ViTO (Visual Test Oracle)

How we slashed our maintenance by 50% using GenAI

Rahul Singh

BlueYonder

4/06/2026

About

Rahul Singh

Staff Software Developer (AI Solutions), at

BlueYonder

Berlin, Germany

LinkedIn

GitHub

HackerRank

stackoverflow

BlueYonder

The world leader in digital supply chain transformation

3000+
Customers across retail, logistics, and manufacturing

23
Of the Top 25 retailer

16
Of the Top 25 Consumer Product Goods

13
Of the Top 25 Logistic Service Providers

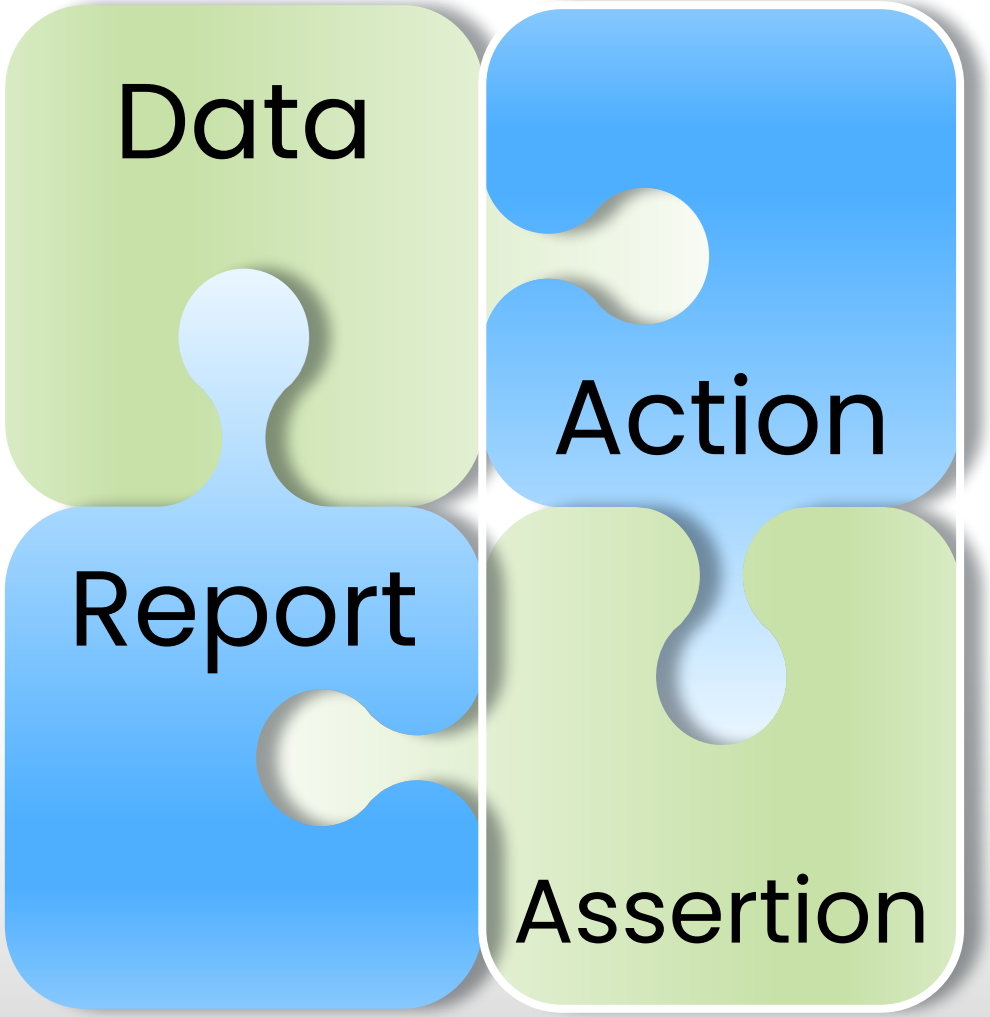
12
Of the Top 25 manufacturers

Our mission is *To empower every person and organization to fulfill their potential*

Context: 4 Components of UI Automation

UI automation can be segregated into 4 key components

- Credentials
- Selectors
- Config data



- Expand dropdown
- Select value
- Click button

- Report status
- Tests' meta data
- Stack trace

- Verify page opens without errors
- Verify widgets are not blank

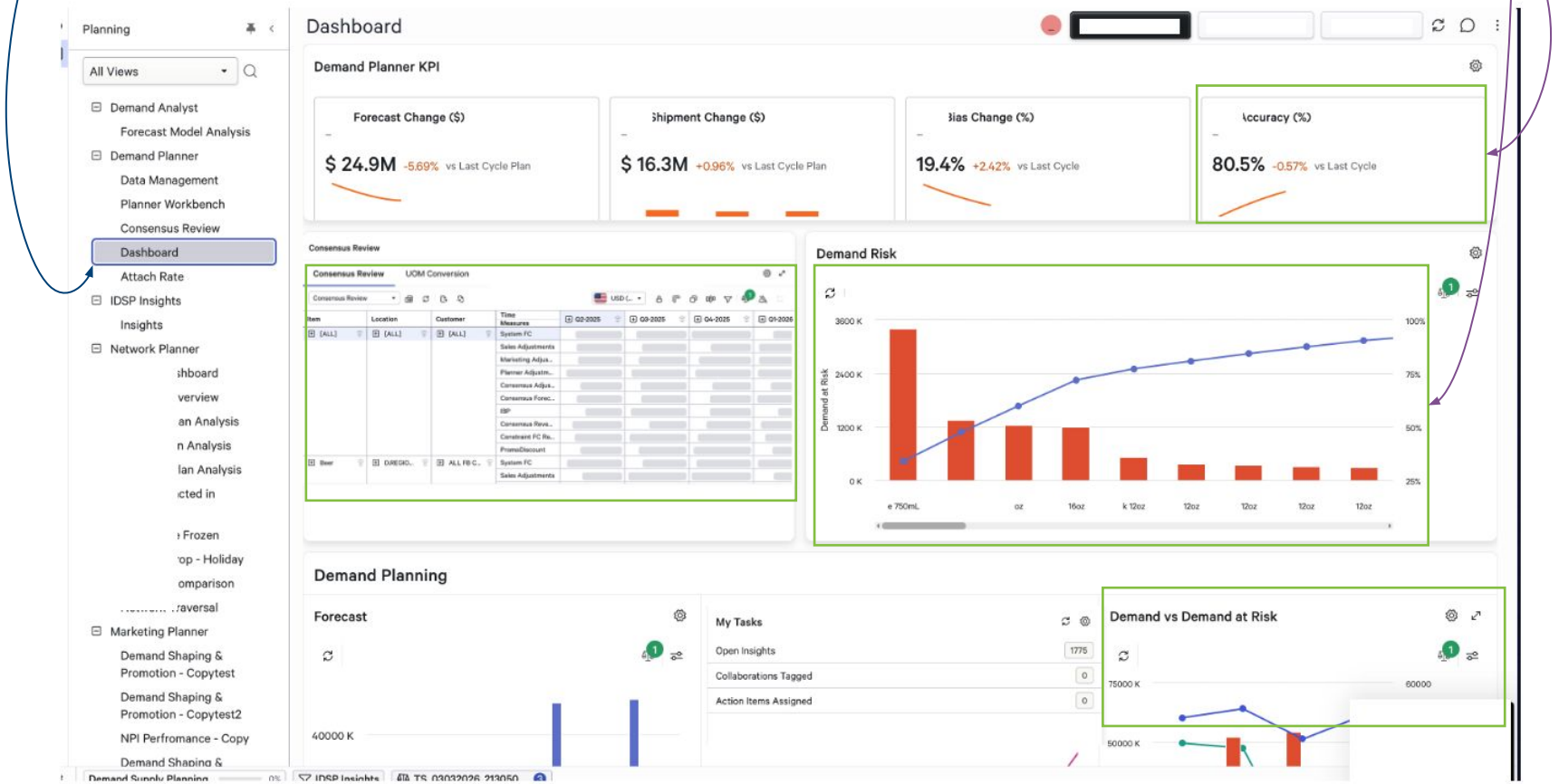
Example Test Case

Action

- Click on a ‘view’

Assertion

- Verify every widget loads without errors



Challenges with Conventional Automation

Scope of Automation

- There are **20+ unique kinds of widgets**. Each kind needs to be coded separately
- **Charts can be a combination**: “bar + line”, “area + bar”. **Every combination needs to be uniquely identified** else the widget goes untested

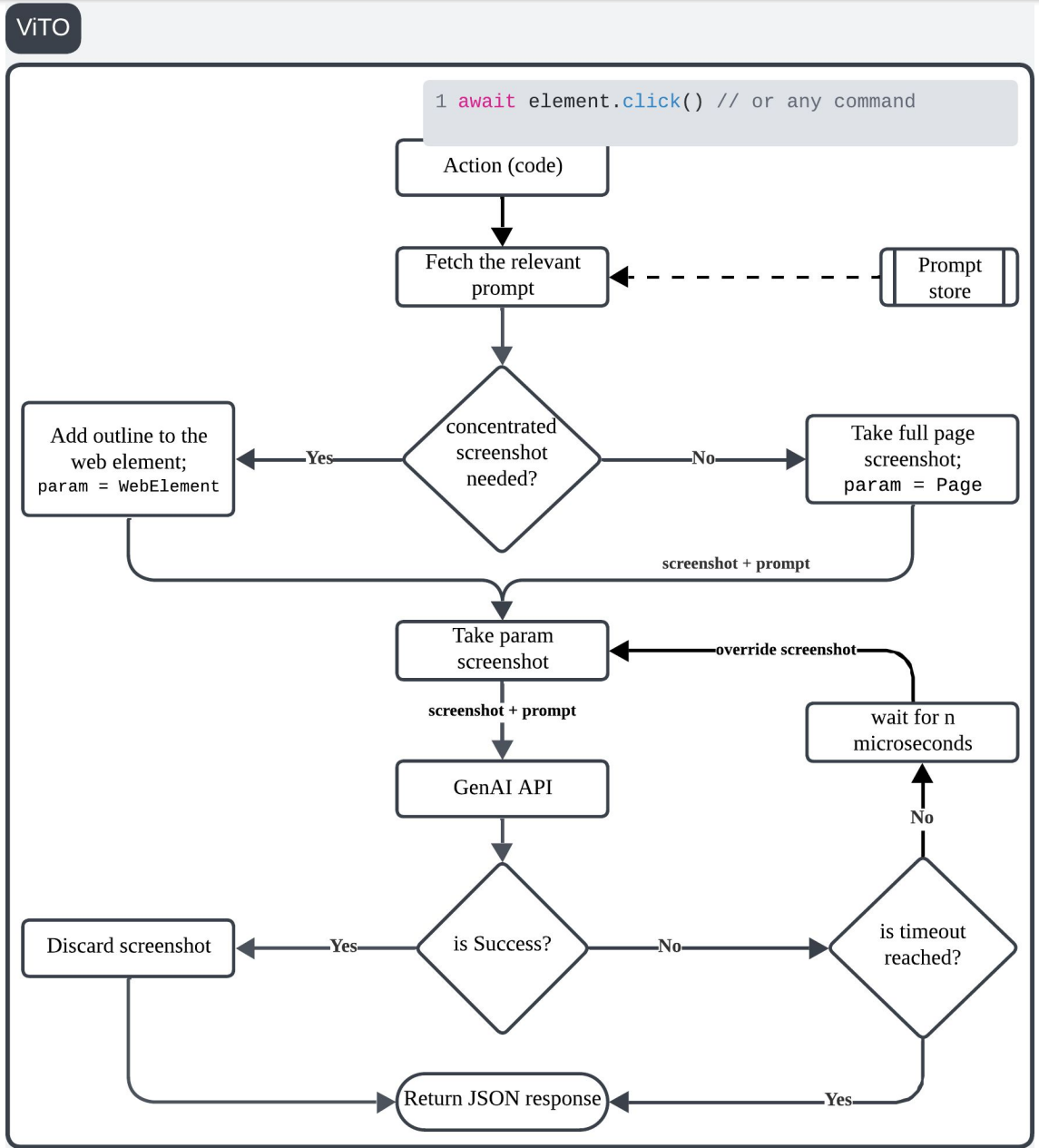
Customised platform

- Since **customer platforms are customised**, some widgets would appear for specific customers
- Out of the **100+ customer platforms** being tested, a new KPI widget gets introduced for one customer. Since we didn’t encounter such widget until then, it was not handled in the code, and thus, caused **false negatives**

Development & maintenance

- Assertion code is **5x larger** than the action code. New feature development is **costly**
- Large maintenance overhead: In 2025, we spent **approx. 4 months** were spent only in **maintenance**
- An API error occurred in CI but never during local development & execution. As a result, the error went unchecked and caused **false positives**

ViTO – Visual Testing Oracle



Hybrid approach

Use code for actions;

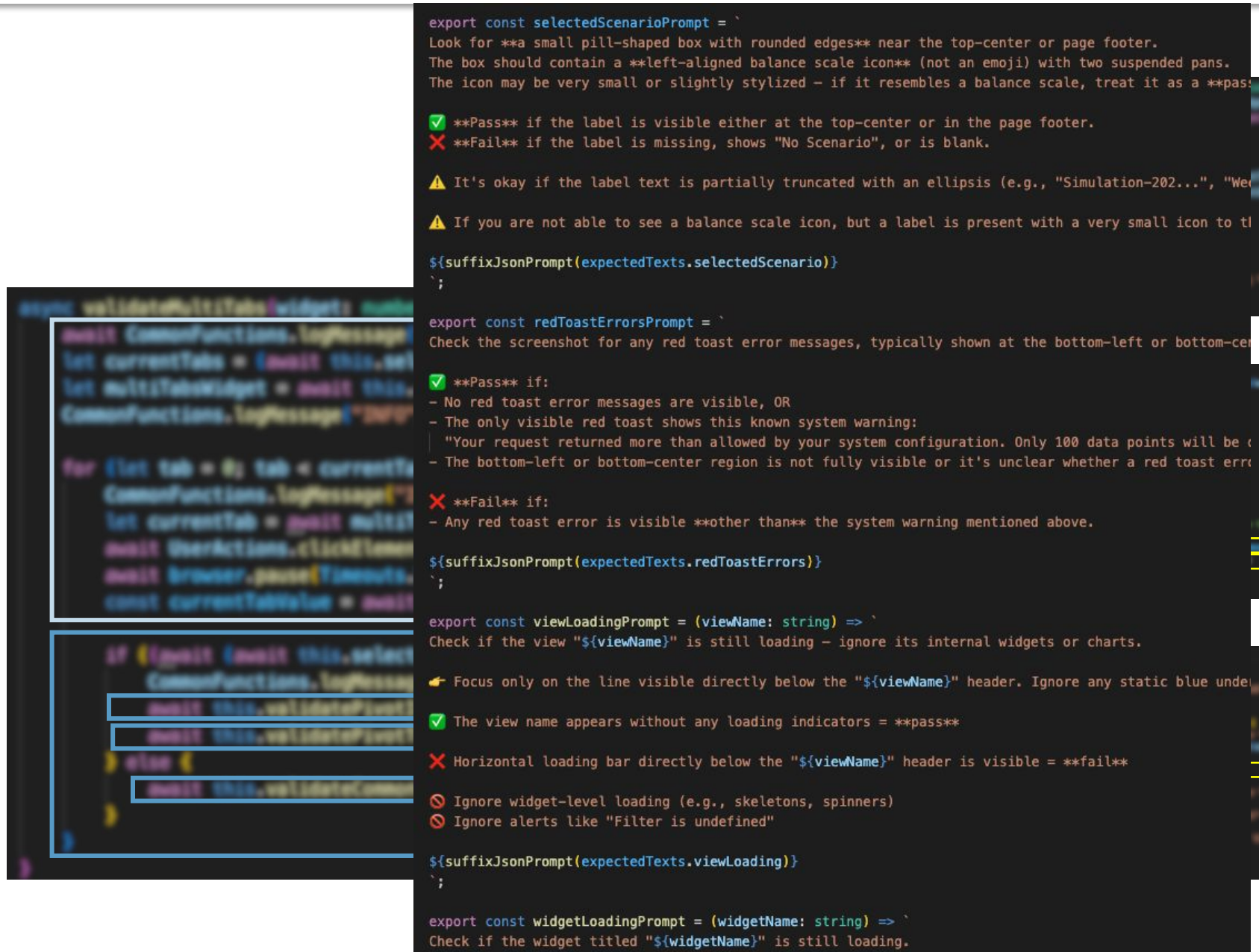
GenAI for assertions

A 3-step process

1. Take a screenshot of the page/element. Feed the screenshot and the prompt to the OpenAI API.
2. If the response is a fail, delete the screenshot and repeat step 1
3. Repeat until timeout provided to the function is reached, or GenAI API responds with ‘Pass’

The code replaced with prompts

ViTO – Visual Testing Oracle



```
export const selectedScenarioPrompt = `
Look for **a small pill-shaped box with rounded edges** near the top-center or page footer.
The box should contain a **left-aligned balance scale icon** (not an emoji) with two suspended pans.
The icon may be very small or slightly stylized – if it resembles a balance scale, treat it as a **pass

✅ **Pass** if the label is visible either at the top-center or in the page footer.
❌ **Fail** if the label is missing, shows "No Scenario", or is blank.

⚠️ It's okay if the label text is partially truncated with an ellipsis (e.g., "Simulation-202...", "We

⚠️ If you are not able to see a balance scale icon, but a label is present with a very small icon to t

${suffixJsonPrompt(expectedTexts.selectedScenario)}
`;

export const redToastErrorsPrompt = `
Check the screenshot for any red toast error messages, typically shown at the bottom-left or bottom-ce

✅ **Pass** if:
- No red toast error messages are visible, OR
- The only visible red toast shows this known system warning:
  "Your request returned more than allowed by your system configuration. Only 100 data points will be
- The bottom-left or bottom-center region is not fully visible or it's unclear whether a red toast err

❌ **Fail** if:
- Any red toast error is visible **other than** the system warning mentioned above.

${suffixJsonPrompt(expectedTexts.redToastErrors)}
`;

export const viewLoadingPrompt = (viewName: string) => `
Check if the view "${viewName}" is still loading – ignore its internal widgets or charts.

👉 Focus only on the line visible directly below the "${viewName}" header. Ignore any static blue unde

✅ The view name appears without any loading indicators = **pass**

❌ Horizontal loading bar directly below the "${viewName}" header is visible = **fail**

🚫 Ignore widget-level loading (e.g., skeletons, spinners)
🚫 Ignore alerts like "Filter is undefined"

${suffixJsonPrompt(expectedTexts.viewLoading)}
`;

export const widgetLoadingPrompt = (widgetName: string) => `
Check if the widget titled "${widgetName}" is still loading.
```

Two Key Components

Prompt Engineering

✅ Left panel shows view names = **pass**
✅ Any content (widgets, placeholders) is visible = **pass**
✅ Even a single widget outline should mean **pass**

❌ Horizontal loading bar directly below the "${viewName}" header is visible = **fail**
❌ Main area is blank with no visible content = **fail**

🚫 Ignore widget-level loading (e.g., skeletons, spinners)
🚫 Ignore alerts like "Filter is undefined"

```
Check the widget named "${widgetName}" for visual issues.
```

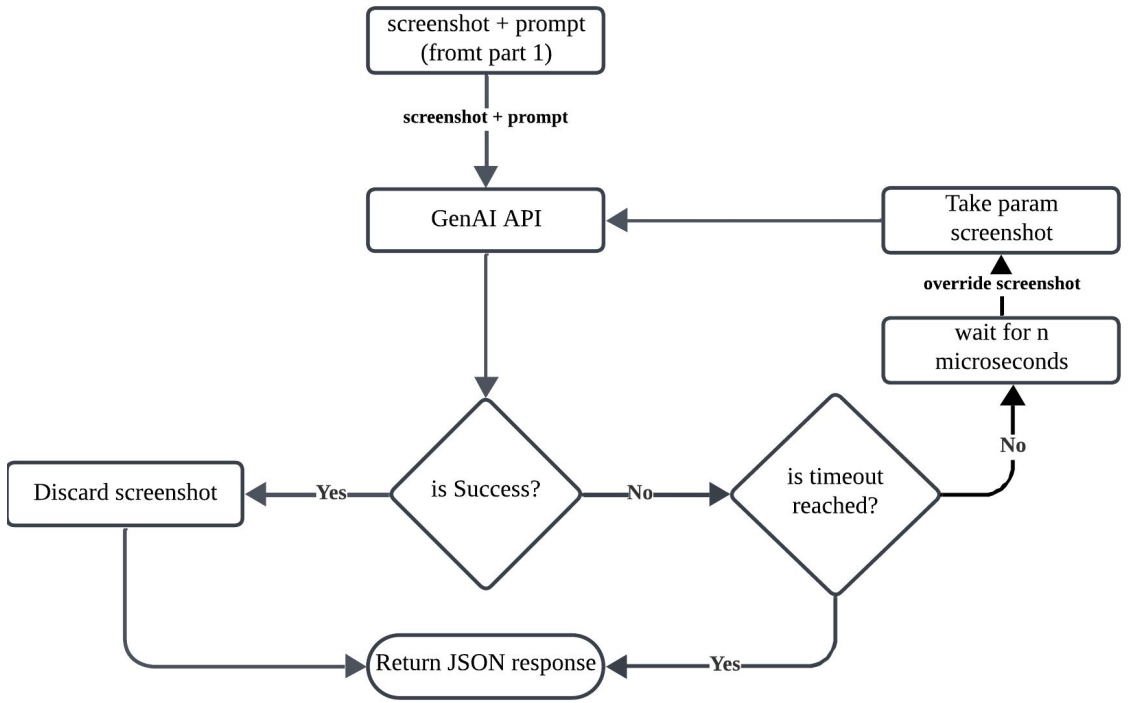
Start | Outline focus | Main body | Suffix prompt

A complex prompt comprises of these many parts

🟥 CRITICAL: Only evaluate the widget that is visually enclosed in a solid red rectangle (outline). This red border uniquely marks the widget to be evaluated.
🚫 Ignore all other widgets, even if they have the same name or are fully visible. Focus only on the one with the red outline.
📸 The red-outlined widget will have its content (chart, table, or data area) directly below its title/header.
👉 If the widget header is not visible, focus on the content area between two vertical red lines.

```
Respond only with a JSON object:
        {
            "result": "${expectedText}" // or "fail",
            "message": "Reason for the result"
        }
```

Response Evaluation & Retry



Two Key Components

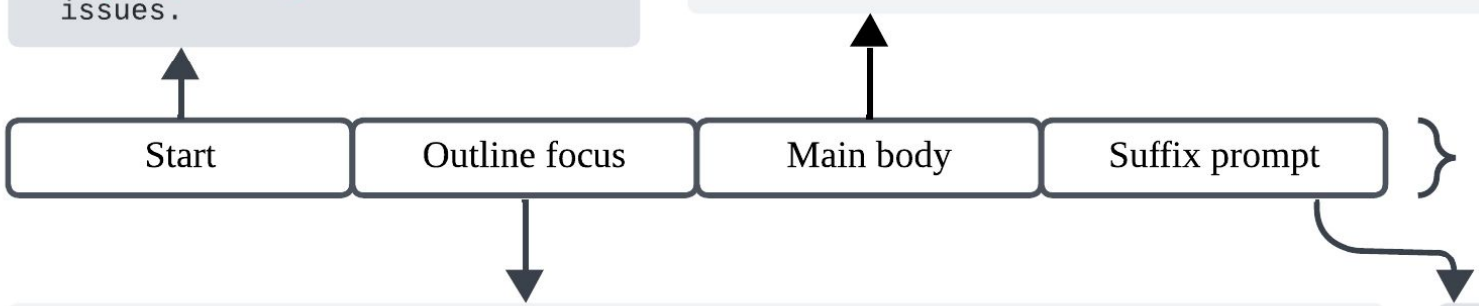
Pr ry

✅ Left panel shows view names = **pass**
✅ Any content (widgets, placeholders) is visible = **pass**
✅ Even a single widget outline should mean **pass**

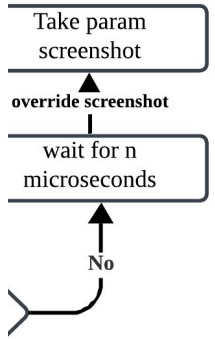
❌ Horizontal loading bar directly below the "${viewName}" header is visible = **fail**
❌ Main area is blank with no visible content = **fail**

🚫 Ignore widget-level loading (e.g., skeletons, spinners)
🚫 Ignore alerts like "Filter is undefined"

```
1 Check the widget named
  "${widgetName}" for visual
  issues.
```



A complex prompt comprises of these many parts

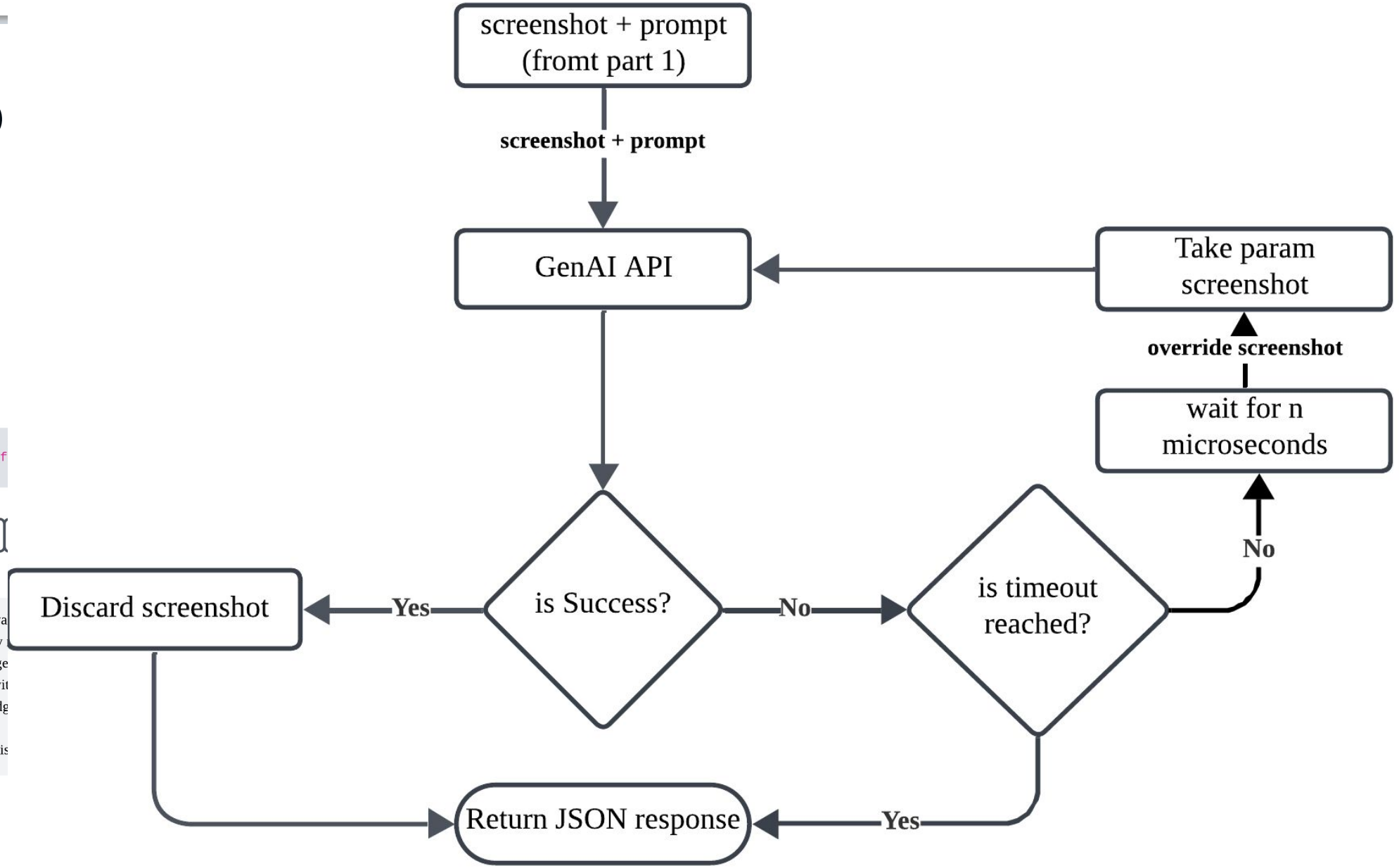


🟥 CRITICAL: Only evaluate the widget that is visually enclosed in a solid red rectangle (outline). This red border uniquely marks the widget to be evaluated.
🚫 Ignore all other widgets, even if they have the same name or are fully visible. Focus only on the one with the red outline.
📸 The red-outlined widget will have its content (chart, table, or data area) directly below its title/header.
👉 If the widget header is not visible, focus on the content area between two vertical red lines.

```
1 Respond only with a JSON
  object:
2         {
3             "result":
  "${expectedText}" // or
  "fail",
4             "message":
  "Reason for the result"
5         }
```

Two Key Components

Pro **ry**




```
1 Check the widget
  "${widgetName}" f
  issues.
```

Start

CRITICAL: Only eva
This red border uniquely
Ignore all other widge
Focus only on the one wit
The red-outlined widg
title/header.
If the widget header is

Gains

Reduction in maintenance	50% reduction in the source code	Change in tests involves simple modification of the prompt	Resilient to minor UI changes than hardcoded assertions	Change in DOM hierarchy/ attributes doesn't cause flaky tests
Faster development	Structured GenAI implementation leads to faster subsequent prompt development	Annotated screenshots in failures with verbose texts, leads to faster debugging	Handles variations (e.g., bar + area charts or new charts) without extra code	
Increased Reliability	Trained on billions of data points, image datasets, AI recognizes all possible interface types in the UI	*Human-Like Visual Understanding*: AI sees the UI like a human would	Detects the Unknown Unknowns: AI can flag unexpected UI anomalies not predefined in code	

Learnings from GenAI

Cost

Image tokens are consumed based on resolution. A full-page image equals 1500 – 1800 tokens

Volume of tests as well as the retry logic, significantly increases the API cost

Replaced full-page screenshots with **concentrated screenshots**. Input tokens reduced by ~40%

Developed *intelligent-image-reducer* to **reduce image resolution**, keeping image ratio intact. This further reduced input tokens by 30%

Increased "wait" between consecutive retries from 400ms to 1000ms. This eliminated rate-limit errors almost entirely

Added rate-limit handling: If error is hit, code waits for a random amt. of time, (1-7 seconds). No consecutive error waits in same or diff. scripts will be same

The code runs for 100+ customers daily. Its equivalent to running it on 100+ diff. platforms every 24 hours.

Limited TPM leads to throttling, leading to the code throwing rate-limit errors

Rate-limit errors

Overkill areas

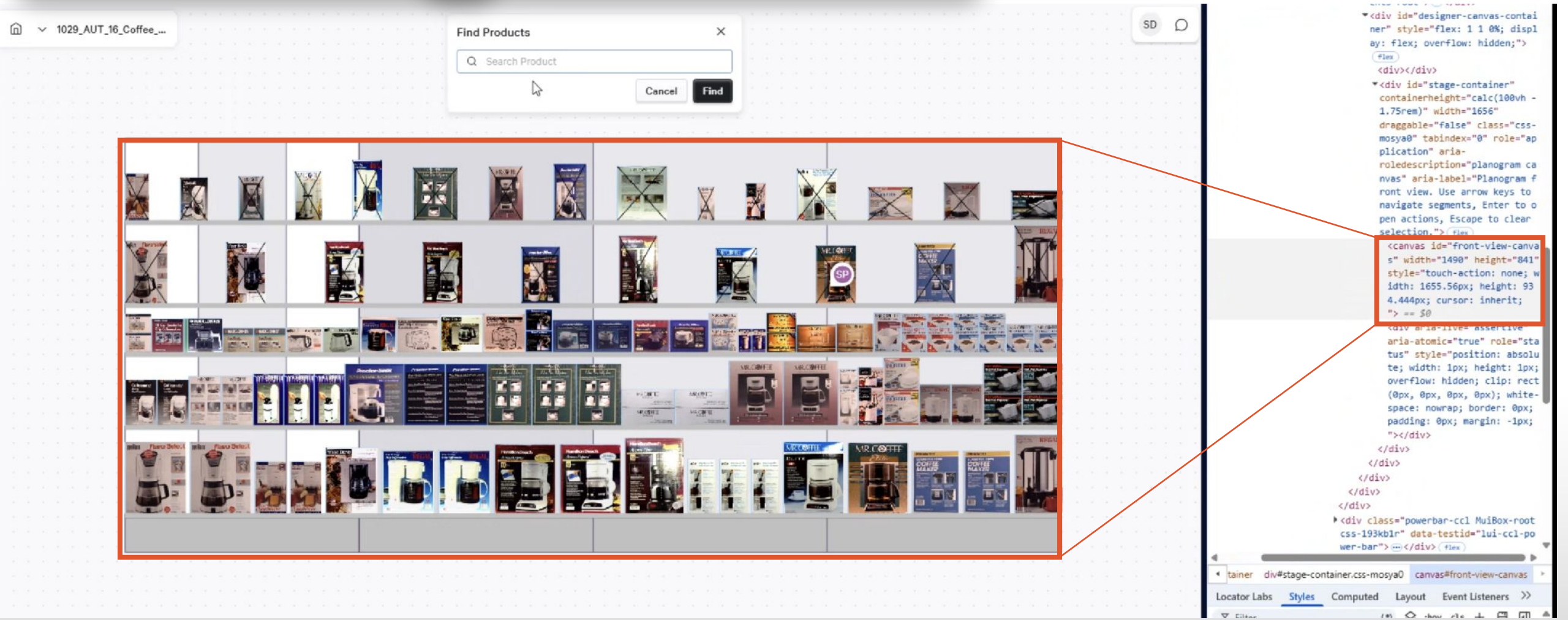
Seeing the gains, we attempted to use GenAI assertions everywhere. But soon, we realised some areas were an overkill and better off with the conventional code

Identified areas where where UI changes were either rare or never. Such **deterministic areas are better handled with conventional code**. E.g., Login flow

What's Next

Planograms

There are no selectors



What's Next

Planograms

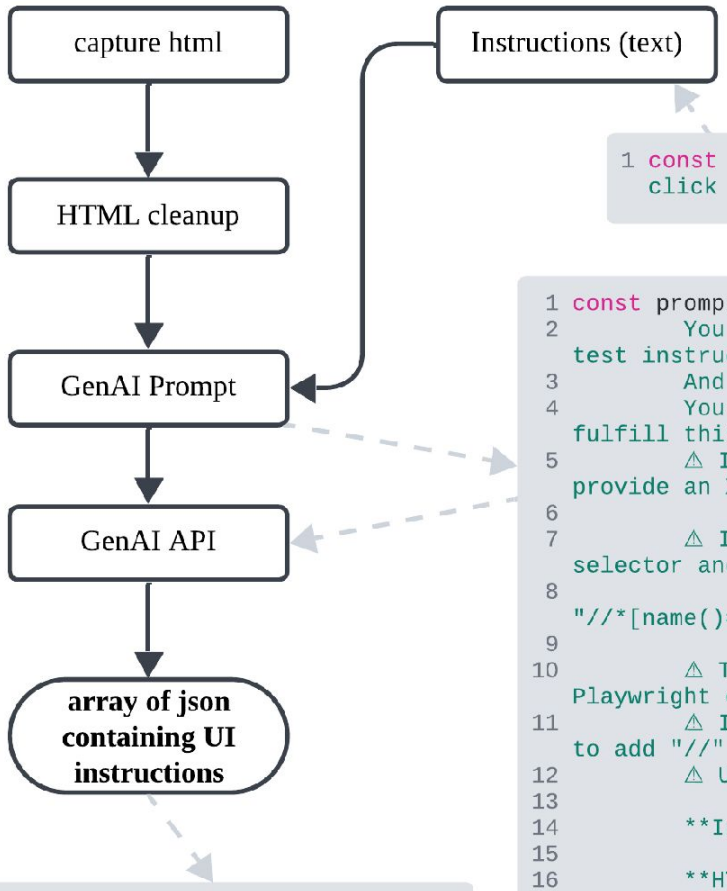
Everything happens in-place

What's in the Future!

Use GenAI for browser actions

What's in the Future!

Finder Agent

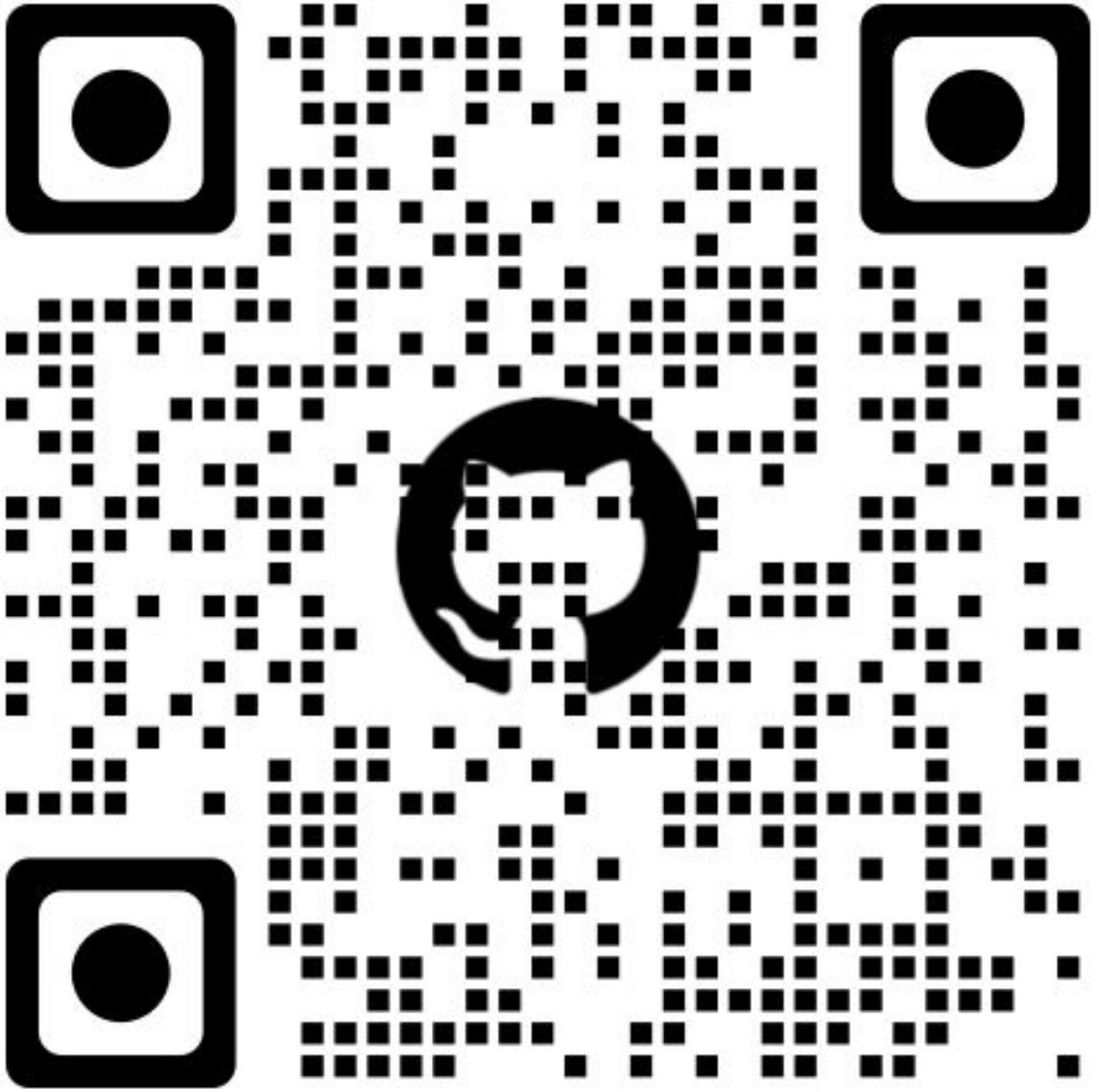



```
1 const instruction = `enter the username "abc@xyz" and
  click the "Next" button`;
```

```
1  const prompt = `
2          You are an expert UI tester. You are given the HTML of a webpage and a
   test instruction.
3          And you are using Playwright as the automation framework.
4          Your task is to identify which selectors (CSS or XPath) are needed to
   fulfill this instruction.
5          ⚠ If you plan to use the text of an element or the word "contains", then
   provide an XPath selector. For everything else, prefer a CSS selector.
6
7          ⚠ If you plan to use "svg" as well as text/"contains", then use an XPath
   selector and make sure to represent it as follows:
8              e.g., "//svg[@data-testid='abc']" should be represented as
   "//*[name()='svg' and @data-testid='abc']"
9
10         ⚠ The selector should be such that it is compatible with Playwright and
   Playwright can interact with the element.
11         ⚠ If you are using the text "following-sibling" in your XPath, make sure
   to add "//" before it to make it a valid XPath.
12         ⚠ Use an XPath selector only if a CSS selector is not possible.
13
14         **Instruction**: ${instruction}
15
16         **HTML**:
17         ${pageHtml}
18
19         **Return a JSON list with each step as follows**:
20         {
21             "action": "fill" | "click" | "select" | "check",
22             "selector": "css selector or text",
23             "value": "optional input text (only for 'fill' or 'select')"
24         }
25         only respond with the JSON and nothing else.
26         `;
```

```
1  [
2      {
3          "action": "fill",
4          "selector": "#email",
5          "value": "abc@xyz.com"
6      },
7      {
8          "action": "click",
9          "selector": "#continue"
10     }
11 ]
```

ViTO boiler plate is live



Disclaimer: No AI was used in generating the slides

Any Questions?