Qt Group

WHITEPAPER

AI を活用したソフトウェア開発を正しく進めるために

AI 支援ワークフローにおける
コード品質確保のベストプラクティス

目次

はじめに

ソフトウェア開発は加速しています。現在ではAIアシスタントがコードを生成し、テストコードの雛形を作成し、リファクタリング案を提示するようになりました。その速度は、個々の開発者が到達できる水準をはるかに超えています。生産性向上の効果は現実のものですが、それに伴うリスクもまた現実のものです。

AIは確率的にコードを生成します。AIが学習しているのは、「正しく見える」出力を生成することであり、必ずしも「実際に正しい」出力を生成することではありません。一般的な商用ソフトウェアであれば、その差異は管理可能な範囲に収まるかもしれません。しかし、自動車、航空宇宙、医療、産業機器といった安全性が重要なシステムではそうはいきません。コーディングガイドラインに違反し、本来のアーキテクチャから逸脱し、到達不能なロジックを含むコードベースは、AIがより速く生成したからといって安全になるわけではありません。

本ホワイトペーパーは、そのギャップに直面しているエンジニアリングチームを対象としています。つまり、AI支援開発による生産性向上の恩恵を享受しながらも、業界で求められるコード品質、規格準拠、監査対応能力を損なうことを望まない組織のためのものです。

解決策は、AIツールの利用を拒否することではありません。AIを決定論的な検証を組み合わせることです。静的コード解析とアーキテクチャ検証は、AIアシスタントと競合するものではありません。むしろAIを補完する存在です。AIは提案し、検証が判断します。本ホワイトペーパーでは、この組み合わせが実際の開発現場でどのように活用できるのかを解説します。具体的には、AxivionがAI支援開発のワークフローにどのように統合されるのか、その統合レベルにはどのようなものがあるのか、さらに、コード量が増加しコードの生成元の追跡が難しくなる中でも、認証や監査に必要なエビデンスを維持するためにどのような開発プロセス上の規律が求められるのかについて説明します。

この変化によって開発者の役割が小さくなるわけではありません。むしろ、その役割は再定義されます。AIが定型的・機械的な作業をより多く担うようになるにつれて、開発者の関心はより上位のレベルへと移っていきます。すなわち、構文を書く作業からアーキテクチャを設計・判断する作業へ、定型コード（ボイラープレートコード）を書く作業から、実装された振る舞いがシステムに本当に求められている振る舞いなのかを判断する作業へと重点が移ります。開発者が管理・監督するコードの量は増加します。そしてそれに伴い、そのコードに関して下す一つひとつの判断の重要性も高まっていきます。

安全規格の適用を受ける業界において、人間が担うこの役割は不可欠です。機能安全規格では、責任と説明責任はツールではなく、人や組織に課せられています。AIアシスタントがリリースを承認したり、監査の場で設計上の判断を説明したり、市場で発生した不具合に対する法的責任を負ったりすることはできません。決定論的な検証の役割は、開発者が適切な判断を下せるよう支援することです。具体的には、問題を早期に検出し、AIの提案を実際のコードベースの状態に基づいたものとし、さらにコードがどのように作成されたかに関係なく、認証や規格適合に必要なエビデンスを維持できるようにします。最終的な責任と判断は、常に開発者が担います。そして、その責任を現実的に果たせるよう支援するのがツールの役割です。

基本原則は非常にシンプルです。人が書いたコードであっても、AIが生成したコードであっても、同じ基準で検証しなければなりません。

AIを活用したソフトウェア開発:
変化、可能性、そして現実

変化:日常的な開発ツールチェーンの一部となったAI

AIアシスタントは、開発者がソフトウェアを開発する方法の中で、今や日常的な存在となっています。コード補完、リファクタリングの提案、テストコードのひな形作成、さらには関数全体の生成まで、これらはもはや実験的な機能ではありません。コンパイラと並んで、IDEの中に組み込まれた標準的なツールとなっています。

その効果は明確です。従来はエンジニアリング作業の多くの時間を費やしていた日常的なタスクにおいて、より多くのコードを、より短時間で作成できるようになり、開発の生産性は大幅に向上しています。

現実:コード量が増えても、正しいコードが増えるとは限らない

しかし、その生産性向上には注意すべき点があります。AIが生成したコードは、本質的に正しいわけでも、安全なわけでも、規格やルールに準拠しているわけでもありません。大規模言語モデル(LLM)が生成するのは、検証済みの出力ではなく、もっともらしく見える出力です。そのため、存在しないAPIを作り出したり、メモリを誤って扱ったり、コーディング規約に違反するパターンを再現したりすることがあります。さらに、コード生成の量と速度が増すことで、この問題は一層深刻になります。より短時間でより多くのコードが生成されるということは、人間が十分にレビューする前に、より多くの潜在的な欠陥がコードベースへ持ち込まれることを意味します。こうした変化に伴い、開発者の役割も変化しています。開発者は単なるコードの作成者から、レビュー担当者、統合担当者、そして最終的な判断を下す責任者へと役割を広げつつあります。しかし、その役割がなくなるわけではありません。技術的な判断、アーキテクチャ上の意図の維持、そして成果物に対する最終的な責任は、依然として人間が担うべきものです。

補完関係:確率的な生成を支える決定論的な検証

静的コード解析とAI支援は競合するものではなく、互いを補完する関係にあります。

AIはパターンに基づいて動作する確率的な仕組みであるのに対し、静的解析は決定論的であり、同じ条件であれば常に同じ結果を返し、その結果を監査可能な形で残すことができます。一方は提案し、もう一方は検証します。 そして 、この補完関係から導かれるのが 、本ホワイトペーパーの中心となる考え方です。 それは、AI が生成したコードも、人間が作成したコードとまったく同じように扱わなければならないということです。適用するルールも同じ、品質ゲートも同じ、求められるエビデンスも同じです。コードの生成元が何であれ、それを検証する義務は変わりません。

AI活用ワークフローにおけるAxivion

例外のないルール適用

出発点となるのは理念ではなく、実際の運用です。リポジトリに取り込まれるAI生成コードはすべて、人間が作成したコードと同様にAxivionによる解析の対象となります。コーディング規約(MISRA 、AUTOSAR C++14、CERT、独自ルールセットなど) やアーキテクチャルールは、コードの作成者に関係なく一律に適用されます。これらのルールに違反するコードは、それが人間によるものであれAIによるものであれ、品質ゲートによってコミットやマージが阻止されます。この考え方は、とりわけ安全規格の適用を受ける業界において重要です。IEC61508 、ISO26262 、IEC62304 、EN50716 といった機能安全関連の規格では、コードが規定されたルールに準拠していることを示す、追跡可能なエビデンスが求められます。そして、その要求はコードがどのように作成されたかによって変わるものではありません。安全関連ソフトウェアにおけるAI活用を扱う ISO/PAS 22440 は現在も策定が進められています。しかし、この規格が成熟するのを待つのではなく、現時点で実践的なアプローチは明確です。それは、AIが生成したコードも既存のコードベースと同じ検証プロセスの対象として扱い、監査や認証に対応できる状態を維持することです。

AIツールの判断基盤としてのAxivion

静的解析は、開発プロセスの後工程で実施する単なるチェックではありません。Axivionは、シンボル情報、コールグラフ、型情報、アーキテクチャレイヤー、適用ルール、そして現在の指摘事項を含む、コードベース全体の正確なモデルを維持しています。このモデルは、LLMベースのAIアシスタントが必要としていながら、通常は十分に得られない、プロジェクト固有の信頼できるコンテキストそのものです。この情報をAIアシスタントから利用できるようにすることで、推測やgrepによる単純な検索に頼る代わりに、実際のプロジェクトに基づく意味的（セマンティック）な情報を活用できるようになります。たとえば、ある関数がどこから呼び出されているのか、モジュールがどのアーキテクチャレイヤーに属しているのか、あるコード行がどのルールに違反しており、その理由が何なのかといった情報です。その結果、AIは周辺のコードから推測した情報ではなく、検証済みの情報に基づいて判断できるようになります。これにより、提案の精度が向上し、存在しない情報や誤った内容を生成してしまうケースを減らすことができます。また、この役割において Axivion は AI 自身を見守るガードレールとしても機能します。人間のレビュアーに届く前に、確率的なAIが見逃したり誤ったりした内容を、決定論的な検証によって検出するのです。

検索ソースとしてのAxivion：コード開発におけるRAG

AxivionとAIアシスタントを結び付ける仕組みが、RAG（Retrieval-Augmented Generation：検索拡張生成）です。特定のプロジェクト向けにモデルを再学習させる方法は、コストが高く、時間もかかり、さらに情報がすぐに陳腐化してしまいます。一方、RAGは問い合わせ時に関連するコンテキストを取得し、それをプロンプトに組み込む仕組みです。これにより、AIモデルは取得したエビデンスに基づいた回答を生成できるようになります。

AIコーディングアシスタントにとって、Axivionは理想的な検索ソースです。解析結果、ルールの根拠、アーキテクチャ上の制約、過去の検出結果といった情報は、構造化されており、最新で、かつ信頼できます。AIアシスタントがルール違反の修正、モジュールのリファクタリング、あるいは検出結果の説明を求められた場合、関連するAxivionの情報を取得し、推測に基づく近似的な回答ではなく、実際のコードベースの状態に基づいて回答できます。

RAG（Retrieval-Augmented Generation）とは？

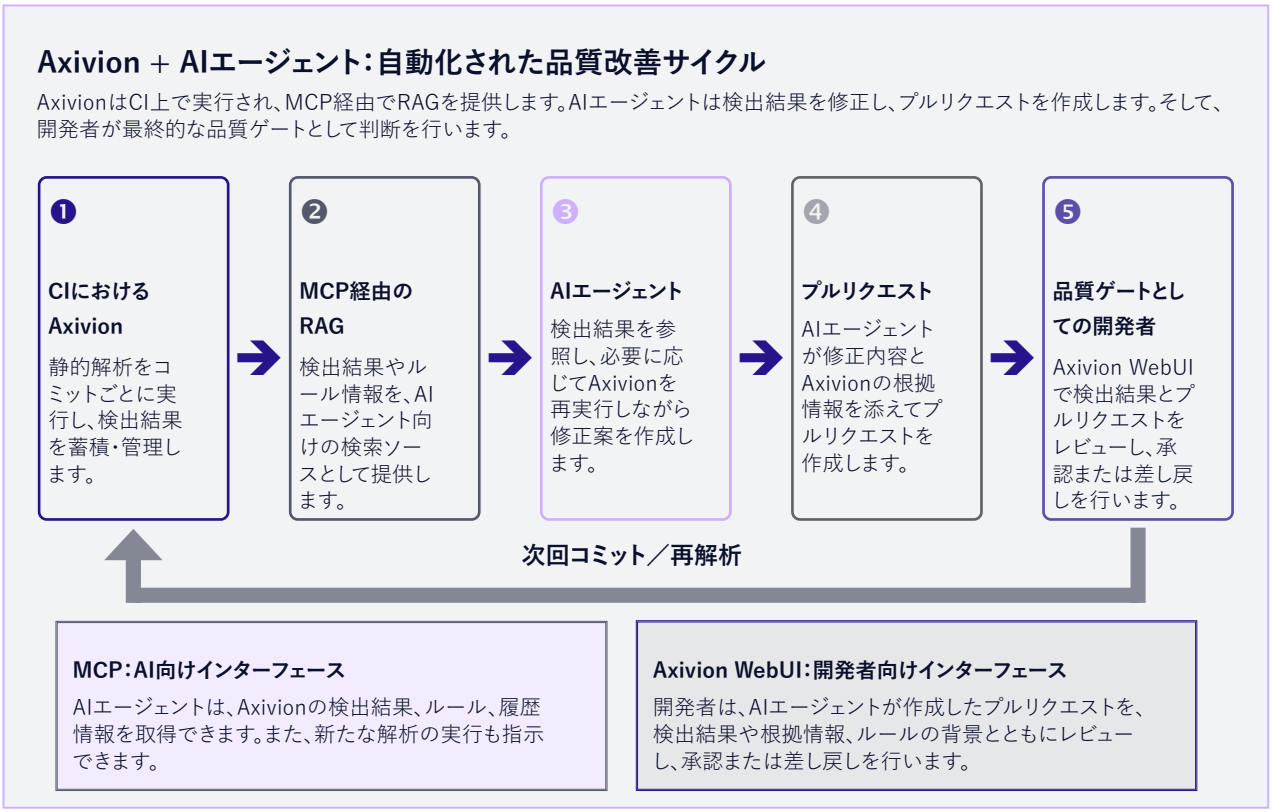
RAG（Retrieval-Augmented Generation：検索拡張生成）は、モデルが学習済みの知識だけに依存するのではなく、問い合わせ時に関連するコンテキストを取得して活用することで、LLMの回答を自社データやプロジェクト固有の情報に基づいたものにする仕組みです。

なぜ重要なのか： RAGを活用することで、AIのハルシネーション（もっともらしい誤情報の生成）を減らすことができます。また、モデルを再学習することなく常に最新の情報を反映できるため、信頼できる情報源を根拠として示しながら回答を生成できるようになります。

2つの統合レベル：
対話型利用と自律エージェント利用

Axivionは、AI支援開発と2つのレベルで連携できます。1つ目はプロトコルレベルの統合です。MCP（Model Context Protocol）サーバーを介して、MCPに対応したあらゆるAIアシスタントやAIエージェントからAxivionの解析結果を利用できるようにします。2つ目は、Axivion VS Codeプラグインによる、IDEに統合されたより深いレベルでの連携です。この2つは競合するものではなく、相互に補完し合う関係にあります。どちらも同じAxivionの解析結果を利用しており、多くの開発チームでは両方を組み合わせて活用することになるでしょう。

レベル1：MCP連携: あらゆるAIアシスタントのツールとしてのAxivion



Model Context Protocol（MCP）は、AIアシスタントが外部ツールを呼び出したり、問い合わせ時に外部データを取得したりするためのオープン標準です。MCPサーバーは利用可能な機能やデータを公開します。IDEに組み込まれたコパイロット、チャットアシスタント、自律型エージェントなど、MCPに対応したAIであれば、それらに接続し、利用可能なツールやデータを確認して活用できます。AxivionはMCPサーバーを提供しており、解析結果や設定情報を、このようなAIが利用可能なコンテキストとして提供します。

MCPサーバーが公開するのは、解析結果の要約ではなく、解析を構成する情報そのものです。具体的には、システムに適用されているルール、コードに対して報告された指摘事項とその発生箇所やコンテキスト、各ルールの根拠や違反を解消するための具体的なヒント、さらに解析結果に影響を与えるプロジェクト固有の設定情報などが含まれます。これには、サードパーティ製コード、生成コード、逸脱管理ワークフロー（Deviation Workflow）、そしてAIアシスタントだけでは推測できないプロジェクト固有の開発ルールや慣習も含まれます。MCP経由で接続されたAIアシスタントは、開発者がAxivionを利用するのと同じように情報へアクセスできます。ただし、その方法は ad-hoc な grep やファイル読み込みではなく、構造化されたツール呼び出しとして行われます。

MCPは汎用的なプロトコルであるため、同じAxivion MCPサーバーを対話型利用と自律型利用の両方で利用できます。

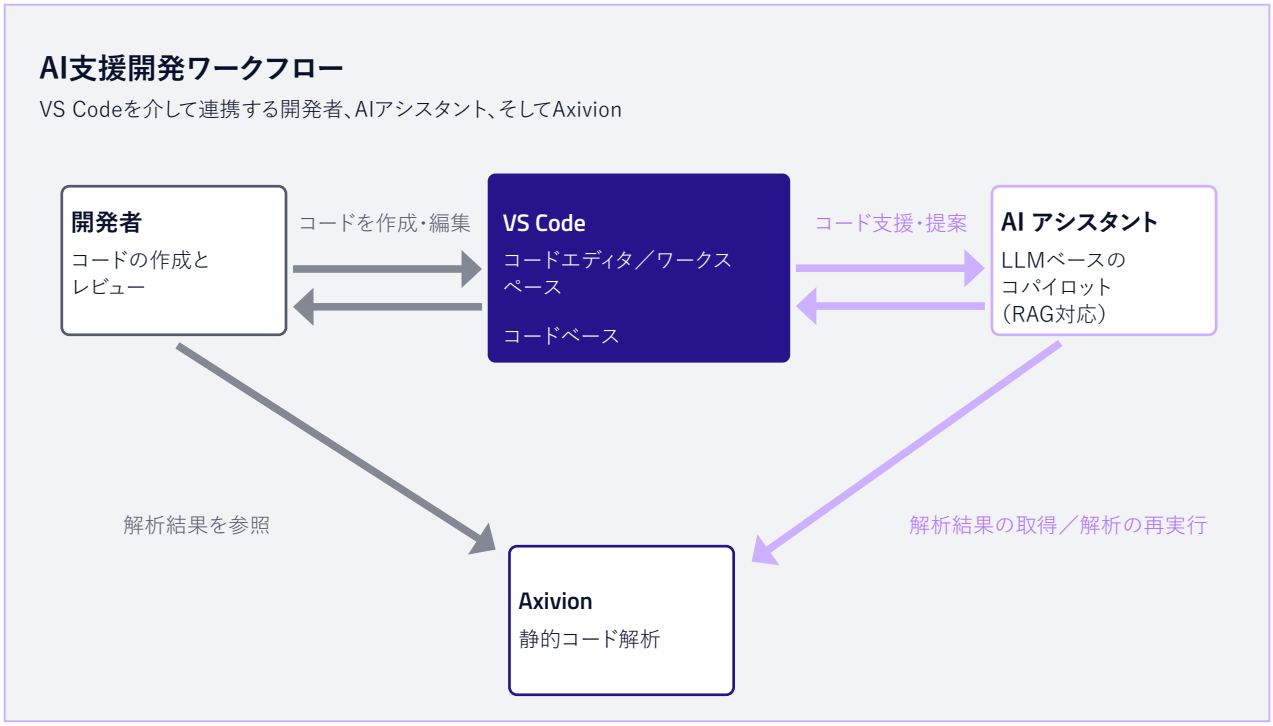
たとえば、IDE上でAIアシスタントと協調作業を行う開発者は
・なぜこのコードが指摘されているのか
・修正方法は何か
・この関数はどこから呼び出されているのか

といった、実際のプロジェクト情報に基づく回答を得られます。一方で、CI上でヘッドレス実行されるAIエージェントも、同じ情報を利用できます。AIエージェントは検出結果を分類・分析し、修正案を含むプルリクエストを作成し、その結果をAxivionの品質ゲートへ戻して人間によるレビューと承認を受けることができます。1つのサーバー、1つの信頼できる情報源（Single Source of Truth）、そして2つの利用形態。それがAxivion MCPサーバーの特徴です。

レベル2：VS Codeプラグイン: IDEに統合されたAxivion

VS Code プラグインは、 Axivionを開発者のエディタに直接統合します。このプラグインはAxivionの解析結果サーバーに接続し、指摘事項を該当するコードとともにエディタ上へインライン表示します。また、開発者だけでなく、同じワークスペースで動作するAIアシスタントも、解析コンテキストを参照したり、ファイル単位、サブプロジェクト単位、あるいはプロジェクト全体に対してローカル解析を再実行したりできます。これらの操作は、IDEを離れることなく行えます。AI支援開発のワークフローにおいて、これにより開発サイクルはより密接に連携したものになります。

AIアシスタントは、現在編集中のファイルに対する指摘事項を確認し、ルールの根拠や推奨される修正方法を参照できます。その情報を踏まえてコードを生成または修正し、さらに開発者がコミットする前にローカル解析を再実行して、その修正によって違反が解消されたことを確認できます。このように、指摘事項は後から確認するチェック結果ではなく、AI支援コーディングにおける重要な入力情報となります。一方で、開発者の役割は変わりません。ほかの変更と同様に、最終的なレビューと承認を行う責任は引き続き開発者が担います。



ベストプラクティス：AIとAxivionを効果的に活用するために

AIが生成した変更は必ずAxivionで解析する

コード生成からコミットまでのプロセスにおいて、解析を省略できない必須のステップとして組み込みましょう。AI アシスタントがファイルを新規作成または修正した場合 、その変更がレビュー対象となる前に Axivion による解析を実行する必要があります。IDE プラグインによるローカル解析でも、マージパイプラインでの解析でも、あるいはその両方でも構いません。重要なのは手順を増やすことではなく、誰が、あるいは何がコードを生成したかによって検証プロセスが変わらないようにすることです。これを行わなければ、AIによる生産性向上は、そのまま未検証コードの増加につながってしまいます。

Axivionの指摘事項をプロンプトに活用する

Axivionが報告した現在の指摘事項を含めてプロンプトを与えた場合、AIアシスタントは、何の情報も与えられない場合よりもはるかに適切な修正を提案できます。ルールID、発生箇所、ルールの根拠、推奨される修正方法といった情報は、LLMが周辺のコードだけから推測することができない、構造化されたプロジェクト固有のコンテキストです。MCP連携を利用すれば、これらの情報は自動的に取得できます。手動でコンテキストを貼り付ける場合でも考え方は同じです。違反しているルールとその根拠を明示したプロンプトは、そうでないプロンプトよりも良い結果をもたらします。

コンテキスト（文脈）なしではなく、違反情報を含めてプロンプトを作成する

「この関数を修正して」とだけ依頼するのは良いプロンプトとは言えません。良いプロンプトは、対象となるAxivionの指摘事項（ルール、発生箇所、根拠など）を具体的に示し、新たな違反を発生させることなく問題を解消するよう依頼します。結果の差は明らかです。前者は解析を通過するかどうかわからない、もっともらしい修正版を生成します。一方で後者は、具体的なルール違反の解消を目的とした修正を生成します。指摘事項は、後から確認するフィルターではなく、AIアシスタントとの対話の出発点として活用しましょう。

AIアシスタントに明示的なルールを与える

AIアシスタントは与えられたルールに従います。そして、ルールを与えないこと自体が、AIに自由な推測を許すルールになってしまいます。そのため、制約条件は毎回明示する必要があります。適用すべきコーディング規約（ MISRA 、AUTOSAR C++14 、CERT 、独自ルールなど）を伝え、既存の違反を修正する際に新たな違反を発生させないこと、指摘事項を抑制しないこと、コードベースに存在しないAPI・型・関数を作り出さないことを明確に指示しましょう。また、Axivionの判断とAI自身の推測が食い違う場合は、Axivionを正しい情報源として扱うことも伝えるべきです。指摘事項は、AIの自信と比較して採用するかどうかを判断する単なる提案ではありません。MCP連携を利用すれば、AIは推測ではなく実際のルールや現在の指摘事項を参照できるため、この一部は自動的に実現されます。しかし、それでもプロンプトでは次の期待事項を明示すべきです。

・Axivionから取得した情報に基づいて修正すること
・指摘事項を回避するのではなく解決すること
・不確実な点は隠さず開発者に提示すること

セッション開始時にこうした指示を数行追加するだけで、その後のAIの振る舞いは大きく変わります。

AI生成コードだからといって品質ルールを緩和しない

AIが生成したコードがルール違反を起こした場合、指摘事項を抑制したり例外を追加したりしたくなるかもしれません。しかし、その誘惑には抗うべきです。ルールセットには、プロジェクトが求める安全性、移植性、保守性に関する考え方が反映されています。AI生成コードのためだけにルールを緩和することは、短期的には問題を回避できても、長期的には機能安全規格が要求するエビデンスの信頼性を損なうことになります。もしルールそのものがプロジェクトに適していないのであれば、意図的に変更し、その判断を文書化してください。AIがルールを満たせなかったという理由だけで、その場しのぎにルールを緩和してはいけません。

AIとAxivion、それぞれの得意分野を理解する

AIアシスタントとAxivionは、それぞれ異なる強みを持っています。この違いを理解することが、両者を効果的に組み合わせる鍵となります。AIアシスタントは、局所的なコード補完や、意図をコードへ変換する作業に優れています。定型的なコード作成やリファクタリングを、人間を上回る速度で実行できます。一方で、プロジェクト全体を確実に理解することは得意ではありません。モジュール間の副作用を見落としたり、存在しないAPIを作り出したり、知らされていないコーディング規約やアーキテクチャルールに違反するコードを生成したりする可能性があります。Axivionは決定論的かつプロジェクト全体を対象とした解析を行います。ルール準拠、アーキテクチャ準拠、データフローの妥当性、デッドコードなどを、毎回同じ結果で検証できます。これは機能安全規格が求める、再現可能で監査可能なエビデンスそのものです。ただし、静的解析が判断できないこともあります。それは、実装された振る舞いが本当にユーザーの求めるものであるかどうかです。その判断は開発者の責任です。AIにはコード作成とリファクタリングを、Axivionには検証と制約の適用を、そして開発者には最終判断を任せましょう。これが、それぞれの強みを最大限に活かす方法です。

開発者が最終的な責任者であることに変わりはない

安全規格の適用を受けるソフトウェア開発において、人間が判断に関与することは単なるベストプラクティスではなく、必須要件です。機能安全規格では、責任と法的責任はツールではなく、人や組織に課せられています。AIアシスタントはコードの作成やリファクタリングを支援し、開発を加速することはできます。しかし、リリースを承認したり、監査の場で設計上の判断を説明したり、市場で発生した不具合に対する法的責任を負ったりすることはできません。そうした責任は、開発者と、それを支えるエンジニアリング組織にあります。

本ホワイトペーパーで紹介したワークフローは、この責任の所在を曖昧にするものではなく、むしろ強化するものです。Axivionは、AIが生成したコードに対しても、人間が作成したコードと同じルールを適用します。MCP連携は、AIアシスタントがプロジェクトの実際の指摘事項や制約に基づいて判断できるようにします。そしてIDEプラグインは、あらゆる変更について、開発者がレビューと承認を行う立場を維持できるよう支援します。AIが定型的・機械的な作業をより多く担うようになることで、開発者の関心はより上位の領域へと移ります。すなわち、実装の意図、アーキテクチャ、そしてシステムが本当に実現すべき振る舞いを見極めるための判断です。

その結果として生まれるのは、開発者の役割の縮小ではありません。むしろ、より大きな影響力を持つ役割への変化です。一方に決定論的な検証、もう一方に確率的なコード生成がある中で、最終的な判断を下すのはエンジニアです。何を開発するのかを決め、何をリリースするのかを承認し、その結果に責任を持つのは、常に人間です。

まとめ

AI支援開発は、もはや未来の話ではありません。すでに多くの開発チームにとって日常的な開発スタイルとなっています。いま問われているのは、AIツールを使うべきかどうかではありません。安全性が求められるソフトウェア開発に必要な品質や規格準拠を損なうことなく、どのようにAIを活用するかです。

本ホワイトペーパーが示した答えはシンプルです。AIが生成したコードも、人間が作成したコードとまったく同じように扱うことです。同じ解析を実施し、同じルールを適用し、同じエビデンスを求める。そしてAxivionを、AIの提案を実際のコードベースの状態に基づいたものにするための、決定論的な基盤として活用します。MCPを通じてあらゆるAIアシスタントやAIエージェントと連携し、VS Code プラグインを通じて IDE に統合されたワークフローを実現し、さらに CI の品質ゲートによって、人間が書いたコードかAIが生成したコードかを区別することなく同じ基準で検証します。

AIが開発速度を向上させる一方で、Axivionはその速度を安心して活用するための検証を提供します。また、AI がもたらす不確実性 —— 存在しない API の生成、コーディング規約違反、アーキテクチャの逸脱など——に対しては、静的解析とアーキテクチャ検証が、人間のレビュアーや顧客の目に触れる前に問題を検出します。

開発者は、プロセス全体を通じて意思決定者であり続けます。それは機能安全規格が求めることであり、優れたエンジニアリングに不可欠な考え方でもあります。変わるのは、開発者が活用できる力の大きさです。より多くのコードを解析できるようになり、より多くの指摘事項を早期に発見できるようになり、そしてツールでは代替できないアーキテクチャ設計やシステムの意図に関わる判断に、より多くの時間を割けるようになります。

AIによるコード生成と決定論的な検証を組み合わせることで、それぞれ単独では実現できない価値が生まれます。より速く、より信頼性が高く、より説明可能なソフトウェア開発。それこそが、AIとAxivionを組み合わせることで実現できる開発の姿です。

Qt Group

www.qt.io/axivion-ai