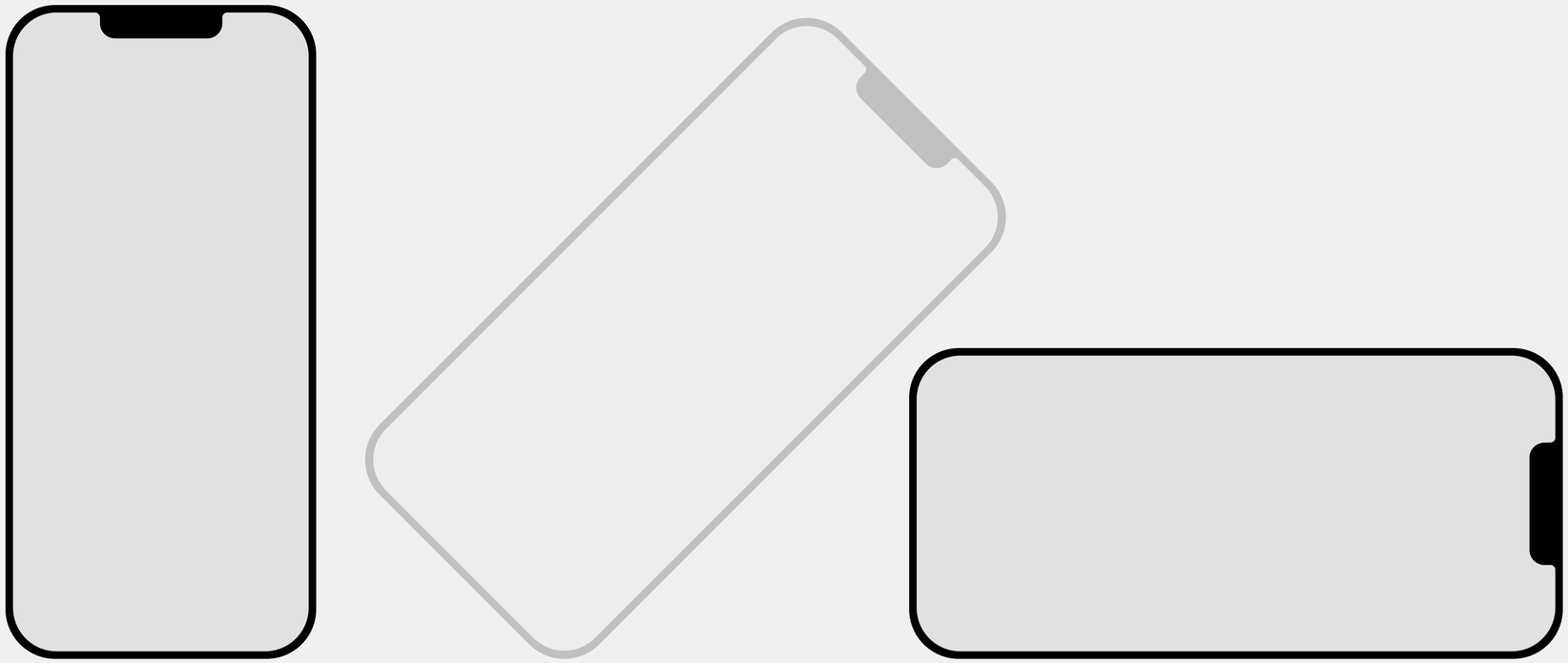
Qt Group

AI時代における信頼性の高いソフトウェア構築

決定論的な基盤へ向けたリーダーズガイド

Qt Group

モバイル端末でお読みですか？
端末を横向きにすると読みやすくなります。



Qt Group

目次

はじめに：
すでに直面している問い

あなたは、安全性が重視されるソフトウェア開発に携わる組織を率いています。開発する製品は、ISO 26262、IEC62304、IEC 61508 といった規格への適合が求められます。そのために、長年かけて開発プロセスを整備し、認証を取得し、技術やノウハウを積み重ねてきました。

しかし今、AI が二つの側面から同時に開発現場へ入り込んできています。

一つは、製品そのものです。製品ロードマップには、AI を活用した機能、認識システム、適応型アルゴリズム、インテリジェントな自動化などが含まれています。市場はそれらを求めており、競合他社もすでに提供を始めています。立ち止まるという選択肢はありません。

もう一つは、開発現場です。エンジニアは日常的に AI アシスタントを活用し、コード生成、テストの提案、ドキュメント作成などを行っています。生産性は向上しています。だからといって、「AI の利用をやめよう」と言うことは現実的ではありません。

そこで、私たちは次の問いに正面から向き合う必要があります。
現在の検証プロセスは、この二つの変化に対応できるでしょうか。

AI が生成したコードは、いまや開発チームがレビューできる速度を超えるペースでコミットされています。AI を活用した機能も、安全性を保証するためのセーフティケースを更新できる速度を上回るペースで設計されています。ダッシュボードに表示される開発スピードは向上しているように見えます。しかし、その一方で、何かが静かに積み上がっていることにも気付いているはずです。それは、開発のスピードとリリースするソフトウェアへの確信との間に広がるギャップです。

それは気のせいではありません。そのギャップは現実に存在し、スプリントを重ねるたびに広がっていきます。

このガイドは、次の二つの望ましくない選択肢のどちらも受け入れたくないと考えるリーダーのために書かれています。

- 安全性を維持するために AI の導入を抑え、その間に競合他社に先を越される
- AI の導入を加速し、既存のプロセスでも対応できることを期待しながら、目に見えないリスクを蓄積していく

しかし、第三の道があります。それは、AI を安全に活用するための決定論的な検証基盤を構築することです。コードが人間のエンジニアによって書かれたものであっても、AI アシスタントによって生成されたものであっても、その違いに関係なく、品質が基盤の設計そのものから生み出される仕組みを構築することです。

すでに今日、この基盤を構築している組織は、AI を複利的な競争優位へと変えています。検証がリリースのボトルネックにならないため、より速く製品を提供できます。また、変更の影響範囲が限定されていることを実証できるため、より大きな確信を持ってイノベーションを進めることができます。

なぜ、今取り組むべきなのでしょうか。

そのギャップは、複利的に広がっていくからです。

脆弱な基盤の上で AI を活用して行われるコミットは、一つひとつが、いずれ支払うことになる技術的負債を積み増していきます。さらに、求められる基準そのものも変化しています。フィジカル AI（自動運転やロボティクスなど）の分野で求められ始めている検証要件は、今後、安全性が重視されるソフトウェア開発全般における基本要件になっていくでしょう。今からこうした能力を備える組織は、それらが必須となったときには、すでに対応できる状態にあります。

このガイドを読み終える頃には、次のことをご理解いただけます。

- なぜ AI は、既存の開発基盤の良し悪しを増幅するのか
- AI を活用して成果を上げる組織と、AI が生み出す複雑さに埋もれてしまう組織を分ける具体的な能力とは何か
- 現在の検証基盤が、AI 時代の開発に対応できるかどうかを見極める方法
- AI の開発スピードに対応できる検証基盤を構築するために必要なこと

それでは始めましょう。

01
AIはコードベースに何をもたらすのか

AI はコードベースに何をもたらすのか

調査によると、ソフトウェア開発の時間のおよそ50%は、変更を加える前に既存のコードを理解することに費やされています。つまり、「すでに存在するコードを理解すること」に、多くの時間が費やされているのです。

私たちはこれを、「理解税」と呼んでいます。これは「コードの理解にかかるコスト」で、すべてのエンジニアが日々支払っています。

AI を活用した開発は、この状況を次の二つの方向へ変化させます。

1. アーキテクチャが検証され、ルールや規格が確実に適用されていれば、AI が生成したコードも、人間が理解しやすい構造の中に収まります。コードの理解に要する負担は管理可能な範囲にとどまり、AI は開発を加速させます。
2. アーキテクチャが文書や人の記憶の中にしか存在せず、ルールや規格の遵守がレビュー担当者の注意力に依存している場合、AI が生成したコードは、既存の複雑さをさらに増幅します。生成には数分しかかからなかったコードでも、それを理解するには何時間もかかるようになります。
   AI は、このソフトウェアの劣化を加速させます。

ソフトウェアの劣化は、予測可能な経過をたどる

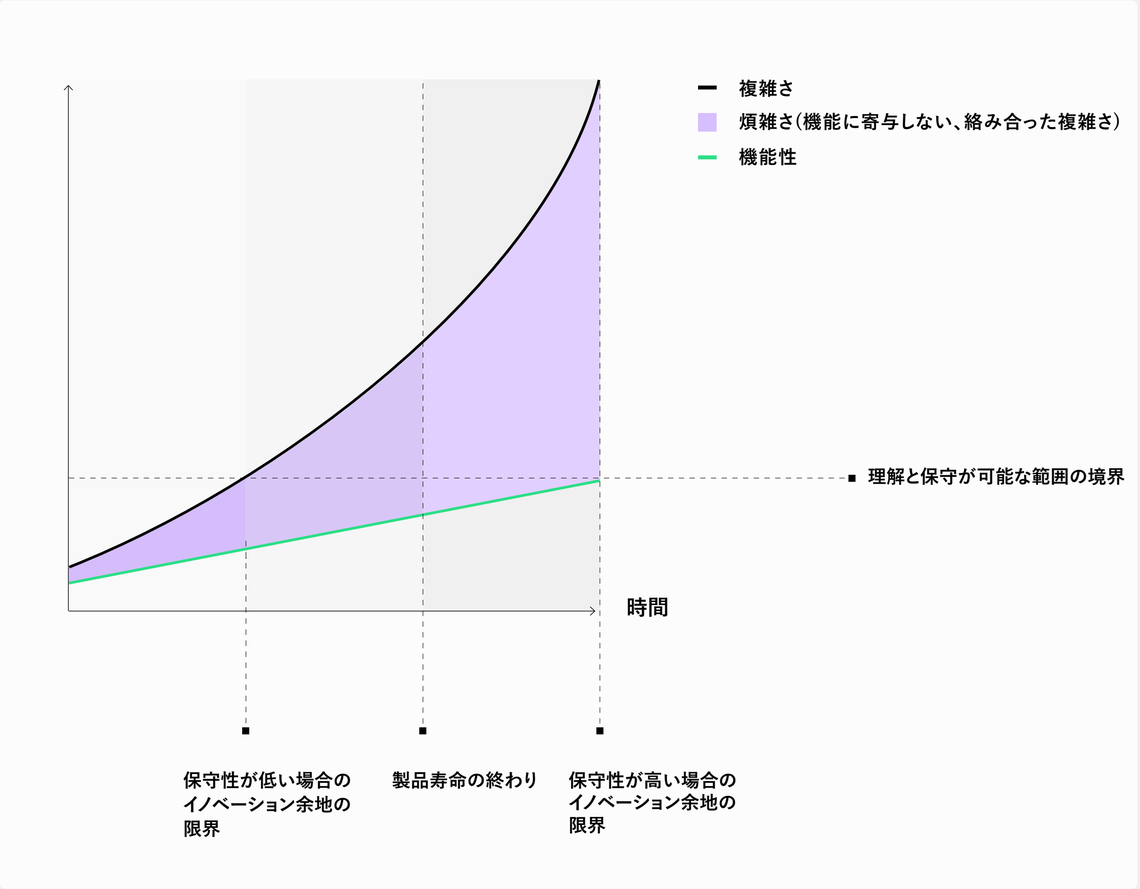
フェーズ1（管理可能な段階）: 近道や場当たり的な対応があっても、まだ管理可能な状態です。開発チームは開発スピードを維持できています。技術的負債は存在しますが、開発の妨げになるほどではありません。

フェーズ2（指数関数的な悪化）: 複雑さの増加が、人間の理解できる速度を上回り始めます。変更を加えるたびに、より多くの調査や分析が必要になります。AI は、開発チームが消化できる以上の速さでコードを増やしていきます。

フェーズ3（破綻）: コードベースは、安全に変更を加えられないほど複雑になります。製品が終わりを迎えるのは、市場の需要がなくなったからではありません。コードを進化させられなくなるからです。

AI は、この流れそのものを変えるわけではありません。AI は、開発基盤の状態に応じて、その進行を加速させます。

フィジカル AI（現実世界で動作する自律システム）を開発する組織では、フェーズ2のソフトウェア劣化は重大な責任問題につながり、フェーズ3に至れば、リコールや規制当局による措置、あるいはそれ以上の深刻な事態を招く可能性があります。しかし、この原則は フィジカル AI に限った話ではありません。AI は、既存の開発基盤の良し悪しを増幅します。中立という選択肢はありません。



02
誤った二者択一

誤った二者択一

私たちは、安全性が重視されるソフトウェア開発において、AIへの対応が次の二つの極端なアプローチに分かれる場面を数多く見てきました。

慎重すぎる対応（Paralysis）: 規格適合を守るためにAIの利用を制限する。その結果、より迅速に開発を進める競合に後れを取ります。市場が先へ進む中、自分たちは慎重に行動しているのだと言い聞かせることになります。

無謀な対応（Recklessness）: AIを積極的に導入し、既存の開発プロセスでも対応できることを期待する。開発スピードの向上を喜ぶ一方で、検証すべき項目は積み上がり続け、人間だけで開発していた場合よりも速いペースでリスクを蓄積してしまいます。

どちらの考え方にも、「スピードと安全性はどちらか一方しか選べない」という前提があります。しかし、本当に問うべきなのは、「**あなたが求めるのは、どのようなスピードなのか？**」ということです。

1. 近道によるスピードは、未来からの前借りです。検証を犠牲にしたスプリントは、そのたびに将来対応しなければならない作業を生み出し、その負担はさらに増幅されて返ってきます。

2. 信頼できる開発基盤から生まれるスピードは、積み重なるほど大きな効果を生みます。検証基盤への投資は、将来のすべてのリリースにわたって継続的な成果をもたらします。

どちらを選んだとしても、AIはあなたが選んだその開発のあり方を増幅します。

第三の道があります。そして、すでにそれを実践している組織があります。

- **AIがコードを生成する**
  人によるレビューの前に、自動化された検証がアーキテクチャや規格への適合性を確認します。エンジニアは、機械には判断できないことに集中して判断を下します。

- **AIが変更案を提案する**
  アーキテクチャ検証によって、その変更の影響範囲が限定されていることを実証します。再検証も、実際に影響を受ける範囲だけに絞り込まれます。

- **AIが開発を加速する**
  決定論的なツールが、監査で求められる規格適合を示すエビデンスを継続的に生成します。

こうした組織は、安全性を維持しながら AI を活用した製品を市場に投入しています。また、AI を活用した開発によって開発スピードを高めながら、コード品質も向上させています。それは、スピードと安全性の両立を可能にする基盤を築いているからです。

03
信頼できる
ソフトウェア基盤

信頼できるソフトウェア基盤

信頼できるソフトウェア基盤とは、組織として備えるべき能力です。つまり、顧客、規制当局、監査担当者、そしてソフトウェアを保守・運用する開発チームのすべてが信頼できるソフトウェアを継続的に提供する能力を指します。

この基盤が、単なる開発プロセスやツール群と異なる点は、品質が個々の担当者の努力の結果ではなく、基盤の設計そのものから生み出されることです。基盤が整備されていれば、品質は「適切なレビュアーが特定のコミットをレビューしたかどうか」に左右されません。

AI を活用した開発によって、この違いはより明確になります。コードが人間によるレビューの速度を上回るペースで生成されるようになると、人の努力に依存した品質確保や規格適合の確認はスケールしません。一方、基盤によって実現される品質はスケールします。


```
Page {
  id: root
  Material.theme: darkModeToggle.checked ? Material.Dark : Material.Light

readonly property bool mobile: Qt.platform.os === "android"
readonly property bool horizontal: width > height
property real sliderWidth: width * 0.15
property real buttonRowWidth: width * 0.12
property real buttonMinWidth: 65

leftPadding: 60
rightPadding: 60
topPadding: 50
bottomPadding: 50

width: 800
height: 600
state: "mobileHorizontal"

Backend {
  id: backend
  rotation1Angle: rotation1Slider.value
  rotation2Angle: rotation2Slider.value
  rotation3Angle: rotation3Slider.value
  rotation4Angle: rotation4Slider.value
  clawsAngle: clawToggle.checked ? 0 : 90
}

Toggle {
  id: darkModeToggle
  text: qsTr("Dark mode")
  anchors.top: parent.top
}

ColumnLayout {
  id: slidersColumn
  spacing: 6
  anchors.bottom: parent.bottom

  LabeledSlider {
    id: rotation1Slider
    Layout.preferredWidth: root.sliderWidth
    Layout.minimumWidth: 160
    labelText: "Rotation 1"
    from: -90
    to: 90
    value: 60
  }

  LabeledSlider {
    id: rotation2Slider
    Layout.preferredWidth: root.sliderWidth
    Layout.minimumWidth: 160
    labelText: "Rotation 2"
    from: -135
    to: 135
    value: 45
  }

  LabeledSlider {
    id: rotation3Slider
    Layout.preferredWidth: root.sliderWidth
    Layout.minimumWidth: 160
    labelText: "Rotation 3"
    from: -90
    to: 90
```

この信頼できるソフトウェア基盤は、3つの柱によって支えられています。

第1の柱：常に実装と整合したアーキテクチャ

文書化して保管されているだけのアーキテクチャは、実装との乖離を待っているようなものです。AI は、このアーキテクチャと実装の乖離を加速させます。高速に生成されるコードは、安全性を確保するための境界を、巧妙で、一見もっともらしく、しかも大規模なコードレビューでは見逃されやすい形で破ってしまうことがあります。

常に実装と整合したアーキテクチャとは、アーキテクチャを機械が読み取れるモデルとして表現し、コミットのたびに実装との整合性を検証することです。アーキテクチャと実装の乖離は直ちに検出され、安全性に関わる境界が守られていることも継続的に検証されます。これにより、監査の直前になって初めてアーキテクチャを再構築し、整合性を確認するといった従来のやり方から脱却できます。

AI 時代の開発では、検証されたアーキテクチャは、複雑さが人間の理解力を超えたときでも開発を正しい方向へ導く地図となります。それがなければ、現実のソフトウェアは、誰一人として全体を理解できないものになってしまいます。

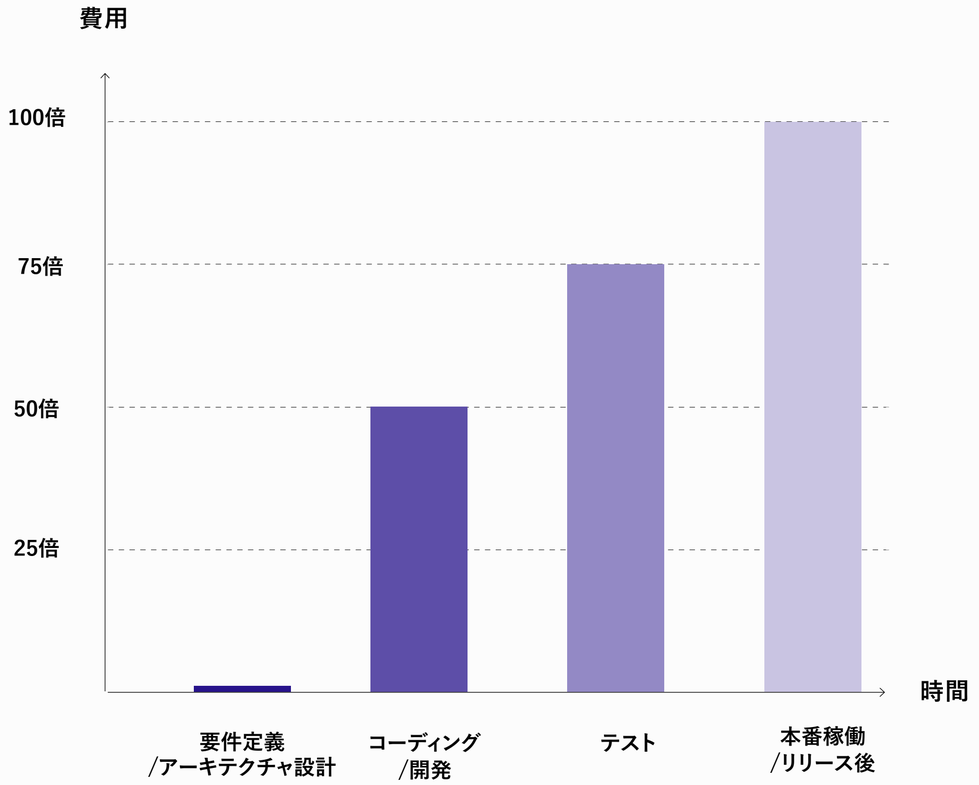
第2の柱：継続的検証

参考までに、不具合を発見するコストは、開発中であれば約1倍です。結合工程では10倍、製品リリース後には100倍、そしてお客様への導入後には1000倍にもなります。

AI はコードの生成を加速します。しかし、検証がそれに見合う速度で行われなければ、不具合は発見されるよりも速いペースで生み出され、欠陥はコストカーブのより高い段階へと持ち越されてしまいます。

継続的検証とは、コミット時点で解析を実行し、不具合を作り込んだその場で検出し、品質チェックを開発プロセスに組み込むことです。そして、この考え方の根底にある原則はシンプルです。プロンプトだけで規格適合を実現することはできません。

AI が本当に必要としているのは、多くの組織で時間とともに失われてきたものです。すなわち、要求仕様書、設計仕様書、検証されたアーキテクチャ、そしてルールや規格を確実に適用・遵守する仕組みです。こうした基盤がなければ、AI コパイロットは、一見もっともらしく見えても、自らが認識していない制約に違反するコードを生成してしまいます。一方、それらの基盤が整っていれば、AI はガードレールを備えた開発の加速装置となります。



第3の柱：明確な責任分担

機械は、形式的な規格適合の確認を効率よく行います。例えば、パラメータ数、結合度メトリクス、規格違反、安全境界の遵守状況などです。疲れることなく、一貫した精度で、しかも高速に処理できます。

一方、人間は、意味的な妥当性を判断します。

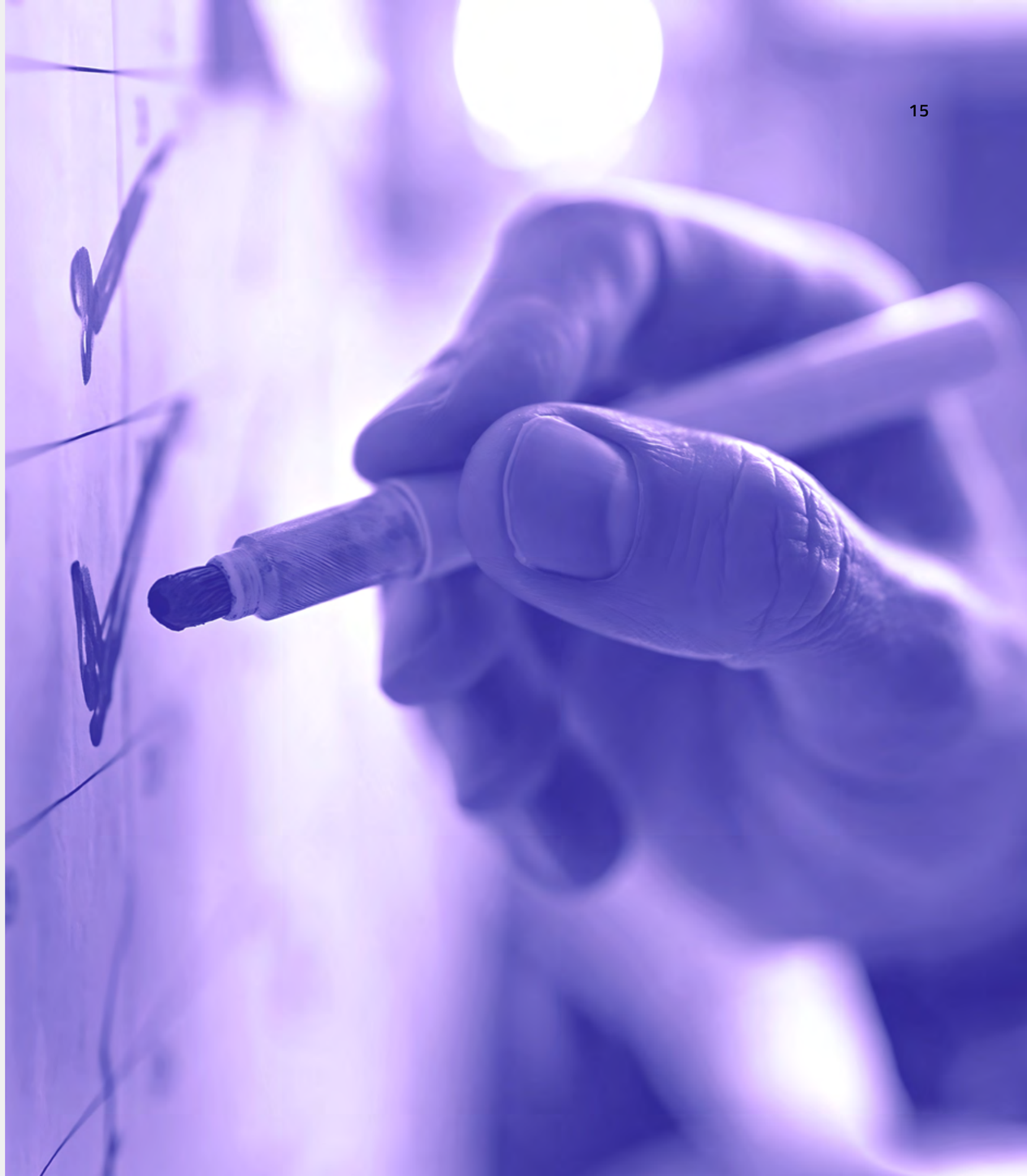
この設計や実装は妥当か。実装は設計意図に沿っているか。この抽象化は適切か。

AI は、人間には及ばない速度でコード、設計パターン、さまざまな選択肢を生成します。

AI 時代の開発では、それぞれの強みを生かした役割分担が重要です。AI は高速にコードを生成し、自動化ツールは生成と同じ速度でアーキテクチャや規格への適合性を検証します。そして、人間のエンジニアは、機械が検出・提示した結果に対して意味的な判断を下します。

問題となるのは、役割を取り違えることです。AI に検証を任せたり、人間が AI の生成したコードを大規模に手作業でレビューしたり、あるいは AI の出力が十分に良さそうだからという理由で検証そのものを省略したりすることです。

どのような方法でコードが作成されたかに関係なく、リリースされるソフトウェアに対する責任は、最終的に開発組織が負います。



04
開発基盤がスケールしないことを示す4つの警告サイン

開発基盤がスケールしないことを示す4つの警告サイン

警告サイン1：AI の開発スピードが検証待ちを生み出している

開発は加速しているのに、検証は追いついていません。その結果、「コード完成」と「検証済みリリース」の間にあるギャップは、スプリントを重ねるたびに広がっていきます。

この検証待ちは、ダッシュボードには表れません。問題が顕在化するのは、いざリリースしようとしたときです。「開発完了」と「リリース可能」の間に、どれほど多くの作業が残っているかに初めて気付くことになります。開発スピードを示す指標は改善しているにもかかわらず、実際のリリースサイクルはむしろ長くなっていきます。

警告サイン2：すべての変更で全面的な再検証が必要になる

AI は一見すると単純な変更を提案します。しかし、検証されたアーキテクチャがなければ、その変更の影響範囲が限定されていることを実証できません。その結果、「どこに影響するか分からないのだから、すべてを検証しよう」という判断になります。

本来なら数日で終わるはずの作業が、数週間かかるようになります。AI による生産性向上は、再検証に費やす工数によって相殺されてしまいます。アーキテクチャによって変更の影響範囲が限定されていることを実証できない組織では、実際に影響を受ける範囲だけに検証対象を絞り込むことができません。その結果、更新サイクルは長期化する一方で、変更の影響範囲を実証できる競合他社は、継続的かつ迅速に改善を繰り返していきます。

警告サイン3：規格適合を示すエビデンスが「発掘作業」になっている

これは、特に安全性が重視されるシステムで顕著な問題です。監査が近づくと、手作業による断片的な情報収集が始まります。必要な文書を探し出し、それが現在の実装を正しく反映しているかを確認し、本来であれば日頃から維持されているはずのトレーサビリティを再構築する——そんな作業に追われることになります。

AI は、この問題をさらに加速させます。生成されるコードが増えれば、それだけ追跡すべき対象も増えます。トレーサビリティが自動化されていなければ、実際のソフトウェアと文書との乖離は、これまで以上の速さで広がっていきます。本来であれば自動的に生成できるはずのレポートを作成するために、何週間もの準備を費やすことになるのです。

もっと詳しく

↗ **記事 - コードとしてのアーキテクチャ：開発者に優しいアーキテクチャ検証アプローチ**

警告サイン4：アーキテクチャがコードではなく、人の記憶の中にある

システムの構造を熟知しているのは、経験豊富なエンジニアです。しかし、その知識を属人的な経験や暗黙知に頼っていると、多くのボトルネックや不要なリスクを生み出してしまいます。

AI は、この脆弱さも増幅します。アーキテクチャの意図を理解することなく、透明性もないまま生成されたコードは、人の頭の中や手作業で管理された文書にしか存在しない設計上の境界を容易に破ってしまいます。もし誰かから、「このコンポーネントが、あのコンポーネントに影響を与えないことを実証できますか？」と尋ねられたとき、あなたのチームは迷わず答えられるでしょうか。

05
信頼できるソフトウェア基盤に必要なもの

信頼できるソフトウェア基盤に必要なもの

機能1：アーキテクチャ検証

アーキテクチャ設計書には、「コンポーネントAはコンポーネントBを呼び出してはならない」と定義されて いるとします。この境界は、安全性を確保するために設けられています。例えば、AがQMレベル、BがASILDであり、Freedom from Interference（FFI） やソフトウェアの分離によって、両者が適切に隔離されていることを実証する必要がある場合です。

AI コパイロットは、こうした設計上の制約を理解していません。AI が最適化するのは「コードを動作させること」であり、自ら認識できない設計上の境界を維持することではありません。その結果、違反は気付かれないまま蓄積し、監査や障害の発生時になって初めて発見されることになります。

これを実現すると

アーキテクチャは、開発環境やツールチェーン全体で利用できる、機械が処理可能な形式で記述されます。さらに、すべてのコミットで自動的に検証が実行され、AUTOSARなどのルールも自動的に適用・検証されます。違反があれば、その場で文脈とともに通知されるため、数週間後にレポートで初めて問題が見つかるような事態を防げます。

導入によって得られること

- 変更の影響範囲が限定されていることを実証し、実際に変更された箇所だけに再検証の対象を限定できる
- 自動化によって、安全性を確保するためのパーティショニングが適切に実現されていることを実証できる
- レガシーシステムのアーキテクチャリカバリーによって、「実際のシステム構造」と「そう思い込んでいた構造」との違いを可視化できる

機能2：AI 時代の開発スピードに対応する静的解析

人間のレビュー担当者でも、規格への適合性を確認することはできます。しかし、それを大規模な開発で一貫して行うことは困難です。疲労や時間的な制約、あるいは特定のルールへの理解不足によって、規格違反を見逃してしまうことがあります。AI が生成したコードは、この問題をさらに増幅します。コードの量は飛躍的に増え、一見するともっともらしく見える一方で、規格違反は気付かれないまま紛れ込んでしまいます。

さらに、規格やコーディングガイドラインが PDF や Wiki の中にあるだけでは、この問題はより深刻になります。エンジニアごとに解釈が異なり、AI コパイロットも、それらを判断するための明確なコンテキストを持っていないからです。

これを実現すると

Lint による表面的なチェックではなく、高度な意味解析を実施します。業界標準やコーディングガイドラインは、自動的に適用・検証されます。また、過去の不具合や知見から得られた学びを反映しながら、組織独自のルールも継続的に進化させることができます。解析はすべてのコミットで実行できる十分な速度を備え、開発環境と統合されることで、開発者はコードを書いているその場で規格違反に気付くことができます。

導入によって得られること

- コードの作成元に関係なく、一貫した品質を確保できる
- 不具合を、修正コストが最も低い開発初期の段階で検出できる
- レビュー担当者によって判断が変わらない、一貫したルール適用を実現できる
- 組織の知見を蓄積し、担当者やチームが変わっても継承できる

機能3：継続的な規格適合を示すエビデンス

機能安全認証では、トレーサビリティが求められます。要求仕様から設計、実装、テスト、そしてコードカバレッジに至るまで、それぞれのつながりを示さなければなりません。従来は、これらのエビデンスを監査の直前になってから遡って収集・整理するのが一般的でした。しかし、このやり方は AI 時代の開発スピードには対応できません。生成されるコードが増えれば、それだけ追跡すべき対象も増えます。変更の頻度が高くなるほど、文書は実装からさらに乖離していきます。

これを実現すると

開発の進行に合わせてトレーサビリティを継続的に記録・管理します。規格適合を示すエビデンスは自動的に生成され、常に現在の状態を反映したレポートを作成できます。

導入によって得られること

- 監査の準備を、数週間ではなく数時間で完了できる
- 変更の影響を即座に把握できる。例えば、要求仕様を変更すると、その影響を受ける実装やテストをすぐに確認できる

機能4：開発フローに統合された検証

開発フローから切り離されたツールは、時間に追われる開発現場では使われなくなります。納期が近づくにつれ、別の解析ポータルを開かなければならないこと自体が負担となり、開発者はそれを避けるようになります。AI が開発スピードを加速させることで、この問題はさらに深刻になります。

これを実現すると

コード解析やアーキテクチャ検証は IDE 上で実行され、違反はほぼリアルタイムで開発者に通知されます。コミット時には検証が自動的に実行され、その結果は文脈とともに表示されるため、断片的で統一されていないレポートに頼ることなく、違反内容やその理由をすぐに理解できます。

導入によって得られること

- 問題を検出するのではなく、未然に防ぐことができる
- 検証が開発スピードの妨げにならないため、持続可能な開発スピードを実現できる
- AI の開発スピードに合わせて基盤が動作することで、品質も AI の開発スピードに対応して向上できる

これらの機能は相互に連携する

これら4つの機能は、それぞれが独立しているのではなく、一つの仕組みとして機能します。アーキテクチャ検証がシステムの境界を定義し、その境界の中で静的解析がルールや規格への適合を検証します。継続的な規格適合を示すエビデンスは、その両方の結果を基に自動的に生成されます。そして、それらすべてを開発フローへ統合することで、検証を継続的かつ無理なく実施できるようになります。

重要なのは、**どれか一つでも欠けると、他の機能の効果も十分に発揮できなくなる**ということです。これらは互いに補完し合いながら、信頼できるソフトウェア基盤全体を支えています。

06
この選択がもたらす複利効果

この選択がもたらす複利効果

検証基盤への投資は、良い方向にも悪い方向にも複利的な効果をもたらします。

検証基盤を構築する組織は、リリースを重ねるたびに優位性を積み上げていきます。アーキテクチャ検証によって、以後の変更はより短時間で検証できるようになります。導入には確かに初期投資が必要ですが、それは一度限りの投資であり、その後のすべてのプロジェクトがその恩恵を受けられます。継続的な規格適合を示すエビデンスによって、監査の準備は数週間ではなく数時間で済むようになり、その効果は認証のたびに繰り返し得られます。さらに、検証が開発フローへ統合されることで、エンジニアは日々の開発を通じて望ましい実装パターンを自然に身につけます。その結果、規格違反は時間とともに減少し、品質は組織全体で継続的に向上します。品質は、レビュー担当者の力量や経験に左右されなくなります。

一方で、その逆もまた真実です。脆弱な基盤の上では、AI が十分に検証されていないコードをコードベースへ次々と追加していくことで、コードの理解に要する負担は増え続けます。変更の影響範囲が限定されていることを実証できないため、検証すべき範囲は際限なく広がります。監査のたびに手作業による対応が必要となり、そのコストは毎回変わることなく発生し続けます。 このように、良い方向と悪い方向の両方で複利的な効果が同時に働くため、組織間の差はスプリントを重ねるたびに広がっていきます。

フィジカル AI の普及は、この変化をさらに加速させています。自律システム向けに求められ始めている検証要件——変更の影響範囲が限定されていることの実証や、計算境界をまたいでも維持されるアーキテクチャの完全性など——は、今後、あらゆる安全性が重視されるソフトウェア開発における基本要件となっていくでしょう。規制の枠組みは変化し、サプライチェーンから求められる要件も厳しくなっています。今からこうした基盤を構築する組織は、それらが標準的な要件となったときには、すでに必要な能力を備えていることになります。

どこから始めるべきか

最初に取り組むべきなのは、アーキテクチャ検証です。これは、残りの3つの機能を支える土台となります。検証されたアーキテクチャの境界は、静的解析に構造的なコンテキストを与えます。変更の影響範囲が限定されていることを実証できるようになれば、実際に影響を受ける範囲だけに再検証の対象を絞り込めます。また、アーキテクチャが明確になることで、監査で求められる構造を備えた規格適合を示すエビデンスを作成できるようになります。

すでにアーキテクチャ設計書がある組織であれば、それを機械が処理できる形式にし、自動検証を実行できるようにすることから始めましょう。最初の検証では、実装が設計からどのように乖離しているのか、設計書には存在する境界がコードには反映されていない箇所はどこか、といった「現実」が明らかになります。そのギャップこそが、改善の出発点です。

正式なアーキテクチャモデルを持たない組織であれば、まずはアーキテクチャリカバリーから始めることをお勧めします。適切なツールを使えば、コードベースから実際のシステム構造を自動的に復元できます。そこで得られるのは、システムの現状を正確に表した姿です。その現状を出発点として、目指すべきアーキテクチャを定義し、それを継続的に適用・検証していくことができます。

導入までの期間と既存環境との統合

アーキテクチャモデルを整備し、基本的なアーキテクチャ検証を開始するまでには、通常数週間程度かかります。組織全体への定着には数か月、十分に成熟した運用に至るには数四半期を要します。しかし、そのどちらを待つ必要もありません。価値は導入の初期段階から着実に積み上がり始めます。

検証基盤は、現在お使いの IDE、CI/CD パイプライン、ツールチェーンと連携して機能します。すでに静的解析を導入している組織であれば、アーキテクチャ検証を追加することで、その価値をさらに高めることができます。

一方で、最も大きな投資となるのはツールそのものではなく、**組織としての取り組み**です。導入を推進するための経営層やマネジメントのコミットメント、そして開発プロセスを見直し、新しいワークフローを定着させようとするチーム全体の取り組みが不可欠です。ツールへの投資は、それによって削減できるエンジニアリング工数と比べれば決して大きなものではありません。そして、適切な検証基盤を構築すれば、その投資は継続的な価値として確実に回収できます。

この選択が本当に左右するもの

ソフトウェア品質に関する最も深刻な問題は、突然表面化するものではありません。少しずつ蓄積していくものです。

その結果として現れるのは、かつては容易に変更できていたコードベースが、いつの間にか手を加えにくいものになっている状況です。以前は計画どおりに進められていた認証作業が、今では場当たり的な対応に追われるようになります。かつてはベテランエンジニアが全体像を把握していたアーキテクチャも、やがて誰一人として完全には説明できないものへと変わっていきます。

AI は、こうした変化をさらに加速させます。コード量は、人間がレビューできる速度を超えて増え続けます。アーキテクチャと実装の乖離は誰にも気付かれないまま進行し、トレーサビリティもスプリントを重ねるたびに実装からさらに遅れていきます。

安全性が重視されるソフトウェア開発において、検証基盤の整備を先送りすることで経営者や開発責任者が負うリスクは、単にダッシュボードの指標が悪化することではありません。ある時点から、チームは新しい機能を開発するよりも、AI が生み出した複雑さを解きほぐすことに、より多くの時間を費やすようになります。リリースのたびに必要となる検証作業は増え続け、監査では、本来もっと早い段階で解消されているべきトレーサビリティの欠落が見つかります。そして、本来であればコミット時点で検出されるべきアーキテクチャ境界の違反が、本番環境で初めて明らかになります。

これらは仮定の話ではありません。AI の開発スピードが検証基盤の能力を上回ったときに、必然的に生じる結果なのです。

このガイドで紹介した機能は、いずれも現在利用可能であり、数週間で運用を開始できます。問われているのは、コードベースがまだ管理できるうちに基盤を構築するのか、それとも、管理できなくなってから取り組むのかということです。

このガイドについて

このガイドについて

このガイドは、Qt Group のソフトウェア品質ソリューションチームが、自動車、医療機器、産業オートメーション、航空宇宙といった、安全性が重視されるソフトウェア開発の分野で、長年にわたり検証基盤の構築を支援してきた経験をもとに作成しました。

本ガイドで紹介した考え方は、Axivion Suite によって実現できます。Axivion Suite は、ソフトウェア品質を「約束する」のではなく、実証することが求められる組織のために設計された、アーキテクチャ検証および静的コード解析ソリューションです。

現在の検証基盤が AI 時代の開発要件にどの程度対応できているかを評価したい場合は、ぜひ当社までご相談ください。技術コンサルテーションを通じて、お客様の状況に応じたご提案をいたします。

さらに詳しく知る

- ebook:AIを活用したソフトウェア開発を正しく進めるために
- ebook: 世界で最も信頼性の高いソフトウェアを構築する: リーダーのためのプレイブック

技術コンサルテーションの予約

- 予約・お問い合わせ
- Axivion Suite について

まとめ

まとめ

AI は開発基盤を増幅します。 強固な基盤は開発を加速させます。一方、脆弱な基盤は、より速いペースで劣化していきます。中立という選択肢はありません。

「AI の導入か、安全性か」という二者択一は誤りです。 選ぶべきなのは、その両方を実現できる基盤を構築することです。

信頼できるソフトウェア基盤は、組織として備えるべき能力です。 常に実装と整合したアーキテクチャ、継続的検証、そして明確な責任分担。品質は、個人の努力ではなく、基盤の設計そのものから生み出されます。

脆弱な基盤には、4つの警告サインがあります。 検証待ちが増え続けている。変更のたびに全面的な再検証が必要になる。規格適合を示すエビデンスを毎回作り直さなければならない。アーキテクチャがコードではなく、人の記憶の中にしか存在していない。

信頼できるソフトウェア基盤は、4つの機能によって支えられています。 アーキテクチャ検証、高度な静的解析、継続的な規格適合を示すエビデンス、そして開発フローに統合された検証です。

優位性は複利的に積み上がります。 早い段階での投資は、あらゆるリリース、あらゆる製品、そしてこれからの年月を通じて、継続的な価値を生み出します。

Qt Group

Qt Group について

Qt Groupの顧客は、180か国以上、70を超える業界にまたがっています。
Qt Groupには約1,100名の従業員が在籍し、2025年の純売上高は2億1,630万ユーロでした。

↗ 詳しくはwww.qt.ioをご覧ください。