	手続き型言語による開発	オブジェクト指向言語による開発	クラウドネイティブでの開発	AI駆動型開発
開発言語	C、COBOL、PL/Iなど	Java、C++、Pythonなど	HTML5/JS、Ruby、PHPなど	Python 特定の言語に制約されるものではないが、Pythonは、AI駆動型開発の特徴であるデータ中心のプロセス、高速なプロトタイピング、豊富なライブラリ、多様な分野との連携、過渡期の方法論でもあり活発なコミュニティ活動の存在が有効だから。
特徴	**手続き中心のプログラミング**: プログラムは一連の手順（手続きや関数）として構成され、順次実行される。 **明確な制御フロー**: プログラムの流れが明確で、コードの読みやすさや理解しやすい。 **低レベルな操作の容易さ**: メモリ管理やハードウェアとの直接的なやり取りが可能で、システムプログラミングに適している。	**オブジェクト中心のプログラミング**: データとそれに関連する操作を一つの「オブジェクト」としてまとめる。 **再利用性と拡張性の向上**: 継承やポリモーフィズムにより、コードの再利用が促進。 **抽象化の強化**: デザインパターンやフレームワークの活用により、高度な抽象化が可能。	**ウェブ技術の活用:** フロントエンドではHTML5やJavaScript、バックエンドではRubyやPHPを使用。 **マイクロサービスアーキテクチャ**: 小さな独立したサービスの集合体としてシステムを構築。 **コンテナ化とオーケストレーション**: DockerやKubernetesを用いてアプリケーションをデプロイ・管理。 **継続的インテグレーション/デリバリー（CI/CD）**: 開発からデプロイまでのプロセスを自動化。	**データ中心の開発:** アルゴリズムよりもデータの質と量が重要。 モデルの訓練と評価: 機械学習モデルの開発が核心であり、反復的なプロセスが必要。 **不確実性の高い開発**: 結果がデータやモデルによって大きく変動し、予測が難しい。 **高度な専門知識の必要性**: データサイエンス、機械学習、深層学習などの専門知識が求められる。
課題	**再利用性の低さ**: コードの再利用が難しく、同様の機能を持つコードが複数存在することがある。 **保守性の問題**: 大規模なプログラムになると、コードの管理や変更が困難。 **抽象化の不足**: 高度な抽象化機能が乏しく、複雑なシステムの設計が難しい。	**設計の複雑性**: オブジェクト指向設計には高度なスキルが必要で、設計ミスが全体に影響を及ぼす可能性がある。 **過剰な抽象化のリスク**: 不必要な抽象化により、コードが複雑化し理解しにくくなる。 **学習コスト**: オブジェクト指向の概念やデザインパターンの習得に時間がかかる。	**技術の急速な変化**: 新しいフレームワークやライブラリが頻繁に登場し、学習が継続的に必要。 **複雑な環境管理**: マイクロサービスやコンテナの管理には高度な知識が求められる。 **セキュリティリスク**: ウェブアプリケーション特有のセキュリティ問題に対処する必要がある。 **パフォーマンスの最適化**: 分散システムにおける通信遅延やリソース管理が課題。	**データ品質の確保**: ノイズやバイアスのない高品質なデータの収集・前処理が難しい。 **モデルの解釈性**: ブラックボックス化しやすいモデルの動作を理解・説明する必要がある。 **倫理的・法的な問題**: データプライバシーや偏見の排除など、倫理的な配慮が求められる。 **計算資源の確保**: 大規模なデータセットやモデルの訓練には高性能なハードウェアが必要。
手順	**ウォーターフォール型の順序立てた手順で進め、明確な要件定義と設計が重要** **要件定義:** システムに必要な機能や性能を明確にする。 システム設計: **基本設計（外部設計）**: システム全体の構成や外部インターフェースを設計。 **詳細設計（内部設計）**: モジュールやアルゴリズムの詳細を設計。 **実装**: COBOLやC言語などの手続き型言語でコーディング。 **テスト**: **単体テスト**: 各モジュールが正しく動作するか確認。 **結合テスト**: モジュール間の連携を検証。 **システムテスト**: システム全体の動作をテスト。 **デプロイと運用**: 本番環境に導入し、システムの運用と保守。	**OOA/OODを活用し、再利用性と拡張性を重視した設計・実装** **要件定義**: ユースケースや機能要件を整理します。 **オブジェクト指向分析（OOA）:** クラスやオブジェクトを抽出し、関係性をモデル化。 **オブジェクト指向設計（OOD）**: クラスの詳細設計やデザインパターンを適用。 **実装**: JavaやC++などのオブジェクト指向言語でコーディング。 **テスト:** **ユニットテスト**: クラスやメソッドの機能を個別にテスト。 **インテグレーションテスト**: オブジェクト間の連携を検証。 **システムテスト**: システム全体の動作をテスト。 **デプロイと運用**: CI/CDを活用し自動化されたデプロイと運用。	**アジャイルとDevOpsを組み合わせ、高速な開発・デプロイと継続的な改善を実現** **要件定義**: ユーザーストーリーやプロダクトバックログを作成。 **アーキテクチャ設計**: マイクロサービスの定義やクラウドリソースの選定。 **インフラ構築**: IaC（Infrastructure as Code）やコンテナ技術を用いてインフラを構築。 **開発環境設定**: CI/CDパイプラインや開発ツールを設定。 **実装**: **フロントエンド開発**: ReactやVue.jsなどでユーザーインターフェースを構築。 **バックエンド開発**: Ruby on RailsやNode.jsなどでAPIやビジネスロジックを実装。 **テストと品質保証**: 自動テストやセキュリティテストを実施。 デプロイとリリース: クラウド環境へデプロイし、継続的にリリース。 **運用とフィードバック**: システムの監視とユーザーフィードバックを活用し、継続的に改善。	**データとモデルを中心に据え、倫理的配慮や不確実性の管理が重要** **導入計画と基盤の構築**：ソフトウェアやツール（GitHub Copilot、Cursorなど）の選定と導入。 **要件定義**：LLMとの対話で要件を段階的に具体化し、リリースされたソフトウェアを元に要件を更新・追加。 **UI/UXデザイン**：手書き画像やAIとの対話でUIのイメージを生成しインクリメンタルにUI設計。 **ソフトウェア設計**：要件から設計案をドキュメントとして作成し、ソフトウェアアーキテクトがレビューして適切な内容になるようにアップデート。 **コーディング**：ソフトウェア設計とUIデザインを基に自動的にコード生成し、エンジニアがレビューと動作確認。 **テスト**：Unit TestやIntegration Testなどのテストケースも LLMがベースを作成し、テスト担当者がレビューと追加指示。 **リリース**：クラウド環境へのリリースやCI/CDパイプラインの作成をIaCテンプレートで実行。
見積方法	**ライン・オブ・コード (LOC) ベースの見積り**: 予測されるコード行数から開発工数を見積もります。 **ファンクションポイント法**: システムの機能（入力、出力、ファイル、外部インターフェース）の数と複雑さを元に見積もります。 **COCOMOモデル**: プロジェクトの規模と複雑性に基づいて工数と期間を予測します。	**ユースケースポイント法**: ユースケースの数と複雑さから工数を見積もります。 **オブジェクトポイント法**: 画面、レポート、3GLコンポーネントなどの数と複雑さを基に見積もります。 **クラスベースの見積り**: 必要なクラスの数や継承関係を考慮して工数を見積もります。 **機能ポイント法のオブジェクト指向版**: オブジェクト指向の特性を反映した機能ポイントを用いて見積もります。	**アジャイル見積もり:** ストーリーポイントやタスクベースの見積もり: ユーザーストーリーやタスクの複雑さを評価。 **エンピリカルなデータの活用:** 過去のプロジェクトデータを基に、機能ごとの工数を見積もり。	**データ準備と前処理の見積もり** 内容: データ収集、クレンジング、ラベリングなど。 考慮点: データの量と質、取得難易度。 **モデル開発・トレーニングの見積もり** 内容: アルゴリズム選定、モデル構築、ハイパーパラメータ調整。 考慮点: モデルの複雑さ、トレーニングに必要な計算資源。 **モデル評価・検証の見積もり** 内容: 精度評価、クロスバリデーション、A/Bテスト。 考慮点: 評価指標の設定、テストデータの準備。 **システム統合・デプロイの見積もり** 内容: AIモデルのAPI化、既存システムとの連携。 考慮点: システムの互換性、スケーラビリティ。 **運用・保守の見積もり** 内容: モデルの再学習、性能監視、バグフィックス。 考慮点: データドリフトへの対応、モデルの更新頻度。