5 別掲資料

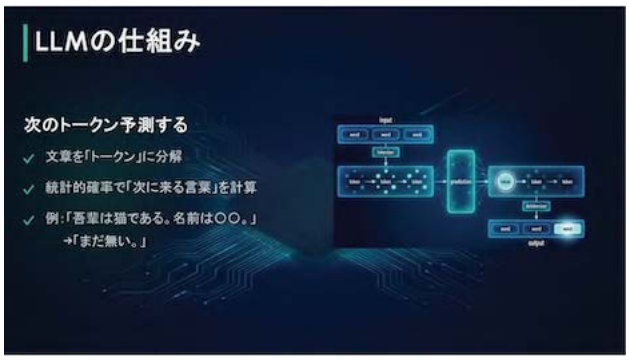
5.1 教育コンテンツ——講義資料

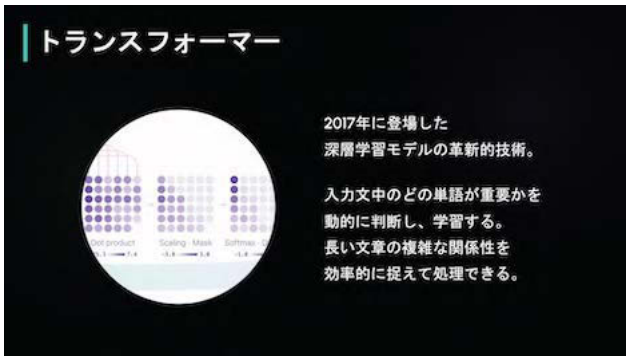


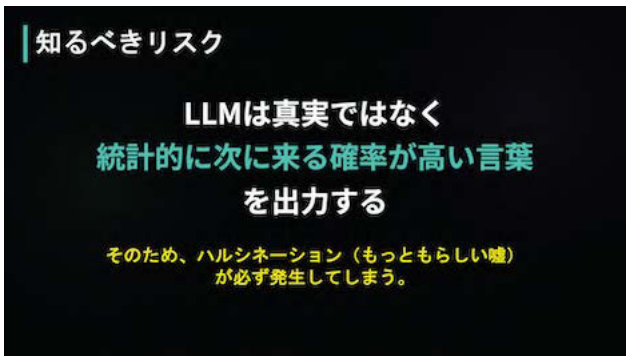














検索と生成AIの根本的な違い

検索	生成AI
図書館型	料理人型
既存のページ（URL）を探し出す。	知識という材料から新しい回答を作る。

検索と生成AIの使い分け

1. 正確性 検索：高い（一次情報） 生成AI：不確実（嘘をつく）	2. 情報の鮮度 検索：リアルタイム 生成AI：過去の学習データ
3. 得意分野 検索：事実確認 生成AI：要約、コード生成	4. 出力 検索：リンク集 生成AI：答えそのもの

現場での使い分け

検索を使う時



最新ライブラリの仕様　エラーコードの原因特定



通信障害などの最新情報

AIを使う時



コードの要約・変換



アイデアの壁打ち



中学生にもわかるように説明して

「検索と生成」のハイブリッド活用術



01
生成AIで「全体像」
構造を提案させ、
思考をショートカット。



02
検索で「裏付け」
最新の公式ドキュメントで
仕様を確認。



03
生成AIで「実装」
確認した仕様に基づき
コードを書かせる。

AIは自信満々に嘘をつくことがある。

「AIが言ったから」は通用しません。

鉄則：必ず検索でファクトチェック！

プロンプトの基本形と目的の提示の仕方

大原則：GIGO

**Garbage In,
Garbage Out**

「ゴミ」を入力すれば、「ゴミ」が出てくる。
指示（プロンプト）の質が、回答の質を決める。

「構造化」された指示へ

単なる「質問」ではなく、3つの要素で定義する。

1. 目的	2. 条件	3. 形式
役割とタスクの定義 「誰として」振る舞うか	制約とコンテキスト 範囲やトーンを限定する	アウトプットの型 Markdown、JSON等

要素1＆2：目的と条件

目的（役割とタスクの定義）

「あなたはシニアJavaエンジニアです」

役割を与えることで、
専門的な視点や用語選びを引き出す。

条件（制約事項とコンテキスト）

「Java 17, Spring Bootを使用」
「初心者向けに」

技術スタックやトーンを指定し、
回答の精度を高める。

要素3：出力形式

出力形式（アウトプットの型）

回答を「どこで使うか」に合わせて型を指定する。



Markdown形式の表で



プログラムで読み込めるよう
JSON形式で

など



コピーしやすいように
コードブロックで囲んで

など

大切なマインドセット

プロンプトは微調整で改善する！

一発で完璧な回答を出すことを目指すのではなく、
対話を通じて「微調整」を繰り返して改善するのがプロのコツ。

AIを使いこなす側へ

「目的・条件・形式」の型を持ち、
エンジニアとしての武器にしよう。

初期設定と安全利用の基本

生成AIの基礎と安全利用の前提

オプトアウトの設定

- デフォルトは「AIの学習に再利用」
- 秘密のコードやアイデアが、将来「他人の回答」として流出するリスクがある。
- 学習利用OFFを徹底しよう！



入力禁止事項（絶対に送信してはいけない3つの情報）

個人情報	機密情報	認証情報
氏名・メールアドレス 顧客リスト	未発表プロジェクト 社内マニュアル	APIキー パスワード

一度クラウドに送信したデータは、自分の手では完全に消去できません。

安全利用のための運用手順

組織でAIを利用する際は、個人の判断ではなく「決められた手順」に従います。

01 利用ツールの限定



会社や学校が許可したツール（学習されない環境）のみを使用する。

02 最終確認の義務



AI生成物は必ず人間がテスト・校閲する。AIは魔法の杖ではなく、責任は人にある。

03 記録の保持



プロンプトと出力ログを保管する。業務判断の透明性を確保するために推奨。

プロの作法と責任

抽象化・匿名化
AIに相談したい時は、
必ず「ダミーデータ」に置き換えること。

最終責任は人間
AIは魔法ではない。
テスト・校閲は人間の義務。



ＡＩは進化し続けるが 原則は変わらない

「情報の機密性を守る」
「出力の責任は人間が負う」

第1回講義まとめ

1. LLMの正体
次の言葉を予測する確率マシン
意味は理解していない

2. 使い分け
事実の確認→「検索」
構成や変換→「生成AI」

3. プロンプト
目的・条件・形式を明確にし
高品質な出力を目指す

4. 安全利用
学習OFF設定（オプトアウト）
機密情報の入力禁止の徹底

Next: 検証テクニック

AIの「嘘」を見破る実践編へ

【科目S1】生成AIリテラシーとAI倫理

02 誤りの起こり方と確かめ方

誤りの型の基本

混同・作り話・過度な一般化等

ハルシネーションとは

言葉を紡ぐマシン

AIは事実を記憶しているわけではありません。
単に「文脈に合う言葉のつながり」を確率的に計算しているだけです。
そのため、事実が不明な場合でも、確率的に「ありそうな」言葉を繋いで、もっともらしい嘘をついてしまいます。



3つの「誤り」の型

1. 作り話
存在しない関数や法律を勝手に発明してしまう「純粋な幻覚」

2. 混同
AとBの情報を混ぜて不正確な事実を作る

3. 過度な一般化
一部の解決策を全体に当てはめる

1. 作り話

AIは「いかにもありそうな名前」を確率的に生成します。

存在しないライブラリ関数 や 架空の書籍 を
自信満々に提案してくるのは、
似たパターンの学習データが大量にあるためです。



2. 混同

情報の混合

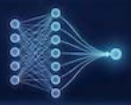
異なるバージョンの仕様や、似たプロジェクトの情報が混ざって出力されます。

例：React v16の書き方が、v18の解説に混入する。

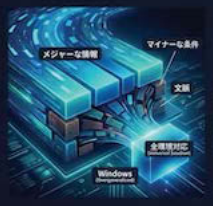


原因

AIの中で「情報の区別」が曖昧なまま、「関連性が高い」という理由だけでひとまとめに引き出されてしまうことが原因です。



3. 過度な一般化



メジャーな情報の圧力

「Windowsだけで有効な解決策」なのに、あたかも全環境で使えるかのように断定してしまうケースです。
出現頻度の高いメジャーな情報が、マイナーな条件や文脈を押しつぶして出力されてしまいます。

エンジニアの鉄則

AIの回答は
「検証が通るまでは、ただの文字列」
一歩引いた検証姿勢が、信頼性を守ります。

確かめ方の基本技術

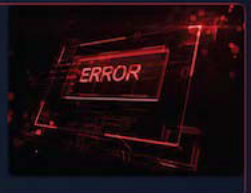
出典確認・複数照合・実行検証

未検証の恐怖

AIの回答をそのまま業務で使うことは、
エンジニアにとって**自殺行為**に等しいです。

テスト（検証）を一度もしないで、
AIの出力をそのまま本番環境にデプロイする

それと同じくらい、
恐ろしく危険な行為だと認識してください。



プロが実践する3ステップ

1. 一次情報
公式サイトやドキュメント
情報の「元ネタ」を確認

2. 複数照合
複数のAIモデルや
検索エンジンで裏取り

3. 実行検証
サンドボックス環境で
実際にコードを動かす

Step 1: 一次情報への到達

必ず自分自身で「公式サイト」や「公式ドキュメント」を確認してください。

- GitHub リポジトリ
- AWS / Google 公式ドキュメント
- RFC（技術標準）

⚠ ※AIが提示するURLは架空の可能性があります。

Step 2: クロスリファレンス

情報を重ね合わせる

マルチモデル検証：ChatGPT, Gemini, Claude等に同じ質問を投げる。

検索エンジンや技術情報サイトでの検証：キーワードをGoogleやQiitaで再検索する。



回答がバラバラなら、そこに「誤り」が潜んでいる可能性があります。⚠

Step 3: サンドボックスでの実行検証

エンジニアにとって、動くものこそが真実です。
本番環境に移行する前に、必ず隔離された「サンドボックス」で実行し、エラーや挙動を確認してください。

⚠ 動く ＝ 正しい とは限りません。
⚠ セキュリティや効率性も自身の目でレビューが必要です。



3つの重点チェック項目

バージョン番号
古い情報と新しい情報が混ざっていませんか？
廃止された機能を提案されている可能性があります。

数値・統計
桁の間違いや、単位の入れ替わりに注意。
確率は計算できても、意味は理解していません。

固有名詞・API引数
関数の引数の順番が逆ではありませんか？
「もっともらしい間違い」の常連です。

最後の一線を守る

品質を保証するのは
いつだって「人間」の仕事

AIは道具であり、責任者ではありません。

言い切り表現の点検と書き直し

AIの癖：自信満々な言い切り（断定表現）

AIは「親切なアシスタント」として振る舞うよう調整されています。
そのため、「よくわかりません」と言葉を濁すよりも、真偽にかかわらず「〇〇が正解です」とはっきり答える傾向があります。
（自信があるから正しい、とは限らない）



警戒すべきキーワード

- 間違いありません
- 100% 安全です
- 完璧に動作します

これらは全て「書き直すべきポイント」の可能性が大です。

プロの技：断定回避

Hedging（ヘッジング）

AIが出力した言い切り表現を、事実に基づいた「誠実な表現」に書き換える技術です。

⚠ ITの世界に「完璧」はありません。リスクを適切に表現しましょう。




書き換えの3つのコツ
1. 条件を付加
2. 根拠を明示
3. 可能性を残す
1. 条件を付加
「〇〇という前提では」
範囲を限定する
2. 根拠を明示
「公式資料によれば」
情報ソースを明示する
3. 可能性を残す
「傾向があります」
「考えられます」
含みを持たせる


実践例 1：脆弱性
AI (Bad)
「このプログラムに
脆弱性はありません」
責任を取りきれない断定
Pro (Good)
「現時点の静的解析の範囲内では、
既知の脆弱性は検出されませんでした」
条件と根拠を明示


実践例 2：エラー原因
AI (Bad)
「エラーの原因は
メモリ不足です」
断定すると他の可能性を
捨てることになる
Pro (Good)
「ログの傾向からメモリ不足が疑われますが、詳細な特定には追加調査が必要です」
可能性と次アクションを提示

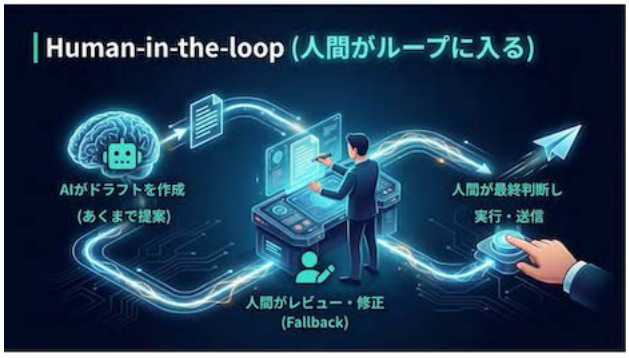

なぜ慎重になるのか？
AIの言葉をそのまま対外的な文書や報告に載せることは、
その間違いの全責任を「会社」が負うことを意味します。
後でバグが見つかった時、
「AIがそう言ったから」はプロの言い訳として通用しません。
AI


大切なマインドセット
「わからない」を正しく伝えること
正直な報告も、立派な
リスク管理能力です。
確証が持てない時は、
無理に断定せず
上司へエスカレーション
しましょう。


信頼されるエンジニアへ
AIに振り回されるのではなく、
その出力を自分の責任でコントロールする。
それがプロフェッショナルの流儀です。


自動化の限界と
人間による最終確認


自動化のパラダイムシフト
従来のAI
ルールベース
「AならばB」の定義に従う。
テストによりコントロール可能。
生成AI (LLM)
自ら判断・生成
大量データから学習したパターンに基づき、自ら判断・生成する（確率で動く）不確実なシステム。
100%の制御は不可能。


Human-in-the-loop（人間がループに入る）
AIがドラフトを作成
（あくまで提案）
人間がレビュー・修正
(Fallback)
人間が最終判断し
実行・送信

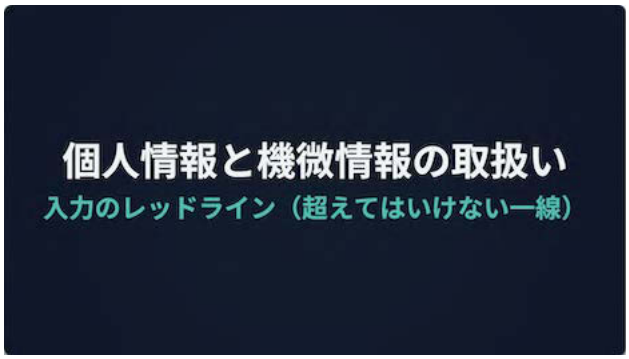

自動化の境界線
失敗の許容度とリスクに基づく判断基準
自動化 OK
失敗してもすぐに修正が効くもの
個人の学習用コード作成
アイデア出しの壁打ち
社内ドキュメントのドラフト
自動化 NG
取り返しがつかないもの
顧客へのメール自動送信
SNSへの自動投稿
検証なしの本番デプロイ


最終責任は「利用者」にある
「AIが言ったから」は、プロの世界では通用しません。
⚠ 著作権侵害 ⚠ セキュリティ事故 ⚠ 誤情報...
全ての責任はあなたと会社が負います。


第2回講義まとめ
疑う力
ハルシネーションを見破る
確かめる技術
一次情報を実行検証する
整える作法
不確実なものを誠実に伝える
守る仕組み
人間が最終責任を持つ


Next: 倫理・権利・プライバシー


【科目S1】生成AIリテラシーとAI倫理
03 倫理・権利・プライバシー


個人情報と機微情報の取扱い
入力のレッドライン（超えてはいけない一線）


なぜ入力制限が必要？
・多くの生成AIサービスでは、
入力データがサーバーに送信・保存されます。
・一度送信したデータは、AIの学習に使われる可能性があり、
自分のPCからファイルを消すようには削除できません。

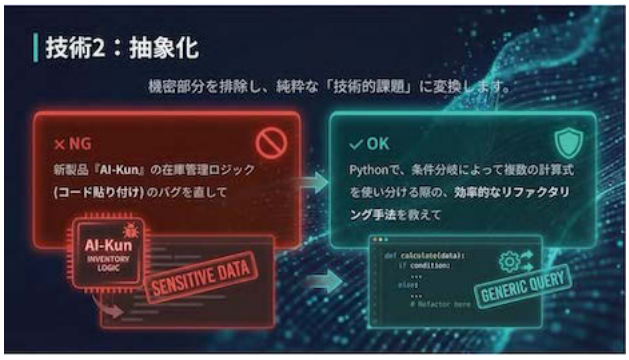

情報の「格付け」3分類
① 個人情報
個人を特定できる情報
原則禁止
② 機微情報・機密情報
漏洩＝損害
絶対禁止 (Red Line)
③ 一般情報
公開ドキュメント等
入力可能
氏名、生年月日
住所
個人のメールアドレス
マイナンバー
認証情報
未公開ソースコード
非公開の会議資料
公開ドキュメント
ホワイトペーパー
一般公開情報

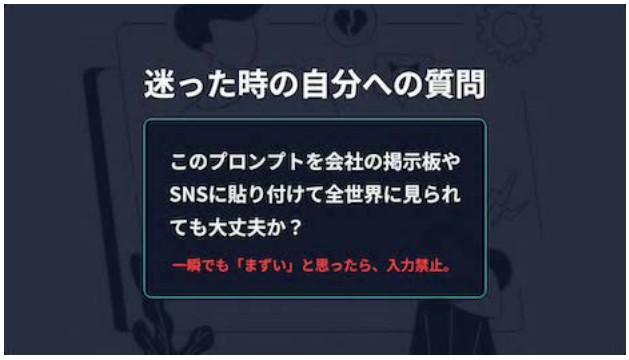

① 個人情報
特定の個人を識別できる情報です。
・氏名、住所、生年月日
・個人のメールアドレス
・マイナンバー
TEST DATA WARNING
テストデータに本物の顧客名簿の一部などを流し込むのは重大なコンプライアンス違反になる可能性があります。

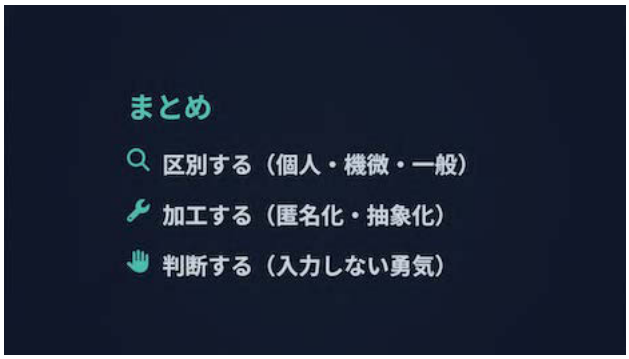

② 機微・機密情報
Red Line
漏洩すると組織に多大な損害を与える情報です。
これらを送るのは、家の鍵を道端に置くのと同じだと思ってください。
認証情報
(パスワード, APIキー)
PASSWORD: *********
API KEY: xxxxx-xxxx-xxxx
未公開コード
(独自のアルゴリズム)
def secret_algorithm():
...
非公開資料
(議事録, 仕様書)
議事録.docx
仕様書.pdf
CONFIDENTIAL


③ 業務上の一般情報
・公開されている技術ドキュメントの要約
・一般的なプログラムに関する質問など
入力可能ですが、組織のルールを確認しましょう。


技術1：匿名化
固有名詞を意味のない記号に置き換えます。
× NG
佐藤さんのメールアドレス
sato@CENSORED.com をチェックする
正規表現を書いて
✓ OK
user_id@domain.com という形式の
メールアドレスをチェックする
正規表現を書いて


技術2：抽象化
機密部分を排除し、純粋な「技術的課題」に変換します。
× NG
新製品『AI-Kun』の在庫管理ロジック
(コード貼り付け) のバグを直して
AI-Kun
INVENTORY LOGIC
SENSITIVE DATA
✓ OK
Pythonで、条件分岐によって複数の計算式
を使い分ける際の、効率的なリファクタリ
ング手法を教えて
GENERIC QUERY


迷った時の自分への質問
このプロンプトを会社の掲示板や
SNSに貼り付けて全世界に見られ
ても大丈夫か？
一瞬でも「まずい」と思ったら、入力禁止。


まとめ
区別する（個人・機微・一般）
加工する（匿名化・抽象化）
判断する（入力しない勇気）


著作権・ライセンスの基本と引用
AI出力の「権利」を守る知識


法的リスクの存在
AIが生成したコードや文章には、
目に見えない「著作権」のルールが関わります。
これらを無視すると、
製品の公開停止や法的トラブルに発展する恐れがあります。


AI生成物の著作権
自動生成物
人間がほぼ手を加えていないもの
原則：著作権なし（誰でも自由に使える）
創作的寄与
プロンプトで指示し、大幅に修正したもの
原則：著作権あり（あなたの著作物になる）


著作権侵害の判断基準
1. 類似性
MATCH
既存の著作物と内容がそっくりであること
2. 依拠性
既存の著作物を元にして作ったこと
意図せず著作権侵害に問われる可能性があることに注意！


ライセンスの落とし穴
ライセンス感染（汚染）
GPLコード等が混入すると、自社製品全体を
GPL公開しなければならなくなるなどのリスク。
著作権ロンダリング
著作権表示が消えた状態で、AIがコードだけ
を「いいとこ取り」して出力する現象。

対策: フィルタリング

GitHub Copilot等の開発支援AIを使う際の必須設定です。

公開コードに一致する提案を推奨する
→無効にして意図しない侵害を防ぐ。

引用の4要件（著作権法）

- **公表された著作物**であること
  未発表のものは引用不可
- **引用の目的上「正当な範囲内」**であること
  報道・研究等の目的で必要最小限
- **公正な慣行**に合致すること
  社会通念上、妥当な方法で引用されていること
- **主従関係**の明確性
  自分の著作物が「主」、引用が「従」という質的・量的バランス

出典表示の具体例

ドキュメントの場合

- ツールの使用方法を明記
- プロンプトと回答文の記載
- 明確な区別
- 出典情報の記載

具体例

AI Response Citation Record

「Gemini3を使用し、『生成AIの特徴を3つ挙げる』というプロンプトに対して、『1. コンテンツの新規生成能力』、『2. 自然言語による直感的な操作』、『3. 多領域への汎用性と創造性の補完』(Gemini 3, 2026) という回答を得た。

出典表示の具体例

コードの場合の表示例（プログラムコード内のコメントアウト）

JavaScript

```
/**
 * [引用・参考]
 * 引用元: [リポジトリ名/プロジェクト名]
 * 著作者: [作成者名]
 * ライセンス: MIT License (https://opensource.org/licenses/MIT)
 * 取得元URL: https://github.com/username/repo/blob/a1b2c3d4/path/to/file.js#L15-L30
 * 著作権表示: Copyright (c) 2024 [作成者名]
 * 備考: 元のコードをベースに、エラーハンドリング処理を追加して最適化。
 */

function originalLogic() {
  // 引用・改変したコードをここに記述
}
```

到達目標の確認

AIが生成したコードにはリスクがある。

- ✓ 公式ドキュメントで確認する
- 出典を正しく明記する

共有時の注意と社内外の線引き

情報の「出口」を管理する責任

「シャドーAI」の危険性

公私の線引き

個人の無料版AIで業務を行うことは、意図せず「**機密を持ち出す**」ことと同じです。

入力内容は学習に利用されるリスクが高いため、必ず組織が指定した環境で行いましょう。



組織内での「知恵」の共有

推奨

プロンプトの共有

「どう指示したか」というプロセスを共有することで、チーム全体の生産性が上がります。

危険

生の回答を丸投げ

検証なしで共有するのはNG。
必ず「自分の検証 結果・意見」を添えてください。

社外・SNSへの発信と「法的・倫理的責任」

- 「AIが書いた」は言い訳になりません。
- 権利侵害や誤情報による不利益など、その全ての法的・倫理的責任は、
- 発信したあなたが背負うことになります。
  （自分の成果物として公開する覚悟が必要）



公開前のチェックリスト

- **匿名化**
  会社名、同僚名、プロジェクト名は削除したか？
- **権利関係**
  既存ライブラリや他人のブログと酷似していないか？
- **検証済みか**
  実際に動作し、脆弱性がないか確認したか？
- **出典表記**
  「AIを活用して作成した」事実を明記したか？

コミュニケーションのマナー



誠実さ
相手はAIのテンプレではなく
あなたの意見を聞きたい

自分の言葉
AIの下書きをそのまま送らず
必ずリライトする

配慮
受け取る側の気持ちを考え
表現を調整する

到達目標の確認

- 個人のアカウントを業務に使わない
- 社外発信時は自己検証を徹底する
- AI利用の出典を明示する

迷った時の相談と記録の残し方

エンジニアとしての「自分」を守る防護柵

「たぶん大丈夫」は危険信号

- 生成AIの技術や法規制は変化し続けています。
- 昨日までの「グレー」が、今日から「禁止」になることも珍しくありません。
- 最も危険なのは「**自分一人の思い込み**」で進めることです。
- そのため、IT業界では「証跡（エビデンス）」を残すことが、自分自身の身を守ることにつながります。



「独断」を避け、「相談」をフローに組み込む



迷ったら、組織内の「**相談窓口**」を活用しましょう。
相談することは、**プロとしての防衛策**です。

あなたを守る「証跡（エビデンス）」としての記録

後から「なぜその判断をしたのか？」と問われた時、あなたを守るのは「**証跡（Evidence）**」だけです。
AIが他人の権利を侵害しても、適切な手順を踏んでいたという「**潔白の証明**」になります。



記録すべき4つの項目



迷った時の3ステップアクション

ストップ＆シンク
少しでも不安があれば
ボタンを押さない

規程の再確認
組織のガイドラインを
再確認する

公式判断を仰ぐ
上司や担当者に質問し
公式判断を仰ぐ

最終的な責任のありか

AIはあなたの代わりに責任を取ってくれません
記録を残し、透明性を確保すること
これがエンジニアの「誠実さ」であり「防衛術」です

第3回講義まとめ

入力・出力
機微情報は匿名化・抽象化
著作権侵害リスクを自覚する

共有・相談
社内外の境界線を守る
独断せず、記録を残す

Next: 組織内AIポリシーと品質評価観点

現場配属の初日から正しく動くために

実際のルールを読み解く総まとめ

【科目S1】生成AIリテラシーとAI倫理

04 組織内AIポリシーと品質評価観点

組織内AIポリシーの最小構成

プロの成果物としての品質を担保する

あなたを守る「安全柵」

AIポリシーはあなたの自由を奪うものではありません。

「ここまでは安全、ここからは危険」という境界線を示し、皆さんをトラブルから守るためのものです。

ITエンジニアとして守るべき「憲法」だと思ってください。



AIポリシーの５つの必須要素



これらはグローバルなAI倫理指針にも基づいた、プロのITエンジニアが守るべき「憲法」のようなものです。

---

1. 公平性と非差別

AIは学習データの偏りを引き継ぎます。

エンジニアの視点

データの選定段階から「このデータに偏りはないか？」と疑い、多様な背景を持つ人々に配慮します。



---

2. 透明性と説明可能性

「ブラックボックス化」を防ぎます。

エンジニアの視点

「AIがこう言ったから」で済ませず、その判断根拠を人間が理解できる形で説明する責任を持ちます。



---

3. 説明責任とガバナンス

AIは責任を取ってくれません。

最終責任は開発する人間にあります。

エンジニアの視点

リスク管理のための監視ルールを整備し「人間がコントロール（監督）している」体制を維持します。



---

4. プライバシーとデータガバナンス

データのライフサイクル全体を管理します。

エンジニアの視点

モデルインバージョン攻撃（学習データ復元）等のリスクを防ぎ、不正データの混入を管理します。



5. 安全性、セキュリティ、堅牢性

サイバー攻撃からAIシステムを防御します。

エンジニアの視点

プロンプトインジェクションや予期せぬ動作による事故を防ぐため、常にセキュリティアップデートと脆弱性対策を行い、システムの安全性を担保します。



エンジニアの心得

「知らなかった」では済みません。

配属されたら

まずはAIポリシーを読み込む

品質観点の基本（事実性・一貫性・再現性 等）

ITエンジニアが見るべき「プロの基準」

AIの生成物に求められる品質

趣味でAIを使う場合

「面白い」かどうか済む世界。

プロが仕事でAIを使う場合

実用に耐え、安全で、価値があるか。

客観的な品質の評価が必要。



ITエンジニアが点検すべき品質の３大指標

1. 正確性

正しく動くか

事実に即しているか

2. 安全性

害がないか

攻撃に強いか

3. 効率性

速いか

安いか

測定できるか

1. 機能的正確性と事実整合性

エンジニアの「腕」

AIのコードが「それっぽく」ても、エラーが出れば欠陥品です。

シンタックス エラーがないか

意図した ロジックか

事実関係に 矛盾はないか

2. 安全性、堅牢性および倫理的整合性

エンジニアの「守り」

技術的に正しくても、

倫理的にNGなら「欠陥品」です。

攻撃に対する堅牢性も重要です。

Check Points:

- 有害な表現が含まれていないか
- プロンプトインジェクション対策



3. 運用効率性とユーザー体験

エンジニアの「価値」

速度

待ち時間は許容範囲か？

(Latency)

コスト

予算内で運用可能か？

(Cost)

体験

形式は適切か？

(Satisfaction)

品質を「数値」で管理する姿勢

「なんとなく良い」ではなく、SLA（品質基準）を設定します。

正確性

95%以上

テスト通過率

応答速度

2秒以内

生成完了まで

記録

基準を満たした証跡を残す

最後の砦は「あなた」

どんなに優れたAIを使っても、

品質保証を行うのは人間です。

作業記録と承認の流れ

プロとしての「証明」を残す技術

① なぜ記録が必要なのか？

記録を残す理由は、単に「上司に報告するため」だけではありません。



② 作業記録（AIログ）の５つの必須項目



③ プロフェッショナルな「承認フロー」



④ 「例外」への向き合い方



透明性が信頼を作る

AIのブラックボックスを、

人間のプロセスで透明にする。

それがAIに支配されないプロの姿です。

見直しと改善の小さな循環

AIと「共に成長する」ITエンジニアへ

AI活用は「運用」が本番

AI導入はゴールではありません。

AIの進化速度はOSやフレームワークの更新より遥かに高速です。

一度作ったプロンプトやルールに固執せず、

小さなPDCAサイクルを回し続ける姿勢が不可欠です。

① プロンプトのバージョン管理とブラッシュアップ



② 「失敗事例」という名の資産共有



③ 現場からポリシーを動かす

ポリシーは不変の「岩」ではありません。

ボトムアップ提案

「検証の結果、安全策を講じれば導入可能」といった技術的根拠に基づき、エンジニアから組織へ提案しましょう。



AIは「道具」であり皆さんは「指揮者」である

AIを調律し続けるのは、
人間の役割です。

全4回講義の軌跡



4回の講義を通じて、皆さんはAIを安全かつ倫理的に使いこなし、AIを最大限に活用する「AI補完型ITエンジニア」の道を歩み始めました。

AIという武器を手に

「AIに代わられる人」ではなく、
「価値を生み出す指揮者」へ。

【科目S2】プロンプト設計とLLM活用

01 LLMの仕組とプロンプトの基本形

LLMの生成の流れ

確率生成・トークン

プロンプトエンジニアリング

- 前回の講義シリーズでは、
  - ・生成AIの基礎構造・限界・社会的影響
  - ・リスク評価と倫理的利用判断
  - ・組織内AIポリシー
  - などについて学びました。
- 今回の講義シリーズでは、
  LLMを自在に操り、高品質なシステムを生み出すための、
  プロンプトエンジニアリングの世界に入ります。



① トークン：AIが理解する「最小単位」

LLMにとっての最小単位は、
「文字」でも「単語」でもなく、「トークン」と呼ばれる断片です。



日本語トークンの数え方



英語

短い単語：1単語 ≒ 1トークンで計算
長い単語：複数のトークンに分割
例: apple → 1トークン
friendship → friend + ship（2トークン）



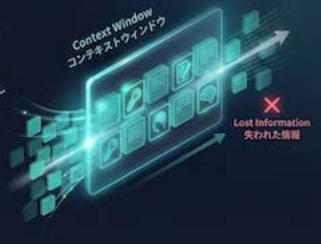
日本語

漢字1文字 ≒ 1〜2トークン
ひらがなも複雑に分割される。
文字の種類（漢字・ひらがな・カタカナ）が多いため、同じ意味の文章でも、
英語よりもトークン数が膨らむ傾向がある。

トークンの重要性 1：情報量の限界

コンテキストウィンドウ

LLMが一度に記憶・処理できる情報の量は、トークン数で決まります。
無駄に長いプロンプトは、重要な情報を押し出してしまうリスクがあります。



トークンの重要性 2：コスト

従量課金制

API利用料の多くは「1,000トークンあたり数円」という体系です。
冗長なプロンプトは、そのままコストの増大に直結します。



② LLMの生成の仕組み（トークンの予測と自己回帰）

LLMの生成は、驚くほど地道な「繰り返し作業」です。



③ 確率の計算：ソフトマックス

確率分布と温度による制御

AI内部の確率分布

LLMは、次に続く可能性のあるすべてのトークンに対して「確率」を割り振ります。

エンジニアの視点：温度（Temperature）

「温度」は、この確率分布をどう扱うかを決めるパラメータです。



高温度：意外性・創造性。確率の低いトークンも選ばれる。

高温度の回答例

例：「ログインパスワードと同じなので教えられない」（現代的なデジタル猫）

なぜAIの出力には「揺れ」があるのか

確率のゆらぎ

LLMの生成は確率に基づいているため、
常に同じトークンが選ばれるとは限らない。
そのため、同じプロンプトを入力して、
微小な確率の差で全く違う文章が生成されることがある。



なぜAIは「嘘」をつくのか

AIの出力は事実ではなく確率に基づくから

AIは事実を探しているのではなく、
統計的に「もっともらしいトークンの並び」
を作っている。



そのため、文脈的に自然であれば、
事実と異なること（嘘）のトークンが選ばれても、
AI自身はそれに気づけない。

出力：文脈的に自然なため、事実と異なること（嘘）が選ばれた。AI自身はそれに気づけない。

エンジニアの役割

「確率の波」を、
プロンプトという
「防波堤」で制御せよ。

入力長と設定が出力に与える影響

なぜ精度が落ちるのか？ なぜ答えが変わるのか？

出力の「揺れ」と「劣化」

同じプロンプトを投げているのに、
なぜ、毎回少しずつ答えが変わるのか？
なぜ、急に精度が落ちることがあるのか？



その原因は「入力の長さ」と「設定」にあります。

入力の長さと「コンテキストウィンドウ」の限界



このコンテキスト内での情報見落としのことを「中だるみ」と呼びます。

パラメータの設定①：温度

温度（Temperature）- AIの「慎重さ」を調整するダイヤル

温度を「低く」
(Low Temperature)：
AIは常に最も確率が高い「王道」の言葉を選ぶ。
→デバッグや事実確認など、正確性と再現性が必要な場合は、温度を低く設定する



温度を「高く」
(High Temperature)：
→キャッチコピーや物語など、「意外性」が必要な時は、温度を「高く」設定する

パラメータの設定②：Top-P

Top-P（核サンプリング）

候補となる言葉の「足切りライン」を決める設定

確率の高い順に並べて、合計が上位P%になるまでの単語に候補を絞り込む。

➡ 例：Top-Pを0.9（90%）に設定すると、確率の高い順に単語を並べて、合計が90%に達するまでに含まれる「上位の候補」だけに選択肢を絞り込み、残りのわずかな可能性は完全に切り捨てる。



温度とTop-Pの違い

温度 と Top-P は似ていますが、次のような違いがあります。

温度

各単語の「当選確率」そのものを変える。



Top-P

選択肢の「範囲」を決める。

(足切りラインの設定)



エンジニアの「2つのモード」

実務では、これらを組み合わせて使い分けます。

正確性重視の場合
（デバッグ・コード生成・要約）

温度：0.0-0.3／Top-P：低め 毎回（ほぼ）同じ結果を出す。



創造性重視の場合
（アイデア出し・物語）

温度：1.0 - 1.5／Top-P: 高め 表現を豊かにする。



ブラウザ版LLMでの対処法

ブラウザ版では温度やTop-Pを直接数値で指定することはできません。
数値の代わりに、「プロンプトによる指示」で制御します。

正確性重視の指示例



「事実のみを簡潔に述べてください。」
「推測や創造的な表現は避けてください。」

創造性重視の指示例



「自由な発想で、独創的かつユーモアを交えて多様なアイデアを出してください。」

再現性の確保

「同じ入力には、同じ結果を」

AIの気まぐれを制御し、信頼されるツールへ。

基本プロンプトの黄金律

目的・条件・出力形式

プロンプトの本質

AIへの指示は、単なる「お願い」ではありません。
ITエンジニアにとってのプロンプトは、
システム開発における「仕様書」や「命令文」であるべきです。
(曖昧な指示は、仕様未定の開発と同じ)



プロンプトの3大要素



1. 目的

誰が（役割・ペルソナ）
何を（最終的なゴール）



2. 条件

「やってはいけないこと」
「必ず守るべき仕様」



3. 出力形式

「回答の見た目の指定」

① 目的

役割を与える

単に「教えて」ではなく、「役割（ペルソナ）」と「最終ゴール」を設定します。

悪い例

SQLのクエリを書いて。

プロの例

あなたはDB管理のスペシャリストです。
パフォーマンス改善のために、以下の処理を行う効率的なSQLクエリを生成してください。

② 条件

出力の精度に最も直結する部分です。
AIが「やってはいけないこと」や「必ず守るべき仕様」を明確にします。

技術スタック

言語はTypeScript、WEBフレームワークはNext.jsを使うこと

対象レベル

プログラミング初学者にもわかるように、1行ごとにコメントを付けること

禁止事項

外部ライブラリは使用せず、標準機能だけで実装すること

③ 出力形式

後の処理がしやすい形を指定します。
エンジニアには必須の視点です。

Markdown
Markdownで、見出しと箇条書きを使い、コードはコードブロックで囲むこと。

表形式
表形式で、メリットとデメリットを比較表にして提示すること。

JSON形式
プログラムで解析できるよう、必ず有効なJSON形式で出力すること。

【実践比較】素人とITエンジニアのプロンプト

同じ「エラー解決」を依頼する場合でも、指示の出し方によって出力結果は大きく変わります。
AIに丸投げせず、プロンプトの構造化を意識してください。

	素人のプロンプト	ITエンジニアの構造化プロンプト
指示内容	Pythonで次のエラーが出たので直して。[エラー文]	あなたはPythonのデバッグ担当です。以下のエラーを解析し、原因と修正案を提示してください。[エラー文]
制約	なし	1. エラーの根本原因を特定する 2. 修正後のコード全体を示す 3. 同じミスを防ぐためのアドバイスを1つ添える
出力形式	AI任せの文章	以下の構成で出力してください: 【原因】 【修正コード】 【ポイント】

AIに丸投げせず、プロンプトの構造化を意識してください。

なぜ構造化が必要なのか？

LLMは「確率」で動いています。

条件を指定すること＝正解へと続く道に案内板を設置してあげること

- 選択肢（トークンの候補）を絞り込んであげることで、
- ハルシネーション（嘘）が入り込む余地を減らし、
- 安定した高品質な回答を引き出すことができる。

エンジニアのアドバイス

テンプレート化

毎回手入力するのは非効率です。
よく使うパターンはテンプレートとして保存し、
『プロンプトの再利用性』を高めましょう。
開発効率を上げる重要な技術です。



プロンプトの黄金律

指示が具体的であればあるほど、
「確率のブレ」は少なくなる。

入力制約と記録（入力・設定・出力）

実務で最も差がつくITエンジニアの資産

実験と検証の繰り返しをプロンプトにも

- 1回でプロンプトが成功することは稀です。
- 科学者が実験データを蓄積するように、エンジニアも記録を残します。
- たまたまうまくいってOKとするのではなく、
- **何度でも、誰でも再現できる状態を目指しましょう。**



① 入力制約の徹底（情報の加工）

エンジニアの防衛術

実務データはプロンプトに入力してはいけません。
必ず**「匿名化されたデータ」**を使いましょう。

- **ダミーデータ**
  構造だけ維持した架空データを使う。
- **コードの匿名化**
  社内ライブラリ名を一般名称に置換する。



② 記録すべき4つの項目（高品質プロンプトの特定）

入力（プロンプト）	設定（パラメータ）	出力（結果）	評価と改善案
どんなプロンプトが効いたか	使用モデル 温度設定	結果をそのまま残す	満足度・エラーや改善案

③ 再現性を確保する（デバッグの考え方）

プログラムのバグ修正と同じく、
出力を安定化させましょう。

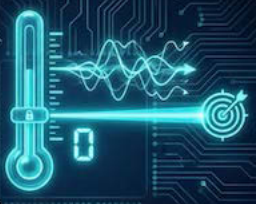
3回テスト
LLMの出力結果の「揺れ」を前提とし、
同じプロンプトを３回試行し、
毎回合格点が出るかを確認しましょう。



検証中は温度設定を固定

Temperature = 0

検証中は温度を「0」に固定します。
これにより、AIの「気まぐれ」を排除し、
プロンプトの純粋な性能のみをテストします。



- 検証中は温度を「0」に固定します。
- これにより、AIの「気まぐれ」を排除し、プロンプトの純粋な性能のみをテストします。

記録を「論理的な根拠」にする

「なぜこのAI回答を
採用したのか？」
と問われた場合に

✕

感覚で選んだと
答えるのはNG。



を根拠として提示せよ！

プロのエンジニアとしての説明責任

第1回講義まとめ



仕組み
トークンと確率予測
による「自己回帰」

制御
- 入力の長さ
- 温度設定

設計
目的・条件・形式
（構造化された指示）

記録
入力制約の厳守
ログによる再現性確保

Next: 基本プロンプトでの結果の揃え方

プロンプトの基本練習と
出力改善ループの実践

【科目S2】プロンプト設計とLLM活用

02 基本プロンプトでの結果の揃え方

再現性のあるコーディング演習

テンプレートに基づくプロンプト作成

ITエンジニアのための「攻め」の技術

講義の狙い

単に動くコードではなく、
「堅牢で、何度実行しても構造が揃う」
コードを出力させる技術を習得します。

AI生成コードの現実（調査結果）：

75%多い
ビジネスロジック、誤った依存関係、制御フローの欠陥
（人間との比較）

2倍多い
エラーハンドリングや例外処理の欠落
（人間との比較）

ケーススタディ：在庫チェックを伴う注文処理

ショッピングサイトのバックエンド機能
注文処理ロジックを作成します。

TypeScript関数の作成

- ✓ 注文リクエスト受信
- ✓ 在庫チェック
- ✓ 合計金額計算（税込）
- ✓ 処理結果の成否を返す



悪いプロンプト（丸投げ）

「在庫数と注文数、単価を受け取って、在庫があれば合計金額を計算して返す関数をTypeScriptで書いてください。在庫がなければエラーにしてください。」

指示が曖昧で、仕様が定義されていません。

出力結果に「揺れ」が生じる可能性

- ✕ 引数の型が any になる
- ✕ 税計算で 誤差 が出る
- ✕ 戻り値が 揺れる (null? error?)

良いプロンプト（制約付き）

目的
在庫引き当てと金額計算を行う堅牢な関数 `processOrder` の作成

条件（制約事項）
- 型定義: `OrderRequest` インターフェースを定義し、
  項目名、単価、注文数、そして在庫数を引数にとるよう指定
- バリデーション: 負の値は具体的なメッセージと共に `Error` を投げる
- 計算精度: Math.floorで端数処理
- エッジケース: 在庫0も考慮
- セキュリティ: 標準機能のみ使用

出力形式の固定

構造を指定することで、実装のブレを防ぎます。

```
// 以下の構造を守ってコードのみを出力せよ
interface OrderRequest {...}
interface OrderResult {...}
function processOrder( req: OrderRequest, stock:
number ): OrderResult {...}
```

```
TypeScript
// ... (interfaceの定義を省略)
function processOrder(req: OrderRequest, stock: number): OrderResult {
  if (req.quantity <= 0) {
    throw new Error("注文数は1以上を指定してください。");
  }
  if (req.unitPrice <= 0) {
    throw new Error("単価は1以上を指定してください。");
  }
  if (stock < req.quantity) {
    throw new Error("在庫が不足しています。");
  }
  const subtotal = req.unitPrice * req.quantity;
  const tax = Math.floor(subtotal * 0.1);
  const totalAmount = subtotal + tax;
  const remainingStock = stock - req.quantity;
// ... (return以下を省略)
```

結果が「揃う」メリット

レビュー効率化

入力と出口が固定されるため
「中身のロジック」だけに集中できる。

後続工程との連携

形式が固定されていれば
テストコード生成等へそのまま投入できる。

再現性を高めるガードレール

否定の制約

「標準以外使わない」
「多用禁止」等、
何をしないかを指定。

境界条件

「0の時」・「負の時」など
AIが間違いやすい急所を
先回りで指示。

型の強制

Interfaceを定義し
AI自身の論理矛盾を
自己抑制させる。

反復実行による揺れの観察

出力の「差分」を観察し、改善せよ

揺れの放置はリスクである

LLMの出力には必ず「揺れ」が生じます。
これを放置することは「品質が運任せになるコード」を
製品に組み込むことと同じです。



実際にプロンプトを３回実行し、差分（出力の違い）を検証してみましょう。

２回目の出力結果

```
TypeScript
(interfaceの定義を省略)
/**
 * 在庫引き当てと金額計算を行う関数
 */
function processOrder(req: OrderRequest, stock: number): OrderResult {
  // バリデーションチェック
  if (req.quantity <= 0) {
    throw new Error(`不正な注文数です: ${req.quantity}。0より大きい値を入力してください。`);
  }
  if (req.unitPrice <= 0) {
    throw new Error(`不正な単価です: ${req.unitPrice}。0より大きい値を入力してください。`);
  }
  if (stock < req.quantity) {
    throw new Error(`在庫不足（現在の在庫: ${stock}, 注文数: ${req.quantity}）`);
  }

  const totalPriceExclTax = req.unitPrice * req.quantity;
  const taxAmount = Math.floor(totalPriceExclTax * 0.1);
  const totalPriceInclTax = totalPriceExclTax + taxAmount;
  const remainingStock = stock - req.quantity;
(return以下を省略)
```

２回目の出力結果の特徴：運用・チーム開発への配慮

1. JSDocによる「コードの仕様」の明文化

プロンプトの「堅牢な（壊れにくい）」という要求を、単にエラーが出ないことではなく、
「後から読む人が使い方を間違えないようにすること」と解釈しています。

2. エラーメッセージへの「具体的コンテキスト」の付与

エラー通知に留まらず、「なぜそのエラーが起きたのか」を即座に特定できる情報が含まれている。
現在の在庫数などの動的な数値も埋め込んでおり、ログを見ただけで原因が特定できる。

3. 曖昧さを排除した「厳密な命名規則」

名前から「何が含まれている数値か」を直感的に理解できるよう、あえて詳細な名前を選択している。
`subtotal`（小計）…広義な名前→`totalPriceExclTax`（税抜合計価格）…厳密な名前に変更。
後の工程での計算ミスや、仕様の勘違いによるバグを未然に防ぐ。

３回目の出力結果

```
TypeScript
(interfaceの定義を省略)
function processOrder(req: OrderRequest, stock: number): OrderResult {
  // 異常値の除外
  if (req.quantity <= 0 || req.unitPrice <= 0) {
    throw new Error("注文数および単価には0以下の数値を設定できません。");
  }

  // 在庫確認（在庫がちょうど0になる場合は許容）
  if (req.quantity > stock) {
    throw new Error("注文数が現在の在庫数を超えています。");
  }

  const amountExcludingTax = req.unitPrice * req.quantity;
  const consumptionTax = Math.floor(amountExcludingTax * 0.10); // 10%の税率を適用、端数は切り捨て
  const amountIncludingTax = amountExcludingTax + consumptionTax;
  const remainingStock = stock - req.quantity;
(return以下を省略)
```

３回目の出力結果の特徴：正確性とメンテナンス性の両立

1. ドメイン（業務）に基づいた厳密な命名

単なるプログラミング用語ではなく、会計や事務などの実業務で使われる用語を変数名に採用している。
`totalPrice`（汎用的な名前）→`amountExcludingTax`（税抜金額）や`consumptionTax`（消費税）
（具体的な用語）に変更。

2. 「ガード節」によるロジックの整理

複雑になりがちなチェック処理（バリデーション）を、効率的な構造に書き換えている。
複数の`if`文で階層化させるのではなく、異常な条件を冒頭でまとめて弾く「ガード節」を採用。

3. 「意図」を伝えるプロセス・コメント

「何をしているか」だけでなく、「なぜそうしているか」という開発者の意図を言語化している。
在庫がちょうどゼロになるケースは許容する」といった、仕様上の迷いが生じやすい箇所に注釈を挿入。
後任の担当者が「これはバグか、仕様か」と悩まないようにしている。

AIの思考の「揺れ」

同じ指示でも、優先する価値観が変わります。

同じ指示

- シンプルさ — 最低限の機能実装 (Run 1)
- デバッグ容易性 — 詳細なエラーログ (Run 2)
- ビジネス厳密性 — 正確な用語と構造 (Run 3)

プロンプトエンジニアリングの極意

揺れを「不安定」と切り捨てない。

揺れの観察をヒントに条件を書き足し、
理想の回答に固定していく。

条件追加・形式固定による調整

AIの「自由度」を奪い、再現性を高める

「推測」を封じ込める

AI生成コードの問題点、特にコンパイルエラーの94%は「型チェックの失敗」です。
AIが勝手な想定でコードを書く「空白」を埋めるため、
4つのガードレールを設置します。



1. 型による「契約」の固定

最も強力なガードレールは
プロンプト内でインターフェースを厳格に定義することです。

```
// 以下の型を必ず使用すること
interface OrderRequest {
  itemName: string;
  unitPrice: number; // 円
  quantity: number;
}

interface OrderResult {
  itemName: string;
  unitPrice: number;
  quantity: number;
  subtotal: number; // 税抜合計
  tax: number;      // 消費税 (10%)
  total: number;    // 税込合計
  remainingStock: number;
}
```

2. 境界条件の「受入基準」化

AIはエラーハンドリングを省略しがちです。

- 条件を明記
  - 「在庫0」の扱い
  - 「負の値」の挙動を指示
- メッセージ固定
  - 「現在の在庫数を含める」など



3. 依存関係の「封鎖」

ハルシネーション対策
AIは存在しない便利なパッケージを
捏造することがあります。
「悪意あるパッケージ」を掴まされる
リスクを排除するため、標準機能の
みに限定します。



4. 処理プロセスの「段階化」(Chain of Thought)

計算ステップの省略や誤差を防ぐため、手順を指示します。
(Chain of Thought)



1. バリデーション
2. 合計金額算出
3. 端数処理

調整後のプロンプト (定義)

```
# 目的在庫引き当てと金額計算を行う堅牢な関数 processOrder を作成する。
# システム制約：言語じゃTypeScript (strictモード)
外部パッケージ使用禁止。
標準の Math クラスのみ使用。
# 下記のインターフェースを必ず使用し、変更しないでください。
TypeScriptinterface OrderRequest { unitPrice: number; quantity: number; }
interface OrderResult { success: boolean; totalAmount: number; message:
string; }
```

調整後のプロンプト (定義)

```
# アルゴリズム手順
1. バリデーション: qty <= 0 or price <= 0 なら Errorを投げる。
2. 在庫チェック: stock < qty なら success: false と 「在庫不足
（現在: ${stock}）」を返す。
3. 計算: subtotal = price * qty tax = Math.floor(subtotal
* 0.1) total = subtotal + tax
4.終了: success: true と結果を返す。
```


調整後のプロンプトを再度3回実行し、以下を確認します。
型定義の一致
インターフェースが完全一致しているか
ロジックの統一
端数処理やエラー文言が同じか
安全性
危険な関数や謎のパッケージが含まれていないか
再現性確認済


KANBAN BOARD
設計とは制約である
「条件」として書き込んだ内容は、
そのまま「評価指標」になります
VALIDAT


差分記録の作成と提出
ITエンジニアにとっての「証跡」の確立


差分記録の作成と提出方法
エンジニアリングとしての「証跡」
実務では、「どの制約が、どのリスクを回避するために追加されたのか」を論理的に説明し、記録に残す必要があります。
開発プロセスへの統合
ここでは、後続の「評価」や「改善ループ」、そしてチーム開発におけるコードレビューの土台となる差分記録の作成と提出方法について学びます。
評価
差分記録
改善ループ
コードレビュー


記録すべき必須要素（前半）
1. 入力の変遷
初期:曖昧なプロンプト
比較記録
最終版:ガードレール追加
初期の「曖昧なプロンプト」と、ガードレールを追加した「最終版」を比較記録。どの条件が効いたかを特定する。
2. 推論設定
MODEL: GPT-4o
TEMP: [Value]
使用モデル（GPT-4o等）とTemperature 設定値を記録。再現性の担保に必須。


記録すべき必須要素 (後半)
3. 揺れと収束
調整前の不一致（型矛盾、命名揺れ）と、調整後の収束状況をコード断片で示す。
Before/After
4. 判断根拠
「なぜ直したか」を記述。
例:「在庫0判定が漏れていたため、境界値を明記した」
Why?
Rationale & Example


Diff分析の「急所」
品質リスクの改善を見ます。
I/F整合性
型チェック失敗が排除されたか？
例外網羅
負の値やNullがガードされたか？
幻覚排除
標準ライブラリのみに限定されたか？


提出時の遵守事項
機密保護
実務コードや個人情報をそのまま含めない。
トレーサビリティ
将来の脆弱性対応のための追跡ログとなる。
エスカレーション
未解決の懸念事項は正直に明記する。


第2回講義まとめ
テンプレート作成
目的・条件・形式を構造化し、曖昧さを排除した「設計図」をAIに提示する。
反復実行
同一条件で複数回実行し、AIの「揺れ」を可視化する。
条件追加・調整
型・境界・依存・段階化のガードレールでAIの自由度を奪い、再現性を高める。
差分記録
修正の判断根拠とBefore/Afterの証跡を残し、品質向上を論理的に説明する。


Next: 出力評価の基本
今回の記録が「採点基準」になる。
事実性・一貫性・再現性
の3軸による出力評価

【科目S2】プロンプト設計とLLM活用

03 出力評価の基本 ― よい答えの見分け方

事実性・一貫性・再現性の点検

3観点（事実性・一貫性・再現性）の点検

品質の「急所」を見抜く目を養う

評価の3観点

1. 事実性
現実の事実
信頼できる情報源
と一致しているか

2. 一貫性
プロンプトで指定した
「条件」や「型定義」
を守っているか

3. 再現性
複数回実行しても
同等の品質が得られるか

AIコードの品質リスク

AI出力の盲信は危険

ITエンジニアとして、
厳格な出力品質の評価
が必要です。



ビジネスロジックの欠陥

人間より

75%

多い

というデータもあります。

① 事実性

一見して10%の消費税を端数切り捨てで計算しており、指示通りに見えます。

```
const subtotal = req.unitPrice * req.quantity;
const tax = Math.floor(subtotal * 0.1);
const totalAmount = subtotal + tax;
const remainingStock = stock - req.quantity;
```

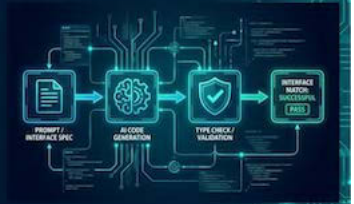
「インボイス制度」という実務コンテキストに照らして、確認する必要があります。
この例は合計額に10%を掛け、1円未満を切り捨てる（Math.floor）計算なのでOK。

② 一貫性

AI生成コードのエラーの94%は「型チェックの失敗」です。

✓評価ポイント

指定した OrderRequest
等のインターフェースを正確
確に使用しているか？
ここを死守しているなら高ス
コアです。



③ 再現性

何度実行しても同等の品質が得られるか。
特定の回だけ「たまたま」良い出力を出していないか？

ガード節による安定
整理された構造が毎回出力される
「安定感」こそが、
実務で採用できるかどうかの境界線。

検証優先箇所の特定（数値・固有名詞）

AIの弱点を突くチェックポイント

2つの検証優先ポイント

 # 数値と境界条件
→境界値分析 (BVA)が有効。
「値の境界（最大値・最小値の端）」
に注目し、その点と、その直上の
値・直下の値を検証する。

固有名詞と外部依存
パッケージハルシネーション
→スロップスクワッティング
などのリスク

これらはAIが特に苦手とする領域です。

数値と境界条件のチェック

3回目の出力結果にある条件式に注目してください。

```
// 在庫確認（在庫がちょうど0になる場合は許容）
if (req.quantity > stock) {
  throw new Error("注文数が現在の在庫数を超えています。");
}
```

在庫が0になる瞬間であっても正常系の処理に進みます。

固有名詞と外部依存のチェック

出力結果にある条件式に注目してください。

```
// 3.金額計算（浮動小数点誤差と端数処理への配慮）
const subtotal = req.price * req.quantity;
const tax = Math.floor(subtotal * 0.1);
const total = subtotal + tax;
```

今回の出力は「標準の Math クラスのみ使用」という制約も遵守しています。

品質ゲートと停止基準の設定
どこまで修正を試みるかの判断基準



品質ゲートとは

ソフトウェア開発において、ある工程から次の工程へ進む（あるいはリリースする）ために満たすべき「最低限の品質基準・条件」のことです。

このゲートを通過できない場合、たとえスケジュールが迫っていても作業は次工程へ進めず、不具合の修正や基準達成のための修正が必要となります。

必須の最低ライン



最低でも以下の基準を満たす必要があります。

- **即時かつ最優先で対処すべき最も深刻な脆弱性がないこと**
  クロスサイトスクリプティングなどの深刻なリスク排除
- **指定したInterfaceとの適合率100%**
  指定したInterface（API、関数シグネチャ、データ型、スキーマなど）との適合率が100%でないと、コンパイルエラーやランタイムエラーを起こす可能性があります。

「停止基準」を設ける

特定の誤りを3回以上繰り返す場合は、再プロンプトを中止します。

例：「消費税10%を5%で計算し続ける」



なぜ中止するのか？

泥沼化	技術的負債	文脈理解
自己修正能力には限界がある	修正ループは開発速度を低下させる	AIのコンテキスト理解の限界点

何度やっても同じ誤りを繰り返す場合、いったん人間が引き取ります。

「停止後のアクション」

- **人間が介入**
  直接コードを修正する
- **分解・再構築**
  要件をより小さな単位に分ける



「AIを使いこなす」／「AIに振り回される」の境界線を、停止基準によって明確化する。

判断根拠の作業記録
共有前提の証跡を残す

説明責任を果たすために

なぜそのコードを「良し」としたのか。
これは個人のメモではなく、チームやクライアントに対して説明責任を果たすための「証跡」です。

「動作確認済み」の一言では不十分です。

具体的な根拠の記述例 ①



I/F適合率 100%
指定した TypeScriptの Interfaceとの完全一致を確認。



境界値チェック
在庫と注文数の境界値の挙動を目視とテストで点検済み。



ビジネスロジック
金額計算において端数処理が明示されているが、合算ルールについては「上位モジュールで検討必要」と判断。

具体的な根拠の記述例 ②



ハルシネーションなし

標準ライブラリ以外の依存がないことを確認。



デバッグ性評価

エラーメッセージに動的な値が含まれており、運用しやすいと判断。

記録を残すメリット



改善ループ

「どの部分を、なぜ直すべきか」AIへの的確なフィードバックが可能になる。



追跡可能性

将来障害が起きた際、「なぜそう判断したか」を追跡できる。

第3回講義まとめ

1. 3つの観点

事実性・一貫性・再現性で厳格に点検する

2. 優先箇所

境界値の不等号やパッケージ捏造を疑う

第3回講義まとめ

3. 停止基準

品質ゲートで担保しループ時は人間が介入

4. 判断記録

採択理由の根拠を残しチーム開発の鍵とする



品質の門番へ

AIを盲信する「利用者」から、**出力を厳格に審査する「品質の門番」**への進化。

Next: 再プロンプトによる改善ループ（演習）

評価表で問題点を抽出

再プロンプトで是正