● 第5章 開発リソースは足りないのに増え続ける重圧

5-1 開発リソースが足りない中で開発速度を上げる

新たな課題の顕在化

改善活動を始めて1ヶ月が経過した。湊のチームは確実に変化していた。

要件定義の事前レビュー会が導入され、仕様変更は週に3-5回から週に1-2回に減った。週1回のリファクタリング時間も確保され、技術的負債の返済が少しずつ進んでいる。朝会も定着し、チーム内のコミュニケーションは改善した。

でも、新たな課題が浮上していた。

金曜日の午後、田中部長から呼び出しがあった。

「湊君、改善活動は良いけど、開発スピードは大丈夫？」

田中部長の表情には、経営陣からのプレッシャーが滲んでいた。

「確かに、仕様変更は減りましたし、品質も向上しています。でも、丁寧にやると時間がかかるんです」

湊は正直に答えた。改善活動によって品質は向上したが、開発速度は以前と変わらないか、むしろ少し遅くなった気がする。

「経営陣からは、もっと早く機能を出してほしいと言われている。改善は良いけど、スピードも大事なんだ」

田中部長の言葉に、湊は返答に困った。品質と速度、どちらも大事なのに、どう両立すればいいのか。

部長からの新たなプレッシャー

チームミーティングで、湊は状況を共有した。

「部長から呼び出されて、開発スピードについて話がありました。改善活動で品質は向上したけど、速度は以前と変わらない。経営会議で『競合に後れを取っている。開発スピードを上げろ』と厳しく言われたそうです」

湊は田中部長から聞いた話を詳しく説明した。

「競合A社が先月、勤怠管理にAI分析機能を追加した。我々も四半期以内に対応しないと、機能優位性を失うと部長は言っていました。それから、営業部長からは『機能が足りないと契約を失う。四半期で強豪と機能優位性として遜色がないように5個の新機能を出してほしい』と要望が来ている。マーケティング部からも『新機能のリリースがないと、プロモーション材料がない。四半期ごとに新機能を出さないと、市場での存在感が薄れる』と言われているそうです」

湊は少し間を置いて、田中部長の言葉を続けた。

「部長は、経営陣からの圧力は理解できるが、開発組織として現実的にどう対応していくべきか、現有リソースでどうにかしてほしいというのが本音だと言っていました。人を増やす予算もないし、時間もない。でも、競合に後れを取るわけにはいかない。開発チームに何か手はないか、と相談されました」

山田が腕を組みながら言った。

「人手を増やすしかないのかな？でも、採用には時間がかかるし、教育にも時間がかかる」

「去年、エンジニアを2人採用したが、オンボーディングに3ヶ月かかり、すぐには戦力にならなかった。その間に競合に先を越された」

湊が田中部長から聞いた過去の経験を共有した。

「採用しても、プロジェクトの文脈を理解するのに時間がかかり、期待した成果が出なかった。正社員を採用するにしても、リードタイムの長さや採用面談などの労力を考えると、四半期というタイムリミットには間に合わないと部長が言っていました」

「予算的に人をこれ以上増やすのは難しいです。経営陣からは、現有リソースでどうにかしてほしいと言われています」

飛鳥が困った様子で答えた。

「過去に業務委託でエンジニアをあまり深く考えずに採用したことがあるんですが、契約の関係もあり、あまりチームに馴染めず逆に開発生産性がチームとして下がってしまい、結局やり直しになりました。業務委託での採用には慎重にならざるを得ないんです」

湊は深く考え込んだ。リソースは増やせない。でも、開発速度を上げる必要がある。どうすればいいのか。

「品質を保ちながら、これだけのスピードを出すのは無理だと思う」

山田が率直に言った。

「営業からの要望は理解できるけど、開発チームのキャパシティを考えると…」

飛鳥も不安そうだった。

「毎日残業が続いて、体力的にも限界です」

佐藤が小さな声で言った。チーム全体に疲労感が広がっていた。

「競合A社はAI活用で開発速度を50%向上させたと発表している。我々も対応しないと、さらに差が開く」

山田が業界の動向を共有した。

「業界全体でAI活用が進んでいて、我々だけが取り残されている感がある」

湊は重圧を感じながらも、何か手はないかと考えていた。

AIエージェントにおける湊の疑問

その時、他チームのエンジニアが湊に声をかけた。

「湊さん、GitHub Copilot使ってる？コード書くの速くなるよ」

「GitHub Copilot...ですか」

「うん。AIがコードを自動生成してくれるから、開発速度が上がる。僕も最近使い始めたけど、確かに速くなった気がする」

湊は興味を持った。もしAIツールで開発速度が上がるなら、試してみる価値があるかもしれない。

その日の夜、湊はGitHub　Copilotを試してみた。確かにコードを書くスピードは速くなった。AIが提案するコードをそのまま使えば、開発時間は短縮できそうではあった。

でも、湊は違和感を覚えた。

生成コードへの違和感

生成されたコードを見ると、確かに動く。でも、コードの品質はどうなのか。保守性は？可読性は？技術的負債を増やすことにならないのか。

湊は生成されたコードをじっくり見直した。確かに動くが、無駄な処理があったり、命名が不適切だったり、コメントが不足していたりする。そのまま使うと、技術的負債が増える可能性がある。

「これって本当に生産性向上なのか？」

湊は疑問に思った。速く書けても、後で直す時間が増えたら意味がない。品質を犠牲にして速度を上げるのは、以前経験した失敗と同じではないか。

翌日のチームミーティングで、湊は疑問を共有した。

「GitHub　Copilotを試してみたんですが、確かにコードを書くスピードは速くなりました。でも、生成されたコードの品質に不安があります。そのまま使うと、技術的負債が増える可能性があると思うんです」

山田が頷いた。

「確かに、AIが生成したコードをそのまま使うのは危険だね。でも、AIをうまく使えば、もっと効果的な活用ができるかもしれない」

「もっと効果的な活用...ですか」

「うん。コード生成だけじゃなく、AIを技術的負債の検出に使ったり、テストケースの生成に使ったり、ドキュメント作成に使ったり。もっと戦略的に使えないだろうか？」

山田の言葉に、湊は興味を持った。AIを「コードを早く書く道具」として使うのではなく、「開発全体を効率化するパートナー」として使う。それは新しい視点だった。

AIエージェントと人間の協働

その日の午後、湊は実際にAIとペアプログラミングを試してみることにした。山田の提案を受けて、GitHub　Copilotを使いながら、実際の機能開発を進めてみる。

湊はAIに「ユーザー登録機能を実装したい」と伝えた。AIは、認証チェック、バリデーション、データベースへの保存を含むコードを提案した。湊は、そのコードをレビューして、プロジェクトの規約に合わせて修正した。例えば、AIが提案したエラーハンドリングを、プロジェクトの標準的な方法に変更した。また、AIが提案した変数名を、プロジェクトの命名規則に合わせて修正した。

そのプロセスで、開発速度が30%向上した。AIがコードの骨組みを作り、人間が細部を調整する。その役割分担が、効率的だった。

「AIは競争相手じゃなく、協働パートナーなんですね」

湊は実感した。AIがコードを提案することで、開発の初期段階が速くなる。でも、そのコードをそのまま使うのではなく、人間がレビューして修正することで、品質も保てる。

翌日のチームミーティングで、湊はペアプログラミングの成果を共有した。

「AIとペアプログラミングを試してみたんですが、実装速度が大幅に向上しました。AIがコードの骨組みを作り、人間が細部を調整する。その役割分担が、効率的でした」

山田も同じように感じていた。

「AIがコードを提案して、人間がレビューして修正する。そのプロセスで、開発速度が上がるし、品質も保てる。これが理想的な使い方だと思う」

山田は続けた。

「でも、AIをうまく使うには、プロンプトエンジニアリングのスキルも必要だ。AIに何を伝えれば、良いコードが提案されるのか。それを学ぶ必要がある」

湊は深く頷いた。AIを戦略的に活用するには、AIの特性を理解し、適切な使い方を学ぶ必要がある。それは、新しいスキルだった。

「AIエージェントとの協働ワークで開発速度が大きく向上した。もっとAIを活用できないか。もしAIエージェントで開発量を確保できるなら、四半期というタイムリミットに対応できるかもしれない」

湊は考え込んだ。

AIエージェントという新しい開発リソース

成功体験を踏まえて、湊はもっとAIを活用できないかと考えていた。その週末、湊は技術ブログを読んでいた。ある記事が目に留まった。

「AIエージェントによる開発量や生産量の確保が可能になり、人を採用するより予算的にもリードタイム的にはスケールの柔軟性がつきやすい」

湊は興味を持った。記事を読み進めると、AIエージェントを活用することで、開発量を確保できるという内容だった。

「これからは人の採用だけでなく、AIエージェントを活用することで開発量を確保する時代になる」

記事には、人を採用する場合とAIエージェントを導入する場合の比較が書かれていた。

- **予算**　人を1人採用するコスト（年収+福利厚生）は年間800-1200万円。AIエージェントの月額料金は10-30万円程度。
- **リードタイム**　人の採用は3-6ヶ月かかる。AIエージェントの導入は1-2週間で完了。
- **教育コスト**　　人の教育には数ヶ月かかる。AIエージェントの設定は数日で完了。
- **スケールの柔軟性**　人の採用は固定費になる。AIエージェントは需要に応じてスケール可能。

湊は考え込んだ。もしAIエージェントで開発量を確保できるなら、四半期というタイムリミットに対応できるかもしれない。

チームでの検討

次の週のチームミーティングで、湊はAIエージェントの可能性を提案した。

「週末、AIエージェントに関する記事を読みました。開発量や生産量の確保が可能になり、人を採用するより予算的にもリードタイム的にはスケールの柔軟性がつきやすいという内容でした」

山田が興味深そうに湊を見た。

「AIエージェント…ですか。具体的にはどんなことができるんですか？」

「記事によると、ポストモーテムの作成、障害分析、コードレビュー支援など、様々なタスクを自動化できるようです」

飛鳥が前のめりになった。

「予算的にもリードタイム的にも柔軟性があるというのは、魅力的ですね。人を採用する場合と比較するとどうですか？」

湊は記事の内容を共有した。

「人を1人採用する場合、年収+福利厚生で年間800-1200万円かかります。でも、AIエージェントは月額10-30万円程度です。年間で120-360万円程度なので、大幅にコストを削減できます」

「リードタイムも、人の採用は3-6ヶ月かかりますが、AIエージェントは1-2週間で導入できます。四半期というタイムリミットには間に合いそうです」

山田が頷いた。

「教育コストも、人の教育には数ヶ月かかりますが、AIエージェントの設定は数日で完了する。これは大きなメリットだね」

湊が続けた。

「それに、今のような状況で急いで人を増やすのは逆効果になる可能性があります。前に読んだ『人月の神話』という本の中で遅れているプロジェクトに人員を追加すると、既存メンバーが新人の教育に時間を取られて、かえってプロジェクト

が遅くなると書いてありました。今のチームは既に疲労感が広がっているので、新人のオンボーディングに時間を割く余裕はないと思います」

山田が深く頷いた。

「確かに、ブルックスの法則だね。急いで人を増やしても、教育コストがかかるだけでなく、既存メンバーの生産性も下がる。AIエージェントなら、その心配がない」

「それから、スケールの柔軟性も重要です。人の採用は固定費になりますが、AIエージェントは需要に応じてスケール可能です。忙しい時期だけ増やすこともできます」

湊が続けた。

AIエージェントの特徴の理解

チーム全員でAIエージェントの特徴を整理した。

予算面でのメリット

- 人を1人採用するコスト（年収+福利厚生）vs AIエージェントの月額料金
- 年間で大幅なコスト削減が可能

リードタイム面でのメリット

- 人の採用（3-6ヶ月）vs AIエージェントの導入（1-2週間）
- 四半期というタイムリミットに対応可能

教育コスト面でのメリット

- 人の教育（数ヶ月）vs AIエージェントの設定（数日）
- すぐに戦力として活用できる

スケールの柔軟性

- 人の採用は固定費、AIエージェントは需要に応じてスケール可能
- 24時間稼働可能

「AIエージェントは、人の代替ではなく、人の補完として使うのが重要だと思います」

山田が補足した。

「確かに、AIエージェントにはできないこともあります。でも、単純な作業や繰り返し作業をAIエージェントに任せることで、人間はより創造的な仕事に集中できます」

最初のAIエージェントの採用

チームで議論した結果、まずはポストモーテム作成AIエージェントの導入を決定した。

「ポストモーテムの作成は、毎回時間がかかっていました。AIエージェントに任せることで、その時間を新機能開発に回せます」

湊が提案した。

「それから、ポストモーテムは比較的タスクが明確で、自動化しやすいです。最初のAIエージェントとしては適切だと思います」

山田が同意した。

翌週、湊はポストモーテム作成AIエージェントの選定を始めた。いくつかのサービスを比較し、試用期間を設けて効果を確認することにした。

2週間後、AIエージェントが生成したポストモーテムを確認した。品質は人間が作成したものと遜色なく、作成時間は大幅に短縮された。

「これは使えますね。ポストモーテム作成にかかっていた時間を、新機能開発に回せます」

湊は満足そうだった。

「でも、AIエージェントが生成した内容を人間がレビューして、必要に応じて修正する必要があります。完全に自動化するのではなく、AIと人間が協働することが重要です」

山田が注意点を指摘した。

湊は深く頷いた。AIエージェントは魔法の杖ではない。でも、適切に使えば、開発量を確保する有効な手段になるかもしれない。

解説：AIを「コードを早く書く道具」以上のパートナーに

ストーリーで描かれる「重圧」

なぜ「開発リソースは足りないのに増え続ける」が重圧になるのか。**何が起きているかが関係者で共有されていない**とき、その板挟みは構造的な課題としてのしかかります。湊のチームが直面したのも同じです。

- **経営陣・競合・営業・マーケからの圧力**　「開発スピードを上げろ」「四半期で10個の新機能」という要求。改善で品質は上がったが開発速度は以前と変わらない
- **人を増やせない制約**　予算制約、過去の採用失敗、オンボーディングに時間がかかり四半期のタイムリミットに間に合わない。現有リソースでどうにかしてほしいと言われる
- **「速く書けても後で直す時間が増えたら意味がない」**　コード生成ツールで書くスピードは上がるが、本質的な生産性向上につながるかという疑問
- **AIをコード生成だけに使う限界**　コードを早く書く道具として使うだけでは、技術的負債の増加や理解不足を招く可能性に気づき始めている

この重圧の背景には、**リソース不足と速度要求の板挟みのなかで、コード生成だけでは本質的な生産性向上にならない構造**があります。AIを戦略的パートナーとして捉え直す必要性に直面しているのです。

なぜコード生成だけでは不十分なのか

湊が気づいたように、AIツールを「コードを早く書く道具」として使うだけでは、本質的な生産性向上にはつながりません。

コード生成ツールを使うと、確かにコードを書くスピードは上がります。しかし以下の問題が発生する可能性があります。

- **品質の低下**　生成されたコードが無駄な処理を含んでいたり、命名が不適切だったりする
- **技術的負債の増加**　そのまま使うと、コードの複雑度が増し、将来的な開発速度が低下する
- **理解不足**　生成されたコードを理解せずに使うと、バグの原因がわからなくなる

AIの戦略的活用とは

AIを戦略的に活用するとは、以下のような使い方をすることです。

- **技術的負債の検出** AIにコードベースを分析させ、問題箇所を特定する
- **テストケースの生成**　AIに仕様書を読ませ、網羅的なテストケースを提案させる
- **ドキュメント作成支援** AIにコードを読ませ、ドキュメントを自動生成する
- **コードレビュー支援** AIにコードを分析させ、潜在的な問題を指摘させる

これらを組み合わせることで、開発全体の効率化が可能になります。

AIは「増幅器（Amplifier）」であり「鏡（Mirror）」である

2025年版DORAレポートが示すメッセージは、**「AIは増幅器（Amplifier）である」**という概念です。AIは組織の強みも弱みも増幅します。高パフォーマンスな組織では開発速度がさらに向上する一方で、課題を抱える組織では問題が深刻化します。

AIは組織の真の能力を映し出す「**鏡（Mirror）」として機能します。AI活用の成否はツール自体ではなく、その組織がすでに持つ仕組みや文化に依存します。**

詳細な手法については、章末のAIとの効果的協働ガイドのワークシートを参照してください。

> ***出典*** *DORA（DevOps Research and Assessment）レポート2025年版『State of AI-Assisted Software Development』、https://youtu.be/Dvo5Hhay-t0?list=TLGGO2hCbXy_mxozMTEyMjAyNQ*

AIエージェントという新しいリソース

湊のチームが検討したように、AIエージェントは開発量や生産量を確保する新しいリソースとして注目されています。

開発リソースの変遷

- **従来のアプローチ**　人の採用によるスケール（固定費がかかり、リードタイムが長い、教育コストが高い）
- **現在のアプローチ**　　AIエージェントによる開発量・生産量確保の時代（予算的・リードタイム的な柔軟性、スケールの柔軟性）

ブルックスの法則と急激な人員増加のリスク

フレデリック・ブルックスが「人月の神話」で提唱した「ブルックスの法則」は、「遅れているソフトウェアプロジェクトに人員を追加すると、さらに遅くなる」というものです。その理由は以下の通りです。

- **教育コスト** 新人の教育に既存メンバーの時間が奪われる
- **コミュニケーションコスト**　チームサイズが増えると、コミュニケーションの複雑度が指数関数的に増加する
- **コンテキスト共有の困難** プロジェクトの文脈を理解するのに時間がかかる

- **既存メンバーの負担増** 新人のサポートで既存メンバーの生産性が低下する

特に、既に疲労感が広がっているチームや、タイムリミットが迫っているプロジェクトでは、急激な人員増加は逆効果になる可能性が高いです。AIエージェントは、このようなリスクを回避しながら開発量を確保できる選択肢として注目されています。

AIエージェントの特徴

- **予算面**　人よりも低コスト（年間で大幅なコスト削減が可能）、月額料金制で固定費を抑えられる
- **リードタイム面** 人よりも短い導入期間（1-2週間）、四半期というタイムリミットに対応可能
- **教育コスト面**　人よりも低い（数日で設定完了）、すぐに戦力として活用できる
- **スケールの柔軟性**　需要に応じてスケール可能、24時間稼働可能、忙しい時期だけ増やすこともできる

AIエージェントの選定と導入

- 選定基準 タスクの明確性、自動化の可能性、ROI
- 導入プロセス POC（概念実証）から段階的導入、効果測定
- 人との協働の重要性　AIエージェントは人の代替ではなく、人の補完として使う

AIエージェントを適切に活用することで、開発量を確保し、四半期というタイムリミットに対応できる可能性があります。

> ***出典***　*フレデリック・P・ブルックス・Jr（著）『人月の神話【新装版】』、丸善出版、2014年4月22日*

5-2 AI活用の実践と組織的課題

AI活用方法のブレインストーミング

チームミーティングで、湊はAI活用について提案した。

「AIをもっと戦略的に使えないだろうか？コード生成以外の用途を考えてみませんか？」

5名のチーム全員が集まり、AI活用のアイデアを出し合った。

「技術的負債の検出はどう？AIにコードベースを分析させて、問題箇所を特定してもらう」

山田が最初に提案した。

「テストケースの自動生成もできるかもしれない。仕様書を読ませて、網羅的なテストケースを提案してもらう」

高橋が続けた。

「ドキュメント作成支援も。コードを読ませて、自動的にドキュメントを生成する」

佐藤が提案した。

「コードレビュー支援も。AIにコードを分析させて、潜在的な問題を指摘してもらう」

別のメンバーが加わった。

ホワイトボードには、AI活用のアイデアが並んでいた。

- 技術的負債の検出
- テストケース自動生成

- ドキュメント作成支援
- コードレビュー支援
- バグの予測
- パフォーマンス分析

湊はそれらを見つめながら言った。

「じゃあ、まずはテストケース自動生成から始めてみませんか？高橋さんと一緒に実験してみたいんです」

「面白い、実験してみよう」

山田が頷いた。

リーダーとして湊が実験の方向性を整理した。まずはテストケース自動生成から始め、効果が確認できたら他の用途にも展開する。段階的に進めることで、リスクを最小限に抑えられる。

テスト自動化でのAI活用

翌週、高橋と湊はテストケース生成の実験を始めた。

高橋は仕様書をAIに読み込ませた。ユーザー登録機能、ログイン機能、データ分析機能。それぞれの仕様をAIに理解させ、テストケースを提案してもらう。

AIは数分でテストケースを返してきた。

「正常系のテストケース：ユーザー名とパスワードを正しく入力した場合、ログインに成功する」

「異常系のテストケース：パスワードが間違っている場合、エラーメッセージが表示される」

「境界値テスト：ユーザー名が255文字を超える場合、エラーメッセージが表示される」

高橋は驚いた。

「これまで見落としていたエッジケースも含まれています。人間が考えると、正常系と基本的な異常系しか思いつかないことが多いんです。でも、AIは境界値テストやエッジケースも提案してくれる」

高橋は興奮しながら続けた。

「それから、テストケースの作成時間も大幅に短縮できました。以前は1つの機能につき、テストケースを作成するのに半日かかっていました。でも、AIを使えば、1時間で完了します」

「それは大きな効果ですね」

湊は感心した。

「でも、AIが提案したテストケースをそのまま使うのではなく、人間がレビューして、必要に応じて修正する必要があります。AIは完璧ではないから」

高橋が補足した。

「そうですね。AIと人間の協働が重要ですね」

自動テスト作成の効率も大幅に向上した。AIが提案したテストケースをベースに、人間がレビューして修正する。そのプロセスで、テストケースの品質も向上した。

高橋は満足そうに言った。

「AIって、QAの仕事を奪うんじゃなくて、強化してくれるんですね。これまで手動でやっていた作業を自動化できるようになって、もっと創造的な仕事に集中できるようになりました」

開発チームとQA部の連携も強化された。AIが生成したテストケースを共有することで、開発チームとQA部の認識のずれが減った。

翌週のチームミーティングで、高橋は成果を報告した。

「テスト自動化の実験が成功しました。開発チームとの連携も強化され、認識のずれが減りました」

湊は高橋の報告を聞きながら、小さな成功を実感していた。AI活用の第一歩が成功した。でも、これで終わりではない。もっと戦略的にAIを活用する必要がある。

技術的負債検出の実験と失敗

テスト自動化の成功を受けて、湊は次のステップを提案した。

「次は、技術的負債の検出を試してみませんか？山田さんと一緒に実験してみたいんです」

「面白い、実験してみよう」

山田が頷いた。

翌週、湊と山田は技術的負債検出の実験を始めた。

湊はAIツールにコードベースを分析させた。勤怠管理モジュール、経費精算モジュール、プロジェクト管理モジュール。それぞれのコードをAIに読み込ませ、問題箇所を特定してもらう。

数時間後、AIは分析結果を返してきた。

「この関数、複雑度が高すぎますね。循環的複雑度が15を超えています。リファクタリングを検討してください」

「ここ、テストカバレッジ不足です。この関数にはテストがありません」

「このコード、重複が3箇所で見つかりました。共通化を検討してください」

湊は結果を見て、少しがっかりした。

「思ったほど精度が高くないですね。表面的な指摘ばかりで、実際の問題を捉えきれていない気がします」

湊は山田に相談した。

「例えば、このモジュール間の依存関係について、AIは何も指摘していません。でも、実際には勤怠管理モジュールと経費精算モジュールの間で、データの整合性を保つための複雑な依存関係があるんです。それが問題の根本原因なのに、AIはそれを理解できていない」

山田は画面を見つめながら、深く考え込んだ。

「これは『コンテキストエンジニアリング』の領域だね」

「コンテキストエンジニアリング...ですか？」

「うん。テストケース自動生成のような単純なタスクは『prompt-engineering』で対応できる。でも、技術的負債検出やドキュメント自動生成、コードレビュー支援は『コンテキストエンジニアリング』の領域に入ってくる」

山田は続けた。

「既存システムを動かすコードベースやKnowledgeをAIリーダブルな状態にした上でAIを動かさないといけない。このチームは以前よりはよくなったとはいえ、ドメインモデリングができていなかったりテストが不十分なため、AIの理解が追いつかず、出力結果がいまいちな状態になっている」

「具体的には、どんな問題があるんですか？」

「まず、ドメインモデルの境界が曖昧だ。勤怠管理モジュールと経費精算モジュールの間の依存関係が、コードからだけでは理解できない。AIはコードの表面だけを見て判断しているから、実際の問題を捉えきれない」

「それから、テストが不十分だ。テストがないと、AIはコードの意図を推測できない。この関数が何をしているのか、なぜこの実装になったのか、AIにはわからない。だから、表面的な指摘しかできない」

「それから、システムの仕様書や設計に関する情報が、AIが読み取れる形式で管理されていない。AIが参照できる『教科書』のような仕組みが必要だ。docs as code化が必要だ」

湊は深く頷いた。

「なるほど。AIをうまく使うには、まずシステム自体をAIリーダブルな状態にする必要があるんですね」

「そうだ。コンテキストエンジニアリングは、AIに適切な文脈を提供する技術だ。コードだけではなく、ドメインモデル、テスト、仕様書、設計ドキュメント。これらをAIが理解できる形式で整備することが重要だ」

コンテキストエンジニアリングへの取り組み

湊とチームは、コンテキストエンジニアリングの重要性を理解した。改善活動を始めることにした。

まず、docs as code化を推進した。システムの仕様書や設計に関する情報を、AIがいつでも正確に読み取れる形式で一元管理することにした。Markdown形式でドキュメントを管理し、Gitでバージョン管理する。AIが参照できる「教科書」のような仕組みを構築した。

次に、ドメインモデリングの改善に取り組んだ。イベントストーミングを活用して、ドメインモデルの境界を明確化した。勤怠管理モジュールと経費精算モジュールの間の依存関係を、イベントストーミングで可視化し、ドメインモデルの境界を明確にした。

そして、テストの充実を進めた。テストカバレッジを向上させ、コードの意図を明確にした。各関数にテストを追加し、なぜその実装になったのか、どんな制約があるのかを、テストコードで表現した。

1ヶ月後、改善活動が一段落した。

改善後の技術的負債検出の実験

湊は、改善後のコードベースを再度AIツールに分析させた。

今度は、AIの分析結果が大きく変わった。

「このモジュール間の依存関係が不明確です。ドメインモデルの境界が曖昧で、影響範囲の把握が困難です。イベントストーミングでのモデリングを検討してください」

「この関数にはテストがありません。テストがないため、変更の影響範囲を把握できません。この関数は認証チェックとキャッシュ機能を実装していますが、テストがないため、これらの機能が正しく動作しているか確認できません」

「このモジュールのコード、特定のメンバーが頻繁に修正しています。他のメンバーが触っていないため、知識が属人化している可能性があります。コードレビューやペアプログラミングで知識を共有することを検討してください」

「使用しているライブラリのバージョンが2つ以上古くなっています。セキュリティパッチが適用されていない可能性があります。バージョンアップを検討してください」

湊は驚いた。AIが指摘した箇所は、確かに問題のあるコードだった。しかも、以前よりもはるかに的確な指摘になっている。

「改善後は、AIの指摘が的確になった。コンテキストエンジニアリングの重要性を実感した」

湊は興奮しながら山田に報告した。

「予想以上の精度で問題箇所を特定できました。特に、モジュール間の依存関係や、テストがないことによる影響範囲の把握の困難さなど、以前は指摘できなかった問題を、正確に指摘できるようになりました」

山田も画面を見つめながら頷いた。

「確かに、これは有用だね。コンテキストエンジニアリングを進めたことで、AIがシステムを正確に理解できるようになった。でも、AIが指摘した全てが問題とは限らない。人間の判断も必要だよ」

「そうですね。AIの指摘を参考にしつつ、人間が最終判断をする。それが重要ですね」

湊は結果をチームで共有した。全員が驚いた。

「AIって、こんなことができるんですね。しかも、システムを改善することで、AIの精度も上がるんですね」

佐藤が感心した様子で言った。

「そうだ。AIをうまく使うには、まずシステム自体をAIリーダブルな状態にする必要がある。それがコンテキストエンジニアリングだ」

山田が補足した。

失敗から学ぶ

すべてがうまくいくわけではない。AI活用にも失敗はあった。

ドキュメント自動生成の実験では、期待した結果が得られなかった。

湊は、コードベース全体をAIに読み込ませ、APIドキュメントを自動生成しようとした。AIは30分で100ページのドキュメントを生成した。でも、湊が読み返してみると、内容が薄い。

「内容が薄い。AIが生成したドキュメントは、表面的な説明しかしていない」

湊が作成したドキュメントを見て、山田が指摘した。

「例えば、この関数の説明。『ユーザー情報を取得する関数です』としか書いてない。でも、実際には、この関数は認証チェックも行っているし、キャッシュも使っている。そういう重要な情報が抜けている」

山田は画面を指差しながら続けた。

「コンテキストが不足している。AIはコードの意図を完全に理解できないから、詳細な説明ができない。コードだけを見ても、なぜその実装になったのか、どんな制約があるのかはわからない」

湊は深く頷いた。確かにAIが生成したドキュメントは、表面的な説明しかしていなかった。

コードレビュー支援の実験でも、問題があった。

湊は、AIにコードレビューを依頼した。AIは100行のコードに対して、50箇所の問題を指摘した。でも、佐藤が確認してみると、実際には問題があるのは10箇所だけだった。

「false　positiveが多い。AIが指摘した問題の半分以上は、実際には問題じゃない」

佐藤が報告した。

「例えば、AIが『この変数名は不適切です』と指摘したけど、実際にはプロジェクトの命名規則に従っている。`user_id`という変数名は、プロジェクトでは標準的な命名規則なんです。でも、AIは一般的な命名規則と比較して、不適切だと判断した」

佐藤は別の例も示した。

「それから、AIが『この関数は長すぎます』と指摘したけど、実際にはこの関数は複雑なビジネスロジックを実装していて、分割すると可読性が下がる。AIはコードの長さだけを見て判断しているけど、コンテキストを理解していない」

湊は深く考え込んだ。

「AIも万能じゃないんですね。コンテキストを理解できないから、誤検出が多い」

「でも、使い方次第で強力なパートナーになる」

山田が続けた。

「ドキュメント自動生成は、完全に自動化するのではなく、AIが下書きを作って、人間が修正する。AIが生成したドキュメントをベースに、人間がコンテキストを追加する。そうすれば、効率的にドキュメントを作成できる」

「コードレビュー支援も、AIが指摘した箇所を人間が確認する。AIが指摘した箇所を、人間がフィルタリングして、本当に問題がある箇所だけを修正する。そうすれば、効果的に使えるよ」

湊は失敗から学んだ。AIを完全に自動化するのではなく、AIと人間が協働する。AIが得意なこと（データ分析、パターン検出）をAIに任せ、人間が得意なこと（コンテキスト理解、意思決定）を人間が担う。それが重要だった。

個人の実装速度は上がったが、チームのアウトプットは伸びない

3週間の実験を経て、効果的な使い方が見えてきた。

湊は実験の結果をまとめた。テストケース生成は成功した。技術的負債検出は、初手は失敗したが、コンテキストエンジニアリングを進めることで成功した。でも、ドキュメント自動生成とコードレビュー支援は、期待した結果が得られなかった。

湊は、何が違ったのかを考えた。テストケース生成は、AIが得意な「パターン検出」に適していた。技術的負債検出は、コンテキストエンジニアリングを進めることで成功した。でも、ドキュメント自動生成とコードレビュー支援は、AIが苦手な「コンテキスト理解」が必要だった。

「AI+人間」のペアプログラミングが最も効果的だった。以前試したように、AIがコードを提案し、人間がレビューして修正する。そのプロセスで、開発速度が30%向上した。AIがコードの骨組みを作り、人間が細部を調整する。その役割分担が、効率的だった。

ボトルネックが移動し、レビュー待ちと設計待ちが増える

AI活用が進む中、チーム内で新たな課題が浮上していた。

湊は個人の実装速度が上がったことを実感していた。AIとペアプログラミングをすることで、以前は10時間かかっていた作業が4時間で完了するようになった。でも、チーム全体のアウトプットが伸びているかというと、そうでもない気がする。

湊は実装を早く終えても、コードレビュー待ちで次の作業に進めないことが増えていた。レビュー依頼を出しても、レビュー担当者が対応できるまでに時間がかかる。以前は実装に時間がかかっていたので、レビュー待ちの時間は相対的に短く感じていた。でも、実装速度が上がった今、レビュー待ちの時間が相対的に長く感じられるようになった。

チームミーティングで、湊は状況を共有した。

「AI活用で個人の実装速度は上がったんですが、チーム全体のアウトプットが思ったほど伸びていない気がします。実装を早く終えても、レビュー待ちで次の作業に進めないことが増えています」

山田が深く頷いた。

「湊さん、それは『ボトルネックの移動』という現象だ。個人の実装能力が2倍になっても、コードレビューや設計の際に必要な合意形成や意思決定のプロセスがそのままなら、組織全体の生産性は頭打ちになる。むしろ『待ち時間』の比率が相対的に増え、ストレスだけが増幅される」

湊は理解した。

「つまり、実装速度が上がっても、レビューや設計のプロセスが変わらなければ、チーム全体の生産性は上がらないということですね」

「その通りだ。AIは個人を強化するが、同時にチームの人数を実質的に増やす効果も持つ。つまり、コミュニケーションラインという新たなボトルネックを顕在化させる。レビュー依頼が増え、レビュー担当者の負荷が増大する。設計の合意形成にも時間がかかる。これが、個人の実装速度が上がってもチームのアウトプットが伸びない理由だ」

佐藤が湊に相談してきた。

「湊さん、AIの提案を逐一確認するのが大変で、かえって時間がかかってしまいます」

湊は山田に相談した。

「山田さん、AIの提案を逐一確認するのが大変だという声が出ています。これはどう対処すべきでしょうか？」

山田は深く頷いた。

「湊さん、それは『AI疲れ』と呼ばれる現象だ。AIから次々と出てくる提案を常に判断し、検証し続けることによる精神的な負荷が蓄積し、かえって生産性が低下してしまう現象です」

「では、どう対処すべきでしょうか？」

「AIの提案を全て確認するのではなく、重要な部分だけに絞るべきだ。AI提案の判断プロセスを最適化することが重要だ」

さらに、別の課題も浮上していた。

経験の浅いエンジニアが、AIが生成したコードの意味を理解しないまま提出し、レビューに余計な時間がかかるという問題だった。

佐藤が困った表情で湊に報告した。

「湊さん、AIが生成したコードをそのまま提出したら、レビューで『なぜこのコードにしたの？』と聞かれて答えられませんでした」

山田は説明した。

「AIが作ったコードへの説明責任は重要だ。経験の浅いエンジニアが、AIが生成したコードの意味や設計思想を十分に理解しないまま提出してしまうと、レビュー担当者からの質問に答えることができず、結果としてレビューに余計な時間がかかったり、本人の成長を妨げたりする危険性がある」

湊は状況を整理していた。個人の実装速度が上がったことで、レビュー依頼の数が増えている。以前は1日あたり2-3件だったレビュー依頼が、今は5-6件になっている。でも、レビュー担当者の数は変わらない。結果として、レビュー待ちの時間が長くなっている。

山田も同じ問題を感じていた。

「レビュー依頼が増えているのに、レビュー担当者の数は変わらない。これがコミュニケーションラインのボトルネックだ。AIが個人を強化する一方で、チームの人数を実質的に増やす効果を持つ。つまり、コミュニケーションコストが増大し、意思決定プロセスがボトルネックになる」

湊は理解した。

「設計の合意形成でも同じことが起きています。実装が早く進むようになったので、設計レビューの頻度が増えています。でも、合意形成のプロセスは変わっていないので、意思決定に時間がかかっています」

湊は理解した。

「AIを使うときは、生成されたコードの意味を理解してから提出する必要があるということですね」

山田はさらに重要な指摘をした。

「湊さん、5-1で説明したように、AIは組織の文化や実力を映し出す『組織の鏡』であり、良いチームでは生産性をさらに加速させる一方で、課題を抱えるチームでは混乱を広げてしまう『増幅器』として機能します」

湊は深く頷いた。

「つまり、AIを使いこなすためには、ツールの操作技術だけでは不十分で、エンジニア自身のスキルがより重要になるということですね」

「その通りだ。AIの提案の良し悪しを判断し、その意図を理解し、自らの言葉で説明できるだけの深い知識と経験が、これまで以上に求められる」

AIエージェントによって、エンジニアは基礎に立ち返る

湊は深く考え込んだ。AIエージェントを使う中で、すべてが一周回ってエンジニアの基礎が必要なことに気づいていた。

設計がよくないと、AIは間違った品質のコードを吐いてくる。ドメインモデルが曖昧なままAIにコード生成を依頼すると、AIはその曖昧さを増幅させたコードを生成する。逆に、設計が明確で、ドメインモデルが整理されていると、AIは適切なコードを生成する。

それから、人が設計をわかっていないと、AIが生成したコードをレビューすることもできない。AIが生成したコードが設計思想に沿っているか、ドメインモデルの境界を守っているか、それを判断するには、レビュー担当者自身が設計を理解している必要がある。

湊は山田に相談した。

「山田さん、AIエージェントを使うと、すべてが一周回ってエンジニアの基礎が必要なことがわかります。設計がよくないとAIは間違った品質のコードを吐いてくるし、人が設計をわかっていないとそれをレビューすることもできない。スピードは早くなっても、逆にどんどん負債が溜まっていく」

山田が深く頷いた。

「その通りだ。AIエージェントは、エンジニアの基礎スキルを増幅させる。基礎がしっかりしているエンジニアは、AIを効果的に使いこなせる。でも、基礎が不十分なエンジニアは、AIを使っても負債を増やすだけだ。つまり、チームにと

って育成が大事なんだ」

湊は理解した。

「AIエージェントを導入するだけでは不十分で、エンジニアの基礎スキルを育成することが重要だということですね」

「そうだ。設計の基礎、ドメインモデリングの基礎、コードレビューの基礎。これらをチーム全体で育成することが、AI時代の開発生産性向上の鍵になる」

湊は深く考え込んだ。AIツールを導入しただけでは、根本的な問題は解決しないのではないか。個人の実装速度は上がったが、チーム全体のアウトプットが伸びない。それは、コードレビューや設計の合意形成・意思決定のプロセスが変わっていないからだ。

「山田さん、ツールの導入だけでは構造は変わらないということですね。個人の実装速度が上がっても、コードレビューや設計のプロセスがそのままなら、組織全体の生産性は頭打ちになる」

山田が深く頷いた。

「その通りだ。生産性を最大化したいなら、意思決定の単位を小さくし、権限を分散させる組織設計が不可欠だ。ツールの導入だけでは、構造は変わらない。コードレビューのプロセスを改善し、要件定義の合意形成プロセスを改善する必要がある」

湊は理解した。

「つまり、AIツールを導入するだけでなく、組織の意思決定プロセスそのものを見直す必要があるということですね」

「そうだ。コードレビューでは、自動承認基準を設定して、レビュー時間を短縮する。設計では、権限を委譲して、意思決定を迅速化する。そうすることで、ボトルネックを解消できる」

ツールの導入だけでは構造は変わらない

湊はチーム全体でAIスキルの標準化・育成に取り組み始めた。

一部の詳しい人だけでなく、チーム全員がAIを効果的に使いこなせるよう、標準化と育成を実施した。AI活用のベストプラクティスを共有し、組織全体に浸透させた。

チームミーティングで、湊はAI活用のガイドラインを提案した。

「AIの提案を全て確認するのではなく、重要な部分だけに絞る。AIが生成したコードは、必ず意味を理解してから提出する。AIに依存しすぎず、自分で考える力を維持する」

山田が補足した。

「AIは補助ツールであり、代替ではない。AIが提案したコードをそのまま使うのではなく、人間がレビューして修正する。それが重要だ」

湊は続けて、組織設計の改善について提案した。

「それから、ボトルネックを解消するために、コードレビューと設計のプロセスを改善したいと思います。まず、コードレビューでは、自動承認基準を設定します。例えば、テストカバレッジが80%以上で、Lintエラーがなく、既存のパターンに従っている場合は、自動承認とします。これにより、レビュー時間を短縮できます」

山田が頷いた。

「良い提案だ。レビュー担当者の負荷を減らし、レビュー待ちの時間を短縮できる。それから、設計の合意形成プロセスも改善する必要がある」

湊が続けた。

「設計では、権限を委譲します。小さな設計変更については、エンジニアとテックリードで合意形成し、部長の承認は不要とします。大きな設計変更のみ、部長の承認を必要とします。これにより、意思決定を迅速化できます」

山田が頷いた。

「確かに、小さな設計変更まで部長の承認を取る必要はないですね。エンジニアとテックリードで判断できる範囲を明確にすることで、意思決定が迅速化されます」

佐藤も頷いた。

「AI疲れを感じていたけど、ガイドラインが明確になれば、もっと効果的に使えるようになりそうです。それに、コードレビューと設計のプロセスが改善されれば、待ち時間も減りそうです」

その結果、AI活用の浸透率が大幅に向上した。チーム全体でAIを効果的に使いこなせるようになり、開発速度の向上につながった。さらに、コードレビューと設計のプロセス改善により、ボトルネックが解消され、チーム全体のアウトプットも向上した。

解説：AI活用の実践と組織的課題

ストーリーで描かれる「重圧」

本質より対症療法が続いているとき、AI活用の「実践と組織的課題」は重くのしかかります。湊のチームが経験した重圧も、この構造から生じています。

- **テスト自動化・技術的負債検出でのAI活用と失敗**　ドキュメント自動生成やコードレビュー支援では期待した効果が出ず、失敗から学ぶ
- **AI疲れ**　AIから次々と出てくる提案を逐一判断・検証し続けることによる精神的な負荷。AIが作ったコードへの説明責任、レビュー時間の増加
- **ボトルネックの移動**　個人の実装速度は上がっても、チーム全体のアウトプッ

トが伸びない。コードレビューや設計の合意形成プロセスがボトルネックになっている

- **「AI+人間」のペアプログラミングが最も効果的**　という発見。組織の鏡としてのAI、設計がよくないとAIは間違ったコードを吐くこと、育成の重要性への気づき

この重圧の背景には、**AI導入で個人が速くなってもボトルネックが移動し、組織設計や協働パターンを見直さないと成果が出ない構造**があります。実践と失敗を経て、効果的な協働と組織的課題への対応段階にあるのです。

AIと人間の役割分担

湊と山田が発見したように、AIと人間が協働することで、より効果的な開発が可能になります。

AIが得意なこと（データ分析、パターン検出、網羅的チェック）をAIに任せ、人間が得意なこと（コンテキスト理解、創意工夫、意思決定）を人間が担う形にすると、開発全体の効率化につながります。

効果的な協働パターン

- **AI+人間のペアプログラミング**　AIがコードを提案し、人間がレビューして修正する。開発速度が30%向上
- **AIによる技術的負債検出 + 人間の判断** AIが問題箇所を特定し、人間が優先順位をつけて対応する
- **AIによるテストケース生成 + 人間のレビュー** AIがテストケースを提案し、人間がレビューして修正する

コンテキストエンジニアリングの重要性

技術的負債検出のような「コンテキストエンジニアリング」の領域では、AIをうまく使うために、まずシステム自体をAIリーダブルな状態にする必要があります。docs　as　code化、ドメインモデリングの明確化、テストの充実を押さえま

す。

ボトルネックの移動

AIで個人の実装速度が上がっても、チームのアウトプットが伸びない理由は明快です。ボトルネックが移動するからです。

個人の実装能力が2倍になっても、コードレビューや設計の際に必要な合意形成や意思決定のプロセスがそのままなら、組織全体の生産性は頭打ちになります。むしろ「待ち時間」の比率が相対的に増え、ストレスだけが増幅されます。

AIは個人を強化しますが、同時にチームの人数を実質的に増やす効果も持ちます。つまり、コミュニケーションラインという新たなボトルネックを顕在化させます。レビュー依頼が増え、レビュー担当者の負荷が増大します。要件定義の合意形成にも時間がかかります。

組織設計の重要性

生産性を最大化したいなら、意思決定の単位を小さくし、権限を分散させる組織設計が不可欠です。ツールの導入だけでは、構造は変わりません。

コードレビューのプロセス改善

- 自動承認基準の設定（テストカバレッジ、Lintエラー、既存パターンへの準拠など）
- レビュー時間の短縮
- レビュー担当者の負荷軽減

設計の合意形成プロセスの改善

- 権限の委譲（小さな設計変更はエンジニアとテックリードで合意、大きな設計変更のみ部長の承認）
- 意思決定の迅速化

- ボトルネックの解消

組織的課題への対応

AI導入により、AI疲れや説明責任の問題などの組織的課題が発生します。AIの提案を全て確認するのではなく重要な部分だけに絞る、AIが生成したコードの意味を理解してから提出する、AIスキルの標準化・育成を実施するなどの対策を講じます。

詳細な手法については、章末のAIとの効果的協働ガイドのワークシートを参照してください。

5-3 開発生産性ツリーによる戦略転換

作業時間が短縮されただけでは、真の生産性向上にはならない

これまでの取り組みで、湊とチームはコンテキストエンジニアリングまで進んでいた。AIがシステムを正確に理解できるようになり、技術的負債検出やテスト自動化などが可能になっていた。

しかしAI活用が進む中、湊はある疑問を抱くようになっていた。

確かにAIエージェントによってエンジニアの作業は楽になっている。でも、本当に生産性が上がっているのだろうか。

湊は具体的な例を思い浮かべた。以前は1つの作業に10時間かかっていたものが、AIの助けを借りて4時間でできるようになった。作業時間は確かに短縮された。でも、残りの6時間をどう過ごすかは、メンバーごとに違う。

あるメンバーは、その時間を使って新しい技術を学んでいる。別のメンバーは、より創造的な仕事に取り組んでいる。でも、中にはその時間を無駄に過ごしてしまっているメンバーもいるかもしれない。

「AIで作業時間が短縮されただけでは、真の生産性向上にはならない。残りの時間をどう使うか、そういったことまで設計するように、きちんとした開発プロセスの再設計やチームのあり方を考えることが必要だと思う」

湊は深く考え込んだ。AIツールを導入するだけでは不十分だ。プロセスそのものを見直し、チーム全体が効果的に働ける仕組みを作る必要がある。

工数分析の過程

その夜、湊は一人オフィスに残っていた。以前に学んだ手法を活用して、過去3ヶ月分の工数データを整理していた。

湊は運用業務の工数分析を進めていた。スプレッドシートにデータを入力しながら、湊はパターンを探していた。AIエージェントをどこに導入すれば効果的なのか。そのためには、まず現状の業務を整理する必要がある。

湊はスプレッドシートを見つめながら考え込んだ。

人件費がある中で、Capacityとして毎月使える人月は固定化されている。その中で「新しい価値に使える工数」と「現在のシステムを運用する工数」で分けて考えることができるのではないか。

湊は開発区分を整理してみた。

「新しい価値に使える工数」

- 新規開発：新しい価値（機能やサービス）を0から作る
- エンハンス開発：既存システムの拡張・変更

「現在のシステムを運用する工数」

- 保守開発：資産価値を耐久年数を維持する・伸ばす（振る舞いを大きく変えずに内部品質を整え、現状維持を行うリファクタリング・バージョンアップ等）
- 運用：障害対応、問い合わせ対応、ルーティンワーク等
- 管理業務、その他：会議、採用等の管理業務

「これが開発生産性ツリーの構造なのかもしれない」

湊は深く考え込んだ。この構造を理解することで、AI戦略を整理できるのではないか。

開発生産性ツリーの発見

翌週のチームミーティングで、湊は開発生産性ツリーの概念を提案した。

「これまでのAI活用の成果を踏まえて、開発生産性ツリーの観点からAI戦略を整理してみませんか？」

湊はホワイトボードに開発区分の構造を描きながら説明を始めた。

「人件費がある中で、Capacityとして毎月使える人月は固定化されています。その中で『新しい価値に使える工数』と『現在のシステムを運用する工数』で分けて考えることができます」

湊は開発区分を詳しく説明し始めた。

「まず、『新しい価値に使える工数』から説明します。これは2つに分けられます」

湊はホワイトボードに書きながら続けた。

「1つ目は新規開発です。これは新しい価値、つまり機能やサービスを0から作る作業です。2つ目はエンハンス開発で、既存システムの拡張・変更を行います」

湊はホワイトボードの別の領域を指しながら続けた。

「次に、『現在のシステムを運用する工数』です。これは3つに分けられます」

「1つ目は保守開発です。資産価値を耐久年数を維持する、あるいは伸ばす作業で、振る舞いを大きく変えずに内部品質を整え、現状維持を行うリファクタリングやバージョンアップなどが該当します」

「2つ目は運用です。障害対応、問い合わせ対応、ルーティンワークなどが含まれます」

「3つ目は管理業務、その他です。会議、採用などの管理業務が該当します」

湊は説明を終えると、少し間を置いてから続けた。

「本来は『新しい価値に使える工数』に沢山使うべきですが、今は障害対応やバグ対応など『現在のシステムを運用する工数』に時間を使っているのではないでしょうか」

湊は少し考え込むようにしてから続けた。

「PMや部長から見ると、エンジニアはみんなずっと開発をしているリソースに見えるかもしれません。しかし、実は運用工数が大きく占めているという誤解も、この区分で数値化することで理解が進むのではないかと思います」

山田が深く頷いた。

「なるほど。各区分を詳しく説明してもらって、構造がよく理解できました。これなら、どこに時間が使われているか一目瞭然だね。開発生産性ツリーという名前がぴったりだ」

飛鳥も興味深そうに見ていた。

「確かに、この構造で整理すると、現状の問題も明確になりますね。AI戦略も考えやすくなりそうです」

湊は続けた。

「この開発生産性ツリーの観点から見ると、AI戦略は2つに分けられます」

チーム内での戦略整理

翌週のチームミーティングで、湊は開発生産性ツリーの概念を提案した。

「これまでのAI活用の成果を踏まえて、開発生産性ツリーの観点からAI戦略を整理してみませんか？」

湊はホワイトボードに開発区分の構造を描きながら説明を始めた。

「人件費がある中で、Capacityとして毎月使える人月は固定化されています。その中で『新しい価値に使える工数』と『現在のシステムを運用する工数』で分けて考えることができます」

湊は開発区分を詳しく説明し始めた。

「まず、『新しい価値に使える工数』から説明します。これは2つに分けられます」

湊はホワイトボードに書きながら続けた。

「1つ目は新規開発です。これは新しい価値、つまり機能やサービスを0から作る作業です。2つ目はエンハンス開発で、既存システムの拡張・変更を行います」

湊はホワイトボードの別の領域を指しながら続けた。

「次に、『現在のシステムを運用する工数』です。これは3つに分けられます」

「1つ目は保守開発です。資産価値を耐久年数を維持する、あるいは伸ばす作業で、振る舞いを大きく変えずに内部品質を整え、現状維持を行うリファクタリングやバージョンアップなどが該当します」

「2つ目は運用です。障害対応、問い合わせ対応、ルーティンワークなどが含まれます」

「3つ目は管理業務、その他です。会議、採用などの管理業務が該当します」

湊は説明を終えると、少し間を置いてから続けた。

「本来は『新しい価値に使える工数』に沢山使うべきですが、今は障害対応やバグ対応など『現在のシステムを運用する工数』に時間を使っているのではないでしょうか」

湊は少し考え込むようにしてから続けた。

「PMや部長から見ると、エンジニアはみんなずっと開発をしているリソースに見えるかもしれません。しかし、実は運用工数が大きく占めているという誤解も、この区分で数値化することで理解が進むのではないかと思います」

山田が深く頷いた。

「なるほど。各区分を詳しく説明してもらって、構造がよく理解できました。これなら、どこに時間が使われているか一目瞭然だね。開発生産性ツリーという名前がぴったりだ」

飛鳥も興味深そうに見ていた。

「確かに、この構造で整理すると、現状の問題も明確になりますね。AI戦略も考えやすくなりそうです」

湊は続けた。

「この開発生産性ツリーの観点から見ると、AI戦略は2つに分けられます」

チームミーティングでの戦略整理

湊はホワイトボードにAI戦略の2つの軸を描きながら説明を続けた。

戦略1: ValueStreamの高速化（Incrementを1.5倍にする）

湊はホワイトボードに「戦略1」と書きながら説明した。

「まず、ValueStreamの各工程をAIネイティブなプロセスで高速化することで、同じ投入工数でより多くの成果物を生み出す戦略です」

「問題を定義するところ」

湊は「問題を定義するところ」と書きながら続けた。

「要求定義をもとに要件定義を詰める部分で、AIを使って意思決定を加速しています。人がPilot、AIがCopilotという役割分担です」

飛鳥が頷いた。

「確かに、要件定義の時間が短縮されて、意思決定が早くなりましたね。AIが情報を整理してくれるので、私たちは意思決定に集中できます」

「問題を解くところ」

湊は「問題を解くところ」と書きながら続けた。

「実装からリリースまでは、作業はAIが主導し、品質は人が担保します。人がCopilot、AIがPilotという役割分担です」

山田が補足した。

「AIがタスクを分解し実装まで進めてくれるので、私たちは品質チェックに集中できます。これにより、実装からリリースまでの工程が大幅に高速化されました」

湊はまとめた。

「ValueStream全体のリードタイム短縮により、Increment（出口）を1.5倍にすることができます」

戦略2: Ops工数のAI置き換え（Capacityを1.5倍にする）

湊は「戦略2」と書きながら説明を続けた。

「次に、Ops工数をAIに置き換えることで、新規開発に回せる工数を増やす戦略です」

工数分析のプロセス

湊は工数分析の結果を示した。

「開発区分における保守開発・運用・管理業務の工数を分析しました。全体の約40%を占めていました」

山田が補足した。

「その中で、AIによる改善インパクトがある部分をプロセスから探しました。例えば、ポストモーテムの作成や障害分析、目標設定の管理などです」

AI Agentの導入と効果

湊は具体的なAI Agentの導入例を説明した。

「まず、ポストモーテムを書くAI　Agentを導入しました。障害対応後のポストモーテム作成が自動化され、工数が大幅に削減されました」

山田が続けた。

「次に、障害分析AIを導入しました。過去の障害パターンを分析し、事前に対策を提案してくれるAIです。これにより、障害対応の工数が削減されました」

湊は管理業務でのAI活用も説明した。

「メンバーの目標設定から評価までをAIがサポートし、管理業務の工数が削減されました。これまで手動で行っていた作業が自動化され、余剰工数が生まれました」

湊はまとめた。

「戦略1によりIncrement（出口）を1.5倍にし、戦略2によりCapacity（入口）を1.5倍にすることで、全体として開発生産性を150%に向上させることができます」

運用業務のAI化

チームミーティングの後、湊は戦略2の具体化に取り組み始めた。

「戦略2を具体化するために、運用業務の効率化を進めましょう。単一プロセスの効率化ではなく、全体最適を考えてAIネイティブなプロセスを設計する必要があります」

山田が補足した。

「その通りだ。既存のやり方を変えずに、無理やりAIツールを当てはめようとすると、かえって手間が増えてしまう。プロセスをAI前提に作り変える必要がある」

翌週、湊は運用業務の工数分析を進めた。過去3ヶ月分のデータを整理し、AIによる改善インパクトがある部分を特定した。

「開発区分における保守開発・運用・管理業務の工数を分析しました。全体の約40%を占めていました。その中で、AIによる改善インパクトがある部分をプロセスから探しました。例えば、ポストモーテムの作成や障害分析、目標設定の管理などです」

チームミーティングで、湊は具体的なAI Agentの導入例を説明した。

「まず、ポストモーテムを書くAI　Agentを導入しました。障害対応後のポストモーテム作成が自動化され、工数が大幅に削減されました」

山田が続けた。

「次に、障害分析AIを導入しました。過去の障害パターンを分析し、事前に対策を提案してくれるAIです。これにより、障害対応の工数が削減されました」

湊は管理業務でのAI活用も説明した。

「メンバーの目標設定から評価までをAIがサポートし、管理業務の工数が削減されました。これまで手動で行っていた作業が自動化され、余剰工数が生まれました」

1ヶ月後、運用業務の効率化により、余剰工数が新規開発に回せるようになった。

「Ops工数の削減により、余剰工数が新規開発に回せるようになりました。開発チーム全体が新規開発に集中できる環境が整いました」

湊はチームメンバーに語りかけた。

山田が頷いた。

「運用業務の効率化により、新規開発に集中できるようになりました。AIを戦略的に活用することで、開発生産性を向上させることができました」

開発生産性ツリーの観点からの整理

湊は開発生産性ツリーの図を指しながら説明を続けた。

「開発生産性ツリーの観点から見ると、予算（財務）起点で考えると、AIへの投資が開発生産性の向上にどうつながるかが明確になります」

山田が頷いた。

「なるほど。AIへの投資が、Capacity（投入工数）とIncrement（成果物）の両方を改善することで、開発生産性を150%に向上させるということですね」

湊は続けた。

「はい。戦略1は、同じ投入工数でより多くの成果物を生み出すことで、Increment（出口）を1.5倍にします。戦略2は、Ops工数を削減して新規開発に回す工数を増やすことで、Capacity（入口）を1.5倍にします」

チームミーティングの後、5名チーム全員がAI戦略の方向性を理解していた。

湊はチームメンバーに語りかけた。

「開発生産性ツリーの観点から整理することで、AI戦略の効果を明確に説明できるようになりました。経営層にも理解してもらいやすい形で説明できると思います」

解説：開発生産性ツリーによる戦略転換

ストーリーで描かれる「重圧」

戦略や評価の枠組みを組み替える段階で現れやすいのが、「単一プロセスの効率化の限界」という重圧です。湊のチームが経験したのも、ここからです。

- **単一プロセスの効率化だけでは真の生産性向上にならない**　という疑問。AIで作業時間が短縮されただけでは不十分だという気づき
- **湊が一人で工数分析を行い、開発生産性ツリーの観点を発見**　新しい価値に使える工数と運用する工数の区別、CapacityとIncrementの関係
- **戦略1（ValueStreamの高速化でIncrementを1.5倍）と戦略2（Ops工数のAI置き換えでCapacityを1.5倍）の整理**　チームで共有し、経営層に戦略の効果を説明しやすくなる手応え
- **開発生産性ツリーの観点からAI戦略の効果を明確に説明できる** という実感

この重圧の背景には、**局所最適から全体最適へ、開発生産性ツリーでAI戦略を整理しないと効果が説明できない構造**があります。戦略転換を言語化し、説明責任を果たす段階にあるのです。

開発生産性ツリーの観点からのAI戦略

湊のチームが実現したように、AIを戦略的に活用するには、開発生産性ツリーの観点から整理すると、戦略が明確になります。

開発生産性ツリーの観点から、AIによる開発生産性向上の戦略を2つに分けられます。

目的 Capacity（投入工数）に対して、Increment（成果物）の量を現状100%から150%にする

戦略1: ValueStreamの高速化（Incrementを1.5倍にする）

- 「問題を定義するところ」: 人がPilot、AIがCopilot（意思決定を加速）
- 「問題を解くところ」： 人がCopilot、AIがPilot（実装からリリースまでを高速化）
- 効果: ValueStream全体のリードタイム短縮により、Increment（出口）を1.5倍にする

戦略2: Ops工数のAI置き換え（Capacityを1.5倍にする）

- 工数分析から始め、AIによる改善インパクトがある部分をプロセスから探す
- 保守開発、運用、管理業務におけるAI Agentの導入
- 効果: Ops工数の削減により、余剰工数を新規開発に回し、Capacity（入口）を1.5倍にする

両戦略の組み合わせ 開発生産性を150%に向上させる

詳細な手法については章末のAIとの効果的協働ガイドのワークシートを参照してください。

5-4 成果と組織への影響

個とチームの生産性向上、両立への道のり

AI活用を始めてから半年が経過した。湊のチームは着実に成果を上げていた。

チームミーティングで、湊は成果をまとめた。

「これまでの取り組みの成果をまとめました。開発速度が30%向上し、バグ検出率は40%向上しています。技術的負債の返済も進み、チーム全体の疲労度も軽減されました」

湊はデータを見せながら続けた。

「AIエージェントの導入により、ポストモーテム作成の時間が80%削減されました。障害分析AIにより、障害対応の工数が30%削減されました。管理業務のAI化により、管理業務の工数が50%削減されました」

山田が補足した。

「開発生産性ツリーの観点から見ると、戦略1によりIncrement（出口）を1.5倍にし、戦略2によりCapacity（入口）を1.5倍にすることで、全体として開発生産性を150%に向上させることができました」

湊は続けた。

「それから、AI活用により、チーム全体のスキルも向上しました。AIを効果的に使いこなせるようになり、開発速度の向上につながっています」

組織への影響

AI活用の成果は、開発チームだけでなく、組織全体に影響を与えていた。

湊は他部署へのヒアリングを始めた。

まずは営業部を訪れた。

「開発チームの頑張りで、顧客満足度が上がってます。バグが減ったことで、お客様からのクレームも減りました」

営業部の担当者は満足そうに言った。

「具体的には、どのくらい改善したんですか？」

「月に10件あったクレームが、今は3件程度になりました。お客様との関係も良くなってます」

次にサポート部を訪れた。

「バグ報告が30%減って、すごく助かってる。以前は毎日のように緊急対応に追われてたけど、今は余裕を持って対応できるようになりました」

サポート部の担当者は感謝の言葉を述べた。

「開発チームの改善が、サポート業務にも良い影響を与えてるんですね」

「はい。バグが減ることで、サポート対応の時間も削減できました。その分、お客様への提案やフォローアップに時間を使えるようになりました」

マーケティング部も訪れた。

「新機能のリリースペースも安定してきた。以前は予定通りにリリースできなかったけど、今は計画通りに進んでいます」

マーケティング部の担当者は、リリース計画の安定性を評価していた。

田中部長との対話

湊は他部署へのヒアリングを終えて、田中部長に報告した。

「部長、AI活用の成果をまとめました。開発速度が30%向上し、バグ検出率は40%向上しています。他部署からの評価も良好です」

田中部長は資料を見つめながら、時々頷いた。

「これは素晴らしい成果だね。経営層も注目しているよ。もしかしたら、来月の経営会議で成果発表の機会が設けられるかもしれない。その時は、湊君に発表してもらいたいと思っている」

湊は深く頷いた。開発生産性ツリーの観点から、AI戦略の効果を明確に説明できる。それに、他部署からの評価も得られている。経営層にも理解してもらえるはずだ。

解説：成果と組織への影響

ストーリーで描かれる「重圧」

前のシーンで湊のチームが経験した「AI活用の成果を組織に示す」という壁は、**成果の可視化と組織への波及の段階**における構造的な課題から生じています。

- **開発速度30%向上、バグ検出率40%向上**　技術的負債返済の進行、チーム疲労度の軽減という定量的な成果
- **営業・サポート・マーケなど他部署からの評価**　顧客満足度の向上、バグ報告の削減、リリース計画の安定化。開発チームの改善が組織全体に影響している実感
- **田中部長から経営会議での成果発表の打診**　　経営層も注目しているという言葉。次のステップへつなぐ可能性
- **「測れない価値」が「測れる価値」になった** という実感の共有

この重圧の背景には、**AI活用の成果が数値と他部署評価で可視化され、組織全体への影響が認められ始めている構造**があります。次のステップ（経営層への説明）へつなぐ段階にあるのです。

段階的な成果

湊のチームが実現したように、AI活用の成果は開発チームだけでなく、組織全体に影響を与えます。

AI活用により、以下の成果が得られます。

- **開発速度の向上** AI+人間のペアプログラミングにより、開発速度が30%向上

- **バグ検出率の向上** AIによるテストケース生成により、バグ検出率が40%向上
- **技術的負債の返済** AIによる技術的負債検出により、技術的負債の返済が進む
- **チーム全体の疲労度軽減**　運用業務のAI化により、チーム全体の疲労度が軽減される

組織への影響

AI活用の成果は、開発チームだけでなく、組織全体に影響を与えます。

- **営業部** 顧客満足度の向上、クレームの削減
- **サポート部** バグ報告の削減、緊急対応の削減
- **マーケティング部** リリース計画の安定化

開発生産性ツリーの観点からの成果

開発生産性ツリーの観点から見ると、AI戦略の効果を明確に説明できます。

- **戦略1** Increment（出口）を1.5倍にする
- **戦略2** Capacity（入口）を1.5倍にする
- **両戦略の組み合わせ** 開発生産性を150%に向上させる

これらの成果を経営層に説明することで、AIへの投資が開発生産性の向上にどうつながるかを明確に示すことができます。

手法5 AIとの効果的協働ガイド

AI活用の基本原則

AIを効果的に活用するためには、以下の原則を押さえます。

1. AIは補助ツールであり、代替ではない

- AIが提案したコードをそのまま使うのではなく、人間がレビューして修正する
- AIが指摘した問題をそのまま信じるのではなく、人間が確認する
- AIの判断を盲信せず、人間の判断を優先する

2. AIと人間の役割分担を明確にする

AIが得意なこと

- 大量のデータ分析
- パターン検出
- 網羅的なチェック

人間が得意なこと

- コンテキスト理解
- 創意工夫
- 意思決定

3. 段階的に導入する

- 小さな実験から始める
- 効果が確認できたら、段階的に拡大する
- 失敗から学び、改善を続ける

AI活用の具体的な手法

技術的負債検出でのAI活用

手順

1. AIツールにコードベースを分析させる
2. AIが指摘した問題箇所を人間が確認する
3. 優先順位をつけて、段階的に改善する

効果

- 網羅的な分析が可能
- 人間では見落としがちな問題を発見
- 時間の短縮

テストケース生成でのAI活用

手順

1. AIに仕様書を読み込ませる
2. AIがテストケースを提案する
3. 人間がレビューして、必要に応じて修正する

効果

- 網羅的なテストケースの生成
- テストケース作成時間の短縮
- エッジケースの発見

コードレビュー支援でのAI活用

手順

1. AIにコードを分析させる
2. AIが潜在的な問題を指摘する

3. 人間が確認して、実際の問題かどうかを判断する

効果

- レビュー時間の短縮
- 見落としがちな問題の発見
- コード品質の向上

AI活用のチェックリスト

導入前

- [ ] AI活用の目的を明確にした
- [ ] チーム全体でAI活用の方針を合意した
- [ ] 小さな実験から始める計画を立てた

導入中

- [ ] AIが提案したコードを人間がレビューしている
- [ ] AIの判断を盲信せず、人間の判断を優先している
- [ ] 失敗から学び、改善を続けている

導入後

- [ ] AI活用の効果を測定している
- [ ] チーム全体でAI活用のスキルを向上させている
- [ ] 継続的に改善を続けている

AIケーパビリティチェックリスト

AI時代に成果を出し続ける組織が持つ7つのケーパビリティを、あなたの組織で確認してください。

1. 明確に周知されたAIスタンス

- [ ] 経営層からAI活用を支持する明確なメッセージが出ている
- [ ] 従業員が業務でAIを利用することを積極的に支援する方針がある
- [ ] AI利用に関するガイドラインやポリシーが整備されている

2. 健全なデータエコシステム

- [ ] 社内データがサイロ化されず、統合されている
- [ ] データへのアクセスが容易で、高品質なデータ基盤がある
- [ ] データの民主化が推進されている

3. 小さなバッチでの作業

- [ ] 変更を小さく分割してリリースしている
- [ ] 迅速なフィードバックサイクルが回っている
- [ ] CI/CDパイプラインが整備されている

4. ユーザー中心の重点

- [ ] ユーザーの課題解決を目的とした開発が行われている
- [ ] 技術ありきではなく、ユーザー価値を起点に考えている
- [ ] ユーザーフィードバックを開発に反映する仕組みがある

5. AIの内部データへのアクセス

- [ ] AIツールが社内ドキュメントやSlackの会話を理解できる環境がある
- [ ] 組織固有のコンテキストをAIが学習できる仕組みがある
- [ ] 情報共有と透明性が重視されている

6. 高品質な内部プラットフォーム

- [ ] 開発者が共通で利用できる高品質なツールやインフラ基盤がある
- [ ] プラットフォームエンジニアリングが推進されている
- [ ] 開発者体験（DevEx）が向上している

7. 強力なバージョン管理プラクティス

- [ ] 全ての変更が追跡されている
- [ ] 問題発生時に迅速に以前の状態へ復元できる
- [ ] トレーサビリティと再現性が確保されている

評価

- 7つ全て該当 AI時代に成果を出し続ける組織の基盤が整っている
- 5-6個該当 良好な状態。残りのケーパビリティを強化する
- 3-4個該当 注意が必要。DevOpsの基本原則から見直す
- 0-2個該当 AI導入前にDevOpsの成熟度を高める必要がある

> ***出典*** *DORA（DevOps Research and Assessment）レポート2025年版『State of AI-Assisted Software Development』*

手法6 プロンプトエンジニアリング事例集

効果的なプロンプトの書き方

AIを効果的に活用するためには、適切なプロンプトを書くことが求められます。

技術的負債検出のプロンプト例

以下のコードベースを分析して、技術的負債の可能性がある箇所を特定してください。

- 循環的複雑度が10を超える関数
- テストカバレッジが70%未満の関数
- コードの重複が3箇所以上ある箇所
- 命名が不適切な変数や関数

各問題箇所について、以下の情報を提供してください
1. 問題の種類
2. 問題の深刻度（高/中/低）
3. 改善の提案

テストケース生成のプロンプト例

以下の仕様書を読んで、テストケースを生成してください。

機能：ユーザー登録
- ユーザー名：3文字以上20文字以下、英数字のみ
- パスワード：8文字以上、英数字と記号を含む
- メールアドレス：有効な形式であること

以下の種類のテストケースを生成してください
1. 正常系のテストケース
2. 異常系のテストケース（各入力項目のバリデーションエラー）
3. 境界値テスト（最小値、最大値、境界値-1、境界値+1）
4. エッジケース（特殊文字、空文字、nullなど）

コードレビュー支援のプロンプト例

以下のコードをレビューして、潜在的な問題を指摘してください。

レビューの観点：
- コードの品質（可読性、保守性、パフォーマンス）
- セキュリティの問題
- バグの可能性
- ベストプラクティスへの準拠

各問題について、以下の情報を提供してください
1. 問題の種類
2. 問題の深刻度（高/中/低）
3. 問題の説明
4. 改善の提案

プロンプトの改善方法

プロンプトを改善するには、以下のポイントを意識してください

- **具体的な指示を出す** 曖昧な指示ではなく、具体的な指示を出す
- **出力形式を指定する** AIがどのような形式で出力すべきかを指定する
- **コンテキストを提供する** AIが判断するために必要な情報を提供する
- **例を示す** 期待する出力の例を示すことで、AIの理解を助ける

章のまとめ

本章では、湊のチームがAIを戦略的に活用し、持続可能な開発を実現する過程を描きました。

主な学び

1. **AIは「コードを早く書く道具」以上のパートナー** AIをコード生成だけに使う

のではなく、開発全体を効率化するパートナーとして使う

2. **AIと人間の協働の重要性**　AIを完全に自動化するのではなく、AIと人間が協働することで、より効果的な開発が可能になる
3. **失敗から学ぶ**　すべてがうまくいくわけではない。失敗から学び、改善を続けることが重要。コンテキストエンジニアリングの重要性を理解し、システム自体をAIリーダブルな状態にする必要がある
4. **AI活用の組織的課題への対応**　AI疲れ、説明責任、AIスキルの標準化・育成など、組織的な課題に対処することが重要
5. **開発生産性ツリーによる戦略転換**　　開発生産性ツリーの観点から、戦略1（ValueStreamの高速化）と戦略2（Ops工数のAI置き換え）を実践することで、開発生産性を150%に向上させる。単一プロセスの効率化ではなく、全体最適を考えてAIネイティブなプロセスを設計する重要性を学ぶ
6. **成果と組織への影響**　AI活用の成果は開発チームだけでなく、組織全体に影響を与える。営業部、サポート部、マーケティング部など、他部署からの評価も得られる

次章への展望

本章でAI活用と持続可能な開発を描いた。次の章では、湊がエンジニアの視点だけでなく、PM、PdM、事業責任者の視点も取り入れた新しい生産性評価の枠組みを提案する過程を描く。