Microsoft

GitHub Copilot
プロンプトリファレンス

2026年1月版

本書について

プロンプトとは、ユーザーが生成 AI に与える「指示」や「質問」のことです。
どのような結果を出力してほしいのかを具体的に伝えるためのメッセージであり、その質や内容が AI の生成する文章やソースコードなどのアウトプットに大きな影響を与えます。

本書では、GitHub Copilot からより適切な応答を生成するためのプロンプトテクニック集を記述します。

目次

1. **GitHub Copilot をはじめる**
   - 1.1 GitHub Copilot のプラン
   - 1.2 Premium リクエスト
   - 1.3 GitHub Copilot の有効化手順
   - 1.4 モデル別に適した用途の紹介
2. **一般的なプロンプトテクニック**
   - 2.1 出力値の実例を提示する
   - 2.2 具体的な指示を与え、推測を減らす
   - 2.3 役割を与え、振る舞いを定める
   - 2.4 出力フォーマットを明記する
   - 2.5 ステップバイステップで考えさせる
   - 2.6 複数回のやり取りで会話履歴を活用する
   - 2.7 GitHub Copilot 自身にプロンプトを生成・改善させる
3. **GitHub Copilot 特有の機能を用いたプロンプトテクニック**
   - 3.1 既存のソースコードやプロジェクトを説明させる
   - 3.2 エラー修正やリファクタリングを指示する
   - 3.3 単体テストを実装させる
   - 3.4 プロジェクト全体をコンテキストとして扱う
   - 3.5 ターミナルの実行結果をコンテキストとして扱う
   - 3.6 Visual Studio Code に関する質問を行う
   - 3.7 明示的に参照ファイルを指定する
4. **GitHub Copilot カスタマイズ機能**
   - 4.1 GitHub Copilot に自動適用されるカスタム命令
   - 4.2 Coding agent で自動適用されるカスタム命令
   - 4.3 再利用可能なカスタムプロンプトファイル
   - 4.4 MCP を利用して外部データをコンテキストとして扱う
5. **GitHub Copilot におけるエージェントコーディング手法紹介**
   - 5.1 Agent mode (Copilot Edits)
   - 5.2 Coding agent
   - 5.3 GitHub Copilot CLI
   - 5.4 Copilot コードレビュー
   - 5.5 スペック駆動開発
6. **ユースケース**
   - 6.1 プロジェクト設計およびタスク化
   - 6.2 既存プロジェクト分析
   - 6.3 マイクロサービスプロジェクトにおける追加機能実装
   - 6.4 単体テストおよび自動化ワークフロー実装
   - 6.5 脆弱性パターンの検出とレポート出力
   - 6.6 SDLC におけるプロンプト例
   - 6.7 ポリシーを使用して安全に GitHub Copilot を使用する

1. GitHub Copilotをはじめる

1.1 GitHub Copilot のプラン

1.1 GitHub Copilot のプラン（ビジネス向け）

GitHub Copilot には、用途やライセンスおよびポリシー管理体制に応じて、複数のプランが用意されています。

ビジネス向けとしては、以下 2 つのプランが利用可能です。

- **GitHub Copilot Business**
  - Organization メンバーの一元管理と Copilot ポリシー制御が可能になります
  - Copilot コーディング エージェント が含まれます
- **GitHub Copilot Enterprise**
  - Copilot Business のすべての機能に加えて、より多くの Premium リクエストの使用枠が含まれます
  - 全てのモデルへのアクセスが含まれます

1.1 GitHub Copilot のプラン（個人向け）

個人の開発者（Organization または Enterprise を通じて Copilot にアクセスできないユーザー）が使用できるプランは、以下の通りです。

- **GitHub Copilot Free**
  - 一部の Copilot 機能への制限付きアクセスが含まれます
- **GitHub Copilot Pro**
  - 無制限の補完、Copilot Chat での Premium モデルへのアクセス、Copilot コーディング エージェント へのアクセス、毎月の Premium リクエストの使用枠が含まれます
- **GitHub Copilot Pro+**
  - Copilot Pro のすべての機能に加えて、より多くの Premium リクエストの使用枠、Copilot Chat で使用できるすべてのモデルへのフル アクセスが含まれます

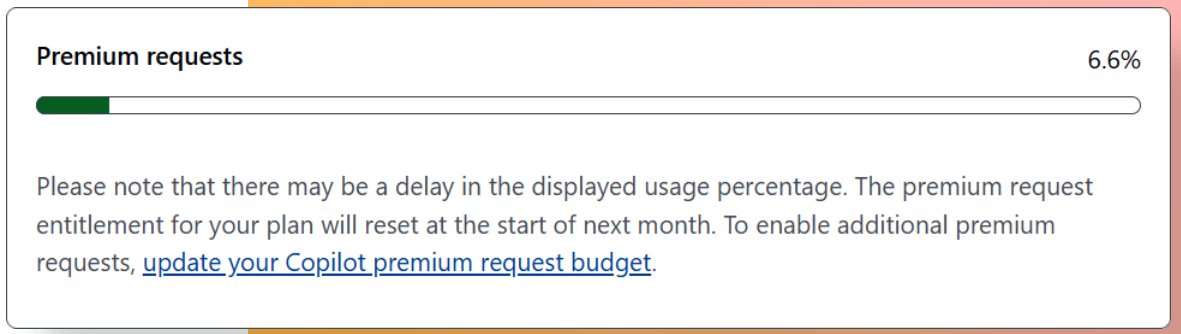
1.2 Premium リクエスト

1.2 Premium リクエスト

一部の Copilot 機能はより高度な処理能力が必要であり、特定処理の回数が Premium リクエストとしてカウントされます。
使用モデルごとに乗数が異なります。

例: Copilot Chat で Claude Opus 4.1 を使用する場合は 10 倍の乗数が適用され、1 回のやり取りが 10 Premium リクエストとしてカウントされます。(2025/12 現在)

参考: GitHub Copilot における要求 - GitHub Enterprise Cloud Docs



1.2 Premium リクエスト 上限数に関して

プランごとに月あたりの最大 Premium リクエスト数が付与されます。

上限に達した場合も、乗数が 0 のモデルを使用する機能であれば引き続き使用することができます。

最大 Premium リクエスト数の超過分を追加購入することもできます。

- “Premium リクエスト有料使用” ポリシーが有効になっている必要があります
- 予算設定で、追加支出が許容されている必要があります

最大 Premium リクエスト数の例：

- Copilot Business の場合は 1 ユーザーにつき 300 / 月
- Copilot Enterprise の場合は 1 ユーザーにつき 1000 / 月

参考: Organization または Enterprise の Premium リクエスト許容量の管理 - GitHub Enterprise Cloud Docs

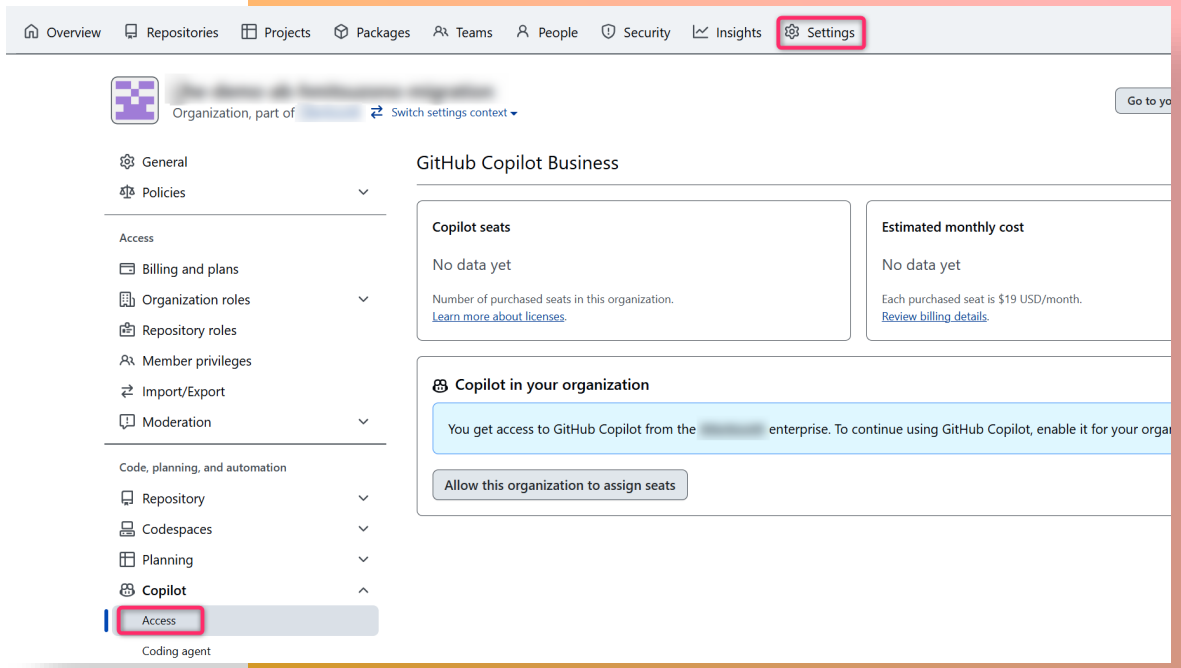
1.3 GitHub Copilot の有効化手順

Organization での Copilot の有効化

Organization Settings の Copilot > Access ページにアクセスします。

もしCopilot が有効化されていない場合は、右記表示になります。

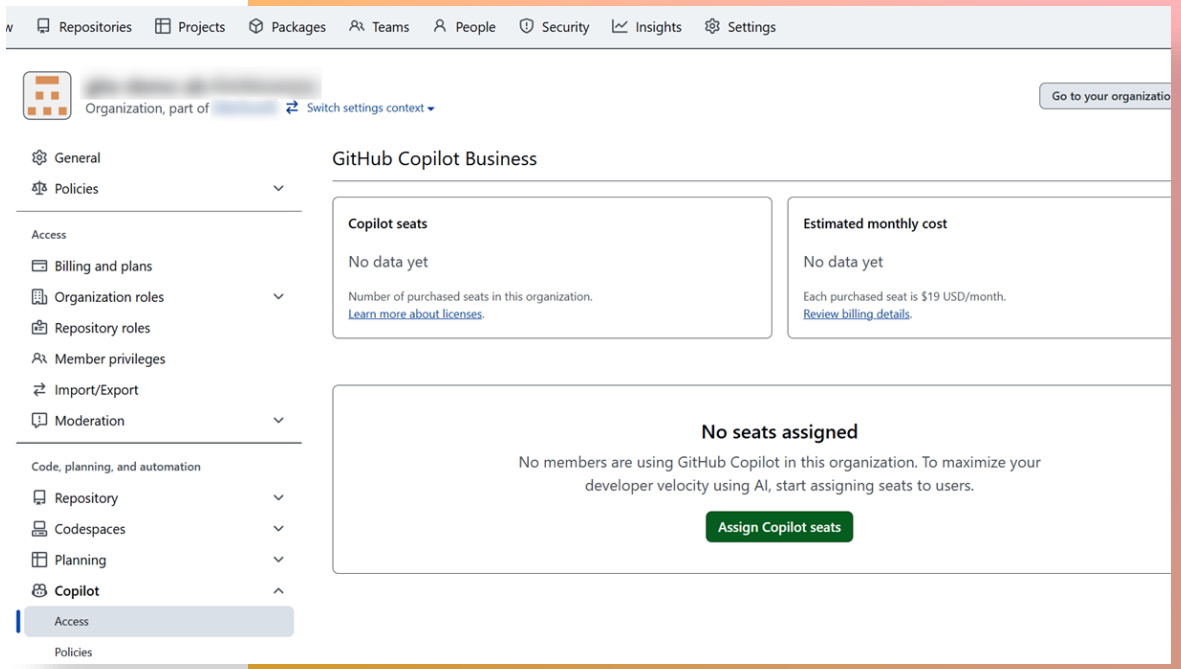
「Allow this organization to assign seats」から有効化することができます。



Organization での Copilot ライセンス割り当て

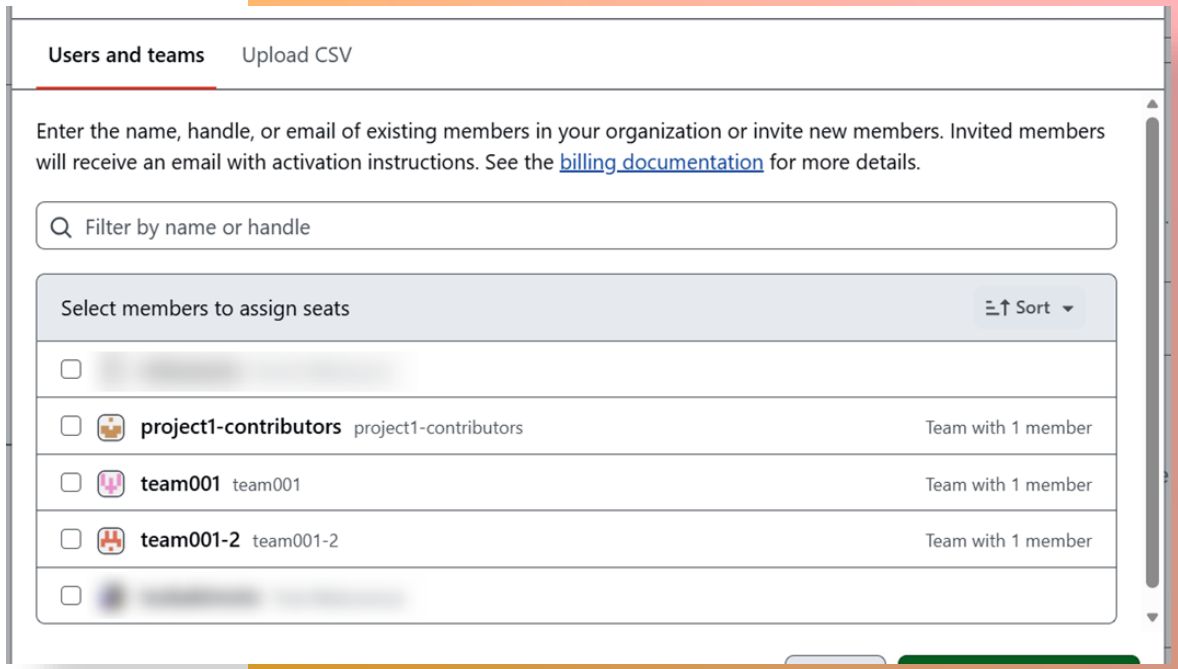
Organization で Copilot が有効化されていれば、右記表示になります。

「Assign Copilot seats」からライセンスを割り当てることができます。



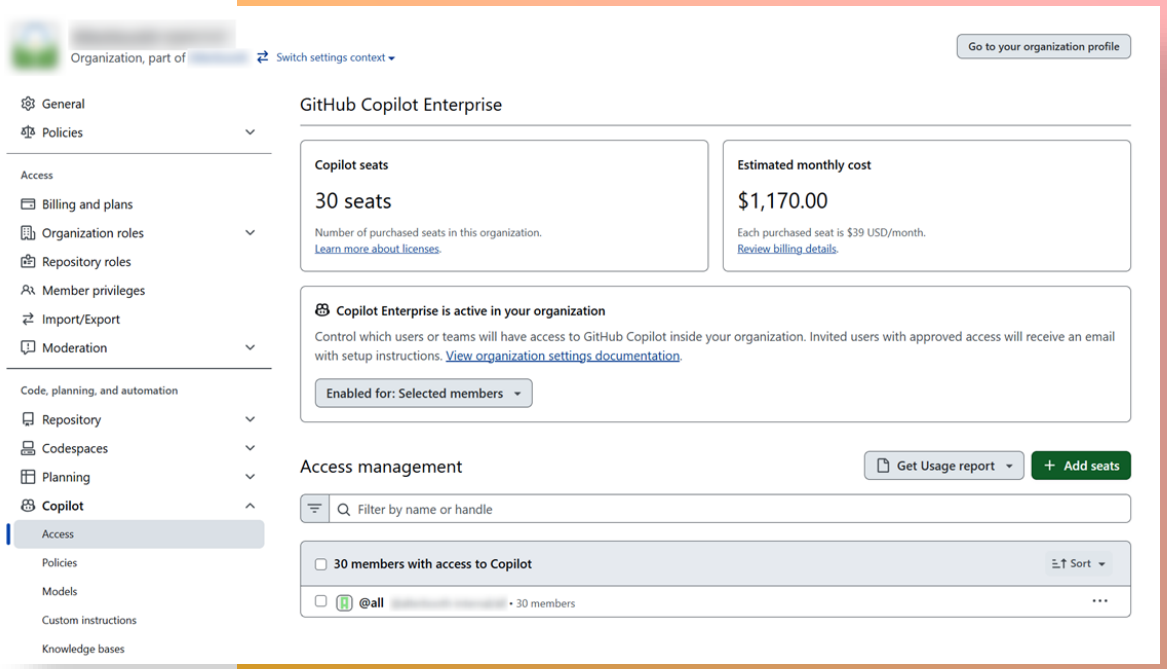
Assign Copilot seats 押下時ポップアップ表示

対象のユーザーまたはチームを選択してライセンスを割り当てます。



Organization での
割り当て対象ユーザー確認

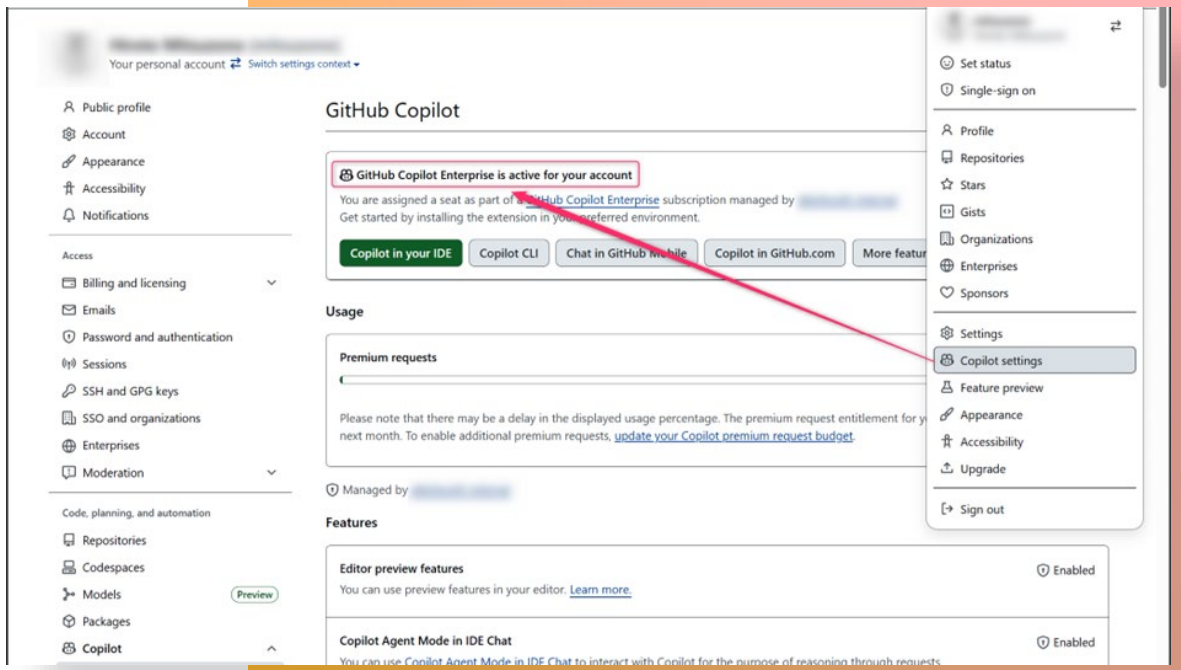
Access management の欄にライセンス適用対象ユーザーまたはそのユーザーが所属するチームが表示されていれば、該当ユーザーは GitHub Copilot を使用することができる状態です。



自身のライセンス適用状態を確認する方法

https://github.com/settings/copilot/features にアクセスします。

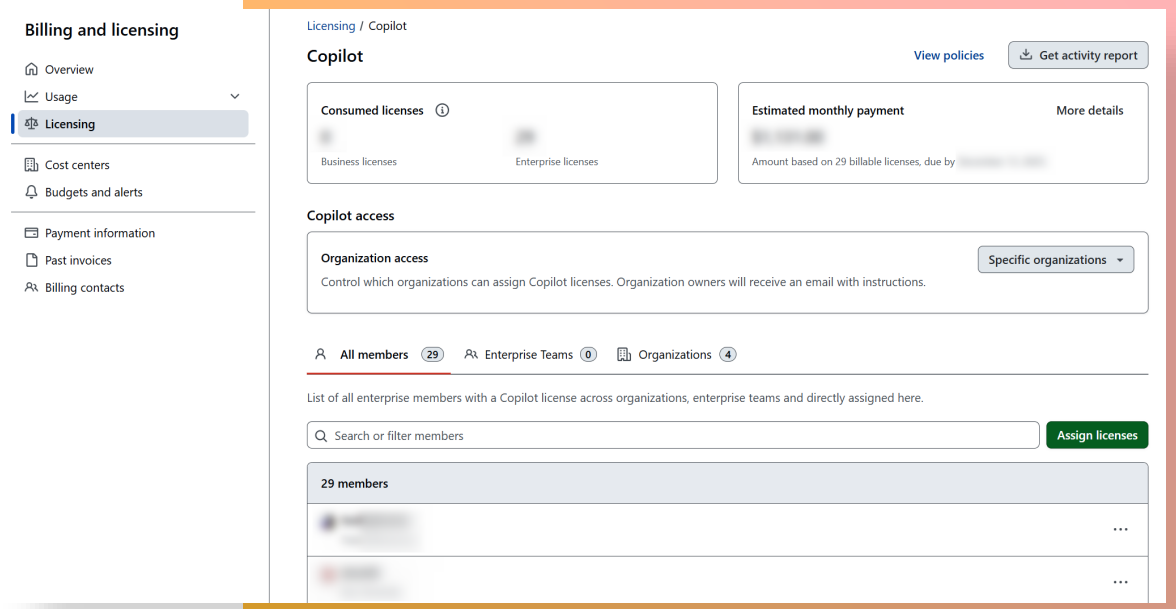
画面上部に GitHub Copilot (プラン名) is active for your account と表示されればライセンスが適用されており、GitHub Copilot が使用可能な状態です。



補足: Enterprise での Copilot ライセンスの割り当て

Organization メンバーに追加することなく、GitHub Copilot Business ライセンスを付与することもできます。

例えば、GitHub Enterprise Cloud のライセンスを付与せずに GitHub Copilot Business ライセンスのみを付与したい場合にこの割り当て方法を活用できます。



1.4 モデル別に適した用途の紹介

1.4 モデル別に適した用途の紹介（2025/12 現在）

GitHub Copilot は複数の AI モデルをサポートしています。
回答の関連性やレイテンシが異なるため、タスクに最適なモデルを選択しましょう。

モデル	適した用途
GPT-4.1	バランスの取れたパフォーマンスとマルチモーダルサポート
GPT-5.2, GPT-5.1, GPT-5	多段階的な問題解決とアーキテクチャレベルのコード分析
GPT-5 mini	軽量推論（インタラクティブなセッションやステップバイステップのコード分析）
GPT-5.1-Codex	高速で正確なコード補完と説明、特に Agent mode でのコーディング
Claude Sonnet 4	信頼性の高い推論、補完
Claude Sonnet 4.5	大規模コードベースに対するより深い推論
Claude Opus 4.1	Anthropic 最高の推論力（Ask mode のみ使用可能）
Claude Haiku 4.5	小規模タスクやプロトタイピング、軽量なコードの説明・変更
Gemini 2.5 Pro	高度なマルチモーダル推論

参考: AI モデルの比較 - GitHub ドキュメント

2. 一般的なプロンプトテクニック

一般的なプロンプトテクニック

本章では、プロンプトエンジニアリングの基礎について学びます。

プロンプトエンジニアリングとは、生成 AI の出力をより期待したものに近づけるために、プロンプトを設計・最適化する技術や手法のことです。

本章の例は GitHub Copilot のためのものですが、手法そのものは GitHub Copilot に限らず、他の生成 AI にも適用できる汎用的な内容となります。

2.1 出力値の実例を提示する

出力値の実例を提示する

- 曖昧さを減らします
  - 抽象的な説明だけでなく、具体的な例を示すことで認識のずれを防ぎます
- 考慮の抜け漏れを減らします
  - 例を与えることで、指示文章内に含まれない（または認識がずれる可能性のある）考慮事項をより正確に把握させます
- 前提条件を把握させます
  - ソースコードの場合、使用する言語やフレームワークの想定を例によって明確に伝えます

2.1 出力値の実例を提示する

使用場面例：

新規関数実装時、関連する関数の実装パターンを提示します。

期待される出力：

提示した実装パターンに従った減算関数が提案されます。

プロンプト例

以下のコード例に従って、subtract関数の実装例を示してください。

```
// 加算
int add(int a, int b) {
  int result = a + b;
  printf("[LOG] add(%d, %d) = %d¥n", a, b, result);
  return result;
}

// 乗算
int multiply(int a, int b) {
  int result = a * b;
  printf("[LOG] multiply(%d, %d) = %d¥n", a, b, result);
  return result;
}
```

【出力例】

```
// 減算
int subtract(int a, int b) {
  int result = a - b;
  printf("[LOG] subtract(%d, %d) = %d¥n", a, b, result);
  return result;
}
```

2.1 出力値の実例を提示する

NG 例使用時：

言語の指定がない場合、想定と異なる言語で実装される可能性があります。こちらの NG 例では特に指定が無く、JavaScript の出力結果が得られました。

想定した処理フローと異なる可能性や、処理が実装されない可能性があります。

NG例

subtract関数の実装例を示してください。

【出力例】

```
function subtract(a, b) {
  return a - b;
}
```

2.1 出力値の実例を提示する

使用場面例：

新規関数実装時、関数の使用例と戻り値の例を提示します。

期待される出力：

提示した関数の呼び出し方および返却値を実現する関数が提案されます。

プロンプト例

文字列を受け取り、その中の母音の数を返す関数を提示してください。
例：
findVowels("hello") returns 2
findVowels("sky") returns 0

【出力例】

```
function findVowels(str) {
    const vowels = ['a', 'e', 'i', 'o', 'u’];
    return str.toLowerCase().split('').filter(char =>
vowels.includes(char)).length;
}

// 使用例
console.log(findVowels("hello")); // 2
console.log(findVowels("sky"));   // 0
```

2.1 出力値の実例を提示する

NG 例使用時：

関数の使用例を提示しなかったため、こちらが質問をした日本語での母音の判定が行われる実装内容が提案されました。

また、関数名も指定が無かったため、自動で生成されました。

NG 例

文字列を受け取り、その中の母音の数を返す関数を提示してください。

【出力例】

```
def count_vowels(text):
    vowels = 'あいうえおアイウエオ'
    return sum(1 for c in text if c in vowels)
```

2.1 出力値の実例を提示する

使用場面例：

コミットメッセージ作成時に、適切なフォーマットの例を提示します。

期待される出力：

提示した例に従ったリファクタリングコミットメッセージが提案されます。

プロンプト例

GitHub Issue ID が 123 番のデータベース接続処理のリファクタリング用コミットを行おうと考えています。
当プロジェクトで使用できるコミットタグは add / update / delete / rename / docs です。
以下のコミット例に従い、適切なコミットメッセージを作成してください。

例1：[add] #1 ユーザー認証機能用の新規クラスを追加
例2：[update] #2 ログイン時のセッション切れ問題を修正
例3：[docs] #10 APIドキュメントを更新

【出力例】
[update] #123 データベース接続処理をリファクタリング

2.1 出力値の実例を提示する

NG 例使用時：

例や条件の提示がない場合、該当プロジェクトの取り決め（コミットタグやタスク ID 等の含め方）と異なるコミットメッセージが生成されます。

NG 例

データベース接続処理のリファクタリング用コミットメッセージを作成してください。

【出力例】

```
refactor: DB接続処理の可読性と保守性を向上

接続プール管理とエラーハンドリングを改善し、リソースリークを防ぐクリーンアップ処理を追加
```

2.2 具体的な指示を与え、推測を減らす

具体的な指示を与え、推測を減らす

- 推測に頼りすぎないよう注意します
  - 指示が不足すると、期待とは全く別の提案や回答がされる可能性が高くなります
  - GitHub Copilot による見落としやエッジケースの補足提案は有用ですが、その提案が必要ない場面では推測させないよう指示を工夫します
- 観点を明確にします
  - 例えばソースコードの修正提案を依頼する場合、コードやエラー文だけでなく、どのような観点で調査・修正を行ってほしいかを具体的に指示します

2.2 具体的な指示を与え、推測を減らす

使用場面例：

ソースコードを、null バリデーションを追加する形で修正指示します。

期待される出力：

null 参照エラーが起きうる箇所へのバリデーションエラーの追加が提案されます。

プロンプト例

以下のC#コードのnull参照エラーを修正し、エラーハンドリングを追加してください：

```
public string getUserLastName(string name)
{
  return name.Split(‘ ‘)[1];
}
```

【出力例】

```
// null参照エラーの修正とエラーハンドリング追加
Public string getUserLastName(string name)
{
  if (string.IsNullOrWhiteSpace(name))
  {
    throw new ArgumentException("nameがnullまたは空です。");
  }

  var parts = name.Split(' ');
  if (parts.Length < 2)
  {
    throw new ArgumentException("姓が見つかりません。");
  }

  return parts[1];
}
```

2.2 具体的な指示を与え、推測を減らす

NG 例使用時：

対象が指定されておらず、期待する箇所の修正が行われない可能性があります。

この例では、行うべき作業がエラー修正なのかリファクタリングなのかが明記されておらず、エラー修正を期待しているものの関数名に関するリファクタリングが実施されました。

NG 例

このコードを直してください：

```
public string getUserLastName(string name)
{
  return name.Split(' ')[1];
}
```

【出力例】

```
public string GetUserLastName(string name)
{
  return name.Split(' ')[1];
}
```

2.2 具体的な指示を与え、推測を減らす

使用場面例：

関数実装の指示をする際に、正常系の戻り値とエラーハンドリングに関する情報を提供します。

期待される出力：

指示した戻り値の形式かつ指示したエラーハンドリングを含む形で関数が実装されます。

プロンプト例

数値が素数かどうかを判定するJavaScript関数を書いてください。
関数は整数を受け取り、その整数が素数の場合はtrueを返してください。
入力が正の整数でない場合は例外処理として扱ってください。

【出力例】

```
function isPrime(n) {
  if (!Number.isInteger(n) || n <= 0) {
    throw new Error('入力は正の整数でなければなりません’);
  }
  if (n === 1) return false;
  if (n === 2) return true;
  if (n % 2 === 0) return false;
  for (let i = 3; i <= Math.sqrt(n); i += 2) {
    if (n % i === 0) return false;
  }
  return true;
}
```

2.2 具体的な指示を与え、推測を減らす

NG 例使用時：

エラーハンドリングを指示しなかったため true/false のレスポンス表現のみが実装されました。

また、言語情報を与えなかったため、Python のソースコード例が提供されました。

既存のコードを参照する場合は言語自体を間違えることはありませんが、ライブラリやバージョン情報に関しては明示するかパッケージ管理ファイル参照を指示します。

NG例

数値が素数かどうかを判定する関数を書いてください。

【出力例】

```
def is_prime(n):
    """nが素数ならTrue、そうでなければFalseを返す"""“
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(n ** 0.5) + 1, 2):
        if n % i == 0:
            return False
    return True
```

2.3 役割を与え、振る舞いを定める

役割を与え、振る舞いを定める

- GitHub Copilot の役割を明示することで、専門的な視点を得ます
  - フロントエンド/バックエンド、特定の言語など、専門分野に特化した回答を得ることができます
- 追加の観点を引き出します
  - 例えば、レビュー依頼時に指定した役割に応じた追加のレビュー項目が期待できます
  - 自分が持っていなかった観点や、指示文章に入れ忘れた項目をカバーできます
- 注視すべきポイントを絞ります
  - バックエンドよりもフロントエンドに特化したアドバイスが欲しい場合にフロントエンドのエキスパートとして回答させるなど、特定の領域に集中したアドバイスを得ることができます

2.3 役割を与え、振る舞いを定める

使用場面例：

SPA のページロードに関するパフォーマンス問題の改善ポイントに関して、バックエンドの視点に重きを置いて確認するよう指示します。

期待される出力：

ページロードの話なのでフロントエンドの話も含みつつ、バックエンド側の改善事項も実装例と共に詳細に出力されます。

プロンプト例

あなたはPythonに精通したバックエンド開発エキスパートです。
パフォーマンス問題に関して、Pythonの最新ベストプラクティスを活用し、実装例および測定方法とともに解決策を提示します。

React + FastAPIのSPAプロジェクトに関して、ページロードの改善点を教えてください。

【出力例】
ページロードのパフォーマンス改善には、フロントエンド（React）とバックエンド（FastAPI）の両方で工夫が必要です。以下に最新ベストプラクティスと実装例、測定方法をまとめます。

1. フロントエンド（React）
～省略～

2. バックエンド（FastAPI）

ベストプラクティス
- **非同期処理の活用** FastAPIはasync/await対応。I/O待ちを短縮。
- **レスポンスの圧縮** `GZipMiddleware`でAPIレスポンスを圧縮。
- **クエリ最適化** ORM（例：SQLAlchemy）の`selectinload`や`joinedload`でN+1問題を回避。
- **キャッシュ導入** Redis等でAPIレスポンスやDBクエリ結果をキャッシュ。

実装例

～省略～

2.3 役割を与え、振る舞いを定める

NG 例使用時：

どの視点で回答してほしいかが明記されておらず、一般的なページロードで確認される事項に関する回答が出力されます。

もしバックエンドに関して詳細な情報が欲しかった場合は、この指示では期待する回答が得られない可能性が高いです。

NG 例

React + FastAPIのSPAプロジェクトに関して、ページロードの改善点を教えてください。

【出力例】

React + FastAPIのSPA（Single Page Application）プロジェクトでページロード時間を改善する主な方法は以下の通りです。

フロントエンド（React側）

1. **コード分割（Code Splitting）**
2. **静的ファイルの最適化**
3. **キャッシュ戦略の導入**
4. **不要なレンダリングの削減**
5. **プリフェッチ・プリロード**

バックエンド（FastAPI側）

1. **APIレスポンスの高速化**
2. **静的ファイルの配信最適化**
3. **キャッシュヘッダーの設定**

2.3 役割を与え、振る舞いを定める

使用場面例：

データベースに対する特定期間のデータ取得クエリの提案を指示します。

期待される出力：

使用する DB、スキーマの情報をもとにしたクエリが提示されます。

プロンプト例

あなたはPostgreSQLの専門家です。
以下のスキーマ情報を考慮して、正確かつ効率的なSQLクエリを作成してください。

スキーマ:

```
- users(id, name, email, created_at)
- orders(id, user_id, amount, created_at)
```

要件:
2024年に10件以上注文したユーザーの名前と合計注文金額を取得してください。

【出力例】

```
SELECT u.name, SUM(o.amount) AS total_amount
FROM users u
JOIN orders o ON u.id = o.user_id
WHERE o.created_at >= '2024-01-01' AND o.created_at < '2025-01-01’
GROUP BY u.id, u.name
HAVING COUNT(o.id) >= 10;
```

2.4 出力フォーマットを明記する

出力フォーマットを明記する

- 出力結果の形式を意図したものに合わせます
- 出力フォーマットとして、拡張子はもちろん、持たせるパラメータやカラムとそれらの型まで与えることが望ましいです
  - 指定しなかった場合、パラメータの過不足や、形式違いの可能性が高くなります
  - ファイル形式によっては、インデントや区切り文字、命名規則を指定することも有効です

2.4 出力フォーマットを明記する

使用場面例：

JSON 形式で構造化された出力を指示します。

期待される出力：

指定された JSON 形式に従った構造化データが提示されます。

プロンプト例

以下のプロジェクト情報を指定したJSON形式で出力してください：

プロジェクト名：Web管理システム開発
期間：2024年4月～6月
予算：500万円

出力形式：

```
{
  "project_name": "",
  "duration": {"start": "", "end": ""},
  "budget": 0,
  "currency": ""
}
```

【出力例】

```
{
  "project_name": "Web管理システム開発",
  "duration": {"start": "2024-04", "end": "2024-06"},
  "budget": 5000000,
  "currency": "JPY“
}
```

2.4 出力フォーマットを明記する

NG 例使用時：

ファイル形式は指定されているもののフォーマットが指定されておらず、期待するフォーマットで出力されない可能性があります。

NG 例

以下のプロジェクト情報をJSON形式で出力してください：

プロジェクト名：Web管理システム開発
期間：2024年4月～6月
予算：500万円

【出力例】

```
{
  "プロジェクト名": "Web管理システム開発",
  "期間": "2024年4月～6月",
  "予算": "500万円"
}
```

2.4 出力フォーマットを明記する

使用場面例：

雑多に提示されたデータを比較する用途で、Markdown の表形式に変換するよう指示します。

期待される出力：

指定された Markdown 表形式での整理されたデータが提示されます。

プロンプト例

以下のデータを指定したMarkdown表形式で出力してください：

データ：
- サービスA：月額1000円、ユーザー数50万人、満足度4.2、コメント24、3プラン
- サービスB：月額1500円、ユーザー数30万人、満足度4.5、レビュー12
- サービスC：月額800円、ユーザー数80万人、満足度3.8、5プラン

出力形式：

```
| サービス名 | 月額料金 | ユーザー数 | 満足度 |
|------------|----------|------------|--------|
| サービスA  | 1,000円  | 50万人     | 4.2/5  |
```

【出力例】

```
| サービス名 | 月額料金 | ユーザー数 | 満足度 |
|------------|----------|------------|--------|
| サービスA  | 1,000円  | 50万人     | 4.2/5  |
| サービスB  | 1,500円  | 30万人     | 4.5/5  |
| サービスC  |   800円  | 80万人     | 3.8/5  |
```

2.4 出力フォーマットを明記する

NG 例使用時：

ファイル形式は指定されているもののフォーマットが指定されておらず、期待するフォーマットで出力されない可能性があります。

NG 例

以下のデータをMarkdown表形式で出力してください：

データ：
- サービスA：月額1000円、ユーザー数50万人、満足度4.2、コメント24、3プラン
- サービスB：月額1500円、ユーザー数30万人、満足度4.5、レビュー12
- サービスC：月額800円、ユーザー数80万人、満足度3.8、5プラン

【出力例】

```
| サービス名 | 月額料金 | ユーザー数 | 満足度 | コメント/レビュー | プラン数 |
|:---------|:------|:--------|:----|:----------|:--------|
| サービスA | 1000円 | 50万人 | 4.2 | コメント24 | 3プラン |
| サービスB | 1500円 | 30万人 | 4.5 | レビュー12 | - |
| サービスC | 800円 | 80万人 | 3.8 | - | 5プラン |
```

2.4 出力フォーマットを明記する

使用場面例：

エラー情報を構造化された XML 形式に変換するよう指示します。

期待される出力：

指定された XML 形式でのエラーレポートが提示されます。

プロンプト例

以下のエラーログを指定したXML形式でレポート出力してください。

エラー内容：

```
- 2025-09-07 14:30:15: SQLException in UserService.login()
- 2025-09-07 14:35:22: NullPointerException in
OrderController.process()
- 2025-09-07 14:40:10: TimeoutException in PaymentGateway.charge()
```

出力形式：

```
<error_report>
  <timestamp>2025-09-07T14:30:15</timestamp>
  <type>SQLException</type>
  <location>UserService.login()</location>
  <severity>high</severity>
</error_report>
```

【出力例】

```
<error_report>
  <timestamp>2025-09-07T14:30:15</timestamp>
  <type>SQLException</type>
  <location>UserService.login()</location>
  <severity>high</severity>
</error_report>
```

～ 省略 ～

2.4 出力フォーマットを明記する

NG 例使用時：

ファイル形式は指定されているもののフォーマットが指定されておらず、期待するフォーマットで出力されない可能性があります。

NG 例

以下のエラーログをXMLに変換してください。

エラー内容：

```
- 2025-09-07 14:30:15: SQLException in UserService.login()
- 2025-09-07 14:35:22: NullPointerException in
OrderController.process()
- 2025-09-07 14:40:10: TimeoutException in PaymentGateway.charge()
```

【出力例】

```
<errors>
  <error>
    <timestamp>2025-09-07 14:30:15</timestamp>
    <exception>SQLException</exception>
    <location>UserService.login()</location>
  </error>

～ 省略 ～
```

2.5 ステップバイステップで考えさせる

ステップバイステップで考えさせる

- 複雑または大規模なタスクを実行する場合、小さな複数のタスクに分割します
  - 特定のタスクの分割作業そのものを指示します
- 出力内容のエラーや齟齬を減らします
  - 指示内容が大きくなるほど、内容が一部抜け落ちてしまう可能性や全体の出力結果もエラーを含むものとなってしまう可能性が高くなります
- タスク項目およびタスク実行結果を再利用します
  - 会話履歴やファイル出力を通して、再利用できるコンテキスト化します
  - 分割した別のタスクの結果を、次のタスクの追加コンテキストとして使用することができます
  - タスクを分割し再利用可能なコンテキスト化することで、途中からの再開やリトライが可能になります

2.5 ステップバイステップで考えさせる

使用場面例：

単語検索パズルの段階的な実装手順を把握します。

期待される出力：

ステップ分けされた実装手順が提案されます。

プロンプト例

単語検索パズルを実装します。実装手順をステップバイステップで提示してください。

【出力例】
単語検索パズルの実装手順は以下の通りです。

1. 仕様の決定
- グリッドサイズ（例：10x10など）
- 隠す単語のリスト
- 単語の配置方向（縦・横・斜め・逆向きなど）

2. グリッドの初期化
- 空の2次元配列を作成

3. 単語の配置アルゴリズム
- 単語をランダムな位置・方向に配置
- 他の単語と重複しないようにする
- 配置できない場合は再試行

4. グリッドの埋め合わせ
- 空いているマスをランダムな文字で埋める

5. ユーザーインターフェースの設計

～省略～

2.5 ステップバイステップで考えさせる

NG 例使用時：

いきなり実装を指示すると、様々な認識齟齬が発生する可能性があります。

アプリケーションの規模が大きいほど、齟齬が発生しうる箇所も多くなります。

事前確認事項や、実装されたソースコードに対して続けて実施すべき手順も明示されません。

NG 例

単語検索パズルを実装してください。

【出力例】
単語検索パズル（ワードサーチ）を実装する簡単なPythonコード例を以下に示します。
このコードは、指定した単語リストをグリッドに隠し、ユーザーが単語を探せるようにします。

````
```python
import random
import string

def create_grid(size):
return [[random.choice(string.ascii_uppercase) for _ in range(size)]
for _ in range(size)]

def place_word(grid, word):
size = len(grid)
direction = random.choice(['H', 'V'])
if direction == 'H':
row = random.randint(0, size - 1)
col = random.randint(0, size - len(word))
for i, c in enumerate(word):
grid[row][col + i] = c

～省略～
````
````

2.5 ステップバイステップで考えさせる

使用場面例：

実装オプションを段階的に分析させ、各オプションのトレードオフを比較検討させます。

期待される出力：

実装パターンとそれぞれの違いが提示され、追加の分析や実装指示に流用することができます。

プロンプト例

```
Issue の要件を踏まえて、実装オプションを列挙し、各オプションのトレードオフを示してください。
[Issue の内容]
## Issue の説明
ユーザーは、最近閲覧したドキュメント、プロジェクト、ファイルに素早くアクセスできるよう、ダッシュボードに「最近の項目」セクションを追加してほしいと要望しています。
## 要件
* ユーザーが最近アクセスした項目を最大10件表示する
* 項目は最終アクセス時刻でソートする
* データベース負荷を最小限に抑える
* データはほぼリアルタイムで更新を反映する
* 大量のアイテムデータセットに対してもパフォーマンスを優先する

【出力例】
## 1. データベースクエリによる都度取得
### 概要
ダッシュボード表示時に「最終アクセス時刻でソートした最新10件」をDBクエリで取得。
### トレードオフ
- メリット: 実装がシンプル。常に最新データを取得可能。
- デメリット: アクセス頻度が高い場合、DB負荷が増大。大量データではクエリ最適化が必須。

## 2. キャッシュ（インメモリ or 分散キャッシュ）活用
### 概要
アクセス履歴の上位10件をRedis等にキャッシュし、ダッシュボード表示時はキャッシュから取得。
### トレードオフ
- メリット: 高速応答（200ms以内達成しやすい）、DB負荷軽減。
- デメリット: キャッシュ更新の整合性管理が必要。リアルタイム性はキャッシュ更新頻度に依存。
～省略～
```

2.6 複数回のやり取りで会話履歴を活用する

複数回のやり取りで会話履歴を活用する

- GitHub Copilot はチャット履歴をコンテキストに含みます
- 一回のやり取りで思った回答が得られなかった場合や汎用的な回答内容となった場合、追加の指示を与えることで深掘りします
  - 例えば先にプロジェクトの全容を把握し、一部の処理に関して続きの質問や指示をすることができます
- 前回の回答結果を次の指示に流用します
  - 例えばエラー対応の場合、エラーの調査を指示し、調査結果をもとに修正を指示することができます
- チャット履歴を含みたくない場合は新しい会話を開始します
  - 履歴は自動的に含まれるため、これまでの履歴がノイズとなり思ったような回答が得られない場合や同じ回答が繰り返しされるようになった場合は、履歴をクリアするために新しい会話を開始することができます
  - 関連性がなくなった、または目的の結果が得られなかったメッセージだけを削除することも可能です

2.6 複数回のやり取りで会話履歴を活用する

使用場面例（単語検索パズル続き）：

単語検索パズルの実装手順をステップバイステップで検討させた後、仕様を検討する部分に関して深掘りします。

期待される出力：

単語検索パズルの仕様に関する検討事項が出力されます。

プロンプト例

「1．仕様の決定」に関して、仕様を決めるうえでの検討事項を5つ、それぞれの回答案を3つずつ教えてください。

【出力例】

```
**1．グリッドサイズ**
- 8×8
- 10×10
- 15×15

**2．隠す単語の数**
- 5語
- 10語
- 15語

**3．単語の配置方向**
- 縦・横のみ
- 縦・横・斜め
- 全方向（逆向き含む）

～省略～
```

2.6 複数回のやり取りで会話履歴を活用する

使用場面例（単語検索パズル続き）：

単語検索パズルの仕様検討事項を回答に含める形で、仕様書の作成を指示します。

期待される出力：

単語検索パズルに関して、ひとつ前のやり取りで回答した仕様検討事項をもとにした仕様書が生成されます。

プロンプト例

仕様の検討事項に関して、下記の通り考えています。

1．グリッドサイズ
- 10×10
2．隠す単語の数
- 5語
3．単語の配置方向
- 縦・横のみ
4．使用する文字種
- 英字のみ
5．ユーザーインターフェース
- コンソール表示

上記をもとに、単語検索パズルアプリの仕様書をMarkdown形式で出力してください。

【出力例】

```
# 単語検索パズルアプリ 仕様書

本アプリは、10×10の英字グリッド上に隠された5つの英単語を探すコンソール表示型の単語検索パズルです。

## 1．仕様詳細
### 1.1 グリッド
- サイズ：10×10
- 各マスには英字（A～Z）が1文字ずつ表示される

～省略～
```

2.6 複数回のやり取りで会話履歴を活用する

使用場面例：

エラーの原因を段階的に調査し、根本的な解決策を導き出します。

期待される出力：

エラーの説明と解消案が提示され、その内容をもとに修正方針を検討します。
また、それら調査結果をもとにした実装を指示することができます。

プロンプト例

(1回目)

```
import sys
items = ["apple", "banana", "cherry"]
print(items[int(sys.argv[1])])
```

このコードを実行した際に発生した以下のエラーについて説明してください。

```
Traceback (most recent call last):
```

～ 省略 ～

```
IndexError: list index out of range
```

(2回目)

このエラーが発生しないように修正するにはどうしたらいいですか？

【出力例】

(1回目)

このエラーは、Pythonスクリプトを実行した際にコマンドライン引数が不足しているため発生しています。

～ 省略 ～

(2回目)

このエラーを防ぐには、コマンドライン引数が指定されているかどうかをチェックし、足りない場合はエラーメッセージを表示して終了する処理を追加します。また、引数が整数かどうか、範囲内かどうかも確認するとより安全です。

～ 省略 ～

2.6 複数回のやり取りで会話履歴を活用する

NG 例使用時：

最初から詳細な修正方法を求めて、エラーの根本原因を理解せずに表面的な対処になります。

一度の質問で全ての情報を得ようとして、的確な回答が得られないまま実装に進んでしまいます。

NG 例

```
import sys
items = ["apple", "banana", "cherry"]
print(items[int(sys.argv[1])])
```

このコードを実行した際に発生した以下のエラーについて詳細に説明して修正方法と予防策とテストコードも教えてください。

```
Traceback (most recent call last):
File "C:¥Users¥Dummy¥Documents¥python-demo¥test.py", line 3, in <module>
print(items[int(sys.argv[1])])
~~~~~^^^^^^^^^^^^^^^^^^^
IndexError: list index out of range
```

【出力例】

エラー詳細

このエラーは `IndexError: list index out of range` です。
原因は、`sys.argv[1]` に指定された値が `items` リストのインデックス範囲外だった場合に発生します。

具体的な流れ

1. コマンドライン引数でインデックスを指定します（例: `python test.py 2

～ 省略 ～

2.7 GitHub Copilot 自身にプロンプトを成・改善させる

GitHub Copilot 自身にプロンプトを生成・改善させる

- プロンプトを GitHub Copilot に生成・改善させます
  - プロンプトの用途を伝えることで、抑えるべきポイントを含んだプロンプトを生成させます
  - 既存のプロンプトに対して、用途をベースとした指示文章の過不足レビューをさせることができます
- 双方向フィードバックを行います
  - 既存のソースコードやコンテキストをもとに、プロンプトへフィードバックします
  - 例えばソースコードの場合、プロンプトと成果物の双方向フィードバックにより互いのずれを減らすことができ、今後のために意図した提案内容となる可能性をより高めることができます

2.7 GitHub Copilot 自身にプロンプトを生成・改善させる

使用場面例：

目的に応じた効果的なプロンプトを自動生成します。

期待される出力：

提示したタスクを処理するために最適化されたプロンプトが提案されます。

プロンプト例

以下のタスクを実行するための最適なプロンプトを生成してください：

タスク：コードレビューをAIに依頼したい
対象：JavaScript/React コード
重視する観点：パフォーマンス、可読性、保守性、セキュリティ

生成するプロンプトには以下を含めてください：
- 適切な役割設定
- 具体的なレビュー観点
- 出力フォーマットの指定

改善提案の形式

【出力例】
あなたは経験豊富なシニアフロントエンド開発者として、JavaScript/Reactコードのレビューを行ってください。

##【レビュー対象】
提供されたJavaScript/Reactコードを以下の観点から詳細に分析してください：

🔍 **重点レビュー観点**
1. **パフォーマンス**
- 不要な再レンダリングの有無
- メモ化（useMemo、useCallback）の適切性
- バンドルサイズへの影響
- 非同期処理の最適化
2. **可読性**
～省略～

2.7 GitHub Copilot 自身にプロンプトを生成・改善させる

使用場面例：

用途の要点をもとに、copilot-instructions.md を生成します。

期待される出力：

要求した条件を満たすためのルールが定義された copilot-instructions.md ファイル内容が提案されます。

プロンプト例

以下の内容に沿って `.github/copilot-instructions.md` を生成してください。

- `.github/copilot-instructions.md` は、GitHub Copilotがプロジェクトのルールに沿ってコード実装を行うために、基底のコンテキストとして使用されるファイルです
C#(.NET 10)のコーディング規約に関する内容を中心に、コーディングルールとそのルールに従う旨を記載してください
実装後に確認すべきチェックリストとそのチェックを行う旨を記載してください

【出力例】

このプロジェクトでは、以下のコーディング規約に従ってコードを実装してください。

```
## C# コーディング規約

### 命名規約

- **クラス名**: PascalCase（例: `UserService`, `OrderManager`）
- **メソッド名**: PascalCase（例: `GetUserById`, `ProcessOrder`）
～省略～

### コーディングスタイル
- インデントはスペース4文字を使用
- 波括弧は新しい行に配置（Allman style）
～省略～
```

2.7 GitHub Copilot 自身にプロンプトを生成・改善させる

使用場面例：

既存のプロンプトを改善します。ここまでの2章全体の内容を踏まえて役割を明記し、改善ポイントもフォーマットを指定して提示させます。

期待される出力：

提示したプロンプトの改善点と、問題点を解決したより高品質なプロンプトが提案されます。

プロンプト例

```
あなたはプロンプト批評の専門家です。
ユーザーが作成したLLMプロンプトを精査し、以下の指示に従って欠点を明らかにしてください。

LLMが従うべきプロンプトを確認し、以下の問題点を特定してください：
- 曖昧さ：文言が複数の解釈をされる可能性がありますか？
- 定義不足：LLMが誤解する可能性のある、定義されていないクラスラベル、用語、または概念はありますか？
- 矛盾、欠落、または曖昧な指示：指示が不完全または矛盾していますか？
- 明示されていない前提：プロンプトは、モデルが明示的に述べられていない機能を実行できると想定していますか？

以下の種類の問題点を**記載しないでください**：
- 新しい指示、ツール呼び出し、または外部情報を作成すること。不足しているツールのうち、追加する必要があるものがわからない場合。
- 確信が持てない問題。

出力形式：
"""
# 問題点
- 番号付きリスト。短い引用スニペットを含めてください。
# 改善点
- 番号付きリスト。変更する修正行とその変更方法を記載してください。
# 修正済みプロンプト
- 元のプロンプトに最小限の編集を加え、すべての改善点を精密に適用した修正済みプロンプト
"""
```

3. GitHub Copilot 特有の機能を用いたプロンプトテクニック

GitHub Copilot 特有の機能を用いたプロンプトテクニック

GitHub Copilot には「スラッシュコマンド(/)」「チャット変数(#)」「チャット参加者(@)」が用意されています。

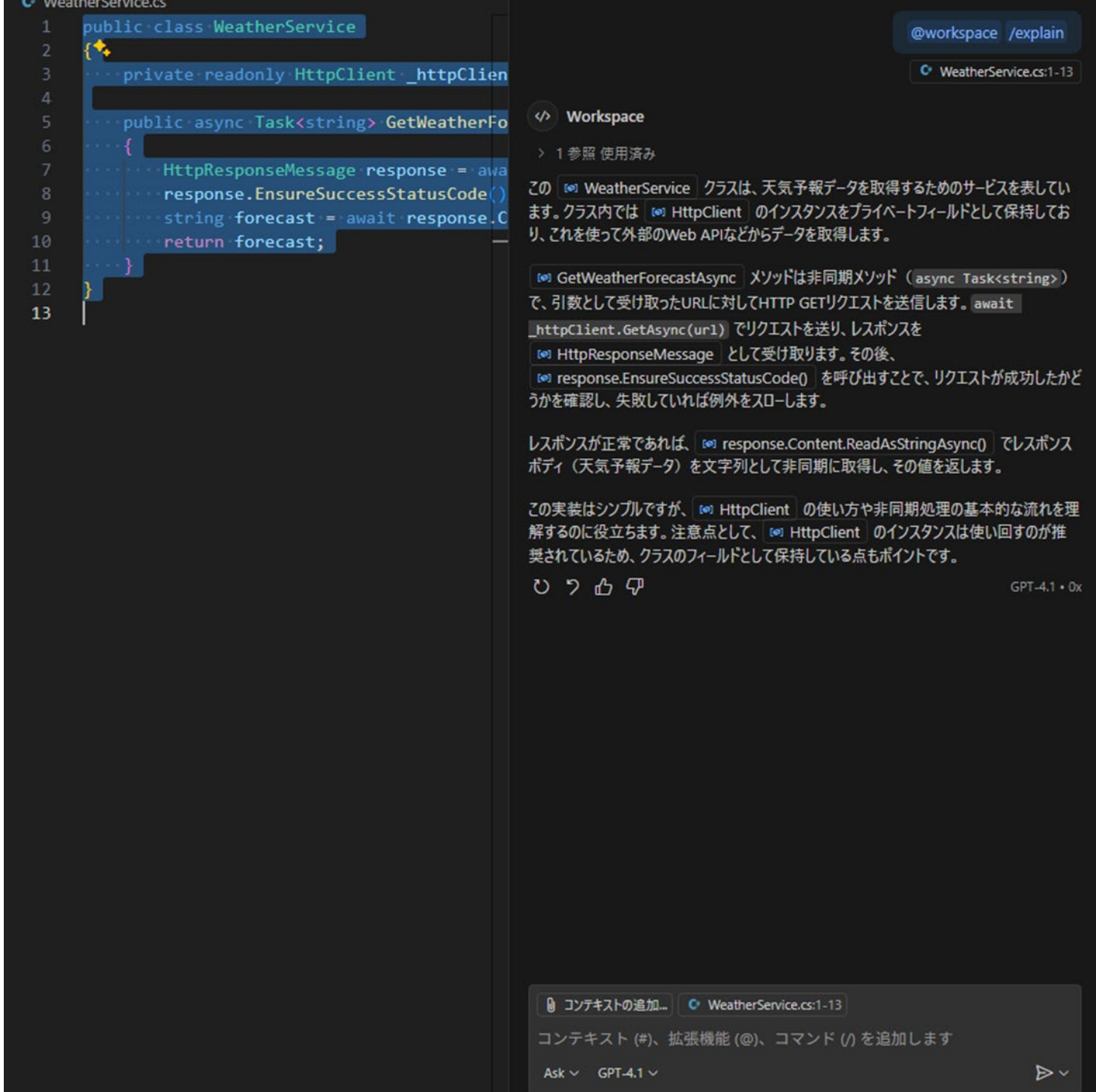
- スラッシュコマンド(/)：標準で用意されているコマンドを使用して、一般的によく使用されるプロンプト（説明、修正、テスト実装など）の入力を削減します
- チャット変数(#)：明示的に特定のコンテキスト（ファイル・コード行・シンボルなど）をプロンプトに含めます
- チャット参加者(@)：ワークスペース・ターミナル・Visual Studio Code・GitHubなど特定の専門知識に関する支援を受けたい際に使用します

本章では、これらの使用に関して有用なプロンプト例を示します。

3.1 既存のソースコードやプロジェクトを説明させる

/explain

開いているファイルまたは選択されたコード行に対して、コードが処理する内容を説明します。



3.1 /explain

使用場面例：

コードブロックの説明を要求します。

期待される出力：

コードブロックの説明が提示されます。

プロンプト例

```
/explain
public class WeatherService
{
  private readonly HttpClient _httpClient = new HttpClient();

  public async Task<string> GetWeatherForecastAsync(string url)
  {
    HttpResponseMessage response = await _httpClient.GetAsync(url);
    response.EnsureSuccessStatusCode();
    string forecast = await response.Content.ReadAsStringAsync();
    return forecast;
  }
}
```

【出力例】

このコードは、天気予報を取得するための WeatherService クラスです。
HttpClient を使って、指定されたURLから天気予報データを非同期で取得します。
GetWeatherForecastAsync メソッドは、URLを受け取り、HTTP GETリクエストを送信します。
レスポンスが正常なら、内容（天気予報）を文字列として返します。

3.1 /explain

使用場面例：

特定のディレクトリ内の特定の処理実装内容に対する説明を要求します。

期待される出力：

platform ディレクトリ内で実装されているロギング処理に関する説明が出力されます。

プロンプト例

```
@workspace /explain #file:platform ロギング処理に関して
```

【出力例（microsoft/vscodeで使用） 】

VS Codeのplatformレイヤーにおけるロギング処理について説明します。

VS Codeのロギングシステムは `log` ディレクトリで管理され、依存性注入パターンとサービス指向アーキテクチャを採用しています。

環境別実装戦略

Browser環境 (`browser/`)
- WebAPIベースのログ出力
- IndexedDBへの永続化
- パフォーマンス測定機能

Electron Main Process (`electron-main/`)
- ファイルシステムへの直接書き込み
- プロセス間通信での集約
- システムログとの連携

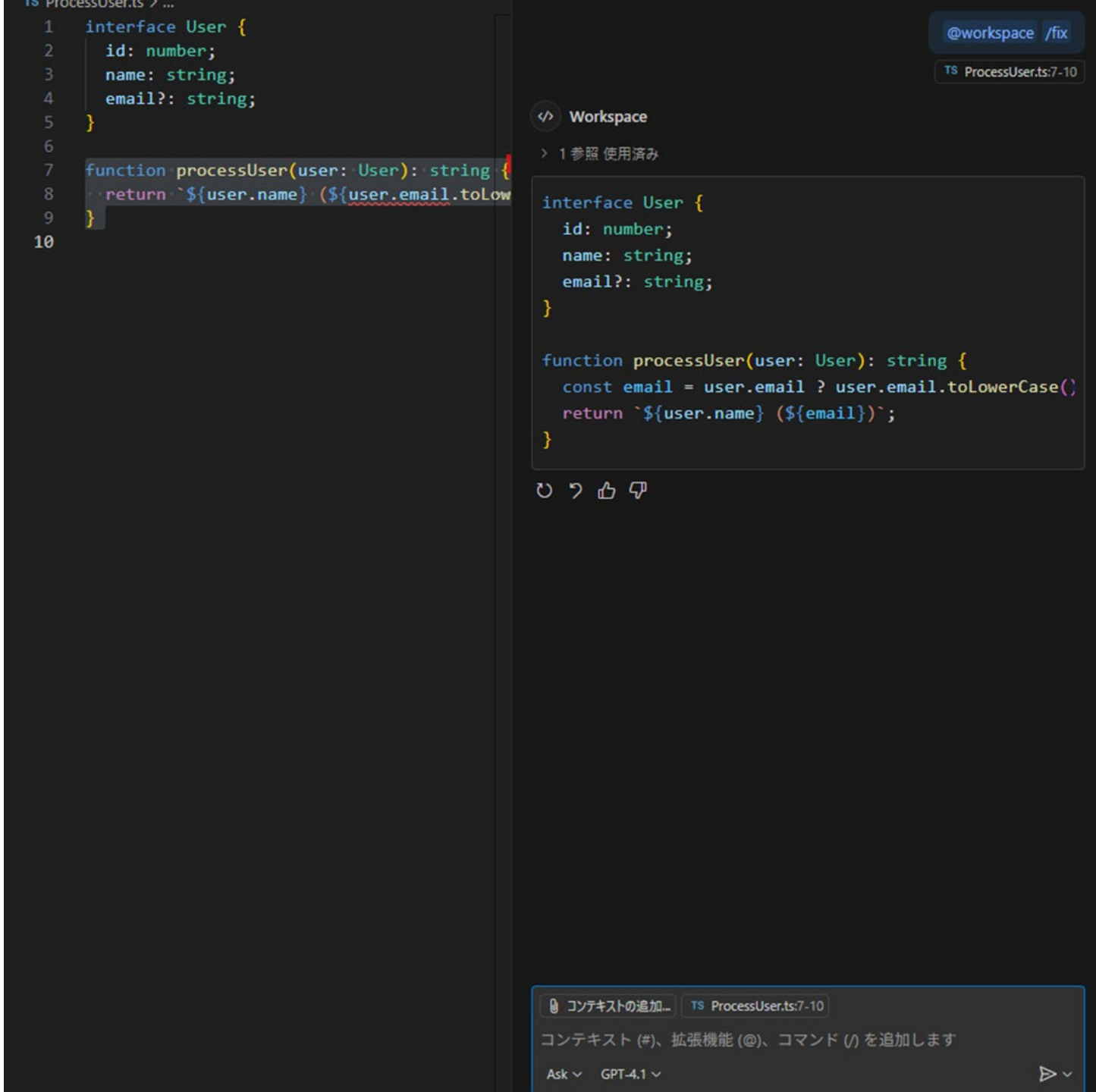
Node.js環境 (`node/`)
- ストリームベースのファイル出力
- ログローテーション機能
- syslogプロトコル対応

～省略～

3.2 エラー修正やリファクタリングを指示する

/fix

開いているファイルまたは選択されたコード行に対して、エラー修正またはリファクタリングを実施します。



3.2 /fix

使用場面例：

既存コードのエラー・改善余地を簡易的に把握・修正提案させます。

期待される出力：

エラー原因の説明と修正されたコードが提示されます。

プロンプト例

```
/fix
interface User {
  id: number;
  name: string;
  email?: string;
}

function processUser(user: User): string {
  return `${user.name} (${user.email.toLowerCase()})`;
}
```

【出力例】

コードの問題を特定しました。
emailプロパティはオプショナル（email?: string）なので、undefinedの可能性があります。
undefined.toLowerCase()を呼び出すとランタイムエラーが発生します。

以下が修正版です：

```
// ...existing code...
function processUser(user: User): string {
return `${user.name} (${user.email?.toLowerCase() || 'No email'})`;
}
// ...existing code...
```

3.2 /fix

使用場面例：

コーディング規約をもとにしたリファクタリングを行います。

期待される出力：

命名規則等をもとにした修正が提案されます。

プロンプト例

/fix 次のコードをC#のコーディング規約に沿って修正してください。

```
public class user
{
  public int id { get; set; }
  public string name { get; set; }
  public string getUserLastName()
  {
    return name.Split(' ')[1];
  }
}
```

【出力例】

以下のようにC#のコーディング規約に沿って修正します：

```
public class User
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string GetUserLastName()
  {
    return Name.Split(' ')[1];
  }
}
```

修正点：

クラス名: user → User (PascalCase)

プロパティ名: id → Id, name → Name (PascalCase)

メソッド名: getUserLastName() → GetUserLastName() (PascalCase)

3.2 /fix

使用場面例：

null チェックを追加するリファクタリングを行います。

期待される出力：

null チェックを追加した修正内容が提案されます。

プロンプト例

```
/fix nullチェックの追加
public string GetUserLastName()
{
  return Name.Split(' ')[1];
}
```

【出力例】

このコードにnullチェックを追加して、Nameがnullの場合や配列の要素が不足している場合の例外を防ぎます。

```
public string GetUserLastName()
{
  if (string.IsNullOrEmpty(Name))
    return string.Empty;

  var nameParts = Name.Split(' ');
  if (nameParts.Length < 2)
    return string.Empty;

  return nameParts[1];
}
```

この修正により以下の問題が解決されます：

Nameがnullの場合のNullReferenceExceptionを防止

Nameが空文字列の場合の処理

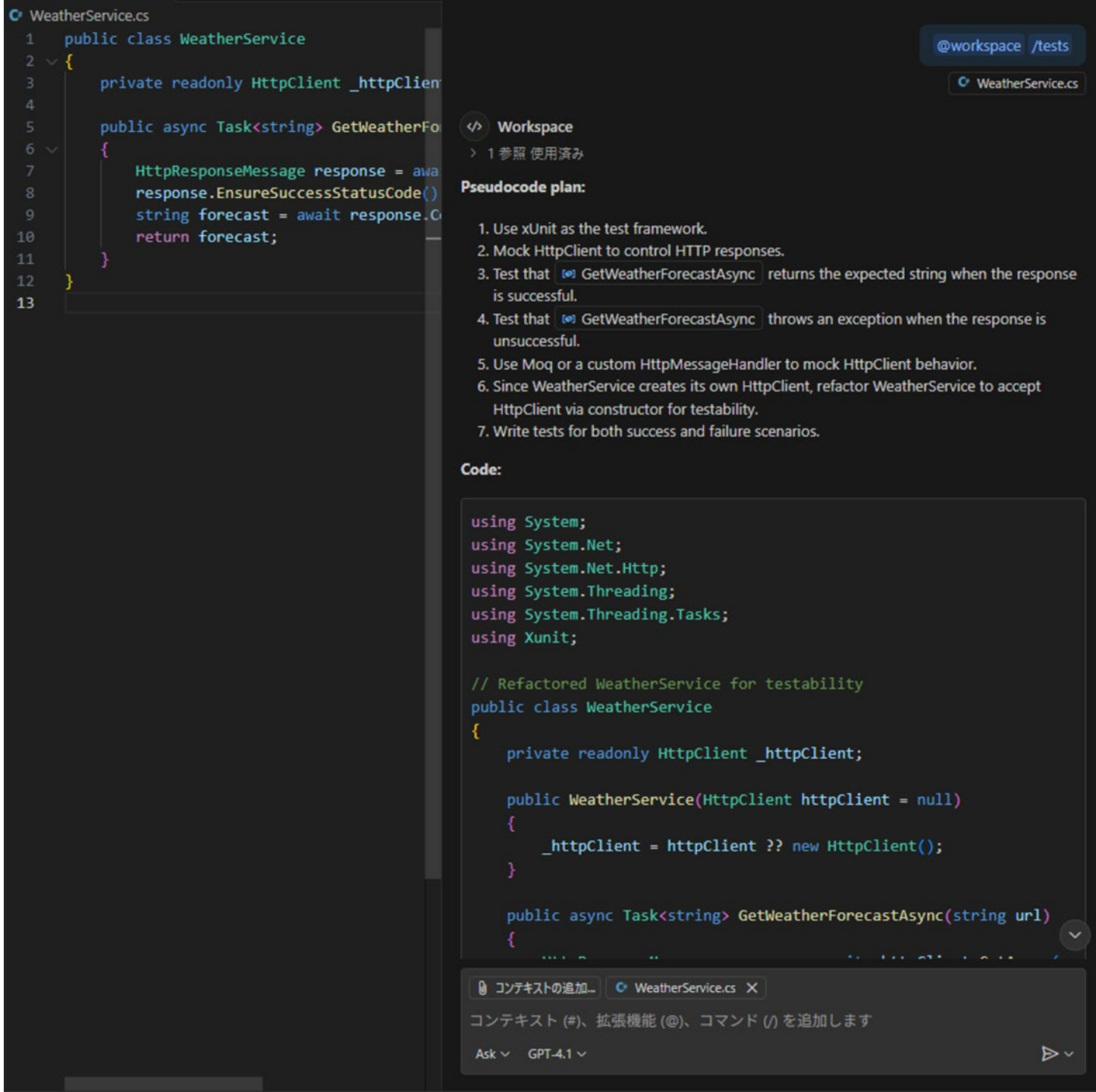
名前にスペースが含まれていない場合（姓のみ、名のみ）のIndexOutOfRangeExceptionを防止

3.3 単体テストを実装させる

/tests

開いているファイルまたは選択されたコード行に対して、単体テストを実装します。

※テストライブラリを予めセットアップしておくことで、GitHub Copilot は使用するテストライブラリを特定できます。



3.3 /tests

使用場面例：

既存の関数に単体テストを追加します。

期待される出力：

関数の単体テストが生成されます。

プロンプト例

```
/tests
public class WeatherService
{
  private readonly HttpClient _httpClient = new HttpClient();
  public async Task<string> GetWeatherForecastAsync(string url)
  {
    HttpResponseMessage response = await _httpClient.GetAsync(url);
    response.EnsureSuccessStatusCode();
    string forecast = await response.Content.ReadAsStringAsync();
    return forecast;
  }
}
```

【出力例】

```
public class WeatherServiceTests
{
  [Fact]
  public async Task
GetWeatherForecastAsync_ValidUrl_ReturnsWeatherData()
  {
    // Arrange
    var expectedResponse = "Sunny, 25°C";
    var mockHttpMessageHandler = new Mock<HttpMessageHandler>();
～省略～
```

3.3 /tests

使用場面例：

選択した関数のみ単体テストを実装します。

期待される出力：

選択した関数に対する単体テストのみが実装されます。

この方法を応用して、特定のテストケースのみの実装を指示することも可能です。

プロンプト例

```
/tests Addのみ
public static class Calculator
{
  public static int Add(int a, int b) => a + b;
  public static int Subtract(int a, int b) => a - b;
  public static int Multiply(int a, int b) => a * b;
}
```

【出力例】

```
public class CalculatorTests
{
  [Fact]
  public void Add_TwoPositiveNumbers_ReturnsSum()
  {
    // Arrange
    int a = 5;
    int b = 3;
    int expected = 8;
    // Act
    int result = Calculator.Add(a, b);
    // Assert
    Assert.Equal(expected, result);
  }

  [Fact]
  public void Add_TwoNegativeNumbers_ReturnsSum()
～省略～
```

3.3 /tests

使用場面例：

外部サービスに依存する処理に関して、モックオブジェクトを使用したテストを実装します。

期待される出力：

選択した関数に対する単体テストのみが実装されます。

この方法を応用して、特定のテストケースのみの実装を指示することも可能です。

プロンプト例

/tests サービスが正しく呼び出されることを確認するためのユニットテストを作成します。サービスオブジェクトをモックします。

```
import service from './service';
export async function load({ params }) {
  const runner = await service.getRunner(params.id);
  return { runner };
}
```

【出力例】

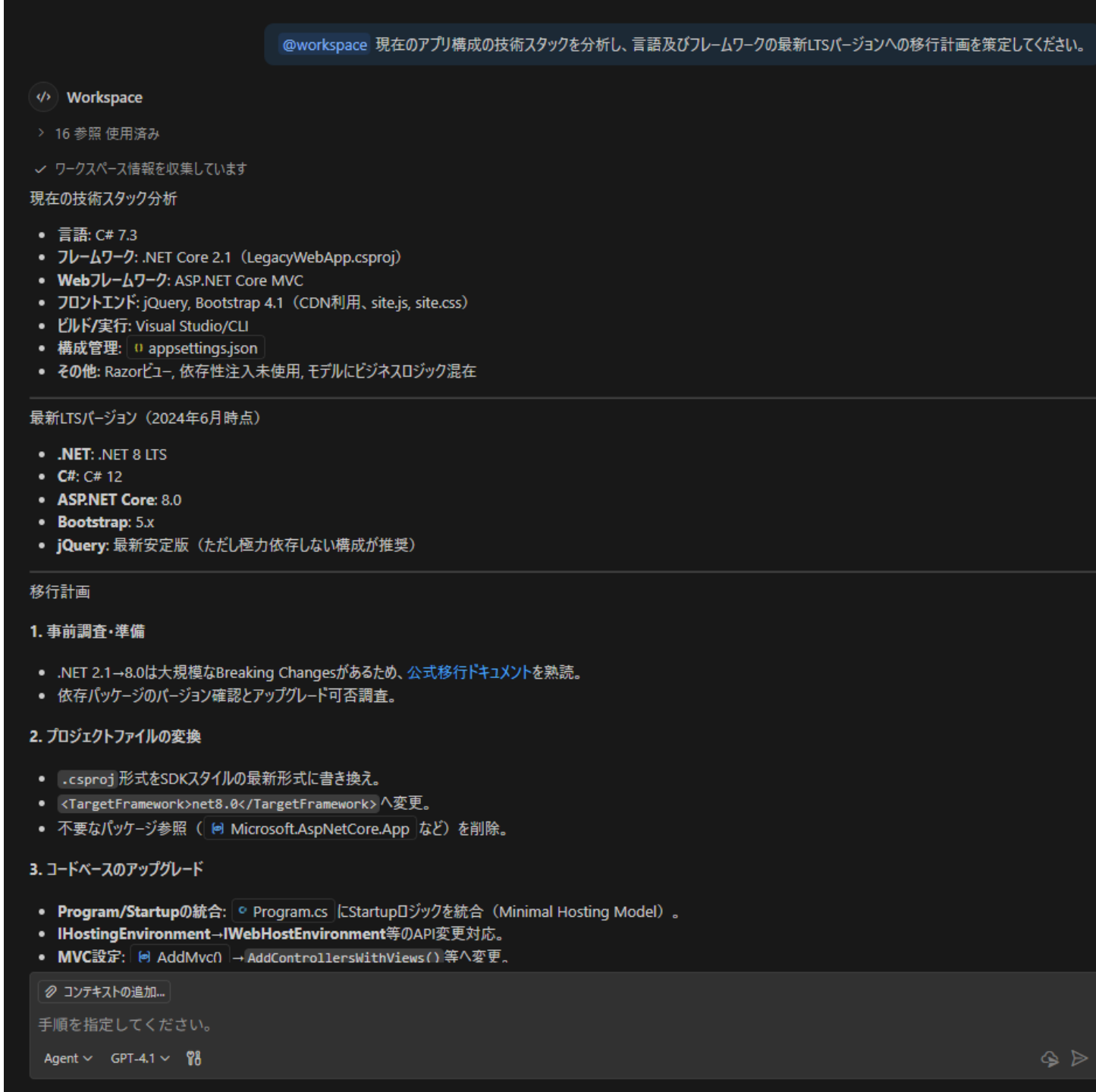
```
import { describe, it, expect, vi, beforeEach } from 'vitest';
import { load } from './+page.server.ts';
// Mock the entire service module
vi.mock('./service', () => ({
  default: { getRunner: vi.fn() }
}));
import service from './service';
describe('load function', () => {
  const mockRunner = { id: '123', name: 'Test Runner’ };
  beforeEach(() => {
    vi.resetAllMocks();
    (service.getRunner as any).mockResolvedValue(mockRunner);
  });
  it('should fetch runner with correct id', async () => {
    const params = { id: '123' };
    const result = await load({ params });
    expect(service.getRunner).toHaveBeenCalledWith(‘123’);
```

～省略～

3.4 プロジェクト全体をコンテキストとして扱う

@workspace

ワークスペース内のコードに関するコンテキストを含めます。
プロジェクトの構造、コードのさまざまな部分の相互作用、またはプロジェクト内の設計パターンを考慮する場合に使用します。



3.4 @workspace

使用場面例：

複雑なマイクロサービス環境で新サービス追加の影響範囲を確認します。

期待される出力：

プロジェクト全体を考慮した詳細な影響分析と実装計画が出力されます。

プロンプト例

@workspace プロジェクトアーキテクチャを分析して、新しいマイクロサービス追加の影響を評価してください。

追加予定のサービス：通知サービス（notification-service）

評価項目：
1. 既存サービスとの依存関係
2. データベース設計への影響
3. API Gateway設定の変更点
4. Docker/Kubernetes設定の更新
5. CI/CDパイプラインの修正要否
6. セキュリティ・認証の考慮事項

【出力例（dotnet-architecture/eShopOnAzureで使用） **】**

1. 既存サービスとの依存関係
- 通知サービスは、注文・バスケット・ユーザー管理など他サービスからイベントを受け取る必要があります。
- 既存サービス（Ordering, Basket, Identity等）からイベントバス（RabbitMQ等）経由で通知サービスへイベント連携が必要です。
- 直接的な依存は少ないですが、イベント契約（IntegrationEvent）の設計・変更が発生します。

2. データベース設計への影響
- 通知サービス用の独立したDB（例：通知履歴、ユーザー通知設定等）が必要です。
- 既存サービスのDBには直接影響しませんが、ユーザーID等の参照設計が必要です。

～省略～

3.4 @workspace

使用場面例：

レガシーシステムの全面的なモダナイゼーションを計画します。

期待される出力：

プロジェクト全体の関係性や構成を把握したうえでのシステム移行計画が提案されます。

プロンプト例

@workspace バックエンド構成の技術スタックを分析し、言語及びフレームワークの最新LTSバージョンへの移行計画を策定してください。

【出力例（dotnet-architecture/eShopOnAzureの例）】

```
## バックエンド技術スタック分析

### 1. プラットフォーム・フレームワーク
- **.NET Core 1.1** (Docker fileで確認: `microsoft/aspnetcore:1.1`)
- **ASP.NET Core MVC 1.1**
- **Entity Framework Core**
- **IdentityServer4**

### 2. データベース
- **SQL Server** (EF Core マイグレーション使用)
～省略～

## 最新LTSバージョンへの移行計画

### フェーズ1: .NET 10 LTS移行（優先度：高）

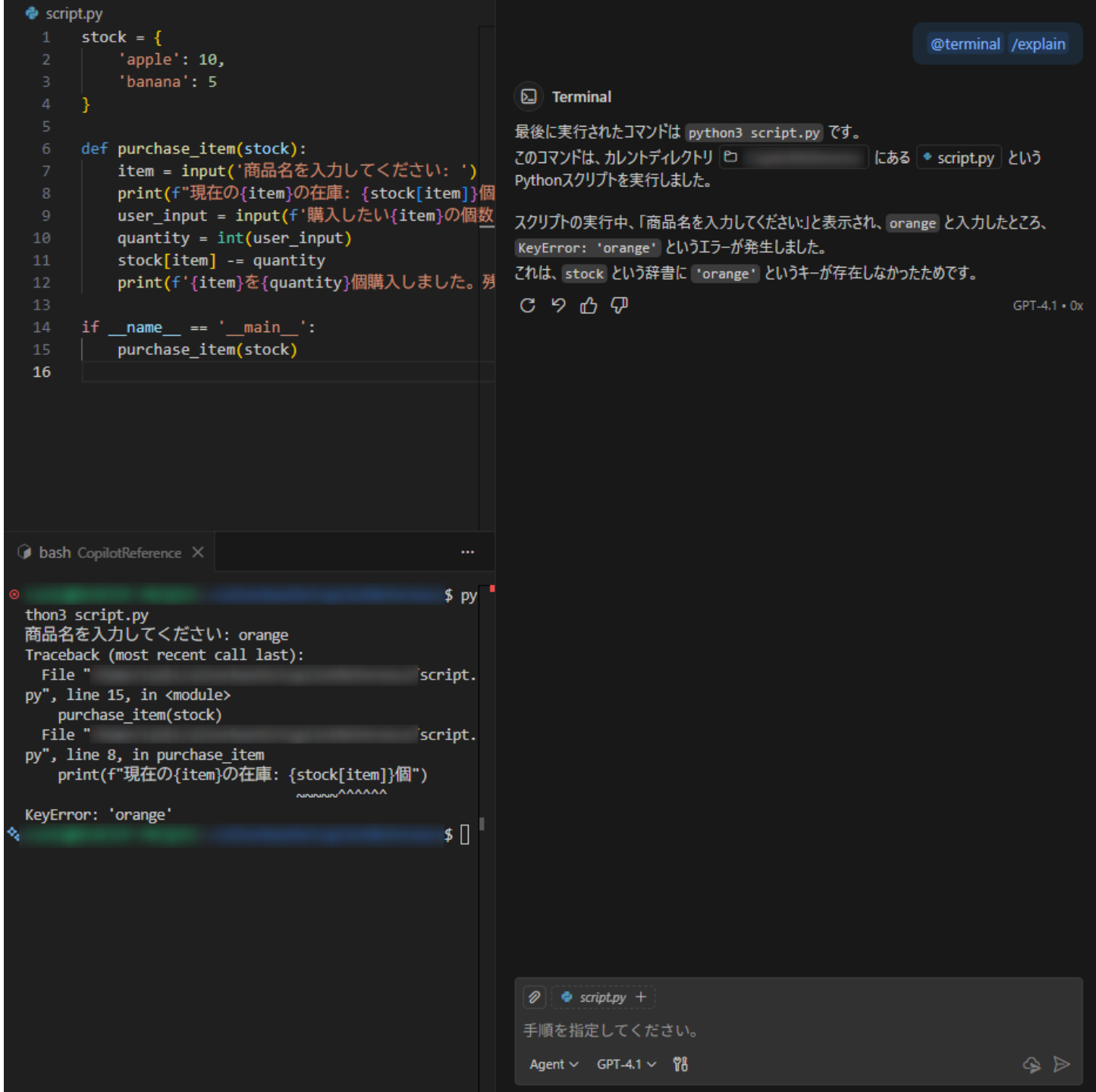
**現状**: .NET Core 1.1 → **目標**: .NET 10 LTS

**移行対象サービス**:
- `Catalog.API`
- `Basket.API`
- `Ordering.API`
～省略～
```

3.5 ターミナルの実行結果をコンテキストとして扱う

@terminal

Visual Studio Code ターミナル シェルとその内容に関するコンテキストを含めます。
ターミナルコマンドの生成またはデバッグに関するヘルプが必要な場合に使用します。



3.5 @terminal

使用場面例：

API サーバーのパフォーマンス問題を調査するために使用する複雑なコマンドを確認します。

期待される出力：

ログ解析のためのコマンドが提案されます。

プロンプト例

@terminal api.logから処理時間が5秒以上かかったリクエストを抽出するコマンドを教えてください。
ログ形式は次の通りです。

```
2025-10-02 18:11:48 [INFO] POST /api/analytics response_time:0.9s
2025-10-02 18:34:20 [INFO] PATCH /api/auth/logout response_time:2.3s
```

【出力例】

```
grep -E "response_time:[5-9]¥.[0-9]+s|response_time:[1-9][0-9]+¥.[0-9]+s" api.log
```

3.5 @terminal

使用場面例：

実行時エラーが発生した際に、ログを解析し原因を特定します。

期待される出力：

エラーログの解析結果と原因の説明が出力されます。

プロンプト例

【CLI例】

```
$ python3 calc.py
Traceback (most recent call last):
File "/sample/calc.py", line 4, in <module>
print(division(5, 0))
～ 省略 ～
```

【プロンプト例】

```
@terminal /explain
```

【出力例】

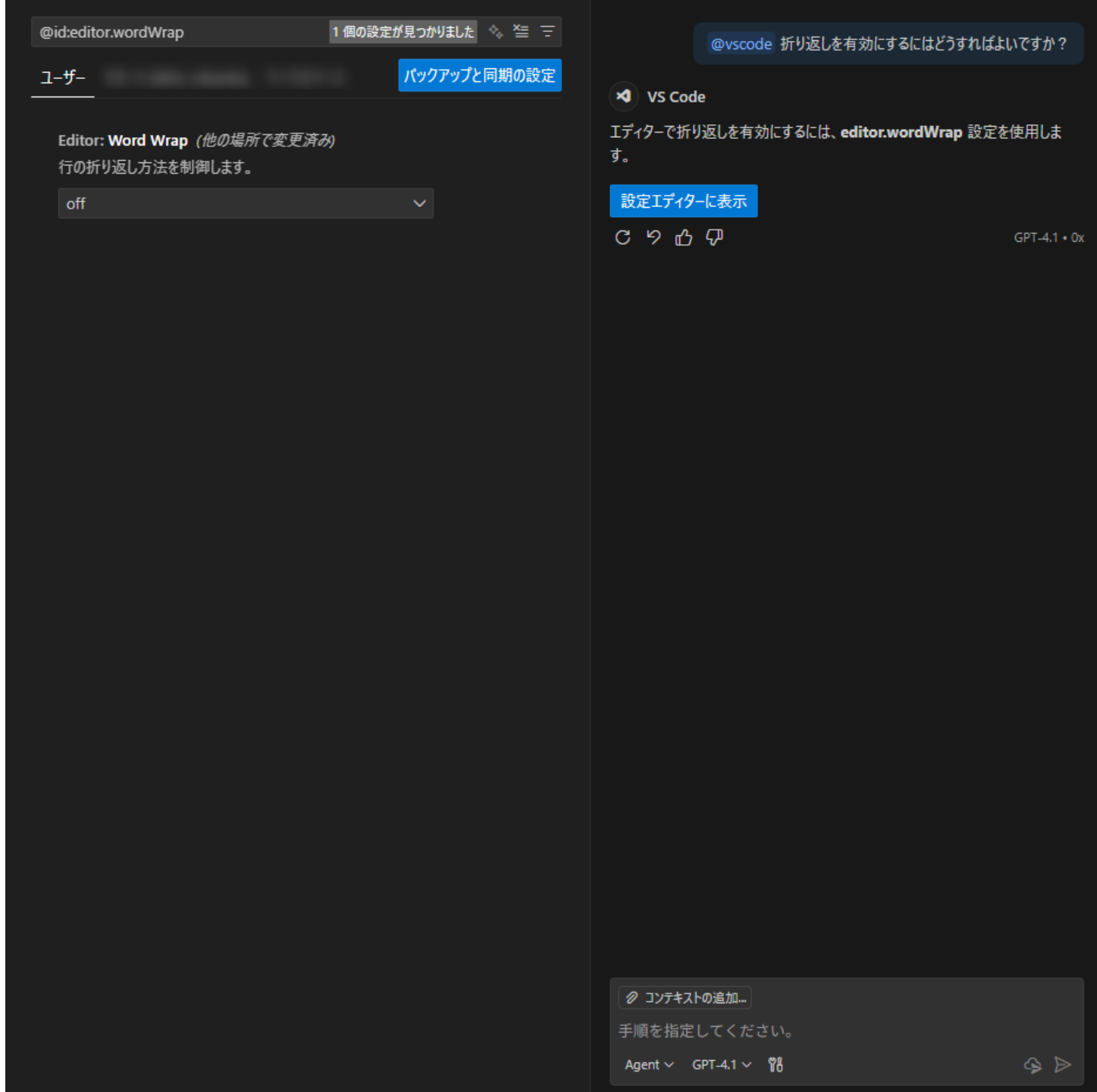
最後のコマンドは、python3 calc.py を実行して calc.py ファイルの Python スクリプトを実行しました。

このスクリプトは division(5, 0) を呼び出して 5 を 0 で割ろうとしましたが、ゼロ除算（ZeroDivisionError）が発生し、エラーで停止しました。

3.6 Visual Studio Code に関する質問を行う

@vscode

Visual Studio Code コマンドと機能に関するコンテキストを含めます。
Visual Studio Code に関するヘルプが必要な場合に使用します。



3.6 @vscode

使用場面例：

Visual Studio Code に関して、エディタ内の折り返し設定方法を確認します。

期待される出力：

折り返しの有効化手順が出力されます。

プロンプト例

`@vscode` 折り返しを有効にするにはどうすればよいですか？

【出力例】

折り返し（ワードラップ）を有効にするには、`editor.wordWrap` 設定を `"on"` に変更してください。

また、ショートカットキー `Alt+Z` で折り返しの切り替えも可能です（コマンド: `View: Toggle Word Wrap`）。

3.6 @vscode

使用場面例：

Visual Studio Code に関して、Gitの操作方法を確認します。

期待される出力：

Git 操作関連の Visual Studio Code コマンドが提案されます。

プロンプト例

```
@vscode Gitの使い方を教えてください。
```

【出力例】

Visual Studio CodeでGitを使うには、以下の手順を実行してください。

1. Gitが有効になっていることを確認します。**git.enabled**設定がtrueになっている必要があります。
2. リポジトリを開くには、**Git: Open Repository**コマンドを使います。
3. リモートの変更を取得するには、**Git: Fetch**コマンドを使います。
4. Gitの出力ログを確認したい場合は、**Git: Show Git Output**コマンドを使います。

Gitの詳細な使い方や機能を拡張したい場合は、マーケットプレイスで「git」関連の拡張機能を検索してください。

3.6 @vscode

使用場面例：

Visual Studio Code に関して、コードのリファクタリングに活用できる機能を確認します。

期待される出力：

リファクタリング操作として活用可能なコマンド例が提案されます。

プロンプト例

@vscode コードのリファクタリングに役立つツールはありますか？

【出力例】

コードリファクタリングに役立つ主なツールと手順です。

1. 対象コードを選択し Refactor... で抽出・インライン化などのリファクタを実行（ショートカット: Ctrl+Shift+R, コマンド: editor.action.refactor）
2. 迅速な修正やインポート整理などは Auto Fix...（ショートカット: Shift+Alt+., コマンド: editor.action.autoFix）を試す
3. 変更結果を差分で確認するには Source Control: Focus on Changes View で変更ビューにフォーカス（コマンド: workbench.scm.focus）
4. 連鎖的な名前変更後など検索結果を更新したい場合は Search: Refresh（コマンド: search.action.refreshSearchResults）
5. 追加のリファクタ支援拡張機能を探すには Marketplace 検索を使用

3.7 明示的に参照ファイルを指定する

#files

コンテキストとして含めるファイルを明示的に指定します。

3.7 #files: ファイル参照機能

使用場面例：

仕様書や API 定義書を参照しながら実装を進めます。

期待される出力：

参照ファイルの内容に準拠した実装とテストコードが提案されます。
※一度では網羅されない可能性があるため、抜け漏れ確認のために繰り返し実行を推奨します。

プロンプト例

`#files:requirements.md #files:api-spec.yaml` の内容に基づいて、ユーザー認証APIを実装してください。

実装要件：
- requirements.mdの機能要件を完全に満たす
- api-spec.yamlのエンドポイント定義に準拠
- セキュリティ要件（JWT、OWASP準拠）を実装
- TypeScript + Express.jsで実装
- 適切なエラーハンドリングとログ出力

実装手順：
1. 仕様書の要件整理
2. エンドポイント設計確認
3. 認証ミドルウェア実装
4. バリデーション実装
5. テストコード作成

参照ファイルの内容と実装の整合性を保ってください。

3.7 #files: ファイル参照機能

使用場面例：

複数の設計ドキュメントとの整合性を確認します。

期待される出力：

各ドキュメントとの整合性チェック結果と修正内容が提案されます。

プロンプト例

実装内容が以下のドキュメントに準拠しているかチェックしてください：

```
#files:design/architecture.md - システム設計書
#files:tests/test-scenarios.md - テストシナリオ定義
#files:docs/coding-standards.md - コーディング規約

チェック項目：
1. アーキテクチャパターンの準拠
2. テストシナリオの網羅性
3. コーディング規約の遵守
4. ドキュメントとコードの整合性

不整合がある場合は修正提案をしてください。
```

3.7 #files: ファイル参照機能

使用場面例：

既存の設定ファイルを参照して環境構築を自動化します。

期待される出力：

設定ファイルの内容に基づいた包括的な環境構築スクリプトが提案されます。

プロンプト例

`#files:package.json #files:docker-compose.yml #files:README.md` を参照して、開発環境のセットアップを自動化してください。

構築項目：

1. 依存関係のインストール
2. 環境変数の設定
3. データベース初期化
4. サービス起動スクリプト
5. ヘルスチェック実装

考慮事項：

- `package.json`の`scripts`セクション活用
- `Docker`環境での開発
- `README.md`の手順との整合性
- 開発者向けドキュメントの更新

クロスプラットフォーム対応（`Windows/Mac/Linux`）も考慮してください。

4. GitHub Copilotカスタマイズ機能

GitHub Copilot カスタマイズ機能

- GitHub Copilot には、「振る舞いを定義するためのコンテキスト」を予め自動で渡すための設定があります
  - コンテキストとして与える情報によって GitHub Copilot の振る舞いを調整することができます
  - チャットに繰り返し追加する必要のあるような前提条件は、この自動読み込みコンテキスト内で定義することが可能です
- また、繰り返し使用するプロンプトをコマンド化するために、カスタムプロンプトファイルの定義手段があります
- 本章では、GitHub Copilot をプロジェクトの内容や進め方に合わせてカスタマイズする方法を学びます

4.1 GitHub Copilot に自動適用されるカスタム命令

4.1 .github/copilot-instructions.md

.github/copilot-instructions.md は、ワークスペース全体に適用される永続的なカスタムインストラクションの設定を定義するファイルです。

このファイルが配置された後、ワークスペース内のすべての Copilot Chat で自動的に適用されます。

4.1 .github/copilot-instructions.md

使用場面例：

チーム開発でコーディング規約を自動的に AI に適用させます。

期待される出力：

copilot-instructions.md ファイルをもとに、プロジェクト全体で統一されたコーディング規約に準拠したコード生成が行われるようになります。

.github/copilot-instructions.md

```
# プロジェクト共通コーディング規約

## 命名規則
- クラス名・インターフェース名：PascalCase
- 変数名・関数名：camelCase
- 定数：ALL_CAPS
- プライベートメンバ：アンダースコア(_)プレフィックス

## TypeScript規約
- 型安全性を最優先
- any型の使用禁止
- Optional chaining (?.) と Nullish coalescing (??) を積極活用
- 関数型プログラミング原則に従う

## エラーハンドリング
- async操作には必ずtry/catch
- Reactコンポーネントには適切なError Boundary
- エラーは必ずコンテキスト情報付きでログ出力

## セキュリティ
- OWASP Top 10対策を必須とする
- ユーザー入力は必ずバリデーション・サニタイズ
- 機密情報はログに出力しない
```

4.1 .github/copilot-instructions.md

使用場面例：

モデルのトーンを自動的に AI に適用させます。

期待される出力：

モデルが過剰に礼儀正しくなるような振る舞いが減ります。

モデルは何をすべきか/何をしないかを知る必要がありますが、例えば「～しないで」の代わりに「～を避けて」のように肯定的な表現をすることがポイントとなります。

.github/copilot-instructions.md

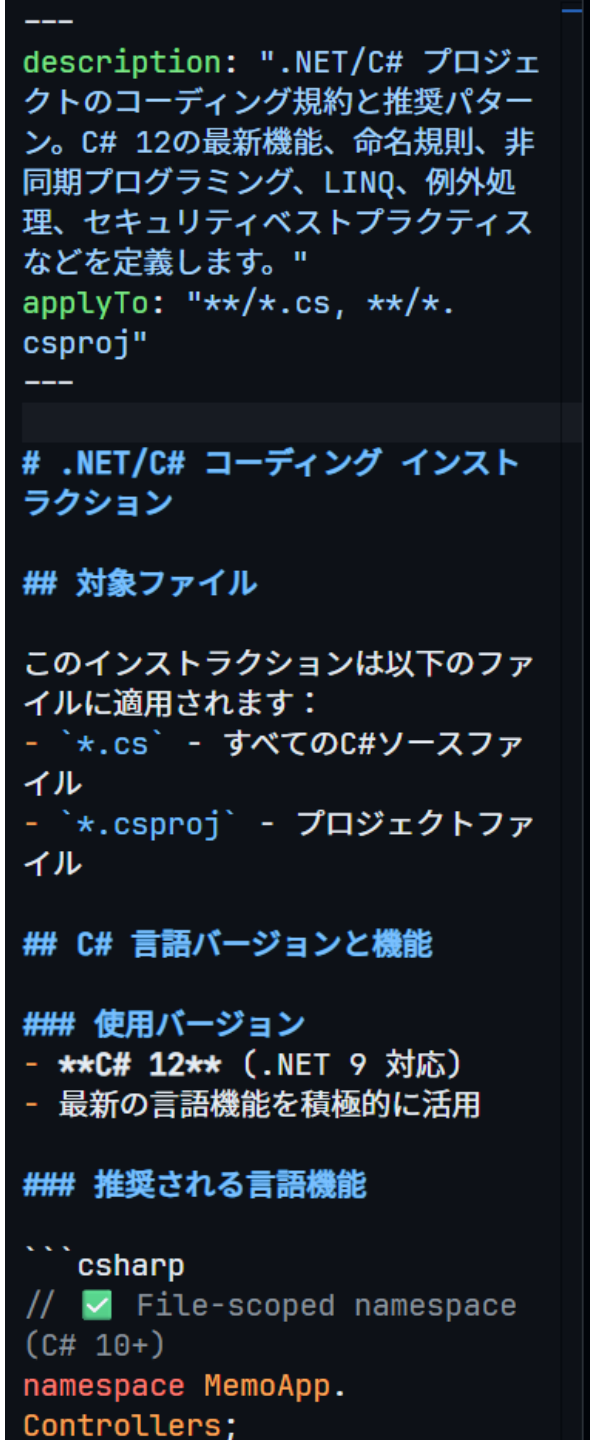
- 間違っていると指摘された場合は、それが真実かどうかを考え、事実に基づいて回答してください
- 「その通りです」や「はい」といった同意の表明は不要です
- 謝罪や譲歩的な発言は避けてください
- 誇張や興奮は避け、実用的に作業を完了してください

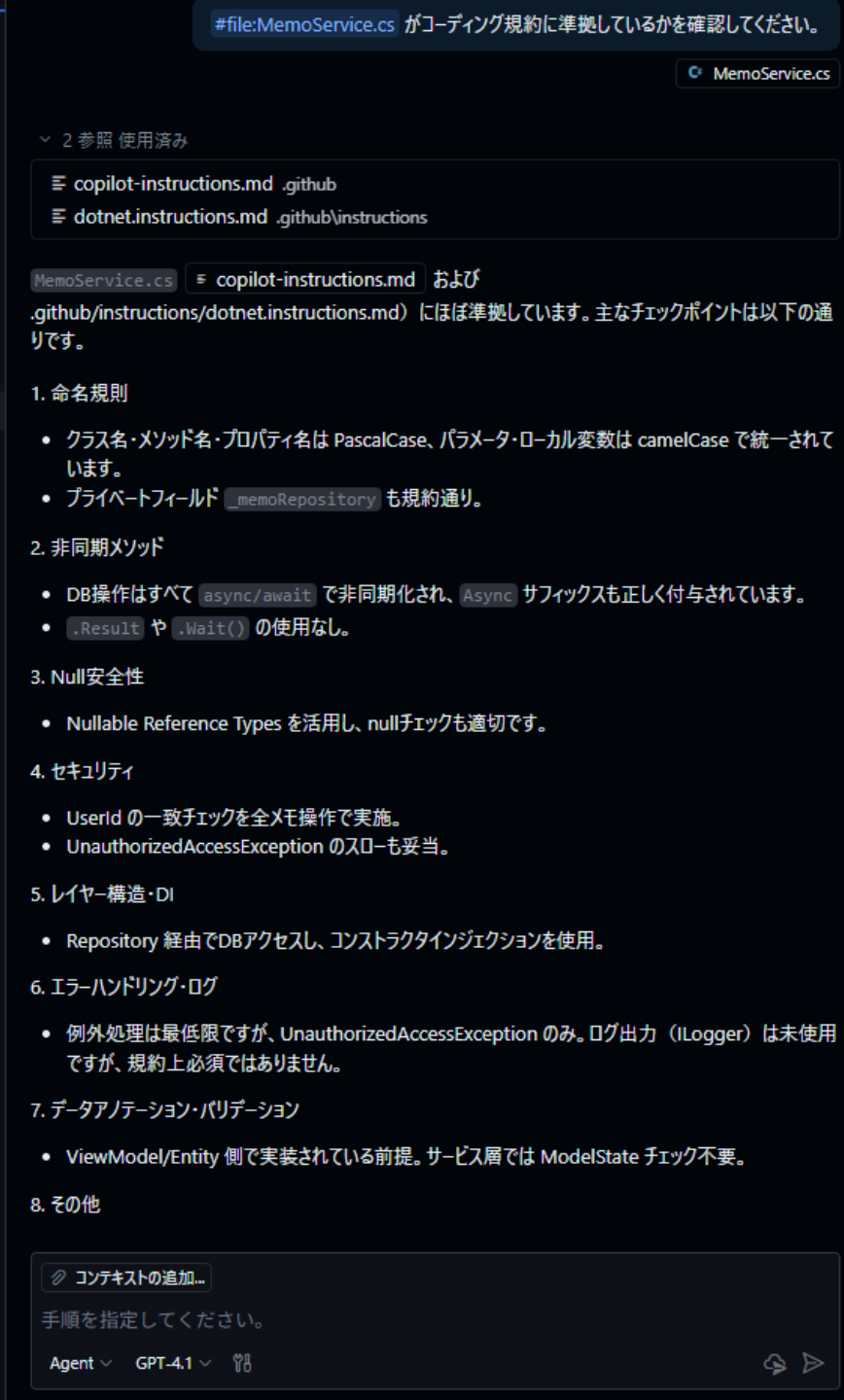
4.2 Coding agent で自動適用されるカスタム命令

4.2 パス指定可能なカスタム命令

.github/instructions/<INSTRUCTIONS_NAME>.instructions.md は、Coding agent および Code review 機能において使用されるインストラクション定義ファイルです。

特定のファイルのみにインストラクションを適用する目的で、適用先のファイルパスを指定することができます。





4.2 パス指定可能なカスタム命令

使用場面例：

Coding agent および Code review のために、特定のファイルタイプやタスクに特化した指示を定義します。

期待される出力：

指定されたファイルタイプ（.ts, .tsx）に対して、React および TypeScript のコーディング規則に則ったコードが生成されるようになります。

.github/instructions/ts.instructions.md

```
---
applyTo: "**/*.ts,**/*.tsx"
---

# React/TypeScript専用指示

## React開発ガイドライン
- 関数コンポーネント + Hooks必須
- クラスコンポーネント使用禁止
- React.FCタイプを使用（children有りの場合）
- 1コンポーネント1ファイル原則
- CSS Modulesでスタイリング

## Hooks使用規則
- useEffect依存配列は必須
- カスタムHooksは"use"プレフィックス
- useCallbackとuseMemoで適切にメモ化
- 条件付きHooks使用は厳禁

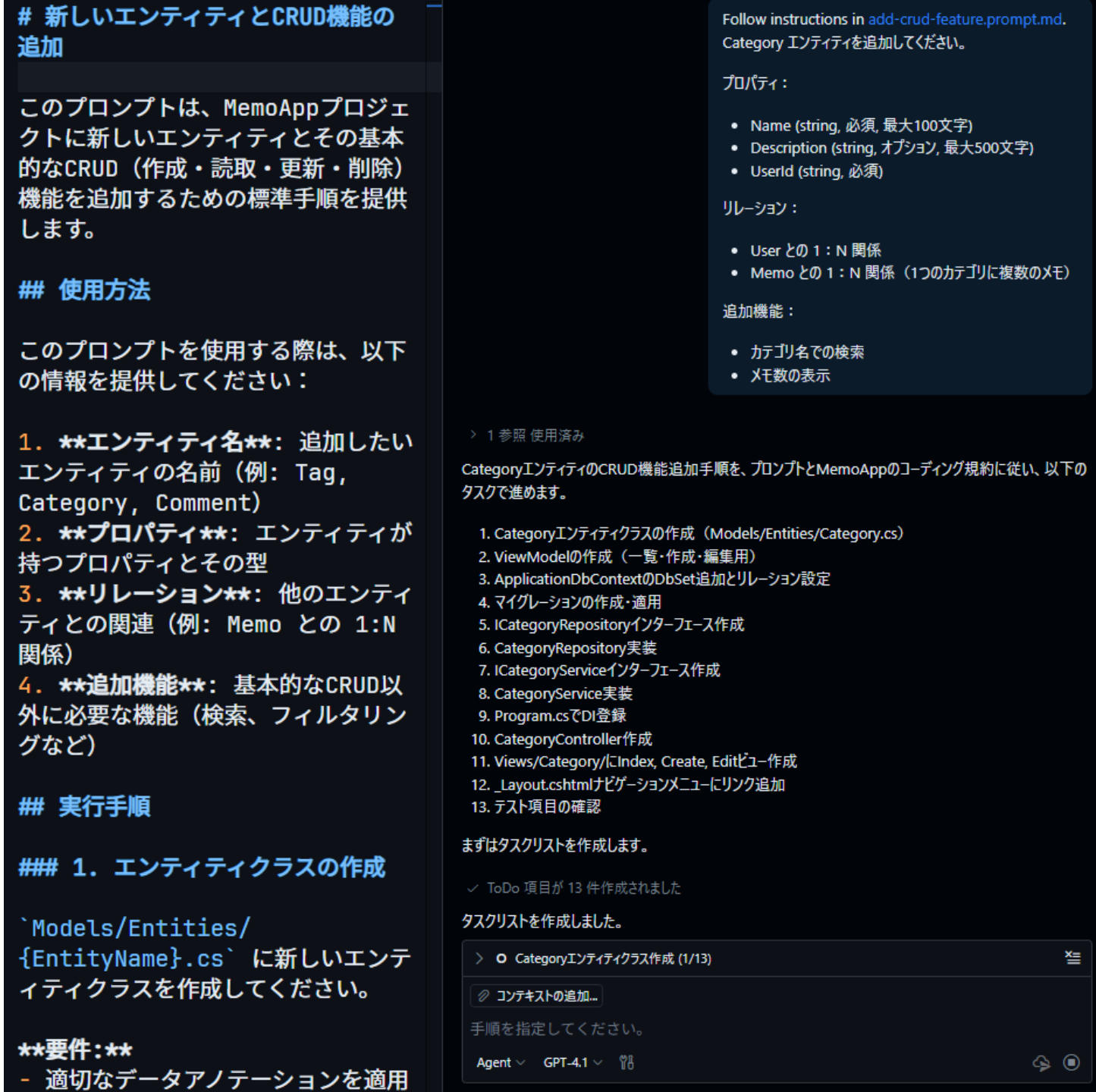
## パフォーマンス重視
- 不要な再レンダリング防止
- React.memoで適切にメモ化
- バンドルサイズ最適化
- Code Splittingの積極活用
```

4.3 再利用可能なカスタムプロンプトファイル

4.3 カスタムプロンプトファイル

<PROMPT_NAME>.prompt.md ファイルとして再利用可能なプロンプトテンプレートを作成します。

Copilot Chat からは、/PROMPT_NAME 形式で使用することができます。



4.3 カスタムプロンプトファイル

使用場面例：

仕様書に基づいた実装作業用のプロンプトを、再利用可能な形で定義します。

期待される出力：

プロジェクトに含まれる特定のドキュメントを読んだうえでのソースコード実装が行われます。

implement.prompt.md

```
---
description: '仕様書に基づいた実装を段階的に実行'
---
以下の手順・規約に沿ってソースコード実装を行ってください。

## 実装手順
1. **仕様書確認**: docs/requirements.md の内容を精読
2. **既存コード分析**: ドキュメント上のディレクトリ構成図をもとに関連コードを検索
3. **実装計画**: 機能を小さな単位に分割
4. **段階的実装**:
   - インターフェース定義
   - 基本機能実装
   - テストコード作成
   - エラーハンドリング追加
5. **品質確認**: docs/quality.md の項目をもとに問題をチェック
6. **ドキュメント更新**: 実装内容をREADMEに反映

## 実装規約
- 型安全性を最優先
- 関数は単一責任原則に従う
- 適切なコメント・JSDoc記述
- テストカバレッジ90%以上維持
```

4.3 カスタムプロンプトファイル

使用場面例：

標準化されたコードレビュープロセス用のプロンプトを、再利用可能な形で定義します。

期待される出力：

予め定められたレビュー観点をもとにコードレビューが行われます。

review.prompt.md

```
---
description: '包括的コードレビューの実行'
---
選択されているコード、または指定されたファイルに対して以下の観点でレビューを実行してください。

## レビュー観点
### ⚡ パフォーマンス
- 計算量・時間複雑度
- メモリ使用効率
- データベースクエリ最適化
- フロントエンド描画性能

### 🖌️ 保守性
- コードの可読性
- 関数・クラスの責任分離
- DRY原則遵守
- 適切なコメント・ドキュメント

### 🧪 品質
- テスト容易性
- エラーハンドリング
- エッジケース対応
- ログ出力の適切性
```

4.3 カスタムプロンプトファイル

使用場面例：

体系的なデバッグプロセス用のプロンプトを、再利用可能な形で定義します。

期待される出力：

予め定められたデバッグ手順をもとにソースコードデバッグが行われ、エラーの分析と修正までが行われます。

debug.prompt.md

```
---
description: '体系的なデバッグとエラー解決'
---
以下の手順・観点に沿ってデバッグを実行してください。

## デバッグ手順
1. **問題の特定**: ビルド結果からエラー・警告を確認
2. **エラー分析**: スタックトレース・エラーメッセージの詳細分析
3. **関連コード調査**: エラーメッセージから関連する使用箇所を検索
4. **根本原因特定**:
- データフロー追跡
- 依存関係の確認
- 環境固有の問題調査
5. **修正実装**: 最小限かつ確実な修正
6. **テスト実行**: 修正後の動作確認
7. **回帰テスト**: 他機能への影響確認

## デバッグ観点
- 型エラー・実行時エラー
- パフォーマンス問題
- メモリリーク
- 非同期処理の問題
- ブラウザ固有の問題
```

4.4 MCP を利用して外部データをコンテキストとして扱う

MCP連携

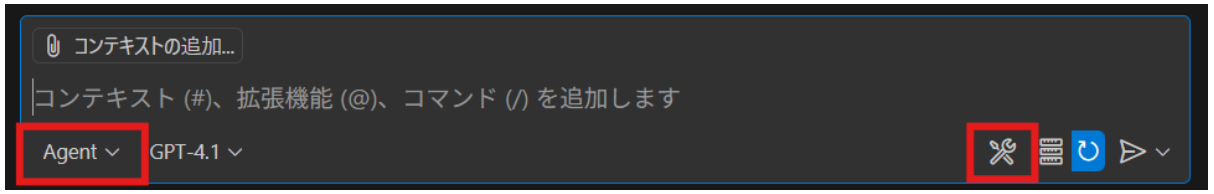
MCP を登録することで、Agent mode 使用時に外部データをコンテキストとして使用することができます。

4.4 MCP 連携（Visual Studio Codeでの設定手順例）

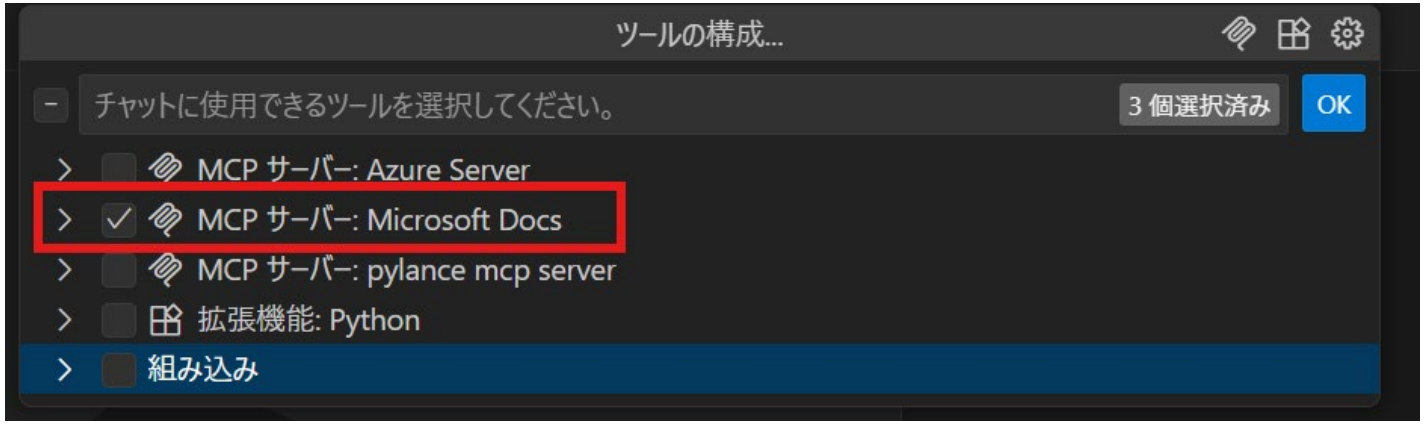
1. コマンドパレットから「MCP: サーバーの追加」を実行します
2. 追加する MCP サーバーの種類を選択します
   - 例: HTTP
3. MCP サーバーの URL を入力します
   - 例: https://learn.microsoft.com/api/mcp
4. MCP サーバーの名前（任意の値）を入力します
   - 例: mcp-mslearn
5. MCP サーバー設定を保存する場所（設定を反映する範囲）を選択します
   - Global: Visual Studio Code インストールディレクトリ下に設定が保存され、全体で使用されます
   - Local: .vscode ディレクトリ下に設定が保存され、単一ワークスペース内のみで使用されます

4.4 MCP 連携（Visual Studio Codeでの設定手順例）

追加した MCP サーバーを確認するには、Copilot Chat を開き、エージェントモードに切り替えて、ツールの構成を押します。



新しく追加した MCP サーバーが一覧に表示されます。



4.4 MCP 連携

使用場面例：

Microsoft Learn の内容をコンテキストに含めたい場合に使用します。

期待される出力：

Microsoft Learn 上の記載内容をもとにした実装内容や質問回答が提案されます。

特に Microsoft 関連技術の具体的な提案や、最新版に関する情報が必要な場合に役立ちます。

mcp.json

```
{
  "servers": {
    "mcp-mslearn": {
      "url": "https://learn.microsoft.com/api/mcp",
      "type": "http"
    }
  },
  "inputs": []
}
```

4.4 MCP 連携

使用場面例：

Microsoft Learn の内容をコンテキストに含めたい場合に使用します。

期待される出力：

Microsoft Learn 上の記載内容をもとにした実装内容や質問回答が提案されます。

特に Microsoft 関連技術の具体的な提案や、最新版に関する情報が必要な場合に役立ちます。

プロンプト例

#mcp-mslearn .NETの最新LTSバージョンに関する情報を教えてください。

【出力例】

公式ドキュメントを確認して、.NET の最新LTS（Long Term Support）バージョンとそのサポート期限、Windows向けのインストール手順（PowerShell/WinGetなど）を日本語で短くまとめます。参照元の公式URLも添えます。

要点（日本語まとめ）

- 最新のLTSバージョン: .NET 10（LTS）
- リリース日: 2025年11月11日（公式リリース）
- サポート終了（EOL）: 2028年11月14日（公式サポートポリシーに基づく）
- 直近のSTS（短期サポート）: .NET 9（STS、例示） – リリース日: 2024年11月12日、サポート終了: 2026年11月10日
- 補足: リリースは毎年11月。偶数バージョンがLTS、奇数バージョンがSTS（サポート期間がそれぞれ3年／2年）。

Windowsでのインストール（簡易手順）

～省略～

参考（公式ドキュメント）

- Releases and support for .NET（サポートポリシー、LTS/STSの説明、サポート期限一覧）
https://learn.microsoft.com/en-us/dotnet/core/releases-and-support
- .NET サポート ポリシー（ライフサイクル）
https://dotnet.microsoft.com/platform/support/policy/dotnet-core
- Install .NET on Windows（インストール手順、PowerShell/WinGetの使い方）
https://learn.microsoft.com/en-us/dotnet/core/install/windows

5. GitHub Copilot における
エージェントコーディング手法紹介

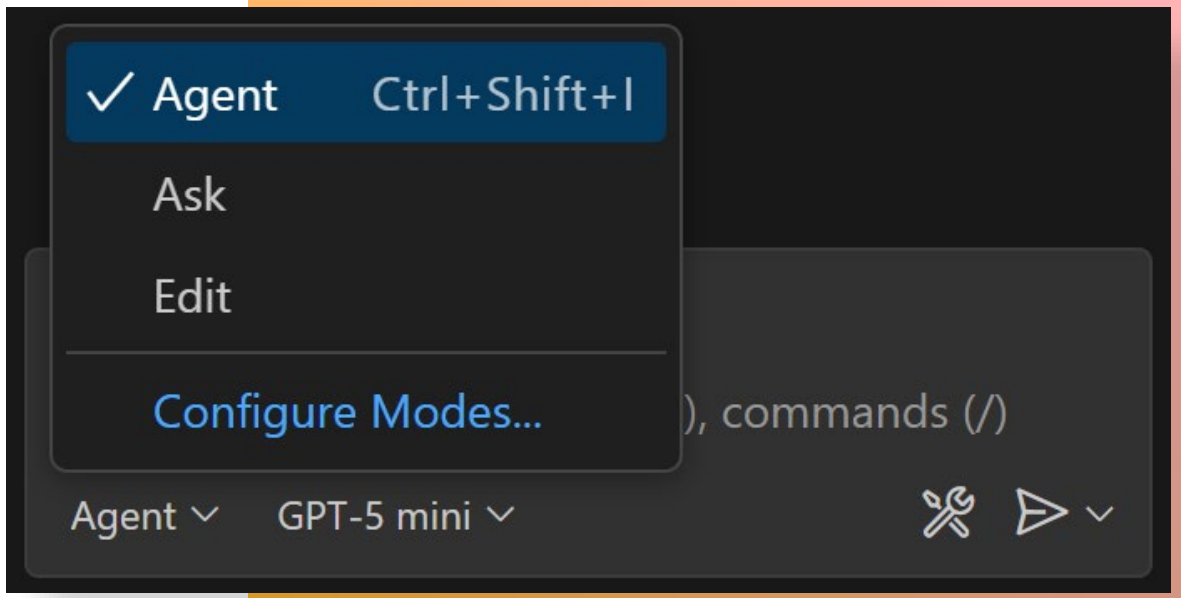
5.1 Agent mode (Copilot Edits)

Copilot Edits

Copilot Edits は、Visual Studio Code、Visual Studio、JetBrainsのIDEで利用可能（2025/12 現在）であり、単一のCopilotチャットプロンプトから複数のファイルで直接変更できる機能です。

チャットウインドウから Edit または Agent を使用することができます。

Edit は Agent に比べて、素早く具体的な変更を指示したい場合や GitHub Copilot のリクエスト数をコントロールしたい場合に使用します。



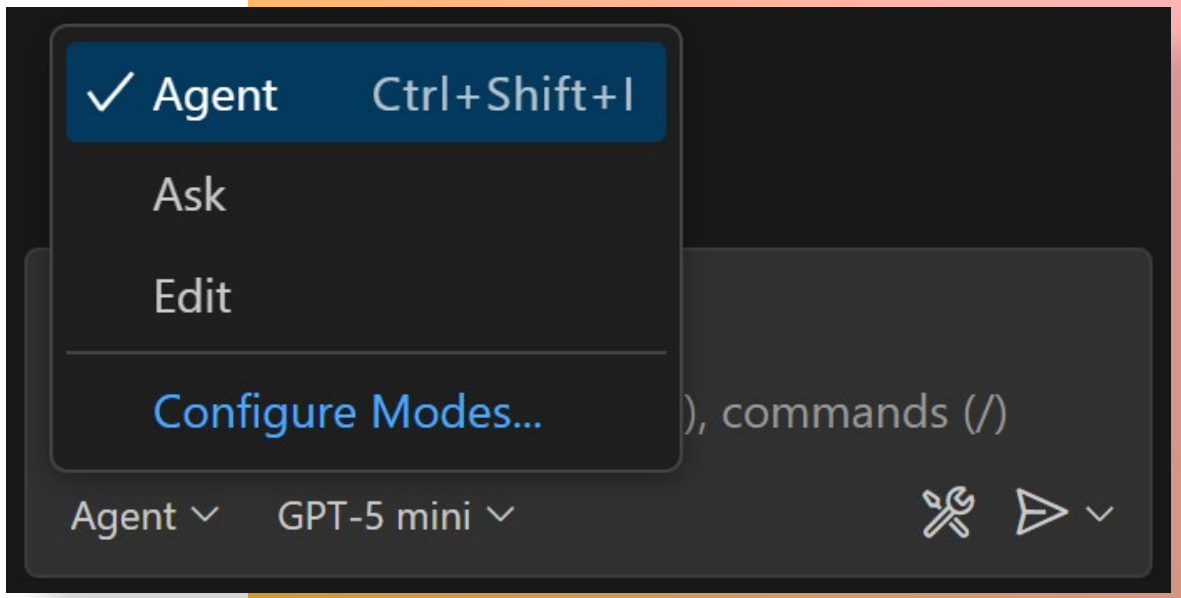
Agent mode

Agent mode は、GitHub Copilot Chat で選択できるモードの一つです。

GitHub Copilot がコードを自律的に編集し、ターミナルコマンドの実行も併用することで与えられたタスクが完了するまで修復を繰り返します。

次のようなユースケースに適しています。

- タスクが複雑で、複数のステップ、繰り返し、エラー処理を含む
- タスクの完了に必要な手順を GitHub Copilot に決定させる
- GitHub Copilot がタスクを実行するために MCP サーバーなどの外部アプリケーションと統合する



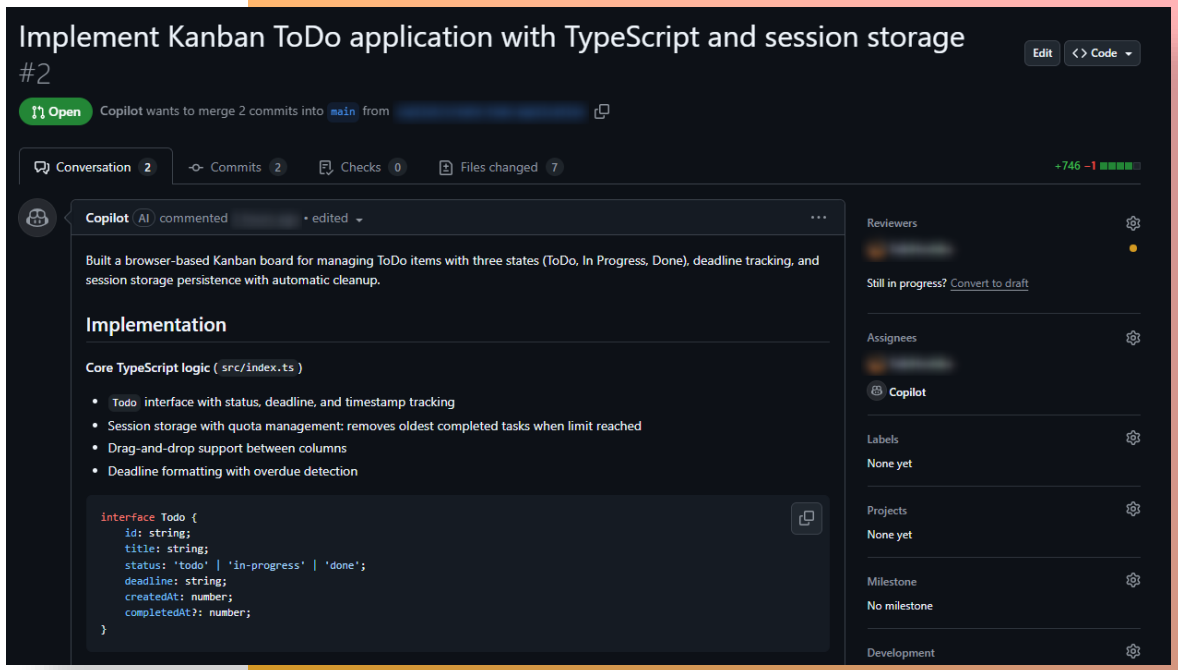
5.2 Coding agent

Coding agent

Coding agent は、GitHub.com 上で GitHub Copilot がバックグラウンドで独立して動作し、人間の開発者と同様にタスクを遂行します。

GitHub Copilot にタスクを委任するには、次の方法があります。

- Issue の assignees に Copilot を割り当てる
- GitHub のエージェントパネルや GitHub Copilot Chat、MCP がサポートされている任意の IDE や CLI ツールで Pull request を作成するよう Copilot に依頼する



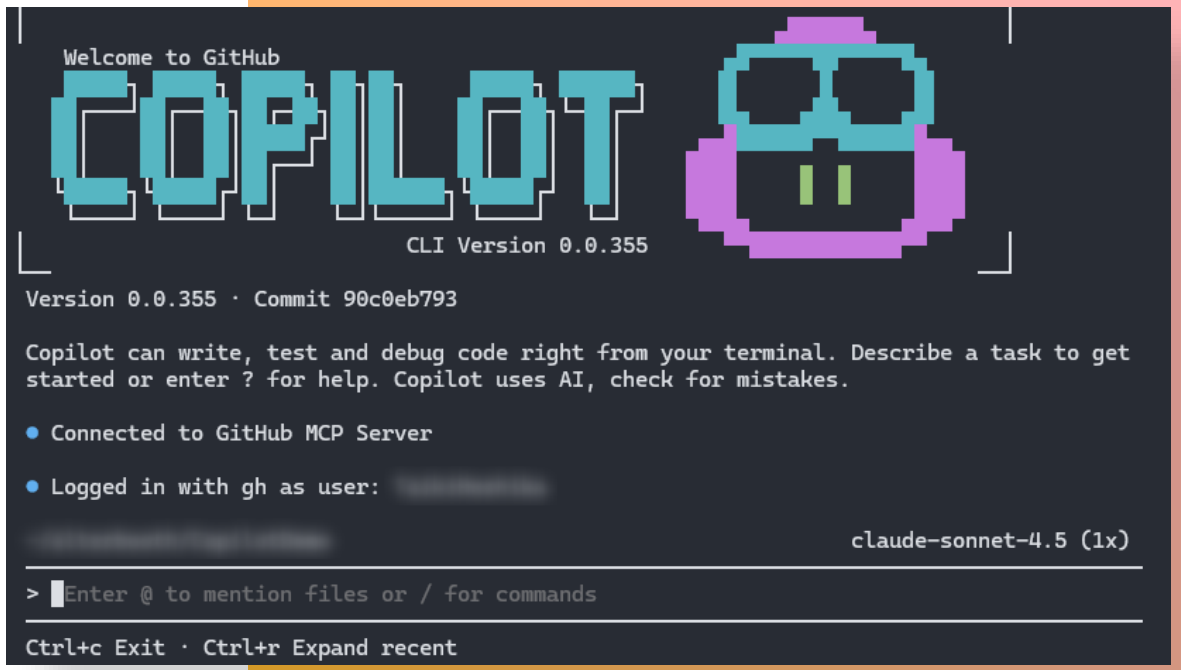
参考: [GitHub Copilot コーディング エージェントについて - GitHub Enterprise Cloud Docs]

5.3 GitHub Copilot CLI

GitHub Copilot CLI

GitHub Copilot CLI を使用して、ターミナルから直接 GitHub Copilot を利用することができます。

GitHub Copilot への質問やタスク実行を促すことができるインタラクティブモードと、単一プロンプトを渡してファイルの編集および実行を自律的に行うプログラムモードがあります。



※ 2025/12 現在パブリックプレビューです。

参考: GitHub Copilot CLI · GitHub

5.4 Copilot コードレビュー

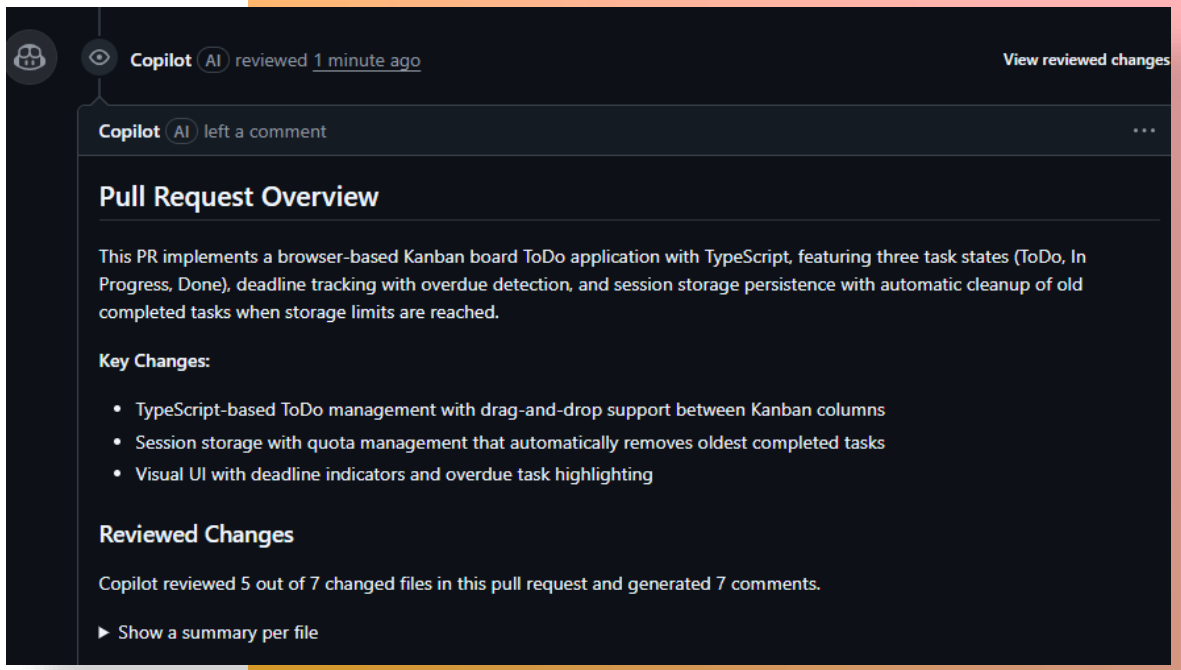
Copilot コードレビュー

Copilot コードレビューは、ファイルの差分または指定したソースコード箇所対して GitHub Copilot がコードレビューを提供する機能です。

GitHub Copilot にコードレビューを依頼するには、次の方法があります。

- Pull Request の Reviewers に Copilot をアサインする
- （対応IDEのみ）未コミットのファイル差分に対して IDE 上でコードレビューを実行する
- （Visual Studio Code のみ）ファイルの特定箇所をハイライトし、コードレビューを実行する

参考: GitHub Copilot コード レビューの使用 - GitHub Enterprise Cloud Docs

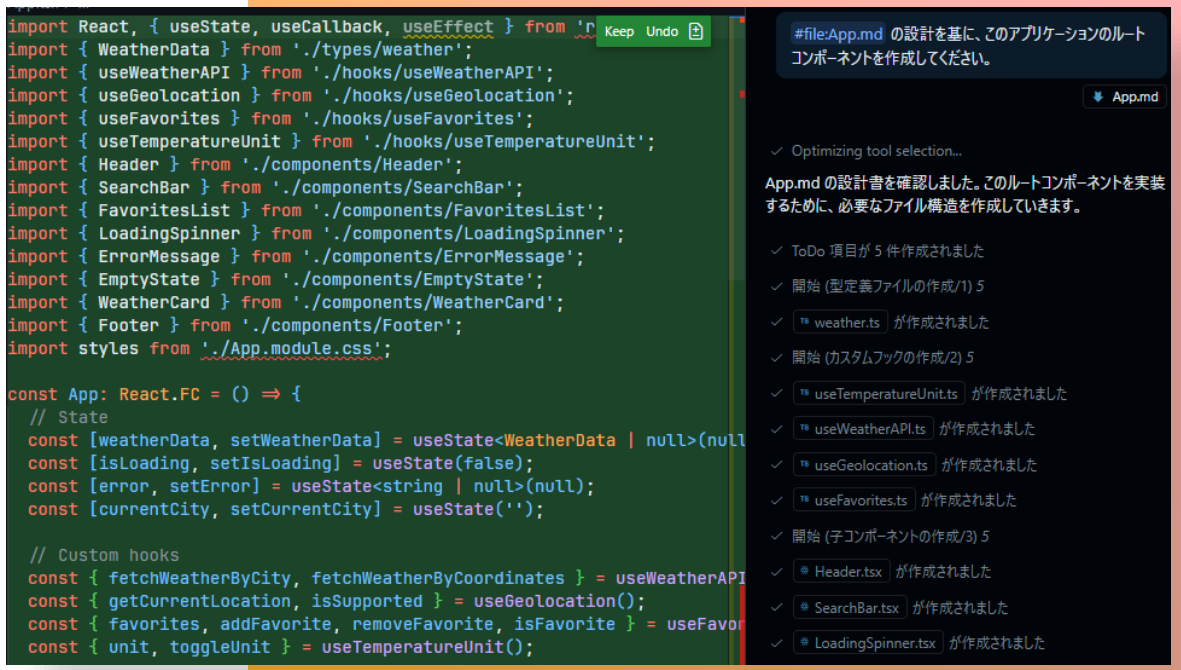


5.5 スペック駆動開発

スペック駆動開発

スペック駆動開発とは、要件定義書や仕様書などのドキュメントを AI を使用して生成・整備し、作成したドキュメントをコンテキストとして AI に渡す形で実装を進めるといった AI 駆動開発手法です。

ドキュメントの更新内容をソースコードに反映することはもちろん、既存の実装からドキュメントを更新させるといった双方向フィードバックにより、人間と AI の認識のずれを少なくします。



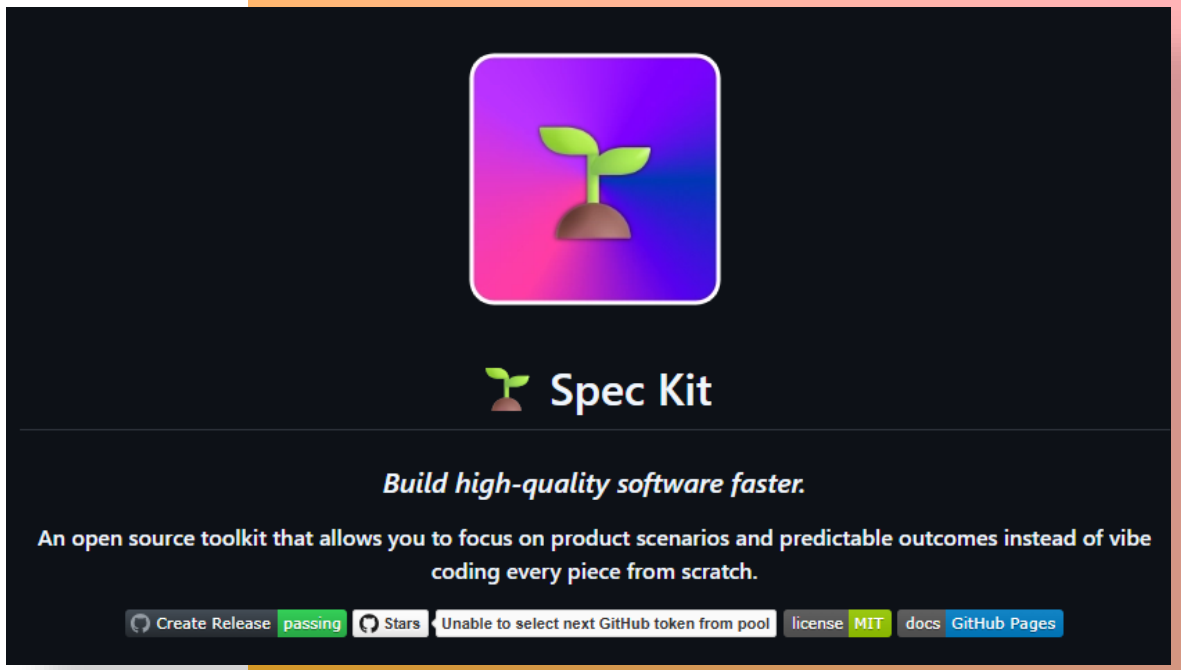
紹介: Spec Kit

Spec Kit は GitHub が提供している OSS で、スペック駆動開発を実践するためのプロジェクトテンプレートです。

GitHub Copilot をはじめ、Claude Code や Gemini CLI など様々な AI コーディングツールに対応しています。

テンプレートには、仕様の検討から作成及び更新のためのプロンプトや、正確な出力パスやその他検証項目のために使用されるスクリプトが含まれます。

参考: GitHub - github/spec-kit: 💫 Toolkit to help you get started with Spec-Driven Development



6. ユースケース

6.1 プロジェクト設計およびタスク化

6.1 プロジェクト設計およびタスク化

ポイント：

設計内容およびタスクリストをファイルとして出力させます。

今後実装指示を行う際、実装レビューを行う際、タスクの続きを指示する際に、ファイルを永続化コンテキストとして参照させることができます。

プロンプト例

```
---
description: "プロジェクト設計およびタスク化を行います"
---
# プロジェクト設計・タスク化プロンプト
入力された要件を分析し、`PROJECTS.md`（設計）と `TASKS.md`（タスク管理）の2ファイルに構造化して出力します。

## 作業フロー
1. **事前確認**: `PROJECTS.md`・`TASKS.md` の存在確認。既存の場合は更新内容を表示し、継続/新規/部分更新を選択
2. **情報収集**: 不足している必須情報（プロジェクト名、目的、技術制約、スケジュール等）をユーザーに質問
3. **生成**: 下記テンプレートに従い2ファイルを生成
4. **検証**: 要件とタスクの紐付け、依存関係の循環参照、工数の妥当性を確認
5. **提示**: ユーザーに提示し承認を得る

## PROJECTS.md 出力内容
- **要件分析**: 機能要件（優先度付き）、非機能要件、制約条件、リスクと緩和策
- **アーキテクチャ**: システム構成図、技術スタック選定理由、データモデル、API設計
- **実行計画**: マイルストーン、依存関係グラフ、クリティカルパス

## TASKS.md 出力内容
**タスク分解ガイドライン:**
- パラメータ: タスク番号、名称、説明、依存関係、工数、優先度、ステータス、チェックボックス
- 粒度: 1タスク = 1-8時間（1日以内完了）
- 各タスクは独立して実行・テスト可能で、明確な完了条件（DoD）を持つ
- ❌NG例:「フロントエンド実装」/ ✅OK例:「ログインフォームUI実装」
```

6.2 既存プロジェクト分析

6.2 既存プロジェクト分析

ポイント：

一度全体の概要を分析し、分析後のアクションに合わせてより詳細確認が必要な分析項目を提示します。

また、その「分析後のアクションに合わせた分析結果」を把握するために、プロンプト文章そのものの生成を指示することもできます。

プロンプト例

当プロジェクトに関して、下記の情報を教えてください。
- 言語
- フレームワーク
- 依存関係
- 外部サービス連携

当プロジェクトに関して、.NET Framework から .NET LTS への移行を検討しています。
まずは移行のために当プロジェクト全体の分析を行います。
分析実施および分析結果からの移行パス提示を指示するプロンプト文章を生成してください。

6.2 既存プロジェクト分析

（「分析後のアクションに合わせた分析結果」を把握するために生成させたプロンプト文章例を提示します。）

分析後のアクションに合わせて分析項目を提示されていること、また、分析後のアクションパスが提案されていることが確認されます。

生成結果例

```
---
description: "既存プロジェクト分析を行います"
---
# .NET Framework から .NET LTS への移行分析
## 分析前の情報収集
- [ ] `*.csproj` / `*.sln` 全ファイル
- [ ] `packages.config` / `app.config` / `web.config`
- [ ] `using` ステートメント頻出パターン（grep検索）

## 分析項目
1. プロジェクト構成
   - ターゲットバージョン、プロジェクトファイル形式（旧形式/SDK）、ソリューション構造
2. 互換性チェック
   - NuGet パッケージの .NET (Core) 対応状況
   - Windows 固有 API、WCF/Remoting/AppDomain 等の非対応技術使用箇所
3. コードベース分析
   - `System.Web` 依存、設定ファイル項目、Binary Serialization 使用箇所
4. マイグレーションタスク
   - **形式**: `- [ ] [優先度:高/中/低] [カテゴリ] タスク内容`
   - **カテゴリ**: プロジェクト構成 / 依存関係 / コード修正 / テスト / デプロイ
5. リスク評価
   - 難易度（低/中/高）、ブロッカー特定、代替技術提案
6. 推奨移行パス
   - **移行戦略**: 段階的移行（既存システム稼働継続）か一括移行（短期間で完全移行）を提案し、理由を明記
   - **移行順序**: 依存関係を考慮した実施順序（例: 共通ライブラリ → ビジネスロジック → UI層）
   - **工数概算**: 各フェーズの所要工数と全体スケジュール
```

6.3 マイクロサービスプロジェクトにおける機能追加

6.3 マイクロサービスプロジェクトにおける機能追加

ポイント：

既存プロジェクトの分析結果をコンテキストに含められるようにします。

また、分析では特に複数サービスの関連性を把握させます。

関連性がコンテキストに含まれることにより、単純な実装タスクの提案に留まらず他サービスとの接続も加味した実装タスク（接続テストなどを含むもの）を提案させることができます。

プロンプト例

既存のマイクロサービスプロジェクトに新機能を追加するために、実装タスクを体系的に生成します。

```
## 事前準備
1. 追加する機能の要件概要を確認
2. 以下を並行取得: サービス一覧、API定義、データモデル、サービス間通信、テストコード

## タスク生成
- **形式**: `- [ ] [優先度:高/中/低] [カテゴリ] タスク名（対象サービス） – DoD: 完了条件`
- **カテゴリ**: スキーマ変更 / API設計 / サービス実装 / サービス間連携 / テスト / ドキュメント / CI/CD / 監視
- **タスク粒度**: 1タスク = 2-8時間
  - ❌NG:「ユーザーサービス実装」
  - ✅OK:「ユーザーサービスのプロフィール更新API実装」
- 各タスクに含める:
  - 対象サービス・ファイルパス
  - 推定工数（h）
  - 依存関係（先行タスク番号）
  - 実装時の注意点

## 出力形式
`docs/feature-tasks.md` に以下を出力:

1. **分析サマリ**: 影響を受けるサービス数、新規API数、変更が必要なエンティティ数
2. **タスクリスト**: 優先度・カテゴリ別に整理されたチェックボックス付きタスク一覧
3. **依存関係図**: 主要タスクの実施順序を示すMermaid図
4. **リスク・注意事項**: 実装時の注意点、既存機能への影響
```

6.3 マイクロサービスプロジェクトにおける機能追加

ポイント：

既存プロジェクトの分析結果をコンテキストに含められるようにします。

また、分析では特に複数サービスの関連性を把握させます。

関連性がコンテキストに含まれることにより、単純な実装タスクの提案に留まらず他サービスとの接続も加味した実装タスク（接続テストなどを含むもの）を提案させることができます。

プロンプト例

本マイクロサービスプロジェクトに[機能名]を実装します。包括的なテスト戦略を提案してください。

要件
1. 関連するすべてのサービスを特定し、それぞれの変更内容を提案してください
   - サービス間の依存関係と影響範囲を分析してください
   - データモデルの変更が必要な場合は、移行戦略を含めて提案してください
   - APIの変更がある場合は、後方互換性の維持方法を提案してください
2. 各層のテストを網羅してください
   - 単体テスト（ユニットテスト）
   - 統合テスト（サービス内）
   - サービス間統合テスト
   - E2Eテスト
3. 各テストについて以下を提案してください
   - テストの目的と範囲
   - 具体的なテストケース例
   - 使用を推奨するテストツール/フレームワーク
   - モック/スタブの戦略
4. テストデータの準備方法
5. CI/CDパイプラインでのテスト実行戦略

機能の詳細
[機能の説明]

6.4 単体テストおよび自動化ワークフロー実装

6.4 単体テストおよび自動化ワークフロー実装

ポイント：

テストの種類・対象を明記します。

ワークフロー内での実施対象を提案させ、決定したものを明記します。

段階的実装や新規差分把握のため、実装進捗を都度出力させます。

プロンプト例

既存プロジェクトに包括的な単体テストを実装し、CI/CDパイプラインを構築します。

事前準備
1. テスト対象範囲（全体 or 特定モジュール）を確認
2. 以下を並行取得：プロジェクト構造、既存テストコード、カバレッジ設定、CI/CD設定、重要な機能・モジュール

実施手順
1. テスト戦略
   - 優先度：クリティカルパス > ビジネスロジック > ユーティリティ
   - 目標カバレッジ率：80%以上（クリティカルパスは100%）
2. 単体テスト実装
   - **テスト粒度**：1関数/メソッドあたり3-5テストケース
   - 正常系（期待値パターン）
   - 異常系・境界値（null、空、最大/最小値）
   - エッジケース
   - モック/スタブ利用（外部依存の分離）
3. CI/CD自動化
   - ワークフロー設定（GitHub Actions/Jenkins等）
   - トリガー：PR作成時、main/developブランチへのpush時
   - テスト結果・カバレッジレポート自動生成

出力形式
`docs/test-implementation.md` および実装ファイルを生成：
1. **実装サマリ**：追加したテスト数、達成カバレッジ率、対象モジュール
2. **テストコード**：既存パターンに従った単体テスト
3. **CI/CD設定**：ワークフロー設定ファイル（`.github/workflows/test.yml`等）
4. **運用ガイド**：テスト実行方法、カバレッジ確認方法

6.5 脆弱性パターンの検出とレポート出力

6.5 脆弱性パターンの検出とレポート出力

ポイント：

GitHub Copilot を使用して、コード内の一般的な脆弱性を見つけ、修正の提案を受けることができます。

（但し、この方法は完全に再現性のあるルールベースのチェックではないため、包括的なセキュリティ分析のためには code scanning を使用することを推奨します。）

プロンプト例

このコードを分析し、潜在的なセキュリティ上の脆弱性を検出し、修正を提案します。

```
function displayName(name) {
  const nameElement = document.getElementById('name-display');
  nameElement.innerHTML = `Showing results for "${name}"`
}
```

【出力例】

```
function displayName(name) {
  const nameElement = document.getElementById('name-display');
  nameElement.textContent = `Showing results for "${name}"`;
}
```

6.5 脆弱性パターンの検出とレポート出力

ポイント：

検出項目（SQL インジェクション、XSS、CSRF など）を明記します。

分析結果をどのように提示させるかを明記しましょう。ここでは、レポート共有の目的で結果をファイル出力させる例を提示します。

また、開発中に随時使用できるよう、繰り返し使用できるプロンプト内容にします。

プロンプト例

プロジェクトのソースコードを分析し、セキュリティ脆弱性を検出してレポートを生成します。

事前準備
1. スキャン範囲（全体 or 特定ディレクトリ）を確認
2. 以下を並行取得：プロジェクト言語・フレームワーク、依存ライブラリ一覧、認証・認可実装箇所、データベースアクセス箇所、外部入力処理箇所

検出対象（OWASP Top 10準拠）
- SQLインジェクション（パラメータ化されていないクエリ）
- XSS（サニタイズされていない出力）
- 認証・認可の不備、機密情報ハードコード
- コマンドインジェクション、パストラバーサル
- 安全でない暗号化（MD5/SHA1/弱い鍵長）
- CSRF対策欠如、XXE、安全でないデシリアライゼーション
- 依存ライブラリの既知の脆弱性

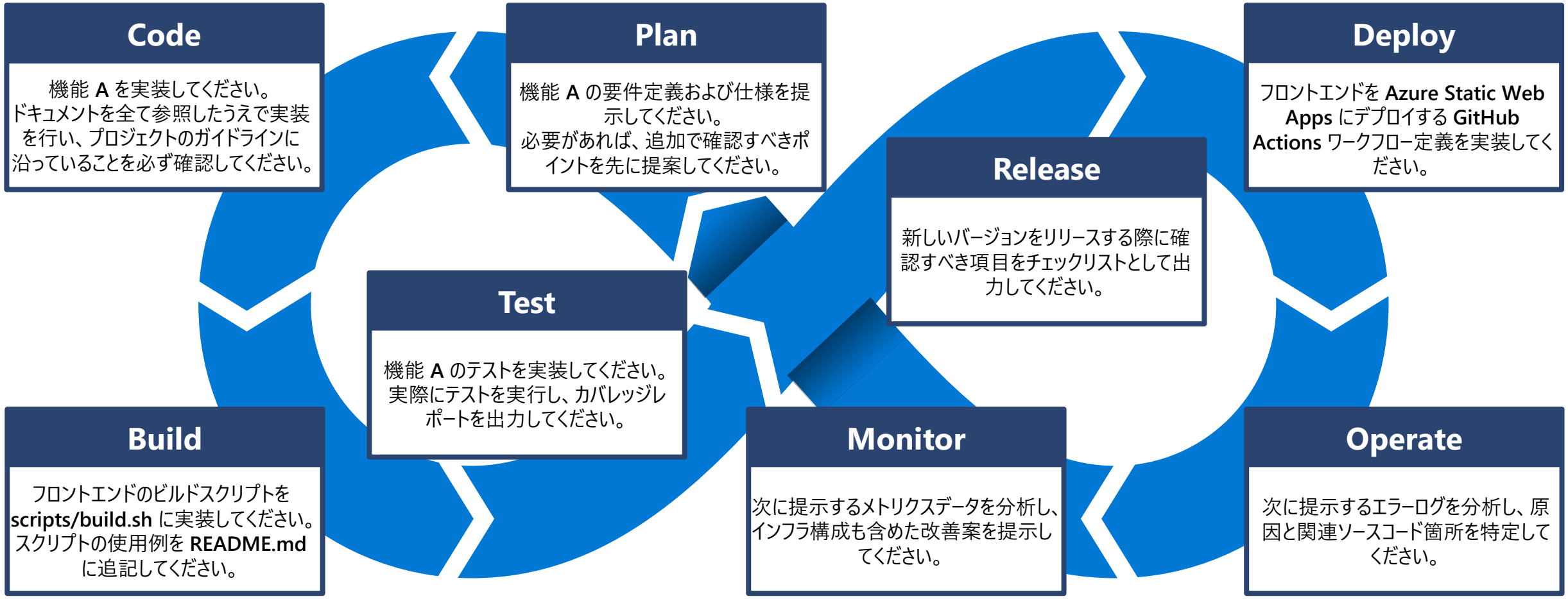
分析手順
1. 言語・フレームワーク固有のセキュリティベストプラクティスを適用
2. パターンマッチングとコンテキスト分析で脆弱性を検出
3. 誤検知の可能性がある場合は `[要確認]` タグを付与
4. 重要度順（Critical > High > Medium > Low）にソート

出力形式
`docs/security-report.md` に以下を出力：
1. **エグゼクティブサマリ**：検出数（重要度別）、リスクスコア、優先対応項目
2. **脆弱性詳細**：種類、重要度、ファイルパス:行番号、スニペット、影響範囲、修正方法
3. **依存ライブラリ脆弱性**：CVE番号、影響バージョン、推奨更新バージョン
4. **改善推奨事項**：全体的なセキュリティ強化策

6.6 SDLC におけるプロンプト例

6.6 SDLC におけるプロンプト例

コーディングに限らず、様々な用途に合わせて GitHub Copilot を活用できます。

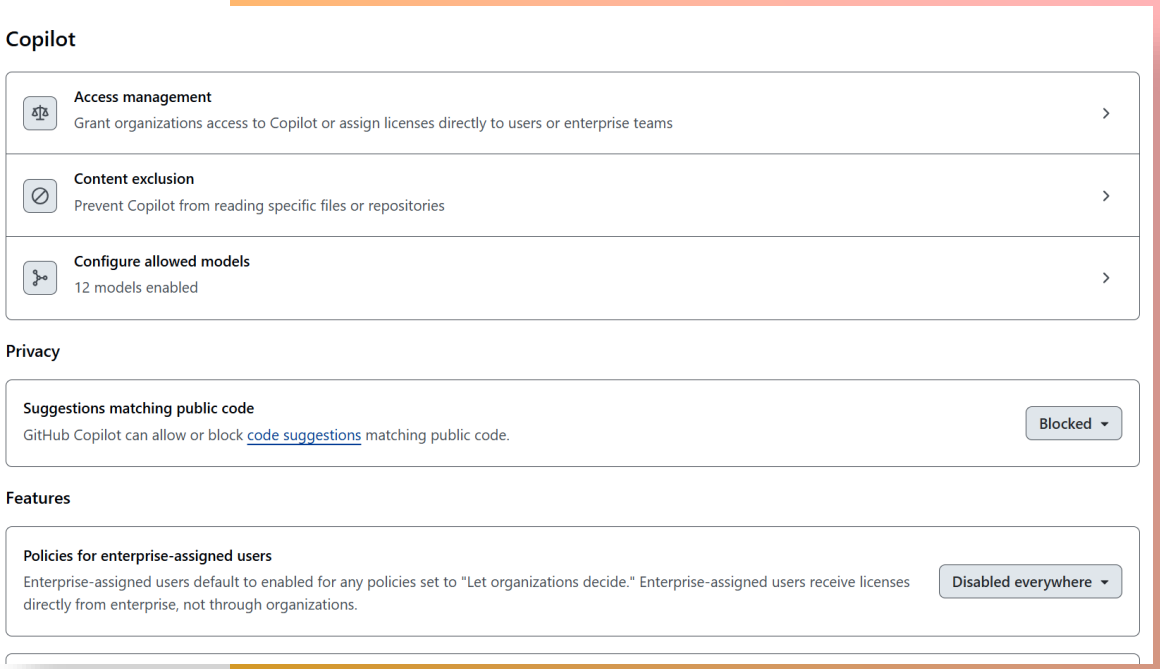


6.7 ポリシーを使用して安全に GitHub Copilot を使用する

6.7 Copilot ポリシーの設定

GitHub Copilot では Enterprise または Organization の単位で、ライセンス管理に加えて各機能や各モデルの使用可否を管理することができます。

また、Premium リクエスト上限超過の利用可否や matching public code のブロックに関する設定もポリシー内で行うことが可能です。



管理対象機能の参考:複数の organization で GitHub Copilot のポリシーが競合するときの機能の利用可能性 - GitHub Enterprise Cloud Docs

Appendix

参考資料

- GitHub Copilot のドキュメント - GitHub ドキュメント
- GitHub Copilot Chat クックブック - GitHub ドキュメント
- GitHub Copilot in VS Code
- Get started with chat in VS Code
- Use custom instructions in VS Code
- Use prompt files in VS Code
- Manage context for AI
- GitHub - github/awesome-copilot: Community-contributed instructions, prompts, and configurations to help you make the most of GitHub Copilot.
- OpenAI Cookbook