Architecture decision record: browser automation framework for E2E testing (Playwright vs Selenium)

1. **Context**

We are in the process of selecting a browser automation framework for our end-to-end (E2E) testing pipeline. This framework will be integral to our CI/CD processes, running tests that simulate real user interactions on our platform. Specifically, the tests will cover scenarios such as user sign-up/sign-in, file uploads, dashboard interactions, and report downloads.

As a **startup**, our focus is on **agile development**, with a need to rapidly iterate and evolve. Our team predominantly works with **TypeScript** and **Python**, and the ability to write tests in these languages is essential. Additionally, the platform contains **interactive charts and dashboards**, which makes it critical that the automation tool supports rich, dynamic UIs well.

The two contenders for this task are **Playwright** and **Selenium**, each with its strengths and trade-offs. We need to evaluate these frameworks based on the features and requirements outlined below.

2. **Considered Options**

- **Playwright** (by Microsoft)
- **Selenium** (by the Selenium Project)

3. **Decision Drivers**

The factors influencing our decision are as follows:

1. **Agile Development**: The chosen tool must enable fast, flexible development cycles.
2. **Language Support**: Our team requires support for both **TypeScript** and **Python**.
3. **Interactive UI Testing**: The ability to test interactive charts, dashboards, and dynamic elements reliably is essential.
4. **Runtime Speed**: While not a primary concern, performance in CI/CD pipelines is a consideration.
5. **Scalability**: We are not planning for massive scaling in the immediate future, but we want to ensure the solution can handle future growth.
6. **Backwards Compatibility**: Legacy systems and compatibility with older browsers are not critical for our project at this time.
7. **Mobile Testing**: While not an immediate focus, the framework should be capable of testing mobile-responsive features or be extensible for such use cases.
8. **Multi-monitor Testing**: Support for multi-monitor configurations is a secondary requirement, especially if we ever scale to testing for more complex user workflows.
9. **File Upload Testing**: The framework must handle file uploads efficiently, a core requirement of our testing needs.

4. **Evaluation Criteria**

- **Ease of Use**: How easy is it to write and maintain tests?
- **Language Support**: Does the framework support TypeScript and Python, the two languages our team uses most frequently?

- **Interactive UI Testing**: How well does the framework handle complex, interactive user interfaces such as charts, file uploads, and dynamic data?
- **CI/CD Integration**: How well does the framework integrate into common CI/CD tools and services?
- **Cross-browser Support**: What browsers are supported and how well do they perform?
- **Performance and Speed**: How quickly do tests run, especially in a CI/CD pipeline?
- **Scalability**: How well can the framework scale if more tests or more complex scenarios are added?
- **Community and Ecosystem**: How active is the framework's community? Are there plenty of integrations and extensions available?

5. **Considerations**

5.1 Playwright

Pros:

1. **Smarter API for Local File Uploads**: Playwright's API for interacting with local files and performing file uploads is simpler and more intuitive. This would make file upload testing easier to implement and maintain.
2. **Syntax and Code Generation**: Playwright has a shorter, more concise syntax. This results in less boilerplate code, which improves maintainability and developer efficiency. Additionally, this shorter syntax improves OpenAI code generation quality, making it easier to automatically generate test scripts.
3. **Interactive UI Testing**: Playwright excels at testing dynamic, interactive web applications, such as those with rich charts, complex user interactions, and real-time updates. It handles WebSockets, WebRTC, shadow DOMs, and other modern web technologies very effectively.
4. **Cross-browser Support**: Playwright supports **Chromium**, **WebKit**, and **Firefox**. It has consistent performance across these browsers, which should cover most of our testing needs.
5. **CI/CD Integration**: Playwright integrates seamlessly with modern CI/CD platforms (GitHub Actions, Jenkins, etc.). It can run tests in parallel across different browsers, optimizing test run times and making it suitable for rapid development.
6. **Fast and Reliable**: Playwright is faster than Selenium in general, especially in headless mode, and more resilient when dealing with asynchronous web elements.

Cons:

1. **Limited Mobile Testing**: While Playwright does support mobile emulation for browsers, it lacks native mobile testing capabilities like Selenium's integration with Appium for true mobile testing.
2. **Smaller Ecosystem**: Playwright is still newer and less established than Selenium. While it has a rapidly growing community and good documentation, it might not yet have the vast ecosystem of plugins and integrations that Selenium offers.
3. **Limited Browser Support**: While Playwright covers the major modern browsers (Chrome, Safari, Firefox), its support for legacy browsers (e.g., Internet Explorer) is not as robust as Selenium's.

5.2 Selenium

Pros:

1. **Longer History and Maturity**: Selenium has been around for a long time and has a proven track record. It's widely used across many teams and industries, which has led to a vast ecosystem of plugins,

integrations, and resources.
2. **Cross-browser and Cross-platform Support**: Selenium supports a **wide variety of browsers** and versions, including **Internet Explorer**, and can also be integrated with various tools like **Docker**, **Selenium Grid**, and **cloud services** for distributed testing.
3. **Mobile Testing**: Selenium, through its integration with **Appium**, is much more robust for mobile testing, including both Android and iOS applications. This makes it the better choice for projects with a mobile-first or mobile-heavy focus.
4. **Multi-monitor Testing**: Selenium provides better support for scenarios involving **multiple monitors** or complex multi-window interactions.

Cons:

1. **Complexity**: Selenium's API is more verbose and explicit. While this can be an advantage in some cases, it means more code to write and maintain, which may reduce developer agility—especially important in a startup environment.
2. **Performance**: Selenium generally runs slower than Playwright, especially in headless mode. This could impact CI/CD pipelines, particularly as the number of tests grows.
3. **Interactive UI Testing**: Selenium is not as smooth as Playwright when it comes to testing modern, interactive web UIs, particularly with charts and real-time data updates. It requires more setup and handling to reliably interact with dynamic content.

6. **Comparison Summary**

Feature	Playwright	Selenium
Ease of Use	Shorter syntax, more intuitive for modern UIs	More explicit, requires more boilerplate
Language Support	TypeScript, Python, JavaScript	TypeScript, Python, Java, Ruby, C#
Interactive UI Testing	Excellent for dynamic, real-time UIs	Handles basic UIs, but more verbose and complex for rich interactions
File Upload Testing	Smarter API for file uploads	More verbose, less intuitive API
CI/CD Integration	Easy integration with GitHub Actions, Jenkins	Strong integration with many CI tools
Mobile Testing	Limited, only emulation	Full support through Appium
Cross-browser Support	Chromium, WebKit, Firefox	Full support across major and legacy browsers
Performance	Fast, optimized for headless testing	Slower, especially in headless mode
Multi-monitor Testing	Limited	Good support for multi-monitor setups
Community and Ecosystem	Growing, good documentation	Large, mature, extensive ecosystem

7. **Decision**

After considering the requirements and trade-offs, **Playwright** is the better choice for our current needs. Its smarter API for local file upload testing, concise syntax, and strong support for interactive UI testing make it an ideal fit for our agile development cycle. The fact that it supports both **TypeScript** and **Python** is critical for our team, and the framework's modern approach to testing will allow us to write clean, maintainable code.

While **Selenium** remains a great tool, particularly for mobile testing, legacy browser support, and multi-monitor setups, it is less well-suited to our current needs. Its verbosity, slower performance, and more complex handling of dynamic UIs like charts make it less optimal for our use case.

8. **Consequences**

- **Immediate Action**: We will adopt **Playwright** for our E2E testing, focusing on testing user flows involving sign-up, sign-in, file uploads, dashboards, and report downloads.
- **Long-Term Considerations**: We will keep an eye on the evolving Playwright ecosystem. If our needs change, particularly around mobile testing or support for legacy browsers, we may revisit Selenium.
- **Training and Documentation**: Development teams will need to become familiar with Playwright's API, particularly for handling dynamic UIs and file uploads.
- **Migration**: Existing Selenium tests (if any) will be gradually migrated to Playwright.

9. **Future Considerations**