNX Runtime Mobile など	同一モデルを両OSで配りたい場合

iOS: Core ML の基本形

```
import CoreML
import Vision

let model = try VNCoreMLModel(for: MobileNetV3().model)
let request = VNCoreMLRequest(model: model) { request, error in
guard let results = request.results as? [VNClassificationObservation] else {
return
}

for result in results.prefix(3) {
print(result.identifier, result.confidence)
}
}

let handler = VNImageRequestHandler(cgImage: image)
try handler.perform([request])
```

Android: ML Kit の基本形

```
import com.google.mlkit.vision.common.InputImage
import com.google.mlkit.vision.text.TextRecognition
import com.google.mlkit.vision.text.latin.TextRecognizerOptions

val recognizer = TextRecognition.getClient(TextRecognizerOptions.DEFAULT_OPTIONS)
val image = InputImage.fromBitmap(bitmap, 0)

recognizer.process(image)
.addOnSuccessListener { result ->
Log.d("OCR", result.text)
}
.addOnFailureListener { error ->
Log.e("OCR", "text recognition failed", error)
}
```

クラウド AI とどう使い分けるか

処理	推奨構成
分類、検出、簡易要約、候補提示	オンデバイス優先
長文生成、外部知識参照、複雑な推論	クラウド優先
遅延を抑えつつ複雑性も必要	ハイブリッド

```
graph TD
A[AI 処理が必要] --> B{100ms級の応答が必要か}
B -->|はい| C[オンデバイスを優先]
B -->|いいえ| D{外部知識や長文生成が必要か}
D -->|はい| E[クラウドAIを優先]
D -->|いいえ| F{機密データか}
F -->|はい| C
F -->|いいえ| G[ハイブリッド構成を検討]
```

モデル共通化の選択肢

ONNX Runtime Mobile のような選択肢を使うと、

iOS / Android の両方で同じモデルを扱いやすくなる。

ただし、最初の1本としては少し重い。

最初から共通化にこだわりすぎず、

まずは どの処理を端末上で動かす価値があるか を決める方が先である。

モデル最適化の考え方

手法	期待できる効果
量子化	サイズ削減、推論高速化
蒸留	精度を保ちながら小型化しやすい
入力解像度の見直し	速度と電池消費を下げやすい

手法	期待できる効果
端末ごとの機能判定	非対応端末での事故を減らせる

まとめ

オンデバイス AI は「何でも端末でやる」ための技術ではない。

速さ・プライバシー・オフライン が重要な処理だけを端末に置くのが基本である。

6.6 AI 時代のモバイル開発

> このセクションで答える問い: モバイル開発に AI は何を変えるのか？

まず押さえるべき前提

AI 機能は急速に一般化しているが、すべてのモバイルアプリに AI が必須 ではない。

必要なのは「AI を入れること」ではなく、「AI を入れたときに UX と運用が破綻しないこと」だ。

AI が入りやすい機能

パターン	例	向いている構成
入力補助	文章補完、要約、校正	クラウド中心、軽い候補生成は端末でも可
視覚補助	OCR、画像説明、物体検出	オンデバイス優先
個人化	並び替え、候補提示	ハイブリッド
業務支援	報告書下書き、FAQ、検索	クラウド中心
会話UI	チャット、音声対話	クラウド中心、前処理は端末側

AI 機能を入れるときの設計原則

1. 失敗時のUI を先に決める
2. データ境界 を明確にする
3. 評価方法 を用意する
4. コスト上限 を作る
5. 人手確認が必要な箇所 を残す

モバイル特有の注意点

モバイルでは、AI の正答率だけでなく次も重要になる。

- 電池消費

- 端末温度

- 通信量

- 権限説明

- 誤検出時のリカバリ導線

たとえば OCR や画像分類は精度が高くても、

カメラ起動から結果表示までが遅いとユーザー体験として失敗しやすい。

まとめ

AI をモバイルに入れるときは、

モデル選定 より UX と境界設計 の方が重要である。

6.7 アプリ配信プラットフォームと料金体系

> このセクションで答える問い: アプリを世に出すまでに何が必要か？

まず知るべきこと

モバイルアプリは、作ることより 配ること の方が厄介な場合がある。

特にストア配信では、技術だけでなく審査、申告、規約、支払い条件を満たさなければならない。

ストアの基本差分

2026年3月15日時点で、開発者アカウントまわりの基本情報は次の通り。

項目	App Store	Google Play
開発者登録	Apple Developer Program 年額 99 USD	Play Console 登録料 25 USD（一回）
企業内配布向け別制度	Apple Developer Enterprise Program 年額 299 USD	別概念。通常は Play 配布や社内配布手段を検討
審査	App Review がある	Play Console の審査とポリシー確認がある
ベータ配布	TestFlight	Internal / Closed / Open testing

> WARNING
>
> 手数料は アプリ種別、課金方式、地域、プログラム参加状況 で変わる。
> 「両ストアとも常に同じ料率」とは限らないので、公開直前に必ず公式料金ページを確認すること。

実務で詰まりやすい審査ポイント

App Store

Apple の公式ガイドラインを前提にすると、特に詰まりやすいのは次の点だ。

1. アカウント作成があるのに、アプリ内から削除導線がない
2. 第三者ログインを主認証に使うのに、Sign in with Apple 相当の要件を満たしていない
3. プライバシー説明やデータ利用申告が不十分
4. レビュー用のデモアカウントや再現手順を渡していない
5. 未完成機能やクラッシュが残っている

Google Play

Google Play では次の確認が重要だ。

1. Data safety の記載
2. 審査用に必要な ログイン情報 や デモ環境 の提供
3. 課金、広告、ユーザーデータ取扱いのポリシー適合
4. アカウント種別や公開経路に応じた追加要件
5. 対象年齢、コンテンツレーティング、権限宣言の整合性

CI/CD の選択肢

料金は変わりやすいので、ここでは役割だけを整理する。

手段	向いている場面
Fastlane	スクリプトで柔軟に自動化したい
Xcode Cloud	iOS 中心で Apple 純正運用を取りたい
GitHub Actions	既存のリポジトリ運用に載せたい
EAS Build	Expo / React Native を中心に回したい
Codemagic / Bitrise	モバイル向けCIを早く整えたい

配信時の実務パターン

```
graph TD
A[Git push] --> B[CI でビルドとテスト]
B --> C{成功したか}
C -->|いいえ| D[修正]
C -->|はい| E[内部配布]
E --> F[iOS は TestFlight]
E --> G[Android は Internal / Closed testing]
F --> H[レビュー]
G --> H
H --> I[段階的リリース]
```

費用を考えるときの内訳

見落としやすい費用は次の通り。

- 開発者アカウント費

- Mac ランナーや CI 分の計算資源

- クラッシュ監視、分析、通知系SaaS

- ストア手数料や課金事業者の手数料

- スクリーンショット、多言語説明文、審査対応の運用工数

まとめ

モバイル配信は、ビルドできる だけでは終わらない。

審査に通る情報設計 と 再現可能な配信パイプライン まで含めて完成である。

6.8 プロジェクト開始の判断表

> このセクションで答える問い: 新規モバイル案件をどこから始めるべきか？

条件	最初の候補
Web チーム中心で最短リリースしたい	Expo / React Native
UI の作り込みが競争力	Flutter か ネイティブ
iOS / Android の自然なUIを保ちつつロジック共有したい	KMP
ストア配信が不要で、Web資産を強く再利用したい	PWA
端末機能やOS統合が中心	ネイティブ

```
graph TD
A[新規モバイル案件] --> B{ストア配信が必要か}
B -->|不要| C[PWA を最初に検討]
B -->|必要| D{チームの中心は誰か}
D -->|Web チーム| E[Expo / React Native]
D -->|モバイル専任| F{端末機能が重いか}
F -->|はい| G[ネイティブ]
F -->|いいえ| H[KMP も候補]
D -->|UI 制御を重視| I[Flutter]
```

6.9 2026年時点の要点

要点	実務での意味
宣言的UIが標準	iOS は SwiftUI、Android は Compose の理解が基礎になる
Expo は本格運用の候補	早く始めて必要時にネイティブへ寄せやすい
KMP は共有ロジック用途で有力	UI 完全共有より、境界を明確にした方が成功しやすい
PWA は依然として強い選択肢	社内ツール、AI フロント、予約系で特に有効
オンデバイス AI は現実的	ただし軽い推論に限定した方が設計しやすい
ストア配信は規約対応が重い	開発初期から審査要件を意識する必要がある

この章のチェックリスト

- [ ] ネイティブ と クロスプラットフォーム と PWA の違いを説明できるか
- [ ] Expo / React Native と Flutter と KMP の使い分けを説明できるか
- [ ] KMP が UI 完全共有ではなくロジック共有に向く理由を説明できるか
- [ ] PWA が有効な案件と厳しい案件を挙げられるか
- [ ] Service Worker と Manifest の役割を説明できるか
- [ ] オンデバイス AI とクラウド AI の使い分けを説明できるか
- [ ] App Store と Google Play で詰まりやすい審査ポイントを説明できるか
- [ ] 配信コストが開発費だけではないことを説明できるか

第7章 クラウドサービス徹底比較 ― AWS / GCP / Azure

> この章を読む前に: ネットワークの基礎（VPC、IP、DNS）と、
> 第5章のデータベース設計、第14章のアーキテクチャ設計を把握していると理解しやすい。

クラウドを選ぶとき、最初に比較すべきなのはサービス数ではない。

見るべきなのは、どの運用モデルに乗るか、どのデータを中心に据えるか、どの組織能力と噛み合うかの3点だ。

AWS、Azure、GCP はどれも十分に本番投入できる。

問題は「どれが最強か」ではなく、「自分たちの制約と優先順位に対して、どれが最も摩擦が少ないか」である。

7.1 クラウドプロバイダの全体像と選定基準

なぜ最初にここを決めるのか

クラウドの選定は、次の領域に長く影響する。

1. 運用体制
2. 採用と育成
3. セキュリティ設計
4. データの置き場所
5. AI / 分析基盤

あとから乗り換えることはできるが、IAM、ネットワーク、監視、課金、社内知識まで含めて移る必要がある。

そのため、初期選定では機能の数より 継続運用のしやすさ を重視した方がよい。

3社の見方

プロバイダ	相対的な強み	向いている状況	注意点
AWS	サービスの幅が広く、選択肢が多い	汎用的な基盤、幅広い採用市場、細かい制御が欲しい	選択肢が多すぎて設計が散りやすい
Azure	Microsoft 製品やエンタープライズ運用との相性がよい	Entra ID、Microsoft 365、Windows 系資産が強い	ポータルや概念が多く、慣れないと追いにくい

プロバイダ	相対的な強み	向いている状況	注意点
GCP	データ分析、サーバーレス、シンプルな運用に強みがある	BigQuery、Cloud Run、データ志向の開発	AWS や Azure より社内経験者が少ない場合がある

選定の判断軸

```
graph TD
A[クラウドを選ぶ] --> B{既存の重い投資はあるか}
B -->|Microsoft 中心| C[Azure を優先確認]
B -->|AWS 中心| D[AWS を優先確認]
B -->|特になし| E{主戦場は何か}
E -->|データ分析 / サーバーレスHTTP| F[GCP を有力候補]
E -->|汎用基盤 / 選択肢の広さ| G[AWS を有力候補]
E -->|社内ID・業務基盤統合| H[Azure を有力候補]
```

実務で先に確認すべき項目

観点	確認内容
ID 基盤	Entra ID、Google Workspace、既存 SSO とどう繋ぐか
データ重力	DB、DWH、ログ、学習データをどこに置くか
運用モデル	VM、Kubernetes、サーバーレスのどれを主軸にするか
調達と規制	リージョン、契約、審査、監査証跡の要件
人材	社内で誰が面倒を見られるか

まとめ

迷ったときの初期仮説としては、次で十分である。

- AWS: 汎用基盤として広く使いたい

- Azure: Microsoft 資産と一体で運用したい

- GCP: データ分析や Cloud Run 中心で素早く進めたい

7.2 コンピュート（計算資源）

なぜ重要か

同じアプリでも、どこで動かすかによって運用コストと障害の出方が大きく変わる。

まずは「VM
か」「コンテナか」「関数か」を決め、その後にベンダー固有サービスへ落とし込むと整理しやすい。

全体像

実行モデル	AWS	GCP	Azure	向いている場面
VM	EC2	Compute Engine	Virtual Machines	常駐プロセス、独自OS設定、既存資産移行
マネージドWeb/PaaS	App Runner / Elastic Beanstalk	App Engine	App Service	Webアプリを早く公開したい
サーバーレスコンテナ	ECS on Fargate	Cloud Run	Container Apps	HTTP API、ワーカー、可変負荷
Kubernetes	EKS	GKE	AKS	K8s を明示的に使う理由がある
関数	Lambda	Cloud Functions	Azure Functions	イベント駆動、短い非同期処理

どれを選ぶべきか

要件	第一候補
常時起動、細かなOS制御	VM
HTTP アプリをなるべく運用せず出したい	サーバーレスコンテナ
数秒〜数分のイベント処理	関数
複数サービスをKubernetesで統一管理したい	Kubernetes
既存のモノリスを比較的素直に載せたい	PaaS か VM

実務での注意点

> WARNING
>
> Kubernetes は「いつでも正解」ではない。 すでにクラスタ運用能力があるか、Kubernetes でしか解けない要件がある場合に限って選ぶ方がよい。

> WARNING
>
> 関数は便利だが、長時間処理や複雑な状態管理には向かない。 たとえば AWS Lambda は 1 回の実行に 15 分の上限があるため、長いバッチや重い推論ではコンテナやジョブ実行の方が素直である。

判断フロー

```
graph TD
A[実行基盤を選ぶ] --> B{HTTP 中心か}
B -->|はい| C{常時起動が必要か}
C -->|いいえ| D[サーバーレスコンテナ]
C -->|はい| E{OS 制御が必要か}
E -->|はい| F[VM]
E -->|いいえ| G[PaaS も候補]
B -->|いいえ| H{イベント処理か}
H -->|はい| I[関数]
H -->|いいえ| J{Kubernetes を使う理由があるか}
J -->|はい| K[Kubernetes]
```

```
J -->|いいえ| L[ジョブ実行 or コンテナ]
```

まとめ

初心者が最初に検討すべき順番は、だいたい次でよい。

1. Cloud Run / Fargate / Container Apps
2. Lambda / Functions
3. VM
4. Kubernetes

Kubernetes は最後に正当化する対象であって、最初に置く前提ではない。

7.3 ストレージ & CDN

なぜ重要か

クラウドの費用と性能を崩しやすいのは、CPU よりも 保存 と 転送 である。

特に画像、動画、ログ、バックアップ、AI 用データはここに集まる。

ストレージの種類

種類	AWS	GCP	Azure	主な用途
オブジェクト	S3	Cloud Storage	Blob Storage	画像、動画、バックアップ、データレイク
ブロック	EBS	Persistent Disk	Managed Disks	VM のディスク
ファイル	EFS	Filestore	Azure Files	共有ファイル、レガシー移行

CDN / エッジ配信

用途	AWS	GCP	Azure
グローバル配信	CloudFront	Cloud CDN	Azure Front Door
DNS	Route 53	Cloud DNS	Azure DNS

CloudFront は AWS の他サービスとの一体感が強い。

Cloud CDN は Google Cloud のロードバランサと合わせて使いやすい。

Azure Front Door は CDN とグローバル入口、WAF の統合体験を取りやすい。

実務で気を付ける点

1. 保存単価 より 転送料 を先に確認する

2. ライフサイクルルールを最初から入れる

3. CDN のキャッシュ無効化とオリジン更新手順を決める

4. AI / 分析用データは「どこに置くと近いか」を意識する

> NOTE
>
> オブジェクトストレージ自体の機能差は大きくない。
> 差が出るのは、周辺サービスとの接続、ライフサイクル管理、転送料、運用慣れである。

まとめ

ストレージで最初に決めるべきことはサービス名ではなく、

保存先、転送量、ライフサイクル の3つである。

7.4 データベースサービス

> 詳細なデータ設計の考え方は [第5章](./ch05-database.md) を参照。

まず押さえるべきこと

クラウドを選ぶ前に、データモデル を決める。

クラウドの都合で RDB を NoSQL に変えたり、その逆をやると失敗しやすい。

代表的な選択肢

ワークロード	AWS	GCP	Azure	向いている場面
標準的な RDB	RDS	Cloud SQL	Azure Database / Azure SQL	一般的な業務システム
PostgreSQL 系の高機能運用	Aurora PostgreSQL	AlloyDB / Cloud SQL for PostgreSQL	Azure Database for PostgreSQL	既存 PostgreSQL を軸にしたい
キー値 / ドキュメント	DynamoDB	Firestore	Cosmos DB	高頻度アクセス、柔軟スキーマ
分析系 DWH	Redshift	BigQuery	Synapse Analytics	分析、集計、BI
キャッシュ	ElastiCache	Memorystore	Azure Cache for Redis	レイテンシ削減

どう選ぶか

条件	候補
業務DBとして堅く運用したい	RDS / Cloud SQL / Azure Database
PostgreSQL を中心に育てたい	Aurora PostgreSQL / AlloyDB / Azure Database for PostgreSQL
アプリがドキュメント指向でリアルタイム性も重視	Firestore / Cosmos DB / DynamoDB
SQL で大規模分析したい	BigQuery / Redshift / Synapse

実務での注意点

> WARNING
>
> BigQuery があるから GCP、Cosmos DB があるから Azure
> のように、単一サービスだけでクラウド全体を決めるのは危険だ。
> 認証、監視、ネットワーク、チーム知識まで含めて見ないと後で歪む。

> WARNING
>
> ベンダー固有DBは魅力的だが、移行コストも増やす。 まずは標準的な PostgreSQL / MySQL
> で足りるかを検討し、足りない理由があるときだけ専用サービスへ進む方が安全だ。

まとめ

データベースは クラウドに合わせる のではなく、ワークロードに合わせる。

そのうえで、周辺運用が最も滑らかなクラウドを選ぶ。

7.5 ネットワーキング & セキュリティサービス

なぜ重要か

クラウドの障害や事故は、アプリコードより 境界の設計ミス で起きることが多い。

特に公開入口、権限、秘密情報、WAF、DNS は初期から整えておくべきである。

基本の対応表

領域	AWS	GCP	Azure
プライベートネットワーク	VPC	VPC	VNet
L4 / L7 入口	ALB / NLB	Cloud Load Balancing	Load Balancer / Application Gateway / Front Door
WAF	AWS WAF	Cloud Armor	Front Door WAF / Application Gateway WAF

領域	AWS	GCP	Azure
DDoS	Shield	Cloud Armor / 標準保護	Azure DDoS Protection
秘密情報	Secrets Manager	Secret Manager	Key Vault
鍵管理	KMS	Cloud KMS	Key Vault / Managed HSM

設計の考え方

1. 公開入口は少なくする
2. 内部通信は認証付きで考える
3. 秘密情報はコードや CI 変数に直書きしない
4. WAF とレート制限を早期に入れる
5. 監査ログを後回しにしない

よくある誤り

誤り	問題
何でもパブリックに出す	攻撃面が増える
入口と内部APIで同じ認証水準にする	横移動に弱くなる
シークレットを GitHub Actions の長寿命シークレットだけで回す	期限管理と棚卸しが崩れる
WAF を本番直前まで入れない	学習期間がなく、誤検知対応も遅れる

まとめ

ネットワークとセキュリティは「最後に付ける機能」ではない。

クラウド設計の最初のレイヤーである。

7.6 AI/ML サービス

この領域だけは特に変化が速い

AI サービスは、モデルカタログ、地域提供、料金、ガードレール機能が短期間で変わる。

したがって、ここでは どのモデルが一番か ではなく、どのプラットフォームで運用しやすいか を見る。

大きな整理

領域	AWS	GCP	Azure
モデル利用基盤	Bedrock	Vertex AI	Azure AI Foundry / Azure OpenAI
ML / MLOps	SageMaker	Vertex AI	Azure Machine Learning
データ統合	S3, Redshift, RDS との連携	BigQuery, Cloud Storage との連携	Microsoft データ基盤との連携

選定の判断軸

観点	何を見るか
モデル選択肢	複数モデルを比較しやすいか
ガバナンス	権限、監査、ネットワーク制御を掛けやすいか
既存データ	DWH、検索、ファイル、業務データに近いか
開発体験	SDK、評価、デバッグ、運用導線が揃うか
地域提供	必要リージョンで使えるか

実務での使い分け

状況	向きやすい選択
複数モデルを横断比較したい	Bedrock / Vertex AI / Azure AI Foundry のモデルカタログ系
既存の MLOps 資産がある	SageMaker / Vertex AI / Azure ML
分析基盤と AI を密に繋ぎたい	BigQuery + Vertex AI など、近いデータ基盤を持つ構成
Microsoft の認証・統制に強く寄せたい	Azure 系

> WARNING
>
> AI プラットフォームは「その日のモデル提供状況」で体験が変わる。
> 設計書にモデル名を固定しすぎず、切り替え可能な抽象 を持たせた方が寿命が長い。

まとめ

AI サービス比較では、性能表より 運用境界 を見た方が実際に役立つ。

モデルが変わっても、権限、評価、監視、データ接続の設計は残るからだ。

7.7 メッセージング & イベント駆動

なぜ必要か

非同期処理、ファンアウト、バッチ連携、ワークフロー自動化は、

アプリケーションが大きくなるほど避けて通れない。

主要サービス

パターン	AWS	GCP	Azure	向いている場面
キュー	SQS	Cloud Tasks / Pub/Sub	Service Bus Queue	バックグラウンド処理、再試行
パブ/サブ	SNS / EventBridge	Pub/Sub	Service Bus Topic / Event Grid	複数購読先への配信
ワークフロー	Step Functions	Workflows	Logic Apps / Durable Functions	複数ステップの業務フロー
ストリーミング	Kinesis	Pub/Sub / Dataflow	Event Hubs	ログ、イベント、ストリーム処理

使い分けの基本

要件	第一候補
1個ずつ確実に処理したい	キュー
同じイベントを複数系へ配りたい	パブ/サブ
状態付きの多段処理を見える化したい	ワークフロー
高スループットの連続イベント	ストリーミング

実務での注意点

1. 再試行 と 冪等性 をセットで考える
2. DLQ を用意する
3. メッセージ順序が必要かを最初に決める
4. ワークフローに業務ロジックを詰め込みすぎない

まとめ

最初の一歩としては、キュー と ワークフロー を区別できれば十分である。

7.8 監視・ログ（クラウドネイティブ観測）

> 詳細は [第11章](./ch11-observability.md) を参照。

全体像

領域	AWS	GCP	Azure
メトリクス	CloudWatch	Cloud Monitoring	Azure Monitor
ログ	CloudWatch Logs	Cloud Logging	Log Analytics
トレース	X-Ray	Cloud Trace	Application Insights

どう始めるべきか

最初から Datadog や New Relic を入れる必要はない。

まずはクラウド標準の監視で次を揃える。

1. レイテンシ

2. エラー率

3. 飽和

4. 主要イベントログ

5. アラート

そのうえで、次の条件になったら外部基盤を検討する。

- 複数クラウドをまたぐ

- SaaS と自前基盤を横断したい

- OpenTelemetry を中心に統一したい

- チーム横断で同じダッシュボード文化を持ちたい

まとめ

監視はプロダクト後半の話ではない。

デプロイ初日から最低限の可観測性を持たせるべきである。

7.9 クラウド選択 まとめフロー

```
graph TD
A[クラウドを選ぶ] --> B{既存の重い投資はあるか}
B -->|Microsoft 中心| C[Azure を第一候補]
B -->|AWS 中心| D[AWS を第一候補]
B -->|特になし| E{主な開発スタイルは何か}
E -->|サーバーレスHTTP / データ分析| F[GCP を強く検討]
E -->|汎用基盤 / 幅広い選択肢| G[AWS を強く検討]
E -->|社内IT統合 / ID統制| H[Azure を強く検討]
```

現時点の整理

結論	こう読むと実務に使いやすい
AWS	選択肢が多い汎用基盤
Azure	Microsoft 統合に強い基盤
GCP	Cloud Run / BigQuery / Vertex AI と相性の良い基盤

この章のチェックリスト

- [ ] AWS、Azure、GCP の違いを「最強論」ではなく判断軸で説明できるか
- [ ] VM、サーバーレスコンテナ、関数、Kubernetes の使い分けを説明できるか
- [ ] オブジェクトストレージと CDN で転送料を先に見る理由を説明できるか
- [ ] DB はクラウド都合ではなくワークロード都合で選ぶべき理由を説明できるか
- [ ] WAF、シークレット管理、監査ログを初期から入れる理由を説明できるか
- [ ] AI/ML サービスではモデル名より運用境界を見るべき理由を説明できるか
- [ ] キュー、パブ/サブ、ワークフローの違いを説明できるか
- [ ] 監視はクラウド標準から始めてよい理由を説明できるか

第8章 PaaS・サーバーレス・BaaS 実践ガイド

この章を読む前に: 第7章のクラウドサービス比較を先に読んでおくとつながりが見えやすい。

この章で大事なのは、「どのサービスが最強か」を探すことではない。

重要なのは、どこまでを自分たちで運用し、どこからをプラットフォームに任せるか を決めることだ。

同じ Web サービスでも、個人開発、MVP、成長期、規制対応のある事業では最適解が変わる。

抽象度が高いほど立ち上がりは速いが、制御できる範囲は狭くなる。

逆に、自由度が高いほど運用責務は重くなる。

8.1 IaaS → PaaS → サーバーレス → BaaS の段階と判断基準

このセクションで答える問い: インフラの抽象度はどう違い、いつ何を選ぶべきか？

全体像

レイヤー	何を自分で持つか	強み	注意点	向いている場面
IaaS	VM、ネットワーク、OS、実行基盤	自由度が高い	運用責務が重い	特殊要件、既存資産移行、厳しい統制
PaaS	アプリコードと設定中心	立ち上がりが速い	深い制御は難しい	Web アプリ、API、社内ツール
サーバーレス	関数やコンテナ単位のコード	スケールを任せやすい	実行時間や状態管理に制約がある	イベント処理、API、ジョブ
BaaS	主にフロントエンドと業務ロジック	MVP が速い	複雑な業務ルールは乗りにくい	認証、CRUD、リアルタイム、プロトタイプ

どう考えるべきか

判断軸は次の4つで十分である。

1. 運用できる人がいるか
2. 端末やネットワークに深く踏み込む必要があるか
3. ビジネスロジックがどれだけ複雑か
4. 早さと自由度のどちらを優先するか

判断フロー

```
graph TD
A[プラットフォームを選ぶ] --> B{最優先は何か}
B -->|最速で出す| C[BaaS か PaaS]
B -->|自由度| D[IaaS か Kubernetes]
B -->|イベント処理| E[サーバーレス]
C --> F{複雑な業務ロジックがあるか}
F -->|はい| G[PaaS + 自前API]
F -->|いいえ| H[BaaS を優先]
```

まとめ

最初の仮説としては次で十分だ。

- BaaS: 認証、DB、ストレージを最速で揃えたい

- PaaS: Web アプリや API を早く安定運用したい

- サーバーレス: イベント駆動やバースト処理に寄せたい

- IaaS: 深い制御が必要で、運用能力もある

8.2 フロントエンド特化 PaaS

> このセクションで答える問い: フロントエンドのホスティングは何を基準に選ぶべきか？

代表的な選択肢

サービス	強み	向いている場面	注意点
Vercel	フレームワーク連携、Preview Deployment、Next.js との相性	Next.js、App Router、BFF 的 API 同居	ベンダー固有機能に寄せると移行コストが上がる
Cloudflare Pages	エッジ配信、Workers との連携、周辺ストレージが近い	グローバル配信、軽いエッジロジック、キャッシュ戦略重視	実行モデルが独特で、Node 前提コードは確認が必要
Netlify	静的サイト、フォーム、ビルドプラグイン、分かりやすい導線	コーポレートサイト、ブログ、静的中心のサイト	最新フレームワーク機能との相性は案件ごとに確認が必要

何で決めるか

判断軸	見るべき点
フレームワーク	Next.js を中心に据えるか
実行場所	エッジ実行が本当に必要か
バックエンド接続	Functions や Workers を同居させたいか
チーム運用	Preview、Branch ごとの検証、権限管理をどう回すか

実務での整理

- Next.js 中心: まず Vercel を検討

- エッジロジックも同居: Cloudflare Pages + Workers を検討

- 静的サイトや小規模CMS連携: Netlify も有力

> NOTE
>
> フロントエンド特化 PaaS は「HTML を置く場所」ではない。
> プレビュー環境、画像最適化、キャッシュ、ドメイン管理、関数実行まで含めた運用基盤として見るべきである。

まとめ

どれを選んでも基本的なホスティングはできる。

差が出るのは フレームワークとの噛み合い と 周辺機能まで含めた運用体験 である。

8.3 フルスタック PaaS

このセクションで答える問い: フロントエンドだけでなく API や DB も含めて素早く動かしたいなら？

代表的な選択肢

サービス	強み	向いている場面	注意点
Railway	立ち上がりが速く、サービス追加が軽い	MVP、内部ツール、小〜中規模API	使用量の増え方は早めに監視する必要がある
Render	Web サービス、ワーカー、Cron を分かりやすく運用できる	常時起動アプリ、Heroku 的な運用	高度なネットワーク制御は限定的
Fly.io	コンテナ中心、リージョン配置、ネットワーク制御	複数リージョン、低レイテンシ API、運用を少し持ちたい案件	PaaS というより軽量インフラに近く、学習量が増える
Coolify	自ホスト型で自由度が高い	自社管理、ベンダーロックイン回避、自己運用	便利なUIがあっても、実態は自前運用である

どう選ぶか

条件	候補
最短で API と DB を置きたい	Railway / Render
常時起動サービスとジョブを素直に分けたい	Render
リージョン配置やネットワークを意識したい	Fly.io
画面は欲しいが、基盤は自社で握りたい	Coolify

実務での注意点

> WARNING
> PaaS = 運用不要 ではない。 バックアップ、障害切り分け、コスト監視、環境差分管理は依然として必要である。

> WARNING
> 自ホスト型 PaaS = ロックイン回避 も半分だけ正しい。 SaaS ロックインは減るが、今度は自分たちが運用責務を持つ。

まとめ

フルスタック PaaS は、自前インフラへ行く前の一段階 として非常に有効だ。

ただし、自由度を買うたびに運用責務も増える。

8.4 BaaS（Backend as a Service）

> このセクションで答える問い: バックエンド開発をどこまで省力化できるか？

代表的な選択肢

サービス	強み	向いている場面	注意点
Supabase	PostgreSQL を中心に、Auth / Storage / Realtime をまとめて使える	SQL ベースで考えたい SaaS、管理画面、業務アプリ	複雑な権限や重い業務処理は自前APIが欲しくなる
Firebase	モバイル / Web SDK、Auth、Firestore、Functions、分析系との接続	モバイル寄り、Google エコシステム、素早いプロトタイプ	データモデルと課金モデルを理解せずに進めると後で詰まりやすい
Appwrite	オープンソースでセルフホストしやすい	自律性や自己運用を重視する組織	マネージド体験より運用知識が要る

BaaS が向いている範囲

- 認証

- ファイルストレージ

- 単純な CRUD

- 通知やリアルタイム更新

- 社内ツールやプロトタイプ

早めに自前APIを持つべき兆候

1. 業務ルールが増えてきた

2. 複数システム連携が増えた

3. 権限判定が画面単位から業務単位になった

4. 監査ログや承認フローが重くなった

> NOTE
>
> BaaS は「卒業すべきもの」ではない。
> 認証やストレージだけ残して、ドメインロジックだけ自前へ移す形でも十分に実用的である。

まとめ

BaaS は バックエンドを無くす のではなく、

バックエンドのうち、差別化にならない部分を外注する と考えると使いやすい。

8.5 サーバーレスコンピュート

> このセクションで答える問い: 関数や軽量実行基盤はどこで使うべきか？

代表的な選択肢

サービス	強み	向いている場面	注意点
AWS Lambda	AWS イベントと接続しやすく、エコシステムが広い	バックグラウンド処理、API、イベント駆動	長時間処理や状態の多い処理には不向き
Cloudflare Workers	エッジで軽いロジックを動かしやすい	認証前段、リダイレクト、ヘッダ変換、軽量API	長いCPU処理や重いライブラリには向かない
Vercel Functions	フロントエンドと一体で扱いやすい	BFF、フォーム処理、軽いAPI	アプリが Vercel 前提になりやすい
Deno Deploy	Deno / TypeScript と相性がよい	Deno 資産の継続利用、軽量API	採用やエコシステムは案件次第

何に向いているか

処理	向く実行基盤
Webhook 受信	Lambda / Functions
認証前段やエッジ制御	Workers
フロントエンド寄り API	Vercel Functions
長いジョブや重い推論	コンテナ or ジョブ実行

実務での注意点

> WARNING
>
> サーバーレスは「無限に小さいサーバー」ではない。
> 実行時間、メモリ、同時実行、接続数、ローカル状態の扱いを理解していないと、すぐに設計が破綻する。

> WARNING
>
> エッジ実行は魅力的だが、何でもエッジに置くべきではない。 データベースが遠い、CPU が重い、依存ライブラリが Node 前提、のいずれかがあるなら中央リージョン実行の方が安定する。

まとめ

サーバーレスは 小さな責務を切り出す と強い。

逆に、長く走る処理や状態の多い処理を押し込む場所ではない。

8.6 コンテナ実行とオーケストレーション

このセクションで答える問い: 本当に Kubernetes が必要か、それともマネージドコンテナで足りるか？

まず押さえるべきこと

多くのチームにとって、最初の正解は Kubernetes ではない。

まずは Cloud Run や ECS/Fargate のような、より単純なコンテナ実行基盤を検討した方がよい。

代表的な選択肢

選択肢	強み	向いている場面	注意点
Cloud Run	ステートレスなコンテナを素早く出せる	HTTP API、ジョブ、マイクロサービス	状態を持つ設計には向かない
ECS / Fargate	AWS 内でコンテナ運用を整理しやすい	AWS 前提の API、ワーカー、ジョブ	VPC やロードバランサ設定は理解が要る
Kubernetes	複数チーム・複数サービスの標準化に強い	プラットフォーム運用、複雑な依存、細かな制御	専門知識と継続運用体制が前提

Kubernetes が本当に必要な兆候

1. 多数のサービスを横断して標準化したい

2. チームごとにデプロイ境界が独立している

3. サイドカー、サービスメッシュ、細かなスケジューリングが必要

4. 既にクラスタ運用経験がある

Kubernetes が早すぎる兆候

- アプリがまだ少ない

- デプロイより仕様変更の方が多い

- Platform / SRE の担当がいない

- 監視やCIすら未整備

まとめ

順番としては次が現実的だ。

1. Cloud Run / ECS / Fargate

2. 必要になったら Kubernetes

8.7 実プロジェクトでの組み合わせ例

> このセクションで答える問い: 規模やフェーズごとに、どの組み合わせが自然か？

1. 個人開発・検証

役割	候補
フロントエンド	Vercel / Netlify / Cloudflare Pages
API	Railway / Render / Vercel Functions
データ	Supabase / Firebase

向く条件:

- まず公開して反応を見たい

- インフラ学習よりプロダクト検証が優先

2. MVP・初期スタートアップ

役割	候補
フロントエンド	Vercel
API	Railway / Render / Cloud Run
認証・DB	Supabase or Firebase
監視	Sentry + プラットフォーム標準ログ

向く条件:

- 少人数で速く回したい

- Preview と本番をはっきり分けたい

- 認証やストレージは自前実装したくない

3. 成長期

役割	候補
フロントエンド	Vercel などの専用ホスティング
API / ワーカー	Cloud Run / ECS / Container Apps
DB	マネージド PostgreSQL、必要なら Redis
非同期	キューとジョブ実行
監視	OpenTelemetry + 外部監視SaaS or 標準監視

向く条件:

- ワーカー、バッチ、通知などが増えてきた

- BaaS だけでは業務ロジックが乗らない

4. 高統制・大規模運用

役割	候補
実行基盤	マネージド Kubernetes or 厳格なコンテナ基盤
DB	マネージドDB + 分析基盤
IaC	Terraform / OpenTofu / Pulumi
配信	専用 CDN / WAF / DNS
監視	標準監視 + 外部可観測性基盤

向く条件:

- 監査、権限、環境分離、可用性要件が重い

- チーム横断のプラットフォーム運用が必要

まとめ

規模が大きいほど IaaS に寄る、とは限らない。

実際には、フロントエンドは PaaS、API はコンテナ、認証やストレージは BaaS / マネージドのような混成構成が多い。

8.8 AI 時代の PaaS / サーバーレス

このセクションで答える問い: AI 機能を持つアプリでは何を基準に基盤を選ぶべきか？

先に結論

AI 機能があるからといって、必ず重いインフラが必要になるわけではない。

実際には、次の3種類に分かれる。

パターン	向く基盤
LLM API を薄く呼ぶ	Vercel Functions / Lambda / Cloud Run
ストリーミング付きチャット	フロントエンド PaaS + サーバーレス or コンテナ
埋め込み生成、RAG 更新、評価バッチ	コンテナ / ジョブ実行 / ワーカー

実務で重要な設計点

1. ストリーミング応答 を扱いやすいか
2. API キーや秘密情報 を安全に置けるか
3. バッチ更新 を分離できるか
4. ログと評価 を残せるか
5. モデル切り替え に耐えられるか

よくある構成

機能	よくある組み合わせ
チャットUI	Vercel + Functions
軽い推論ゲートウェイ	Workers / Lambda
RAG の文書格納	Supabase / PostgreSQL / マネージドDB
埋め込み更新や再処理	Cloud Run Jobs / ECS Tasks / バッチ基盤

> WARNING
>
> AI 機能は、HTTP リクエストの裏で終わらないことが多い。
> 埋め込み更新、再ランキング、評価、監査ログなどの裏処理をどこで回すかまで考えないと、PaaS だけでは足りなくなる。

まとめ

AI 時代の基盤選定では、モデル性能 より 非同期処理と運用導線 を見た方が失敗しにくい。

8.9 FinOps - コスト最適化戦略

このセクションで答える問い: PaaS やサーバーレスのコストはどう管理すべきか？

まずやるべきこと

PaaS や BaaS は立ち上がりが速い一方で、何に課金されているか を見失いやすい。

最初から次を入れる。

1. 予算アラート
2. 環境ごとのタグやラベル
3. 所有者の明確化
4. デプロイ前の概算確認

コストが膨らみやすい場所

領域	典型例
データ転送	CDN、画像配信、外部API
ストレージ	ログ、バックアップ、アップロード
BaaS	読み書き、MAU、ストレージ、関数
サーバーレス	実行時間、同時実行、外部呼び出し
AI	トークン、埋め込み、評価ジョブ

ツールの考え方

種類	役割
クラウド標準の予算アラート	まず入れるべき基本機能
Infracost のような事前試算ツール	IaC 変更の差分確認
Vantage のような横断分析ツール	複数サービスや複数クラウドの可視化

まとめ

FinOps は大企業の専用品ではない。

小規模でも 月額が読めない状態 を放置しないことが重要だ。

8.10 選択フローチャート

```
graph TD
A[基盤を選ぶ] --> B{最優先は何か}
B -->|検証速度| C[BaaS + フロントエンド PaaS]
B -->|Web/API の素早い公開| D[PaaS]
B -->|イベント処理| E[サーバーレス]
B -->|深い制御| F[IaaS / Kubernetes]
D --> G{業務ロジックは重いか}
G -->|はい| H[PaaS + 自前API]
G -->|いいえ| I[PaaS 単体でも可]
```

この章のチェックリスト

- [ ] IaaS、PaaS、サーバーレス、BaaS の違いを説明できるか
- [ ] フロントエンド特化 PaaS をフレームワーク相性で選ぶ理由を説明できるか
- [ ] フルスタック PaaS が「運用不要」ではない理由を説明できるか
- [ ] BaaS を使いながら自前APIへ段階移行する考え方を説明できるか
- [ ] サーバーレスを長時間ジョブに押し込まない理由を説明できるか
- [ ] Kubernetes が早すぎる兆候を説明できるか
- [ ] AI 機能ではストリーミングと非同期ジョブの置き場が重要な理由を説明できるか
- [ ] PaaS / BaaS のコストを最初から監視すべき理由を説明できるか

第9章 IaC・コスト最適化・エッジ戦略

この章を読む前に: 第7章、第8章を先に読むと、クラウド基盤と PaaS の役割分担がつながりやすい。

インフラ運用が難しくなるのは、サーバー台数が増えたときではない。

誰が何を変えたのか が分からなくなり、いくらかかっているのか が見えなくなり、どこで処理すべきか を誤ったときに一気に崩れる。

この章では、次の3つをまとめて扱う。

1. IaC で変更を再現可能にする
2. FinOps でコストを見える化する
3. エッジ と リージョン の役割を分ける

9.1 IaC（Infrastructure as Code）

このセクションで答える問い: インフラをコードで管理するとはどういうことか？

IaC の本質

IaC は「インフラをコードで作る」ことより、

インフラの変更を再現可能にする ための手段だ。

手作業のコンソール操作は早く見えるが、次の問題を残しやすい。

- 再現できない
- 監査できない
- レビューできない
- 環境差分が増える

主な選択肢

ツール	特徴	向いている場面	注意点
Terraform	宣言的で普及が広い	HCL に抵抗がなく、既存資産も多い組織	将来のライセンス方針を含めて判断する必要がある
OpenTofu	Terraform 系の互換性を持つ OSS 系列	Terraform 系の記法を使いたいが OSS 志向が強い	互換性に期待しすぎず、使う機能の差分は確認が必要

ツール	特徴	向いている場面	注意点
Pulumi	汎用プログラミング言語で書ける	TypeScript / Python / Go で統一したい	アプリコードと IaC の境界が曖昧になることがある
AWS CDK	AWS に強く寄せた抽象化	AWS 専用で進める	AWS 外へは広げにくい
SST	アプリとインフラを近い場所で扱いやすい	モダンなフルスタックアプリを自前基盤で運用したい	IaC 汎用ツールというよりアプリ基盤フレームワーク寄り

何で選ぶか

判断軸	見るべき点
クラウド範囲	AWS 専用か、複数クラウドか
言語嗜好	宣言型を好むか、汎用言語で書きたいか
既存資産	既にある module / provider / construct を流用できるか
チーム体制	アプリ開発者がそのまま触るのか、基盤専任が持つのか

State とロックの扱い

IaC ツールで最も事故になりやすいのは state の管理である。

state には、現在の実環境との差分を計算するための情報が入る。

この中には機密情報や内部IDが含まれることがある。

> WARNING
>
> state はローカルPCの作業ファイルではない。
> チームで使うなら、共有バックエンド、暗号化、アクセス制御、ロックを最初に決めるべきである。

実務上の注意

1. remote state を使う
2. 暗号化とアクセス制御を入れる
3. ロック方式はツールと backend に合わせる
4. plan / preview をレビューに組み込む
5. 手動変更による drift を定期的に検知する

> NOTE
>
> Terraform 系では S3 backend のロック方式や推奨構成が更新されている。 以前よく見かけた DynamoDB ベースの手法をそのまま写経するのではなく、今使うバージョンの公式 backend ドキュメントを確認した方が安全である。

いつから IaC を入れるべきか

「大きくなったら IaC」では遅いことが多い。

少なくとも次は早めにコード化した方がよい。

- 本番ネットワーク

- 本番データベース

- DNS

- シークレット参照先

- 共有キューやストレージ

逆に、試作段階ではすべてを厳密に IaC 化しなくてもよい。

重要なのは 共有される本番基盤 から先にコード化することだ。

まとめ

IaC の選定は、ツール宗派で決める話ではない。

state を安全に持てるか、レビューに組み込めるか、チームが継続して読めるか の3点で考えると失敗しにくい。

9.2 コスト最適化戦略（FinOps）

> このセクションで答える問い: クラウド費用の暴走をどう防ぐか？

FinOps をどう捉えるか

FinOps は「安くする活動」ではない。

費用の発生理由を説明できる状態を作ること が先にある。

そのため、最初の一歩は削減テクニックではなく、可視化である。

まず入れるべきもの

項目	理由
予算アラート	想定外の増加に気づける
タグ / ラベル / 所有者	どのチーム、どの環境の費用か追える
環境分離	開発・検証・本番の費用を混ぜない
日次または週次レビュー	変化を早く見つける

> WARNING
>
> Budget は上限装置ではないことが多い。 AWS、GCP、Azure
> いずれも、予算通知は「止める」より「知らせる」機能として理解した方が正確である。

コストが増えやすい場所

領域	典型例
コンピュート	常時稼働、過剰サイズ、不要環境の放置
データ転送	CDN、リージョン間通信、外部API
ストレージ	ログ、バックアップ、保持期間の長すぎるデータ
DB	過大スペック、読み取り増加、接続爆発
観測	ログを何でも全文保存する
AI	トークン、埋め込み、再処理ジョブ、評価実行

最適化の順番

1. 不要なものを消す
2. サイズを下げる
3. 保存期間を見直す
4. 確定負荷に割引制度を当てる
5. アーキテクチャ自体を見直す

代表的な手段

手段	向いている場面
予約 / Savings 系	ベースロードが読める
Spot / Preemptible 系	中断に耐えるジョブ
IaC 事前試算	変更前にコスト差分を確認したい
Kubernetes コスト可視化	クラスタ利用量をチーム別に見たい

AI ワークロードのコスト

AI では、インスタンス費用より 呼び出し量 と 再実行回数 が支配的になることが多い。

特に見落としやすいのは次である。

- 長いプロンプトの常時送信
- 同じ埋め込みの再計算
- 評価や再ランキングの裏処理

- ストリーミング中のキャンセル未処理

AI コストを抑える基本

1. タスクごとにモデルを分ける
2. キャッシュを入れる
3. 埋め込み更新を非同期化する
4. 再試行回数とタイムアウトを制御する
5. 観測コストも含めて見る

ツールの考え方

種類	役割
クラウド標準の予算・コスト分析	最初に入れる基本機能
Infracost のような事前試算	PR 時点で差分を見たい
Vantage / OpenCost / Kubecost のような横断可視化	複数サービスや Kubernetes をまとめて見たい

まとめ

FinOps の第一歩は、節約術ではなく 責任の所在を見える化すること である。

9.3 マルチリージョン・エッジコンピューティング戦略

> このセクションで答える問い: どこで処理を動かすべきか？

まず区別すべきこと

CDN、エッジ実行、マルチリージョン は同じではない。

方式	何を近づけるか	向いているもの
CDN	静的ファイル	JS、CSS、画像、動画
エッジ実行	軽い動的処理	リダイレクト、認証前段、キャッシュ判定、軽い個人化
マルチリージョン	アプリ本体やデータ	グローバルユーザー、地域分散、災害耐性

エッジが向く処理

- 認証前の判定
- Bot / レート制限

- 地域別の振り分け

- キャッシュ確認

- 軽いヘッダ変換やA/B振り分け

エッジが向かない処理

- 重い CPU 処理

- 大きな依存ライブラリを要する処理

- 状態を長く持つワークフロー

- 遠いデータベースに毎回アクセスする処理

> WARNING
>
> エッジに置いても、データベースが中央リージョンにあるなら、その往復は消えない。
> 「関数が近い」ことと「データも近い」ことは別問題である。

典型的な配置

```
ユーザー
↓
CDN / エッジ
├── 静的配信
├── 認証前段
└── 軽い分岐
↓
リージョナル API / コンテナ
├── 認証
├── 業務ロジック
└── DB アクセス
↓
DB / キュー / ストレージ
```

マルチリージョンを選ぶ前に確認すること

1. 書き込み整合性をどう保つか

2. 障害時の切替を自動化できるか

3. 運用チームが複雑化に耐えられるか

4. 本当にレイテンシが課題か

まとめ

多くのプロダクトでは、単一リージョン + CDN + 限定的なエッジ処理 で十分である。

マルチリージョンは、必要性を証明してから入れる方が安全だ。

9.4 AI 時代のクラウドアーキテクチャ

> このセクションで答える問い: AI ワークロードを含む基盤はどう設計すべきか？

まず押さえるべきこと

AI ワークロードが増えると、設計で重要になるのは次である。

1. 同期処理 と 非同期処理 の分離
2. ストリーミング応答
3. 評価と監査ログ
4. モデル切り替え
5. データの近さ

典型構成

```
クライアント
↓
フロントエンド / エッジ
├── 入力検証
├── レート制限
└── キャッシュ確認
↓
アプリ API
├── 認証・権限
├── プロンプト構築
├── RAG 前処理
└── モデル呼び出し
↓
モデル実行基盤
├── 外部モデル API
├── マネージド推論基盤
└── 自前推論基盤
↓
非同期ワーカー
├── 埋め込み更新
├── バッチ評価
├── 再ランキング
└── 監査ログ整形
```

どこで推論するか

選択肢	向いている場面	注意点
外部モデル API	まず価値検証したい	依存先、トークン費用、地域要件
マネージド推論基盤	クラウド内で統制を強めたい	クラウド依存が深くなる
自前推論基盤	利用量が大きく、制御も必要	GPU 運用、評価、配備が重い

設計原則

1. モデル名をコードに埋め込みすぎない
2. 入力と出力を記録できる境界を作る
3. 失敗時の代替経路を用意する
4. 同期リクエスト内で終わらない仕事はワーカーへ逃がす
5. キャッシュと再利用を前提にする

実務での注意点

> WARNING
>
> AI 基盤の失敗は、モデル精度だけで起きるわけではない。
> ほとんどは、タイムアウト、再試行、キュー詰まり、コスト急増、監査不能で起きる。

> WARNING
>
> エッジ AI や軽量推論は有効だが、すべてをエッジに寄せる必要はない。
> 軽い前処理や判定だけをエッジに置き、重い推論はリージョナルに集約する構成の方が安定することが多い。

まとめ

AI 時代のアーキテクチャでは、モデルの種類 より 処理の分解の仕方 の方が重要である。

エッジ、API、ワーカー、データ基盤をそれぞれ無理のない責務に分けるべきだ。

この章のチェックリスト

- [ ] IaC の目的を「コード化」ではなく「再現性の確保」として説明できるか
- [ ] Terraform / OpenTofu / Pulumi / CDK / SST の違いを判断軸で説明できるか
- [ ] state を安全に扱うために必要な条件を説明できるか
- [ ] Budget が通知機能であり、上限装置ではないことを説明できるか
- [ ] FinOps の第一歩が可視化である理由を説明できるか
- [ ] CDN、エッジ、マルチリージョンの違いを説明できるか
- [ ] エッジに向く処理と向かない処理を説明できるか
- [ ] AI 基盤では同期処理と非同期処理を分けるべき理由を説明できるか

第10章 CI/CD とデプロイ戦略

この章を読む前に: Git の基本と、第7章〜第9章の基盤理解があるとつながりやすい。

CI/CD は「自動化のための自動化」ではない。

目的は、変更を速く出すこと と 壊れたときにすぐ止めて戻せること を同時に満たすことだ。

この章では、CI/CD プラットフォーム選定、デプロイ戦略、Feature Flags、アーティファクト管理、GitOps、AI 活用の境界を整理する。

10.1 CI/CD の基本概念と重要性

このセクションで答える問い: CI/CD は何を自動化し、何を人間が判断すべきか？

CI と CD

用語	意味
CI	変更ごとにテスト、静的解析、ビルドを自動実行する
CD	テスト済みの成果物を環境へ届ける流れを自動化する

```
graph LR
A[コード変更] --> B[CI: リント / 型 / テスト]
B --> C[ビルド]
C --> D[成果物保管]
D --> E[ステージング配布]
E --> F[本番デプロイ]
```

手動デプロイの問題

- 再現性がない

- 手順漏れが起きる

- 誰が何をしたか追いにくい

- 夜間や障害時の対応が属人化する

まとめ

CI/CD は「人間を不要にする仕組み」ではなく、

人間が見るべき判断点だけを残す仕組み である。

10.2 CI/CD プラットフォーム

このセクションで答える問い: どの CI/CD ツールを選ぶべきか？

主な選択肢

ツール	強み	向いている場面	注意点
GitHub Actions	GitHub と一体で扱える	GitHub 中心のチーム	Action の第三者依存が増えやすい
GitLab CI	GitLab の SCM、レジストリ、環境管理と近い	GitLab 一体運用	GitHub 前提チームには導線が変わる
CircleCI	CI 専用サービスとしての運用体験	既存VCSに依存しすぎたくない	別サービス管理が増える
Jenkins	自由度が非常に高い	既存資産、オンプレ、強い自己運用	保守負荷が重い
Dagger	パイプラインをコードとして共通化しやすい	ローカルとCIの差分を減らしたい	これ単体で CI サービスになるわけではない

何で選ぶか

判断軸	見るべき点
SCM 一体運用	GitHub / GitLab と同居させたいか
自己運用	ランナーやエージェントを自前で持つか
セキュリティ	OIDC、権限分離、監査ログをどう扱うか
パイプライン表現	YAML 中心か、コード中心か

最低限の CI ワークフロー例

```
name: ci

on:
pull_request:
push:
branches: [main]

jobs:
test:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v4

- name: Node をセットアップ
uses: actions/setup-node@v4
with:
node-version: 22
cache: npm

- name: 依存関係をインストール
```

```
run: npm ci

- name: リント
run: npm run lint

- name: 型チェック
run: npm run typecheck

- name: ユニットテスト
run: npm run test
```

セキュリティ上の基本

1. 長寿命のクラウド資格情報をできるだけ置かない

2. OIDC や短命トークンを優先する

3. サードパーティ Action は信頼できるものを選び、可能なら commit SHA で固定する

4. 本番デプロイ権限と通常CI権限を分ける

> WARNING
>
> CI/CD は開発補助ではなく、本番系への入口でもある。
> そのため、パイプライン自体をアプリ本体と同じかそれ以上に守る必要がある。

まとめ

迷ったら GitHub を使っているなら GitHub Actions、GitLab を使っているなら GitLab CI から始めるのが自然だ。

乗り換える理由は、統制、既存資産、自己運用要件が出てからで十分である。

10.3 デプロイ戦略

> このセクションで答える問い: 本番を壊さずにリリースするには？

主な戦略

戦略	仕組み	強み	注意点
Rolling	順番に入れ替える	単純	戻しに時間がかかることがある
Blue-Green	新旧環境を並べて切り替える	切り戻しが速い	二重環境が必要
Canary	一部トラフィックだけ新バージョンへ流す	リスクを限定できる	メトリクス監視が前提
Feature Flags	デプロイと機能公開を分ける	機能単位で止めやすい	フラグ管理を怠ると複雑化する

どう選ぶか

条件	向く戦略
小規模でシンプルに進めたい	Rolling
切り戻し速度を最優先	Blue-Green
影響を観測しながら広げたい	Canary
コード公開と機能公開を分けたい	Feature Flags

実務で重要なこと

デプロイ戦略そのものより重要なのは、次の3点である。

1. ヘルスチェック
2. ロールバック手順
3. デプロイ直後の監視

> WARNING
>
> Canary や Blue-Green は、監視がなければただ複雑なだけになる。
> エラー率、レイテンシ、主要業務指標を見て止める仕組みが前提である。

まとめ

戦略選定の基準は「最先端かどうか」ではなく、

壊れたときにどれだけ速く戻せるか である。

10.4 Feature Flags（機能フラグ）とデプロイの分離

> このセクションで答える問い: なぜ Feature Flags を使うのか？

役割

Feature Flags は、新機能を デプロイしたが、まだ全員には見せない 状態を作るための仕組みだ。

これにより次が可能になる。

- 段階的公開
- 社内ユーザーだけ先行公開
- 緊急停止

- A/B テスト

主な選択肢

ツール	強み	向いている場面	注意点
LaunchDarkly	マネージド運用、管理機能が厚い	本格運用、複数チーム、分析連携	SaaS 依存になる
Unleash	OSS で自前運用も可能	自律性を重視する組織	自前運用責務がある
Flipt	軽量でシンプル	自ホスト志向、小〜中規模	周辺機能は自分で補う前提がある

フラグ運用の原則

1. 永続フラグ と 短命フラグ を区別する

2. 役目が終わったフラグは消す

3. 認可や秘密情報の代わりに使わない

4. どの画面と処理に効くかを追えるようにする

> WARNING
>
> Feature Flag は便利だが、放置すると最も分かりにくい負債の1つになる。
> 「いつ消すか」を最初から決めておくべきである。

まとめ

Feature Flags はデプロイを安全にする。

ただし、それ自体が設定基盤になるので、運用ルールなしで増やしてはいけない。

10.5 テスト自動化パイプライン

> このセクションで答える問い: CI では何を、どの順番で実行すべきか？

基本の順番

段階	目的
リント	形式と基本ルールの確認
型チェック	型の破壊を早く検出
ユニットテスト	ロジックの正しさを確認
インテグレーションテスト	外部接続や境界を確認

段階	目的
E2E テスト	ユーザー操作全体を確認
セキュリティスキャン	依存関係やコードの問題を確認
ビルド	成果物を実際に作れるか確認

原則

1. 速い検査を先に

2. 遅い検査は絞って後ろに

3. flake を放置しない

4. ローカルでも再現できる形にする

何が壊れやすいか

- E2E を増やしすぎて遅くなる

- 失敗原因が分からないまま再実行文化になる

- テストデータの初期化が不安定になる

- 依存サービスの一時障害で全体が赤くなる

まとめ

CI の理想は、速い、再現できる、失敗理由が分かる の3つを満たすことだ。

10.6 アーティファクト管理

> このセクションで答える問い: ビルドした成果物をどう扱うべきか？

管理対象

- コンテナイメージ

- npm / Python / Maven などのパッケージ

- SBOM

- 署名情報

- ビルドログやテスト結果

主なレジストリ

種類	代表例	向いている場面
GitHub 系	GHCR, GitHub Packages	GitHub 中心運用
クラウド系	ECR, Artifact Registry, Azure Container Registry	そのクラウドで実行する
汎用系	Artifactory, Docker Hub など	複数言語や複数基盤をまとめたい

原則

1. mutable tag だけに依存しない
2. digest で追跡できるようにする
3. 保持期間を決める
4. 脆弱性スキャンを組み込む
5. 可能なら SBOM と署名を付ける

> NOTE
>
> 「latest を pull すればよい」という運用は追跡性を壊しやすい。 どの digest がどのデプロイに使われたかを残すべきである。

まとめ

アーティファクト管理は保管庫ではなく、配布の信頼性 を支える仕組みである。

10.7 GitOps とインフラ自動管理

> このセクションで答える問い: GitOps はいつ有効で、いつ不要か？

GitOps とは何か

GitOps は、Git にある定義を本番の正とみなし、環境側がそれに合わせる 運用である。

主戦場は Kubernetes であり、すべてのデプロイ方式に必要なわけではない。

主な選択肢

ツール	強み	向いている場面	注意点
Argo CD	UI が充実し、可視化しやすい	複数チーム、可視化重視	機能が多く、運用設計が必要
Flux	軽量で Git / Kustomize / Helm との親和性が高い	シンプルに GitOps を回したい	UI 前提ではない

GitOps が向いている場面

- Kubernetes を本格運用している

- 環境差分を Git で厳密に管理したい

- 手動 kubectl を減らしたい

- 監査と再現性が重要

GitOps が不要な場面

- Vercel や Render など PaaS の Git 連携で十分

- まだ Kubernetes を使っていない

- Git に置くべき定義が定まっていない

> WARNING
>
> GitOps を入れても、設計の曖昧さは消えない。 Git
> に何を置き、何を環境変数やシークレット管理へ逃がすかを決めないと、かえって複雑になる。

まとめ

GitOps は Kubernetes 運用の有力な型だが、すべての基盤の標準ではない。

必要な場所でだけ使うべきである。

10.8 AI 時代の CI/CD

> このセクションで答える問い: AI を CI/CD にどう組み込むべきか？

AI に向く作業

向く作業	理由
CI 失敗ログの要約	人間が原因にたどり着くまでの時間を短縮できる
PR 要約	変更点の把握を速められる
リリースノート草案	定型文の自動化と相性がよい
テスト候補の提案	影響範囲を考える補助になる
セキュリティ指摘の整理	ノイズを圧縮しやすい

AI を信用しすぎてはいけない作業

- 本番デプロイの自動承認

- セキュリティ修正の無審査適用

- ロールバック判断の全自動化

- 監査ログの要約だけに依存する判断

実務での原則

1. AI は助言者であり、最終承認者ではない

2. 入力データを制御する

3. 秘密情報を渡しすぎない

4. 出力を監査可能にする

5. 人間レビューの責任を消さない

> WARNING
>
> AI を CI/CD に入れるときの本当の論点は、精度より 権限 である。
> 書き込み権限、デプロイ権限、チケット操作権限をどこまで与えるかを先に決めるべきだ。

まとめ

AI は CI/CD を速くできるが、信頼境界を曖昧にすると危険も増える。

読む・要約する・提案する までは相性がよく、承認する・反映する は慎重に扱うべきである。

この章のチェックリスト

- [ ] CI と CD の違いを説明できるか
- [ ] CI/CD プラットフォームを SCM、運用責務、セキュリティで比較できるか
- [ ] Rolling、Blue-Green、Canary、Feature Flags の違いを説明できるか
- [ ] Feature Flags を認可や秘密情報の代わりに使ってはいけない理由を説明できるか
- [ ] テスト自動化で速い検査を先に置く理由を説明できるか
- [ ] アーティファクト管理で digest と保持期間が重要な理由を説明できるか
- [ ] GitOps が Kubernetes 向けの運用型であることを説明できるか
- [ ] AI を CI/CD に入れるとき、何を自動化して何を人間に残すべきか説明できるか

第11章 監視・オブザーバビリティ

この章を読む前に: 第7章〜第10章を読んでいると、デプロイ後に何を見続けるべきかがつながりやすい。

監視は「落ちたら気づくための仕組み」ではない。

本来の目的は、何が壊れそうか、実際にどこで壊れたか、なぜそうなったか を追える状態を作ることだ。

この章では、メトリクス・ログ・トレース、ツール選定、アラート設計、SLO、インシデント対応、AI時代の観測項目を整理する。

11.1 監視の3本柱

このセクションで答える問い: 何を集めればサービスの状態が分かるのか？

全体像

種類	何が分かるか	典型例
メトリクス	量の変化	エラー率、レイテンシ、CPU、キュー長
ログ	何が起きたか	例外、監査記録、業務イベント
トレース	どこで時間が使われたか	API → DB → 外部API の流れ

役割分担

- メトリクス: 異常を最初に見つける

- ログ: 異常の文脈を確認する

- トレース: 遅い箇所や依存先を特定する

よくある誤り

1. ログだけで何とかしようとする

2. メトリクスはあるが、誰も閾値を見直さない

3. トレースを入れたが、サンプリング設計がない

まとめ

3本柱は代替関係ではない。

検知、説明、経路特定 をそれぞれ分担している。

11.2 監視ツール選定

このセクションで答える問い: 監視基盤は何を基準に選ぶべきか？

主な選択肢

方式	代表例	強み	注意点
統合SaaS	Datadog, New Relic など	立ち上がりが速い。運用を減らしやすい	料金やベンダー依存を把握する必要がある
OSS 組み合わせ	Grafana, Prometheus, Loki, Tempo など	柔軟で自律性が高い	自分たちが運用する責務がある
エラー特化	Sentry など	エラー調査の初速が出る	全体観測の代わりにはならない

OpenTelemetry の位置づけ

OpenTelemetry は監視ツールそのものではなく、計測データを標準形式で扱うための共通基盤だ。

これを入れておくと、収集先や可視化先を後から変えやすくなる。

何で選ぶか

判断軸	見るべき点
導入速度	すぐ可視化したいか
自律性	ベンダー依存をどこまで許容するか
人員	監視基盤を自前運用できるか
既存基盤	既に使っているクラウドやSaaSと噛み合うか
コスト構造	ホスト課金、ログ量課金、ユーザー課金のどれが支配的か

実務的な整理

- まず早く始めたい: 統合SaaS

- 基盤を自分たちで握りたい: Grafana 系 + Prometheus 系

- 最初に最低限: エラー監視を先に入れる

- 将来の選択肢を残したい: OpenTelemetry を前提にする

> NOTE
>
> 監視ツール選定で本当に重要なのは、ダッシュボードの見た目ではない。 誰が運用するか と どの課金が膨らみやすいか の2点である。

まとめ

監視ツールは「何が一番高機能か」より、

自分たちが使い続けられるか と 収集コストを説明できるか で選ぶべきである。

11.3 アラート設計とオンコール体制

> このセクションで答える問い: いつ、誰に、どの強さで通知するべきか？

原則

アラートは「異常があること」ではなく、

今すぐ人間が対応すべきこと を知らせるためのものだ。

レベル分けの基本

レベル	例	通知先
Critical	サービス全体停止、主要決済失敗、認証不能	オンコール担当
Warning	エラー率上昇、遅延増加、キュー詰まり	担当チーム
Info	デプロイ完了、容量警告、日次集計	チーム共有

良いアラートの条件

1. 行動可能

2. 再現条件が明確

3. runbook がある

4. 誰が受けるか が決まっている

悪いアラートの例

- CPU が一瞬上がっただけで即通知

- 原因も手順も分からない

- 受信者が広すぎる

- 同じ内容が何度も来る

> WARNING
> アラート疲れは、本物の障害より危険になることがある。
> 反応しないアラートを増やすくらいなら、最初は少なく鋭くした方がよい。

オンコール体制

オンコールでは次を決めておく。

- 当番表
- エスカレーション順序
- 重大度の判定基準
- 初動手順
- 休日・夜間のルール

まとめ

アラート設計の目的は、通知を増やすことではない。

本当に起こしてよい通知だけを残すこと である。

11.4 LLM オブザーバビリティ

> このセクションで答える問い: LLM や AI エージェントでは何を監視するべきか？

従来監視と違う点

LLM は「落ちる」だけでなく、「もっともらしく間違う」。

そのため、インフラ監視だけでは不十分になる。

見るべき観点

観点	何を見るか
品質	期待した形式で返っているか、評価が落ちていないか
コスト	トークン数、呼び出し回数、再試行回数
レイテンシ	初回トークンまでの時間、応答完了までの時間
安全性	禁止応答、ツール誤呼び出し、権限逸脱
RAG	検索ヒット率、引用品質、再ランキングの効果

AI 特有の計測項目

- プロンプトテンプレートのバージョン

- モデル名とパラメータ

- ツール呼び出し回数

- 1 リクエストあたりの合計コスト

- 人手評価や自動評価の結果

ツールの考え方

方式	代表例	向いている場面
LLM 観測専用SaaS	LangSmith, Langfuse など	プロンプトやチェーンを追いたい
既存監視への統合	Datadog 系、OpenTelemetry 系	既存監視基盤に寄せたい
自前記録	DB + ログ + 評価基盤	厳密に制御したい

> WARNING
> 「幻覚率」を単純な1つの数値で正確に測れるとは限らない。
> 実務では、評価セット、人手確認、フォーマット逸脱率、ツール失敗率など複数の信号で見る方が現実的である。

まとめ

LLM 監視は、インフラ監視の上に 品質監視 を足す作業だと考えると整理しやすい。

11.5 SLI / SLO / SLA

> このセクションで答える問い: サービス品質の目標をどう定義するか？

用語

用語	意味
SLI	何を測るか
SLO	どこまでを目標にするか
SLA	顧客と何を約束するか

典型例

- SLI: 5分間の成功率

- SLO: 月間 99.9% 成功率

- SLA: 99.9% を下回った場合の契約上の扱い

可用性の目安

可用性	月間の許容停止時間
99.0%	約 7時間12分
99.5%	約 3時間36分
99.9%	約 43分
99.95%	約 21分
99.99%	約 4分

何を SLO にすべきか

SLO は CPU 使用率ではなく、ユーザー体験に近いものから決めた方がよい。

例:

- リクエスト成功率

- ログイン成功率

- 決済完了率

- 検索応答時間

> NOTE
>
> 「全部 99.99%」のような目標設定は危険だ。
> 高い目標は運用コストも大きく引き上げるため、ビジネス価値と釣り合っているかを確認する必要がある。

まとめ

SLO は監視の目標値であり、監視項目の一覧ではない。

何を守るか を先に決め、その後でダッシュボードを作るべきである。

11.6 インシデント対応とポストモーテム

> このセクションで答える問い: 障害が起きたらどう動くべきか？

基本フロー

```
graph TD
A[障害を検知] --> B[影響範囲を把握]
B --> C[暫定対応で止血]
C --> D[復旧確認]
D --> E[ポストモーテム]
E --> F[再発防止策]
```

初動でやること

1. 影響範囲を把握する

2. デプロイや直近変更を確認する

3. まず止血する

4. 役割分担を決める

ポストモーテムで書くこと

- 何が起きたか

- いつ気づいたか

- 影響は何だったか

- 何が復旧に効いたか

- 再発防止策は何か

> IMPORTANT
> ポストモーテムは犯人探しではない。
> 個人の失敗ではなく、システムがどう防げなかったかに焦点を当てるべきである。

AI をどう使うか

AI は次の補助には向いている。

- ログ要約

- 時系列の整理

- 初期仮説の列挙

- ポストモーテム草案の作成

ただし、次は人間が判断すべきである。

- 重大度

- 顧客影響

- ロールバックや遮断

- 再発防止策の優先順位

まとめ

インシデント対応では、完全な理解 より 安全な復旧 が先である。

原因分析は復旧後に行う。

11.7 オブザーバビリティの全体像

典型スタック

```
graph TB
A[アプリケーション] --> B[計測 SDK / Agent]
B --> C[Collector / Aggregator]
C --> D[Metrics Store]
C --> E[Log Store]
C --> F[Trace Store]
D --> G[Dashboard]
E --> G
F --> G
G --> H[Alerting]
H --> I[On-call]
```

重要な設計点

1. どこで収集するか

2. どこで加工するか

3. どこで保存するか

4. どれだけ保持するか

5. 誰が見るか

まとめ

オブザーバビリティはツールの名前を並べることではない。

収集、保存、可視化、通知、対応 が繋がって初めて機能する。

この章のチェックリスト

- [ ] メトリクス、ログ、トレースの違いを説明できるか
- [ ] 監視ツールを導入速度、自律性、運用人員、課金構造で比較できるか
- [ ] OpenTelemetry が監視ツールではなく計測の共通基盤であると説明できるか
- [ ] 行動可能なアラートとノイズアラートの違いを説明できるか
- [ ] LLM 観測で品質、コスト、レイテンシ、安全性を分けて説明できるか
- [ ] SLI、SLO、SLA の違いを説明できるか
- [ ] ポストモーテムが犯人探しでない理由を説明できるか
- [ ] オブザーバビリティが収集から通知までの一連の仕組みであると説明できるか

第12章 ソフトウェアテスト

この章を読む前に: フロントエンドまたはバックエンドの基本的な開発経験があると具体例と結びつけやすい。

テストの目的は、コードの正しさを証明することではない。

本来の目的は、安心して変更できる状態を作ること である。

この章では、テストの種類、ツール選定、カバレッジ、TDD、AI 生成コード時代の検証方針を整理する。

12.1 テスト戦略

このセクションで答える問い: どの種類のテストを、どこに厚く置くべきか？

基本の考え方

すべてを E2E で確認するのは遅く、壊れやすい。

逆に、ユニットテストだけでは実際の接続不具合を拾えない。

そのため、次のように役割を分けて考える。

層	何を見るか	特徴
ユニット	ロジック単体	速い
インテグレーション	複数部品の接続	現実に近い
E2E	ユーザー視点の主要フロー	遅いが安心感が高い

原則

1. ユニットテストを土台にする

2. 重要な境界にはインテグレーションテストを置く

3. 主要業務フローだけ E2E で押さえる

判断フロー

```
graph TD
A[対象機能] --> B{純粋なロジックか}
B -->|はい| C[ユニットテスト]
B -->|いいえ| D{外部依存やDBがあるか}
D -->|はい| E[インテグレーションテスト]
D -->|いいえ| F{ユーザー導線として重要か}
F -->|はい| G[E2Eテスト]
F -->|いいえ| H[ユニット中心で十分]
```

まとめ

「何でも E2E」も「何でもモック」も偏っている。

重要なのは、壊れたときに困る場所 に応じてテスト種別を配置することだ。

12.2 テストの種類

> このセクションで答える問い: ユニット、インテグレーション、E2E はどう違うのか？

比較

種類	何を確認するか	外部依存	典型例
ユニット	関数やクラスの振る舞い	基本的に持たない	税計算、バリデーション
インテグレーション	部品同士の接続	DB、キュー、外部API など	API と DB の保存確認
E2E	ユーザーの操作全体	本番に近い構成	ログインから購入完了まで
契約テスト	サービス間のインターフェース	API 境界	BFF とバックエンドの契約確認

モックの使い方

モックは便利だが、増やしすぎると本物とのズレ を隠す。

良い使い方:

- 外部課金APIの高額呼び出しを避ける

- 時刻や乱数を安定化する

- 失敗条件を意図的に作る

悪い使い方:

- DB 接続や HTTP 境界を全部モックで済ませる

- 実際のシリアライズ差分を見ない

- 契約不整合をモックで隠す

まとめ

モックはテストを速くするが、現実そのものにはしない。

境界の一部は本物で確認する必要がある。

12.3 テストフレームワーク選定

このセクションで答える問い: 何のツールを使ってテストを書くべきか？

主な選択肢

分野	主な選択肢	向いている場面
TypeScript / JavaScript	Vitest, Jest	フロントエンド、Node.js
Python	pytest	API、データ処理
Go	testing, testify など	Go サービス
ブラウザ E2E	Playwright, Cypress	Web UI

TypeScript / JavaScript

ツール	強み	注意点
Vitest	Vite 系との相性がよい。モダン構成に馴染みやすい	既存の Jest 資産が多い案件では移行コストがある
Jest	長く使われており情報が多い	ESM やモダン構成では追加調整が必要な場面がある

E2E

ツール	強み	注意点
Playwright	複数ブラウザ、並列実行、トレース確認がしやすい	機能が多く、最初は覚えることが多い
Cypress	学習導線が分かりやすい	要件次第では制約を感じやすい

選び方

- 新規の TypeScript 系: Vitest

- 既存 Jest 資産が多い: Jest 継続

- 新規 E2E: Playwright を優先検討

- 既存 Cypress 資産が多い: Cypress 継続 も合理的

> NOTE
>
> テストツールは流行より既存資産との整合性が重要である。 既に十分動いている Jest や Cypress を、性能比較だけで無理に移行する必要はない。

まとめ

新規ならモダンな選択肢が有力だが、

既存プロジェクトでは移行コストも含めて判断すべきである。

12.4 TDD / BDD の実践的な位置づけ

> このセクションで答える問い: TDD や BDD はいつ有効か？

TDD

TDD は次のサイクルで進める。

1. 失敗するテストを書く

2. 最小限の実装で通す

3. リファクタする

向いている場面:

- ロジックが複雑

- 入出力が明確

- ライブラリやドメインモデルを固めたい

BDD

BDD は、要件を振る舞いとして整理したいときに役立つ。

特に業務フロー、承認、権限、シナリオ分岐が多いときに有効である。

注意点

> WARNING
> TDD も BDD も教義ではない。 書く価値が高い場所に使うべきで、すべてのコードに機械的に適用するものではない。

まとめ

TDD はロジックを固める道具、BDD は要件を揃える道具である。

万能ではないが、刺さる場面では強い。

12.5 カバレッジの考え方

このセクションで答える問い: カバレッジはどこまで目指すべきか？

まず押さえるべきこと

カバレッジは品質そのものではない。

通った行が多いことと、重要な欠陥を防げていることは別問題である。

現実的な使い方

使い方	説明
差分確認	新規変更で未テスト箇所が増えていないかを見る
リスク判定	重要ロジックが薄いままになっていないかを見る
チーム対話	どこが未検証かを共有する

何が危険か

- 100% だけを目的にする

- スナップショットだけでカバレッジを稼ぐ

- 重要ロジックより薄い枝葉を大量にテストする

> WARNING
>
> 高いカバレッジでも、仕様の抜けや接続不良は普通に残る。
> カバレッジは「見えていない場所の地図」として使う方が実務的である。

まとめ

カバレッジは目標値より、重要箇所が本当に検証されているか を見るために使うべきである。

12.6 AI 時代のテスト戦略

このセクションで答える問い: AI 生成コードが増えたとき、テストは何を変えるべきか？

何が変わるか

AI がコードを書くようになると、コード量は増えやすい。

その一方で、次の問題も増える。

- 見た目は正しいが境界条件が甘い

- モックに依存したテストが増える

- 変更範囲が広いのにレビューが追いつかない

- 同じような実装が複数箇所に散る

対応方針

1. インテグレーションテスト を厚くする

2. 差分ベースのレビュー を強める

3. 契約テスト や 境界テスト を重視する

4. AI が書いたテスト もレビューする

5. セキュリティスキャン を一緒に回す

AI に向くこと

- テストケースのたたき台生成

- 境界条件の候補出し

- 失敗時ログの要約

- 重複したテストの整理案

AI に任せすぎてはいけないこと

- テストの妥当性判断

- 重要業務フローの優先順位付け

- セキュリティの最終判断

- 本番障害の再現確認

> WARNING
> AI が生成したテストは「ある程度もっともらしい」ことが多い。だからこそ危険で、本当に何を検証しているかを人間が読まないと意味がない。

まとめ

AI 時代に必要なのは、テストを減らすことではない。

何を AI に補助させ、何を人間が保証するか の境界を明確にすることである。

この章のチェックリスト

- [ ] ユニット、インテグレーション、E2E、契約テストの違いを説明できるか
- [ ] 重要な境界にインテグレーションテストを置く理由を説明できるか
- [ ] モックの使いすぎが危険な理由を説明できるか
- [ ] 新規プロジェクトと既存プロジェクトでツール選定の考え方が違うことを説明できるか
- [ ] TDD と BDD が有効な場面を説明できるか
- [ ] カバレッジが品質そのものではない理由を説明できるか
- [ ] AI 生成コードではなぜ境界テストとインテグレーションテストが重要になるか説明できるか
- [ ] AI 生成テストをそのまま信用してはいけない理由を説明できるか

第13章 セキュリティ

この章を読む前に: バックエンド（第4章）の基本概念、特に認証・認可を把握していること。

セキュリティは「あとで強化する」ものではなく、最初から設計に含めるべき品質だ。

公開されたWebアプリケーションやAPIは、規模に関係なく常時スキャンされる。

AI機能を組み込む場合は、従来のWebセキュリティに加えて、プロンプト注入や過剰なツール実行のような新しい攻撃面も増える。

13.1 Webアプリケーションセキュリティの基礎とLLM固有リスク

このセクションで答える問い: どの脅威を優先して考えるべきか。AI機能を入れると何が増えるのか。

なぜ必要か

脆弱性の多くは「難しい暗号の失敗」ではなく、認可漏れ、入力値の扱い、設定ミス、古い依存関係の放置といった実装ミスから生まれる。まずは代表的な失敗パターンを理解する。

全体像

OWASP Top 10 の現行の安定版は 2021 であり、Webアプリケーションの代表的な失敗を整理している。

まず優先して押さえるべき項目は次の通り。

脅威	典型例	基本対策
認可の不備	他人の注文やファイルにアクセスできる	サーバー側で毎回認可判定。ID直指定を信用しない
インジェクション	SQL、コマンド、テンプレートへの注入	パラメータバインド。シェル連結禁止。危険APIの限定
設定ミス	デバッグ機能の公開、不要ポート、緩いCORS	本番設定の分離。最小権限。公開面の棚卸し
脆弱な依存関係	既知の脆弱性を持つライブラリを放置	継続的な更新とスキャン。ロックファイル管理
識別・認証の失敗	弱いセッション管理、平文パスワード、MFA不備	実績ある認証基盤の利用。セッション保護。MFA/パスキー
ログと監視の不足	侵害に気づけない、追跡できない	監査ログ、異常検知、運用手順の整備

LLM機能を入れると増える攻撃面

OWASP Top 10 for LLM Applications 2025 では、Prompt Injection が LLM01 として扱われている。

LLMは「命令」と「データ」を完全には分離できないため、悪意ある入力や文書を処理したときに

想定外の動作を起こしうる。

リスク	例	基本対策
直接プロンプト注入	チャット入力で「前の指示を無視して秘密を出せ」と誘導	権限を持たせすぎない。重要操作は人間承認
間接プロンプト注入	WebページやPDF内の隠れた命令をLLMが読む	取得データを不信任入力として扱う。ツール利用を制限
過剰なエージェンシー	LLMがメール送信、課金、削除を勝手に実行	ツールごとの許可制、スコープ制限、監査ログ
不適切な出力処理	LLM出力をそのままHTMLやSQLに流し込む	出力を再検証。構造化出力でも最終検証を入れる
機密情報の露出	システムプロンプトや社内文書の漏えい	参照可能データの最小化。マスキング。応答監査

2025年版の分類で押さえるべき観点

OWASP の 2025 年版では、単に「プロンプト注入」だけでなく、

周辺の設計不備も独立したリスクとして整理されている。

特に実装で見落としやすいのは次の4つ。

項目	何が問題か	設計上の意味
LLM02 Sensitive Information Disclosure	モデルや周辺システムから機密情報が漏れる	参照可能データを絞る。応答を監査する
LLM05 Improper Output Handling	出力を信用しすぎて下流で実行してしまう	HTML、SQL、コマンド、メール本文は再検証する
LLM06 Excessive Agency	ツール権限が強すぎる	読み取り専用と更新系を分ける。高リスク操作は承認制にする
LLM08 Vector and Embedding Weaknesses	RAGの検索基盤自体が攻撃面になる	文書投入経路、メタデータ、検索結果の扱いを設計する

プロンプト注入への考え方

> IMPORTANT
>
> プロンプト注入は「完全防御」よりも「被害を小さくする設計」が重要。
> LLMを信頼境界の内側に置かず、誤動作しても致命傷にならない構造にする。

実務では次の順番で守る。

1. LLMに強い権限を与えない

2. 高リスク操作は人間承認にする

3. 外部文書や検索結果を不信任入力として扱う

4. ツール呼び出しを明示的な許可制にする

5. 入出力とツール実行を監査ログに残す

まとめ

まずは従来のWeb脅威を確実に潰す。AI機能を入れる場合は、

「LLMが間違っても壊れない構造」を追加で設計する。

13.2 認証・認可の実装とパスキー

> このセクションで答える問い: 認証と認可をどう分けて設計するか。パスキーはいつ使うべきか。

なぜ必要か

侵害事故では「認証方式そのもの」よりも、認可漏れやセッション管理の不備が原因になることが多い。

本人確認と権限判定を分けて考えないと、ログインできているのに他人のデータへアクセスできる、

という事故が起きる。

全体像

項目	意味	失敗例	対策
認証	その人が誰かを確認する	平文パスワード、弱いMFA、雑なセッション管理	実績ある認証基盤、MFA、パスキー
認可	何をしてよいかを決める	他人のデータ更新、管理者機能の誤公開	サーバー側で毎回権限判定。拒否を既定値にする
セッション管理	ログイン状態を安全に維持する	トークン漏えい、Cookie設定不備	HttpOnly Secure SameSite、期限管理、失効処理

パスキーの位置づけ

パスキーは FIDO2 / WebAuthn を使った認証方式で、パスワードの代替として普及している。

秘密鍵は利用者のデバイス側に保持され、サーバーは公開鍵を使って検証する。

そのため、パスワード入力型よりフィッシング耐性が高い。

実装上のポイントは次の通り。

観点	実務上の判断
導入先	新規サービスでは最初から候補に入れる。既存サービスではパスワードと並行導入でもよい
技術基盤	可能なら Auth0、Okta、Cognito など既存基盤を使う

観点	実務上の判断
ブラウザ要件	WebAuthn は安全なコンテキスト（通常は HTTPS）で使う
復旧導線	デバイス紛失時の復旧手順を必ず設計する
併用方針	パスワード廃止を急がず、移行期間を設けてもよい

パスキー管理画面で持つべき情報

MDN や passkeys.dev の実装ガイドを見ると、導入より運用UIの方が軽視されやすい。

最低でも次を表示できると、紛失時の対応がしやすい。

- 登録済みパスキーの一覧

- それぞれの名前や利用端末の手がかり

- 最終利用時刻

- バックアップ可能か、すでにバックアップ済みか

- 個別削除の導線

複数パスキー前提で設計する

> TIP
> パスキーでは「1アカウントに1認証器」ではなく、複数パスキーを持てる設計が自然。
> 日常利用する端末に加えて、別の認証器を予備として登録できるようにする。

日常用のプラットフォーム認証器に加えて、別端末やローミング認証器を

予備として登録できるようにしておくと、アカウント復旧の多くを自己解決できる。

discoverable credentials の考えどころ

ユーザー名入力なしでサインインさせたい場合は discoverable credentials が便利だが、

ハードウェア認証器では保存数の上限や、認証器上に識別情報が残ることを考える必要がある。

一般利用では便利さが勝つことが多いが、高い秘匿性を求めるケースでは設計判断が要る。

パスキー導入時の注意点

> IMPORTANT
> パスキーは強力だが、アカウント復旧まで含めて設計しないと運用で詰まる。 1台の端末だけに依存させない。

最低限、次を用意する。

1. 複数デバイスまたは複数認証器の登録

2. メール確認や本人確認を伴う復旧フロー

3. 管理画面での認証器一覧と無効化機能

認可で外してはいけない点

- UIでボタンを隠しても認可にはならない

- APIごとにサーバー側で権限判定する

- オブジェクト単位の認可を行う

- 管理者権限は役割名ではなく操作単位で見直す

まとめ

認証はできるだけ既存基盤を使い、認可は自分たちの業務ルールに合わせて丁寧に実装する。

パスキーは有力な選択肢だが、導入時は復旧導線まで必ず含める。

13.3 シークレット管理

> このセクションで答える問い: APIキーや秘密鍵をどこに置き、漏えいしたらどう動くべきか。

なぜ必要か

シークレットは漏えいした瞬間に被害が始まる。しかも、コードに埋め込むと Git 履歴やログ、

チャット、CI出力にまで広がりやすい。

全体像

方法	向いている場面	注意点
.env	ローカル開発	Git管理しない。配布はテンプレートのみ
CI/CD シークレット	ビルドやデプロイ	権限を最小化。出力に出さない
クラウドのシークレット管理	ステージング・本番	監査ログ、ローテーション、アクセス制御を使う
Vault系製品	大規模、動的シークレット	運用コストに見合う場合のみ採用

実務での原則

1. .env はローカル限定にする

2. .env.example には値を入れない

3. 本番ではシークレット管理サービスを使う

4. ログ、例外、画面にシークレットを出さない

5. 漏えい時は履歴修正より先に失効とローテーションを行う

スキャンの自動化

GitHub などのホスティング基盤には secret scanning や push protection がある。

検出に頼り切らず、そもそも「長寿命の秘密を増やさない」「人手で配布しない」設計にする。

secret scanning と push protection の違い

仕組み	役割
secret scanning	すでにリポジトリや履歴に入った秘密を検出する
push protection	push の時点で秘密の混入を検知し、そもそも入れにくくする

実務では両方必要だが、優先すべきは push protection のような混入前ブロックだ。

検出だけでは履歴、CI、フォークへの拡散を止められない。

まとめ

ローカルは .env、本番はシークレット管理サービス、漏えい時は即ローテーション。

シークレットは「どこに置くか」だけでなく「漏れたら何分で止められるか」で評価する。

13.4 依存関係管理とサプライチェーンセキュリティ

> このセクションで答える問い: ライブラリ、ビルド、生成物のどこを守るべきか。

なぜ必要か

現代のアプリケーションは大量の依存関係と外部サービスの上に成り立っている。

自分で書いたコードが安全でも、依存ライブラリやビルド経路が侵害されれば同じ被害になる。

全体像

対象	典型的なリスク	基本対策
アプリ依存関係	既知脆弱性、悪意ある更新	Dependabot / Renovate、ロックファイル、検証付き更新
CI/CD	侵害されたアクション、漏えいした権限	短命資格情報、OIDC、第三者アクションの固定
生成物	何から作られたか追跡できない	署名、ハッシュ、SBOM、保存ポリシー

対象	典型的なリスク	基本対策
コンテナ	ベースイメージの脆弱性、過剰権限	最小イメージ、定期再ビルド、イメージスキャン
AI開発物	生成コードやモデルの由来が曖昧	生成物レビュー、評価セット、採用経路の記録

Dependabot / Renovate の使い分け

ツール	向いている場面
Dependabot	GitHub中心で、まず脆弱性通知と基本更新を回したい
Renovate	更新ルールを細かく制御したい。モノレポや複数エコシステムをまとめたい

SBOM の役割

SBOM（Software Bill of Materials）は「このアプリに何が入っているか」を一覧化する文書だ。

脆弱性の特定、規制対応、インシデント時の影響調査で役立つ。

代表的な形式は SPDX と CycloneDX。

SLSA の理解で注意する点

> IMPORTANT
>
> 古い資料にある「SLSA Level 1〜4 の単純な一直線の表」は、現在の理解としては粗い。 現行の SLSA は複数トラックの要求を組み合わせて評価する考え方になっている。

2026年3月15日時点の現行版は SLSA v1.2 で、少なくとも次のトラックを押さえると実務に結びつきやすい。

トラック	何を見るか	まず効く場面
Build Track	生成物の来歴、ビルド、検証	コンテナ、バイナリ、パッケージ配布
Source Track	ソース管理、履歴、技術的統制、レビュー	GitHub / GitLab などの開発フロー

Source Track の簡略イメージは次の通り。

レベル	焦点
Source L1	バージョン管理されている
Source L2	履歴と source provenance が残る
Source L3	組織の技術的統制を継続適用する
Source L4	二者レビューを要求する

実務では、細かい等級名を暗記するより、次の順で整備する方が重要。

1. ビルドを手元作業ではなく自動化する

2. 生成物の来歴を残す

3. 署名や検証を入れる

4. デプロイ前に検証を自動化する

5. ビルド環境の権限を分離する

AI生成コードをどう扱うか

AI生成コードは「信用できない下書き」として扱う。

特に次は人間が必ず見る。

- 認証・認可

- SQLやテンプレート処理

- 暗号化と鍵管理

- 外部APIや課金処理

- エラーハンドリングとタイムアウト

まとめ

依存関係、CI/CD、生成物をまとめてサプライチェーンとして見る。

更新の自動化だけでは足りず、来歴、署名、検証までつなげて初めて運用になる。

13.5 ブラウザ防御とHTTPヘッダ

> このセクションで答える問い: アプリケーション本体以外で何を守れるか。

全体像

項目	役割	補足
Content-Security-Policy	スクリプトや読み込み元を制限する	XSS対策の中心。最初は report-only からでもよい
Strict-Transport-Security	HTTPS利用を強制する	HTTP運用を残さない
X-Content-Type-Options: nosniff	MIME推測を防ぐ	静的配信でも入れておく
Cookie属性	セッション保護	HttpOnly Secure SameSite を基本にする
CSRF対策	意図しない送信を防ぐ	SameSite だけで済ませず、必要ならトークン併用
フレーム埋め込み制御	クリックジャッキング対策	frame-ancestors や X-Frame-Options を使う

CSP の考え方

CSP は「雑に全部禁止して終わり」ではない。まずは実際の読み込み元を整理し、

不要なインラインスクリプトや野良スクリプトを減らす。CSPを後付けで入れるのではなく、

最初から守りやすいフロントエンド構成にする方が楽だ。

まとめ

HTTPヘッダは低コストで効果が大きい。CSP、HSTS、Cookie属性は早い段階で整える。

13.6 セキュリティテストと運用

> このセクションで答える問い: セキュリティをどう継続的に確認するか。

全体像

手法	何を見るか	タイミング
SAST	ソースコードの危険パターン	開発中、CI
DAST	動作中アプリへの疑似攻撃	ステージング、定期診断
依存スキャン	既知脆弱性	CI、定期実行
Secret scanning	秘密情報の混入	コミット時、PR時
IaC / コンテナスキャン	インフラ設定とイメージ	デプロイ前

運用で必要なこと

1. 重要操作と認証イベントを監査ログに残す

2. アラートを誰が受け、どう初動するか決める

3. 漏えい時のローテーション手順を文書化する

4. AI機能には評価セットとレッドチーム観点の確認を入れる

> WARNING
> スキャンツールを入れただけでは安全にならない。
> 検出結果を誰が見て、どの期限で直すかまで決めて初めて機能する。

まとめ

セキュリティは単発の診断ではなく継続運用だ。

CIの自動検査、監査ログ、インシデント手順をつなげて回す。

この章のチェックリスト

- [ ] 認証と認可の違いを説明できるか
- [ ] 認可漏れ、インジェクション、設定ミス、依存脆弱性の違いを説明できるか
- [ ] プロンプト注入の直接・間接パターンを区別できるか
- [ ] パスキー導入時に復旧導線が必要な理由を説明できるか
- [ ] .env を Git 管理しない理由と、本番での保管先を説明できるか
- [ ] SBOM の役割を説明できるか
- [ ] SLSA を単なる「4段階表」ではなく、来歴と検証の仕組みとして理解しているか
- [ ] CSP、HSTS、Cookie属性の役割を説明できるか
- [ ] SAST、DAST、依存スキャン、secret scanning の役割を使い分けられるか

第14章 ソフトウェアアーキテクチャ

この章を読む前に: バックエンド（第4章）とデータベース（第5章）の基礎を理解していること。

アーキテクチャは「あとで綺麗にするための飾り」ではなく、

チームがどの速度で変更し続けられるかを決める設計だ。

重要なのは流行の名前を知ることではなく、要件と運用能力に対して

過不足のない構造を選ぶことにある。

14.1 モノリス、モジュラーモノリス、マイクロサービス

このセクションで答える問い: どこまで分割すべきか。いつ分けるべきか。

なぜ必要か

分割は自由度を増やす一方で、通信、監視、デプロイ、障害切り分けのコストも増やす。

「小さく分けるほど良い」は誤りで、実際にはチームの運用能力が上限になる。

全体像

形	特徴	向いている場面	主なコスト
モノリス	1つのアプリケーションとして動く	初期開発、小〜中規模、少人数チーム	変更範囲が広くなりやすい
モジュラーモノリス	1つのデプロイ単位だが内部境界が明確	ドメインが増えてきたが、運用は単純に保ちたい	境界を守る規律が必要
マイクロサービス	サービスごとに分離し独立運用する	複数チームの独立性が重要、部分ごとに性質が大きく異なる	通信、整合性、運用が複雑

判断軸

観点	モノリス寄り	マイクロサービス寄り
チーム構成	1チーム中心	複数チームが独立している
デプロイ要求	同時リリースで問題ない	独立リリースが必須
データ整合性	強い整合性を単純に保ちたい	境界ごとの整合性で割り切れる
運用体制	SRE/運用人数が限られる	監視、CI/CD、障害対応を分散で回せる
性能特性	全体がほぼ同じ性質	一部だけ極端に高負荷・高信頼性が必要

> IMPORTANT
> 迷ったら モノリスかモジュラーモノリスから始める。 先に分散すると、組織が抱えられない複雑さだけが増えやすい。

段階的な進め方

1. 最初はモノリスで素早く仕様を固める

2. 変更が頻発する領域ごとにモジュール境界を明確化する

3. 独立デプロイが必要になった部分だけ外に切り出す

まとめ

分割は目的ではなく手段だ。チームの独立性と運用能力が揃うまでは、

一体で動く構造の方が強い。

14.2 レイヤー設計と依存方向

> このセクションで答える問い: コードをどう整理すると変更に強くなるか。

なぜ必要か

機能が増えると、画面、業務ルール、データアクセス、外部API呼び出しが混ざりやすい。

この混在を放置すると、小さな変更でも副作用が読めなくなる。

全体像

クリーンアーキテクチャ、ヘキサゴナルアーキテクチャ、オニオンアーキテクチャは

名前こそ違うが、狙いは近い。

共通する要点:

1. ビジネスルールを中心に置く

2. DBや外部APIなどの詳細は外側に置く

3. 依存方向を内側へ向ける

実装の最小形

小〜中規模では、次の3層で十分なことが多い。

層	責務
入口層	HTTP、CLI、ジョブ起動など外部との接点
業務層	業務ルール、ユースケース、検証
データ層	DB、キャッシュ、外部サービスとの接続

例:

```
src/
├── features/
│   ├── orders/
│   │   ├── controller.ts
│   │   ├── service.ts
│   │   └── repository.ts
│   └── users/
├── shared/
└── config/
```

抽象化しすぎない

> WARNING
> 単純なCRUDに対して、ポート、アダプタ、ユースケース、DTO、ファクトリを過剰に分けると、
> 変更しやすくなる前に読みにくくなる。

次の場合だけ抽象化を強める価値がある。

- 業務ルールが複雑

- 外部依存を差し替えたい

- テストや再利用の都合で境界が明確に必要

まとめ

アーキテクチャ名より、依存方向と責務分離の方が重要。

まずは小さなレイヤー分離から始める。

14.3 同期処理、非同期処理、イベント駆動

このセクションで答える問い: 直接呼び出すべきか、キューやイベントに流すべきか。

なぜ必要か

すべてを同期APIでつなぐと、1つの遅延や障害が全体に連鎖する。

一方で、最初から非同期やイベント駆動に寄せすぎると、整合性とデバッグが難しくなる。

全体像

方式	向いている場面	注意点
同期呼び出し	すぐ返答が必要。処理の流れが単純	下流障害が上流に波及しやすい
ジョブキュー	重い処理、再試行が必要、後で結果を返せばよい	状態管理と重複実行対策が必要
イベント駆動	複数の関係者が同じ出来事に反応する	順序、再送、最終整合性を理解する必要がある

最初に入れるべき非同期化

次は同期APIから切り出しやすい。

- メール送信

- 画像変換

- レポート生成

- 外部通知

- AI推論やバッチ評価

CQRS と Event Sourcing の位置づけ

CQRS は読み取りモデルと書き込みモデルを分ける考え方、

Event Sourcing は状態変化をイベントとして保存する考え方だ。

どちらも強力だが、常に必要ではない。

向いているのは次のような場合。

- 監査証跡が重要

- 複数の読み取りビューが必要

- 状態変化の履歴そのものが価値を持つ

まとめ

最初は同期でよい。遅い処理と副作用から順に非同期化する。

イベント駆動は、疎結合の利益が運用複雑性を上回る場面で使う。

14.4 モノレポとポリレポ

このセクションで答える問い: コードベースを1つのリポジトリにまとめるべきか。

全体像

戦略	向いている場面	注意点
モノレポ	共有コードが多い。同じチームが複数アプリを触る	CIや依存管理の整理が必要
ポリレポ	リリース単位が独立している。権限境界を分けたい	共有コードの更新が分散しやすい

判断軸

- フロントエンドとバックエンドで型やスキーマを共有したいならモノレポが有利

- チームや権限が明確に分かれているならポリレポの方が素直

- まずはリポジトリ数ではなく、リリース単位とレビュー導線で決める

ツールの位置づけ

モノレポでは、パッケージマネージャの workspaces に加えて、

必要ならタスク実行やキャッシュの仕組みを足す。

Node.js 系では pnpm workspaces、Turborepo、Nx などが代表例だが、

どれを使うかより「依存境界を壊さない」「不要な全体ビルドを避ける」方が重要。

まとめ

モノレポかポリレポかに絶対の正解はない。

共有変更のしやすさと独立運用のしやすさのどちらを優先するかで決める。

14.5 クライアントごとのAPI設計: BFF と GraphQL

> このセクションで答える問い: Webとモバイルで必要なデータが違うとき、APIをどう組むか。

なぜ必要か

クライアントが増えると、共通APIだけでは過剰取得や不足取得が起こりやすい。

画面ごとの都合でバックエンドが引きずられると、変更コストが上がる。

全体像

手法	向いている場面	注意点
共通REST API	クライアントが少ない。要求が似ている	画面最適化には限界がある

手法	向いている場面	注意点
BFF	Webとモバイルで要求が大きく違う	BFFごとに責務が膨らみやすい
GraphQL	取得項目の柔軟性が必要	N+1、認可、スキーマ運用に注意

判断の仕方

1. まずは共通APIで始める

2. クライアント差分が大きくなったらBFFを検討する

3. データ取得の柔軟性が支配的な課題ならGraphQLも候補に入れる

まとめ

BFFは「複数クライアントの都合を整理するための層」であり、最初から必須ではない。

14.6 ドメイン駆動設計の要点

> このセクションで答える問い: 複雑な業務ルールをどう整理するか。

なぜ必要か

業務が複雑になると、画面やDB中心の設計だけでは言葉がずれ始める。

同じ「注文」でも、販売、配送、請求で意味が違うことがある。

全体像

DDD の重要な点は、パターンを全部導入することではなく、

業務の境界に沿ってモデルを分けることだ。

概念	要点
境界づけられたコンテキスト	業務上の文脈ごとにモデルを分ける
ユビキタス言語	業務側と開発側で同じ言葉を使う
集約	一貫性を守る単位を定める

まず取り入れるとよいもの

1. 用語を揃える

2. コンテキストごとにフォルダやモジュールを分ける

3. 更新の一貫性をどこで守るか決める

まとめ

DDD は複雑なドメインで効く。小さなCRUDで無理にフルセット導入する必要はない。

14.7 AI機能を前提にしたアーキテクチャ

> このセクションで答える問い: LLMやエージェントを入れると、設計のどこが変わるのか。

なぜ必要か

AI機能は通常のAPIより遅く、高価で、非決定的で、外部依存が強い。

そのため、従来の同期的なCRUD設計をそのまま当てると扱いづらい。

全体像

論点	設計の要点
応答時間	すぐ返す処理と、待たせてよい処理を分ける
コスト	キャッシュ、再試行制御、モデル切り替えを考える
安全性	ツール実行権限を絞り、人間承認を入れる
監査性	入力、参照文書、モデル、出力、ツール実行を追えるようにする
評価	通常テストに加えて評価セットを維持する

実務での定石

- AI推論は同期APIに詰め込みすぎない

- 長い処理はジョブ化し、進捗取得やストリーミングを使う

- モデル呼び出しをアプリ全体に散らさず、境界を作る

- マルチエージェントは単一エージェントで足りなくなってから導入する

> IMPORTANT
>
> マルチエージェントは「高度だから優れている」のではない。
> 追加されるのは自由度だけでなく、状態管理、監査、障害解析の複雑さでもある。

まとめ

AI機能を入れると、性能よりも非決定性と運用性が設計の中心課題になる。

単一モデル、単一実行経路で足りるなら、そこから広げない方が強い。

この章のチェックリスト

- [ ] モノリス、モジュラーモノリス、マイクロサービスの違いを説明できるか
- [ ] 分割判断をチーム構成、運用能力、整合性要件で考えられるか
- [ ] レイヤー設計で依存方向をどう揃えるべきか説明できるか
- [ ] 同期、ジョブキュー、イベント駆動の使い分けを説明できるか
- [ ] モノレポとポリレポをリリース単位と共有コードの観点で選べるか
- [ ] BFF と GraphQL の役割の違いを説明できるか
- [ ] DDD の境界づけられたコンテキストの意味を説明できるか
- [ ] AI機能を入れるときに、応答時間、コスト、安全性、監査性をどう設計に入れるか説明できるか

第15章 チーム開発のプラクティス

この章を読む前に: Git の基本（第1章1.4節）を理解していること。

良いチームは、個々のエンジニアが特別に優秀だから強いのではなく、

同じ判断を何度でも再現できるから強い。

ブランチ運用、レビュー、ドキュメント、オンボーディングといった地味な仕組みが、

開発速度と品質の上限を決める。

15.1 Git ブランチ戦略

このセクションで答える問い: どのブランチ運用を選ぶべきか。どこまでルールを増やすべきか。

なぜ必要か

チーム開発では、変更を安全に統合する仕組みが必要だ。

ブランチ戦略が曖昧だと、レビュー抜け、競合、リリース事故が増える。

全体像

戦略	向いている場面	注意点
GitHub Flow	小〜中規模チーム。継続的デプロイ	main の健全性をCIで守る必要がある
Trunk-Based Development	小さな変更を高頻度で統合できるチーム	テスト、自動化、Feature Flag が弱いと破綻しやすい
リリースブランチ併用	リリース承認や保守版管理が必要	ブランチ寿命が長くなりすぎないようにする

実務での原則

1. ブランチは短命にする

2. PRは小さく分ける

3. main への直接コミットは原則避ける

4. マージ条件をCIとレビューで明確にする

5. 長寿命ブランチで仕様差分を溜めない

迷ったときの基準

> TIP
> 迷ったら GitHub Flow で十分なことが多い。ルールを増やすのは、既存の運用で明確に困ってからでよい。

まとめ

ブランチ戦略は複雑さではなく、統合頻度とリリースのしやすさで選ぶ。

まずは短命ブランチと小さなPRを徹底する。

15.2 コードレビューの文化と進め方

このセクションで答える問い: レビューを形骸化させず、品質向上に結びつけるにはどうするか。

なぜ必要か

コードレビューの目的は承認スタンプではない。

バグの早期発見、一貫性の維持、知識共有が主な価値だ。

全体像

原則	内容
小さなPR	変更量を絞るとレビュー精度が上がる
観点を揃える	正しさ、保守性、性能、セキュリティを見る
具体的に指摘する	「ダメ」ではなく、理由と代案を書く
期限を決める	レビュー待ちを放置しない
学習に使う	背景や設計意図を説明する

レビューで最低限見る項目

1. 仕様どおりに動くか
2. 認証・認可や入力検証に穴がないか
3. 失敗時の挙動が定義されているか
4. テストが本当に意図を検証しているか
5. 変更後に運用で困らないか

AIを使う場合の注意点

AIによる要約やレビュー補助は有効だが、最終判断は人間が持つ。

特に次は自動コメントだけで済ませない。

- 認可

- 課金

- データ削除

- 外部公開API

- セキュリティ設定

> WARNING
> LGTM だけ返すレビューは、やらないのと大差がない。変更を読んだ形跡が残るレビューを行う。

まとめ

レビュー品質は、PRサイズ、観点の明確さ、応答速度で決まる。

AIは補助に使い、人間は責任のある判断に集中する。

15.3 ドキュメンテーション

> このセクションで答える問い: 何を文書化し、どこまで残すべきか。

なぜ必要か

コードは「何をしているか」は示せても、「なぜそうしたか」は残しにくい。

背景が残っていないと、同じ議論と同じ失敗を繰り返す。

最低限必要な文書

種別	内容	置き場所
README	概要、起動方法、主要コマンド	リポジトリ直下
ADR	設計判断と理由	docs/adr/
運用手順書	障害対応、ローテーション、再起動手順	docs/ または運用基盤
API仕様	エンドポイント、入力、出力、制約	OpenAPI など

ADR の最小テンプレート

```
# ADR-001: 認証基盤の選定

## 状態
```

```
採用

## 背景
何に困っていたか。何を比較したか。

## 決定
何を選んだか。

## 理由
なぜそれを選んだか。

## 影響
得るものと失うもの。
```

README に最低限入れるもの

1. このリポジトリが何か
2. 起動手順
3. 必要な環境変数
4. テスト方法
5. よくある詰まりどころ

AI時代に文書の価値が上がる理由

AI支援を使うと、ドキュメントは人間向けだけでなく、

将来の自分やAIへのコンテキストにもなる。

ただし、文書が古いと誤誘導の原因になるので、量より更新性を優先する。

まとめ

README と ADR を最小限でよいので残す。

書くことより、コード変更に合わせて更新されることの方が重要だ。

15.4 開発プロセスの選び方

> このセクションで答える問い: スクラム、カンバンなどをどう選び、どう軽く運用するか。

全体像

方式	向いている場面	注意点
スクラム	一定期間で成果を区切りたい	儀式だけ増やすと重い

方式	向いている場面	注意点
カンバン	割り込みが多い。継続的に流れる仕事	WIP制限が曖昧だと詰まる
ハイブリッド	開発と運用が混在する	ルールの責任分界を明確にする

判断の基準

- 割り込みが多いならカンバン寄り

- 目標を期間で区切りたいならスクラム寄り

- 現実には両方混ざるので、最初から完璧な形を目指さない

実務での原則

1. ルールは少なく始める

2. ボトルネックを可視化する

3. ふりかえりで1つだけ改善する

4. プロセスを守ること自体を目的にしない

まとめ

良いプロセスは、現場の摩擦を減らす。

チームに合わない儀式は減らし、効くルールだけ残す。

15.5 技術的負債の管理

> このセクションで答える問い:「後で直す」をどう放置せずに扱うか。

なぜ必要か

技術的負債はゼロにできないが、見えなくすると危険だ。

負債そのものより、「誰も把握していない状態」がまずい。

全体像

種類	例	対応
コード負債	重複、長大関数、テスト不足	リファクタリング、テスト追加
設計負債	境界が曖昧、責務集中	境界整理、段階的分離
運用負債	手作業デプロイ、属人的手順	自動化、手順書整備
AI運用負債	生成コードの検証不足、評価セット欠落	チェックリスト整備、評価の自動化

管理の仕方

1. 負債を一覧にする

2. 影響と返済コストを見積もる

3. 通常開発の中で少しずつ返す

4. セキュリティ負債だけは先送りしない

返済の原則

- 触った箇所は少し良くして返す

- 大きな負債は小さく分けて返す

- 新機能開発と完全に切り離しすぎない

まとめ

負債管理は精神論ではなく在庫管理に近い。

見える化し、優先順位を付け、継続的に減らす。

15.6 オンボーディングと知識共有

> このセクションで答える問い: 新しいメンバーが早く戦力になるには何が必要か。

なぜ必要か

オンボーディングが弱いと、同じ質問が繰り返され、レビューコストも上がる。

逆に初期導線が整っていれば、新メンバーは早く安全に動ける。

最初の1週間で達成したいこと

1. 開発環境を再現できる

2. README を読んで起動できる

3. 小さなPRを1本出してマージまで体験する

4. システム構成と主要な業務用語を説明してもらう

知識共有で効く仕組み

施策	効果
ペア作業	暗黙知を短時間で伝えやすい
小さな最初のタスク	成功体験と導線確認を同時に作れる
ADR / README	背景知識を非同期で共有できる
定期共有会	局所知識の偏りを減らせる

AI支援を使う新人教育での注意点

AIは学習補助になるが、生成結果を読まずに採用する癖は危険だ。

新メンバーには次を明示する。

1. AIの提案は必ず読み、説明できる状態にする

2. テストを通しただけで理解したことにはしない

3. 分からない箇所はレビューで質問する

まとめ

オンボーディングでは、環境構築、最初のPR、業務文脈の理解を早く回す。

知識共有は善意任せにせず、仕組みにする。

15.7 AI時代のチーム開発

> このセクションで答える問い: AIを使う前提で、チームのルールはどう変わるべきか。

全体像

AIが増やすのは実装速度だけではない。レビュー量、検証負荷、情報管理の重要性も増える。

そのため、チームとしての最低ルールを持っておく必要がある。

論点	ルール例
責任分界	本番反映や重要変更の承認は人間が行う
データ持ち出し	機密コードや顧客情報をどこまで送ってよいかを定める
レビュー	AI生成コードであることを隠さない
評価	生成物の品質をテストとレビューで確認する
再現性	有効だったプロンプトや手順をチームで共有する

チームに必要な共通認識

1. AIは補助者であり、責任主体ではない

2. 速く書くことより、速く安全に直せることが重要

3. 生成物は「下書き」として扱う

4. ドキュメントとテストの価値はむしろ上がる

> IMPORTANT
>
> AI導入で最初に整えるべきなのは、ツールの比較表ではなく、
> 「どの情報を送れるか」「誰が最終承認するか」「何を必ず検証するか」の3点だ。

まとめ

AI時代のチーム開発で大事なのは、速さそのものではなく、速さを安全に扱う運用だ。

ツール選定より先に、責任、情報管理、検証のルールを決める。

この章のチェックリスト

- [ ] 自分のチームに合うブランチ戦略を説明できるか
- [ ] レビューで最低限見るべき観点を共有できているか
- [ ] README、ADR、運用手順書の役割を説明できるか
- [ ] プロセスを目的化せず、改善対象として扱えているか
- [ ] 技術的負債を可視化して優先順位づけできているか
- [ ] オンボーディングで最初の1週間に何を達成すべきか説明できるか
- [ ] AI利用時の責任分界、情報管理、検証ルールを定めているか

第16章 AI駆動開発

この章を読む前に: プログラミングとWeb開発の基礎（第1章から第4章）を理解していること。

AIを使った開発は、単なるコード補完ではなく、

設計、実装、検索、テスト、レビューの流れ全体を変えつつある。

ただし、重要なのは「何でもAIに任せる」ことではない。

どこまで任せ、どこで人間が責任を持つかを設計できるチームだけが、

速度と品質を両立できる。

16.1 AI コーディング支援ツールの使い分け

このセクションで答える問い: AIツールは何が違うのか。どう選べばよいのか。

なぜ必要か

AIツールは一括りに見えて、得意な作業が違う。

単一ファイルの補完に強いもの、リポジトリ全体の編集に強いもの、

ターミナル操作まで含めて自律的に進めるものがある。

全体像

種別	向いている作業	典型例
インライン補完	定型コード、テストの下書き、軽い補完	IDE統合型アシスタント
チャット支援	設計相談、コード説明、エラー原因の整理	IDE内チャット、Webチャット
エージェント型	複数ファイル修正、調査、実行、反復修正	ターミナル型、IDE型エージェント
UI生成型	たたき台の画面作成、プロトタイプ	ブラウザ型UI生成ツール

選定の判断軸

観点	見るべき点
編集範囲	単一ファイル中心か、複数ファイル横断か
ツール実行	テスト、検索、シェル操作まで任せるか
レビューしやすさ	差分が小さく追えるか、途中経過が見えるか
情報管理	送信されるコードやログを組織ルールで許容できるか

観点	見るべき点
再現性	うまくいった手順をテンプレート化できるか

実務での基本ルール

1. 小さな補完は軽いツールで済ませる
2. 大きな変更だけエージェント型を使う
3. 自動生成コードはそのままマージしない
4. 高リスク領域は人間がレビューする

> IMPORTANT
> AIツールは「速く書く装置」ではあるが、「正しいことを保証する装置」ではない。
> レビュー、テスト、運用判断は残る。

まとめ

ツール選定は人気や話題性ではなく、編集範囲、情報管理、レビューしやすさで決める。

まずは補完とチャットから始め、必要になったらエージェント型を足す。

16.2 LLM アプリケーション構築の基本パターン

> このセクションで答える問い: AI機能をアプリに組み込むとき、どこから始めるべきか。

なぜ必要か

AI機能は最初からRAGやマルチエージェントを入れる必要はない。

多くの機能は、段階的に複雑さを上げた方が失敗しにくい。

実装の進め方

1. 単純なプロンプト + 構造化出力 から始める
2. 社内知識や最新情報 が必要なら検索を足す
3. 外部操作 が必要ならツール呼び出しを足す
4. 単一フローで足りない 場合だけエージェント化する

全体像

要素	役割	いつ必要か
プロンプト設計	指示、制約、出力形式を定める	常に必要
構造化出力	JSONなどで機械処理しやすくする	アプリ連携時に重要
RAG	外部知識を参照させる	社内文書、FAQ、最新情報が必要なとき
ツール呼び出し	APIやDB、検索を実行する	回答だけでなく操作が必要なとき
ストリーミング	途中結果を返して体感待ち時間を下げる	応答が長いとき
ファインチューニング	特定出力や振る舞いを揃える	プロンプトやRAGで足りないときの後段

プロンプト設計の基本

- 役割を明示する

- 入力データと制約を分けて書く

- 期待する出力形式を固定する

- 「やってはいけないこと」も書く

構造化出力は「JSONっぽく」ではなくスキーマで縛る

アプリケーションがモデル出力を後続処理に使うなら、

自由文をパースするよりスキーマ付きの構造化出力を使う方が安定する。

実務では次の違いを意識する。

方式	何を保証しやすいか
自由文	人間が読む説明
JSON モード	JSONとして壊れていないこと
スキーマ付き構造化出力	必須項目、型、列挙値まで含めた整合性

ツール定義そのものがプロンプトになる

ツール呼び出しを使う場合、モデルはツール名、説明、入力スキーマを読んで判断する。

つまり、ツール定義は単なるメタデータではなく、モデルへの指示文でもある。

設計のコツ:

1. ツール名を曖昧にしない

2. 「いつ使うか」「使ってはいけないか」を説明に書く

3. 入力スキーマを厳密にする

4. 更新系ツールは読み取り系と分ける

RAG を入れる前に考えること

RAG は万能ではない。検索対象の文書が古い、断片化しすぎている、正解が存在しない場合は、

検索を足しても品質は上がらない。

まず確認する。

1. 参照したい知識が明文化されているか

2. 文書の更新頻度はどれくらいか

3. どの単位で分割すると意味が保てるか

4. 引用元を返す必要があるか

エージェント化は最後に検討する

> WARNING
> 単一の呼び出しで済む問題にエージェントを入れると、速度もコストもデバッグ難度も悪化しやすい。

エージェントが向くのは次のような場合。

- 複数のツールを順番に使う必要がある

- 途中結果を見て計画を修正する必要がある

- 長い作業を非同期で進めたい

まとめ

AI機能は、プロンプト、検索、ツール、エージェントの順に複雑さを上げる。

最初から全部盛りにしない。

16.3 モデル選定とルーティング

> このセクションで答える問い: どのモデルを選ぶべきか。どう使い分けるべきか。

なぜ必要か

モデル選定は「一番賢いものを選ぶ」では終わらない。

品質、速度、コスト、コンテキスト長、マルチモーダル対応、データの置き場所が絡む。

判断軸

観点	意味
品質	指示追従、推論力、要約精度、コード品質

観点	意味
速度	体感待ち時間に直結する
コスト	リクエスト単価だけでなく再試行や失敗も含む
コンテキスト長	長文や大規模コードベースを扱えるか
マルチモーダル	画像、音声、PDFを扱うか
展開形態	API利用か、プライベート環境か、ローカルか

実務での使い分け

クラス	向いている用途
小型モデル	分類、抽出、整形、軽い補助
汎用モデル	チャット、要約、通常のコード支援
推論重視モデル	複雑な計画、難しいバグ調査、判断補助
オープン/ローカルモデル	データ持ち出し制限、低遅延、独自制御

ルーティング戦略

1. まず小型モデルで試す
2. 品質が足りないときだけ上位モデルへ上げる
3. 失敗時のフォールバックを用意する
4. 重要処理はキャッシュと再試行制御を入れる

MCP の位置づけ

Model Context Protocol (MCP) は、モデルとツールやデータソースを接続するための共通的な方法だ。

価値は「新しい賢いモデル」ではなく、ツール統合の作り直しを減らせることにある。

MCP を理解するときは、3つの要素を分けると混乱しにくい。

要素	主な用途	呼び出し主体
Resources	参照データやコンテキストを渡す	アプリケーション側
Tools	モデルが必要に応じて実行する	モデル側
Prompts	再利用可能な手順や定型ワークフロー	アプリケーション側

特に大事なのは、Resources は参照中心、Tools は実行中心という違いだ。

この区別が曖昧だと、読み取りだけで済むはずのものに不要な実行権限を与えやすい。

まとめ

モデル選定はモデル名の比較表より、用途別の使い分けが重要だ。

単一モデル固定より、軽いものから重いものへ段階的に使う方が運用しやすい。

16.4 AI API、フレームワーク、関連パッケージの選び方

このセクションで答える問い: AI機能を実装するとき、どのレイヤーから選び始めるべきか。

なぜ必要か

AI実装では、モデルAPI、抽象化SDK、エージェント/ワークフローのフレームワーク、

補助パッケージ が一気に視界へ入る。

ここで全部を同時に入れると、学習コストもデバッグ範囲も急に広がる。

重要なのは、まず最短の1経路で動かし、必要になった層だけ後から足すことだ。

選定の順番

1. まずどの API 系統で比べるか決め、その中で単一の API / SDK で要件を満たせるか確認する

2. 複数プロバイダ比較や UI 向けの共通化が必要なら抽象化SDKを足す

3. 複数ステップ、長時間実行、人間承認が必要ならフレームワークを足す

4. 予算管理、スキーマ検証、観測などは不足が確認できた部分だけ補う

レイヤーごとの役割

層	役割	最初の選択	足す条件	典型例
公式 API / SDK	モデル、埋め込み、組み込みツールへ直接アクセスする	まずここから始める	単一の API 経路で十分、固有機能を使いたい	OpenAI Responses API、Anthropic Messages API、Gemini API
抽象化SDK	プロバイダ差分、ストリーミング、構造化出力、ツール呼び出しを揃える	後から追加する	モデル切り替え、UI 実装、比較実験が増えた	Vercel AI SDK、PydanticAI
オーケストレーション層	状態管理、分岐、再開、人間承認、長い処理を扱う	いきなり入れない	複数ステップのワークフローを明示的に制御したい	LangChain / LangGraph、LlamaIndex
ゲートウェイ / ルータ	認証、予算、フォールバック、負荷分散を集約する	組織課題が出るまで入れない	複数チーム、複数プロバイダ、共通運用が必要	LiteLLM、Vercel AI Gateway、Cloudflare AI Gateway

API選定で見るべき点

観点	確認したいこと
API の系統	ネイティブAPI、モデルカタログAPI、OpenAI互換 / 自前サービングのどれを比べているか
状態管理	単発リクエストだけか、会話状態、バッチ、バックグラウンド実行、Live セッションまで持てるか
組み込み機能	Web検索、ファイル検索、コード実行などをプロバイダ側で持っているか
構造化出力	JSON Schema や型定義を使って安定した出力を得られるか
マルチモーダル	画像、PDF、音声、動画を同じ経路で扱えるか
運用機能	バッチ、キャッシュ、非同期実行、再試行をどう扱えるか
展開先	直API、クラウド経由、プライベート環境など組織要件に合うか
互換性	既存 SDK を流用して比較できるか、ただし本番でも十分か

> TIP
>
> OpenAI互換 のような互換レイヤは比較や短期移植には便利だが、
> そのプロバイダ固有の機能を使う本番運用では、ネイティブAPIの方が素直なことが多い。

API は 3 系統に分けて比べる

OpenAI Responses API、Bedrock Converse API、vLLM は、

どれも「LLM を呼ぶ API」ではあるが、役割は同じではない。

比較表を作るときは、まず次の3系統を分ける。

系統	代表例	強み	注意点
ネイティブAPI	OpenAI Responses API、Anthropic Messages API、Gemini API / Live API	新機能、組み込みツール、構造化出力が最初に揃いやすい	切り替え時に認証、レスポンス形式、ツール仕様の差分を吸収する必要がある
モデルカタログAPI	Bedrock Responses / Converse API、Azure AI Foundry / Azure OpenAI、Vertex AI Model Garden	複数モデルを同じクラウド運用で比較しやすく、IAM や監査も寄せやすい	共通化される分、各社固有機能は遅れて見えることがある
OpenAI互換 / 自前サービング	vLLM などの OpenAI 互換サーバー	既存の OpenAI クライアントを流用しやすく、オープンモデルや閉域環境に向く	完全互換ではない。検索、コード実行、運用機能は自前実装が増えやすい

代表的な API の比較

API	強い点	比較時に見る点	向いている場面
OpenAI Responses API	組み込みツールが多く、構造化出力やバックグラウンド実行まで一つの流れで扱いやすい	Web検索、ファイル検索、コード実行、background mode を本当に使うか	直APIで素早く機能を出したい。検索やファイル処理も同じ API で寄せたい
Anthropic Messages API	ツール呼び出し、プロンプトキャッシュ、引用付き応答、バッチ実行を組み合わせやすい	長いコンテキストの再利用、引用の必要性、バッチ処理の有無	文書要約、根拠提示、長いシステムプロンプトを繰り返し使う処理

API	強い点	比較時に見る点	向いている場面
Gemini API / Live API	Google Search 連携、Files API、音声や動画を含むリアルタイム対話が強い	検索連動の必要性、ファイル処理の流れ、Live セッション設計	検索付き体験、音声UI、マルチモーダルなアプリ
Bedrock Responses / Converse API	複数ベンダモデルを AWS の権限、監査、ネットワーク制御の中で比較しやすい	Responses、Converse、Invoke のどれを標準経路にするか	AWS 前提で複数モデル比較や統制を進めたい
Azure AI Foundry / Azure OpenAI	モデル切り替えを寄せやすい経路と、OpenAI 固有機能を広く使う経路を分けて持てる	Foundry の共通推論 API を取るか、Azure OpenAI の固有機能を取るか	Azure 標準化、Entra ID、閉域要件、OpenAI 系機能の両立が必要
Vertex AI Model Garden	マネージド API で使うか、オープンモデルを自前エンドポイントへ載せるかを選びやすい	マネージド利用で足りるか、専用エンドポイントや自前デプロイが必要か	GCP 前提で Gemini とオープンモデルの両方を比べたい
vLLM などの OpenAI 互換 API	既存クライアントを流用しやすく、社内 GPU やオープンモデルの検証がしやすい	互換差分、サービング運用、監査やレート制御を自前で持てるか	自前推論、コスト最適化、データを外に出しにくい案件

フレームワーク選定の目安

状況	有力な選択	理由
TypeScript の Web アプリで、チャットUIやストリーミングを早く作りたい	Vercel AI SDK	プロバイダ差分、ストリーミング、構造化出力、ツール呼び出しをまとめて扱いやすい
Python で、型付き出力とアプリコードの整合を重視したい	PydanticAI	Pydantic ベースで出力型を先に決めやすく、複数プロバイダも扱える
モデルやツールの抽象化を持ちつつ、単純なエージェントから始めたい	LangChain	共通インターフェースで始めやすく、必要に応じて下位層へ降りられる
長時間実行、再開、人間承認、状態管理が重要	LangGraph	ワークフローを明示的に持ち、耐障害性や中断再開を設計しやすい
RAG やデータ接続が中心で、取り込みから検索までをまとめたい	LlamaIndex	データ接続、検索、ワークフローの文脈が強い

後半で検討する運用寄りの選択肢

OpenAI、Anthropic、Gemini のような直APIで要件を固めた後に、

比較対象へ入れやすいのは次のような運用寄りの経路だ。

選択肢	位置づけ	強み	後半で見る理由
Vercel AI Gateway	Web アプリ向けのゲートウェイ	複数モデル接続を寄せやすく、Vercel AI SDK と組み合わせて UI 実装も揃えやすい	最初から入れると、UI の課題と API の課題が混ざりやすい
Cloudflare Workers AI / AI Gateway	エッジ実行とゲートウェイをまとめる経路	エッジで推論や前処理を置きやすく、単一エンドポイントで観測や制御を寄せやすい	グローバル低遅延やエッジ要件が見えてから比べた方が判断しやすい
OpenRouter	複数プロバイダを束ねるルータ API	多くのモデルを一つの API で比較しやすく、短期の比較実験やフォールバック設計に向く	ネイティブAPIの固有機能より、比較速度を優先するときに価値が出る

選択肢	位置づけ	強み	後半で見る理由
Together AI / Fireworks AI / Groq / Replicate	オープンモデルや特化モデルを API で借りるホスト型推論基盤	自前GPUなしでオープンモデル、画像、音声、動画系の実験を始めやすい	どのモデル系統を使いたいか見えるまで候補を広げすぎない方がよい

関連パッケージの選び方

AI専用パッケージを増やしすぎる前に、次を先に揃える。

領域	まず選ぶもの	理由
スキーマ定義	Zod / Pydantic	構造化出力とアプリ側の型をずらしにくい
再試行・タイムアウト	既存の HTTP クライアントやジョブ基盤	AI専用ライブラリを足さなくても足りることが多い
観測	既存のログ、メトリクス、トレース基盤	まず失敗率、遅延、コストが見える状態を作る
検索基盤	既存DB拡張や既存検索基盤	いきなり専用基盤を増やすより単純に始めやすい

実務での基本方針

1. まず ネイティブAPI、モデルカタログAPI、OpenAI互換 / 自前サービング のどれを比べるか決める
2. その系統の中で 2〜3 候補だけを同じ評価セットで比較する
3. 最初は単一API + 構造化出力 で始め、文字列パースを減らす
4. プロバイダ切り替えの必要が出たら抽象化SDKを足す
5. ワークフローが複雑になったらオーケストレーション層を導入する
6. 組織全体の運用問題が出たらゲートウェイを検討する

まとめ

AIスタックの選定は、賢そうなフレームワークを最初に選ぶこと ではない。

まず どの API 系統を比べるか を決め、

その上で 公式 API → 抽象化SDK → オーケストレーション → ゲートウェイ の順で、

必要になった層だけ足す方が、速度も保守性も両立しやすい。

16.5 埋め込み、ベクトル検索、RAG の実装

> このセクションで答える問い: 検索付きAIをどう実装し、どこで失敗しやすいか。

なぜ必要か

RAG の品質は、LLM本体より検索の設計で決まることが多い。

埋め込みモデルやベクトルDBの名前だけで精度は決まらない。

全体像

要素	役割
文書分割	どの単位で意味を保ったまま検索可能にするか
埋め込み	文書と質問を検索可能なベクトルに変換する
検索	類似度やキーワードで候補を取る
再ランキング	候補の順番を整える
回答生成	取得した根拠をもとに回答する

実務で効く順番

1. 文書の品質を整える
2. 分割単位とメタデータを見直す
3. ベクトル検索だけでなくキーワード検索も検討する
4. 引用元を返す
5. 評価セットで検索精度を確認する

ベクトルDB選定の考え方

選択肢	向いている場面
既存DB拡張	小〜中規模、まず素早く始めたい
マネージドな専用サービス	運用負荷を減らしたい、大規模化する
検索エンジン併用	キーワード検索と意味検索を両方使いたい

> TIP
>
> 小さく始めるなら、既存DBや既存検索基盤に載せる方がシンプルなことが多い。
> 先に検索品質を確認し、スケールが問題になってから専用基盤を検討する。

まとめ

RAG の成功はベクトルDBの名前ではなく、文書整備、分割、評価の設計で決まる。

検索品質の検証なしに基盤だけ豪華にしても改善しない。

16.6 AI機能の評価とテスト

このセクションで答える問い: 非決定的な出力をどう評価するか。

なぜ必要か

AI機能は同じ入力でも出力が揺れる。

そのため、通常の単体テストだけでは品質を十分に測れない。

評価の層

層	目的
決定的テスト	JSON形式、必須項目、禁止語、例外処理を確認する
オフライン評価	用意した問題集で品質比較する
モデルベース評価	別モデルで採点や比較を補助する
人間レビュー	本当に役立つか、安全かを見る
本番指標	クリック率、解決率、再問い合わせ率などを見る

実務で最低限必要なもの

1. 代表的な質問や入力の評価セット

2. 失敗してはいけないケースの一覧

3. プロンプト変更時の再評価

4. 本番ログから評価セットへ戻す仕組み

評価セットを作るときの原則

OpenAI や Anthropic の評価ガイドでは、汎用ベンチマークより

自分たちのタスクに即した eval-driven development を強く勧めている。

最初にそろえたいのは次の4種類。

ケース	例
正常系	期待どおりに答えてほしい代表ケース
境界系	曖昧な指示、長文、欠損入力
禁止系	答えてはいけない要求、権限外の操作
回帰系	過去に失敗した実例

judge を使うときの注意点

モデルベース評価は便利だが、万能ではない。

公式ガイドでも、人間評価との較正を入れずに judge だけで運用することは勧めていない。

実務では次の順が安定する。

1. まずルールベースで判定できる項目を機械化する

2. 難しい評価だけ model-as-judge を使う

3. 定期的に人間がサンプル監査する

エージェントは最終回答だけでなく trace も評価する

ツール使用や複数ステップの処理では、最終回答だけ見ても不十分なことがある。

途中で不要なツールを呼びすぎていたり、権限外の操作を試みていたりするからだ。

そのため、エージェント評価では次も見る。

- どのツールを呼んだか

- 何回呼んだか

- 禁止されたツールを触っていないか

- 途中の失敗から正しく回復したか

外部ベンチマークの使い方

SWE-bench のような外部ベンチマークは参考になるが、

自分たちのタスクにそのまま一致するとは限らない。

外部ベンチマークは候補絞り込みに使い、最終判断は自分たちの評価セットで行う。

まとめ

AI評価は「一発で正解したか」より、「実運用で許容できる品質か」を見る。

評価セットはプロンプトと同じくらい重要な資産だ。

16.7 AI開発のリスク、倫理、運用制約

> このセクションで答える問い: AI機能を出すとき、何を必ず確認すべきか。

全体像

論点	典型的なリスク	基本対策
ハルシネーション	もっともらしい誤答	引用、根拠提示、評価セット
バイアス	属性による不当な出力差	テストケースの多様化、レビュー
プライバシー	個人情報や機密情報の送信	データ最小化、マスキング、利用規約確認
著作権・ライセンス	生成物や学習元の扱いが曖昧	利用条件の確認、高リスク用途での法務確認
セキュリティ	プロンプト注入、過剰なツール実行	最小権限、人間承認、監査ログ
コスト	想定外のAPI課金	レート制限、キャッシュ、モデル分岐

高リスク領域での原則

- 医療、法務、金融などは人間確認を外さない

- 重要な判断をAIだけに委ねない

- AI出力であることを必要に応じて明示する

- ログと評価結果を残す

まとめ

AI機能は便利だが、通常のソフトウェアより不確実性が高い。

だからこそ、境界、監査、評価を先に設計する。

この章のチェックリスト

- [ ] AI支援ツールを補完型、チャット型、エージェント型で使い分けられるか
- [ ] LLMアプリをプロンプト、検索、ツール、エージェントの順で段階的に設計できるか
- [ ] モデル選定を品質、速度、コスト、コンテキストで説明できるか
- [ ] ネイティブAPI、モデルカタログAPI、OpenAI互換 / 自前サービング API を分けて比較できるか
- [ ] 公式 API、抽象化SDK、オーケストレーション層をどの順で足すべきか説明できるか
- [ ] RAG の品質が文書分割や検索設計に左右される理由を説明できるか
- [ ] 評価セット、決定的テスト、人間レビューの役割を説明できるか
- [ ] AI機能でハルシネーション、プライバシー、コストをどう管理するか説明できるか

第17章 2026年の注目技術と動向

> この章を読む前に: 本書の前半を読んでいると、各トピックの位置づけを理解しやすい。
> この章は特に変化が速いため、記載の基準日は 2026年3月と考える。

トレンド章で大事なのは「当たる予言」をすることではない。

何が本流になりそうか、何がまだ試行段階か、どの条件が揃ったら採用すべきかを

見分けられるようにすることだ。

2026年に見えている5つの流れ

1. AIが追加機能ではなく基盤機能になった

2. フロントエンドはクライアント一辺倒からサーバー併用へ戻っている

3. エッジ実行とオンデバイス推論が選択肢として現実的になった

4. 型、安全性、来歴の管理が設計時点の要件になった

5. 開発体験は個人最適より組織最適へ移っている

17.1 サーバー中心UIとストリーミング

> このセクションで答える問い: フロントエンドはどこまでサーバー側に戻っているのか。

全体像

React の Server Components や、フレームワーク側で提供される Server Actions / Server Functions

の流れにより、UIの一部をサーバーで扱う設計が定着しつつある。

ここでの本質は「全部サーバーに戻す」ことではなく、次を分けることだ。

- サーバーで安全に処理したいもの

- クライアントで即時反応したいもの

- ストリーミングで体感待ち時間を下げたいもの

2026年3月時点での現在地

React 19 系では、Server Components はアプリ利用者向けの機能としては stable と整理されている。

ただし React 公式は、react-server 実装や bundler 向けの基盤APIについては、

フレームワーク実装者向けであり、19.x の間でも変わりうると明記している。

実務上の意味は次の通り。

1. アプリ開発者は、素の独自実装よりフレームワーク経由で使う

2. react-server-dom-* を深く直接扱う設計は避ける

3. RSC 採用時はフレームワークの追従リリースを追う

Server Functions / Server Actions で外しやすい点

React 公式の use server ドキュメントでは、引数はすべてクライアントから制御可能な入力として扱うべきだと明示している。

便利に見えても、内部関数ではなく公開された更新エンドポイントに近い。

最低限守る。

- 引数を必ず検証する

- 認証と認可をサーバー側で毎回行う

- 戻り値と引数は直列化可能な型に寄せる

- 更新系と参照系の責務を分ける

実務での見方

向いているケース	注意点
データ取得と表示が近い画面	認証・キャッシュ・再検証の整理が必要
フォーム中心の業務UI	サーバー側の入力検証が必須
段階的表示が効く画面	読み込み順序やエラー時UXを考える

まとめ

フロントエンドは「SPAかSSRか」の二択ではなくなっている。

サーバーとクライアントの境界を、画面単位ではなく責務単位で引く時代になっている。

17.2 オンデバイスAIとエッジAI

> このセクションで答える問い: AI推論をクラウド以外で動かす価値は何か。

なぜ注目されるか

すべての推論をクラウドに送ると、遅延、通信費、プライバシー、オフライン耐性が課題になる。

軽い推論を端末側やエッジで処理する設計は、以前より現実的になっている。

全体像

配置	向いている処理
オンデバイス	顔検出、軽い分類、音声前処理、プライバシー重視処理
エッジ	近い場所で返したい推論、前処理、キャッシュ
クラウド	重い生成、長い推論、集約データが必要な処理

実務での定石

1. 端末側で軽く判定する

2. 不確実なケースだけクラウドへ送る

3. 個人情報は可能なら端末側で削る

まとめ

エッジAIとオンデバイスAIは、クラウドを置き換えるものではない。

ハイブリッド構成で遅延とプライバシーを調整する手段として価値がある。

17.3 WebAssembly、WASI、軽量ランタイム

> このセクションで答える問い: Wasm はどこで実用になっているのか。

なぜ注目されるか

軽量で分離しやすい実行環境がほしい場面では、Wasm が有力な選択肢になっている。

特にブラウザ以外の実行、プラグイン、エッジワークロードで存在感が増している。

全体像

向いている場面	理由
エッジ処理	起動が軽く、分離しやすい
プラグイン実行	外部コードを比較的安全に閉じ込めやすい
CPU集約処理	画像変換、圧縮、解析などで使いやすい

向いている場面	理由
複数言語資産の再利用	既存コードを別環境へ持ち込みやすい

注意点

- 何でもWasmにすればよいわけではない

- I/Oや運用の都合ではコンテナの方が自然な場面も多い

- チームがデバッグしやすい運用を優先する

まとめ

Wasm は「コンテナの完全な後継」ではない。

軽量性、分離、配布のしやすさが効く場所で選ぶ。

17.4 型共有とフルスタック契約

> このセクションで答える問い: 型安全なフルスタック開発はどこまで進んでいるのか。

全体像

フロントエンド、バックエンド、DB、バリデーションの契約をできるだけ揃える流れは強い。

ただし、ここで重要なのは特定ライブラリの勝敗ではなく、契約の一元化だ。

よく使われる方向性

方向性	例
スキーマを共通化する	zod、OpenAPI、GraphQL Schema
API型を共有する	tRPC、コード生成付きREST/GraphQL
DBスキーマをコード化する	型付きORM、マイグレーション管理

見るべきポイント

- 型だけでなくランタイム検証があるか

- フロントとバックで二重管理になっていないか

- 外部公開APIにも同じ方式をそのまま使えるか

まとめ

2026年の潮流は「RESTかGraphQLか」より、「契約をずらさない」ことにある。

17.5 Platform Engineering と内部開発プラットフォーム

このセクションで答える問い: 開発者体験を組織的に整える動きはなぜ強まっているのか。

なぜ注目されるか

チーム数が増えると、デプロイ、権限申請、監視設定、テンプレート整備を

各チームが個別にやるのは非効率になる。

そこで、共通の足場を作る Platform Engineering の重要性が増す。

全体像

目的	例
セルフサービス化	新サービス作成、環境払い出し、デプロイ
標準化	テンプレート、監視、セキュリティ設定の共通化
可視化	サービス一覧、責任者、運用状態の把握

最小の内部開発プラットフォーム

Backstage 系の実践を見ると、最初から巨大ポータルを作る必要はない。

多くの組織では、次の3点がそろうだけでも効果が出る。

要素	役割
Software Catalog	サービス一覧、責任者、依存関係を見える化する
Software Templates	新規サービス作成をテンプレート化する
Docs as Code	ドキュメントをコードと同じリポジトリで管理する

この3つがあると、開発者体験の改善だけでなく、運用責任の所在や標準化の徹底にも効く。

いつ必要か

- 複数チームが同じ基盤で動く

- 同じセットアップを何度も繰り返している

- 監視や権限設定が属人化している

まとめ

Platform Engineering は大企業だけの話ではない。

同じ作業を何度も繰り返しているなら、小さな内部プラットフォームから始める価値がある。

17.6 サプライチェーンセキュリティの常態化

このセクションで答える問い: セキュリティで何が「特別対応」から「標準作業」に変わったのか。

全体像

依存関係管理、SBOM、署名、来歴、短命資格情報といった仕組みは、

一部の高規制業界だけの話ではなくなっている。

実務で定着しつつあるもの

- 依存関係の自動通知
- SBOM の生成
- 成果物の署名と検証
- OIDC など短命資格情報によるデプロイ
- 生成物の来歴確認

まとめ

サプライチェーン対策は「厳しい組織だけの追加要件」ではなく、

通常開発の一部として組み込まれつつある。

17.7 コストとサステナビリティの統合管理

このセクションで答える問い: コスト最適化と環境配慮はどう結びつくのか。

全体像

AIや常時稼働基盤の普及で、計算資源の使い方そのものが設計課題になっている。

未使用環境の停止、キャッシュ、適切なモデル選定は、コストにも環境負荷にも効く。

実務で効く観点

観点	具体策
未使用資源の削減	開発環境の自動停止、不要ストレージ削除
処理効率	キャッシュ、バッチ化、過剰再試行の抑制
AI負荷の最適化	小型モデル優先、必要時のみ上位モデル
可視化	コストと利用量をチーム単位で追う

まとめ

GreenOps は独立した美徳としてより、日常的な無駄削減の延長として進める方が続く。

17.8 技術採用の見取り図

> このセクションで答える問い: 何を今すぐ採用し、何を観察対象に置くべきか。

2026年時点のざっくりした分類

位置づけ	技術の例	取り扱い
標準化しやすい	パスキー、RAG、評価セット、SBOM、自動依存更新	通常開発に組み込む
条件つきで採用	サーバー中心UI、BFF、エッジ推論、内部開発プラットフォーム	要件に合えば採用
観察しながら試す	マルチエージェント、Wasm拡大利用、より強い型駆動設計	小さく実験する
流行を追いすぎない	目的が曖昧な全面刷新	まず課題を定義する

まとめ

技術採用は「新しいから」ではなく、「今の課題に効くか」で決める。

観察、実験、標準化を分けて考えると判断しやすい。

17.9 AI時代にトレンドを読む視点

> このセクションで答える問い: 流行に振り回されずに意思決定するにはどうするか。

まず見るべき質問

1. それは今の制約を本当に解くか

2. チームが運用できるか

3. セキュリティと監査を入れられるか

4. 後戻りできる導入手順になっているか

2026年の本質

2026年の技術変化を一言でまとめるなら、

「AIが前提になったことで、境界、契約、評価、運用の重要性が増した」

ということだ。

速い技術ほど、検証と運用の土台が要る。

まとめ

トレンドを追うときは、製品名より構造変化を見る。

境界、契約、評価、運用が強くなる方向の技術は、比較的長持ちしやすい。

この章のチェックリスト

- [ ] サーバー中心UIが再注目されている理由を説明できるか
- [ ] オンデバイスAI、エッジAI、クラウドAIの役割分担を説明できるか
- [ ] Wasm が向く場面と、コンテナの方が自然な場面を区別できるか
- [ ] 型共有の本質が「特定ライブラリ」ではなく「契約の一元化」にあると説明できるか
- [ ] Platform Engineering が必要になる組織条件を説明できるか
- [ ] サプライチェーン対策が通常開発に組み込まれつつある理由を説明できるか
- [ ] コスト最適化とサステナビリティが両立しやすい理由を説明できるか
- [ ] 技術採用を標準化、条件つき採用、観察対象に分けて考えられるか

付録A 技術選定フレームワーク

A.1 技術選定チェックリスト

新しい技術を導入するときは、流行より先に次を確認する。

観点	確認したいこと
問題適合性	本当に今の課題を解くか
運用適合性	チームが運用、監視、障害対応まで回せるか
学習コスト	何を新しく覚える必要があるか
相互運用性	既存スタックとつながるか
セキュリティ	権限、脆弱性対応、来歴確認はしやすいか
ライセンス	商用利用や配布に問題がないか
移行可能性	失敗したときに戻せるか
ドキュメント	公式情報が十分にあるか

追加で聞くべき質問

1. 今の不満を言語化できているか
2. 導入しない場合のコストは何か
3. 小さく試す方法はあるか
4. 3か月後に見直す観点は何か

A.2 ADR テンプレート

判断の背景を残すための最小テンプレート。

```
# ADR-001: {タイトル}

## 状態
提案中 / 採用 / 廃止

## 背景
何に困っているか。どんな制約があるか。

## 選択肢
```

```
- 選択肢A
- 選択肢B
- 選択肢C

## 決定
何を選んだか。

## 理由
なぜそれを選んだか。

## 影響
得るもの、失うもの、移行時の注意点。
```

運用ルール

- ADR は履歴として残す

- 判断が変わったら新しいADRを作る

- コードと同じリポジトリで管理する

A.3 意思決定マトリクスの使い方

数値化は正解を出すためではなく、何を重視しているかを見えるようにするために使う。

例:

評価軸	重み	候補A	候補B	候補C
チーム経験	25%	5	3	4
学習コスト	15%	4	5	3
運用しやすさ	20%	4	3	5
セキュリティ	20%	4	4	4
将来の拡張性	20%	3	5	4

> NOTE
> 点数は客観的真実ではない。議論の材料として使う。

A.4 AI時代に追加で見るべきこと

AI前提の開発では、従来の選定軸に次も足す。

観点	具体例
生成しやすさ	定型コードやテストを作りやすいか
レビューしやすさ	AIが出した差分を人間が追いやすいか
評価しやすさ	テスト、Evals、静的検査を入れやすいか
情報管理	機密データを外へ出さずに運用できるか
ドキュメントとの整合	README や ADR と矛盾なく更新できるか

誤りやすい考え方

- AIが得意だからその技術を選ぶ

- 生成量が多いから保守もしやすいと思い込む

- 人間の理解コストを無視する

まとめ

AI時代でも、最終的に残るのは保守できるコードと運用できる仕組みだ。

AIとの相性は補助条件であり、主目的ではない。

A.5 見直しタイミング

技術選定は一度決めて終わりではない。次のときに見直す。

1. チーム構成が大きく変わった

2. 運用コストが想定を超えた

3. セキュリティやライセンス条件が変わった

4. 主要な制約が変化した

付録B 推奨学習リソースとロードマップ

B.1 学習の進め方

学習では「広く知る」より「小さく作り切る」方が重要だ。

順序としては、次の流れが安定しやすい。

1. Webとプログラミングの基礎

2. 画面とAPIの往復

3. DB、認証、テスト

4. デプロイ、監視、セキュリティ

5. AI機能や運用改善

B.2 段階的ロードマップ

Phase 1: 基礎

テーマ	まず学ぶもの
Webの基本	HTML、CSS、HTTP
プログラミング	JavaScript / TypeScript または Python
開発環境	Git、ターミナル、エディタ

Phase 2: Web開発

テーマ	まず学ぶもの
フロントエンド	React などの主要フレームワーク
バックエンド	REST API、入力検証、エラーハンドリング
データ	SQL、PostgreSQL、マイグレーション

Phase 3: チーム開発

テーマ	まず学ぶもの
品質	テスト、レビュー、CI
運用	Docker、デプロイ、監視
セキュリティ	認証、認可、OWASPの基本

Phase 4: 応用

テーマ	まず学ぶもの
アーキテクチャ	境界分割、非同期処理、設計判断
AI開発	プロンプト、RAG、評価セット
改善	負債管理、開発プロセス、内部標準化

B.3 まず見るとよい学習リソース

公式ドキュメント

リソース	用途
MDN Web Docs	Web標準の基礎
TypeScript 公式ドキュメント	型システムの理解
React / Next.js 公式ドキュメント	フロントエンド実装
PostgreSQL 公式ドキュメント	SQLとDBの基礎
Docker 公式ドキュメント	コンテナの基礎
GitHub Actions 公式ドキュメント	CI/CD の基本
OWASP	セキュリティの基礎

補助的な教材

リソース	用途
freeCodeCamp	初学者向けの手を動かす学習
The Odin Project	フルスタックの学習導線
roadmap.sh	分野ごとの見取り図
The Missing Semester	シェルやGitなど周辺技術

AI関連を学ぶとき

AI機能については、各プロバイダや主要OSSの公式ガイドを優先する。

二次情報より、まず一次情報に当たる癖をつける。

B.4 学習の原則

原則	内容
手を動かす	読むだけで終わらせず、小さく実装する
完成させる	学習用でもデプロイや公開まで一度通す
公式情報を読む	まず一次情報で確認する
記録を残す	README、メモ、ADRで学びを言語化する
AIを補助として使う	説明、要約、たたき台に使い、理解は自分で確認する

B.5 AIを使った学習での注意点

AIは学習速度を上げやすいが、理解を飛ばすと後で詰まる。

守るとよいルール

1. 生成されたコードを読んで説明できる状態にする

2. 動いたことと理解したことを混同しない

3. バグ修正の理由を言葉にする

4. 分からないところは公式ドキュメントに戻る

初学者向けの使い方

- エラーメッセージの意味を聞く

- コードの説明をさせる

- 練習問題を出させる

- たたき台を出させたあと、自分で直す

避けたい使い方

- コードを読まずに貼る

- テストや検証を飛ばす

- 毎回AIに答えだけ聞く

B.6 経験者の学び直し

既存の開発経験がある人は、文法の完全習得より、差分の把握を優先すると早い。

例

- オブジェクト指向に慣れているなら、TypeScriptの型とフロントエンドの状態管理に集中する

- バックエンド経験者なら、Reactとブラウザ実行モデルを先に押さえる

- 運用経験者なら、AI機能の評価や監査設計に強みを活かせる

まとめ

学習では「自分に欠けている前提」を先に埋める。

全部を最初から学び直す必要はない。

付録C 用語集

本書で繰り返し登場する用語を、短く整理する。

A

ADR（Architecture Decision Record）: 設計判断とその理由を残す文書。→ 第15章, 付録A

API（Application Programming Interface）: ソフトウェア同士が情報をやり取りするための接点。→ 第4章

B

BFF（Backend for Frontend）: クライアントごとに最適化したバックエンドの層。→ 第14章

ブランチ: Git で本線から分けて作業するための分岐。→ 第1章, 第15章

C

CI/CD: コード変更のテスト、ビルド、デプロイを自動化する仕組み。→ 第10章

CSP（Content Security Policy）: ブラウザが読み込めるスクリプトやリソースを制限する仕組み。→ 第13章

コンテナ: アプリケーションと実行環境をまとめて扱う単位。Docker が代表例。→ 第1章

D

DDD（Domain-Driven Design）: 業務の構造に合わせてソフトウェアを設計する考え方。→ 第14章

Docker: コンテナを作成、配布、実行するための代表的なツール。→ 第1章

E

E2Eテスト: ユーザー操作に近い流れでアプリ全体を検証するテスト。→ 第12章

エッジコンピューティング: 利用者に近い場所で処理を行い、遅延を減らす考え方。→ 第9章, 第17章

エンベディング（Embedding）: テキストや画像をベクトルへ変換する処理。→ 第16章

F

Feature Flags: 機能の有効・無効をデプロイと切り離して制御する仕組み。→ 第10章

Fine-tuning: 既存モデルを追加学習して特定用途向けに調整すること。→ 第16章

G

Git: ソースコードの履歴を管理する分散型バージョン管理システム。→ 第1章

GitOps: Git の状態を正として、インフラやデプロイを管理する考え方。→ 第10章

GraphQL: クライアントが必要なデータ形を指定して取得できるAPI方式。→ 第4章, 第14章

H

ハルシネーション: LLMが事実でない情報をもっともらしく出力すること。→ 第16章

I

IaC（Infrastructure as Code）: インフラ設定をコードとして管理する手法。→ 第9章

J

JWT（JSON Web Token）: 認証や認可情報の受け渡しに使われる署名付きトークン形式。→ 第4章

K

Kubernetes: コンテナの配置、スケール、運用を自動化する基盤。→ 第7章

L

LLM（Large Language Model）: 大量のテキストで学習した言語モデル。→ 第16章

LoRA: ファインチューニングを軽量化するための学習手法。→ 第16章

M

MCP（Model Context Protocol）: モデルとツールやデータを接続するための共通的な方法。→ 第16章

マイクロサービス: 機能ごとに独立したサービスに分けるアーキテクチャ。→ 第14章

マイグレーション: DBスキーマの変更履歴を安全に管理する仕組み。→ 第5章

モノリス: 1つのアプリケーションとして機能をまとめる構造。→ 第14章

O

OAuth 2.0: 外部サービスへ認証を委任するための標準的な仕組み。→ 第4章

OpenTelemetry: メトリクス、ログ、トレースを共通形式で扱うための標準。→ 第11章

OWASP: Webアプリケーションセキュリティに関する代表的なコミュニティと資料群。→ 第13章

P

PaaS: インフラ管理の多くをサービス側に任せ、アプリ開発に集中しやすくする形態。→ 第8章

Passkeys: パスワードの代わりに使う認証方式。FIDO2 / WebAuthn に基づく。→ 第13章

Postmortem: 障害後に原因と再発防止策を整理するふりかえり。→ 第11章

プロンプトインジェクション: LLMに意図しない指示を埋め込む攻撃。→ 第13章

R

RAG（Retrieval-Augmented Generation）: 外部知識を検索してからLLMに渡す構成。→ 第16章

REST: HTTPメソッドとURLを中心にリソースを扱うAPI設計。→ 第4章

S

SBOM（Software Bill of Materials）: ソフトウェアに含まれる構成要素の一覧。→ 第13章, 第17章

SLA / SLI / SLO: サービス品質の契約、指標、目標を整理する考え方。→ 第11章

SLSA: ソフトウェアサプライチェーンの来歴と検証を強化するための枠組み。→ 第13章, 第17章

SPA（Single Page Application）: ページ全体を再読み込みせずに画面を更新するWebアプリの形。→ 第3章

SSR（Server-Side Rendering）: サーバー側でHTMLを生成して返す表示方式。→ 第3章

T

TDD（Test-Driven Development）: テストを先に書いてから実装を進める開発手法。→ 第12章

tRPC: TypeScript の型をクライアントとサーバーで共有しやすくする仕組み。→ 第17章

トークン: LLMが入力や出力を扱うときの分割単位。→ 第16章

V

ベクトルDB: ベクトル化されたデータを類似度検索できるデータベース。→ 第16章

W

WebAssembly（Wasm）: ブラウザやサーバーで動く軽量なバイナリ実行形式。→ 第17章

WebSocket: ブラウザとサーバーの双方向通信を維持する仕組み。→ 第4章

WSL: Windows 上で Linux 環境を使うための仕組み。→ 第1章