DevOpsからCQ by AIへ
──生成AIによるQA判断の自動化と、失敗から学び続けるループ構築への第一歩

From DevOps to CQ by AI
Automating QA Judgments with Generative AI and a First Step toward a Loop that Keeps Learning from Failures

株式会社カカクコム　食べログカンパニー 開発本部 品質管理部 QAチーム
QA Team, Quality Management Department, Development Division, Tabelog Company, Kakaku.com, Inc.
○助川 隆俊　　黎 園園[1)]　　荻野 恒太郎[2)]
○Takashi Sukegawa　　Yuanyuan Li[1)]　　Kotaro Ogino[2)]

Abstract　DevOps and continuous delivery automated deterministic processing, but QA's probabilistic judgment remained manual. We are working toward Continuous Quality by AI, in which AI carries these judgments through to a release decision without manual QA. This paper reports that test design, automated test coding, and the repair of the defects behind failing tests now run as one chain with no human judgment in between. Each was validated in a separate project; in the third, an agent loop raised an E2E suite of 196 cases from 42.9% to 100%. What held the chain together was not model capability but structuring each judgment's inputs, splitting work between the LLM and deterministic processing, and building the verification items ourselves.

1.　はじめに

近年のソフトウェア開発では、コード変更量とリリース頻度が増し続けている。DevOps および継続的デリバリー（CD）は、事前に定義された決定論的な処理を自動化することで、開発速度と品質の両立を支えてきた[1]。しかしQAの工程には、リスク分析、テスト設計、自動テストコーディング、コードレビュー、欠陥箇所の特定、再発防止策の立案、リリース可否判断など、確率論的な判断を伴う作業が残されている。何をどこまで確認するか、何をもって正しいとするかをその都度決めるため事前に手続きを定義できず、自動化されにくい。開発速度が上がるほど、ここが制約になる。

我々はこれらの確率論的判断を生成AIに担わせ、手動QAを経ずにリリース可否まで到達できる状態を目指している。この状態を Continuous Quality by AI（以下 CQ by AI）と呼び、チームの長期目標に掲げている。本論文が報告するのは、その途上で到達した次の状態である。

テスト設計から、自動テストの実装、実行、故障の修復までが、人の判断を挟まずに一連で回るようになった。

ここでいう故障の修復とは、自動テストが検出した故障を起点に、その対象である開発対象（プロダクトコード）に含まれる欠陥を修正することを指す。自動テスト自体を修正することではない。

貢献は3点である。第一に、テスト設計、自動テストコーディング、欠陥箇所の特定と修正という3つの確率論的判断について、AIが判断を担える状態になったことを実案件で確認した（3章、4章、5章）。第二に、この3つをそれぞれ別の案件で検証し、各章の出力が次章の入力になる形で繋がることを示した（2章）。第三に、3つに共通していた条件と、まだ担えていない判断との差を述べる（6章）。なお3章と4章の一部は技術ブログ[2][3]で公開済みである。

2.　確率論的判断と本論文の範囲

2.1　QA工程における判断の整理

QAの工程は、JSTQB[4]のテストプロセスではテスト計画、テスト分析、テスト設計、テスト実装、テスト実行、テスト完了に分けられ、これに実装段階の静的テストと、リリース後に見つかった故障や欠陥から学びを返すフィードバックが加わる。

各工程の作業は、決定論的処理と確率論的判断に区分できる。決定論的処理とは、テストの実行、実行結果の合否判定、静的解析ルールの適用など、事前に手続きを定義できるもので、CDおよび既存

株式会社カカクコム　食べログカンパニー 開発本部 品質管理部 QAチーム　チームリーダー
Team Leader, QA Team, Quality Management Department, Development Division, Tabelog Company, Kakaku.com, Inc.
〒150-0022　東京都渋谷区恵比寿南3丁目5番7号　デジタルゲートビル　TEL: 03-5725-4554　e-mail: sukegawa_takashi2@kakaku.com
Digital Gate Bldg., 3-5-7 Ebisu-Minami, Shibuya-ku, Tokyo 150-0022, Japan
1) 同 AI4QAチーム　テックリード（Tech Lead, AI4QA Team）　　2) 同 品質管理部　部長（General Manager, Quality Management Department）
キーワード：DevOps、生成AI、自動テスト、テスト設計、確率論的判断、フィードバックループ

の基盤ですでに自動化されている。一方の確率論的判断は1章で挙げた各作業である。DevOps／CDが自動化したのは前者であり、後者は人の手に残った。本研究が対象とするのはこの後者である。

2.2 本論文の範囲

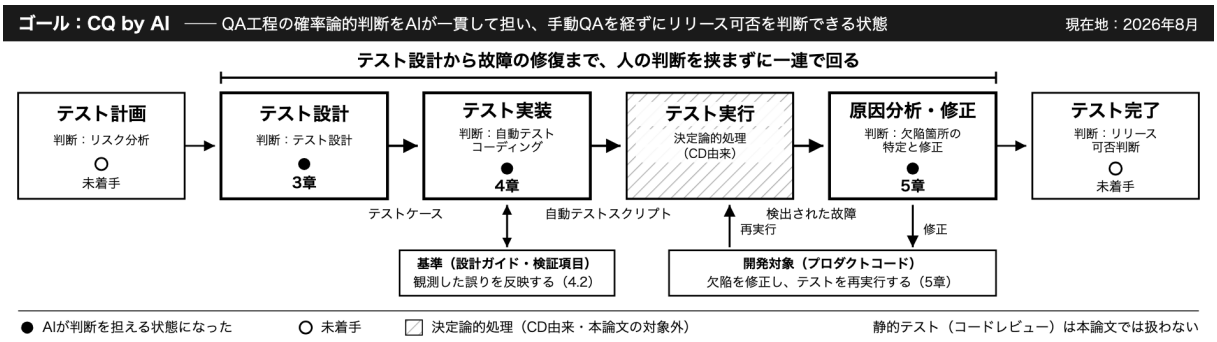

図1　QA工程の確率論的判断と、本論文で到達した範囲

図1に、確率論的判断を工程の順に並べ、どの判断をAIが担える状態になったかを示す。担えるようになったのは、テスト設計（3章）、自動テストコーディング（4章）、欠陥箇所の特定と修正（5章）の3つである。入口のリスク分析と出口のリリース可否判断は未着手であり、静的テストにあたるコードレビューは本論文では扱わない。

3つのあいだにあるテスト実行は決定論的処理で、CD由来の基盤がすでに自動化している。確率論的判断が3つ揃い、そのあいだを決定論的処理が繋いだ結果、テスト設計から故障の修復までが人の判断を挟まずに一連で回る。受け渡されるのは、テストケース、自動テストスクリプト、そしてテストが検出した故障である。

図の下段には、3つの判断が書き換える対象を2つ置いた。1つは基準で、AIに渡す自動テスト設計ガイドと検証項目を指す。AIによるコード実装中に観測した誤りをここへ反映し、次の生成に効かせる（4.2）。もう1つは開発対象で、故障の原因である欠陥を修正したうえでテストを再実行する（5章）。前者はテスト実装の中で閉じ、後者はテスト実行へ戻る。

なお、3つの判断はそれぞれ別の案件で検証したものである。3章は既存機能の改修案件、4章はメール取り込み機能の拡張案件、5章は販売管理システムの新機能開発プロジェクトを対象とした。1つの案件で端から端まで通した検証は行っていない。

3. テスト設計判断のAI化

本章が扱うのは、JSTQBのテスト設計工程における判断のうち、次の2つである。1つ目は、既存機能の改修案件において、用意するテストケースを、過去のテストケースをそのまま流用するもの、テスト対象のプロダクトの改修内容に合わせて過去のテストケースを修正するもの、新たにテストケースを作成するものの3つに振り分ける判断である。2つ目は、それぞれのテストケースで何を照合すればテスト合格となるかの判定ルールを決める判断である。対象は2025年6月の既存機能改修案件で、取り組みの一部は技術ブログ[2]で公開済みである。

3.1 2つの判断に必要な情報

テストケースを3つに振り分けるには、過去にどのテストケースがあるかと、そのうちどれがビジネス影響に直結する機能のテストかが分かっている必要がある。テスト対象のプロダクトの改修が影響する範囲のうち、ビジネス影響に直結する機能については過去のテストケースをそのまま流用し、仕様が変わった箇所については過去のテストケースを修正するためである。テスト合格かの判定ルールを決めるには、過去のテストケースがそれぞれ何を照合していたかが必要である。

しかし、過去のテストケースは案件ごとに表計算ファイルへ分かれており、ビジネス影響に直結する機能のテストケースを検索できなかった。それぞれのテストケースが何を照合していたかも、表計算ファイルを開いて読むまで分からなかった。そのため、どのテストケースを流用しどのテストケースを修正するかも、何を照合すればテスト合格とするかも、実際にはベテランのQAエンジニアに聞くしかなかった。テスト設計の判断をそのまま生成AIに任せても具体性のないテストケースしか出てこなかったのは、この情報が渡っていなかったためである。

3.2　テストケースの判定ルールの構造化

そこで過去のテストケースをJSON形式でGitに集約し、ソースコードと同様に一元管理する形に改めた。本論文ではこれをQAナレッジと呼ぶ。QAナレッジには、過去のテストケースのうちビジネス影響に直結する機能のテストケースが集約されており、テストケースごとにテスト合格かの判定ルールが明確になるよう、フィールドとして明示している。

テスト合格かの判定ルールにあたるのが確認内容のフィールドである。飲食店のネット予約の登録であれば、予約完了画面で照合すべき日付、時間、人数、コース、キャンセルポリシーなど9項目が入る。テストパターンのフィールドには、予約の方法やキャンセルポリシーの有無など、掛け合わせて確認すべき6つの条件が入る。テスト合格かの判定ルールを修正するときに、確認すべき条件の組み合わせを引くために使う。

3.3　2つの判断のワークフロー

そのうえで、テストケースの振り分けと、テスト合格かの判定ルールを決める判断を、生成AIのワークフローとして実装した。

第一は流用である。テスト対象のプロダクトの改修が影響する範囲のうち、ビジネス影響に直結する機能については、対応する過去のテストケースをQAナレッジから検索してそのまま使う。3.2でビジネス影響に直結する機能のテストケースを集約したため、改修された機能から決定論的に検索できる。テスト合格かの判定ルールも、過去のテストケースのものをそのまま使う。

第二は修正である。テスト対象のプロダクトの改修で仕様が変わった箇所については、対応する過去のテストケースを検索したうえで、テスト合格かの判定ルールを書き換える。QAナレッジのテストパターンを入力に、仕様が変わった後にどの条件の組み合わせを確認すべきかをデシジョンテーブル技法で展開し、その組み合わせに合わせて確認内容のフィールドを書き換える。QAエンジニアが頭の中で行っていた「テスト対象のプロダクトの改修箇所に合わせて過去のテストケースを修正する」という手順を、そのまま置き換えたものである。

第三は新規である。仕様書やデザイン、設計書から新たにテストケースを作成することにあたるが、本稿執筆時点では稼働していない。

3.4　結果

本案件ではテストケースを432件生成した。うち89%（384件）が、テスト対象のプロダクトの改修内容に合わせて過去のテストケースを修正したもので、残り11%（48件）は過去のテストケースをそのまま流用したものである。新たにテストケースを作成する振り分けは稼働しておらず、0件である。

この結果から、既存機能の改修が中心の案件の範囲では、テストケースの振り分けと、テスト合格かの判定ルールを決める2つの判断を生成AIが担える状態になったと考えている。この種の案件では、振り分けのほとんどが過去のテストケースの修正に収束するためである。

あわせて、すべて手動でテストケースを作成した場合の想定工数25人日に対し実績は19.8人日で、5.2人日を削減した。ただし比較対象は実績ではなく想定値である。

4.　自動テストコーディング判断のAI化

本章が扱うのは、JSTQBのテスト実装工程における判断、すなわちテストケースを自動テストスクリプトに落とす判断である。対象は2025年10月の既存機能の拡張案件で、自動テストには Cucumber、Selenium、PageObject モデルを用いる。

4.1　自動テストコーディングにおける2つの判断

テストケースを自動テストスクリプトに落とす作業には、2つの判断がある。1つは設計判断で、コードを書く前に、どのPageObjectを使うか、既存のStepで足りるか、どのセレクタを指すかを決める。もう1つはレビュー判断で、書き上がった自動テストスクリプトに誤りがないかを点検する。

4.2　生成前の設計ガイドと、生成後の検証項目

自動テストスクリプトの設計判断には、自動テスト設計ガイドを、コードを書き始める前に渡す。4.1で挙げた3つの問いのそれぞれについて、どう決めるかを定めたものである。どのPageObjectを使うかは、画面操作をPageObjectとComponentのメソッドに置き、Step Definitionから直接呼ばない。既存

のStepで足りるかは、新しいメソッドを追加する前に、既存のメソッドで足りないかを `grep` で確かめる。どのセレクタを指すかは、HTMLファイルを読んで要素の実在を確かめ、推測で書かない。生成された自動テストスクリプトのレビュー判断には、検証項目を、生成の後に渡す。生成された自動テストスクリプトのどこを点検するかを13項目に定めたもので、生成AI自身に点検させる（表1）[3]。

自動テスト設計ガイドと検証項目は、自動テストと同じGitリポジトリに置いたMarkdownのファイル1つに収めている。コミット前のレビューを起動するコマンドの定義がこのファイルを参照しており、コミットの前にこのコマンドを起動して13項目を点検する。

表1　自動テストスクリプトの検証項目

No	検証項目	No	検証項目
1	メソッド参照検証	8	Component責任分離確認
2	XPath/CSSセレクター検証	9	Modal Component条件初期化確認
3	XPathコンテキスト分析	10	Modal Component職責分離確認
4	TypeScript編成検証	11	既存ツールメソッド活用確認
5	パラメーター化転送検証	12	Feature Step完全性検証
6	既存Step重複チェック	13	タイムアウト設定適切性検証
7	既存メソッド機能満足度検証		

項目の多くは、人がコードベースで確かめていたことを、生成後に点検する形に置き換えたものである。メソッドやセレクタが実在するか（1、2）、既存のStepやメソッドで足りないか（6、7）、Componentの責務が分かれているか（8、9、10）である。

この13の検証項目は、初版から2回改訂されており、3つのバージョンがある。初版は5項目で、2025年9月に定めた。11月のFlakyテストの修正で12項目、同月の欠陥修正で13項目へ拡張した。本章の対象案件を実施した10月の時点では初版の5項目だった。拡張のきっかけは、いずれも自動テストスクリプトに見つかった誤りである。たとえば13番目の「タイムアウト設定適切性検証」は、生成された自動テストスクリプトに、開発対象の欠陥を隠してしまう長いタイムアウトが指定されていたのを受けて追加した項目である。観測した誤りをAIに渡す基準へ反映し、同じ誤りが次の生成で繰り返されないようにしている。改訂はいずれも案件の実装または欠陥修正の作業の中で行われており、別立ての改善活動としては行っていない。自動テスト設計ガイドも、同じ期間に複数回改訂している。

4.3　結果

生成精度は、生成AIが出力したまま人手で修正されなかった行数を、追加された総行数で除した値である。表2に、PageObject層とStep Definition層を合算した生成精度を示す。本章の対象案件を層ごとに分けると、PageObject層が97.3%（805行）、Step Definition層が92.6%（1,033行）である。

13の検証項目を確立した2025年11月以降の3案件も、同じ定義で調べた。13の検証項目のもとで実施した案件でも同じ水準が出るかを見るためである。この期間の規模あるテスト作成は14件で、いずれも13の検証項目のもとで実施されており、うち生成物の切り分けが記録に残る3案件を対象とした。

表2　生成精度（PageObject層とStep Definition層の合算）

案件	母数	生成精度
本章の対象案件（2025年10月）	1,838行	94.7%
2026年6月	506行	98.4%
2026年7月（a）	837行	95.5%
2026年7月（b）	2,325行	98.7%
2026年の3案件　計	3,668行	97.9%

2026年の3案件で人が書き換えたのは3,668行のうち77行である。39行はlintやprettierによる整形で、残る38行はレビューで実装の仕方を見直して元に戻したものである。

5.　欠陥箇所の特定と修正のAI化

本章が扱うのは、自動テストが検出した故障から欠陥箇所を特定し、開発対象に含まれる欠陥の修正方法を決める判断である。修正するのは自動テストではなく、テストの対象である開発対象（プロダクトコード）である。

この判断をAIに担わせる取り組みは、大規模なリポジトリを対象とした事例が報告されている[5]。同事例は、AIに渡す情報を静的解析やテスト実行の結果で補うほど修正の成功率が上がることを示し、モデルの能力ではなく渡す情報の設計が結果を分けるとしている。本章の対象はE2Eテストであり、失敗の原因を機械的に絞り込む手段が使えない点で条件が異なる。

5.1　実験の目的と方法

自動テストが検出した故障を起点に、人が内容を見て判断しなくてもAIが開発対象の欠陥を正しく修正できるかを確かめた。対象は、開発の過程で100件以上の欠陥が見つかったプロジェクトである。E2Eテスト196本のうち112本が失敗している状態を出発点とし、以降は5.2に示す自己修復ループだけを回した。

人が欠陥箇所を判断するとき見ているのは、期待値と実際値がどの項目で異なるかと、その値やエラーメッセージが開発対象のどのコードに現れるかである。AIに渡すのもこの2つで、作り方は5.2で述べる。

ループは1周ごとに止まるようにしてあり、1周が終わるたびに人が結果を確認する。ループの仕組みに改善すべきところがあれば改善してから、次の周を実行する。人が確認するのはループの仕組みで、AIが行った個々の修正の内容には立ち入らない。

5.2　ループの構成

図2にループの1周を示す。AからFまではすべてAIが実行する。このうちC診断、D修復計画、E実装の修正の3つをLLMが判断し、BとFは決まった処理を機械的に行う。

Bでは、テスト結果から期待値と実際値の不一致を項目ごとに全件取り出し、不足・余剰・順序違いに分ける。Cでは、欠陥箇所がどのファイルにあるかを、文言の実在を確認したうえで記録する。確認できなければ空で渡し、推測によるファイル指定は禁じている。DとEは、このBの差分とCの記録を入力に用いる。テストレポートやバグチケットも生成されるが、これらは人が状況を把握するための出力である。

Fでは、修正した対象について単体テストを実行する。あわせて、修正が開発対象の外に及んでいないかを3種のガードで確かめる。単体テストのアサーションを削除していないか、自動生成ファイルをコミットに含めていないか、自動テスト側のリポジトリを書き換えていないかである。いずれかに触れた場合は修正をロールバックし、通ったものだけをコミットする。コミットしたうえでAに戻り、2次検証としてテストを全件再実行する。

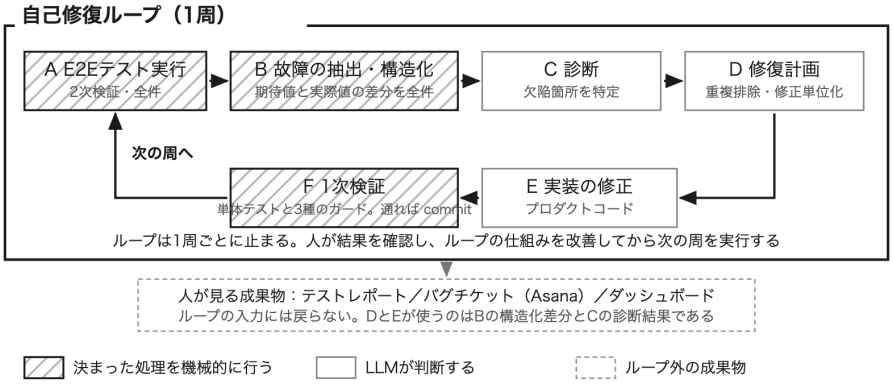

図2　自己修復ループ（1周）

5.3　結果

表3　イテレーションごとの成功数

Iter	1	2	3	4	5	6	7	8	9
成功	84	113	160	169	174	190	194	195	196
失敗	112	83	36	27	22	6	2	1	0
成功率	42.9%	57.7%	81.6%	86.2%	88.8%	96.9%	99.0%	99.5%	100%
前周からの増減	—	+29	+47	+9	+5	+16	+4	+1	+1

表3に9回のイテレーションの推移を示す。成功率はE2Eテスト実行での成功率である。42.9%から始まり、9周目にすべてのテストが成功した。大きく動いたのは2周目から3周目と、5周目から6周目である。

5.4　ループ中に人が改善した箇所

各周のあいだに人が確認し、ループの仕組みを改善した箇所は14件である。表4に、分類ごとの内訳を示す。

表4　人が改善した14件

分類	件数	内訳
診断・分析の質	3	①故障の圧縮バイアス　②根本原因の捏造　③情報の切り詰め
コード修正の逸脱	3	④検証前コミットによる大規模退行　⑤期待値の反転　⑥テスト側リポジトリの書き換え
運用・状態管理	5	⑦偽収束　⑧ロックの残留　⑨並行実行の競合　⑩起票の重複　⑪実行監視の分割
設定・環境	3	⑫参照パスの誤り　⑬テスト用DBの未作成　⑭ハードコードされた日付

このうち成功率への影響が大きかった3件を述べる。

②根本原因の捏造。手がかりの乏しい故障に対し、診断を担うモデルが実在しない共通の根本原因を生成していた。診断を上位モデルの2段構成に変え、文言の実在を確認してから対象ファイルを記録する形に改めたところ、成功数が純増16件伸びた。

③情報の切り詰め。期待値と実際値が13項目食い違った故障で、約800文字ある差分を先頭200文字に切り詰めてAIに渡していたため、先頭2項目しか届かず、残りが直らないままだった。切り詰めをやめ、差分を全件構造化して渡す形に変えた。

⑤期待値の反転。単体テストを通すために既存の期待値を反転させる修正が、86コミット中7件あった。コミット前にアサーションの削除行を検出するガードを設け、検出した場合は実装ごとロールバックする形にした。

6. 考察

6.1 テスト設計から欠陥の修正までが一連で回るようになった

1章で述べた状態を、あらためて具体的に述べる。テスト設計でテストケースを決め、それを自動テストスクリプトに落とし、テストを実行し、故障が出れば欠陥箇所を特定して開発対象を修正する。この一続きを、途中で人が判断を挟まずに繋げられるようになった。

繋がるとは、前の工程の出力がそのまま次の工程の入力になり、人が中身を見て判断してから渡す作業がないことである。テストケースはJSON、自動テストスクリプトはコード、故障は期待値と実際値の構造化された差分で、いずれも次の工程がそのまま読める。あいだにあるテスト実行もCD由来の基盤が決定論的に行うため、工程と工程のつなぎ目に人が読んで解釈し直す作業がない。

これができるようになったのは、3章から5章の3つの取り組みによる。テスト設計ではQAナレッジ、自動テストコーディングでは設計ガイドと13の検証項目、欠陥箇所の特定と修正では故障の構造化とLLMに任せる判断の範囲の限定によって、それぞれの判断をAIに任せられるようになった。

残る人の関与は、4章のレビューによる実装方式の見直しと、5章の各周の確認および仕組みの改善である。いずれも工程のあいだの受け渡しには入っていない。

6.2 なぜAIに判断を任せられたのか

テスト設計、自動テストコーディング、欠陥箇所の特定と修正という3つの判断には、共通していたことが3点ある。

第一に、それぞれの判断に必要な情報を、判断をAIに任せる前に構造化した。AIに判断を任せるには、モデルの能力を上げることより先に、判断の材料を揃えることが必要だったためである。3章では、過去のテストケースを流用・修正・新規の3つに振り分ける判断に、過去にどのテストケースがあるかと、そのうちどのテストケースがビジネス影響に直結する機能のテストかが必要だった。これをQAナレッジへの集約によって満たした。それぞれのテストケースで何を照合すればテスト合格となるかの判定ルールを決める判断には、過去のテストケースが何を照合していたかが必要だった。これをQAナレッジの確認内容のフィールドに明示した。

4章では、どのPageObjectを使い既存のStepで足りるかを決める設計判断と、生成された自動テストスクリプトに誤りがないかを点検するレビュー判断のいずれにも、既存のPageObjectとStepに何があるかと、対象の画面のHTMLの構造が必要だった。設計判断には自動テスト設計ガイドとして生成の前に、レビュー判断には13の検証項目として生成の後に明示した。

5章では、自動テストが検出した故障から欠陥箇所を特定し、開発対象に含まれる欠陥の修正方法を決める判断に、期待値と実際値がどの項目で異なるかが必要だった。これをテストの実行結果から、不足・余剰・順序違いに分けた差分として取り出した。

第二に、LLMには確率論的判断だけを担わせた。判断を要さない作業までLLMに任せると、その作業で誤りが生じ、確率論的判断に集中できないためである。3章では、過去のテストケースをそのまま流用する振り分けを、ビジネス影響に直結する機能のテストケースをQAナレッジに集約することで、改修された機能から決定論的に検索する処理にした。5章では、欠陥箇所の診断、修復計画、実装の修正をLLMに任せ、故障の抽出と構造化、修正後の単体テストの実行とガードは決定論的な処理として実装した。

第三に、AIが何を見て決め、何を点検するのかという項目を人が定めた。項目が定まっていれば、LLMが担うのは各項目をどう確認するかだけになり、確率論的判断に集中できるためである。4章では、どのPageObjectを使うかなどをどう決めるかを自動テスト設計ガイドに定め、生成された自動テ

ストスクリプトのどこを点検するかを13の検証項目に定めた。5章では、修正が開発対象の外に及んでいないかを確かめる3種のガードを定めた。この項目を作り込むのは人の作業である。

6.3 限界

本研究には次の限界がある。

第一に、3つの判断はそれぞれ別の案件で検証したものであり、1つの案件で端から端まで通した検証は行っていない。

第二に、テスト設計判断は既存機能の改修の範囲でしか担えていない。新規の観点を起こす経路は稼働しておらず（0件）、生成精度も測定していない。

第三に、示した数値は単一の技術スタックにおける少数の対象から得た。生成精度は行ベースで測っており機能的な正しさは評価しておらず、5章の実験は1案件・9イテレーションである。統制された比較ではなく実運用の観察にとどまる。

7. おわりに

本論文は、確率論的判断をAIに担わせる取り組みの途上で到達した状態を報告した。テスト設計、自動テストコーディング、欠陥箇所の特定と修正という3つの判断をAIが担えるようになり、前の工程の出力が次の工程の入力になる形で繋がった。ただし3つは別々の案件で検証したものであり、1つの案件を通した実績ではない。

これができたのは、判断に必要な情報を先に構造化し、判断を要さない作業は決定論的な処理にし、検証とレビューの項目を人が作り込んで渡したからである。モデルの能力を上げたのではない。

図1に照らせば、入口のリスク分析と出口のリリース可否判断が未着手のまま残っている。この2つも、判断に必要な情報を構造化し、判断を要さない作業を切り分け、確認すべき項目を人が定めることで担えるようになると考えている。そこまで届けば、DevOpsとCDが決定論的な処理で実現した開発速度と品質の両立を、確率論的判断まで広げられる。それがCQ by AI であり、今回はその第一歩である。

参考文献

[1] 荻野恒太郎、アジャイル/DevOps開発における品質保証と信頼性、日本信頼性学会誌、Vol.42、No.2、pp.71-78、2020

[2] 助川隆俊、AIによる手動QAの自動化：食べログQAチームの挑戦、その第一歩、食べログテックブログ、2025、https://tech-blog.tabelog.com/entry/ai-manual-qa-automation-first-step （2026年8月17日参照）

[3] 黎園園、AIによる手動QAの自動化：自動テストコーディングのAI化でテスト実行工数を52%削減、食べログテックブログ、2025、https://tech-blog.tabelog.com/entry/ai-for-qa-automation-test （2026年8月17日参照）

[4] JSTQB、テスト技術者資格制度 Foundation Level シラバス 日本語版 Ver.4.0、2023

[5] C. Maddila et al.、Agentic Program Repair From Test Failures at Scale: A Neuro-Symbolic Approach With Static Analysis and Test Execution Feedback、IEEE Transactions on Software Engineering、Vol.52、No.8、pp.2446-2462、2026