

Droidkaigi 2026

行政アプリ全画面TalkBack対応を、 デザイナー協業とAIエージェントで 自動化していく実体験

Making an Entire Government App TalkBack-Ready: Automating Accessibility with Designers and AI Agents

GovTech 東京

田澤 健二 Kenji Tazawa

自己紹介 (Self Introduction)

田澤 健二 Kenji Tazawa

所属

**GovTech東京 デジタル事業本部
デジタルサービス開発グループ**

GovTechTokyo Digital Business Division
Digital Service Development Group

役割

東京アプリのAndroidアプリ開発

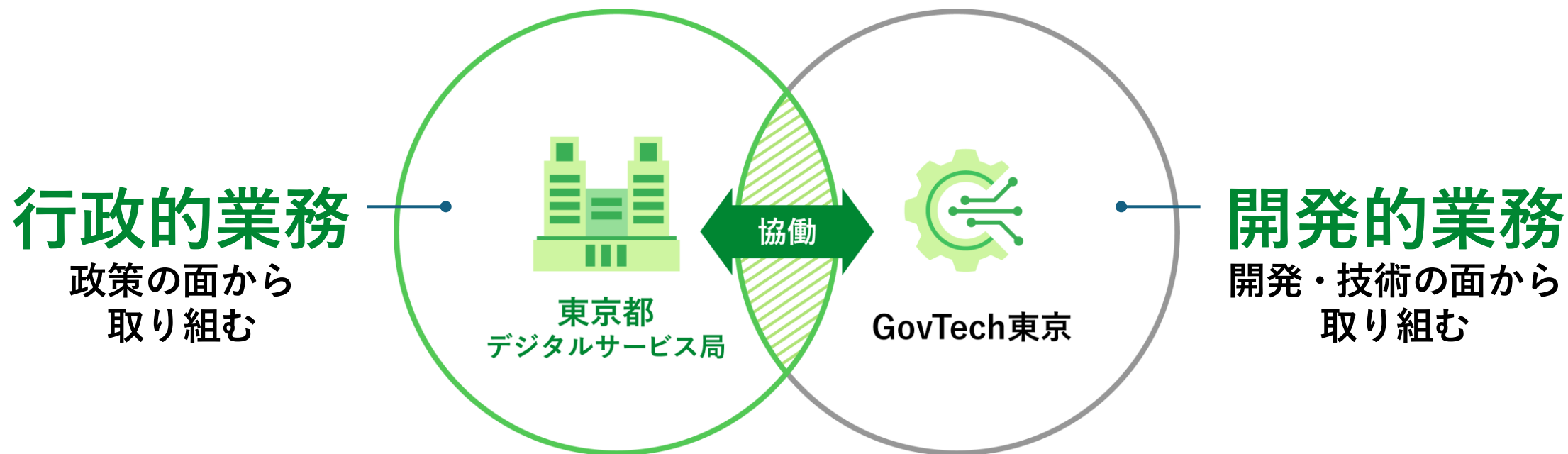
Android App Engineer



GovTech東京とは

行政課題を、テクノロジーで解決する専門家集団

GovTech Tokyo — Specialists Who Solve Public Sector Challenges Through Technology



2023年に東京都の外側に「政策連携団体」（都と一体となって仕事をする団体）として設立
我々以外にも多くのチームがあり、東京都の島や村を含む区市町村のDXを進めています。

東京アプリとは



スマホ一つで、行政とつながる

- 2025年2月17日リリース（640万DL/2026年5月末時点）
- 東京都 と GovTech東京 が連携して開発・運営
- **GovTech東京 で 完全内製化進行中**

今後のサービス展開

1. 都民一人ひとりにあった情報が届く「プッシュ機能」
2. 生成AIが行政手続を分かりやすく案内
3. 簡単に給付が受け取れる
4. 施設入場をスムーズにする「デジタル都民証」

What is the TokyoGov App?

- Released on February 17, 2025 (6.4 million downloads as of the end of May 2026)
- Developed and operated jointly by the Tokyo Metropolitan Government and GovTech Tokyo

はじめに

“ エラーが出ても、何が原因で、どう直せばいいのかわからない。何度やっても、先に進めない。

— 視覚障害のある方の声 (GovTech東京「都民テスター」調査より)

画面読み上げ（スクリーンリーダー）を頼りにしている人にとっては、未対応の画面 が手続き全体を止めてしまうことがある。

Feedback from a tester with visual impairment:

"When an error occurs, I have no idea what caused it or how to fix it. No matter how many times I try, I just can't move forward."

For people who rely on screen readers, an unsupported screen can bring the entire process to a halt.

行政サービスは選べない

民間サービス

使いにくければ、
ユーザーは離れる



ほかに選択肢がある

行政サービス

使いにくくても、
代替となる
サービスがない。



代わりがない

When public services are inaccessible, users are not simply inconvenienced — they are effectively excluded from the programs and entitlements those services provide. Unlike private alternatives, there is no substitute.

**行政は
すべての人にサービスを
届ける責任があります**

Government must ensure
its services are accessible to all.

**その画面、
“読み上げ”に対応していますか？**

Is Your App Screen Reader-Ready?

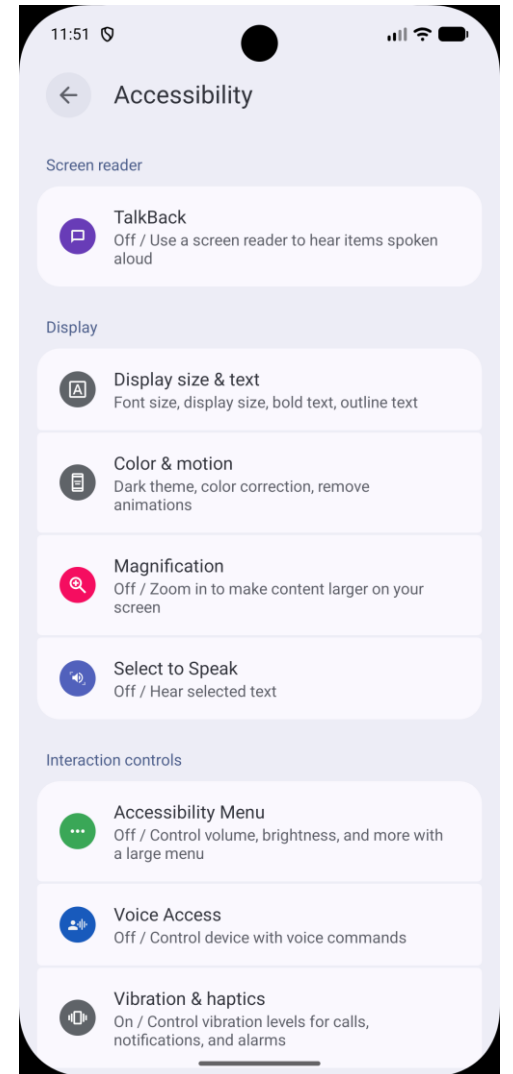
TalkBack対応、取り組んでいますか？

「contentDescription」が null、「一応、何か指定する」になっていませんか？

Is TalkBack support on your application?

アクセシビリティ対応の大まかな種類

- 画面サイズ対応
- フォントサイズ、ディスプレイサイズ対応
- キーボード操作対応
- 音声操作対応
- **TalkBack対応**
- 端末の向き
- 多言語化対応



※ 他にも画像、コントラスト比などのデザイン対応なども存在

今日のゴール

Today's Goal

**TalkBack対応は仕組み化することで
継続的に開発が可能になる、
と持ち帰ってもらうこと**

Make TalkBack support sustainable by building
it into your process.

使用しているツール

- ※ AIエージェントはClaude Codeを使用
- ※ デザインツールはFigmaを使用
- ※ TalkBackの読み上げ内容の管理はFigmaを使用

Tools we use

- * Claude Code as the AI agent
- * Figma as the design tool
- * TalkBack announcement content is managed in Figma

もくじ

1. なぜAIエージェントを用いたワークフローを組んだのか
2. ワークフロー構築時に気をつけたこと
3. デザイナーとAIエージェントとのワークフロー
 - 指示文作成
 - TalkBack実装
 - 実機検証
 - AIエージェントでのバグ紹介
4. まとめ

Index

1. Why we built a workflow around an AI agent
2. What we were careful about when building the workflow
3. The overall workflow between designers and the AI agent
 - Writing the instructions
 - Implementing TalkBack support
 - On-device testing
 - Introducing Bugs Found via the AI Agent
4. Wrap-up

なぜAIエージェントを用いた ワークフローを組んだのか

Why we built a workflow around an AI agent

なぜAIエージェントを用いたワークフローを組んだのか

Why we built a workflow around an AI agent

エンジニア視点

Engineer's perspective

 **TalkBack対応の実装に
精通したエンジニアが不在**

No engineer on the team was
well versed in implementing TalkBack support

なぜAIエージェントを用いたワークフローを組んだのか

Why we built a workflow around an AI agent

エンジニア視点

Engineer's perspective

 読み上げ指示を
semantics propertyに
変換するのは工数がかかる

Translating announcement content into Compose semantics
properties

one case at a time takes a lot of effort

【補足】Semantics Properties ※40種類以上

[Reference] Semantics Properties * more than 40 of them

<code>SemanticsPropertyKey<CollectionInfo></code>	<code>CollectionInfo</code>	Cmn
<code>SemanticsPropertyKey<CollectionItemInfo></code>	<code>CollectionItemInfo</code>	Cmn
<code>SemanticsPropertyKey<ContentDataType></code>	<code>ContentDataType</code>	Cmn
<code>SemanticsPropertyKey<List<String>></code>	<code>ContentDescription</code>	Cmn
<code>SemanticsPropertyKey<ContentType></code>	<code>ContentType</code>	Cmn
<code>SemanticsPropertyKey<Unit></code>	<code>Disabled</code>	Cmn
<code>SemanticsPropertyKey<AnnotatedString></code>	<code>EditableText</code>	Cmn
<code>SemanticsPropertyKey<String></code>	<code>Error</code>	Cmn
<code>SemanticsPropertyKey<FillableData></code>	<code>FillableData</code>	Cmn
<code>SemanticsPropertyKey<Boolean></code>	<code>Focused</code>	Cmn
<code>SemanticsPropertyKey<Unit></code>	<code>Heading</code>	Cmn
<code>SemanticsPropertyKey<Unit></code>	<code>HideFromAccessibility</code>	Cmn
<code>SemanticsPropertyKey<ScrollAxisRange></code>	<code>HorizontalScrollAxisRange</code>	Cmn
<code>SemanticsPropertyKey<Ime></code>	<code>ImeAction</code>	Cmn
<code>SemanticsPropertyKey<AnnotatedString></code>	<code>InputText</code>	Cmn
<code>SemanticsPropertyKey<InputTextSuggestionState></code>	<code>InputTextSuggestionState</code>	Cmn
<code>SemanticsPropertyKey<Unit></code>	<code>InvisibleToUser</code>	Cmn
This property is deprecated. Use `hideFromAccessibility` instead.		
<code>SemanticsPropertyKey<Boolean></code>	<code>IsContainer</code>	Cmn
This property is deprecated. Use `isTraversalGroup` instead.		
<code>SemanticsPropertyKey<Unit></code>	<code>IsDialog</code>	Cmn
<code>SemanticsPropertyKey<Boolean></code>	<code>IsEditable</code>	Cmn
<code>SemanticsPropertyKey<Unit></code>	<code>IsPopup</code>	Cmn
<code>SemanticsPropertyKey<Boolean></code>	<code>IsSensitiveData</code>	Cmn
<code>SemanticsPropertyKey<Boolean></code>	<code>IsShowingTextSubstitution</code>	Cmn
<code>SemanticsPropertyKey<Boolean></code>	<code>IsTraversalGroup</code>	Cmn
<code>SemanticsPropertyKey<Unit></code>	<code>LinkTestMarker</code>	Cmn
Marks a link within a text node (a link is represented by a <code>androidx.compose.ui.text.LinkAnnotation</code>) for identification during automated testing.		
<code>SemanticsPropertyKey<LiveRegionMode></code>	<code>LiveRegion</code>	Cmn

<https://developer.android.com/reference/kotlin/androidx/compose/ui/semantics/SemanticsProperties>

なぜAIエージェントを用いたワークフローを組んだのか

Why we built a workflow around an AI agent

エンジニア視点

Engineer's perspective

 **約30画面と35個の共通コンポーネントを
複数名で対応すると
実装差異やコードの品質低下が起こりうる**

With about 30 screens and more than 35 shared components being handled by multiple people, discrepancies in implementation and a decline in code quality can occur.

なぜAIエージェントを用いたワークフローを組んだのか

Why we built a workflow around an AI agent

エンジニア視点

Engineer's perspective

実装差異をなくし、品質を担保したい

We wanted to eliminate implementation drift and guarantee quality

なぜAIエージェントを用いたワークフローを組んだのか

Why we built a workflow around an AI agent

デザイナー視点

Designer's perspective

 画面のコンポーネント1つ1つに
読み上げ内容の指示を書くのは、負担が大きい

Writing announcement instructions for every single
component on a screen is a heavy burden

なぜAIエージェントを用いたワークフローを組んだのか

Why we built a workflow around an AI agent

デザイナー視点

Designer's perspective

 当初から約30画面が想定されており、
全て手動で対応するにはミスが発生しやすい

About 30 screens were planned from the start,
and doing all of them by hand invites mistakes

なぜAIエージェントを用いたワークフローを組んだのか

Why we built a workflow around an AI agent

デザイナー視点

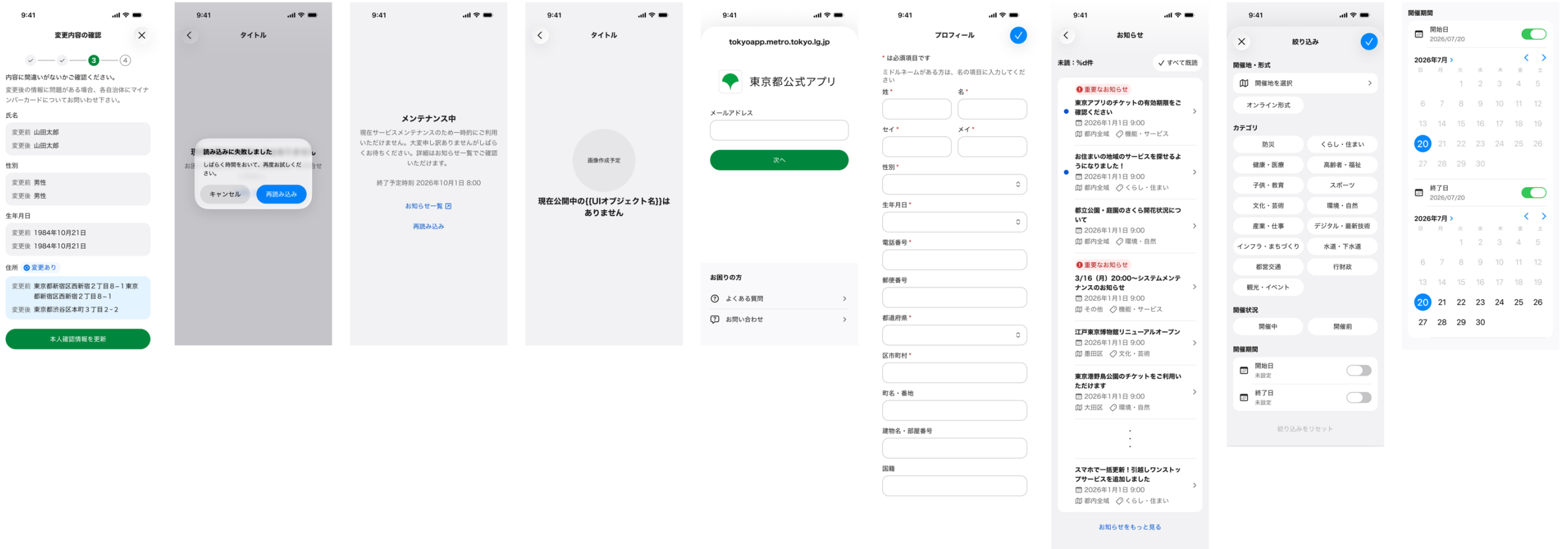
Designer's perspective

デザイナーの工数、負荷を下げたい

We wanted to cut the designer's workload and effort

30画面のイメージ（一部）

What 30 screens looks like



工数、実装差異、 バグの発生、運用負荷の観点から 品質として危うい状態になることを想定

Between the effort, the implementation drift,
the bugs and the operational load, we expected quality
to end up on shaky ground

もくじ

1. なぜAIエージェントを用いたワークフローを組んだのか
2. ワークフロー構築時に気をつけたこと
3. デザイナーとAIエージェントとのワークフロー
 - 指示文作成
 - TalkBack実装
 - 実機検証
 - AIエージェントでのバグ紹介
4. まとめ

Index

1. Why we built a workflow around an AI agent
2. What we were careful about when building the workflow
3. The overall workflow between designers and the AI agent
 - Writing the instructions
 - Implementing TalkBack support
 - On-device testing
 - Introducing Bugs Found via the AI Agent
4. Wrap-up

ワークフローを構築する際に 気をつけたこと

What we were careful about when building the workflow

気をつけたこと 1

What we were careful about

TalkBack実装の第一歩

**アプリ固有の共通componentのみ対象
小さくフローを組みはじめた**

We started the flow small, covering only the app's own shared components first

気をつけたこと 1

What we were careful about

小さく始めることにより

実装時のバグ、エラーを
特定・改善しやすいサイズに

Starting small let us cover every component pattern and keep the scope

small enough to pinpoint and fix the causes of errors.

気をつけたこと 1

What we were careful about

共通component単位なら

様々なUIパターンを効率良く検証可能

画面単位だと暫く扱わないcomponentも先回りで対応可能

Working per shared component lets you verify many patterns efficiently

Going screen by screen leaves some components untouched

- Text
- Button
- List
- TextField
- Dialog
- Etc..

気をつけたこと 1

What we were careful about

画面単位での実装は

**共通componentで品質が担保できてから
とした。**

We decided to move to screen-by-screen implementation only after quality could be guaranteed in the common components.

気をつけたこと2

What we were careful about

**AIによる自動実装は、
100%に近い精度を目指さなかった**

We did not aim for near-100% accuracy from AI-driven implementation

気をつけたこと2

What we were careful about

スピードを出して精度を上げていくには実コードに反映しやすいレベルでの指示文が必要になり情報量が増える。

そのため指示文の情報が多くなりシンプルではなくなる。

Raising accuracy quickly requires instructions detailed enough to map straight onto real code, which means more information.

That means maintaining the instructions, and they stop being simple.

気をつけたこと2

What we were careful about

**“指示文”をレビューするデザイナーの
負荷が上がり、
工数削減を含めた当初の目的から考えると
本末転倒になりえる**

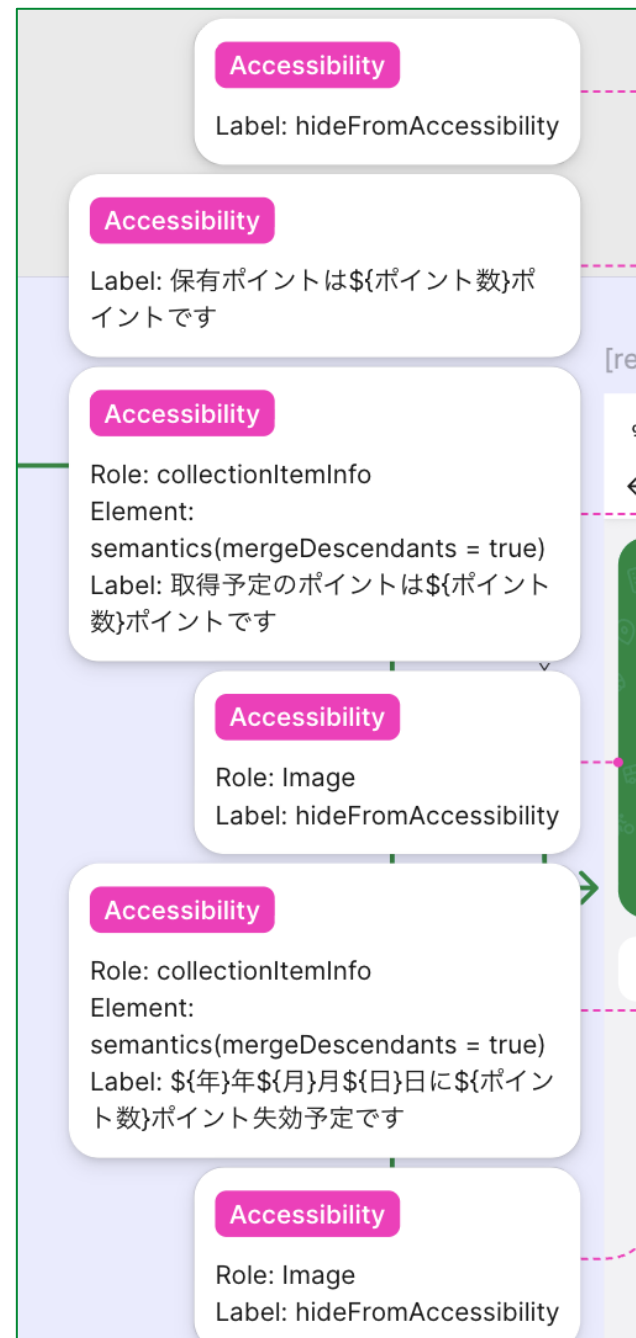
That raises the review load on the designer, which
defeats our original purpose of reducing effort in the first place

補足：指示文

Reference: about Instruction text'

指示文とは、AIエージェントが解釈できる形式にしたTalkBack実装の仕様。Roleの指定や読み上げ内容の指示をまとめたもの。

Instruction text' refers to the specification that AI agents or humans reference when implementing TalkBack. Specifically, it's text that specifies things such as the role and the content to be read aloud.



もくじ

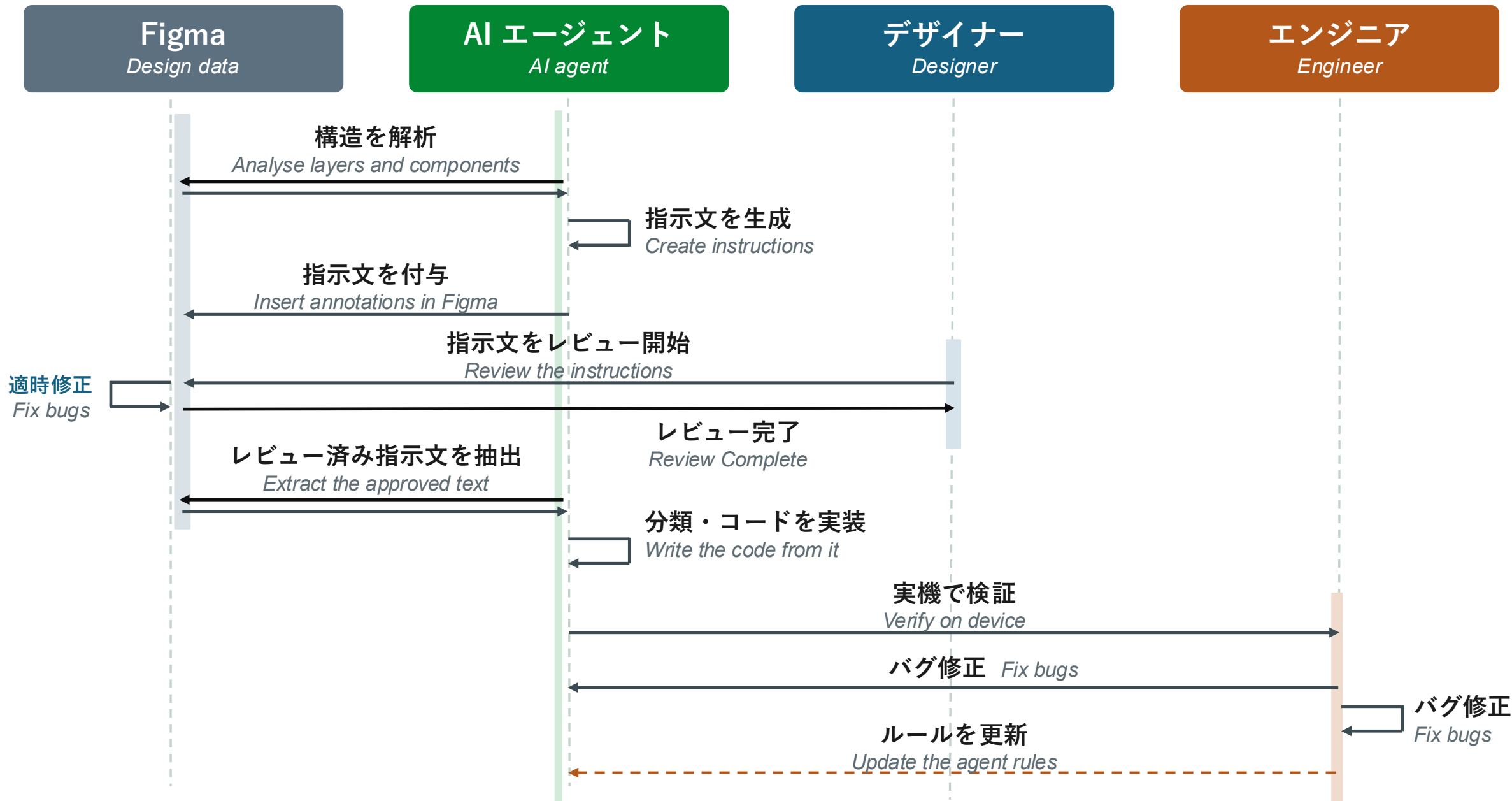
1. なぜAIエージェントを用いたワークフローを組んだのか
2. ワークフロー構築時に気をつけたこと
3. デザイナーとAIエージェントとのワークフロー
 - 指示文作成
 - TalkBack実装
 - 実機検証
 - AIエージェントでのバグ紹介
4. まとめ

Index

1. Why we built a workflow around an AI agent
2. What we were careful about when building the workflow
3. The overall workflow between designers and the AI agent
 - Writing the instructions
 - Implementing TalkBack support
 - On-device testing
 - Introducing Bugs Found via the AI Agent
4. Wrap-up

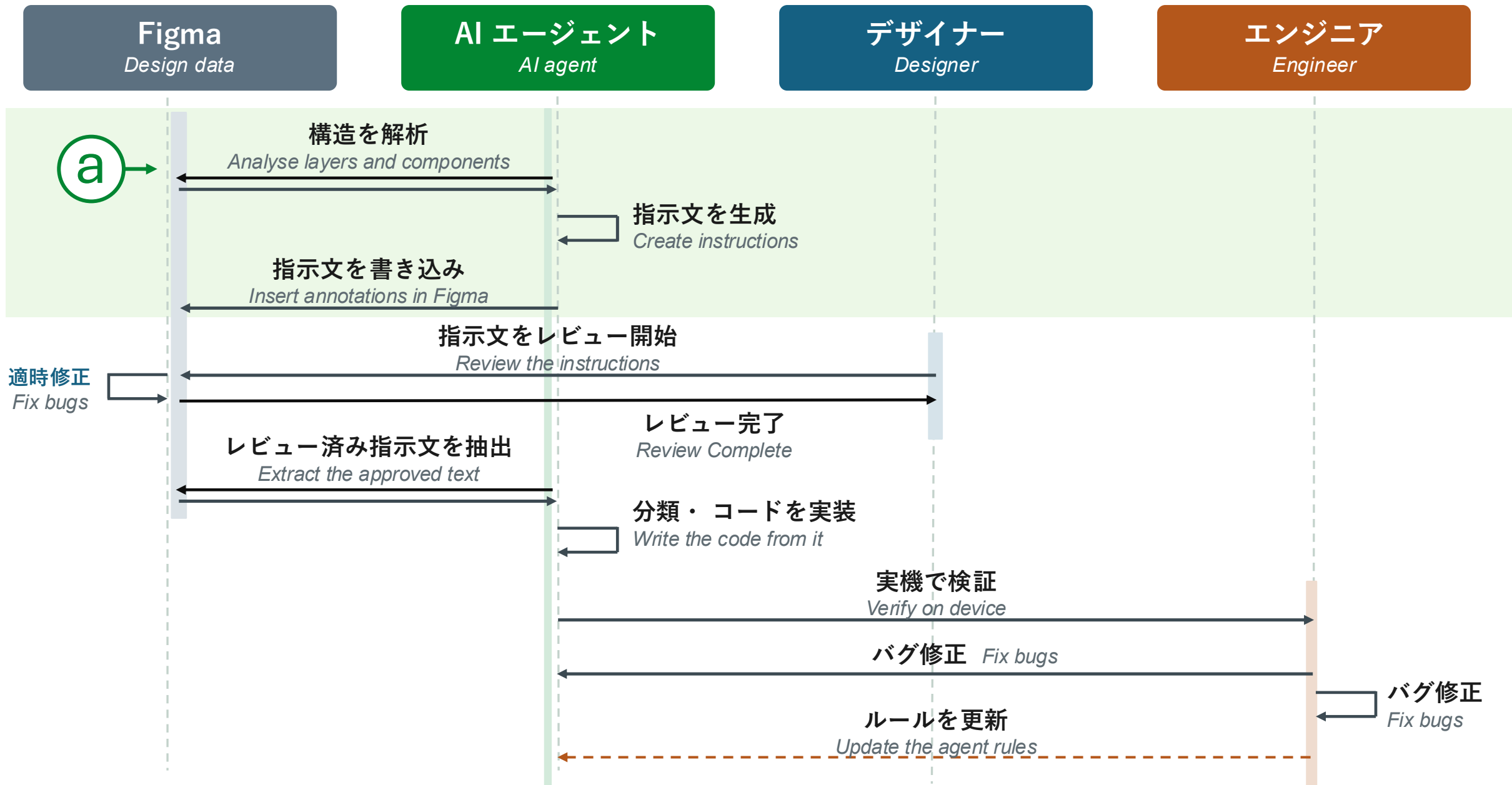
デザイナーとAIエージェントとの ワークフロー全体像

The overall workflow between designers and the AI agent



指示文作成 - AIエージェント

Writing the instructions - AI agent



a. 構造の解析

Analyze the structure

- **Figma MCPサーバー経由で、対象デザインのFigma URLからget_metadata でノード構造を把握**
- **各画面内の全componentを網羅的に洗い出し**
- **componentの詳細情報をget_design_context で取得**
- Grasp the node structure from a Figma URL via get_metadata on the Figma MCP server
- Exhaustively enumerate every component on each screen
- Fetch each component's details with get_design_context

気付き

What we noticed

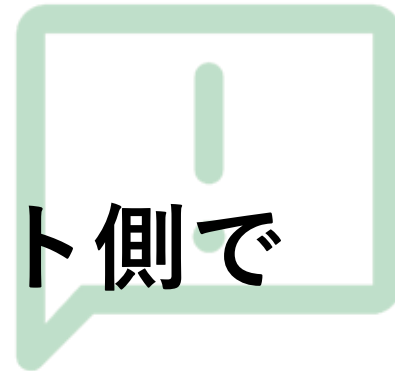
**Figmaのレイヤーが深くなるとAIエージェント側で
UIの構造を把握するのに時間がかかる。**

もしくは、

途中で探索処理を止めてしまうことがあった。

When the Figma layers get deep, the AI agent takes a long time to understand the UI structure.

Sometimes it stopped exploring partway through.



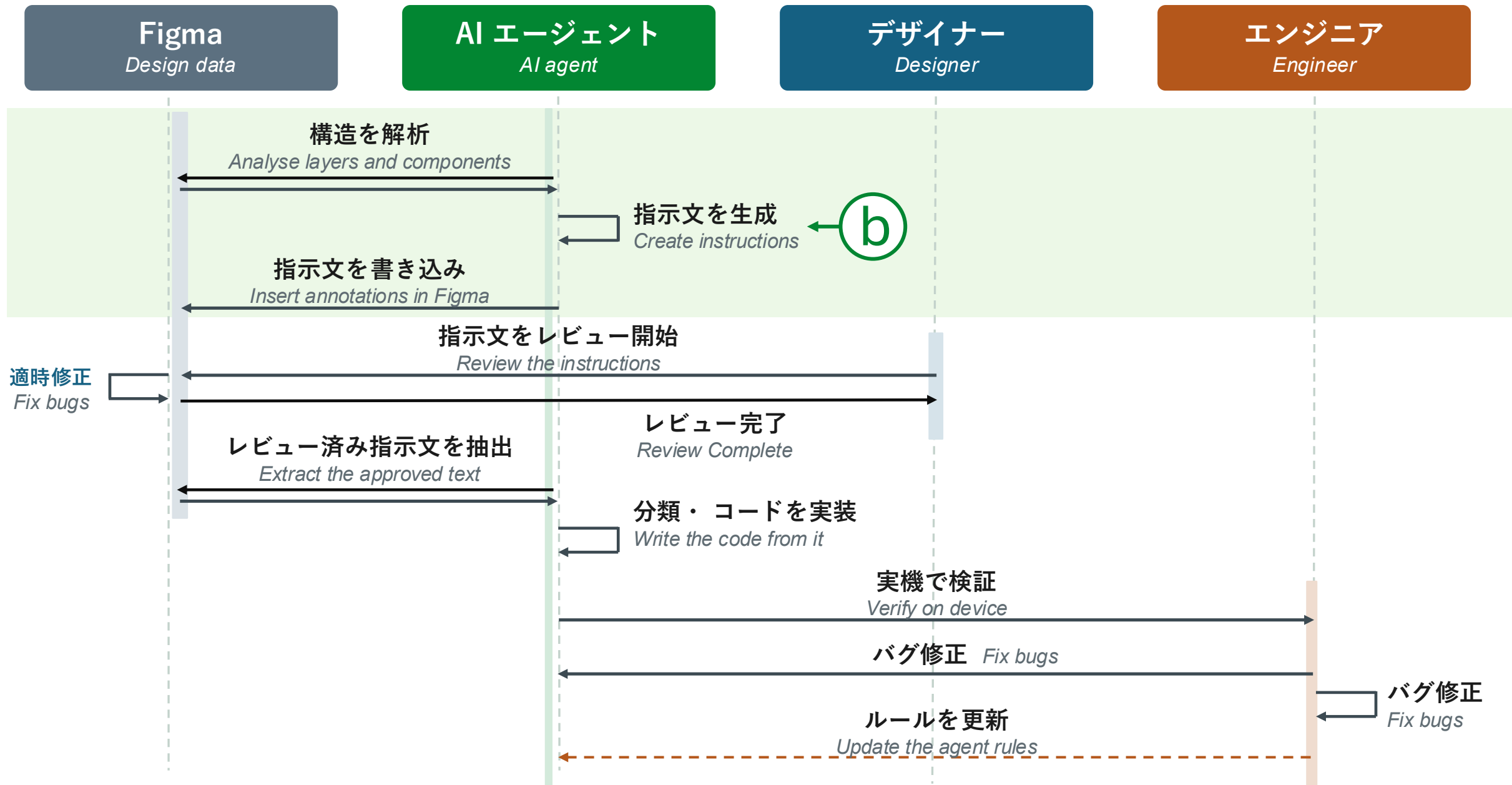
気付きへの対応

How we responded



**Figmaで定義されているデザインの
最大のレイヤー数を把握し、
最大レイヤー数まで探索するように指示することで
安定して動作するようにした。**

We measured the maximum layer depth in the Figma designs and had the agent explore all the way to that depth, which made it run reliably.



b.指示文を生成

Create instructions

- **Code Connect 対応しているcomponentかの判定を行う**
 - **対象の場合はCode Connectのメインコンポーネントを参照して指示文を生成**
-
- Determine whether the component is covered by **Code Connect**
 - If it is, reference the Code Connect main component

b.指示文を生成

Create instructions

Code Connect対象外の場合は、component名、Figmaのレイヤー名、componentの要素のテキスト内容を元にRoleなどと紐づけ、指示文を生成

For components outside the scope of Code Connect, we link them to Role (and similar attributes) based on the component name, the Figma layer name, and the text content of the component's elements, and generate the instruction text from that.

指示文生成の仕組み

How instruction generation works

Code Connect 対象component の読み上げ指示文

Components covered by Code Connect

ボタン

ノードラベル	Sub-element	Role	Label	Element
PrimaryButton (内部遷移)	ボタン全体	Button	<code>\${labelText}</code>	—
PrimaryButton (外部遷移)	ボタン全体	Button	<code>\${labelText}</code> 外部サイトへ移動	—

指示文生成の仕組み

How instruction generation works

Code Connect非対象ノード（Figmaレイヤー名 / 要素のテキスト内容での命名）の読み上げ指示文

Nodes not covered by Code Connect (named by Figma layer name / inner text)

ノードラベル	判定基準	Role	Label	Element
image XX（Figmaレイヤー名）	装飾画像	Image	なし（hideFromAccessibility）	—
タイトルテキスト（内容命名）	ページ・セクションの主見出し	heading	—	—
注記テキスト（内容命名）	補足情報テキスト	—	contentDescription	—

気付き

What we noticed



**Figmaのレイヤー名と内容の命名
の方針が整備できていないと、
Role/Label情報との紐づけが上手くいかないことが増
える。**

Without a well-defined convention for naming Figma layers and their content, failures to correctly associate Role/Label information will become more frequent.

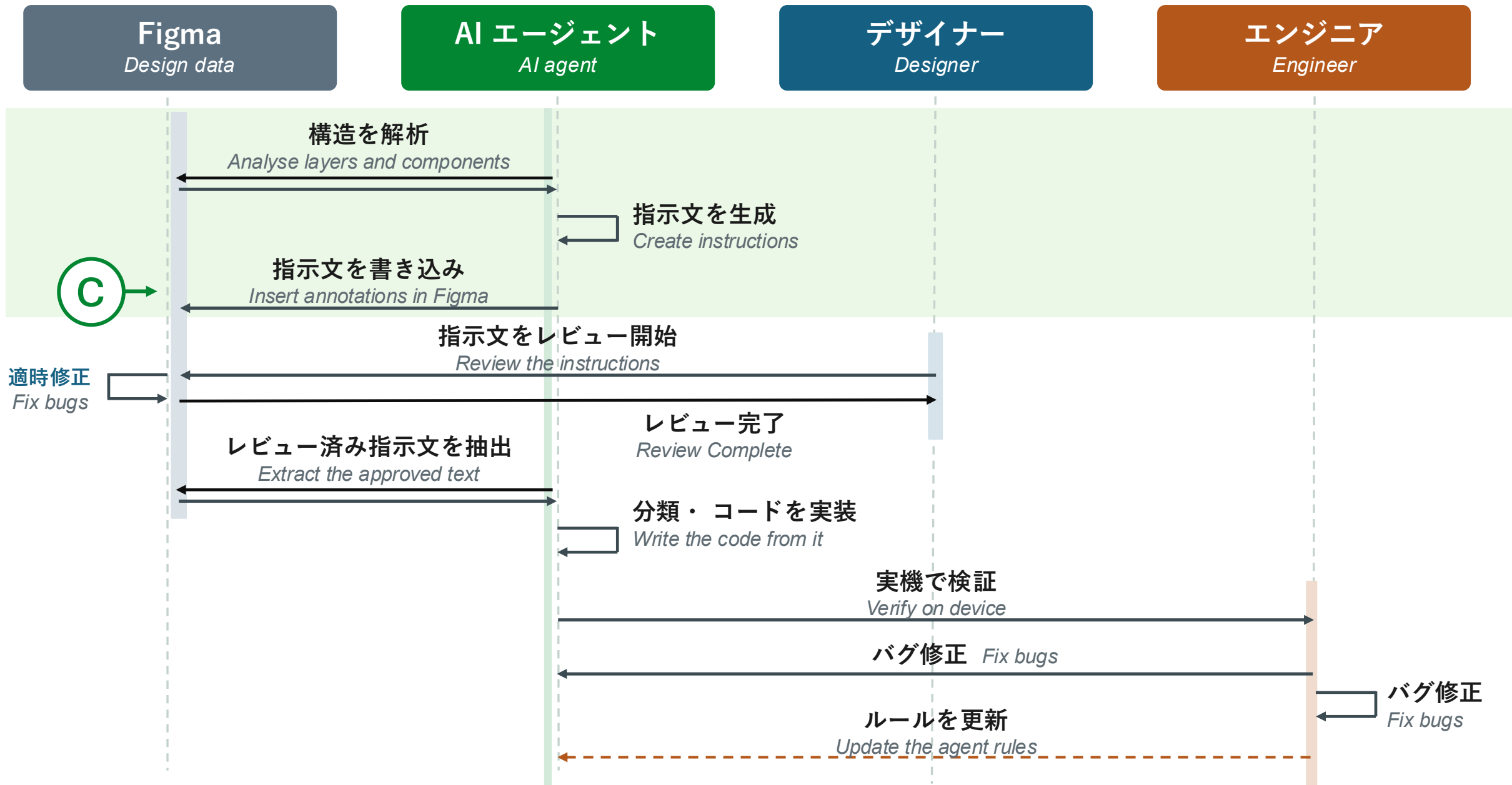
気付きへの対応

How we responded



**Figmaで扱うレイヤーの命名規則を
デザイナー、エンジニア間で予め決めておく
Role/Labelの紐づけ、指示文作成の漏れが減らせる。
コミュニケーションコストも減る。**

Agreeing on Figma layer naming conventions between designers and engineers ahead of time reduces oversights, and also lowers communication costs.



c.書き込みと記録

Write back & record

**Figma書き込みツール経由で指示文をFigmaのアノテーションに自動送信（独自プラグイン）
メタデータをデザイン情報に紐づけることができるためアノテーションを使用。**

Automatically push the instruction annotations via a Figma write tool (our own plugin)

We use annotations because they let you attach metadata to design information

アノテーション送信プラグインを自作

How instruction generation works

**Figma MCPサーバー経由では
アノテーションを作成し書き込むことができなかった。**

It wasn't possible to create and write annotations via the Figma MCP server.



アノテーション送信プラグインを自作

How instruction generation works

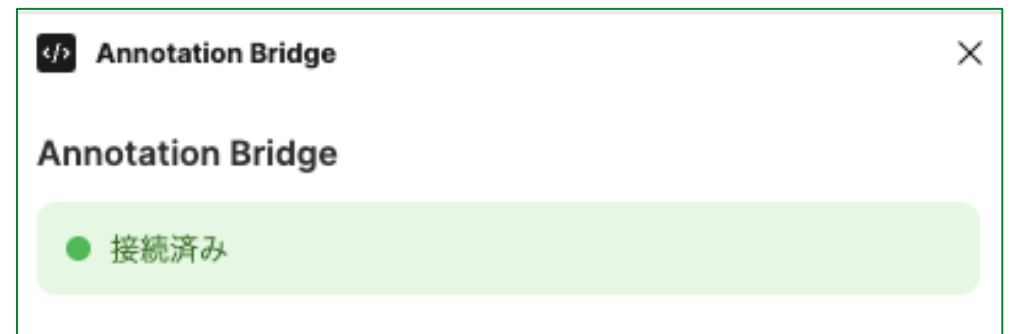
Claude Code → ローカルブリッジサーバー
→ Figmaプラグイン という経路でコマンドを中継

Commands are relayed along the path Claude Code -> local bridge server -> Figma plugin

```
🚀 Annotation Bridge サーバー起動 (port 3055)  
POST http://localhost:3055/annotations  
GET http://localhost:3055/health
```

Figma でプラグインを起動してください
停止するには Ctrl+C

🔌 Figma プラグインが接続しました



c.書き込みと記録

Write back & record

- 出力ログを残す
 - 生成された指示文の検証ができるようにツリー構造でログ出力
 - 間違いがあった場合に参照して改善に活かす。
- Keep output logs
 - Logs are printed as a tree so you can judge whether the generated instructions are right

書き込みと記録

How instruction generation works



```
## 画面全体構造
```

```
SCREEN_NAME (nodeId)
```

```
├ TokyoAlert (nodeId) - role: contentDescription / Label: ${警告テキスト}
```

```
└ Contents (nodeId)
```

```
├ image 47 (nodeId) - Role: Image / Label: なし ← Figmaレイヤー名
```

```
├ TokyoTagItem 「xxx」 (nodeId) - Label: xxx (アイコンは hideFromAccessibility)
```

```
├ タイトルテキスト (nodeId) - role: heading ← data-name なし・内容から命名
```

```
├ TokyoLabelItem h2 「xxx」 (nodeId) - role: heading
```

```
├ TokyoDisplay xxx (nodeId) - role: heading (h3ラベル)
```

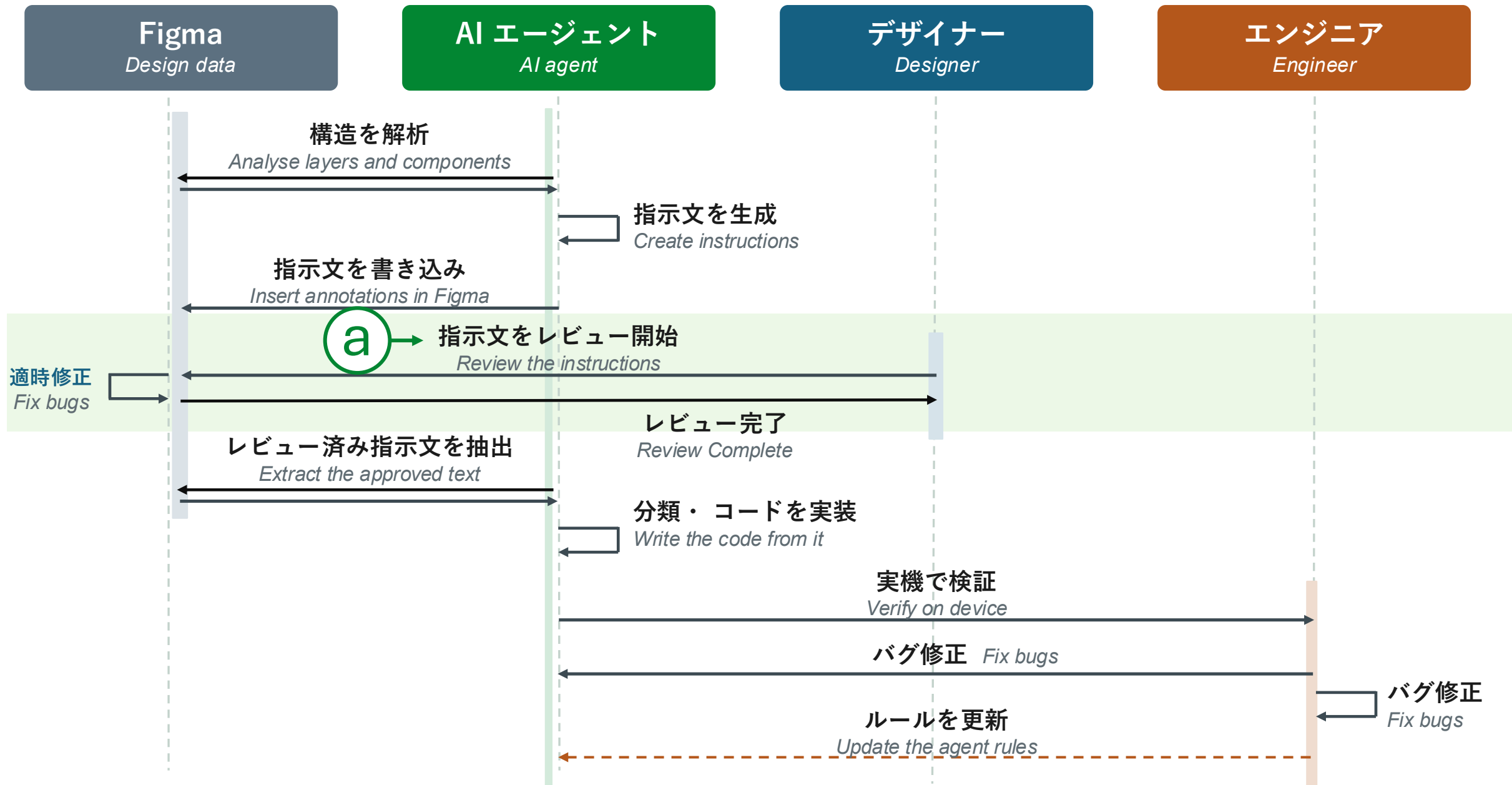
```
└ TokyoArea xxx (nodeId) - role: heading
```

```
├ TokyoLinkItem xxx - Role: Button / Label: xxx
```

```
└ TokyoLinkItem xxx - Role: Button / Label: xxx
```

指示文レビューの仕組み - デザイナー

How instruction review works - designer

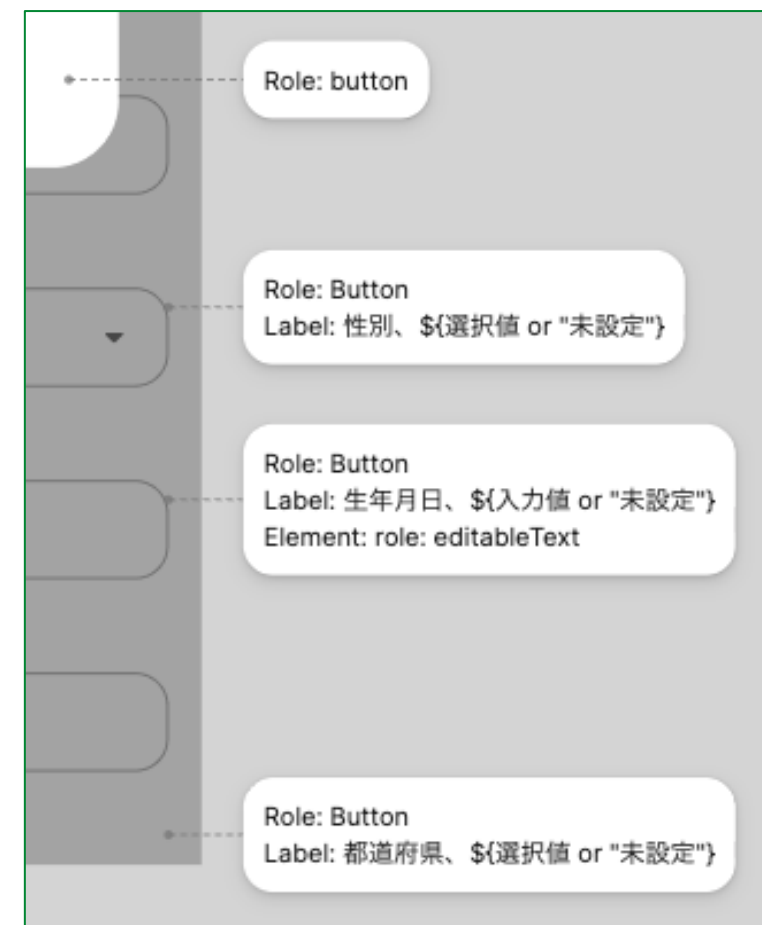


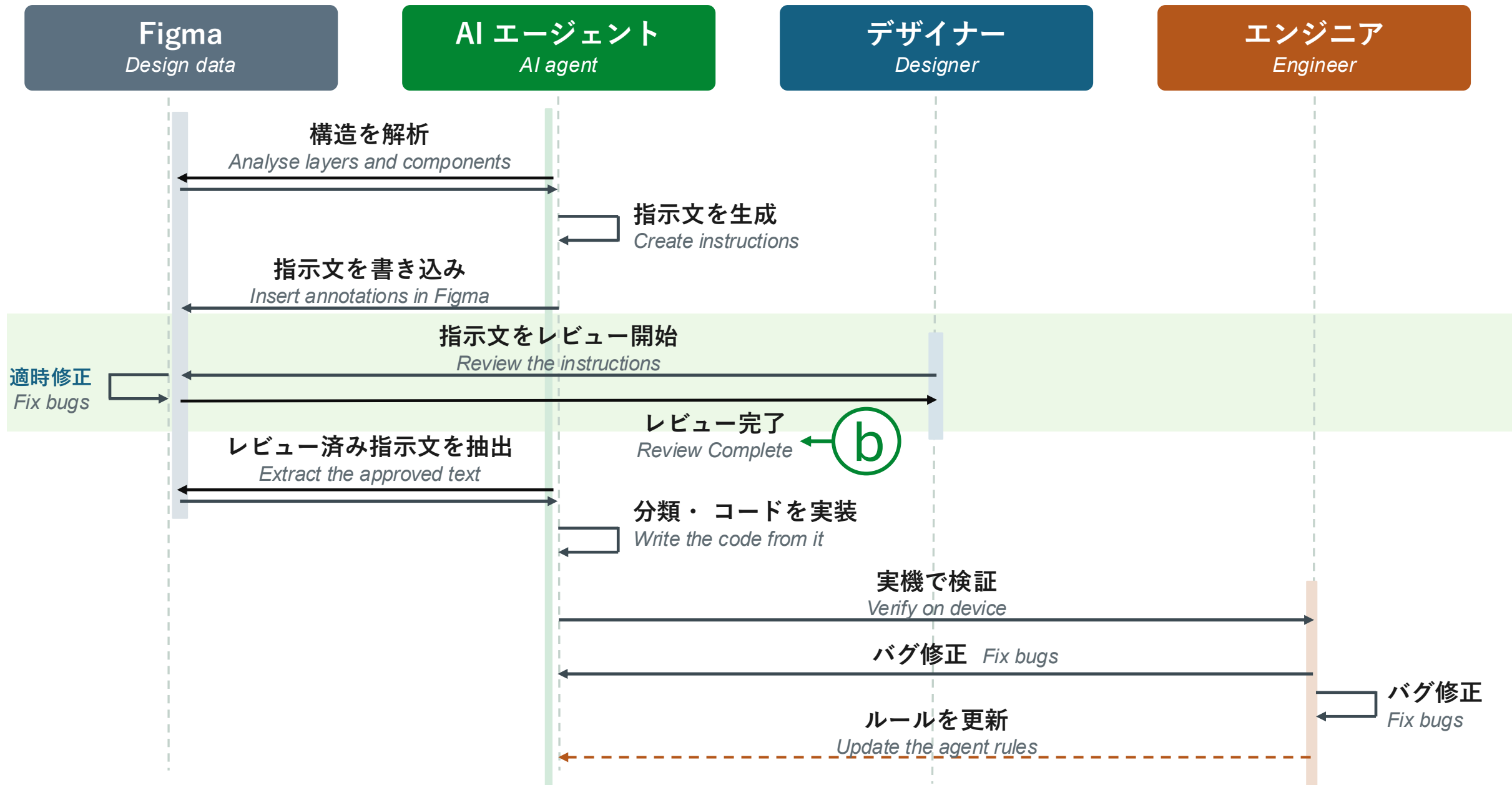
a.指示文レビュー

How instruction review works

生成されたアノテーションの内容を レビュー

Review the content of the generated annotations





b.レビュー完了

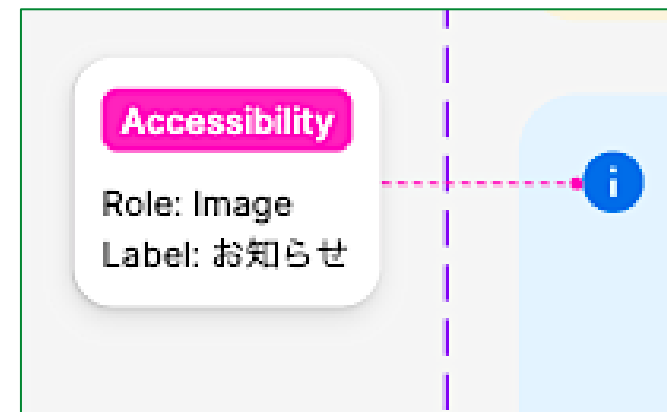
How instruction review works

アノテーションのCategory に[Accessibility]を追加することでレビュー完了とする。

Categoryを追加する意味は、レビュー漏れの有無の可視化と後続の実装で使用するため。

Adding [Accessibility] to the annotation's category marks the review as done.

The category makes unreviewed items visible, and it's also used by the implementation step that follows.



補足：指示文機能の始まり

Reference: how the instruction feature started

文言生成の自動化は、担当デザイナーから「Claudeを使って自動化できないか」という相談があったことがきっかけ。

デザイナー側から改善提案が自然に出てくる関係性があったことも、この取り組みを後押しした。

Automating the wording came directly out of the designer herself asking whether it could be automated with Claude.

Having a relationship where improvement ideas come naturally from the design side is part of what got this off the ground.

エンジニア起点でも成立する

It works starting from the engineering side too
エンジニア側からも

「こういう仕組みを作れますが試しますか」と
能動的に提案するだけでも十分に成立する取組みで
す。

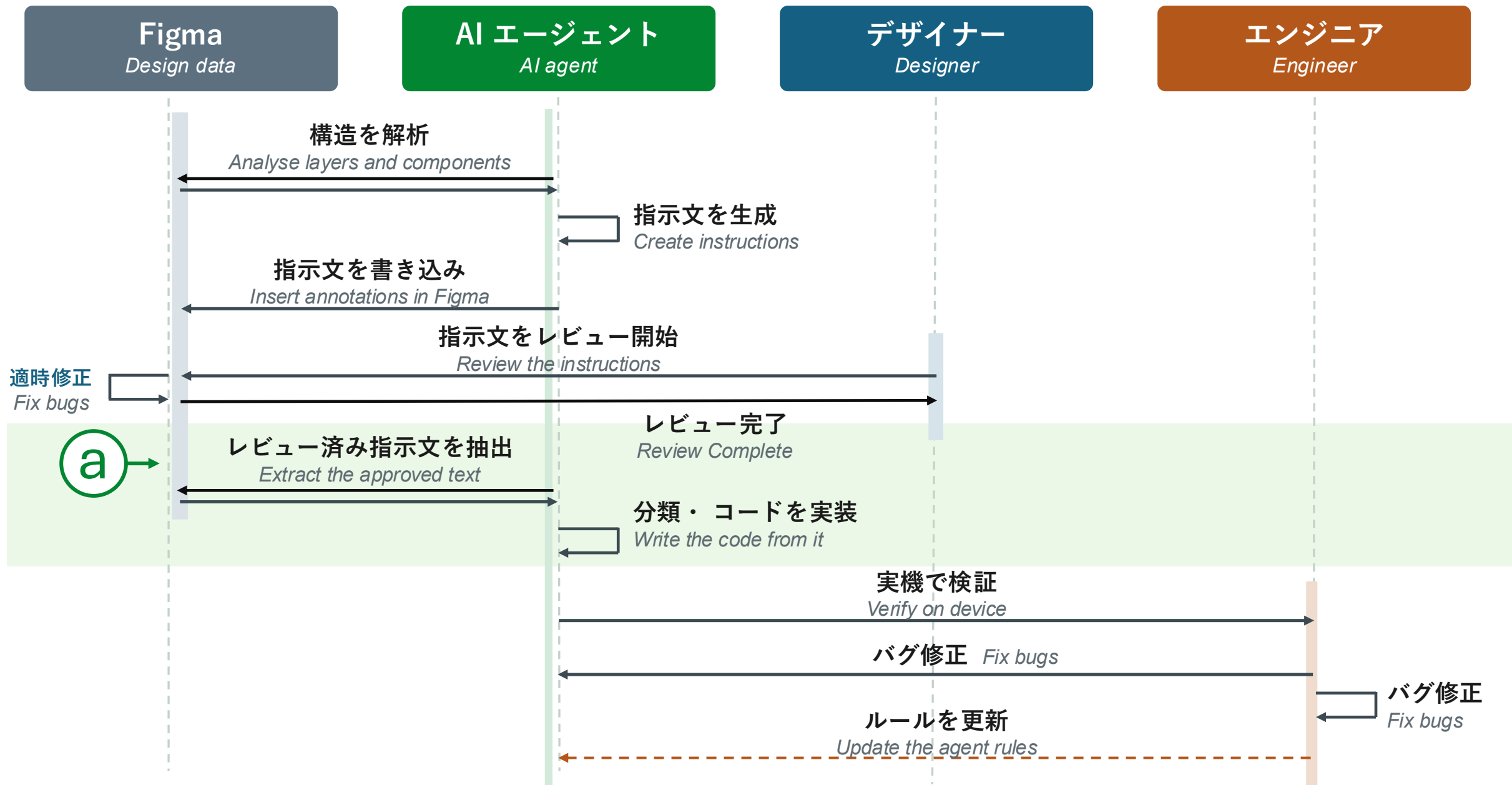
This works just as well if an engineer proactively offers: "I can build something like this - want to try it?"



TalkBack実装

- AIエージェント/エンジニア

Implementing TalkBack support - AI agent / engineer



a. 指示文の抽出

Extract the instructions

Figma MCPサーバー経由で指示文の抽出

Extract the instructions via the Figma MCP server

a.指示文の抽出

Extract the instructions

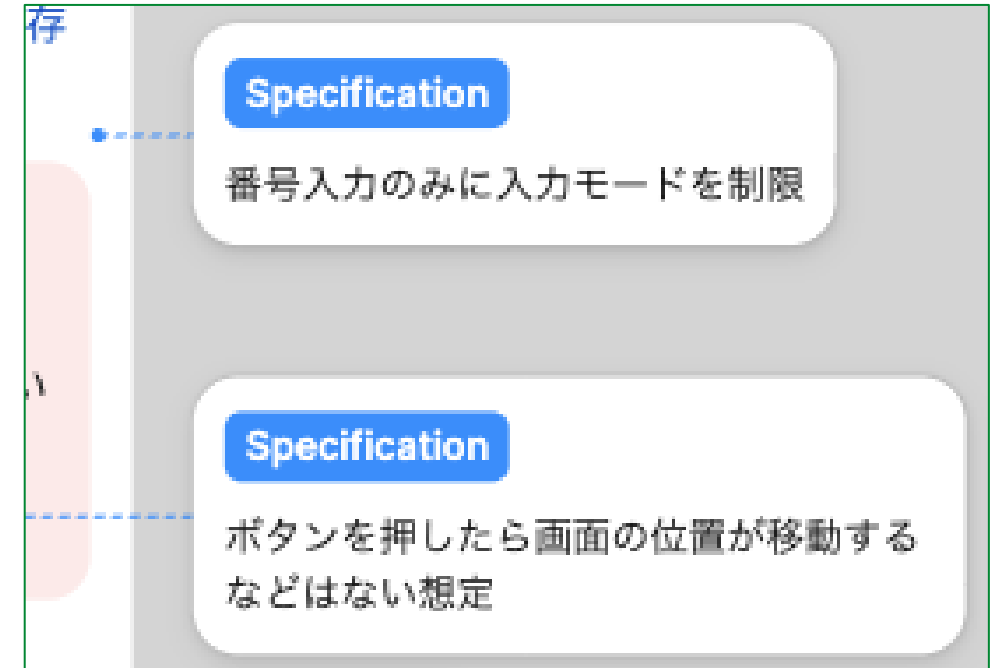
- `mcp_figma_use_figma` を呼び出し、Figmaプラグイン環境上で自作JSを実行
 - 対象ノード配下を再帰探索し、Accessibilityカテゴリの アノテーションのみを抽出
 - Call `mcp_figma__use_figma` and run our own JS inside the Figma plugin environment
 - Recursively walk everything under the target node and extract only Accessibility-category annotations
- * Annotation categories are used for things other than Accessibility as well

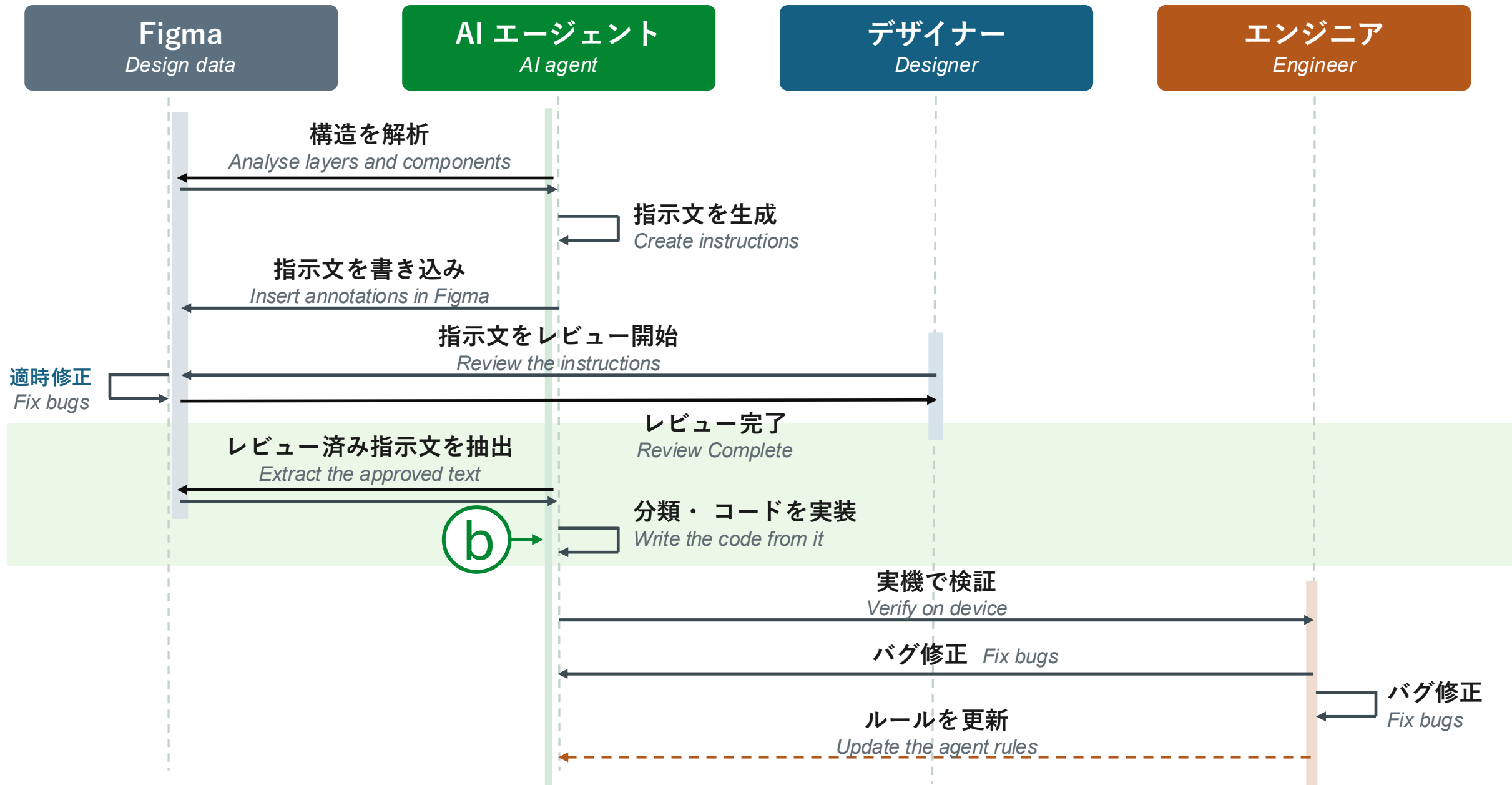
a. 指示文の抽出

Extract the instructions

アノテーションのCategoryは
Accessibility以外にも利用さ
れている

Annotation categories are used for
things other than Accessibility as well





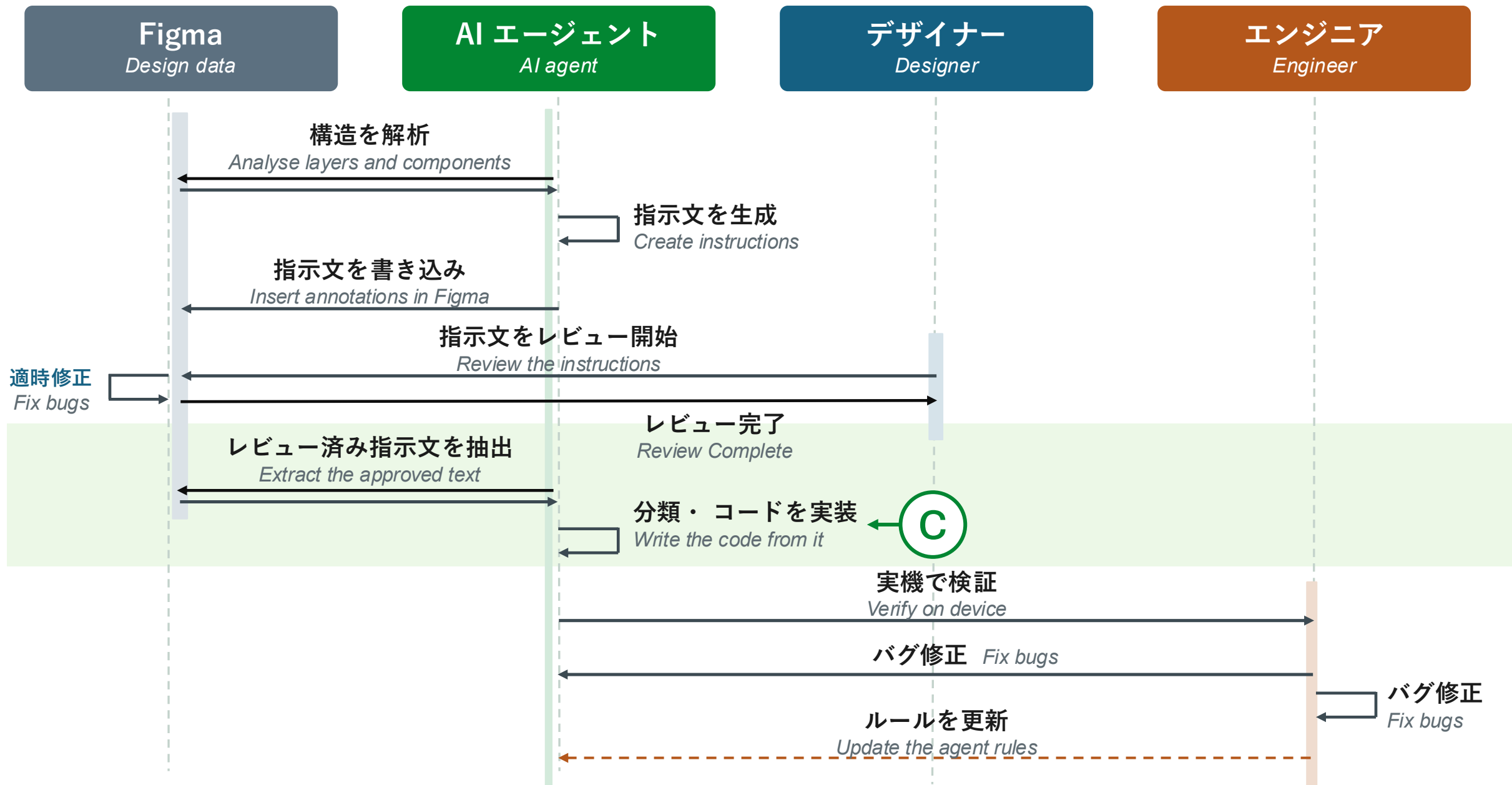
b.実装タイプで分類

Classify by implementation type

**読み取った指示文と指示対象のComponentを解釈し
3つの実装タイプに分類。**

Interpret the extracted instruction text and the target component, and classify them into three implementation types.

- **Screen type**
- **Item type**
- **Component type**



c.実装

Implement

ScreenType

画面表示時、画面の状態変化時、
などで
何をどう読み上げるか定義。

Implement what to read aloud and how to read it — for example, when the screen is displayed, or when the screen's state changes.

c.実装

Implement

ScreenType

- 画面固有の見出し
- 読み上げ順序
- フォーカス管理
- 状態変化など
- Screen-specific heading structure
- Announcement order
- Focus management
- State changes, and so on



c.実装

Implement

例：ScreenTypeの実装

Example: implementing a Screen type

semantics property : paneTitle

画面が切り替わったことを知らせるために使用

A screen-reader announcement indicating that the screen has switched.



```
Scaffold(  
  topBar = {...},  
  ...  
  modifier = modifier.semantics { paneTitle = “お知らせ” },  
)
```

c.実装

Implement

例：ScreenTypeの実装

Example: implementing a Screen type

semantics property : liveRegion

- 画面上のテキストや状態が動的に更新された時に読み上げ
- ユーザー操作で変化する値に付与(未読件数・検索結果件数等)
- Announced when text or state on screen updates dynamically
- Attached to values that change with user action (unread count, number of search results, etc.)



```
Text(  
    text = “未読：%d件”,  
    modifier =  
        Modifier.semantics {  
            contentDescription =  
                “${unreadCount}件が未読”  
            liveRegion =  
                LiveRegionMode.Polite  
        },  
)
```

c.実装

Implement

ItemType

ListやGridの中で同じ構造が繰り返されるItemが対象となる。

Itemの内部で複数のcomponentをどう合成して読み上げるか定義。

Defines how multiple components inside a repeating Item in a List or Grid are composed into a single announcement.

c.実装

Implement

ItemType

- 1回のフォーカス移動で1件分の情報が過不足なく伝わるよう定義する。
- component同士が意図せず重複して読まれる「二重読み上げ」を防ぐ。
- Define it so that a single focus movement conveys exactly one complete unit of information — no more, no less.
- Prevent the "double announcement" where components get read out redundantly.

c.実装

Implement

例：Item Typeの実装

Example: implementing an Item type

semantics property : clearAndSetSemantics

- 自身とその配下にあるすべての子コンポーネントのアクセシビリティ情報を消去し、新しく指定した情報だけで上書きする
- Clears the accessibility information of the node and all of its children, replacing it with only what you newly specify

c.実装

Implement

Accessibility

role: ListView

エレメント: まとめる

ラベル:

`${if(未読バッチが存在する){未読}},`

`${if(重要ラベル存在する){たいせつなお
しらせ}},`

`${お知らせタイトル},`

配信日時 `${年月日},`

開催地 `${開催地},`

カテゴリ `${カテゴリ名}`

例：ItemTypeの実装

Example: implementing an Item type

// 読み上げ分の生成

```
buildList {  
    if (item.isImportant) add(importantLabel)  
    add(item.title)  
    if (item.date != null) add(dateLabel)  
    if (item.area != null) add(areaLabel)  
    if (item.category != null) add(categoryLabel)  
}.joinToString("、")
```

c.実装

Implement

ComponentType

単体で完結するcomponentのどう読み上げるか定義。

「このボタンは何をするボタンか」「この画像は何を表しているか」を、componentを1つだけ見て判断。

Defines how a self-contained component is announced.

"What does this button do?" and "What does this image show?" are answered by looking at that one component alone.

c.実装

Implement

ComponentType

周囲に何が並んでいるか、リストの何番目か、画面のどんな状態か、といった文脈には一切依存しない。

It depends not at all on context - what sits around it, where it falls in a list, or what state the screen is in.



c.実装

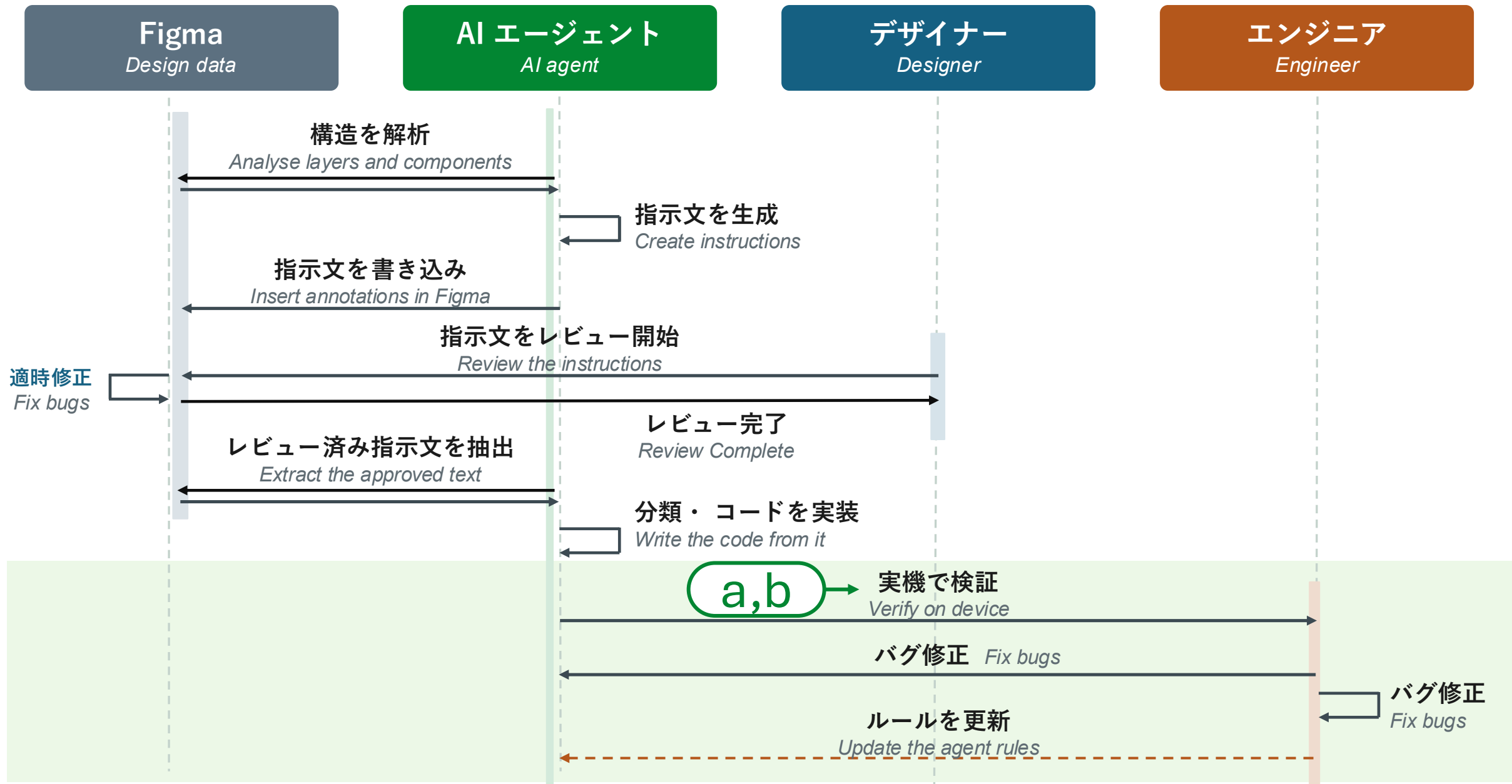
Implement

実装する内容に応じて、タイプに分類することで何に注意するかを明確にすることができました。

By classifying into types, we were able to clarify what points require attention depending on the kind of implementation."

実機検証 - エンジニア

On-device testing - engineer

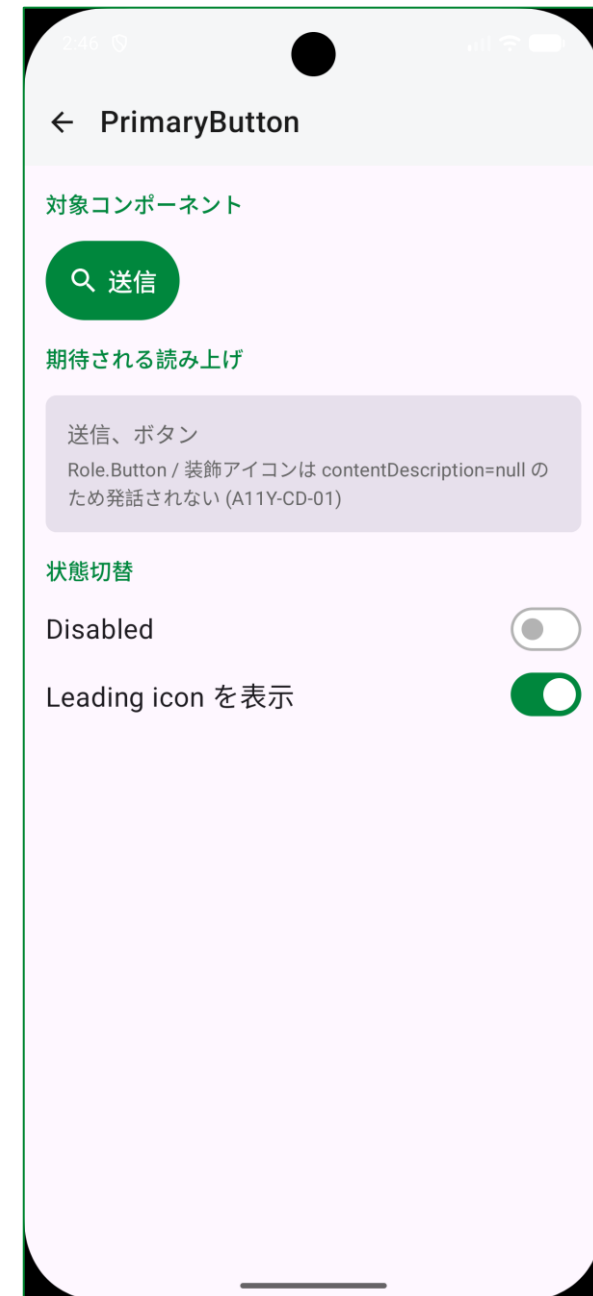
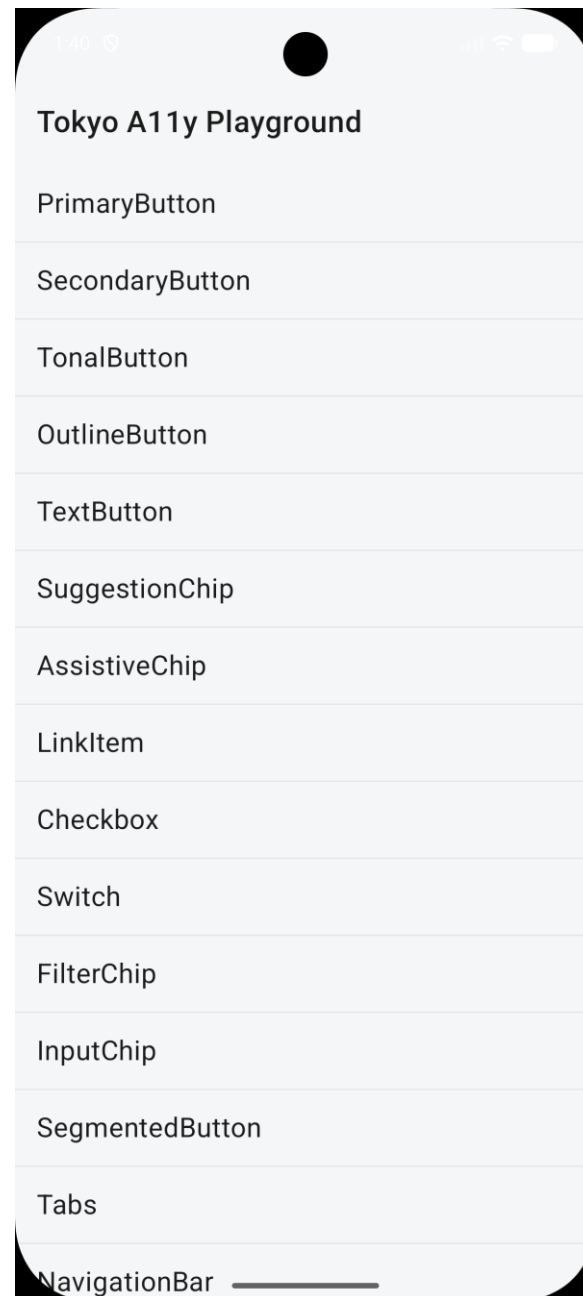


a.Component単体で 確認

Check in isolation

共通コンポーネント用の 検証アプリを作成

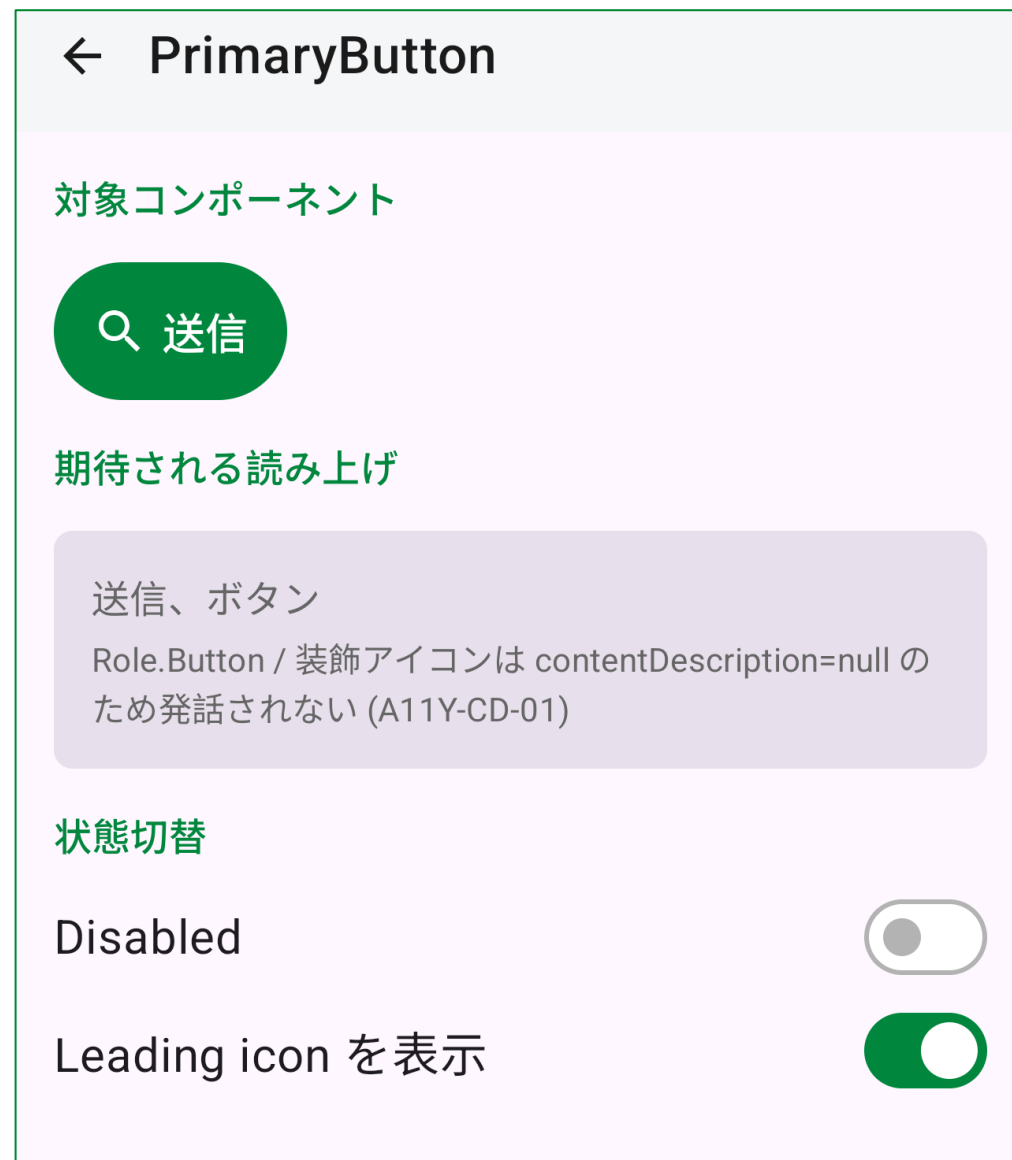
We prepared an app for the
shared components



a.Component単体で確認

Check in isolation

- 35個の共通コンポーネントを全て網羅
- 各コンポーネントの状態に合わせて、読み上げ想定文が表示される
- Every shared component is covered
- The expected announcement is displayed for each component state

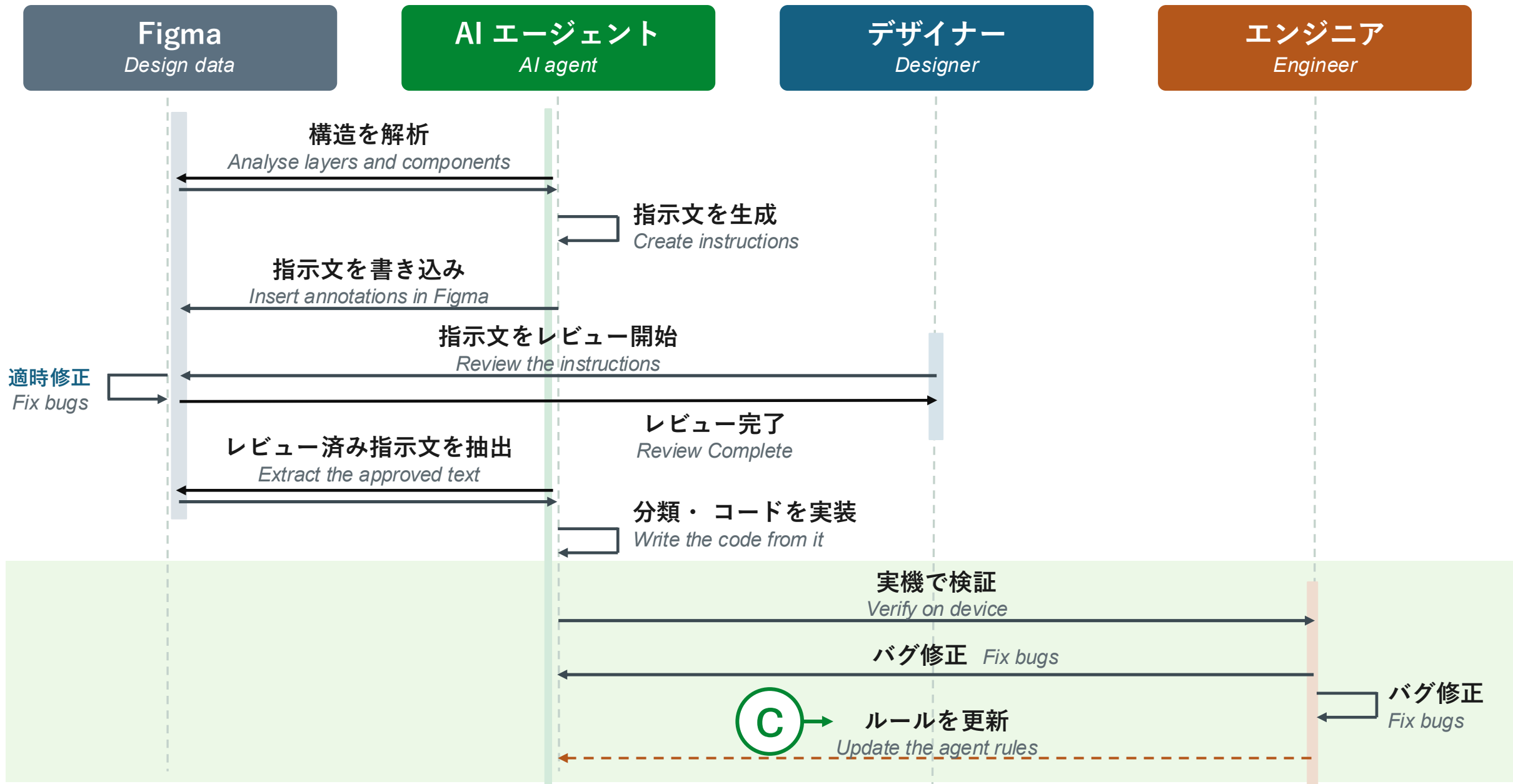


b.アプリで確認

Check in app

- 画面遷移時の読み上げ
- 状態が変わったテキストが読み上げられるか
- Announcements on screen transitions
- Whether text is announced when its state changes





c.修正・Rule化

Check in app

実機検証で発見されたバグパターンは、Claude CodeのRuleに反映する。

We keep feeding bug patterns found in on-device testing back into Claude Code's Rules.

AIエージェントでのバグ紹介

Introducing Bugs Found via the AI Agent

バグ① Componentタイプ

Bug 1: Component type

二重読み上げ (contentDescription + 内部 Text)

Buttonは内部のTextプロパティをマージして読み上げる。プロパティが重複しないように指定する必要がある。

Double announcement (contentDescription + inner Text)

A Button announces its internal Text properties merged together. You need to make sure the properties don't overlap/duplicate.

バグ① Componentタイプ

Bug 1: Component type

❌ 2重読み上げ: 「送信 送信、ボタン」と読まれる

Double announcement: read as "Submit Submit, button"

```
Button(  
  onClick = { },  
  modifier = Modifier.semantics {  
    role = Role.Button  
    contentDescription = "送信"  
  },  
) {  
  Text(text = "送信") // ← Text プロパティがマージノードに残る  
}
```

✅ 2重読み上げなし: 「送信、ボタン」と読まれる

No double announcement: read as "Submit, button"

```
Button(  
  onClick = { },  
  modifier = Modifier.semantics {  
    role = Role.Button  
  },  
) {  
  Text(text = "送信")  
}
```

バグ② Screenタイプ

Bug 2: Screen type

liveRegionの表示・非表示の切り替え時に、TalkBack対象の現在位置が強制リセットされ、画面先頭（戻るボタン等）へジャンプして意図しない読み上げが発生する。

対象の要素をデータ構造から完全に削除してしまうため発生。

When a liveRegion node is toggled visible/invisible, TalkBack's current focus position is forcibly reset, jumping to the top of the screen (e.g., the back button) and triggering an unintended announcement.

Because the target element is completely removed from the data structure, this occurs.

バグ② Screenタイプ

Bug 2: Screen type

✗ 表示非表示を制御 したためフォーカスが移動

Toggling visibility caused the focus to move

```
if (text.isNotEmpty()) {  
    Box(modifier = Modifier.size(1.dp).semantics {  
        contentDescription = text  
        liveRegion = LiveRegionMode.Polite  
    })  
}
```

✓ 要素を常時存在させ、contentDescription の値変化のみで発火させる

Keep the node always present, and trigger only via contentDescription value changes

```
Box(modifier = Modifier.size(1.dp).semantics {  
    contentDescription = text  
    liveRegion = LiveRegionMode.Polite  
})
```

 **AIエージェントの生成コードは、
一見正しく見えても実機検証なしには
品質を担保できない**

Code generated by an AI agent can look correct
and still offer no quality guarantee
without on-device testing

Ruleを作りメンテナンスを 継続することで、実装コストを増やさず バグの再発を防ぎやすくなる

Building Rules and keeping them maintained makes it
easier to stop bugs from recurring without adding
implementation cost

**ワークフローを回し続けることで、
AIエージェントに知見が溜まり
プロダクトの品質が向上していく**

Keep the workflow turning and knowledge accumulates
in the AI agent, raising product quality over time

もくじ

1. なぜAIエージェントを用いたワークフローを組んだのか
2. ワークフロー構築時に気をつけたこと
3. デザイナーとAIエージェントとのワークフロー
 - 指示文作成
 - TalkBack実装
 - 実機検証
 - AIエージェントでのバグ紹介
4. まとめ

Index

1. Why we built a workflow around an AI agent
2. What we were careful about when building the workflow
3. The overall workflow between designers and the AI agent
 - Writing the instructions
 - Implementing TalkBack support
 - On-device testing
 - Introducing Bugs Found via the AI Agent
4. Wrap-up

まとめ

Wrap-up

まとめ

Wrap-up

**TalkBack対応を"仕組み"として作り込めば、
画面が増えても継続的に開発が可能**

By building TalkBack support into the system as a structured mechanism, development can continue sustainably even as the number of screens increases.

まとめ

Wrap-up

**はじめてからワークフロー全体を
一度に構築する必要はありません**

There's no need to build the entire workflow all at once from the start.

まとめ

Wrap-up

見つけたバグを機械的な形で書き戻す
「バグパターン集」のような
Rule作りだけからでも、十分に効果があります

Even just building Rules - a "bug pattern catalog" that mechanically writes back the bugs you find - is plenty effective

ご清聴有難う御座いました

Thank you for listening