

投稿日 2024/10/11

AI初学者向けGitHub Copilot導入事例

はじめに

TISでは学生向けに開発体験型のイベントを開催しており、その中で生成AI技術の活用を体験してもらうべくGitHub Copilotを導入することにしました。

参加者に向けた事前アンケートで、大半がAIを利用したプログラミングに不慣れであると想定されたため、短期間(5日間)のイベントでGitHub Copilotを活用して成果を出してもらうためには、単にライセンスを用意するだけでは不十分であり、事前の準備が不可欠であると考えました。

本稿では、AI活用経験の浅い人にも短期間で成果を出してもらえるよう検討したこと、実際にやってみてわかったこと、今後に活用できることについて記載します。

利用シーンの選定

GitHub Copilotはチャット機能（GitHub Copilot Chat）も含めると様々な利用シーンが想定されます。

まずは具体的な利用シーンを洗い出し、その利用シーンを実現するにはどのような準備が必要か検討しました。

★マークが付与されたものが今回活用することにしたものです。

利用シーン	説明	準備	コメント
コード生成	やりたいことをコメント等に記載してコードを生成する	特になし	初学者が狙ったコードを生成させるのは難しい。生成されたコードの妥当性を判断するのも難しい。
既存と似たようなコード生成	繰り返し出現する似たようなコードを生成する	雛形になるソースコード	今回のイベントでは繰り返しが必要な場面が乏しいため除外した。
コメント生成	ソースコードからコメントを生成する	特になし	有用であるが、独力で作業する点にはあまり寄与しない。
テストコード生成	ソースコードからテストコードを生成する	特になし	今回のイベントではテストコードはスコープ外とした。

利用シーン	説明	準備	コメント
既存コードを説明させる★	既存コード中の不明な箇所を選択して説明させる	特になし	解説の精度が高く有用と判断。無闇に検索エンジンで調べるより効率的。
エラーを説明させる★	発生したエラーを説明させる	特になし	解説の精度が高く有用と判断。無闇に検索エンジンで調べるより効率的。
エラーを修正させる	発生したエラーを修正させる	特になし	提案結果から別のエラーが発生しうる。ある程度の内容理解は必要で「エラーを説明する」で代替する。
ワークスペースの説明をさせる★	ワークスペース内のコードを説明させる	プロジェクト固有の情報を記載したドキュメント、ソースコードコメント	「xxx画面のコードはどこ？」のようなアプリ固有の質問に回答が得られ、スタッフへの質問回数低減が見込める。
やりかたをざっくり聞く★	やりたいことをざっくり聞く	プロジェクト固有の情報を記載したドキュメント、ソースコードコメント	「xxx一覧を日付でソートしたい」のようなアプリ固有の質問に回答が得られ、スタッフへの質問回数低減が見込める。
ソースコードレビュー	指定したソースコードの改善点を聞く	レビュー観点を記載したドキュメント。	修正した箇所だけ指摘するのが難しい。また、補助的なものでありレビューアの代替にはならない。

検討した結果、我々はGitHub Copilotによるコード生成よりも、GitHub Copilot Chatによる説明を活用することにしました。理由は、知識・経験が不十分な状態では生成されたコードの妥当性を判断するのが難しいからです。これまでの試行で、知識・経験が不十分な状態でAIを利用すると以下のような悪循環が起こり得ることがわかっていました。

- AIが提示したコードをよく吟味せず適用する
- 利用者が理解できないエラーが発生する
- さらにエラーをAIに質問し、別のエラーが発生する

このような悪循環を抑止するには、GitHub Copilot Chatで説明を受け、ある程度理解したうえで自身で判断し取り込むというプロセスが必要と判断しました。

GitHub社様の説明によると、GitHub Copilotは反応速度重視、GitHub Copilot Chatは回答精度を重視するようモデルを選択しているとのことで、実際に試用してもGitHub Copilot Chatの回答は優秀であると思われました。

以上の点から、GitHub Copilot Chatの活用を軸に準備を進めることにしました。

利用シーンの具体例

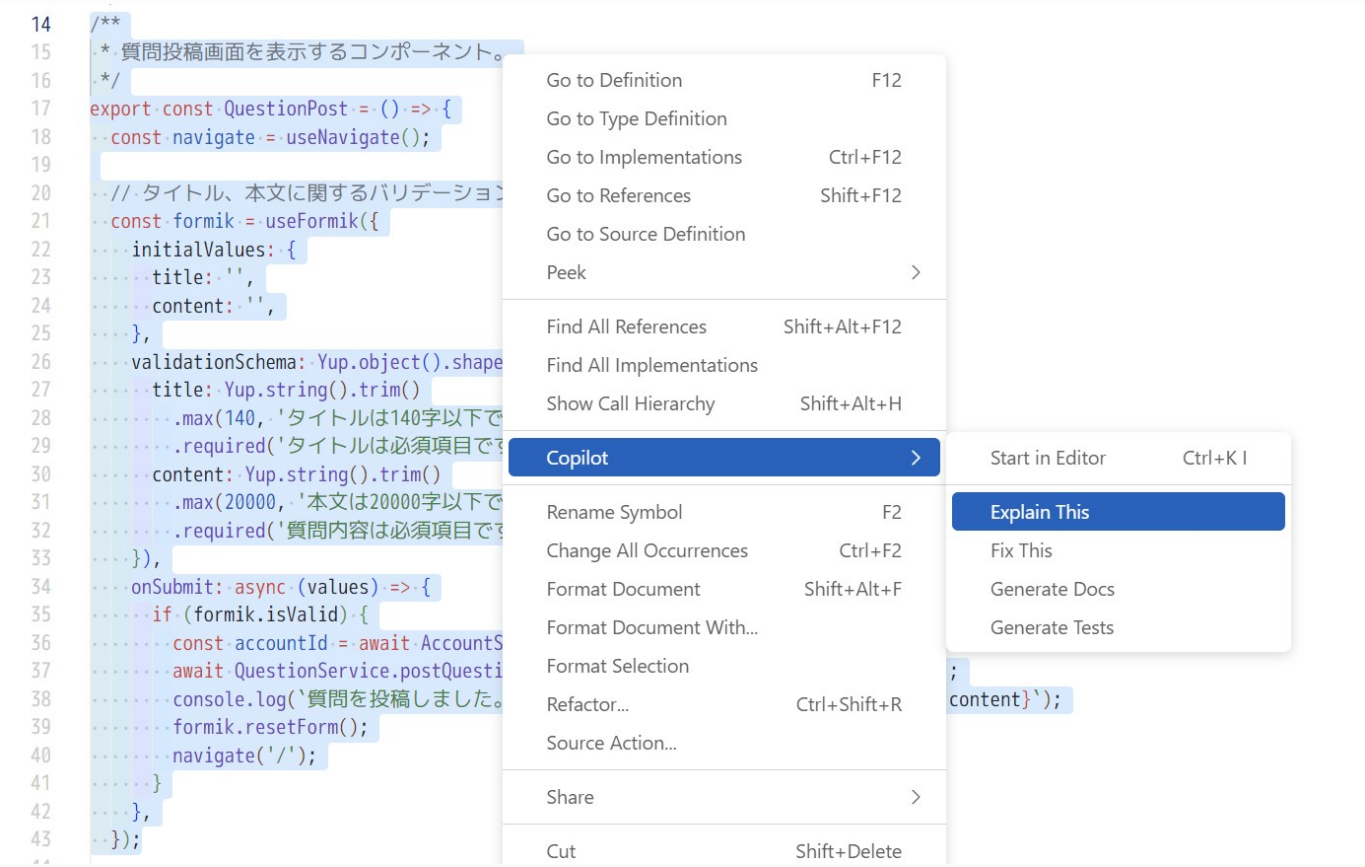
利用シーンの具体例を簡単に説明します。

「既存コードを説明させる」


今回のイベントは土台となるアプリがある状態で、そこに機能追加・修正を加えていきます。そのため既存のソースコード理解が必須となります。

GitHub Copilot Chatは任意のソースコードの箇所に対して解説ができます。既存ソースコード理解はどのようなプロジェクトでも非常に重要ですが、技術的な質問に回答できる人は限られています。このような質問をAIが回答してくれるのはとても有用です。

具体的な使用例を以下に示します。ソースコードの任意の箇所で「Explain this（これを説明する）」を選択します。



この例では、以下のような回答が得られます。

 **GitHub Copilot** used @workspace /explain

Used 1 reference >

このコードは、FormikとYupを使用してフォームのバリデーションと送信を管理しています。FormikはReactのフォームを作成・管理するためのライブラリで、YupはJavaScriptのオブジェクトのバリデーションを行うライブラリです。

まず、`useFormik` フックを使用してFormikのインスタンスを作成しています。このフックはオブジェクトを引数に取り、そのオブジェクト内でフォームの初期値、バリデーションスキーマ、送信時の動作を定義します。

`initialValues` プロパティでは、フォームの初期値を設定しています。ここでは、`title` と `content` の2つのフィールドがあり、どちらも初期値は空文字列です。

`validationSchema` プロパティでは、Yupを使用してバリデーションスキーマを定義しています。`title` フィールドは、空白を削除した後、最大140文字であること、そして必須であることが要求されています。同様に、`content` フィールドも空白を削除した後、最大20000文字であること、そして必須であることが要求されています。

`onSubmit` プロパティでは、フォームが送信されたときの動作を定義しています。まず、フォームが有効かどうかをチェックします。フォームが有効であれば、ログインしているアカウントのIDを取得し、そのIDとともに質問のタイトルと内容をサーバーに送信します。その後、コンソールにメッセージを出力し、フォームをリセットし、ホームページにリダイレクトします。

以上が、このコードの説明です。

「エラーを説明させる」

ソースコードを修正するうちに、エラーや警告が発生することがあります。このようなエラーは初学者にとっては対応が難しく、闇雲に検索エンジンで調査をしても理解が追いつかずに解決しないことがあります。GitHub Copilotは今まさにエラーが起きている箇所について解説をしてくれるため、初学者にとってわかりやすい解説となります。

以下に簡単な例を挙げます。ここでは、ある変数がundefinedである可能性があるがその考慮がされていないためエラーとなっています。

エラー箇所で「Explain using Copilot」を選択します。



The screenshot shows a code editor with the following TypeScript code:

```
const question = questionDetail.question;
const answers = questionDetail.answerList;
let answerCount;
if (answers === undefined) {
  answerCount = '0';
}
```

A red squiggly line is under the `questionDetail` property access. A context menu is open with the following options:

- Quick Fix
- Fix using Copilot
- Explain using Copilot

Below the code, the 'Workspace' section is expanded, showing the error message:

このエラーは、`questionDetail` が `undefined` である可能性があるために発生しています。これを解決するためには、`questionDetail` が `undefined` でないことを確認する条件を追加する必要があります。

以下のように修正できます：

- `questionDetail` が `undefined` である場合にはローディングメッセージを表示します。
- `questionDetail` が存在する場合にのみ、`question` と `answers` を使用します。

前述の「既存コードを説明させる」の項と同様に、技術的な質問に回答できる人は限られているため、AIが回答してくれることは非常に有用です。

「ワークスペースの説明をさせる」

プログラミング言語やライブラリに関する一般的な質問ではなく、開発中のワークスペースに関して質問がしたくなることはよくあります。GitHub Copilot Chatはワークスペース内部のリソースを参照してくれますので、README.md等のテキストファイルに情報を記載しておくことで、そのワークスペース固有の質問に適切に回答してくれるようになります。



The screenshot shows the GitHub Copilot Chat interface. At the top, there is a prompt: `@workspace` モックのダミーデータの初期値ってどこにある？

The response is displayed in a light blue box with the following content:

Workspace

Used 12 references

モックのダミーデータの初期値は、`src/mocks/data`ディレクトリ内に定義されています。

具体的には、以下のファイルにダミーデータが定義されています：

- `src/mocks/data/accountData.ts`
- `src/mocks/data/answerData.ts`
- `src/mocks/data/questionData.ts`

これらのファイルは、`src/mocks/data/index.ts`で読み込まれ、初期化されています。

プロジェクトのドキュメントは、記載があったとしても見落としてしまうことがありますが、GitHub Copilot Chatを仲介することでドキュメントの該当箇所を探す行為をAIに任せられます。これにより、質問者が知り

に聞いて回答を引き出すことができます。

ただし、質問によっては適切な回答を得るためにドキュメントの整備が必要になります。どのような質問がされるか、その質問にどのような回答をさせたいかを事前に検討し、望んだ回答が得られるようにドキュメントを調整する必要があります。

この例の場合ですと、どのディレクトリにどのようなファイルが存在するか記載して、以下のようなテキストドキュメント（markdown）を配置しています。

プロジェクト構成

パス	説明
docs	設計書等のドキュメント類
openapi	OpenAPI関連
src	ソースコード
README.md	プロジェクトの説明資料

ソースコードの構成

パス	説明
src/backend	バックエンドサーバと通信するための部品
src/backend/generated-rest-client	OpenAPI定義から自動生成されるコード
src/components	Reactコンポーネント
src/components/pages	画面の1ページに当たるコンポーネント
src/components/parts	ページを構成するための部品コンポーネント
src/components/Router.tsx	ルーティング定義したコンポーネント
src/mocks	モック
src/mocks/data	モックDBに初期投入されるデータ
src/mocks/handlers	モックが行う処理を記載したハンドラ
src/mocks/models	モックDBを扱うための型定義
src/services	業務処理。主にバックエンドとの通信

これは、人間（参加者である学生）に読んでもらうためではなく、GitHub Copilotに示唆するために配置しています。そのため書き方もAIが理解しやすいように微調整をしています。

例えば、ディレクトリ構造を出力するにはWindowsのtreeコマンドのようなものが便利で、人間にとってはわかりやすいものとなります。しかし、AIにとっては構造を読み取りにくく、上記のような質問にうまく回答してくれませんでした。1対1のテーブル形式で書き直すことで適切な回答が得られるようになりました。

【AIが理解しにくい例：Windows treeコマンド出力結果を流用したもの】

```
└─backend                // バックエンドサーバと通信するための部品
|   └─generated-rest-client  // OpenAPI定義(openapi.yaml)から自動生成
└─components            // Reactコンポーネント
|   └─pages              // 画面の1ページに当たるコンポーネント
|   └─parts              // ページを構成するための部品コンポーネント
└─mocks                 // モック
|   └─data               // モックDBに初期投入されるデータ
|   └─handlers           // モックが行う処理を記載したハンドラ
|   └─models             // モックDBを扱うための型定義
└─services              // 業務処理。主にバックエンドとの通信
```

「やりかたをざっくり聞く」

こちらも考え方は「ワークスペースの説明をさせる」と同様です。**想定したハウツーを予めドキュメントに記載しておくことで、そのプロジェクト固有の質問に回答**できるようにします。

以下の例では、OpenAPI定義を変更する場合にどのような作業が必要か？を質問しています。このような質問に対する回答はプロジェクトによって異なるはずで、一般的な回答では役に立ちません。プロジェクト固有のプロセスに関する質問に対しても回答できるよう、ドキュメントを用意しています。また、ある程度**狙った回答が得られるように、事前に質問を試行して、ドキュメントの内容を微調整**しています。

Workspace

> Used 12 references

OpenAPI定義を変更した後に必要な作業は以下の通りです：

1. `openapi.yaml`の修正

リクエストやレスポンス、モデルに変更が入るため、`openapi.yaml`を修正します。`openapi.yaml`を修正するには、[Swagger Editor](#)というエディタを利用すると便利です。

2. APIクライアントの自動生成

`openapi.yaml`を修正したら、リクエストを送信する部品であるAPIクライアントを自動生成しなおす必要があります。`src/backend`配下のAPIクライアントを自動生成するためには、下記のコマンドをターミナルで実行してください。

```
npm run openapi:gen
```

差分に自動生成したコードが検出されれば成功です。

3. `msw`の修正

モックサーバー（`msw`）を使用している場合、以下の修正が必要です：

3.1 モデルの修正

`src/mocks/models`ディレクトリ内のモデル定義を修正します。

3.2 ハンドラの修正

`src/mocks/handlers`ディレクトリ内のハンドラを修正します。新しいエンドポイントや変更されたエンドポイントに対応する処理を追加・修正します。

3.3 データの修正

`src/mocks/data`ディレクトリ内のモックデータを修正します。新しいフィールドや変更されたフィールドに対応するデータを追加・修正します。

まとめ

これらの手順を踏むことで、OpenAPI定義の変更に伴う必要な修正を行い、アプリケーションが正しく動作するようにします。

導入・教育

新しいツールは単に与えるだけではなかなか使ってもらえません。ツールが役に立つというところを、実際にやってみせて納得してもらうことが必要だと考え、**30分程度の短い実演をイベントに組み込みました**。上記で説明したような利用シーンをデモすることで便利さを実感してもらえるよう工夫しました。

しかしこれだけでは、全ての参加者がすぐに活用してくれるところまでは至りません。便利そうだと

から自分が使ってみるまではまだ隔たりがあります。

そこで、質問を受けたときに、その質問を解決するよう一緒にGitHub Copilot Chatを使ってみせるということを行いました。自分が今まさに困っている事象に対して、AIが有用な回答をしてくれることがわかると自分が使うことのメリットを強く実感できるようになります。

また、チームの誰かが使い方を覚えると、使い方をチーム内で教え合って全体の習熟度が上がるという事象が見られました。このことから、全員の底上げも重要だが、一部のメンバーだけでも早く使いこなしてもらうような施策も有用であると思われました。

注意点

初学者は、AIが生成したコードを理解せずそのまま使ってしまう、意図しない結果になることがあります。

これを防ぐため、AIが生成したコードを理解して使用する必要がある点を説明しました。

AIから得られた回答が望んだものでない場合は、さらに質問したり要望を付け加えることで望んだ回答を得られることも説明しました。

イベントでの導入結果

参加学生が、スタッフの助力なく独力で仕上げられる範囲が増えました。特に、簡単な機能（≒シンプルなプロンプトで狙った回答を得られる機能。例：表示順の並び替え処理）の場合、例年にない速度で実装できた方が複数見られました。より多くの学生を受け入れられるようスタッフ1人あたりの担当数を増やしたいというねらいがあったのですが、参加学生がAIを活用してくれたことで、この点には寄与できたと考えます。

一方で、適切でない回答を受け入れてしまい、その修正に時間を費やしてしまうという事象も発生しました（今回は学生向けイベントで、参加者の自主性を損ねないように留意した点も一因と考えられます）。実際のプロジェクトでは、開始序盤に手厚いサポート体制を構えておくことで、このような事象を軽減できると思われます。

まとめ

今回得られた知見の要点を以下に記載します。

- AIを導入する際は、メンバーにどう利用してもらうかを事前に想定して準備する必要がある
- ドキュメントやコメント等は人間だけでなくAIにも読まれる前提で書く必要がある
- 実際に活用されるようになるまでは、実演やペアプロなどの機会を作ってツールに親しんでもらうようにする
- GitHub Copilotのコード解説、エラー解説は精度が高く、特別な準備も必要ないのですぐに活用できる
- 現時点ではAIはあくまでアシスタントであり、利用者の理解を大きく超えたことができるようになるわけではない

今後の展望

今回は学生向けのイベントでのAI活用でしたが、ここで得られたノウハウはプロジェクトでも活用できると考えます。

AIに回答させるアプローチはメンバーが多いほどスケールします。例えば、アーキテクトチームがプロジェクトメンバーに必要な助力を先回りして検討し、AI活用できるような基盤を用意できれば、少ない有識者でも多くのメンバーの手助けができる可能性があります。

人間が質問に答えることも引き続き必要ですが、こちらは有識者が少なく規模を拡大させにくいという特性があります。従来は、Q&A集のようなドキュメントを充実させることで対応していましたが、このアプローチはドキュメントが肥大化してメンバーが全量把握することが難しくなるという問題がありました。人間とドキュメントの中間にAIを介在させることで、このような問題を回避しつつ、メンバーの増加にも対応できるのではないかと期待されます。

これからは、ソースコード、コメント、ドキュメントなども、AIに読ませることを意識して整備する必要があり、この点はGitHub Copilot以外のAI技術を利用しても変わらず応用できると思われます。

今回は、未経験者・初学者向けにAIを導入する事例として、学生向けイベントでの導入例をご紹介しました。今回の事例が、AI導入に活用頂ければ幸いです。

[/* Recommend */](#)

「Generative AI（生成AI）」のおすすめ記事はこちら

この記事に関連する記事もお読みください。

 Generative AI（生成AI）**NEW**
OctoNihon Forumイベント発表資料「GitHub Copilotを活用した大規模開発の 今 と 未来」公開
2025/06/06  4

 Generative AI（生成AI）
GitHub Copilotを活用した大規模開発～オフショア開発での実践と知見～
2025/02/26  16

 Generative AI（生成AI）
GitHub Copilotを用いたAIによるコードレビューの活用
2025/01/28  26

最近投稿された記事も用意しました。

 Generative AI（生成AI）**NEW**
「金融業界特化型AIエージェントワークショップ」開催レポート
2025/06/13  8

 Generative AI（生成AI）**NEW**
OctoNihon Forumイベント発表資料「GitHub Copilotを活用した大規模開発の 今 と 未来」公開
2025/06/06  4

 ITアーキテクチャ **NEW**
生成AIの新潮流：AIエージェント勉強会を開催しました
2025/05/02  12

「Generative AI（生成AI）」で最も読まれている記事を以下にまとめています。

 Generative AI（生成AI）
GitHub Copilotを用いたAIによる

 Generative AI（生成AI）**NEW**
OctoNihon Forumイベント発表

 Generative AI（生成AI）
GitHub Copilot

Cookie利用について

コードレビューの活用

2025/01/28  26

資料「GitHub Copilotを活用した
大規模開発の 今 と 未来」公開

2025/06/06  4

開発 ~オフショア開発での実践と
知見~

2025/02/26  16