



POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Mariusz Januszek

Nr albumu 34224

Kompleksowe porównanie narzędzi do testowania aplikacji internetowych

Praca magisterska napisana pod
kierunkiem:

**dr inż. Mariusz Trzaska,
prof. PJATK**

Warszawa, sierpień 2026

Streszczenie

Praca dotyczy istotnego problemu, jakim jest wybór odpowiedniego narzędzia do automatyzacji testów typu *End-to-End* aplikacji internetowych. Duża liczba rozwiązań dostępnych na rynku oraz ich odmienna architektura sprawiają, że decyzja o wyborze frameworka dla konkretnego projektu stała się nietrywialnym zadaniem. Wybór ten wpływa bezpośrednio na stabilność testów, koszt ich utrzymania oraz tempo pracy zespołu odpowiedzialnego za jakość oprogramowania.

Praca ma na celu kompleksowe porównanie trzech popularnych frameworków z ekosystemu *JavaScript*, którymi są Playwright, Cypress oraz WebdriverIO. Podstawą porównania jest własna implementacja analogicznych testów w języku *TypeScript*, zorganizowanych w dwunastu kategoriach tematycznych i wykonywanych na lokalnie uruchamianej aplikacji GAD.

Część jakościowa opisuje różnice składniowe i architektoniczne, które ujawniły się dopiero podczas pisania testów. Część ilościowa opiera się na pomiarach każdej z siedemnastu badanych kombinacji frameworka, przeglądarki i trybu równoległości. Obie części zamyka macierz porównawcza wraz z rekomendacjami.

Słowa kluczowe:

testy End-to-End, automatyzacja testów, testowanie aplikacji internetowych, Playwright, Cypress, WebdriverIO, porównanie frameworków, TypeScript

Spis treści

Streszczenie.....	2
1. Wstęp.....	6
1.1. Cel pracy	6
1.2. Rozwiązania przyjęte w pracy	6
1.3. Rezultaty pracy	7
1.4. Organizacja pracy	7
2. Testowanie aplikacji internetowych	8
2.1. Rodzaje testów oprogramowania	8
2.2. Testy End-to-End – charakterystyka i znaczenie	8
2.3. Rola automatyzacji w procesie wytwarzania oprogramowania	9
2.4. Przegląd istniejących porównań narzędzi E2E	9
3. Wykorzystane narzędzia i technologie	11
3.1. Playwright.....	11
3.1.1. Architektura i mechanizm działania	11
3.1.2. Automatyczne oczekiwanie	12
3.1.3. System lokatorów.....	12
3.2. Cypress.....	13
3.2.1. Architektura i mechanizm działania	13
3.2.2. Łańcuch poleceń i mechanizm ponawiania.....	13
3.2.3. Ograniczenia architektoniczne	14
3.3. WebdriverIO	15
3.3.1. Architektura i mechanizm działania	15
3.3.2. Explicit waits i zarządzanie referencjami	16
3.3.3. Ograniczenia architektoniczne	16
3.4. Porównanie konfiguracji i ekosystemu	17
3.5. Aplikacja testowa GAD	17
3.6. Środowisko testowe	18
4. Metodologia badania	19
4.1. Zakres i kryteria porównania	19
4.2. Kategorie testowe – uzasadnienie wyboru.....	20
4.3. Metryki porównawcze	22
4.3.1. Czas wykonania testów	22
4.3.2. Stabilność testów	22
4.3.3. Objętość kodu	23

4.3.4. Zużycie zasobów systemowych	23
4.3.5. Wsparcie raportowania	23
4.4. Metodologia zbierania danych	24
4.5. Decyzje projektowe	25
4.5.1. Brak wzorca Page Object Model	25
4.5.2. Dane testowe osadzone w scenariuszach	25
4.5.3. Aplikacja testowa uruchamiana lokalnie	26
4.5.4. TypeScript jako wspólny język.....	26
4.5.5. Spójna struktura folderów i nazewnictwo plików.....	26
5. Implementacja scenariuszy testowych	27
5.1. Podstawowe interakcje z elementami	27
5.2. Elementy dynamiczne i mechanizmy czekania.....	29
5.3. Zaawansowane selektory i Shadow DOM	31
5.4. Formularze i walidacje.....	33
5.5. Pobieranie plików	37
5.6. Tabele i paginacja	38
5.7. Przechwytywanie sieci i mockowanie API.....	41
5.8. Autoryzacja i zarządzanie sesją	43
5.9. Iframes i zagnieżdżone konteksty	46
5.10. Canvas i Drag & Drop	49
5.11. WebSockets i komunikacja w czasie rzeczywistym	53
5.12. Alerty, okna i pop-upy	57
6. Wyniki pomiarów i analiza porównawcza.....	61
6.1. Czas wykonania testów.....	61
6.1.1. Tryb sekwencyjny	61
6.1.2. Tryb równoległy.....	65
6.1.3. Porównanie między przeglądarkami	67
6.2. Stabilność testów	68
6.3. Objętość kodu	70
6.4. Zużycie zasobów systemowych	72
6.5. Raportowanie	74
6.6. Macierz porównawcza	75
7. Interpretacja wyników	78
7.1. Kluczowe różnice między frameworkami	78
7.2. Ograniczenia poszczególnych narzędzi	79
7.3. Rekomendacje doboru frameworka	80

7.4. Ograniczenia przeprowadzonego badania.....	82
7.5. Kierunki dalszych badań.....	82
8. Podsumowanie.....	84
Bibliografia	85
Wykaz rysunków.....	89
Wykaz tabel	90
Wykaz listingów kodu.....	92

1. Wstęp

Aplikacje internetowe stanowią obecnie podstawowy kanał dostarczania usług cyfrowych, a ich liczba, złożoność i tempo rozwoju systematycznie rosną. Na odbiór produktu oprócz oferowanych funkcjonalności wpływają również takie aspekty jakościowe jak niezawodność działania, responsywność interfejsu oraz spójne zachowanie w różnych przeglądarkach. Zapewnienie tych właściwości wymaga systematycznego testowania oprogramowania na wszystkich etapach jego wytwarzania.

Szczególną rolę w weryfikacji jakości aplikacji internetowych odgrywają testy typu *End-to-End (E2E)*, które symulują rzeczywiste interakcje użytkownika z systemem, począwszy od interfejsu graficznego, przez warstwę logiki aplikacji, aż po komunikację sieciową i trwałość danych. Ze względu na koszt ich ręcznego wykonywania testy *E2E* są współcześnie niemal zawsze automatyzowane, a na rynku dostępnych jest wiele konkurencyjnych narzędzi służących temu celowi. Do najpopularniejszych w ekosystemie Node.js [1] należą obecnie Playwright, Cypress oraz WebdriverIO. Narzędzia te różnią się architekturą, sposobem komunikacji z przeglądarką oraz zakresem wspieranych scenariuszy, przez co wybór frameworka dla konkretnego projektu stał się trudną decyzją inżynierską.

1.1. Cel pracy

Celem pracy jest kompleksowe porównanie trzech frameworków do automatyzacji testów *E2E* aplikacji internetowych: Playwright, Cypress oraz WebdriverIO. Podstawę porównania tworzy własna implementacja analogicznych scenariuszy testowych we wszystkich trzech narzędziach. Obejmuje podstawowe interakcje z elementami strony oraz scenariusze zaawansowane, sprawdzające zachowanie frameworków w technicznie wymagających obszarach współczesnych aplikacji webowych.

Porównanie prowadzone jest w dwóch uzupełniających się wymiarach. Wymiar jakościowy obejmuje analizę różnic składniowych i architektonicznych, które ujawniły się podczas implementacji analogicznych scenariuszy. Wymiar ilościowy opiera się na pomiarach obejmujących między innymi czas wykonania testów, ich stabilność, objętość kodu oraz zużycie zasobów systemowych. Pracę kończy zestawienie zbiorcze oraz wskazania doboru narzędzia.

1.2. Rozwiązania przyjęte w pracy

W pracy przyjęto podejście empiryczne, którego podstawą jest własna implementacja testów wykonana równoległe w trzech narzędziach. Analogiczne scenariusze zaimplementowano niezależnie w każdym z frameworków w języku *TypeScript*, dzięki czemu z porównania wyeliminowano zmienną w postaci języka programowania. Obiektem testów jest edukacyjna aplikacja GAD (*GUI API Demo*) [2], uruchamiana lokalnie, co zapewniło pełną kontrolę nad środowiskiem, powtarzalność pomiarów oraz niezależność od warunków sieciowych.

Dla zachowania spójności metodologicznej podjęto szereg świadomych decyzji projektowych, w tym identyczną strukturę testów we wszystkich frameworkach oraz jednakowy podział scenariuszy na kategorie tematyczne. Pomiary ilościowe przeprowadzono w kontrolowanym środowisku sprzętowym według powtarzalnej procedury pomiarowej. Każdą badaną kombinację frameworka i przeglądarki uruchamiano wielokrotnie, a uzyskane wyniki zagregowano statystycznie. Szczegółowe uzasadnienie tych decyzji przedstawiono w rozdziale 4.

1.3. Rezultaty pracy

Rezultatem części praktycznej jest obszerny zestaw testów *E2E* zaimplementowanych równolegle w trzech frameworkach wraz z infrastrukturą pomiarową. W części badawczej zebrano dane pomiarowe i poddano je analizie porównawczej. W toku badania zidentyfikowano rzeczywiste różnice architektoniczne między narzędziami. Dotyczą one systematycznych ograniczeń poszczególnych frameworków w wybranych obszarach funkcjonalnych oraz odmiennych modeli synchronizacji testu ze stanem aplikacji, które przekładają się na stabilność i zwężłość kodu testowego.

Ponadto pomiary wykazały różnice wydajnościowe między frameworkami i silnikami przeglądarek, przy jednocześnie wysokiej stabilności wyników wszystkich badanych narzędzi. Na podstawie zebranych danych opracowano macierz porównawczą oraz rekomendacje doboru frameworka do typu projektu, przedstawione w rozdziałach 6 i 7.

1.4. Organizacja pracy

Praca składa się z ośmiu rozdziałów. Rozdział 2 wprowadza czytelnika w tematykę testowania aplikacji internetowych. Omówiono w nim rodzaje testów oprogramowania, charakterystykę podejścia *End-to-End*, rolę automatyzacji w procesie wytwarzania oprogramowania oraz istniejące porównania narzędzi tej klasy. Rozdział 3 przedstawia porównywane frameworki wraz z ich architekturą i mechanizmami działania, a także aplikację testową GAD oraz środowisko badawcze. W rozdziale 4 opisano metodologię badania, obejmującą kryteria porównania, kategorie testowe, przyjęte metryki oraz decyzje projektowe. Rozdział 5 jest najobszerniejszą częścią pracy i dokumentuje implementację scenariuszy testowych we wszystkich kategoriach wraz z porównaniem składni trzech narzędzi. W rozdziale 6 zaprezentowano wyniki pomiarów ilościowych oraz ich analizę, natomiast rozdział 7 zawiera interpretację uzyskanych rezultatów, rekomendacje oraz kierunki dalszych badań. Pracę zamyka rozdział 8, w którym podsumowano osiągnięte wyniki.

2. Testowanie aplikacji internetowych

Rozdział opisuje podstawy teoretyczne pracy. Omówiono w nim miejsce testów *E2E* wśród pozostałych rodzajów testów oprogramowania, ich charakterystykę oraz znaczenie automatyzacji w nowoczesnym procesie wytwarzania oprogramowania. Rozdział zamyka przegląd istniejących porównań narzędzi *E2E* wraz z identyfikacją luk uzasadniających podjęcie niniejszego badania.

2.1. Rodzaje testów oprogramowania

Zgodnie z definicją przyjętą w sylabusie ISTQB [3], testowanie oprogramowania jest procesem oceny jakości produktu i redukcji ryzyka wystąpienia awarii w środowisku produkcyjnym. Obejmuje weryfikację zgodności z wymaganiami oraz walidację spełnienia potrzeb użytkowników. W praktyce testy klasyfikuje się według poziomów, na których są przeprowadzane. Najczęściej wyróżnia się:

- **testy jednostkowe** (ang. *unit tests*) – weryfikują pojedyncze, izolowane komponenty kodu (np. funkcje, klasy, moduły), są szybkie i tanie w utrzymaniu oraz precyzyjnie lokalizują defekty,
- **testy integracyjne** (ang. *integration tests*) – sprawdzają poprawność współpracy między komponentami lub systemami, np. komunikację aplikacji z bazą danych lub zewnętrznym API,
- **testy *End-to-End*** – weryfikują kompletne przepływy w systemie z perspektywy użytkownika końcowego, zwykle poprzez automatyczne sterowanie rzeczywistą przeglądarką.

Wzajemną relację tych poziomów opisuje model piramidy testów, spopularyzowany przez Cohna [4]. Zgodnie z nim podstawę strategii testowania powinny stanowić liczne testy jednostkowe, warstwę środkową powinny tworzyć testy integracyjne, a wierzchołek stosunkowo nieliczne testy *E2E*. Uzasadnieniem tej proporcji jest rosnący wraz z poziomem piramidy [5] koszt tworzenia, czas wykonania i podatność na niestabilność testów. Model piramidy nie zmniejsza jednak wartości testów wysokopoziomowych. Wskazuje jedynie, że każdy poziom weryfikuje inną klasę defektów i żaden nie zastępuje pozostałych. Testy *E2E*, choć najkosztowniejsze, jako jedyne potwierdzają poprawność systemu w konfiguracji zbliżonej do produkcyjnej. Szerszy kontekst wytwarzania systemów informatycznych, w tym praktyczne aspekty implementacji wspomnianych poziomów testów w cyklu życia oprogramowania, przedstawiono w [6].

2.2. Testy *End-to-End* – charakterystyka i znaczenie

Główną rolą testu *E2E* jest symulacja zachowania rzeczywistego użytkownika. Test uruchamia przeglądarkę, nawiguje po aplikacji, wchodzi w interakcję z elementami interfejsu (m.in. klika, wypełnia formularze, przeciąga elementy) i weryfikuje obserwowalne rezultaty tych działań. W przeciwieństwie do testów niższych poziomów test *E2E* obejmuje jednocześnie warstwę prezentacji, logikę kliencką, komunikację sieciową i *backend*. Dzięki temu wykrywa defekty integracyjne niewidoczne w izolowanych komponentach, takie jak błędy konfiguracji, niezgodności kontraktów API czy problemy specyficzne dla konkretnego silnika przeglądarki.

Do głównych zalet testów *E2E* należą: wysoka wartość biznesowa weryfikowanych scenariuszy (np. przepływy krytyczne, takie jak logowanie czy finalizacja transakcji), pokrycie integracyjne całego stosu technologicznego oraz skuteczność w wykrywaniu regresji po zmianach w systemie. Wady tego podejścia są równie dobrze widoczne. Testy *E2E* są wolne, ponieważ

pojedynczy scenariusz trwa sekundy lub minuty, podczas gdy test jednostkowy wykonuje się w milisekundach. Są też kosztowne w utrzymaniu, gdyż zmiany w interfejsie użytkownika wymuszają aktualizację selektorów i scenariuszy. Są także podatne na niestabilność (ang. *flakiness*), rozumianą jako skłonność do niedeterministycznych wyników w identycznych warunkach wykonania, bez zmian w kodzie testu ani aplikacji. Luo i inni [7] wskazują, że najczęstszą przyczyną niestabilności są błędy synchronizacji związane z oczekiwaniem asynchronicznym (ang. *async wait*). Tłumaczy to, dlaczego mechanizmy oczekiwania na elementy stały się jednym z głównych pól różnicowania współczesnych frameworków *E2E* i jednym z centralnych wątków niniejszej pracy.

2.3. Rola automatyzacji w procesie wytwarzania oprogramowania

Współczesne metodyki wytwarzania oprogramowania, takie jak podejścia zwinne (ang. *Agile*) oraz kultura DevOps, opierają się na krótkich cyklach wydawniczych i częstej integracji zmian. W takim modelu ręczna regresja całego systemu przed każdym wydaniem jest niewykonalna czasowo i kosztowo. Rozwiązaniem stosowanym w takim modelu jest automatyzacja testów włączona w proces ciągłej integracji i ciągłego dostarczania (CI/CD, ang. *Continuous Integration / Continuous Delivery*) [8], w którym każda porcja zmian jest automatycznie budowana i weryfikowana za pomocą testów, a wynik tej weryfikacji warunkuje dalsze etapy wdrożenia.

Automatyzacja testów *E2E* przynosi w tym kontekście kilka konkretnych korzyści. Skracą pętlę informacji zwrotnej dla programistów, wykonuje systematyczne testowanie na wielu przeglądarkach oraz redukuje długofalowy koszt regresji. W literaturze i praktyce formułowane są również argumenty ostrożnościowe. Zautomatyzowany zestaw testów *E2E* tworzy kod, który sam wymaga utrzymania. Niestabilne testy obniżają zaufanie zespołu do wyników i mogą prowadzić do ignorowania rzeczywistych awarii. Czas wykonania obszernych zestawów *E2E* bywa wąskim gardłem potoku CI/CD. Z tych powodów badane w niniejszej pracy właściwości, czyli szybkość wykonania, stabilność, objętość kodu i ergonomia diagnozowania błędów, bezpośrednio przekładają się na opłacalność automatyzacji w danym projekcie.

2.4. Przegląd istniejących porównań narzędzi E2E

Porównania narzędzi do automatyzacji testów webowych są obecne w źródłach takich jak literatura naukowa czy w materiałach branżowych. Najbliższym tematycznie odniesieniem dla niniejszej pracy jest badanie Monia i Pańczyk [9], w którym porównano trzy narzędzia: Playwright, Cypress oraz WebdriverIO, przyjmując jako kryteria średni czas wykonania testu oraz procentowe zużycie procesora i pamięci operacyjnej. Badanie to opiera się jednak na niewielkim zestawie scenariuszy testowych, co ogranicza reprezentatywność wyników i utrudnia wnioskowanie o zachowaniu narzędzi w trudnych obszarach. Podobny profil ma analiza Tymoshchuka [10], obejmująca narzędzia Selenium, Playwright i Cypress oraz metryki statystyczne (średnia, odchylenie standardowe, współczynnik zmienności), przeprowadzona jednak na trzech scenariuszach o zróżnicowanej złożoności. Zbliżone założenia przyjęto również w pracy Gosik i Miłosz [11], gdzie poddano analizie Cypress i Playwright dla aplikacji zbudowanych we frameworku Angular. Porównano w niej czas wykonania, zużycie pamięci oraz długość kodu testów na kilku scenariuszach.

Kolejną grupą są prace dyplomowe realizowane w PJATK. Pasik [12] przeprowadził analizę narzędzi do testów wydajnościowych aplikacji internetowych. Jest to wzorzec metodologiczny porównania wielokryterialnego, ale dotyczy innej domeny niż testy funkcjonalne *E2E*. Chotza [13]

porównała podejście *E2E* w Selenium i Playwright na podstawie studium przypadku, bez uwzględnienia Cypress oraz bez metryk stabilności mierzonych wielokrotnymi uruchomieniami.

W materiałach branżowych m.in. blogi producentów, raporty firm narzędziowych, dominują zestawienia deklaracyjnych cech oraz proste benchmarki czasowe, często oparte na pojedynczych scenariuszach i pozbawione opisu metodologii umożliwiającego replikację. Na tym tle można wskazać powtarzające się luki w istniejących analizach:

- małe zestawy scenariuszy niepokrywające trudnych obszarów, takich jak Shadow DOM, ramki zagnieżdżone (ang. *iframes*), WebSockets czy zarządzanie wieloma oknami,
- brak wielokrotnych uruchomień pomiarowych, uniemożliwiający rzetelną ocenę stabilności i rozrzutu wyników,
- porównywanie implementacji w różnych językach programowania, wprowadzające dodatkową zmienną zakłócającą,
- ograniczenie analizy do pojedynczej przeglądarki, bez zbadania różnic między ich silnikami,
- pomijanie metryk jakościowo-kosztowych, takich jak objętość kodu potrzebnego do realizacji analogicznego scenariusza.

3. Wykorzystane narzędzia i technologie

Na rynku dostępnych jest wiele narzędzi służących do automatyzacji testów aplikacji webowych. Na potrzeby niniejszej pracy wybrano trzy frameworki: Playwright, Cypress oraz WebdriverIO. Dobór ten wynika z ich pozycji w ekosystemie Node.js [1], odmiennych podejść architektonicznych oraz możliwości pisania testów w języku *TypeScript*.

W rozdziale przedstawiono ogólną charakterystykę każdego z narzędzi, obejmującą ich architekturę oraz główne mechanizmy wpływające na stabilność testów. Szczegóły implementacyjne celowo pominięto, ponieważ zaprezentowano je w rozdziale 5, przy omawianiu konkretnych scenariuszy testowych. Końcowa część rozdziału zawiera zestawienie różnic konfiguracyjnych i ekosystemowych pomiędzy analizowanymi narzędziami, a także opis aplikacji testowej oraz środowiska pomiarowego wykorzystanego w badaniu.

3.1. Playwright

Playwright [14] jest frameworkiem *open-source* do automatyzacji testów *End-to-End*, rozwijanym przez firmę Microsoft od 2020 roku. Powstał z myślą o nowoczesnych aplikacjach webowych i dostarcza wbudowany zestaw mechanizmów:

- własny *test runner*,
- bibliotekę asercji,
- izolację środowiska testowego,
- równoległe wykonywanie testów,
- narzędzia deweloperskie.

Podstawowym środowiskiem uruchomieniowym Playwright jest Node.js, jednak framework umożliwia również tworzenie testów w językach:

- *Python*,
- *C# (.NET)*,
- *Java*.

Narzędzie natywnie obsługuje przeglądarki Chromium, Google Chrome, Microsoft Edge i Firefox oraz silnik WebKit wykorzystywany w przeglądarce Safari.

3.1.1. Architektura i mechanizm działania

Playwright [15] steruje przeglądarkami bezpośrednio za pomocą ich wewnętrznych mechanizmów komunikacyjnych. Ponieważ standardowe wersje Firefox i WebKit nie udostępniają wszystkich funkcji potrzebnych do automatyzacji, Playwright dostarcza własne, zmodyfikowane wersje tych przeglądarek pobierane razem z narzędziem. Za wymianę danych między frameworkiem a przeglądarką odpowiada warstwa transportowa Playwright, oparta na dwukierunkowym kanale komunikacji.

Struktura frameworka opiera się na hierarchicznym modelu obiektów:

- *Browser* – reprezentuje uruchomioną przeglądarkę z wieloma kontekstami,
- *BrowserContext* – izolowana sesja z odrębnymi ciasteczkami i pamięcią lokalną,
- *Page* – pojedyncza karta wewnątrz kontekstu,
- *Frame* – ramka osadzona w obrębie strony.

Każdy *BrowserContext* zapewnia izolację porównywalną z odrębnym profilem przeglądarki, jednak jego utworzenie jest wielokrotnie szybsze niż uruchomienie nowej instancji *Browser*.

3.1.2. Automatyczne oczekiwanie

Główną cechą wyróżniającą Playwright spośród porównywanych narzędzi jest mechanizm automatycznego oczekiwania na elementy (ang. *auto-wait*) [16]. Przed wykonaniem każdej akcji framework weryfikuje następujące warunki:

- lokator odnosi się do dokładnie jednego elementu,
- element jest widoczny,
- element jest aktywny,
- element jest stabilny (nie znajduje się w trakcie animacji),
- element jest zdolny do interakcji (nie jest zasłonięty).

Domyślny czas oczekiwania (ang. *timeout*) dla pojedynczego testu w Playwright [17] wynosi 30 sekund i może być konfigurowany zarówno globalnie, jak i na poziomie pojedynczej operacji. Asercje *expect* cechują się tzw. *retry-ability*. W przypadku asercji wykonywanych na lokatorze, framework cyklicznie ponawia sprawdzenie warunku aż do upłynięcia limitu czasu. Asercje *expect* [18] mają własny, niezależny *timeout*, którego domyślna wartość wynosi 5 sekund. Połączenie *auto-wait* dla akcji oraz *retry-ability* dla asercji usuwa większość typowych przyczyn *flakiness* w testach E2E, bez konieczności jawnego konfigurowania mechanizmów ponawiania.

3.1.3. System lokatorów

Podstawą lokalizacji elementów w Playwright jest system lokatorów (ang. *locators*) [19]. Lokator to deklaracyjny opis sposobu odnalezienia elementu, którego sprawdzenie następuje dopiero w momencie wykonania konkretnej akcji lub asercji. Mechanizm ten określany jest jako leniwa ewaluacja (ang. *lazy evaluation*).

Playwright zaleca konstruowanie lokatorów w oparciu o metody odzwierciedlające sposób, w jaki użytkownik postrzega stronę:

- ***getByRole()*** – wyszukiwanie na podstawie roli ARIA (np. *button*, *link*, *checkbox*),
- ***getByText()*** – wyszukiwanie według treści tekstowej,
- ***getByLabel()*** – wyszukiwanie elementów formularza na podstawie tekstu etykiety,
- ***getByPlaceholder()*** – wyszukiwanie pól wejściowych na podstawie tekstu placeholdera,
- ***getByTestId()*** – wyszukiwanie przy użyciu dedykowanego atrybutu testowego (domyślnie *data-testid*),

- **`getByAltText()`** – wyszukiwanie elementów (najczęściej obrazów) na podstawie tekstu alternatywnego,
- **`locator()`** – uniwersalna metoda przyjmująca selektor CSS lub XPath.

3.2. Cypress

Cypress [20] jest frameworkiem *open-source* do automatyzacji testów *End-to-End*, rozwijanym przez firmę Cypress.io. Pierwsza publicznie dostępna wersja 1.0 ukazała się w 2017 roku. Cypress został zaprojektowany z myślą o programistach *frontend*, co znajduje odzwierciedlenie w jego interaktywnym modelu pracy opartym na graficznym interfejsie (*Test Runner*) oraz wbudowanych mechanizmach wspierających debugowanie testów.

W odróżnieniu od frameworku Playwright, Cypress ogranicza się do jednego ekosystemu technologicznego, oferując wsparcie wyłącznie dla języków *JavaScript* oraz *TypeScript*.

Cypress natywnie obsługuje przeglądarki Chromium, Google Chrome, Microsoft Edge, Firefox i Electron (wersja Chromium z frameworkiem Electron dostarczana przez Cypress). Obsługę silnika WebKit (opartą na buildzie z projektu Playwright) [21] wprowadzono w wersji 10.8, jednak do dziś ma ona status funkcjonalności eksperymentalnej.

3.2.1. Architektura i mechanizm działania

Architektura Cypress [22] zasadniczo odbiega od architektury pozostałych analizowanych narzędzi. Kod testowy wykonywany jest wewnątrz tej samej instancji przeglądarki, w której działa testowana aplikacja. Oznacza to, że sterowanie przeglądarką nie odbywa się za pośrednictwem zewnętrznego protokołu (takiego jak Chrome DevTools Protocol w Playwright czy WebDriver w WebdriverIO). Test działa w tym samym środowisku co aplikacja i ma bezpośredni dostęp do drzewa DOM oraz obiektu ***window***.

Takie podejście skraca czas komunikacji i daje bezpośredni dostęp do stanu aplikacji, jednak wiąże się również z ograniczeniami architektonicznymi, które omówiono szerzej w podrozdziale 3.2.3.

Cypress udostępnia API w postaci tzw. łańcucha komend (ang. *command chain*) [23], czyli sekwencji wywołań rozpoczynających się od globalnego obiektu ***cy***. W przeciwieństwie do synchronicznego modelu *JavaScript*, polecenia nie są wykonywane natychmiast w momencie ich wywołania, lecz trafiają do wewnętrznej kolejki komend (ang. *command queue*), z której Cypress przetwarza je sekwencyjnie, niezależnie od synchronicznego przepływu funkcji testowej. Polecenia nie zwracają wyników w klasyczny sposób. Są one przekazywane kolejnej komendzie w łańcuchu, co jest odmienną logiką w sposobie budowania struktury testów.

3.2.2. Łańcuch poleceń i mechanizm ponawiania

Cypress wykorzystuje model łańcuchowy, w którym kolejne polecenia mogą być łączone w sekwencję wywołań poprzez dołączanie następnych operacji. Wzorec ten znany jest w literaturze inżynierii oprogramowania jako płynny interfejs (ang. *fluent interface*). Termin został wprowadzony przez Fowlera i Evansa [24] w 2005 roku na określenie API, w którym sekwencja łańcuchowych wywołań metod tworzy czytelny, zbliżony do języka naturalnego opis operacji. W przypadku Cypress implementacja tego wzorca różni się jednak od jego klasycznej postaci. Kolejne ogniwa łańcucha nie

zwracają bezpośrednio wyniku wykonania poprzedniej operacji, lecz dodają polecenia do wewnętrznej kolejki komend, która jest wykonywana sekwencyjnie przez framework.

Polecenia takie jak `cy.get()`, `cy.contains()` czy `cy.find()`, mają wbudowany mechanizm automatycznego ponawiania (ang. *retry-ability*). Oznacza to, że w sytuacji gdy element nie istnieje w drzewie DOM lub powiązana asercja nie jest spełniona, framework cyklicznie ponawia próbę aż do powodzenia operacji albo do upłynięcia limitu czasu.

Domyślny limit czasu, określony parametrem `defaultCommandTimeout` [25], wynosi 4 sekundy i może być konfigurowany zarówno globalnie, jak i na poziomie pojedynczego polecenia.

Mechanizm *retry-ability* pełni funkcję zbliżoną do mechanizmu *auto-wait* w Playwright, jednak opiera się na odmiennej koncepcji:

- w Playwright *auto-wait* dotyczy akcji (np. kliknięcia, wprowadzania tekstu) i weryfikuje warunki dostępności elementu przed ich wykonaniem,
- w Cypress *retry* odnosi się głównie do zapytań (np. `cy.get()`) oraz powiązanych z nimi asercji (`should()`), ponawiając cały łańcuch operacji aż do spełnienia warunku.

Polecenia modyfikujące stan aplikacji, takie jak `.click()` czy `.type()`, sprawdzają dostępność elementu przed wykonaniem operacji, jednak samą akcję wykonują tylko raz.

Asynchroniczna kolejka komend ma jeszcze jedną konsekwencję. Polecenia Cypress nie zwracają wyniku operacji, lecz obiekt łańcucha dopisujący kolejne ogniwo do kolejki. Sam wynik, określany jako podmiot (ang. *subject*), jest przekazywany dopiero w trakcie wykonania i pozostaje zwykłym obiektem *JavaScript*. Tych wyników nie można przypisywać do zmiennych [26], ani oczekiwać operatorem *await*. Dostęp do nich zapewnia metoda `.then()` oraz aliasy tworzone poleceniem `.as()`.

3.2.3. Ograniczenia architektoniczne

Architektura Cypress, oparta na wykonywaniu testów bezpośrednio w przeglądarce, wiąże się z szeregiem ograniczeń, które twórcy narzędzia określają jako świadome kompromisy projektowe (ang. *trade-offs*) [27]. Do najważniejszych należą:

- **brak obsługi wielu kart przeglądarki** – Cypress nie umożliwia kontrolowania więcej niż jednej otwartej karty ani wielu instancji przeglądarki w ramach jednego testu. Scenariusze tego typu wymagają zastosowania dodatkowych narzędzi (np. integracji z Puppeteer [28]),
- **ograniczenia w testowaniu *cross-origin*** – pojedynczy test jest domyślnie powiązany z jedną superdomeną. Realizacja scenariuszy obejmujących różne źródła wymaga użycia polecenia `cy.origin()`,
- **brak natywnej równoległości lokalnej** – Cypress nie oferuje wbudowanego mechanizmu równoległego wykonywania testów w środowisku lokalnym. Równoległość możliwa jest z wykorzystaniem usługi Cypress Cloud [29], która dystrybuuje testy pomiędzy różne maszyny w środowisku ciągłej integracji (ang. *Continuous Integration*).

Ograniczenia te wyznaczają zakres scenariuszy możliwych do realizacji w Cypress i wpływają na sposób ich projektowania.

3.3. WebdriverIO

WebdriverIO [30] to framework *open-source* przeznaczony do automatyzacji testów *E2E* aplikacji webowych. Jego najnowsza stabilna wersja 9 [31] ukazała się 15 sierpnia 2024 roku, natomiast projekt jest rozwijany od 2011 roku. Narzędzie oferuje bibliotekę poleceń wraz z wbudowanym mechanizmem służącym do uruchamiania testów. Integracja z Appium [32] pozwala na rozszerzenie zakresu zastosowania WebdriverIO o testowanie aplikacji mobilnych działających m.in. na systemach Android i iOS.

Podobnie jak w przypadku Cypress, WebdriverIO jest przeznaczone do działania w środowisku Node.js, a tym samym oferuje wsparcie dla języków programowania *JavaScript* i *TypeScript*.

Framework obsługuje następujące przeglądarki i odpowiadające im sterowniki:

- Chrome – ChromeDriver
- Firefox – GeckoDriver
- Microsoft Edge – Edge Driver
- Internet Explorer – InternetExplorerDriver
- Safari – SafariDriver (tylko na macOS)

Sterownik pełni funkcję pośrednika pomiędzy frameworkiem a automatyzowaną przeglądarką, tłumacząc polecenia automatyzacji na operacje wykonywane w środowisku przeglądarki. WebdriverIO od wersji 8.14 umożliwia również automatyczne pobieranie i konfigurowanie sterowników, dzięki czemu ich ręczna instalacja nie jest wymagana w typowych konfiguracjach.

3.3.1. Architektura i mechanizm działania

WebdriverIO kieruje przeglądarką używając standardu W3C WebDriver [33]. Standard ten opisuje sposób zdalnego sterowania przeglądarką, niezależny od języka programowania oraz systemu operacyjnego. Dzięki tej architekturze kod testowy działa w każdej przeglądarce, w której producent zapewnia sterownik zgodny ze standardem.

Framework zbudowany jest z trzech warstw [34]:

- **warstwa protokołu** (ang. *Protocol Bindings*) – podstawowe polecenia standardu WebDriver, przekazujące odpowiedź sterownika bez żadnej dodatkowej obsługi,
- **warstwa poleceń** (ang. *Standalone Mode*) – wygodniejsze polecenia nałożone na protokół np. system selektorów,
- **The WDIO Testrunner** – warstwa uruchamiająca testy, dzieląca pliki ze scenariuszami między równoległe działające procesy i zarządzająca sesjami przeglądarki.

W kodzie testów wykorzystywane są dwie z trzech opisanych warstw. Warstwa protokołu nie jest wywoływana bezpośrednio przez testy, lecz stanowi podstawę komunikacji pomiędzy frameworkiem a przeglądarką.

Pojedyncze polecenie testowe np. kliknięcie w przycisk, odpowiada jednemu poleceniu protokołu WebDriver, realizowanemu jako osobne żądanie HTTP. Trafia ono do sterownika przeglądarki (ang. *browser driver*), czyli procesu pośredniczącego, który tłumaczy polecenia i wykonuje je w przeglądarce [35].

3.3.2. Explicit waits i zarządzanie referencjami

Synchronizacja w WebdriverIO jest częściowo zautomatyzowana. Zgodnie z dokumentacją [36] framework sam oczekuje, aż element stanie się interaktywny. Zakres warunków uwzględnianych podczas automatycznego oczekiwania jest jednak węższy niż w przypadku mechanizmu *auto-wait* dostępnego w Playwright, dlatego w niektórych scenariuszach konieczne jest zastosowanie jawnych mechanizmów oczekiwania (ang. *explicit waits*).

Najczęściej wykorzystywane metody automatycznego oczekiwania obejmują:

- **`waitForExist()`** – oczekiwanie na pojawienie się elementu w drzewie DOM,
- **`waitForDisplayed()`** – oczekiwanie, aż element stanie się widoczny,
- **`waitForClickable()`** – oczekiwanie, aż element stanie się możliwy do kliknięcia,
- **`waitForEnabled()`** – oczekiwanie, aż element będzie dostępny,
- **`waitUntil()`** – oczekiwanie na spełnienie dowolnego warunku zdefiniowanego przez użytkownika.

Domyślny limit czasu dla wymienionych metod definiowany jest globalnie w pliku konfiguracyjnym lub indywidualnie dla pojedynczego wywołania.

Konieczność stosowania jawnych metod oczekiwania w kodzie testowym pokazuje jedną z głównych różnic względem Playwright oraz Cypress, gdzie analogiczne mechanizmy oczekiwania realizowane są w dużej mierze automatycznie przez framework.

Drugą cechą charakterystyczną architektury WebdriverIO jest sposób zarządzania referencjami do elementów DOM. Metoda **`$()`** zwraca obiekt reprezentujący element, którego stan jest weryfikowany względem aktualnego drzewa DOM w momencie wykonania operacji. Z kolei metoda **`$$()`** zwraca zmaterializowaną kolekcję elementów (ang. *materialized collection*), czyli zbiór referencji ustalonych w momencie jej wywołania. W konsekwencji, jeżeli pomiędzy pobraniem kolekcji a wykonaniem operacji struktura DOM ulegnie zmianie, np. w wyniku przeładowania listy elementów, przechowywane referencje mogą nie odpowiadać już aktualnemu stanowi DOM.

Problem ten nie występuje w Playwright dzięki mechanizmowi leniwej ewaluacji lokatorów, natomiast w Cypress jego wpływ ograniczany jest przez wbudowany mechanizm *retry-ability*.

3.3.3. Ograniczenia architektoniczne

Dużym ograniczeniem WebdriverIO w wersji 9 jest brak użytecznego w praktyce mechanizmu mockowania żądań sieciowych. Klasyczny tryb W3C WebDriver nie przewiduje natywnego API do przechwytywania ruchu sieciowego, co ogranicza możliwość korzystania z mechanizmów takich jak **`browser.mock()`** bez wykorzystania rozszerzeń opartych na CDP lub protokołu WebDriver BiDi [37]. Sytuacja ta jest konsekwencją szerszej zmiany architektonicznej w wersji 9, w której usunięto pakiet devtools. We wcześniejszych wersjach implementował on komunikację z przeglądarką poprzez Puppeteer i stanowił podstawę mockowania ograniczonego do przeglądarek opartych na rodzinie przeglądarek Chromium. Zespół utrzymujący projekt uzasadnił tę zmianę dążeniem do koncentracji na standardzie W3C oraz przyjęciem protokołu WebDriver BiDi jako standardu uzupełniającego klasyczny WebDriver.

Alternatywą pozostaje tryb BiDi, zdefiniowany w odrębnej specyfikacji, jednak jego implementacja w WebdriverIO w wersji 9 pozostaje eksperymentalna i niestabilna. Sesje testowe ulegają nieoczekiwanym przerwaniom, co utrudnia praktyczne wykorzystanie tego mechanizmu. Z tego powodu we wszystkich pomiarach opisanych w pracy wymuszono klasyczny tryb W3C WebDriver.

3.4. Porównanie konfiguracji i ekosystemu

W poprzednich podrozdziałach omówiono każdy z trzech frameworków z osobna, ze szczególnym uwzględnieniem ich architektury wewnętrznej oraz mechanizmów wpływających na projektowanie testów. W niniejszym podrozdziale zestawiono wszystkie trzy narzędzia obok siebie, koncentrując się na praktycznych aspektach konfiguracji.

Tabela 1 zestawia najważniejsze parametry konfiguracyjne i cechy ekosystemu wszystkich trzech porównywanych frameworków.

Tabela 1. Porównanie konfiguracji i ekosystemu trzech frameworków E2E (stan na maj 2026). Źródło: opracowanie własne na podstawie dokumentacji [14], [20], [30].

Aspekt	Playwright	Cypress	WebdriverIO
Plik konfiguracyjny	playwright.config.ts	cypress.config.ts	wdio.conf.ts
Rozszerzenie plików testowych	.spec.ts	.cy.ts	.spec.ts
Test Runner	@playwright/test (wbudowany)	Mocha (wbudowany)	Mocha, Jasmine lub Cucumber
Biblioteka asercji	expect (wbudowana)	Chai (wbudowana)	expect-webdriverio (wbudowana) Chai (zewnętrzna)
Domyślna przeglądarka	Chromium (build Playwright)	Electron (Chromium)	Chrome (przez chromedriver)
Wsparcie WebKit	tak	eksperymentalne	nie
Równoległość lokalna	tak [38]	brak (wymaga Cypress Cloud [39])	tak [40]
Domyślne limity czasu	limit testu 30s, asercje expect 5s	4s	10s (w badanej konfiguracji)
Mechanizm oczekiwania	auto-wait (akcje + asercje)	retry-ability (zapytania + asercje)	explicit waits + ograniczony auto-wait
Natywny tryb interaktywny	UI Mode, Inspector	Test Runner GUI	brak
Integracja z Allure	allure-playwright	allure-cypress	@wdio/allure-reporter
Licencja	Apache 2.0	MIT	MIT

3.5. Aplikacja testowa GAD

Jako wspólny obiekt testów dla wszystkich trzech frameworków wykorzystano aplikację GAD (GUI API Demo) [41]. Jest to aplikacja webowa *open-source* stworzona i utrzymywana przez zespół

jaktestowac.pl [42]. Aplikację przygotowano z myślą o nauce automatyzacji testów i ćwiczeniu technik kontroli jakości na scenariuszach odzwierciedlających realne problemy współczesnych aplikacji webowych.

GAD składa się z trzech zasadniczych warstw:

- **Warstwa interfejsu użytkownika (GUI)** – zawiera elementy charakterystyczne dla współczesnych aplikacji webowych m.in. formularze, tabele, listy dynamiczne, okna modalne czy alerty,
- **Warstwa REST API** – zestaw punktów końcowych (ang. *endpoints*) udostępniających operacje CRUD na zasobach: artykułach, komentarzach, użytkownikach,
- **Warstwa danych** – prosta baza danych w formacie JSON z możliwością czyszczenia i przywracania stanu początkowego.

Aplikacja udostępnia również sekcję *practice*, zawierającą zestaw stron przygotowanych pod konkretne wyzwania automatyzacyjne, w tym scenariusze logowania, autoryzacji i zarządzania sesją.

3.6. Środowisko testowe

Wszystkie pomiary opisane w pracy przeprowadzono w jednolitym środowisku sprzętowym i programowym. Zastosowanie tego samego komputera oraz tej samej konfiguracji oprogramowania dla wszystkich narzędzi ogranicza wpływ różnic infrastrukturalnych na uzyskane wyniki i zapewnia bezpośrednią porównywalność metryk. Specyfikację środowiska przedstawiono w Tabeli 2.

Tabela 2. Specyfikacja środowiska testowego. Źródło: opracowanie własne.

Komponent	Specyfikacja
System operacyjny	Windows 11 Pro 64-bit
Procesor	AMD Ryzen 5 7600
Pamięć RAM	32 GB
Node.js	22.14.0
PowerShell	5.1
Google Chrome	147+
Mozilla Firefox	148+
Microsoft Edge	147+
Aplikacja GAD	v2.8.3
Playwright	1.56.1
Cypress	15.9.0
WebdriverIO	9.24.0

4. Metodologia badania

W niniejszym rozdziale przedstawiono ramy metodologiczne przeprowadzonego badania. Zdefiniowano zakres analizy, uzasadniono wybór kategorii testowych oraz opisano zestaw metryk wykorzystanych do oceny frameworków. Rozdział zamyka opis procedury zbierania danych oraz decyzji projektowych podjętych z myślą o wiarygodności i powtarzalności wyników.

4.1. Zakres i kryteria porównania

Przedmiotem badania są trzy frameworki służące do automatyzacji testów typu *End-to-End* (E2E) aplikacji internetowych:

- Playwright [14] w wersji 1.56.1,
- Cypress [20] w wersji 15.9.0,
- WebdriverIO [30] w wersji 9.24.0.

O wyborze narzędzi zdecydowała ich pozycja w ekosystemie *JavaScript*, mierzona liczbą pobrań z rejestru npm [1] w latach 2022–2025. Jak przedstawiono na Rysunku 1, najszybszy wzrost liczby pobrań odnotowano dla Playwright, wolniejszy dla Cypress, a najniższy dla WebdriverIO.

Zestawienie tych narzędzi pozwala na porównanie trzech różnych podejść do automatyzacji testów:

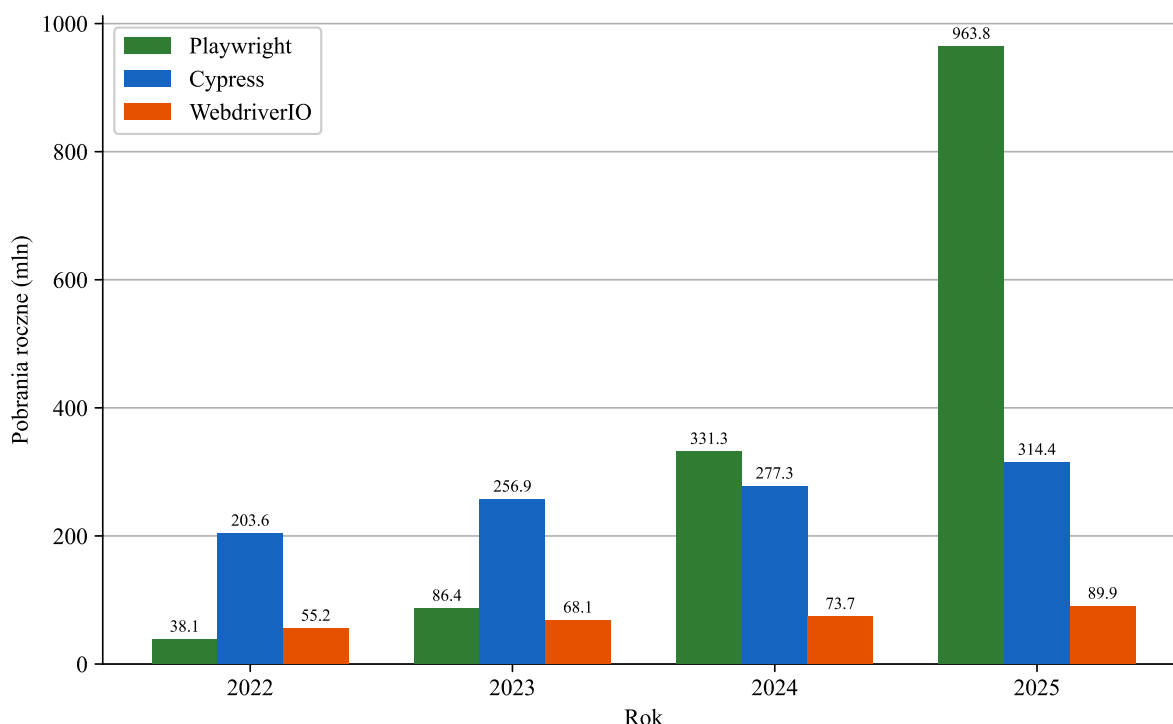
- **Playwright** – najmłodsze z porównywanych narzędzi, sterujące przeglądarką przez natywne protokoły,
- **Cypress** – framework o ugruntowanej pozycji wśród deweloperów *frontend*,
- **WebdriverIO** – klasyczne rozwiązanie oparte na standardzie W3C WebDriver.

Zakres porównania obejmuje łącznie 1492 testy typu *End-to-End*, zorganizowane w 12 kategoriach testowych. Wszystkie scenariusze zostały zaimplementowane od podstaw w języku *TypeScript*. Zastosowanie wspólnego języka programowania eliminuje wpływ różnic technologicznych na wyniki porównania.

Do badania włączono scenariusze spełniające następujące kryteria:

- odwzorowanie typowych wyzwań spotykanych w testach E2E aplikacji internetowych,
- możliwość implementacji w co najmniej dwóch analizowanych frameworkach,
- potencjał do ujawnienia różnic architektonicznych między narzędziami.

Scenariusze trywialne, w których frameworki zachowują się identycznie, zostały celowo pominięte. Różnice w liczbie testów między frameworkami wynikają z ograniczeń technicznych poszczególnych narzędzi, a nie z niekompletności implementacji. W Tabeli 3 zestawiono kategorie, w których liczba testów różni się między frameworkami wraz z uzasadnieniem.



Rysunek 1. Dynamika pobrań rocznych narzędzi *E2E* z serwisu npm w latach 2022–2025.
Źródło: opracowanie własne na podstawie danych z npm-stat.com [1].

Tabela 3. Różnice w liczbie przygotowanych testów między frameworkami.
Źródło: opracowanie własne.

Kategoria	Playwright	Cypress	WebdriverIO	Uzasadnienie
5. Pobieranie plików	38	34	21	Ograniczenia WebdriverIO oraz Cypress w weryfikacji zawartości pobranych plików.
7. Przechwytywanie sieci i mockowanie API	57	57	0	Niedziałająca metoda <code>browser.mock()</code> w WebdriverIO v9+ [43].
12. Alerty, okna i pop-upy	75	73	75	Architektoniczne ograniczenie Cypress – brak obsługi wielu kart przeglądarki (ang. <i>multi-tab</i>) [27].

4.2. Kategorie testowe – uzasadnienie wyboru

Zdefiniowano 12 kategorii testowych, z których każda odpowiada odrębnemu wyzwaniu technicznemu spotykanemu w praktyce testowania aplikacji internetowych. Przy doborze kategorii kierowano się dwoma głównymi kryteriami:

- pokryciem szerokiego spektrum interakcji użytkownika z aplikacją,
- potencjałem ujawniania różnic architektonicznych – analogicznie jak przy doborze scenariuszy przedstawionych w podrozdziale 4.1.

Kategorie obejmują podstawowe interakcje użytkownika, jak również zaawansowane scenariusze związane z obsługą dynamicznych elementów, komunikacją sieciową oraz zarządzaniem sesją użytkownika.

Wyszczególniono trzy kategorie:

- elementy dynamiczne i mechanizmy czekania,
- selektory i Shadow DOM,
- przechwytywanie ruchu sieciowego i mockowanie API.

Ujawniają one fundamentalne różnice w podejściu do automatyzacji testów. W Tabeli 4 przedstawiono pełną listę kategorii wraz z krótkim uzasadnieniem różnic oraz liczbą testów.

Tabela 4. Kategorie testowe – uzasadnienie różnic i liczba testów.

Źródło: opracowanie własne.

Nr	Kategoria	Liczba testów (PW/CY/WDIO)	Aspekt różnicujący narzędzia
1	Podstawowe interakcje z elementami	8 / 8 / 8	Selekcja elementów bez stabilnych ID – porównanie strategii lokatorów.
2	Elementy dynamiczne i mechanizmy czekania	35 / 35 / 35	Bezpośrednie porównanie trzech modeli synchronizacji: <i>auto-wait</i> (Playwright), <i>auto-retry</i> (Cypress) oraz <i>explicit waits</i> (WebdriverIO).
3	Zaawansowane selektory i Shadow DOM	25 / 25 / 25	Automatyczna penetracja Shadow DOM (Playwright) kontra manualna konfiguracja (Cypress, WebdriverIO).
4	Formularze i walidacje	42 / 42 / 42	Wywoływanie zdarzeń <i>blur</i> , obsługa natywnej walidacji HTML5.
5	Pobieranie plików	38 / 34 / 21	Ograniczenia WebdriverIO i Cypress w weryfikacji zawartości plików.
6	Tabele i paginacja	40 / 40 / 40	Iteracja po dynamicznie ładowanych danych, sortowanie.
7	Przechwytywanie sieci i mockowanie API	57 / 57 / 0	Niedziałająca metoda <code>browser.mock()</code> w WebdriverIO v9+ – dyskwalifikacja frameworka w scenariuszach mockowania API.
8	Autoryzacja i zarządzanie sesją	61 / 61 / 61	Obsługa <i>cookies</i> , <i>session storage</i> , natywnych dialogów logowania.
9	Iframes i zagnieżdżone konteksty	43 / 43 / 43	Dostęp bez przełączania kontekstu (Playwright, Cypress) vs jawne przełączanie aktywnego kontekstu (WebdriverIO).
10	Canvas i Drag & Drop	59 / 59 / 59	Reaktywne oczekiwanie na zakończenie animacji (Playwright) kontra stałe opóźnienie (Cypress, WebdriverIO).

Nr	Kategoria	Liczba testów (PW/CY/WDIO)	Aspekt różnicujący narzędzia
11	WebSockets i komunikacja <i>real-time</i>	41 / 41 / 41	Podgląd ramek połączenia i reaktywne wykrycie połączenia (Playwright) kontra weryfikacja wyłącznie przez stan interfejsu (Cypress, WebdriverIO).
12	Alerty, okna i pop-upy	75 / 73 / 75	Brak natywnej obsługi wielu kart (<i>multi-tab</i>) w Cypress.

4.3. Metryki porównawcze

Do oceny analizowanych frameworków zastosowano zestaw pięciu metryk:

- czas wykonania testów,
- stabilność testów,
- objętość kodu,
- zużycie zasobów systemowych,
- wsparcie raportowania.

Metryki dobrano w taki sposób, aby umożliwić ocenę narzędzi pod względem wydajności, wygody użytkowania oraz kosztu implementacji testów.

4.3.1. Czas wykonania testów

Metrykę zdefiniowano jako całkowity czas wykonania zestawu testów, liczony od uruchomienia narzędzia testowego do zakończenia ostatniego testu. Pomiar przeprowadzono w dwóch trybach: sekwencyjnym i równoległym. Tryb sekwencyjny pozwala porównać wydajność frameworków w warunkach izolowanych, a równoległy ocenić ich skalowalność.

W celu zredukowania wpływu jednorazowych anomalii na wyniki, każdą konfigurację (framework × przeglądarka) uruchomiono dziesięciokrotnie w trybie sekwencyjnym oraz w przypadku Playwright i WebdriverIO pięciokrotnie w trybie równoległym. Dla trybu równoległego przyjęto cztery procesy robocze (ang. *workers*) jako wartość pośrednią między połową rdzeni fizycznych a połową wątków logicznych, zbliżoną do domyślnej heurystyki Playwright [38]. Zebrane pomiary poddano agregacji statystycznej, obejmującej średnią arytmetyczną, medianę, odchylenie standardowe oraz wartości minimalne i maksymalne.

4.3.2. Stabilność testów

Stabilność testów określono na podstawie współczynnika niestabilności (ang. *flakiness*) [7], definiowanego jako skłonność testu do dawania niedeterministycznych wyników (*pass/fail*) w identycznych warunkach wykonania, bez zmian w kodzie testu, ani w testowanej aplikacji.

Przyjęto następujące kryterium klasyfikacji: test uznawano za niestabilny, jeśli w serii uruchomień danej kombinacji (10 w trybie sekwencyjnym, 5 w równoległym) wystąpił co najmniej jeden wynik pozytywny i co najmniej jeden negatywny. Testy kończące się niepowodzeniem we wszystkich uruchomieniach (ang. *always fail*) traktowano jako deterministyczne. Przyczyny ich niepowodzenia omówiono w rozdziale 6.

Współczynnik *flakiness* obliczono jako stosunek liczby testów niestabilnych do łącznej liczby testów w danej konfiguracji (framework × przeglądarka), wyrażony w procentach.

4.3.3. Objętość kodu

Objętość kodu zmierzono przy użyciu metryki LOC (*Lines of Code*). Pomiar wykonano narzędziem *cloc* [44] z filtrem dla języka *TypeScript*. Dzieli ono linie na trzy kategorie: puste (ang. *blank*), komentarz (ang. *comment*) oraz właściwy kod (ang. *code*). Analizie poddano wyłącznie kategorię *code*. Wyniki przedstawiono w dwóch ujęciach:

- liczba linii kodu w kategorii testowej,
- liczba linii kodu przypadająca na pojedynczy test.

Normalizacja do postaci LOC/test pokazuje różnice w liczbie testów pomiędzy frameworkami. Wyższą wartość tej metryki interpretowano jako większy koszt implementacyjny scenariusza testowego.

4.3.4. Zużycie zasobów systemowych

Mierzono wykorzystanie pamięci operacyjnej oraz obciążenie procesora podczas wykonywania testów. Pomiar zrealizowano za pomocą dedykowanego skryptu *PowerShell*, który próbował zużycie zasobów co jedną sekundę. Dla każdego frameworka zbierano wartości średnie oraz maksymalne.

Metryka ta ma charakter pomocniczy. Służy do identyfikacji anomalii oraz ogólnego porównania obciążenia systemu przez poszczególne frameworki.

4.3.5. Wsparcie raportowania

Ocena wsparcia raportowania miała charakter jakościowy. Analiza obejmowała:

- natywne raportowanie,
- integrację z Allure,
- zrzuty ekranu,
- nagrania wideo,
- *trace viewer*.

W Tabeli 5 zestawiono natywne możliwości raportowania każdego z narzędzi.

Tabela 5. Porównanie natywnych mechanizmów raportowania.
Źródło: opracowanie własne.

Aspekt	Playwright	Cypress	WebdriverIO
Natywny reporter	HTML Reporter [45]	Spec Reporter [46]	Spec Reporter [47]
Integracja z Allure	tak	tak	tak
Zrzut ekranu po niepowodzeniu	screenshot: 'only-on-failure'	automatyczne	hook afterTest (manualny)

Aspekt	Playwright	Cypress	WebdriverIO
Nagrywanie wideo	konfiguracja (video: 'on-first-retry') [48]	automatyczne [49]	brak
Trace viewer	wbudowany [50]	brak	brak

4.4. Metodologia zbierania danych

Aby wyniki były rzetelne i powtarzalne, opracowano jednolitą procedurę pomiarową obejmującą wszystkie zdefiniowane wcześniej metryki. Zaprojektowano ją tak, aby minimalizować wpływ czynników zewnętrznych oraz umożliwić pełne powtórzenie badania.

Pomiarami sterował orkiestrator zaimplementowany jako skrypt *PowerShell*. Odpowiadał on za przygotowanie środowiska przed każdym uruchomieniem, nadzór nad przebiegiem testów oraz zebranie wyników. Wspólnym źródłem danych o statusach testów był format Allure, obsługiwany przez wszystkie trzy frameworki. Dedykowane skrypty przekształcały surowe raporty do plików CSV i agregowały je statystycznie. Przerwa dziesięciu minut między kolejnymi uruchomieniami stabilizuje stan systemu operacyjnego. W konfiguracji WebdriverIO wymuszono klasyczny tryb protokołu W3C WebDriver flagą **`wdio:enforceWebDriverClassic: true`**, z przyczyn omówionych w podrozdziale 3.3.3.

Procedura pomiarowa składa się z następujących kroków:

- **Przygotowanie środowiska** – uruchomienie aplikacji GAD pod adresem **`localhost:3000`**, zamknięcie zbędnych procesów w tle oraz zapewnienie stabilnego stanu systemu operacyjnego;
- **Pomiar czasu wykonania** – każdy framework uruchamiano dziesięciokrotnie w trybie sekwencyjnym (1 proces roboczy), a Playwright i WebdriverIO dodatkowo pięciokrotnie w trybie równoległym (4 procesy robocze). Czas wykonania odczytywano z natywnego podsumowania generowanego przez narzędzie testowe (Playwright: *summary*, Cypress: *Run Finished*, WebdriverIO: *Spec Files*). Wyniki zapisywano w pliku CSV;
- **Pomiar stabilności** – seria dziesięciu uruchomień dla każdej pary framework × przeglądarka. Wyniki (sukces/niepowodzenie) każdego testu rejestrowano w raportach Allure, a następnie porównywano statusy między uruchomieniami. Testy o zmiennym statusie klasyfikowano jako niestabilne zgodnie z definicją z podrozdziału 4.3.2;
- **Pomiar LOC** – jednorazowy pomiar narzędziem *cloc* z podziałem na kategorie. Skrypt *PowerShell* iteruje po folderach kategorii w każdym repozytorium, a wyniki zapisuje do pliku CSV;
- **Pomiar RAM/CPU** – monitor zasobów uruchamiany równoległe z testami, próbkujący co jedną sekundę zużycie pamięci i procesora przez procesy uczestniczące w wykonaniu testów. Wyniki eksportowano do pliku CSV i agregowano skryptami pomiarowymi;
- **Ocena raportowania** – przegląd natywnych raportów oraz raportów Allure wygenerowanych po uruchomieniach. Ocena jakościowa zgodnie z kryteriami z podrozdziału 4.3.5.

Osobnego omówienia wymaga sposób identyfikacji procesów objętych pomiarem zasobów. W pierwotnej wersji monitor budował drzewo procesów na podstawie identyfikatora procesu nadrzędnego, wychodząc od procesu uruchamiającego testy. Podejście to poprawnie obejmowało procesy potomne przeglądarek Firefox i Edge, jednak walidacja pomiarów wykazała, że gubi ono część

procesów renderujących silników Chromium oraz WebKit. System operacyjny przepina je do innych procesów macierzystych, przez co przestają one być widoczne jako potomkowie procesu testowego. Z tego powodu monitor przeprojektowano na podejście oparte na porównywaniu migawek listy procesów. Przed rozpoczęciem uruchomienia skrypt rejestruje działające już w systemie procesy o nazwach charakterystycznych dla frameworków, sterowników i przeglądarek, traktując je jako tło. Podczas pomiaru do zużycia zalicza wyłącznie procesy o tych nazwach, które pojawiły się po starcie uruchomienia, niezależnie od ich relacji z procesem nadrzędnym. Procesy aplikacji GAD monitorowane są odrębnie i nie wliczają się do wyniku frameworka.

Mimo tej zmiany pomiar nie obejmuje pełnego drzewa procesów przeglądarek uruchamianych przez zewnętrzne sterowniki, co dotyczy przede wszystkim WebdriverIO. Zarejestrowane wartości należy traktować jako dolne oszacowania rzeczywistego zużycia, a porównania między frameworkami interpretować w ujęciu względnym. Konsekwencje tego ograniczenia dla interpretacji wyników omówiono w podrozdziale 6.4, a jego wpływ na wiarygodność badania w podrozdziale 7.4.

Wszystkie pomiary wykonano na tym samym komputerze i w tym samym środowisku (specyfikacja w podrozdziale 3.6), co eliminuje zmienność wynikającą z opóźnień sieciowych i różnic infrastrukturalnych.

4.5. Decyzje projektowe

W trakcie implementacji testów podjęto kilka decyzji projektowych, których wspólnym celem było porównanie narzędzi w tych samych warunkach. Dotyczyły one architektury projektów testowych, organizacji danych oraz środowiska wykonawczego.

4.5.1. Brak wzorca Page Object Model

W żadnym z analizowanych projektów testowych nie zastosowano wzorca *Page Object Model* (POM). Decyzja ta wynikała z dążenia do możliwie bezpośredniego porównania natywnych możliwości frameworków. Zastosowanie dodatkowej warstwy abstrakcji w postaci POM mogłoby maskować różnice w ergonomii API oraz wpływać na objętość kodu. Implementacja testów bezpośrednio na selektorach i asercjach oferowanych przez framework pozwala ocenić go precyzyjniej, a także przeprowadzić wiarygodne porównanie zwięzłości i czytelności kodu.

W środowisku produkcyjnym stosowanie wzorca POM jest jednak powszechnie rekomendowane. Według dokumentacji Playwright [51] wzorzec ten ułatwia porządkowanie obszernych zestawów testów oraz ich późniejsze utrzymanie. Obiekty strony przechowują selektory elementów w jednym miejscu, tworząc API wielokrotnego użytku, które eliminuje powtarzanie kodu. W niniejszym badaniu zrezygnowano z POM wyłącznie w celu zachowania spójności metodologicznej między porównywanymi frameworkami.

4.5.2. Dane testowe osadzone w scenariuszach

W analizowanych scenariuszach dane testowe zostały osadzone bezpośrednio w plikach testowych, zamiast wydzielać je do zewnętrznych plików z danymi testowymi (ang. *fixtures*). Takie podejście daje możliwość zachowania samowystarczalności każdego scenariusza oraz usuwa dodatkową warstwę abstrakcji, która mogłaby wpływać na objętość kodu lub utrudniać analizę różnic pomiędzy narzędziami. Dzięki temu pełny kontekst testu znajduje się w jednym pliku.

Świadomą ceną tej decyzji jest częściowe duplikowanie danych między scenariuszami. W środowisku produkcyjnym stosowanie *fixtures* pozostaje rekomendowaną praktyką, zwłaszcza przy dużej liczbie testów korzystających z identycznych zestawów danych.

4.5.3. Aplikacja testowa uruchamiana lokalnie

Jako obiekt testów wybrano aplikację GAD (*GUI API Demo*) uruchamianą lokalnie na adresie **localhost:3000**. Decyzja ta dała kontrolę nad środowiskiem testowym i zwiększyła powtarzalność wyników.

Uruchomienie aplikacji lokalnie usuwa z porównania wpływ czynników zewnętrznych, takich jak opóźnienia sieciowe, niestabilność usług zewnętrznych czy zmiany w infrastrukturze. Dodatkowo możliwe było resetowanie stanu aplikacji pomiędzy kolejnymi uruchomieniami testów, co ograniczyło ryzyko wystąpienia efektów ubocznych. Ograniczenia wynikające z użycia jednej aplikacji testowej omówiono szerzej w podrozdziale 7.4.

4.5.4. TypeScript jako wspólny język

Wszystkie projekty testowe zaimplementowano w języku *TypeScript*. Wybór wspólnego języka eliminuje dodatkową zmienną wpływającą na wyniki porównania. Różnice w implementacji wynikają wyłącznie z architektury i możliwości frameworków, a nie z cech języka programowania.

TypeScript jest obecnie dominującym językiem w ekosystemie *JavaScript*. Według ankiety „State of JavaScript 2025” [52], przeprowadzonej wśród około 12 000 deweloperów, 40% respondentów deklaroowało wyłączone użycie *TypeScript*, co pokazuje wzrost o 6 punktów procentowych w porównaniu z poprzednim rokiem. Jedynie 6% ankietowanych używało wyłącznie *JavaScript*. Dane te wskazują na wzrost popularności *TypeScript*.

Dodatkowym argumentem przemawiającym za wyborem *TypeScript* jest jego natywna obsługa we wszystkich analizowanych frameworkach, co pozwoliło na zachowanie spójności implementacji oraz ograniczenie dodatkowej konfiguracji.

4.5.5. Spójna struktura folderów i nazewnictwo plików

W każdym z analizowanych repozytoriów zastosowano identyczną strukturę folderów odpowiadającą dwunastu kategoriom testowym. Zachowano również analogiczne nazewnictwo plików testowych, różniące się jedynie rozszerzeniem charakterystycznym dla danego frameworka (**.spec.ts** dla Playwright i WebdriverIO, **.cy.ts** dla Cypress).

Decyzja ta ma znaczenie organizacyjne i metodologiczne, ponieważ daje możliwość bezpośredniego porównania analogicznych scenariuszy między frameworkami oraz jednoznacznego przypisania metryk (czasu wykonania, LOC) do konkretnej kategorii w każdym projekcie. Spójna struktura katalogów poprawia także czytelność pracy.

5. Implementacja scenariuszy testowych

W niniejszym rozdziale przedstawiono sposób implementacji scenariuszy testowych w trzech analizowanych narzędziach. Każdy podrozdział odpowiada jednej z dwunastu kategorii testowych i zawiera krótki opis scenariuszy, porównanie implementacji we wszystkich frameworkach z fragmentami kodu oraz obserwacje dotyczące różnic architektonicznych i ograniczeń.

Fragmenty kodu pochodzą bezpośrednio z repozytoriów projektów testowych. W celu zachowania zwięzłości, listingi ograniczono wyłącznie do fragmentów obrazujących omawiane zagadnienia. Analogiczne implementacje prezentowane są w stałej kolejności: Playwright, Cypress, WebdriverIO. Pełne pliki testowe dostępne są w repozytoriach autora.

Listingi w niniejszym rozdziale opatrzone kolorowaniem składni wzorowanym na wyświetlaniu kodu w edytorze. Znaczenie poszczególnych kolorów zestawiono w Tabeli 6.

Tabela 6. Konwencja kolorowania składni stosowana w listingach kodu.
Źródło: opracowanie własne.

Przykład	Znaczenie
<code>const, await</code>	słowa kluczowe języka
<code>getByRole()</code>	wywołania funkcji i metod
<code>buttonLocator</code>	zmienne
<code>.length</code>	właściwości i pola obiektów
<code>'tekst'</code>	łańcuchy znaków
<code>5000, true</code>	liczby i wartości logiczne
<code>// komentarz</code>	komentarze

5.1. Podstawowe interakcje z elementami

Pierwsza kategoria obejmuje podstawowe interakcje z elementami strony, których nie można jednoznacznie zidentyfikować za pomocą stabilnych atrybutów ID. Już na tym etapie widoczna jest zasadnicza różnica w podejściu do lokalizacji elementów. Playwright posługuje się leniwymi lokatorami (ang. *lazy locators*) [19], które przechowują jedynie zapytanie selekcyjne i rozwiązują odwołanie do elementu dopiero w momencie wykonania akcji. Cypress operuje na łańcuchu poleceń z mechanizmem automatycznego ponawiania (ang. *auto-retry*) [25], wykorzystując między innymi selektory zgodne z jQuery. WebdriverIO w wyniku wywołania `$$()` zwraca zmaterializowaną tablicę elementów (ang. *materialized collection*) – statyczną migawkę stanu DOM z chwili pobrania.

Jak przedstawiono na Listingu 1, w Playwright lokator tworzony jest deklaratywnie metodą `getByRole()` z metodą pozycyjną `.first()`, a o synchronizację z aplikacją dba wbudowany mechanizm oczekiwania.

Listing 1. Lokalizacja i kliknięcie elementu w Playwright.

```
const firstBookNowButton = page.getByRole('button', { name: 'Book Now'
}).first();
await firstBookNowButton.click();
```

W Cypress selekcja realizowana jest przez selektor `:contains()` w połączeniu z metodą pozycyjną `.first()`. Całość stanowi element łańcucha poleceń z automatycznym ponawianiem (Listing 2).

Listing 2. Lokalizacja i kliknięcie elementu w Cypress.

```
const bookNowButtonsSelector = 'button:contains("Book Now")';
cy.get(bookNowButtonsSelector).first().click();
```

Jak pokazano na Listingu 3, w WebdriverIO wywołanie `$$()` materializuje tablicę elementów, a wskazanie konkretnej karty następuje przez ręczne indeksowanie.

Listing 3. Lokalizacja i kliknięcie elementu w WebdriverIO.

```
const bookNowButtons = await $$('button=Book Now');
const firstBookNowButton = bookNowButtons[0];
await firstBookNowButton.click();
```

Różnice między tymi trzema modelami najlepiej widać w wariancie, w którym lista elementów zmienia się w trakcie testu. Ponieważ tablica zwrócona przez `$$()` jest statyczną migawką, po dodaniu nowej karty do DOM, odwołanie do wcześniej pobranej tablicy może zakończyć się wyjątkiem **StaleElementReferenceException** (nieaktualnej referencji do elementu). W praktyce wymaga to ponownego pobrania kolekcji elementów po zmianach w DOM wpływających na jej zawartość. Ponowne pobranie kolekcji przedstawia Listing 4.

Listing 4. Ponowne pobranie tablicy elementów po modyfikacji w DOM w WebdriverIO.

```
const placeCards = await $$('#places > div');
const newPlaceCard = placeCards[placeCards.length - 1];
```

W Playwright i Cypress ten sam scenariusz nie wymaga żadnych dodatkowych kroków. Leniwy lokator Playwright oraz łańcuch poleceń Cypress odwołują się do elementu w chwili wykonania akcji, co automatycznie uwzględnia aktualny stan DOM. Zestawienie zastosowanych podejść do selekcji elementów przedstawiono w Tabeli 7.

Tabela 7. Mechanizmy selekcji i indeksowania elementów w pierwszej kategorii testowej.
Źródło: opracowanie własne.

Cecha	Playwright	Cypress	WebdriverIO
Selekcja po tekście	<code>getByRole({ name })</code>	<code>:contains()</code>	<code>\$\$('button=...')</code>
Wskazanie pozycji	<code>first()</code> , <code>nth()</code> , <code>last()</code>	<code>first()</code> , <code>eq()</code> , <code>last()</code>	indeksowanie <code>[0]</code> , <code>[n-1]</code>
Model referencji	leniwy lokator	łańcuch poleceń	zmaterializowana tablica
Reakcja na zmianę DOM	automatyczna	automatyczna	wymaga ponownego pobrania
Synchronizacja	automatyczna	automatyczna	częściowo automatyczna + jawne oczekiwanie

5.2. Elementy dynamiczne i mechanizmy czekania

Druga kategoria obejmuje scenariusze z elementami pojawiającymi się i zmieniającymi stan w czasie. Wykorzystano stronę, na której przycisk pojawia się po około dwóch sekundach, a po kolejnych dwóch przechodzi ze stanu nieaktywnego do aktywnego – łącznie daje to cztery sekundy opóźnienia. Dalsze scenariusze obejmują przejścia stanów elementu oraz oczekiwanie na zniknięcie wskaźnika ładowania przy nieskończonym przewijaniu.

Kategoria ta kładzie nacisk na odmienne podejście modelu synchronizacji testu z aplikacją. Playwright opiera się na mechanizmie *auto-wait* [16] wbudowanym w każdą akcję. Cypress stosuje *auto-retry* polecenia aż do upływu limitu czasu. WebdriverIO posiada mechanizmy automatycznego oczekiwania dla wybranych operacji, jednak w przypadku bardziej złożonych warunków synchronizacji lub dłuższych opóźnień często wymaga zastosowania jawnych mechanizmów oczekiwania (ang. *explicit wait*).

W Playwright kliknięcie wykonywane jest bez dodatkowej synchronizacji. Przedstawiony na Listingu 5 mechanizm *auto-wait* samodzielnie wstrzymuje akcję, aż element pojawi się na stronie i będzie gotowy do kliknięcia.

Listing 5. Interakcja z opóźnionym elementem w Playwright.

```
const buttonLocator = page.getByTestId('dti-button-element-1');
await buttonLocator.click();
```

W Cypress automatyczne ponawianie osiąga podobny efekt, jednak domyślny limit czasu polecenia **defaultCommandTimeout** wynosi 4 sekundy. To mniej niż element potrzebuje, aby stać się gotowym do interakcji, dlatego w teście jawnie podniesiono limit (Listing 6).

Listing 6. Interakcja z opóźnionym elementem w Cypress.

```
const buttonSelector = '[data-testid="dti-button-element-1"]';
cy.get(buttonSelector, { timeout: 10000 })
  .should('be.visible').and('be.enabled').click();
```

W WebdriverIO synchronizację zapewnia wywołana przed kliknięciem metoda **waitForClickable()** z jawnie określonym limitem czasu (Listing 7).

Listing 7. Interakcja z opóźnionym elementem w WebdriverIO.

```
const buttonLocator = await $('[data-testid="dti-button-element-1"]');
await buttonLocator.waitForClickable({ timeout: 10000 });
await buttonLocator.click();
```

Te same prawidłowości dotyczą oczekiwania na zmianę stanu elementu. W scenariuszu weryfikującym przejście ze stanu **NOT PRESENT**, przez **PRESENT BUT DISABLED**, do **ENABLED** Playwright wykorzystuje asercje automatycznie oczekujące na spełnienie warunku, Cypress realizuje ten sam efekt poprzez mechanizm *auto-retry* asercji, natomiast WebdriverIO wymaga oddzielnych wywołań **waitForDisplayed()** oraz **waitForEnabled()**.

Odrębnym przypadkiem jest oczekiwanie na zniknięcie elementu, zilustrowane scenariuszem ze wskaźnikiem ładowania (ang. *loading spinner*) przy nieskończonym przewijaniu. Po przewinięciu strony, wskaźnik pojawia się na czas pobierania kolejnych elementów i znika po zakończeniu

ładowania. Test oczekuje najpierw na jego pojawienie się, a potem na zniknięcie. Każdy framework realizuje oczekiwanie na zniknięcie odmiennym podejściem.

W Playwright służy do tego zanegowana asercja widoczności **`not.toBeVisible()`** (Listing 8).

Listing 8. Oczekiwanie na zniknięcie wskaźnika ładowania w Playwright.

```
await expect(loadingIndicator).toBeVisible();
await expect(loadingIndicator).not.toBeVisible();
```

W Cypress stosowana jest ponawiana asercja **`should('not.be.visible')`** z podniesionym limitem czasu (Listing 9).

Listing 9. Oczekiwanie na zniknięcie wskaźnika ładowania w Cypress.

```
cy.get(loadingIndicatorSelector).should('be.visible');
cy.get(loadingIndicatorSelector, { timeout: 5000 }).should('not.be.visible');
```

W WebdriverIO ten sam efekt uzyskuje się flagą **`reverse`** w metodzie **`waitForDisplayed()`** (Listing 10).

Listing 10. Oczekiwanie na zniknięcie wskaźnika ładowania w WebdriverIO.

```
await loadingIndicator.waitForDisplayed({ timeout: 5000 });
await loadingIndicator.waitForDisplayed({ timeout: 10000, reverse: true });
```

Zestawienie mechanizmów oczekiwania przedstawiono w Tabeli 8.

Tabela 8. Mechanizmy oczekiwania w drugiej kategorii testowej.
Źródło: opracowanie własne.

Cecha	Playwright	Cypress	WebdriverIO
Model oczekiwania	<i>auto-wait</i>	<i>auto-retry</i>	częściowo automatyczne + jawne
Domyślne limity czasu	limit testu (30s), asercje <i>expect</i> (5s)	<code>defaultCommandTimeout</code> (4s)	argument <code>timeout</code> (10s)
Oczekiwanie na pojawienie	automatyczne	automatyczne	<code>waitForDisplayed()</code>
Oczekiwanie na aktywację	<code>toBeEnabled()</code>	<code>should('be.enabled')</code>	<code>waitForEnabled()</code>
Oczekiwanie na zniknięcie	<code>not.toBeVisible()</code>	<code>should('not.be.visible')</code>	<code>waitForDisplayed({ reverse: true })</code>

Implementacja w Cypress i WebdriverIO pokazuje praktyczną konsekwencję silniejszego oparcia synchronizacji na jawnych limitach czasu. Muszą one być świadomie dobierane do opóźnień występujących w aplikacji. W Cypress oznacza to podniesienie limitu polecenia, a w WebdriverIO podaniem go wprost przy operacji oczekiwania. Dobór limitów czasu staje się zatem częścią kodu testu, a nie jest winą architektury frameworka.

5.3. Zaawansowane selektory i Shadow DOM

Trzecia kategoria dotyczy selektorów trudnych do jednoznacznego wskazania. Obejmuje trzy przypadki:

- elementy z dynamicznie generowanymi identyfikatorami,
- elementy ukryte w pojedynczym i zagnieżdżonym Shadow DOM,
- zamknięty Shadow DOM.

Kategoria ta dobrze pokazuje różnice między narzędziami pod względem nakładu pracy i czytelności kodu.

Pierwszym przypadkiem są dynamiczne identyfikatory. Atrybut **ID** jest tu generowany na nowo przy każdym załadowaniu strony, więc nie może służyć jako podstawa selekcji. Element trzeba zatem wskazać po stabilnym atrybucie, takim jak **data-testid**, albo po jego treści.

Playwright i Cypress oferują do tego związane, deklaratywne filtry treści. WebdriverIO daje również możliwość wybrania elementów po tekście, używając selektorów częściowo tekstowych ***=** lub wyrażeń XPath. Brakuje mu jednak równie krótkiego filtra całego wiersza. W analizowanym scenariuszu, który wymagał dopasowania wiersza i dalszej pracy na nim, zastosowano ręczną iterację.

Jak pokazano na Listingu 11, Playwright odnajduje wiersz deklaratywnie metodą **filter({ hasText })**.

Listing 11. Selekcja wiersza po treści przy dynamicznych ID w Playwright.

```
const rowLocator = page.locator('tr').filter({ hasText: cityName });
await expect(rowLocator).toContainText('Japan');
```

Analogiczne zadanie w Cypress realizuje **contains()** w połączeniu z nawigacją **parent()** (Listing 12).

Listing 12. Selekcja wiersza po treści przy dynamicznych ID w Cypress.

```
cy.get('tr').contains(cityName).parent('tr').as('cityRow');
cy.get('@cityRow').should('contain.text', 'Japan');
```

Odmienne wygląda to w WebdriverIO. Przy podejściu iteracyjnym konieczne jest pobranie kolekcji wierszy i sekwencyjne porównywanie ich treści, co zwiększa liczbę operacji i podatność kodu na błędy (Listing 13).

Listing 13. Selekcja wiersza po treści przy dynamicznych ID w WebdriverIO.

```
const rows = await $$('tr');
let targetRow: WebdriverIO.Element | undefined;
for (const row of rows) {
  const text = await row.getText();
  if (text.includes(cityName)) {
    targetRow = row;
    break;
  }
}
expect(await targetRow!.getText()).to.include('Japan');
```

Drugim wyzwaniem jest Shadow DOM. Jest to odseparowane drzewo DOM ukryte za granicą *shadow root*. Sposób jej przekroczenia zasadniczo różnicuje frameworki. Playwright automatycznie przenika przez otwarty (ang. *open*) Shadow DOM, dzięki czemu element lokalizowany jest tak samo jak w zwykłym DOM (Listing 14). Cypress wymaga jawnego wejścia do *shadow root* metodą `.shadow()` [53] (Listing 15), co zwiększa liczbę operacji niezbędnych do uzyskania dostępu. WebdriverIO stosuje analogiczne, jawne otwarcie *shadow root* metodą `shadow$()` [54] wywołaną na elemencie-gospodarzu (*ShadowHost*). Dostęp do Shadow DOM w WebdriverIO przedstawia Listing 16.

Listing 14. Dostęp do Shadow DOM w Playwright.

```
const shadowInput = page.getByTestId('shadow-name-input');
await shadowInput.fill(testName);
```

Listing 15. Dostęp do Shadow DOM w Cypress.

```
cy.get('#shadow-host').shadow().find('input[data-testid="shadow-name-input"]').clear().type(testName);
```

Listing 16. Dostęp do Shadow DOM w WebdriverIO.

```
const shadowHost = await $('#shadow-host');
const shadowInput = await shadowHost.shadow$('[data-testid="shadow-name-input"]');
await shadowInput.setValue(testName);
```

Różnica pogłębia się przy zagnieżdżonym Shadow DOM, gdzie element kryje się za dwiema granicami *shadow root*. W Playwright głębokość zagnieżdżenia nie wpływa na kod – odwołanie pozostaje pojedyncze. W Cypress zastosowano natomiast opcję `includeShadowDom: true`, która automatycznie przenika wszystkie poziomy (Listing 17).

Listing 17. Dostęp do zagnieżdżonego Shadow DOM w Cypress.

```
cy.get('[data-testid="nested-shadow-name-input"]', { includeShadowDom: true })
  .clear().type(testName);
```

W badanej konfiguracji WebdriverIO (tryb WebDriver Classic) posłużono się jawnym łańcuchem wywołań `shadow$()`, po jednym dla każdej granicy. Jego długość rośnie liniowo wraz z głębokością zagnieżdżenia. Warto zaznaczyć, że w trybie WebDriver BiDi, WebdriverIO przenika Shadow DOM automatycznie poprzez polecenia `$` i `$$`. Jawny łańcuch wynikał zatem z protokołu zastosowanego w pomiarach, a nie z ograniczenia samego frameworka. To właśnie liniowy wzrost długości łańcucha, a nie sama obecność Shadow DOM, obciąża w tym podejściu utrzymanie testów przy złożonych komponentach (Listing 18).

Listing 18. Dostęp do zagnieżdżonego Shadow DOM w WebdriverIO.

```
const deeperHost = await nestedShadowHost.shadow$('#deeper-nested-shadow-host');
const nameInput = await deeperHost.shadow$('[data-testid="nested-shadow-name-input"]');
```

Zamknięty (ang. *closed*) Shadow DOM pozostaje niedostępny we wszystkich trzech frameworkach. Zestawienie sposobów obsługi trudnych selektorów przedstawiono w Tabeli 9.

Tabela 9. Obsługa trudnych selektorów w trzeciej kategorii testowej.
Źródło: opracowanie własne.

Wyzwanie	Playwright	Cypress	WebdriverIO
Dynamiczne ID (filtr po tekście)	<code>filter({ hasText })</code>	<code>contains()</code>	<code>*=</code> , XPath lub pętla <code>for...of</code>
Shadow DOM (poziom 1)	automatyczna penetracja	<code>.shadow().find()</code>	<code>shadow\$()</code>
Zagnieżdżony Shadow DOM	automatyczna penetracja	<code>includeShadowDom: true</code>	łańcuch <code>shadow\$()</code>
Zamknięty (<i>closed</i>) Shadow DOM	niedostępny	niedostępny	niedostępny

Z perspektywy inżyniera automatyzacji testów przedstawione różnice przekładają się na kilka praktycznych aspektów. Zwięzłe, deklaratywne mechanizmy, takie jak filtry treści oraz automatyczna penetracja Shadow DOM, ograniczają ilość kodu pomocniczego. Obniża to koszt utrzymania testów i ułatwia ich czytanie osobom dołączającym do projektu. Rozwiązania wymagające ręcznej iteracji lub jawnego łańcucha przejść przez granice *shadow root* są z kolei bardziej podatne na błędy i wrażliwsze na zmiany w strukturze aplikacji. Przykładowo dodanie kolejnego poziomu zagnieżdżenia podczas refaktoryzacji komponentu wymusza modyfikację kodu testu. W konsekwencji wybór frameworka wpływa na zapis scenariusza, a także na długoterminową odporność zestawu testów na zmiany. Każdy dodatkowy poziom zagnieżdżenia komponentu wymusza w tym podejściu zmianę w kodzie testu.

5.4. Formularze i walidacje

Czwarta kategoria obejmuje obsługę formularzy oraz weryfikację mechanizmów walidacji danych wejściowych. Różnice między frameworkami ujawniają się w pięciu obszarach:

- sposób wprowadzania danych,
- mechanizm wyzwalania zdarzenia utraty fokusu (ang. *blur*),
- asercje na dynamicznie zmienianym stanie elementu,
- odczyt natywnych atrybutów walidacji HTML5,
- przesyłanie plików.

Pierwszym analizowanym aspektem jest sposób wprowadzania danych do pól formularza. W badanej implementacji w Playwright tekst wprowadzano metodą `fill()`. Wprowadzanie wartości do pola w Playwright ilustruje Listing 19.

Listing 19. Wprowadzenie wartości do pola w Playwright.

```
await emailInput.fill('invalidemail.com');
```

W Cypress wykorzystano przedstawioną na Listingu 20 metodę `type()`, która symuluje wciskanie kolejnych klawiszy.

Listing 20. Wprowadzenie wartości do pola w Cypress.

```
cy.get(emailInputSelector).type('invalidemail.com');
```

W WebdriverIO zastosowano metodę **setValue()**, która zastępuje dotychczasową wartość pola (Listing 21).

Listing 21. Wprowadzenie wartości do pola w WebdriverIO.

```
await emailInput.setValue('invalidemail.com');
```

Drugim aspektem jest wyzwalanie zdarzenia utraty fokusu (ang. *blur*), od którego zależy uruchomienie walidacji pola. W Playwright fokus odbierany jest dedykowaną metodą **blur()** (Listing 22).

Listing 22. Wyzwolenie zdarzenia *blur* w Playwright.

```
await emailInput.blur();
```

W Cypress utratę fokusu wymusza metoda **blur()** w łańcuchu poleceń (Listing 23).

Listing 23. Wyzwolenie zdarzenia *blur* w Cypress.

```
cy.get(emailInputSelector).focus().blur();
```

WebdriverIO nie udostępnia natomiast dedykowanej metody **element.blur()**, dlatego w badanej implementacji zdarzenie wyzwalano przez przeniesienie fokusu na inny element, poprzez kliknięcie w element **body**, co ilustruje Listing 24.

Listing 24. Wyzwolenie zdarzenia *blur* przez kliknięcie poza polem w WebdriverIO.

```
await $('body').click(); // blur
```

Trzecim aspektem jest asercja na dynamicznie zmienianym stanie elementu polegająca na dodaniu klasy **valid** do walidacji. W Playwright wykorzystano asercję **toHaveClass()**, której wbudowany mechanizm automatycznego oczekiwania (ang. *auto-wait*) ponawia sprawdzenie aż do spełnienia warunku lub upływu limitu czasu (Listing 25).

Listing 25. Weryfikacja stanu walidacji w Playwright.

```
await expect(emailValidIcon).toHaveClass(/bvalid\b/);
```

W Cypress analogiczną rolę pełni asercja **should('have.class', ...)** z automatycznym ponawianiem (ang. *auto-retry*) zaprezentowana na Listingu 26.

Listing 26. Weryfikacja stanu walidacji w Cypress.

```
cy.get(emailValidIconSelector).should('have.class', 'valid');
```

W WebdriverIO odczyt klasy realizowany jest metodą **getAttribute('class')**, a dopasowania dokonuje biblioteka asercji chai. Sama asercja chai nie ponawia jednak odczytu elementu, dlatego gdy zmiana klasy następuje z opóźnieniem, konieczne jest jawne oczekiwanie z użyciem **browser.waitFor()** (Listing 27).

Listing 27. Jawne oczekiwanie na zmianę klasy walidacji w WebdriverIO.

```
await browser.waitUntil(
  async () => {
    const cls = await emailValidIcon.getAttribute('class');
    return /\bvalid\b/.test(cls);
  },
  { timeout: 5000 }
);
```

Czwartym aspektem jest odczyt natywnych atrybutów walidacji HTML5 (**type**, **required**). Frameworki różnią się normalizacją wartości atrybutów logicznych. W Playwright, przedstawiony na Listingu 28, atrybut logiczny **required** reprezentowany jest jako pusty łańcuch znaków.

Listing 28. Weryfikacja natywnych atrybutów walidacji HTML5 w Playwright.

```
await expect(emailInput).toHaveAttribute('type', 'email');
await expect(emailInput).toHaveAttribute('required', '');
```

W Cypress weryfikacja atrybutu **required** sprowadza się do sprawdzenia jego obecności, bez odwołania do wartości (Listing 29).

Listing 29. Weryfikacja natywnych atrybutów walidacji HTML5 w Cypress.

```
cy.get(emailInputSelector).should('have.attr', 'type', 'email').and('have.attr', 'required');
```

W badanej konfiguracji WebdriverIO metoda **getAttribute('required')** zwracała natomiast wartość **'true'**, co wymagało odmiennego sformułowania asercji (Listing 30).

Listing 30. Weryfikacja natywnych atrybutów walidacji HTML5 w WebdriverIO.

```
expect(await emailInput.getAttribute('type')).to.equal('email');
expect(await emailInput.getAttribute('required')).to.equal('true');
```

Piątym aspektem jest przesyłanie plików (ang. *file upload*) realizowane poprzez ukryte pole **input[type="file"]**. W Playwright plik wskazywany jest metodą **setInputFiles()**, która przyjmuje ścieżkę do pliku w systemie plików, pokazaną na Listingu 31. Metoda działa również na ukrytym polu, bez dodatkowych obejść.

Listing 31. Przesłanie pojedynczego pliku w Playwright.

```
await fileInput.setInputFiles(testFilePath);
```

W Cypress wykorzystano metodę **selectFile()**, której przekazuje się zawartość pliku utworzoną w pamięci za pomocą **Cypress.Buffer**. Ponieważ pole jest ukryte, konieczne jest wymuszenie akcji opcją **{ force: true }** (Listing 32).

Listing 32. Przesłanie pojedynczego pliku w Cypress.

```
cy.get(fileInputSelector).selectFile({
  contents: Cypress.Buffer.from('This is a test file content.'),
  fileName: 'test-file.txt',
  mimeType: 'text/plain',
}, { force: true });
```

W WebdriverIO przesłanie pliku wymaga dwóch kroków: metoda **`browser.uploadFile()`** przesyła plik do węzła wykonującego test i zwraca jego zdalną ścieżkę, którą następnie ustawia się w polu metodą **`setValue()`** (Listing 33).

Listing 33. Przesłanie pojedynczego pliku w WebdriverIO.

```
const remoteFilePath = await browser.uploadFile(testFilePath);
await fileInput.setValue(remoteFilePath);
```

Największe różnice ujawniają się przy przesyłaniu wielu plików jednocześnie. Playwright (**`setInputFiles([...])`**) oraz Cypress (**`selectFile([...])`**) obsługują tablicę plików natywnie. W badanej konfiguracji WebdriverIO metoda **`setValue()`** nie obsługuje natomiast wielu plików, dlatego zastosowano obejście polegające na wstrzyknięciu obiektu **`DataTransfer`** za pośrednictwem **`browser.execute()`**, które przedstawia Listing 34.

Listing 34. Obejście przesyłania wielu plików przez wstrzyknięcie `DataTransfer` w WebdriverIO.

```
await browser.execute(() => {
  const dt = new DataTransfer();
  dt.items.add(new File(['This is a test file content.'], 'test-file.txt', {
type: 'text/plain' }));
  dt.items.add(new File(['Second test file content.'], 'second-file.txt', { type:
'text/plain' }));
  const fileInput = document.querySelector('#fileInput') as HTMLInputElement;
  Object.defineProperty(fileInput, 'files', { value: dt.files, configurable: true
});
  fileInput.dispatchEvent(new Event('change', { bubbles: true }));
});
```

Uzupełnieniem analizowanych scenariuszy były testy formularza wielokrokowego, w których wybór wartości z list rozwijanych realizowano metodą **`selectOption()`** w Playwright, **`select()`** w Cypress oraz **`selectByAttribute('value', ...)`** w WebdriverIO. Nawigację pomiędzy kolejnymi krokami oraz weryfikację komunikatów o brakujących polach zaimplementowano analogicznie we wszystkich frameworkach, wykorzystując opisane wcześniej mechanizmy synchronizacji.

Zestawienie różnic w obsłudze formularzy i walidacji w czwartej kategorii testowej przedstawiono w Tabeli 10.

Tabela 10. Obsługa formularzy i walidacji w czwartej kategorii testowej.
Źródło: opracowanie własne.

Aspekt	Playwright	Cypress	WebdriverIO
Wprowadzanie danych	<code>fill()</code>	<code>type()</code> (symulacja klawiszy)	<code>setValue()</code>
Wyzwalanie zdarzenia blur	<code>blur()</code>	<code>blur()</code>	<code>\$('.body').click()</code> (obejście)
Asercja zmiany klasy	<code>toHaveClass()</code> (auto-wait)	<code>should('have.class')</code> (auto-retry)	<code>getAttribute() + browser.waitUntil()</code>
Atrybut logiczny required	pusty łańcuch ('')	obecność atrybutu	wartość 'true'

Aspekt	Playwright	Cypress	WebdriverIO
Wybór z listy rozwijanej	<code>selectOption()</code>	<code>select()</code>	<code>selectByAttribute()</code>
Przesłanie pojedynczego pliku	<code>setInputFiles</code> (ścieżka)	<code>selectFile()</code> + { force: true }	<code>uploadFile()</code> + <code>setValue()</code>
Przesłanie wielu plików	<code>setInputFiles</code> ([...])	<code>selectFile</code> ([...])	obejście <code>DataTransfer</code> + <code>browser.execute()</code>

Praktyczne znaczenie tych różnic widać przede wszystkim przy asercjach na dynamicznie aktualizowanym interfejsie. Wbudowane mechanizmy oczekiwania w Playwright i Cypress upraszczają je, usuwając potrzebę jawnego oczekiwania. W WebdriverIO realizacja tego samego scenariusza wymaga większej liczby instrukcji ze względu na brak dedykowanej metody `element.blur()`, dwuetapowy proces przesyłania pliku oraz konieczność stosowania obejścia przy wielu plikach. Różnica ta znajduje odzwierciedlenie w wartości metryki LOC omówionej w rozdziale 6.

5.5. Pobieranie plików

Piąta kategoria obejmuje pobieranie plików tekstowych oraz binarnych. Wykorzystano stronę z przyciskiem inicjującym pobranie, podczas którego zmienia się stan interfejsu, czyli komunikat statusu, pasek postępu, blokada przycisku. Kategoria ta wyraźnie różnicuje frameworki pod względem poziomu wsparcia i wygody weryfikacji pobrania, przekładając się na liczbę przypadków testowych zrealizowanych w badaniu.

W Playwright pobranie przechwytywane jest zdarzeniem `waitForEvent('download')` [55], po czym plik można zapisać metodą `saveAs()`, a następnie odczytać jego treść i strukturę binarną z dysku. Pobranie i weryfikację treści pliku w Playwright prezentuje Listing 35.

Listing 35. Pobranie i weryfikacja treści pliku w Playwright.

```
const downloadPromise = page.waitForEvent('download', { timeout: 30000 });
await downloadBtn.click();
const download: Download = await downloadPromise;

await download.saveAs(tempFilePath);
const fileContent = fs.readFileSync(tempFilePath, 'utf-8');
expect(fileContent).toContain('Business Performance Report');
```

Cypress nie udostępnia natywnego obiektu pobrania, dlatego w badaniu zastosowano weryfikację pośrednią: przechwycenie wywołania API metodą `cy.intercept()` i sprawdzenie nagłówek odpowiedzi. Alternatywnie `cy.request()` pozwala zweryfikować treść zwróconą przez API. Weryfikacja dotyczy zatem odpowiedzi serwera, a nie pobranego pliku (Listing 36).

Listing 36. Pośrednia weryfikacja pobrania przez przechwycenie API w Cypress.

```
cy.intercept('POST', '/api/practice/download/generate').as('downloadRequest');
cy.get('#download-report-btn').click();
cy.wait('@downloadRequest').then((interception) => {
  expect(interception.response?.headers['content-type']).to.include('text/plain');
});
```

W WebdriverIO weryfikację ograniczono do stanu interfejsu: testowane jest pojawienie się komunikatu *Download complete* oraz klasy **success**. W przyjętej implementacji nie analizowano pobranego pliku ani odpowiedzi sieciowej (Listing 37).

Listing 37. Weryfikacja stanu interfejsu po pobraniu w WebdriverIO.

```
await downloadBtn.click();
await browser.waitUntil(
  async () => (await statusMessage.getText()).includes('Download complete'),
  { timeout: 30000 }
);
const statusClass = await statusMessage.getAttribute('class');
expect(statusClass).to.include('success');
```

Playwright, dysponując bezpośrednim dostępem do pliku, pozwolił zrealizować najwięcej praktycznie uzasadnionych przypadków (38), w tym walidację treści i struktury binarnej. Cypress, opierając się na odpowiedzi API, objął ich mniej (34). W WebdriverIO, gdzie weryfikację ograniczono do stanu interfejsu, zrealizowano ich najmniej (21). Różnica wynika przede wszystkim z poziomu wsparcia oferowanego przez narzędzia w przyjętym podejściu testowym. Zestawienie przedstawiono w Tabeli 11.

Tabela 11. Weryfikacja pobierania plików w piątej kategorii testowej.
Źródło: opracowanie własne.

Cecha	Playwright	Cypress	WebdriverIO
Mechanizm weryfikacji	zdarzenie <i>download</i> (natywne)	przechwycenie API (cy.intercept)	stan interfejsu
Analiza pobranego pliku	tak (odczyt z dysku)	nie analizowano	nie analizowano
Walidacja treści w badaniu	pełna (tekst i binarna)	nagłówki i treść odpowiedzi API	stan interfejsu
Walidacja	wykonana	niewykonana	niewykonana
Liczba przypadków testowych	38	34	21

Pobieranie plików jest kategorią, w której różnice nie mają charakteru składniowego, tylko charakter związany z poziomem wsparcia i wygodą weryfikacji rezultatu. Zakres weryfikacji przełożył się bezpośrednio na liczbę przypadków testowych 38, 34 i 21.

5.6. Tabele i paginacja

Szósta kategoria obejmuje pracę z tabelą danych w trzech obszarach: odczycie i sortowaniu kolumn, paginacji oraz nieskończonym przewijaniu (ang. *infinite scroll*). Każdy z obszarów eksponuje inny aspekt różnic między frameworkami.

Jako pierwszy zbadano odczyt i sortowanie kolumny. Wspólnym elementem scenariuszy sortowania jest pobranie wartości całej kolumny i porównanie ich z posortowaną kopią. Różnica dotyczy nie samej asercji, lecz sposobu pozyskania danych. Playwright udostępnia przedstawioną na Listingu 38 metodę zbiorczą **allTextContents()**, która jednym wywołaniem zwraca tablicę tekstów wszystkich dopasowanych elementów.

Listing 38. Zbiorczy odczyt kolumny w Playwright.

```
const page1Ids = await tableBody.locator('tr td:first-child').allTextContents();
```

Cypress nie oferuje zbiorczej metody odpowiadającej `allTextContents()`. Odczyt realizuje się wewnątrz łańcucha `.then()`, iterując po elementach za pomocą jQuery (Listing 39).

Listing 39. Odczyt kolumny w Cypress.

```
cy.get('#dataGridBody').find('tr').then(($rows) => {  
  const ids: number[] = [];  
  $rows.each((index, row) => {  
    const idText = Cypress.$(row).find('td').first().text();  
    if (idText) ids.push(parseInt(idText, 10));  
  });  
});
```

W WebdriverIO odczyt zrealizowano przez materializację kolekcji wierszy metodą `$$()`, a następnie pobranie komórki i jej tekstu w pętli `for...of`. W przedstawionej implementacji liczba jawnie zapisanych operacji jest tu największa (Listing 40).

Listing 40. Odczyt kolumny w WebdriverIO.

```
const rows = await tableBody.$$('tr');  
const ids: number[] = [];  
for (const row of rows) {  
  const cells = await row.$$('td');  
  const id = await cells[0].getText();  
  if (id) ids.push(parseInt(id, 10));  
}
```

Dalsza logika weryfikacji jest już identyczna we wszystkich frameworkach – odczytaną tablicę porównuje się z jej posortowaną kopią.

Drugim obszarem jest paginacja. Po kliknięciu przycisku następnej strony aplikacja pobiera dane asynchronicznie, dlatego przed asercją należy poczekać na faktyczną zmianę zawartości. Trzy frameworki obsługują tę funkcjonalność odmiennie. Playwright stosuje warunek na zmianę treści informacji o stronie, korzystając z ponawianej asercji (Listing 41).

Listing 41. Synchronizacja zmiany strony w Playwright.

```
const page1Text = await pageInfo.textContent();  
await nextButton.click();  
await expect(pageInfo).not.toHaveText(page1Text || '');
```

Cypress synchronizuje zmianę strony, czekając na zakończenie przechwyconego żądania API. Definiuje alias metodą `cy.intercept()`, a po kliknięciu oczekuje na jego rozwiązanie metodą `cy.wait('@alias')`. W prostszych przypadkach w badanej implementacji zastosowano także stałe opóźnienia `cy.wait(800)` (Listing 42).

Listing 42. Synchronizacja zmiany strony aliasem żądania w Cypress.

```
cy.intercept('GET', /\//api/v2/data/random/employees/).as('pageApiCall');  
cy.get('#nextPage').click();  
cy.wait('@pageApiCall');
```

W WebdriverIO w badanej implementacji zmianę strony zsynchronizowano stałą pauzą **browser.pause()** wstawianą po kliknięciu. Framework udostępnia również mechanizmy warunkowe (np. **waitUntil()**), jednak w zastosowanym rozwiązaniu posłużono się pauzą (Listing 43).

Listing 43. Synchronizacja zmiany strony w WebdriverIO.

```
await nextButton.click();
await browser.pause(800);
const finalPageText = await pageInfo.getText();
```

Trzecim obszarem jest nieskończone przewijanie. Pojawianie się kolejnych elementów wyzwała się przewinięciem strony do końca, po czym należy odczekać na zniknięcie wskaźnika ładowania i zweryfikować zwiększoną liczbę elementów. Jak zilustrowano na Listingu 44, Playwright przewija stronę przy użyciu **page.evaluate()**, a następnie korzysta z automatycznie oczekującej asercji **toHaveCount()**. Lokatory są dynamiczne, więc nie wymagają ponownego pobrania.

Listing 44. Nieskończone przewijanie w Playwright.

```
await page.evaluate(() => window.scrollTo(0, document.body.scrollHeight));
await expect(loadingIndicator).not.toBeVisible({ timeout: 5000 });
await expect(items).toHaveCount(20);
```

Cypress przewija okno przeglądarki przez **cy.window()** lub **cy.scrollTo('bottom')** i weryfikuje liczbę elementów ponawianą asercją **should('have.length', ...)** (Listing 45).

Listing 45. Nieskończone przewijanie w Cypress.

```
cy.window().then((win) => {
  win.scrollTo(0, win.document.body.scrollHeight);
});
cy.get('#loading', { timeout: 5000 }).should('not.be.visible');
cy.get('#items-container .item').should('have.length', 20);
```

WebdriverIO przewija stronę skryptem **browser.execute()** i oczekuje na zniknięcie wskaźnika metodą **browser.waitUntil()**. Kluczowa różnica polega na tym, że po aktualizacji stanu DOM, kolekcję elementów trzeba pobrać ponownie metodą **\$\$()**, ponieważ wcześniej zmaterializowana tablica nie odzwierciedla nowych elementów (Listing 46). Jest to ten sam mechanizm dezaktualizacji referencji, który opisano w podrozdziale 5.1.

Listing 46. Nieskończone przewijanie w WebdriverIO.

```
await browser.execute(() => {
  window.scrollTo(0, document.body.scrollHeight);
});
await browser.waitUntil(async () => !(await loadingIndicator.isDisplayed()), {
  timeout: 5000 });
items = await $$('#items-container .item');
expect(items.length).toEqual(20);
```

Zestawienie sposobów obsługi tabel, paginacji i przewijania przedstawiono w Tabeli 12.

Tabela 12. Obsługa tabel, paginacji i nieskończonego przewijania w szóstej kategorii testowej. Źródło: opracowanie własne.

Cecha	Playwright	Cypress	WebdriverIO
Odczyt kolumny	<code>allTextContents()</code>	<code>.then((\$rows) => each)</code>	<code>\$\$() + for...of</code>
Synchronizacja zmiany strony	warunek <code>not.toHaveText()</code>	alias <code>intercept</code> <code>cy.wait('@...')</code>	<code>browser.pause()</code> (w badaniu)
Nieskończone przewijanie	<code>page.evaluate()</code>	<code>cy.window()</code> <code>.scrollTo()</code>	<code>browser.execute()</code>
Aktualizacja kolekcji po przewinięciu	lokatory dynamiczne	<i>auto-retry</i> długości	ponowne pobranie <code>\$\$()</code>

Kategoria ta ujawnia dwa źródła kruchości. Pierwszym są stałe pauzy, które w badanej implementacji wystąpiły w WebdriverIO (**`browser.pause()`**) oraz w prostszych testach paginacji Cypress (**`cy.wait()`**). Dobranie wartości pauzy jest z natury kruche. Zależy od wydajności środowiska, przez co test dostrojony na jednej maszynie może okazać się niestabilny na innej. Playwright oraz Cypress w wariancie z aliasem unikają tego problemu, oczekując na warunek lub zakończenie żądania zamiast na upływ czasu. Drugim źródłem jest konieczność ręcznego ponownego pobrania kolekcji elementów po zmianie DOM w WebdriverIO, podczas gdy dynamiczne lokatory Playwright i mechanizm ponawiania Cypress aktualizują się samoczynnie. Wpływ tych różnic na liczbę instrukcji, czas wykonania oraz stabilność, mierzone metrykami objętości kodu (LOC) i *flakiness*, przeanalizowano w rozdziale 6.

5.7. Przechwytywanie sieci i mockowanie API

Siądma kategoria obejmuje scenariusze przechwytywania żądań sieciowych (ang. *network interception*) oraz podmiany odpowiedzi serwera (ang. *API mocking*). Testy weryfikują zachowanie aplikacji po zwróceniu własnych danych (status 200), kodów błędów (m.in. 400, 401, 429, 500), wymuszonym opóźnieniu odpowiedzi oraz w trybie braku połączenia. Kategoria ta najsilniej różnicuje analizowane narzędzia, ponieważ ujawnia zarówno odmienny styl API, jak i ograniczenia przechwytywania ruchu sieciowego w przyjętej konfiguracji WebdriverIO.

Playwright realizuje przechwytywanie imperatywnie. Metoda **`page.route()`** [53], [54] rejestruje przechwytywacz dla określonego wzorca adresu, natomiast właściwa odpowiedź jest konstruowana w asynchronicznej funkcji zwrotnej za pomocą **`route.fulfill()`**. W przedstawionej implementacji ciało odpowiedzi zostało zserializowane za pomocą metody **`JSON.stringify()`**. Listing 47 zawiera implementację mockowania odpowiedzi API ze statusem 200.

Listing 47. Mockowanie odpowiedzi API ze statusem 200 w Playwright.

```
await page.route('**/api/v1/data/weather**', async (route) => {
  await route.fulfill({
    status: 200,
    contentType: 'application/json',
    body: JSON.stringify(mockWeatherData), }); });
```

Cypress stosuje podejście deklaratywne: pojedyncze wywołanie `cy.intercept()` [58] przyjmuje wzorzec adresu oraz obiekt opisujący odpowiedź. Ciało odpowiedzi podaje się bezpośrednio jako obiekt, bez ręcznej serializacji, a alias nadany metodą `.as()` umożliwia późniejszą weryfikację żądania konstrukcją `cy.wait('@weatherApi')` (Listing 48).

Listing 48. Mockowanie odpowiedzi API ze statusem 200 w Cypress.

```
cy.intercept('GET', '**/api/v1/data/weather**', {
  statusCode: 200,
  body: mockWeatherData,
}).as('weatherApi');
```

W przyjętej konfiguracji WebdriverIO nie wykorzystano mockowania żądań sieciowych, a cała kategoria została wyłączona z pomiarów dyrektywą `exclude` w pliku `wdio.conf.ts`. Decyzja ta wynika z opisanej w podrozdziale 3.3.3 niedostępności praktycznie użytecznego mechanizmu `browser.mock()` [43] w WebdriverIO w wersji 9, będącej następstwem usunięcia pakietu `devtools`. Tryb WebDriver Classic, wymuszony w konfiguracji flagą `wdio:enforceWebDriverClassic: true`, nie udostępnia natywnego API do przechwytywania ruchu sieciowego. Alternatywą jest protokół WebDriver BiDi, którego potencjał w tym obszarze omówiono jako kierunek dalszych badań w podrozdziale 7.5.

Zestawienie mechanizmów przechwytywania i mockowania przedstawiono w Tabeli 13.

Tabela 13. Mechanizmy przechwytywania żądań i mockowania API w siódmej kategorii testowej. Źródło: opracowanie własne.

Cecha	Playwright	Cypress	WebdriverIO
Metoda przechwytywania	<code>page.route()</code> + <code>route.fulfill()</code>	<code>cy.intercept()</code>	nie wykorzystano (WebDriver Classic)
Styl API	imperatywny (asynchroniczna funkcja zwrotna)	deklaratywny	–
Format ciała odpowiedzi	wymaga <code>JSON.stringify()</code>	obiekt przyjmowany bezpośrednio	–
Weryfikacja żądania	nasłuch w funkcji zwrotnej	alias <code>.as()</code> + <code>cy.wait()</code>	–
Symulacja opóźnienia	opóźnione wywołanie <code>route.fulfill()</code>	właściwość <code>delay</code>	–

W tej kategorii widoczne są dwie zasadnicze różnice. Pierwsza dotyczy stylu API: w Cypress odpowiedź jest definiowana deklaratywnie poprzez przekazanie obiektu opisującego jej właściwości, natomiast w Playwright odpowiedź konstruowana jest w asynchronicznej funkcji zwrotnej, co wymaga również jawnej serializacji jej treści w przedstawionej implementacji. Widać ją również przy symulacji opóźnienia: w analizowanych testach realizowano je w Cypress właściwością `delay`, a w Playwright opóźnionym wywołaniem `route.fulfill()`. Drugą różnicą jest wbudowane aliasowanie żądań w Cypress, które upraszcza weryfikację, oraz sprawdzenie parametrów wysyłanych żądań. Wpływ tych różnic na objętość kodu, mierzoną metryką LOC, przeanalizowano w rozdziale 6.

5.8. Autoryzacja i zarządzanie sesją

Ósma kategoria obejmuje uwierzytelnianie użytkownika oraz zarządzanie sesją w aplikacji GAD. Scenariusze oparto na czterech zagadnieniach:

- weryfikacji tokena JWT (ang. *JSON Web Token*) zapisanego w ciasteczkach (ang. *cookies*) po zalogowaniu,
- utrzymaniu sesji podczas nawigacji po stronach chronionych,
- usunięciu tokena po wylogowaniu,
- logowaniu przez natywne okno dialogowe ***prompt()*** przeglądarki.

We wszystkich scenariuszach wykorzystano tego samego użytkownika testowego i identyczną procedurę logowania, co skutkuje bezpośrednim porównaniem mechanizmów obsługi sesji.

Na tym tle zarysowują się cztery różnice architektoniczne: model dostępu do ciasteczek, sposób weryfikacji adresu URL po nawigacji, dostępność wbudowanego mechanizmu buforowania sesji (ang. *session caching*) oraz obsługa natywnych dialogów przeglądarki.

Jak przedstawiono na Listingu 49 Playwright pobiera wszystkie ciasteczka pojedynczym wywołaniem asynchronicznym metody ***context.cookies()***, zwracającym tablicę, w której właściwy wpis odnajduje się przez filtrowanie po nazwie.

Listing 49. Pobranie i weryfikacja ciasteczka z tokenem w Playwright.

```
const cookies = await context.cookies();
const tokenCookie = cookies.find((cookie) => cookie.name === 'token');
expect(tokenCookie).toBeDefined();
```

Cypress udostępnia dedykowane, łańcuchowe polecenie ***cy.getCookie()*** operujące bezpośrednio na pojedynczym ciasteczku, a dla całego zestawu jest dostępna metoda ***cy.getAllCookies()***. Asercję wykonuje się wewnątrz metody ***.then()*** (Listing 50).

Listing 50. Pobranie i weryfikacja ciasteczka z tokenem w Cypress.

```
cy.getCookie('token').then((cookie) => {
  expect(cookie).to.not.be.null;
});
```

WebdriverIO, analogicznie do Playwright, zwraca pełną tablicę za pomocą metody ***browser.getCookies()*** i wymaga jawnego ***await*** oraz ręcznego filtrowania (Listing 51). Do asercji wykorzystano bibliotekę chai.

Listing 51. Pobranie i weryfikacja ciasteczka z tokenem w WebdriverIO.

```
const cookies = await browser.getCookies();
const tokenCookie = cookies.find((cookie) => cookie.name === 'token');
expect(tokenCookie).to.not.be.undefined;
```

Różnica polega na poziomie abstrakcji: Cypress oferuje semantyczne polecenia operujące na pojedynczym ciasteczku (***cy.getCookie()***), ***cy.getAllCookies()***, ***cy.clearCookie()***), podczas gdy Playwright i WebdriverIO udostępniają jedną metodę zwracającą całą kolekcję, którą filtruje się samodzielnie.

Weryfikacja adresu URL po przejściu na stronę chronioną odsłania znaną już różnicę w synchronizacji. Playwright stosuje automatyczne oczekiwanie w asercji **`expect(page).toHaveURL()`**, Cypress mechanizm automatycznego ponawiania w **`cy.url().should('include', ...)`**, natomiast WebdriverIO wymaga jawnego odpytywania warunku przez **`browser.waitUntil()`** z ustalonym limitem czasu. Frameworki różnią się także sposobem izolacji stanu między testami. W Playwright izolację zapewnia osobny kontekst przeglądarki tworzony dla każdego testu. Cypress domyślnie czyści ciasteczka oraz pamięć lokalną i sesyjną, a także resetuje stan strony przed każdym testem używając mechanizmu izolacji testu (ang. *test isolation*) [59], dzięki czemu nie wymaga jawnego czyszczenia. W badanej implementacji WebdriverIO, stan czyszczono jawnie, wywołaniem **`browser.deleteCookies()`** w bloku **`beforeEach`**.

Kolejną różnicą jest dostępność wbudowanego mechanizmu buforowania sesji, eliminującego ponowne uwierzytelnianie w kolejnych testach. Playwright nie udostępnia takiego mechanizmu. Zbliżony efekt można uzyskać, zapisując stan sesji metodą **`context.storageState()`** i wykorzystując go ponownie przy tworzeniu nowego kontekstu przeglądarki (Listing 52). Samo wywołanie tej metody nie przywraca jednak sesji automatycznie [60].

Listing 52. Odczyt stanu sesji metodą `storageState()` w Playwright.

```
const state = await context.storageState();
const tokenCookie = state.cookies.find((c) => c.name === 'token');
expect(tokenCookie).toBeDefined();
```

Cypress udostępnia przedstawiony na Listingu 53 natywny mechanizm **`cy.session()`** [61], który jednorazowo wykonuje logowanie, buforuje stan sesji i automatycznie przywraca go w kolejnych testach.

Listing 53. Buforowanie sesji prymitywem `cy.session()` w Cypress.

```
cy.session('user-session', () => {
  cy.visit('/login');
  cy.get('input[placeholder="Enter User Email"]').type(testUser.email);
  cy.get('input[placeholder="Enter Password"]').type(testUser.password);
  cy.get('#loginButton').click();
  cy.url().should('include', '/welcome');
});
```

WebdriverIO nie udostępnia wbudowanego mechanizmu odpowiadającego funkcjonalnością **`cy.session()`**. W badanej implementacji utrzymanie sesji weryfikowano przez ponowne wywołanie funkcji **`performLogin()`** oraz porównanie wartości tokena przed i po nawigacji, bez warstwy buforującej.

Scenariusz zawierający logowanie modalne najsilniej różnicuje frameworki, ponieważ wymaga obsługi natywnego okna **`prompt()`** przeglądarki, które w aplikacji GAD pojawia się synchronicznie podczas ładowania strony. Playwright realizuje to deklaratywnie, rejestrując nasłuch zdarzenia **`dialog`** przed nawigacją i przekazując dane uwierzytelniające metodą **`dialog.accept()`** [62] (Listing 54).

Listing 54. Obsługa natywnego dialogu `prompt()` w Playwright.

```
page.on('dialog', async (dialog) => {
  if (dialog.type() === 'prompt') {
    await dialog.accept(validCredentials);
  }
});
await page.goto('/practice/login-modal.html');
```

Cypress nie steruje rzeczywistym oknem dialogowym – zamiast tego podmienia funkcję `window.prompt` zaślepką (ang. *stub*) w callbacku `onBeforeLoad`, wykonywanym przed załadowaniem strony. Zwracana wartość symuluje dane wpisane przez użytkownika (Listing 55).

Listing 55. Podmiana funkcji `prompt` zaślepką w Cypress.

```
cy.visit('/practice/login-modal.html', {
  onBeforeLoad(win) {
    cy.stub(win, 'prompt').returns(validCredentials);
  },
});
```

W WebdriverIO obsługa tego scenariusza wymagała mostu do instancji *Puppeteer* uzyskiwanej metodą `browser.getPuppeteer()`. Wynika to z ograniczenia protokołu WebDriver Classic, w którym zgodnie ze specyfikacją W3C [33] okna dialogowe pojawiające się w trakcie nawigacji są automatycznie odrzucane. Uniemożliwia to przekazanie do nich danych natywnym API alertów takich jak: `browser.acceptAlert()`, `browser.sendAlertText()`. Ponieważ dialog w omawianym scenariuszu uruchamia się podczas ładowania strony, konieczne było zarejestrowanie nasłuchu zdarzenia `dialog` przed nawigacją na stronie kontrolowanej przez protokół Chrome DevTools (CDP), uzupełnione jawnym opóźnieniem `browser.pause()`, co widać na Listingu 56.

Listing 56. Obsługa natywnego dialogu przez most CDP/Puppeteer w WebdriverIO.

```
const puppeteer = await browser.getPuppeteer();
const page = (await puppeteer.pages())[0];
page.on('dialog', async (dialog: any) => {
  await dialog.accept(credentials);
});
await browser.url('/practice/login-modal.html');
await browser.pause(1000);
```

Spośród trzech rozwiązań Playwright zapewnia najbardziej zwięzłą obsługę, sterując rzeczywistym dialogiem natywnie. Cypress nie wchodzi w interakcję z prawdziwym oknem przeglądarki, lecz zastępuje wywołanie `prompt` zaślepką na poziomie obiektu `window`. Podejście WebdriverIO opiera się na moście CDP do instancji Puppeteer, co ogranicza je do przeglądarek opartych na rodzinie przeglądarek Chromium. Zestawienie omówionych mechanizmów przedstawiono w Tabeli 14.

Tabela 14. Obsługa autoryzacji i sesji w analizowanych frameworkach.
Źródło: opracowanie własne.

Aspekt	Playwright	Cypress	WebdriverIO
Pobranie ciasteczek	<code>context.cookies()</code> (tablica)	<code>cy.getCookie()</code> (dedykowane)	<code>browser.getCookies()</code> (tablica)

Aspekt	Playwright	Cypress	WebdriverIO
Weryfikacja URL po nawigacji	<code>toHaveURL()</code> (automatyczne oczekiwanie)	<code>cy.url().should()</code> (automatyczne ponawianie)	<code>browser.waitUntil()</code> (jawne oczekiwanie)
Izolacja sesji między testami	osobny kontekst przeglądarki (domyślnie)	Automatyczna	<code>Browser.deleteCookies()</code> (w badaniu)
Buforowanie sesji	<code>Context.storageState()</code> (zapis stanu)	<code>cy.session()</code> (natywny prymityw)	brak wbudowanego mechanizmu
Natywny dialog <code>prompt()</code>	<code>page.on('dialog')</code> (sterowanie natywne)	<code>cy.stub(win, 'prompt')</code> (zaślepka funkcji)	<code>Browser.getPuppeteer()</code> + CDP (most do Puppeteer)

Najmniejsza objętość kodu w tej kategorii przypada implementacji Cypress – to efekt dedykowanych poleceń do obsługi ciasteczek i sesji oraz braku jawnej składni ***async/await***. Wpływ tych różnic na czas wykonania i stabilność omówiono w rozdziale 6.

5.9. Iframes i zagnieżdżone konteksty

Dziewiąta kategoria obejmuje interakcje z elementami umieszczonymi wewnątrz ramek (ang. *iframes*). Scenariusze oparto na czterech stronach testowych:

- prostej ramce poziomu pierwszego,
- formularzu w zagnieżdżonej ramce,
- naprzemiennym przełączaniu między ramką a stroną główną,
- ramce bez przewidywalnego identyfikatora.

Pierwszy scenariusz dotyczy dostępu do elementu znajdującego się w ramce poziomu pierwszego. Pokazuje on odmienne modele wejścia w kontekst ramki.

W Playwright dostęp realizuje metoda ***frameLocator()*** [63], która zwraca leniwy lokator ograniczony do kontekstu ramki, bez stanu pośredniego i bez konieczności jawnego przełączania. Wbudowane mechanizmy oczekiwania zwalniają z jawnej synchronizacji. Dostęp do elementu w ramce poziomu pierwszego dokumentuje Listing 57.

Listing 57. Dostęp do elementu w ramce poziomu pierwszego w Playwright.

```
const iframe = page.frameLocator('iframe');
const clocksContainer = iframe.locator('#clocks');
await expect(clocksContainer).toBeVisible();
```

Cypress nie udostępnia dedykowanego, wbudowanego polecenia wejścia w kontekst ramki. Oficjalnie zalecanym podejściem jest odczyt właściwości ***contentDocument.body*** [64] elementu ramki, zastosowany w badaniu, lub wykorzystanie wtyczki `cypress-iframe`. Pierwsze z tych podejść polega na pobraniu właściwości ***contentDocument.body*** elementu ramki i opakowaniu jej za pomocą metody ***cy.wrap()***, co umożliwia stosowanie standardowych poleceń Cypress. Pokazany wzorec działa dla ramek z tej samej domeny (ang. *same-origin*). W badanej implementacji ramkę selekcionowano po nazwie znacznika (***cy.get('iframe')***), co jest wystarczające przy pojedynczej

ramce na stronie. W przypadku większej liczby ramek zalecany jest selektor atrybutowy, na przykład `cy.get('iframe[data-cy="..."]')`, zapewniający jednoznaczne wskazanie elementu (Listing 58).

Listing 58. Dostęp do elementu w ramce poziomu pierwszego w Cypress.

```
cy.get('iframe')
  .its('0.contentDocument.body')
  .should('not.be.empty')
  .then(cy.wrap);
```

W WebdriverIO dostęp wymaga jawnego przełączenia aktywnego kontekstu metodą `browser.switchFrame()`. Konieczne jest uprzednie pobranie referencji do elementu ramki oraz oczekiwanie na jego istnienie. Dopiero po przełączeniu kolejne wywołania `$()` odnoszą się do wnętrza ramki (Listing 59).

Listing 59. Dostęp do elementu w ramce poziomu pierwszego w WebdriverIO.

```
const iframe = await $('iframe');
await iframe.waitForExist({ timeout: 5000 });
await browser.switchFrame(iframe);
```

Kolejny scenariusz obejmuje ramkę zagnieżdżoną na drugim poziomie, w której wypełniany jest formularz. Różnice między narzędziami są tu pokazane wyraźniej, ponieważ dostęp do elementu wymaga przejścia przez dwa poziomy zagnieżdżenia.

Jak pokazano na Listingu 60, w Playwright kolejne poziomy obsługuje się przez łańcuchowe wywołania `frameLocator()`, niezależnie od głębokości zagnieżdżenia i bez stanu pośredniego.

Listing 60. Dostęp do ramki zagnieżdżonej (poziom drugi) w Playwright.

```
page.frameLocator('iframe').frameLocator('#weather-iframe');
```

W Cypress każdy poziom zagnieżdżenia wymaga osobnego pobrania `contentDocument.body` oraz dodatkowego wywołania `.then()`, przez co długość i złożoność łańcucha poleceń rosną wraz z głębokością (Listing 61).

Listing 61. Dostęp do ramki zagnieżdżonej (poziom drugi) w Cypress.

```
cy.get('iframe')
  .its('0.contentDocument.body')
  .then((outerBody) => {
    return cy.wrap(outerBody)
      .find('#weather-iframe')
      .its('0.contentDocument.body')
      .then(cy.wrap);
  });
```

Jak przedstawiono na Listingu 62, w WebdriverIO każdy poziom wymaga odrębnego przełączenia kontekstu, dlatego dostęp do ramki wewnętrznej poprzedza przełączenie do ramki zewnętrznej.

Listing 62. Dostęp do ramki zagnieżdżonej (poziom drugi) w WebdriverIO.

```
const outerIframe = await $('iframe');
await outerIframe.waitForExist({ timeout: 5000 });
await browser.switchFrame(outerIframe);
```

```
const innerIframe = await $('#weather-iframe');
await innerIframe.waitForExist({ timeout: 5000 });
await browser.switchFrame(innerIframe);
```

Trzeci scenariusz polega na naprzemiennym przełączaniu między ramką a stroną główną. W Playwright oraz Cypress elementy strony nadrzędnej pozostają dostępne bezpośrednio – odpowiednio przez `page.locator()` oraz `cy.get()` niezależnie od wcześniejszego dostępu do ramki, bez przełączania i bez czyszczenia stanu.

W WebdriverIO powrót do strony głównej wymaga jawnego wywołania `browser.switchFrame(null)`. W badanych scenariuszach kontekst resetowano dodatkowo w bloku `afterEach`. Brak takiego resetu skutkowałby wykonywaniem kolejnych asercji w nieaktualnym kontekście (Listing 63).

Listing 63. Powrót do strony głównej w WebdriverIO.

```
await browser.switchFrame(null);
const mainTitle = await $('h2');
await mainTitle.waitForDisplayed({ timeout: 5000 });
```

Ostatni scenariusz dotyczy ramki pozbawionej przewidywalnego identyfikatora, co wymusza selekcję elementu ramki na podstawie atrybutu lub pozycji.

W Playwright dowolny selektor CSS, w tym `[data-testid="..."]`, można przekazać wprost do `frameLocator()`, również dla ramek zagnieżdżonych (Listing 64).

Listing 64. Selekcja zagnieżdżonej ramki po atrybucie data-testid w Playwright.

```
page.frameLocator('[data-testid="dti-simple-iframe"]').frameLocator('#inner-iframe');
```

W Cypress ramkę zewnętrzną wskazuje się standardowym selektorem `cy.get()`, na przykład po atrybucie `data-testid`, a w razie braku jakiegokolwiek identyfikatora, przez selekcję pozycyjną metodą `.first()`. Dalszy dostęp do ramki wewnętrznej przebiega według omówionego wcześniej wzorca opartego na `contentDocument.body` (Listing 65).

Listing 65. Selekcja ramki bez identyfikatora przez pozycję w Cypress.

```
cy.get('iframe')
  .first()
  .its('0.contentDocument.body')
  .then(outerBody => {
    return cy.wrap(outerBody)
      .find('iframe')
      .first()
      .its('0.contentDocument.body')
      .then(cy.wrap);
  });
```

W WebdriverIO metoda `browser.switchFrame()` [65] przyjmuje referencję do elementu, adres URL lub wartość `null`, lecz nie przyjmuje bezpośrednio selektora atrybutu `data-testid`. W praktyce wymusza to dwuetapowy wzorec: najpierw pobranie elementu ramki metodą `$()`, a następnie przełączenie kontekstu na pobraną referencję (Listing 66).

Listing 66. Selekcja zagnieżdżonej ramki bez identyfikatora w WebdriverIO.

```
const outerIframe = await $('iframe');
await outerIframe.waitForExist({ timeout: 5000 });
await browser.switchFrame(outerIframe);

const innerIframe = await $('#inner-iframe');
await innerIframe.waitForExist({ timeout: 5000 });
await browser.switchFrame(innerIframe);
```

W Tabeli 15 zestawiono różnice w obsłudze ramek między analizowanymi narzędziami. Wpis dotyczący czyszczenia kontekstu odnosi się do przyjętego sposobu organizacji scenariuszy testowych, nie zaś do bezwzględnego wymogu frameworka.

Tabela 15. Porównanie obsługi ramek w trzech frameworkach.

Źródło: opracowanie własne.

Aspekt	Playwright	Cypress	WebdriverIO
Mechanizm dostępu	<code>frameLocator()</code>	<code>cy.get(...).its('0.contentDocument.body')</code>	<code>browser.switchFrame()</code>
Natywne wsparcie ramek	tak	nie (<code>contentDocument</code> lub wtyczka)	tak
Model kontekstu	leniwy lokator, bez stanu pośredniego	trawersacja DOM, bez stanu pośredniego	jawne przełączanie aktywnego kontekstu
Zagnieżdżanie	łańcuch <code>frameLocator()</code>	łańcuch <code>.its() + .then()</code>	sekwencja <code>switchFrame()</code>
Powrót do strony głównej	bezpośredni <code>page.locator()</code>	bezpośredni <code>cy.get()</code>	<code>browser.switchFrame(null)</code>
Czyszczenie kontekstu	niewymagane	niewymagane	wymagane
Synchronizacja	automatyczne oczekiwanie	automatyczne ponawianie	częściowo automatyczna + jawne oczekiwanie

Różnice te przekładają się wprost na objętość kodu. Konieczność jawnego przełączania kontekstu, pobierania referencji do elementów ramki przed każdym przełączeniem oraz resetowania kontekstu w bloku **afterEach** zwiększa liczbę instrukcji w WebdriverIO względem deklaracyjnych łańcuchów **frameLocator()** w Playwright.

5.10. Canvas i Drag & Drop

Dziesiąta kategoria obejmuje scenariusze trudne do zautomatyzowania klasycznymi metodami opartymi na drzewie DOM, ponieważ ich stan przechowywany jest poza nim – w obiekcie **canvas** renderowanym przez bibliotekę Chart.js lub w danych zdarzeń przeciągania (ang. *drag&drop*). Scenariusze podzielono na cztery obszary:

- symulację natywnych zdarzeń przeciągania i upuszczania plików,
- obsługę własnych procedur reagujących na te zdarzenia (ang. *custom handlers*),
- weryfikację zawartości elementu **canvas** z wykresem pogodowym,
- interakcję klawiaturową z grą Maze.

Kategoria ta ujawnia dwie różnice. Pierwszą jest sposób generowania zdarzeń niezwiązanych z prostym kliknięciem. Cypress udostępnia bezpośrednie polecenie **trigger()** [66], które wywołuje zdarzenia bez ręcznego wstrzykiwania funkcji *JavaScript*, podczas gdy w Playwright i WebdriverIO zdarzenie konstruowane jest wewnątrz funkcji wstrzykiwanej do strony. Drugą jest synchronizacja z animacją wykresu, gdzie Playwright posługuje się oczekiwaniem reaktywnym na warunek, a w przyjętych implementacjach Cypress i WebdriverIO zastosowano stałe opóźnienie.

Pierwszym analizowanym aspektem było wyzwalanie natywnych zdarzeń przeciągania, na przykład podświetlenia strefy upuszczania po wystąpieniu zdarzenia **dragenter**. Jak przedstawiono na Listingu 67, w Playwright zdarzenie konstruowane i wysyłane jest wewnątrz funkcji przekazanej do metody **page.evaluate()**, wykonywanej w kontekście strony.

Listing 67. Wysłanie syntetycznego zdarzenia dragenter w Playwright.

```
const dragEnterEvent = new DragEvent('dragenter', {
  bubbles: true,
  cancelable: true,
});
dropzone.dispatchEvent(dragEnterEvent);
```

W Cypress to samo zdarzenie wyzwalane jest pojedynczym poleceniem **trigger()**, bez konieczności wstrzykiwania do strony funkcji *JavaScript* (Listing 68).

Listing 68. Wysłanie syntetycznego zdarzenia dragenter w Cypress.

```
cy.get(dropZone).trigger('dragenter', { bubbles: true, cancelable: true });
```

W WebdriverIO zdarzenie tworzone jest analogicznie jak w Playwright, wewnątrz funkcji przekazanej do metody **browser.execute()** (Listing 69).

Listing 69. Wysłanie syntetycznego zdarzenia dragenter w WebdriverIO.

```
const dragEnterEvent = new DragEvent('dragenter', {
  bubbles: true,
  cancelable: true,
});
dropzone.dispatchEvent(dragEnterEvent);
```

Różnica uwidacznia się również przy symulacji przesłania pliku. Ważne jest to, że w badanej aplikacji strefa dostarczania oparta jest na ukrytym elemencie **input[type=file]** (**#dragdropfile**), dlatego przesłanie pliku realizowano przez ustawienie zawartości tego elementu, a nie przez natywne zdarzenie przeciągania pliku.

W Playwright służy do tego dedykowana metoda **setInputFiles()** przyjmująca bufor z zawartością pliku (Listing 70).

Listing 70. Przesłanie pliku przez ukryty element input w Playwright.

```
await fileInput.setInputFiles({
  name: 'test-image.png',
  mimeType: 'image/png',
  buffer: createTestImageBuffer('test-image'),
});
```

W Cypress wykorzystywane jest polecenie `selectFile()` z opcją `{ force: true }` wymaganą dla ukrytego elementu (Listing 71).

Listing 71. Przesłanie pliku przez ukryty element input w Cypress.

```
cy.get(fileInput).selectFile(  
  {  
    contents: createTestImageBuffer(),  
    fileName: 'test-image.png',  
    mimeType: 'image/png',  
  },  
  { force: true });
```

WebdriverIO nie udostępnia odpowiednika żadnej z tych metod. W przyjętej implementacji obiekt `DataTransfer` oraz instancję `File` budowano ręcznie wewnątrz `browser.execute()`, dekodując zawartość z formatu `base64` na tablicę bajtów, co znacząco wydłuża zapis. Pełną procedurę zawiera Listing 72.

Listing 72. Ręczna budowa DataTransfer i przesłanie pliku w WebdriverIO.

```
await browser.execute((base64Data: string) => {  
  const binaryString = atob(base64Data);  
  const bytes = new Uint8Array(binaryString.length);  
  for (let i = 0; i < binaryString.length; i++) {  
    bytes[i] = binaryString.charCodeAt(i);  
  }  
  
  const dataTransfer = new DataTransfer();  
  const file = new File([bytes], 'test-image.png', { type: 'image/png' });  
  dataTransfer.items.add(file);  
  
  const input = document.getElementById('dragdropfile') as HTMLInputElement;  
  input.files = dataTransfer.files;  
  input.dispatchEvent(new Event('change', { bubbles: true }));  
}, createTestImageBase64());
```

Kolejnym zagadnieniem była weryfikacja zawartości elementu `canvas`. Ponieważ wykres renderowany jest na płótnie graficznym i nie tworzy węzłów DOM, dostęp do danych uzyskano poprzez odwołanie do instancji Chart.js udostępnionej w obiekcie `window`. Sposoby tego dostępu różnią się przede wszystkim przyjętym podejściem do typowania w języku `TypeScript`, będącym decyzją implementacyjną, a nie cechą frameworków. W implementacji Playwright zastosowano przedstawione na Listingu 73 globalne rozszerzenie interfejsu `Window`, dzięki czemu kolejne odwołania do `window.myCharts` nie wymagały lokalnych rzutowań typów.

Listing 73. Odczyt etykiet wykresu z instancji Chart.js w Playwright.

```
const labels = await page.evaluate(() => {  
  const chart = window.myCharts['sampleChart'];  
  return chart.data.labels;  
});
```

W implementacji Cypress dostęp realizowany jest metodą `cy.window()`, a obiekt okna rzutowany jest asercją typu (`win as any`), co oznacza rezygnację z kontroli typów na rzecz zwiezłości (Listing 74).

Listing 74. Odczyt etykiet wykresu z instancji Chart.js w Cypress.

```
cy.window().then((win) => {  
  const chart = (win as any).myCharts['sampleChart'];  
  const labels = chart.data.labels;  
  // ...  
});
```

W implementacji WebdriverIO, odczyt wykonywany jest metodą **browser.execute()**, przy czym rozbudowane rzutowanie typu definiowane jest *inline* przy każdym wywołaniu, ponieważ nie zastosowano globalnej deklaracji interfejsu **Window** (Listing 75).

Listing 75. Odczyt etykiet wykresu z instancji Chart.js w WebdriverIO.

```
const labels = await browser.execute(() => {  
  const chart = (window as Window & { myCharts?: Record<string, { data: { labels:  
string[] } }> }).myCharts!['sampleChart'];  
  return chart.data.labels;  
});
```

Najważniejsza dla stabilności jest synchronizacja z animacją wykresu. Chart.js [67] renderuje wykres z animacją trwającą domyślnie 1000 ms, dlatego odczyt danych przed jej zakończeniem może być zawodny. Oczekiwano na zakończenie animacji poprzez sprawdzenie właściwości **animating** instancji wykresu udostępnianej w obiekcie **window**. Rozwiązanie to jest więc specyficzne dla badanej aplikacji. Jak pokazano na Listingu 76, w Playwright zastosowano oczekiwanie reaktywne metodą **page.waitForFunction()**, ponawiające sprawdzenie warunku do chwili, gdy właściwość **animating** przyjmie wartość fałszywą.

Listing 76. Reaktywne oczekiwanie na zakończenie animacji wykresu w Playwright.

```
await page.waitForFunction(() =>  
  window.myCharts?.['sampleChart'] && !window.myCharts['sampleChart'].animating  
);
```

W Cypress w przyjętej implementacji posłużono się stałym opóźnieniem **cy.wait()** o wartości dobranej z zapasem względem czasu animacji. W WebdriverIO analogicznie zastosowano stałe wstrzymanie metodą **browser.pause()**.

Należy zaznaczyć, że stałe opóźnienie w obu przypadkach jest cechą przyjętej implementacji, a nie ograniczeniem narzędzi. Cypress oferuje mechanizm ponawiania asercji (**cy.window().should()** z funkcją zwrrotną), a WebdriverIO metodą **browser.waitUntil()** umożliwiającą oczekiwanie reaktywne. W badanych scenariuszach renderowania wykresu zdecydowano się jednak na opóźnienie stałe, podczas gdy składnia **page.waitForFunction()** w Playwright w naturalny sposób zachęca do podejścia reaktywnego.

Czwartym obszarem była interakcja klawiaturowa z grą Maze, w której ruch sterowany jest klawiszami strzałek. Weryfikacja stanu gry opierała się na sprawdzeniu obecności komórki z klasą **player** po wykonaniu sekwencji ruchów. W Playwright klawisze wysyłane są metodą **page.keyboard.press()** (Listing 77).

Listing 77. Sterowanie klawiaturą i weryfikacja stanu gry w Playwright.

```
await page.keyboard.press('ArrowUp');  
const playerCell = mazeContainer.locator('.cell.player');  
await expect(playerCell).toHaveCount(1);
```

W Cypress sekwencję klawiszy przekazuje się poleceniu `cy.get('body').type()` z notacją typu `{uparrow}` (Listing 78).

Listing 78. Sterowanie klawiaturą i weryfikacja stanu gry w Cypress.

```
cy.get('body').type('{uparrow}{downarrow}{leftarrow}{rightarrow}'.repeat(5));
cy.get('#mazeContainer .cell.player').should('have.length', 1);
```

W WebdriverIO służy do tego metoda `browser.keys()`, a stan gry odczytywany jest poprzez zmaterializowaną tablicę elementów zwróconą przez `$$()` (Listing 79).

Listing 79. Sterowanie klawiaturą i weryfikacja stanu gry w WebdriverIO.

```
await browser.keys('ArrowUp');
const playerCells = await mazeContainer.$$('.cell.player');
expect(playerCells.length).to.equal(1);
```

Zestawienie omówionych mechanizmów przedstawiono w Tabeli 16.

Tabela 16. Obsługa Canvas i Drag & Drop w analizowanych frameworkach.
Źródło: opracowanie własne.

Aspekt	Playwright	Cypress	WebdriverIO
Wyzwalanie zdarzeń drag	<code>page.evaluate()</code> (wstrzyknięcie funkcji)	<code>trigger()</code> (polecenie wbudowane)	<code>browser.execute()</code> (wstrzyknięcie funkcji)
Symulacja przesłania pliku	<code>setInputFiles()</code> (bufor)	<code>selectFile()</code> (<code>{ force: true }</code>)	ręczna budowa <code>DataTransfer</code> w <code>browser.execute()</code>
Dostęp do instancji Chart.js	<code>page.evaluate()</code> + globalny <code>Window</code>	<code>cy.window()</code> + (<code>win as any</code>)	<code>browser.execute()</code> + rzutowanie <i>inline</i>
Oczekiwanie na animację	<code>waitForFunction()</code> (reaktywne)	<code>cy.wait()</code> (stałe, w badaniu)	<code>browser.pause()</code> (stałe, w badaniu)
Symulacja klawiatury	<code>page.keyboard</code> <code>.press()</code>	<code>cy.get('body')</code> <code>.type()</code>	<code>browser.keys()</code>

Zwiężłość implementacji Cypress wynika z polecenia `trigger()` zastępującego blok wstrzykiwanej funkcji oraz z rzutowania (*win as any*) zamiast rozbudowanych deklaracji typów. Najwyższa objętość WebdriverIO jest następstwem ręcznej budowy obiektów `DataTransfer` i `File` oraz powtarzanego rzutowania typu *inline* przy każdym wywołaniu `browser.execute()`. Szczegółową analizę objętości kodu zawiera rozdział 6.

5.11. WebSockets i komunikacja w czasie rzeczywistym

Jedenasta kategoria obejmuje scenariusze testowania komunikacji dwukierunkowej realizowanej protokołem WebSocket. Testy weryfikują nawiązanie połączenia, wysłanie wiadomości i jej pojawienie się w interfejsie, odbiór wiadomości w czasie rzeczywistym między wieloma klientami oraz zachowanie aplikacji po utracie połączenia. Kategoria ta różnicuje narzędzia przede wszystkim na poziomie dostępu do samego protokołu. Tylko jedno z nich daje bezpośrednio obserwować ramki (ang. *frames*) połączenia, podczas gdy pozostałe sprawdzają komunikację przez stan interfejsu.

Pierwszym analizowanym aspektem było nawiązanie połączenia i wysłanie wiadomości. W Playwright połączenie wykrywane jest metodą `page.waitForEvent('websocket')`, która wstrzymuje dalsze kroki do czasu otwarcia gniazda. Framework wdraża też przedstawiony na Listingu 80 podgląd ruchu na poziomie pojedynczych ramek dzięki zdarzeniom `framesent` oraz `framereceived` rejestrowanym na obiekcie połączenia.

Listing 80. Nasłuchiwanie ramek połączenia WebSocket w Playwright.

```
page.on('websocket', (ws) => {
  ws.on('framesent', (frame) => {
    wsLog.push({ type: 'sent', timestamp: Date.now(), data:
frame.payload.toString() });
  });
  ws.on('framereceived', (frame) => {
    wsLog.push({ type: 'received', timestamp: Date.now(), data:
frame.payload.toString() });
  });
});
```

Cypress nie daje podglądu samego połączenia WebSocket. W testach częściowo zastąpiono go poleceniem `cy.intercept()`, które obejmuje ruch *HTTP/HTTPS* i pozwala zaobserwować jedynie żądanie nawiązania połączenia, bez wglądu w pojedyncze ramki czy wiadomości. Ustabilizowanie połączenia zapewniono stałym opóźnieniem `cy.wait(500)`, a poprawność wysłania wiadomości sprawdzano przez jej obecność w drzewie DOM (Listing 81).

Listing 81. Wysłanie i weryfikacja wiadomości w interfejsie w Cypress.

```
cy.visit('/practice/websocket-chat-v1.html');
cy.wait(500); // Wait for WebSocket connection
// ...
cy.get('#messageInput').type(testMessage);
cy.get('.chat-container .chat-button').click();
cy.get('#messages').should('contain.text', testMessage);
```

W WebdriverIO również nie można obserwować ramek połączenia. Gniazdo ustabilizowano stałą pauzą `browser.pause(500)`, a wysłanie wiadomości potwierdzano jej pojawieniem się w interfejsie. Otwarcie połączenia weryfikowane jest pośrednio. Pomyślne dołączenie do czatu dowodzi, że gniazdo jest aktywne (Listing 82).

Listing 82. Wysłanie wiadomości i ustabilizowanie połączenia w WebdriverIO.

```
await browser.url('/practice/websocket-chat-v1.html');
await browser.pause(500);
// ...
await messageInput.setValue(testMessage);
await sendButton.click();
```

Kolejnym aspektem był odbiór wiadomości w czasie rzeczywistym między dwoma klientami. W Playwright każdego użytkownika reprezentuje osobny kontekst przeglądarki (ang. *browser context*) tworzony metodą `browser.newContext()`, zapewniający izolowane sesje, ciasteczka i niezależne połączenia WebSocket. Każdy kontekst dysponuje własnym obiektem strony, więc przełączanie kontekstu nie jest potrzebne. Symulację dwóch klientów w odrębnych kontekstach przedstawia Listing 83.

Listing 83. Symulacja dwóch klientów w odrębnych kontekstach w Playwright.

```
const context1 = await browser.newContext();
const context2 = await browser.newContext();
const page1 = await context1.newPage();
const page2 = await context2.newPage();
// ...
await sendMessage(page1, testMessage);
await expect(messagesContainer2).toContainText(testMessage, { timeout: 5000 });
```

W Cypress w badanej implementacji nie zrealizowano scenariusza wieloklientowego, ze względu na ograniczenia architektoniczne narzędzia (praca w obrębie jednej karty). Dokumentacja [27] wskazuje obejście z użyciem pakietu *@cypress/puppeteer* dla przepływów wielokartowych. Scenariusze wieloużytkownikowe zastąpiono dodatkowymi przypadkami jednorazowymi m.in. dostępem do obiektu połączenia przez *cy.window()*, wysyłką i odbiorem w obrębie jednej sesji oraz weryfikacją kolejności wiadomości (Listing 84).

Listing 84. Obejście jednorazowe z dostępem do okna przez *cy.window()* w Cypress.

```
cy.window().then((win) => {
  cy.log('Checking WebSocket connection state...');
});
sendMessage('Testing WebSocket connection');
cy.get('#messages').should('contain.text', 'Testing WebSocket connection');
```

W WebdriverIO drugiego klienta symulowano kolejną kartą otwieraną metodą *newWindow()*, która automatycznie przełącza kontekst na nowo otwartą kartę. Karty współdzieli sesję przeglądarki. Przełączanie między nimi realizuje metoda *switchWindow()*. W badanej implementacji użyto wariantu *switchToWindow()* przyjmującego uchwyt okna (ang. *window handle*). Po zakończeniu testu, kartę zamyka się i przywraca poprzedni kontekst (Listing 85).

Listing 85. Symulacja dwóch klientów w odrębnych kartach w WebdriverIO.

```
const window1 = await browser.getWindowHandle();
await browser.newWindow(await browser.getUrl().then(url => new
URL('/practice/websocket-chat-v1.html', url).href));
const window2 = await browser.getWindowHandle();
// ...
await browser.switchToWindow(window1);
await sendMessage(testMessage);
await browser.switchToWindow(window2);
```

Trzecim aspektem była obsługa rozłączenia i symulacja trybu offline, który był realizowany odmiennie w trzech frameworkach. Jak przedstawiono na Listingu 86, w Playwright tryb offline wymusza się metodą *context.setOffline(true)*, a zamknięcie gniazda obserwuje się przez zdarzenia *close* i *socketerror*.

Listing 86. Wymuszenie trybu offline w Playwright.

```
await context.setOffline(true);
await page.waitForFunction(() => !navigator.onLine);
// ...
await context.setOffline(false);
```

W Cypress nie ma wbudowanej symulacji offline, dlatego utratę połączenia odwzorowano namiastką w postaci przeładowania strony (*cy.reload()*) i nawigacji wstecz (*cy.go('back')*), co przedstawia Listing 87.

Listing 87. Odzworowanie rozłączenia przeladowaniem i nawigacją wstecz w Cypress.

```
cy.reload();
cy.wait(500);
// ...
cy.go('back');
cy.wait(500);
cy.get('#loginScreen').should('be.visible');
```

W WebdriverIO warunki sieciowe wymusza metoda `browser.setNetworkConditions({ offline: true })`. Jest to niestandardowe i nieudokumentowane polecenie specyficzne dla rodziny przeglądarek Chromium [68], działające za pośrednictwem protokołu Chrome DevTools (CDP), przez co nie może być stosowane w innych przeglądarkach. W trybie WebDriver BiDi [37], WebdriverIO udostępnia w zamian dedykowaną metodę do emulacji warunków sieciowych `emulation.setNetworkConditions()` (Listing 88).

Listing 88. Wymuszenie trybu offline przez CDP w WebdriverIO.

```
await browser.setNetworkConditions({ offline: true, latency: 0, throughput: 0 });
await browser.pause(2000);
// ...
await browser.setNetworkConditions({ offline: false, latency: 0, throughput: -1 });
```

Zestawienie zastosowanych podejść przedstawiono w Tabeli 17.

Tabela 17. Obsługa komunikacji WebSocket w jedenastej kategorii testowej.
Źródło: opracowanie własne.

Aspekt	Playwright	Cypress	WebdriverIO
Wykrycie połączenia	<code>waitForEvent('websocket')</code> (warunek)	<code>cy.wait(500)</code> (stałe opóźnienie)	<code>browser.pause(500)</code> (stałe opóźnienie)
Podgląd ramek	<code>ws.on('framesent' / 'framereceived')</code>	brak; jedynie żądanie <code>upgrade</code> <code>cy.intercept</code>	brak
Symulacja wielu klientów	<code>browser.newContext()</code> (izolacja)	jedna karta; obejście przez <code>@cypress/puppeteer</code> (poza badaniem)	<code>newWindow()</code> + <code>switchWindow()</code>
Symulacja trybu offline	<code>context.setOffline(true)</code>	<code>cy.reload()</code> / <code>cy.go('back')</code> (namiastka)	<code>setNetworkConditions()</code> przez CDP (tylko Chromium)

Kategoria ta ujawnia różny poziom dostępu narzędzi do protokołu WebSocket. Playwright jako jedyny daje możliwość podejrzenia pojedynczej ramki oraz wykrycia połączenia warunkiem, zamiast używania stałego opóźnienia, podczas gdy Cypress i WebdriverIO sprawdzają komunikację przez stan interfejsu i pauzy o stałej długości. Różny jest też sposób symulacji wielu klientów: pełna izolacja kontekstów w Playwright, współdzielone karty z ręcznym przełączaniem w WebdriverIO oraz zastąpienie scenariuszy wieloużytkownikowych przypadkami jednoużytkownikowymi w Cypress. Wynikające stąd różnice w objętości kodu (LOC) oraz stabilności *flakiness*, na które wpływają m.in. stałe opóźnienia i ręczne przełączanie kontekstu, przeanalizowano w rozdziale 6.

5.12. Alerty, okna i pop-upy

Dwunasta kategoria obejmuje trzy obszary:

- obsługę natywnych okien dialogowych,
- pracę z dodatkowymi oknami i kartami,
- obsługę pop-upów (modali) budowanych w warstwie DOM.

Jedną z największych różnic architektonicznych dotyczy pracy z wieloma oknami. Cypress w kontraście z pozostałymi narzędziami, nie udostępnia natywnego mechanizmu obsługi więcej niż jednej karty przeglądarki.

Pierwszym obszarem jest obsługa natywnych okien dialogowych przeglądarki (*alert*, *confirm*, *prompt*). W Playwright rejestruje się asynchroniczny nasłuch zdarzenia **dialog**, który musi zostać stworzony przed akcją wyzwalającą dialog. Zarejestrowany nasłuch jest przy tym zobowiązany zatwierdzić lub odrzucić dialog metodą **dialog.accept()** lub **dialog.dismiss()**, w przeciwnym razie kliknięcie może zostać zablokowane. Rodzaj okna rozpoznaje się metodą **dialog.type()**. Listing 89 pokazuje przechwycenie i akceptację natywnego dialogu w Playwright.

Listing 89. Przechwycenie i akceptacja natywnego dialogu w Playwright.

```
page.on('dialog', async (dialog: Dialog) => {
  alertMessage = dialog.message();
  await dialog.accept();
});
await alertBoxBtn.click();
```

W Cypress natywne alerty są akceptowane automatycznie i nie blokują wykonania testu. Treść dialogu przechwytuje nasłuch **cy.on('window:alert')**, a zastosowanie zaślepki funkcji (ang. *stub*) daje dodatkowo możliwość weryfikacji argumentu wywołania (Listing 90).

Listing 90. Przechwycenie treści alertu przez zaślepkę w Cypress.

```
const alertStub = cy.stub();
cy.on('window:alert', alertStub);
cy.get('[data-testid="dti-alert-box-btn"]')
  .click()
  .then(() => {
    expect(alertStub).to.be.calledWith('Alert box invoked by button click!');
  });
```

W WebdriverIO obsługę realizują metody zgodne ze standardem W3C WebDriver: **getAlertText()** oraz **acceptAlert()**. Ponieważ aplikacja GAD nie zawsze wyzwała dialog natywny, konieczne było objęcie wywołania konstrukcją **try/catch**. Ważne jest to, że wykorzystana do sprawdzenia obecności okna metoda **isAlertOpen()** nie wchodzi w skład standardowego API frameworka. W dokumentacji WebdriverIO oznaczono ją jako nieudokumentowaną komendę zależną od rodziny przeglądarek Chromium, co ogranicza jej przenośność na inne przeglądarki (Listing 91).

Listing 91. Obsługa natywnego dialogu metodami sterownika w WebdriverIO.

```
await alertBoxBtn.click();
await browser.waitUntil(async () => await browser.isAlertOpen(), {timeout: 2000});
const alertMessage = await browser.getAlertText();
```

Drugim obszarem jest otwarcie dodatkowej karty i weryfikacja jej zawartości – scenariusz najwyraźniej różniący analizowane narzędzia. W Playwright nowe okno przechwytywane jest przez `context.waitForEvent('page')` [69], który zwraca pełnoprawny obiekt **Page**. Przedstawiona na Listingu 92 nowa karta pozostaje w pełni interaktywna, a oba okna są dostępne równocześnie.

Listing 92. Przechwycenie dodatkowego okna jako obiektu Page w Playwright.

```
const newPagePromise = context.waitForEvent('page');
await openNewPageBtn.click();
const newPage = await newPagePromise;
await newPage.waitForLoadState();
expect(newPage.url()).toContain('data-viewer.html');
```

Cypress nie obsługuje pracy z wieloma kartami przeglądarki, co stanowi ograniczenie architektoniczne wynikające z modelu wykonywania testów w obrębie pojedynczego okna. W badanej implementacji zastosowano obejście polegające na zastąpieniu metody `window.open` zaślepką przechwytyującą docelowy adres URL, a następnie nawigacji do tego adresu w tej samej karcie poleceniem `cy.visit()` (Listing 93). Rozwiązanie to jest świadomym wyborem przyjętym na potrzeby eksperymentu, nie zaś docelowym wzorcem. W dokumentacji Cypress wskazywane są również obejścia oparte na wtyczkach, m.in. `@cypress/puppeteer`, których nie zastosowano.

Listing 93. Obejście braku obsługi wielu kart w Cypress.

```
cy.stub(win, 'open').callsFake((url) => {
  capturedUrl = url;
  return null;
});
cy.get('#openNewPageBtn').click();
cy.then(() => {
  cy.visit(capturedUrl.startsWith('/') ? capturedUrl : `/${practice}/new-window-
v1/${capturedUrl}`);
});
```

W WebDriverIO obsługa wielu okien wymaga ręcznego zarządzania uchwytami (ang. *window handles*). Po akcji otwierającej okno oczekuje się jawnie na pojawienie się drugiego uchwytu, identyfikuje nowy uchwyt i przełącza do niego kontekst. W badanej implementacji wykorzystano metodę `switchToWindow()`. Dokumentacja WebDriverIO wskazuje przy tym `switchWindow()` jako wygodniejszy wariant, dopuszczający dopasowanie okna po adresie URL lub tytule. Każdy test musi samodzielnie zamknąć okno i przywrócić kontekst pierwotny (Listing 94).

Listing 94. Ręczne zarządzanie uchwytami okien w WebDriverIO.

```
const originalWindow = await browser.getWindowHandle();
await openNewPageBtn.click();
await browser.waitUntil(
  async () => (await browser.getWindowHandles()).length > 1,
  { timeout: 10000 }
);
const allWindows = await browser.getWindowHandles();
const newWindow = allWindows.find((w: string) => w !== originalWindow) as string;
await browser.switchToWindow(newWindow);
// ...
await browser.closeWindow();
await browser.switchToWindow(originalWindow);
```

Trzecim obszarem są pop-upy realizowane w warstwie DOM. W aplikacji GAD jest to modal z przyciskami Accept i Cancel, wyświetlany bez udziału natywnego dialogu przeglądarki. W odróżnieniu od alertów, elementy te traktowane są jak zwykłe węzły DOM, dlatego różnice sprowadzają się do strategii lokalizacji przycisku oraz modelu oczekiwania. Jak pokazano na Listingu 95, Playwright posługuje się selektorem tekstowym **:has-text()** wraz z automatycznym oczekiwaniem na zmianę stanu interfejsu.

Listing 95. Obsługa modala DOM w Playwright.

```
const cancelBtn = page.locator('.modal-footer button:has-text("Cancel")');
await cancelBtn.click();
await expect(modal).toBeHidden();
await expect(cancelAlert).toContainText('Modal was cancelled by user!');
```

Cypress lokalizuje przycisk poleceniem **.contains()** w łańcuchu z automatycznym ponawianiem asercji. Zamknięcie modala weryfikuje przez sprawdzenie, że element został usunięty z drzewa DOM (Listing 96).

Listing 96. Obsługa modala DOM w Cypress.

```
cy.get('.modal-footer button').contains('Cancel').click();
cy.get('.modal').should('not.exist');
cy.get('[data-testid="dti-simple-alert-with-custom-message"]')
  .should('be.visible')
  .and('contain.text', 'Modal was cancelled by user!');
```

WebdriverIO lokalizuje przycisk za pomocą wyrażenia XPath z funkcją **text()**, a na zniknięcie modala oczekuje jawnie metodą **browser.waitUntil()**, co przedstawia częściowo automatyczny model synchronizacji z jawnym oczekiwaniem, ujętym na Listingu 97.

Listing 97. Obsługa modala DOM w WebdriverIO.

```
const cancelBtn = await $('//div[contains(@class,"modal-
footer")]//button[text()="Cancel"]');
await cancelBtn.click();
const modal = await $('.modal');
await browser.waitUntil(async () => !(await modal.isDisplayed()), { timeout: 5000
});
```

Różnice w obsłudze alertów, dodatkowych okien i pop-upów zestawiono w Tabeli 18.

Tabela 18. Obsługa alertów, okien i pop-upów w analizowanych frameworkach.
Źródło: opracowanie własne.

Aspekt	Playwright	Cypress	WebdriverIO
Natywny dialog (alert / confirm)	page.on ('dialog')	auto-akceptacja; cy.on ('window:alert')	acceptAlert() (W3C)
Treść / obecność dialogu	dialog .message()	zaślepka cy.stub	getAlertText(); isAlertOpen() w Chromium
Otwarcie dodatkowej karty	waitForEvent ('page')	brak natywnej; window.open + cy.visit()	getWindowHandles()

Aspekt	Playwright	Cypress	WebdriverIO
Przełączanie / sprzątanie okien	obiekty <code>Page</code> ; <code>close()</code>	praca w jednej karcie	<code>switchWindow()</code> ; <code>closeWindow()</code>
Selektor przycisku pop-upa	<code>:has-text()</code>	<code>.contains()</code>	<code>XPath text()</code>
Model oczekiwania na pop-up	<i>auto-wait</i>	<i>auto-retry</i>	<code>waitUntil()</code>

Najwyraźniejszą różnicą jest model pracy z wieloma oknami. Playwright udostępnia każde okno jako odrębny, w pełni interaktywny obiekt. WebdriverIO wymaga jawnego zarządzania uchwytami i przełączania kontekstu. Wobec braku natywnej obsługi, Cypress opiera się na obejściu ograniczającym scenariusz do pojedynczej karty. W obszarze pop-upów różnice są mniejsze i sprowadzają się do strategii selektorów oraz modelu oczekiwania. Różnice te mają też wymierne konsekwencje dla objętości kodu. W badanej kategorii implementacja w WebdriverIO wymagała znacząco większej liczby linii kodu niż pozostałe narzędzia, co wynika z konieczności jawnego zarządzania uchwytami okien oraz jawnego oczekiwania. Wpływ tych różnic na objętość kodu i czas wykonania omówiono w rozdziale 6.

6. Wyniki pomiarów i analiza porównawcza

W tym rozdziale przedstawiono zbiorcze wyniki pomiarów ilościowych przeprowadzonych zgodnie z metodologią opisaną w rozdziale 4 oraz ich analizę porównawczą. Kolejne podrozdziały omawiają czas wykonania testów w trybie sekwencyjnym i równoległym, stabilność testów, objętość kodu, zużycie zasobów systemowych oraz raportowanie. Całość zamyka macierz porównawcza, która agreguje wszystkie metryki wraz z wykresem radarowym profili frameworków.

6.1. Czas wykonania testów

Pomiary czasu wykonania objęły 17 kombinacji framework × przeglądarka × liczba procesów roboczych, a ich zakres zestawiono w Tabeli 19. Łącznie przeprowadzono 135 pełnych uruchomień zestawów testowych, odpowiadających ponad 67 tysiącom wykonań pojedynczych testów. Miarą czasu jest czas rzeczywisty całego uruchomienia, rejestrowany przez orkiestrator pomiarowy opisany w podrozdziale 4.4. Zestaw nie obejmuje kombinacji Cypress + WebKit ani WebdriverIO + WebKit. Tryb równoległy Cypress pominięto z przyczyn omówionych w podrozdziale 6.1.2.

Tabela 19. Zakres pomiarowy. Źródło: opracowanie własne.

Tryb	Kombinacje	Uruchomień na kombinację	Uruchomień łącznie	Liczba wykonanych testów
Sekwencyjny (1 proces roboczy)	10	10	100	50 000
Równoległy (4 procesy robocze)	7	5	35	17 230
Razem	17	15	135	67 230

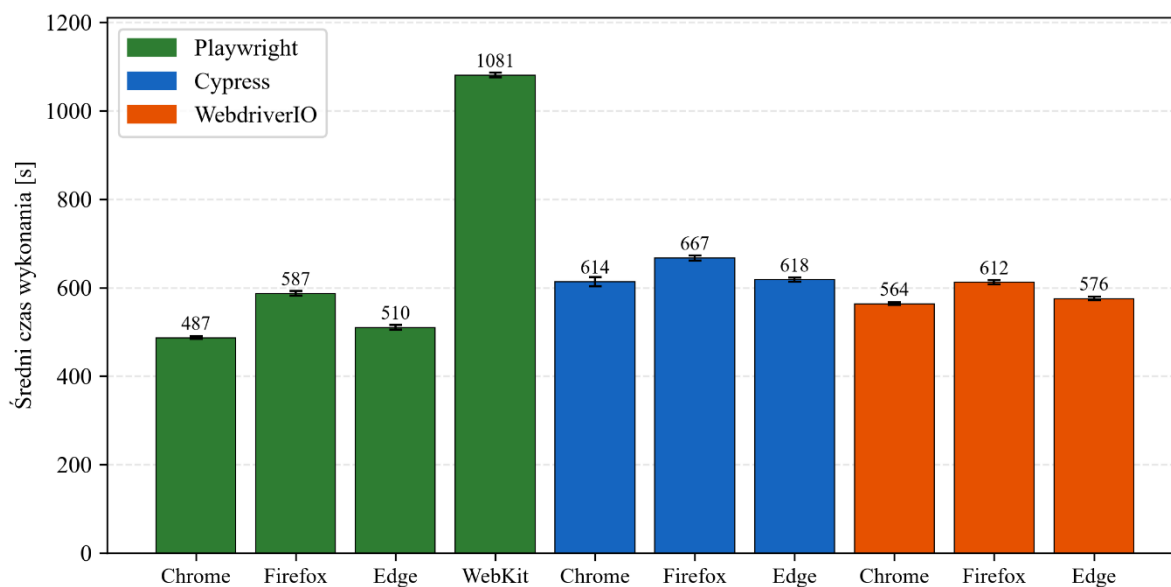
6.1.1. Tryb sekwencyjny

Wyniki pomiarów w trybie sekwencyjnym zestawiono w Tabeli 20. Rozpiętość rezultatów okazała się znacząca. Najkrótszy średni czas uzyskała kombinacja Playwright + Chrome, która wykonuje pełny zestaw testów średnio w 486,6 s, czyli w nieco ponad 8 minut. Na ostatnim miejscu znalazła się kombinacja Playwright + WebKit z czasem 1080,7 s, czyli około 18 minut. Ten sam zestaw testów wykonuje się więc 2,22 razy dłużej wyłącznie za sprawą zmiany silnika przeglądarki. Wynik kombinacji z silnikiem WebKit wymaga jednak zastrzeżenia interpretacyjnego. Znaczna część tego czasu to oczekiwanie na wyczerpanie limitów czasu w testach, które w tym silniku nie przechodzą. Czas ten nie pokazuje zatem w pełni szybkości wykonania scenariuszy zakończonych powodzeniem. Dekompozycję tego zjawiska przedstawiono w podrozdziale 6.1.3.

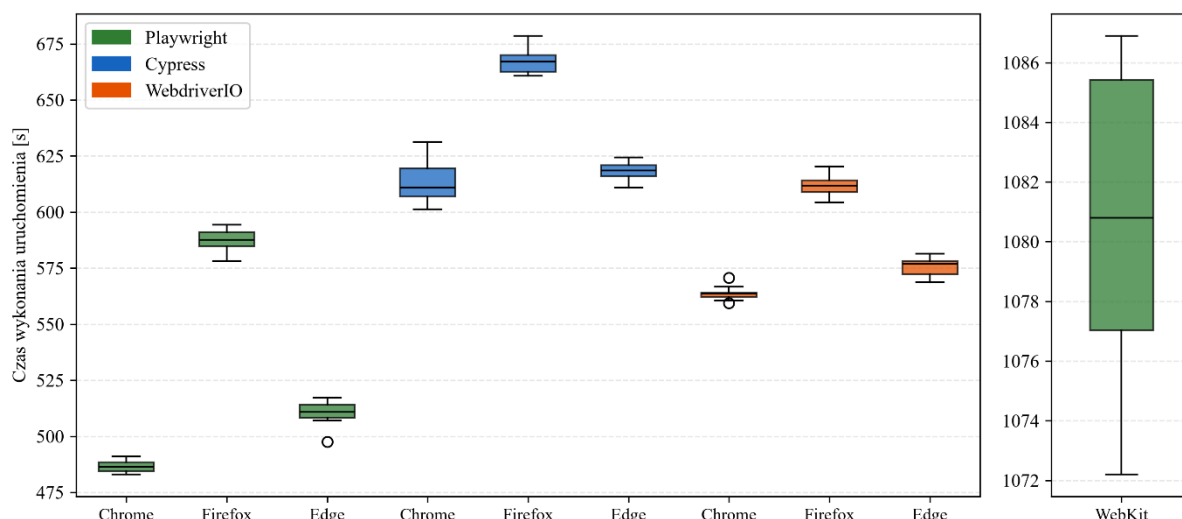
Tabela 20. Czas wykonania testów w trybie sekwencyjnym (1 proces roboczy, 10 uruchomień na kombinację). Źródło: opracowanie własne.

Kombinacja	Testy	Śr. [s]	Mediana [s]	Std [s]	Min [s]	Max [s]	CV [%]	Pass rate [%]
Playwright Chrome	524	486,6	486,5	2,9	483,0	491,0	0,59	99,81
Playwright Firefox	524	587,0	587,7	5,4	578,2	594,3	0,93	99,62
Playwright Edge	524	510,3	510,9	5,6	497,5	517,3	1,09	99,98
Playwright WebKit	524	1080,7	1080,8	5,2	1072,2	1086,9	0,48	91,51
Cypress Chrome	518	613,8	610,8	10,6	601,2	631,3	1,73	100,00
Cypress Firefox	518	667,4	667,2	5,8	660,8	678,6	0,86	99,96
Cypress Edge	518	618,0	618,5	4,3	610,9	624,3	0,69	100,00
WebdriverIO Chrome	450	563,7	563,6	3,1	559,4	570,6	0,56	100,00
WebdriverIO Firefox	450	612,2	611,8	4,9	604,3	620,3	0,81	92,89
WebdriverIO Edge	450	575,6	576,8	4,4	568,7	581,4	0,77	99,93

Średnie czasy wykonania wszystkich kombinacji przedstawiono na Rysunku 2, a rozkłady czasów z poszczególnych uruchomień na Rysunku 3.



Rysunek 2. Średni czas wykonania zestawu testów w trybie sekwencyjnym. Źródło: opracowanie własne.



Rysunek 3. Rozkład czasów wykonania w poszczególnych uruchomieniach (tryb sekwencyjny). Źródło: opracowanie własne.

Zestawy testowe różnią się jednak liczebnością, dlatego miarodajne porównanie wymaga normalizacji czasu wykonania do pojedynczego testu, której wyniki przedstawiono w Tabeli 21. Po jej zastosowaniu w przeglądarce Chrome najkrótszy czas wykonania uzyskał Playwright (929 ms), wyprzedzając Cypress (1185 ms) o około 22% oraz WebdriverIO (1253 ms) o około 26%. Playwright pozostaje najszybszym frameworkiem również w pozostałych przeglądarkach podczas gdy WebdriverIO osiąga najdłuższy czas wykonania.

Tabela 21. Normalizacja czasu pojedynczego testu w trybie sekwencyjnym. Źródło: opracowanie własne.

Framework	Chrome [ms]	Firefox [ms]	Edge [ms]	WebKit [ms]
Playwright	929	1120	974	2062
Cypress	1185	1288	1193	n.d.
WebdriverIO	1253	1360	1279	n.d.

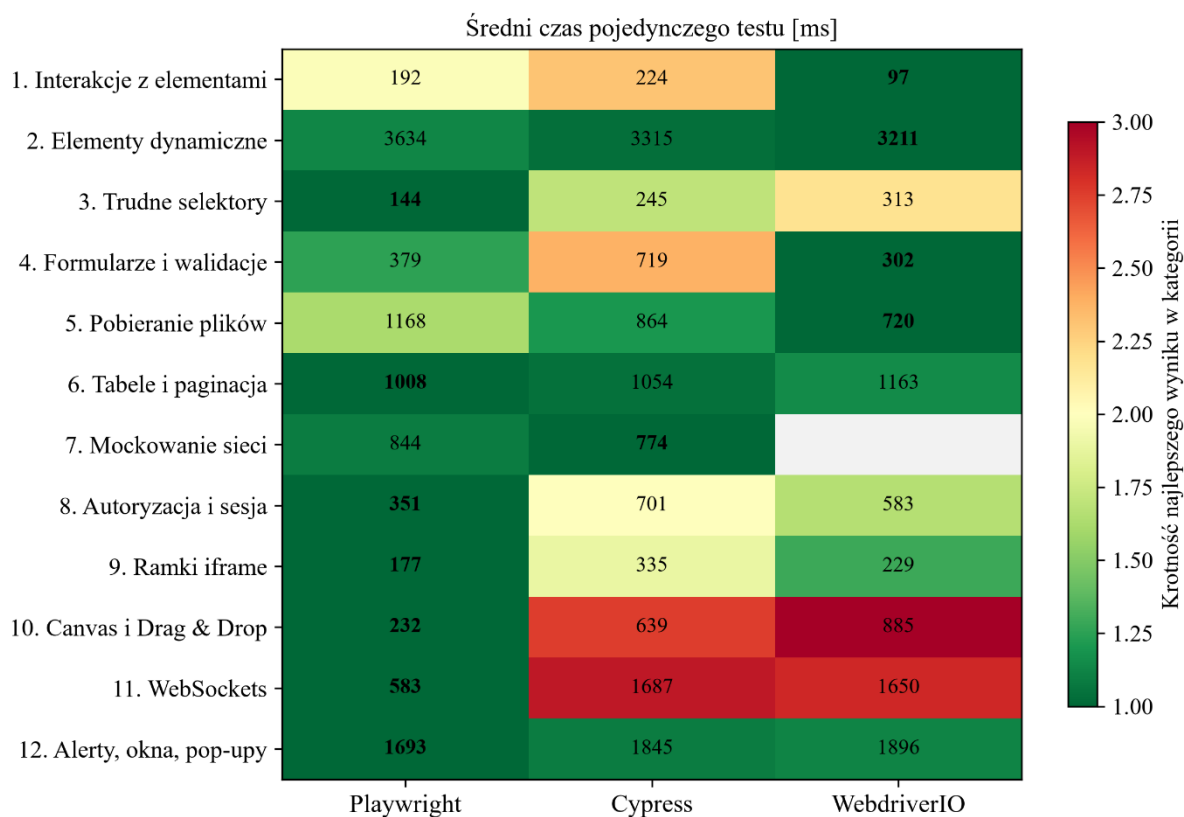
Warto zwrócić uwagę na powtarzalność pomiarów. Współczynnik zmienności (CV) czasu wykonania dla żadnej z analizowanych kombinacji nie przekroczył 2% (Tabela 20), pozostając tym samym poniżej przyjmowanego w literaturze progu 5–10%, uznawanego za wartość wskazującą na stabilność pomiarów. Wysoką powtarzalność zachowały również kombinacje z testami deterministycznie odrzucanymi, czyli Playwright + WebKit oraz WebdriverIO + Firefox (por. w podrozdziale 6.2). Testy te kończą się niepowodzeniem w identyczny sposób w każdym uruchomieniu, przez co nie wprowadzają losowości do pomiaru.

Zbiorczy czas wykonania jest wypadkową bardzo zróżnicowanych wyników w poszczególnych kategoriach testowych. W Tabeli 22 zestawiono średni czas pojedynczego testu w każdej z dwunastu kategorii dla bazowej konfiguracji Chrome, a najkrótszy czas w danej kategorii wyróżniono pogrubieniem. Playwright uzyskał najkrótszy czas w siedmiu kategoriach, WebdriverIO w czterech, a Cypress w jednej, w której WebdriverIO nie uczestniczy. Za tym podziałem kryje się czytelny wzorec. WebdriverIO okazuje się najszybszy tam, gdzie interakcje są proste i krótkie, np. przy interakcjach z elementami czy formularzami. Niski narzut pojedynczych komend przeważa

wówczas nad korzyściami z automatycznego oczekiwania. Z kolei Playwright wygrywa w kategoriach złożonych, takich jak autoryzacja, ramki *iframes*, Canvas, WebSockets czy obsługa okien, gdzie natywne API i automatyczne oczekiwanie redukują liczbę operacji synchronizacyjnych. Wartości w kategorii 5 należy interpretować ostrożnie, ponieważ WebdriverIO implementuje tam wyraźnie większy zakres scenariuszy niż pozostałe frameworki (por. w podrozdziale 4.2). Rozkład czasów w podziale na kategorie zwizualizowano na Rysunku 4.

Tabela 22. Średni czas pojedynczego testu w podziale na kategorie (Chrome, tryb sekwencyjny). Źródło: opracowanie własne.

Kategoria	Playwright [ms]	Cypress [ms]	WebdriverIO [ms]
1. Podstawowe interakcje z elementami	192	224	97
2. Elementy dynamiczne i mechanizmy czekania	3634	3315	3211
3. Zaawansowane selektory i Shadow DOM	144	245	313
4. Formularze i walidacje	379	719	302
5. Pobieranie plików	1168	864	720
6. Tabele i paginacja	1008	1054	1163
7. Przechwytywanie sieci i mockowanie API	844	774	—
8. Autoryzacja i zarządzanie sesją	351	701	583
9. Iframes i zagnieżdżone konteksty	177	335	229
10. Canvas i Drag & Drop	232	639	885
11. WebSockets i komunikacja real-time	583	1687	1650
12. Alerty, okna i pop-upy	1693	1845	1896



Rysunek 4. Średni czas pojedynczego testu w podziale na kategorie testowe (Chrome, tryb sekwencyjny). Źródło: opracowanie własne.

6.1.2. Tryb równoległy

Pomiary w trybie równoległym przeprowadzono z czterema procesami roboczymi dla Playwright oraz WebdriverIO. Każdą kombinację uruchomiono 5 razy w tych samych warunkach środowiskowych co pomiary sekwencyjne. W pomiarze nie uczestniczył Cypress, ponieważ framework nie oferuje natywnej równoległości lokalnej. Dostępna w nim paralelizacja rozdziela testy pomiędzy osobne maszyny agenta CI, ale wymaga to usługi chmurowej Cypress Cloud [27].

Wyniki zestawiono w Tabeli 23. Przyspieszenie (ang. *speed-up*) obliczono jako stosunek średniego czasu sekwencyjnego do średniego czasu równoległego danej kombinacji, a efektywność wykonania równoległego jako iloraz przyspieszenia i liczby procesów roboczych.

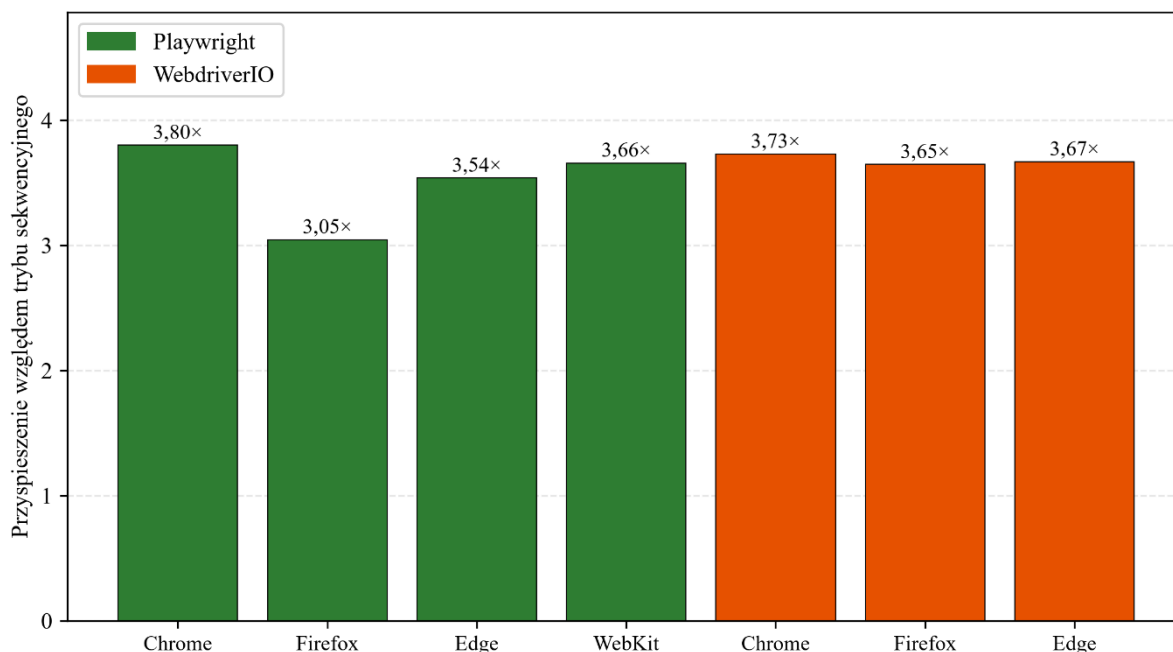
Tabela 23. Czas wykonania testów w trybie równoległym (4 procesy robocze, 5 uruchomień na kombinację). Źródło: opracowanie własne.

Kombinacja	Śr. [s]	Std [s]	CV [%]	Przyspieszenie	Efektywność [%]	Pass rate [%]
Playwright Chrome	128,1	0,9	0,66	3,80×	95,0	100,00
Playwright Firefox	192,8	1,4	0,72	3,05×	76,2	99,62
Playwright Edge	144,2	1,6	1,08	3,54×	88,5	100,00
Playwright WebKit	295,5	2,1	0,72	3,66×	91,5	91,56

Kombinacja	Śr. [s]	Std [s]	CV [%]	Przyspieszenie	Efektywność [%]	Pass rate [%]
WebdriverIO Chrome	151,1	1,0	0,66	3,73×	93,2	100,00
WebdriverIO Firefox	167,9	0,9	0,51	3,65×	91,2	92,84
WebdriverIO Edge	156,9	1,2	0,76	3,67×	91,8	100,00

Oba frameworki skalują się bardzo dobrze. Przyspieszenie mieści się w przedziale 3,05–3,80× przy czterech procesach roboczych, co odpowiada efektywności od 76% do 95%. Największe przyspieszenie uzyskał Playwright w Chrome, najmniejsze zaś ten sam framework w Firefox. WebdriverIO utrzymuje niemal stałą efektywność, przekraczającą 90% niezależnie od przeglądarki, co wskazuje na jednorodny koszt wykonania równoległego w architekturze opartej na niezależnych sesjach WebDriver. Słabsza skalowalność pary Playwright + Firefox koresponduje z podwyższonym narzutem tej kombinacji już w trybie sekwencyjnym. W efekcie względny narzut Firefox rośnie z +20,6% w trybie sekwencyjnym do +50,5% w trybie równoległym względem Chrome. Wartości przyspieszenia wszystkich kombinacji przedstawiono na Rysunku 5.

Warto spojrzeć na te wyniki także w wartościach bezwzględnych. Najszybsza kombinacja, Playwright + Chrome, wykonuje pełny zestaw testów w nieco ponad dwie minuty. Nawet najwolniejsza kombinacja równoległa, Playwright + WebKit, jest szybsza od najszybszej kombinacji sekwencyjnej (por. Tabela 20 i Tabela 23). Powtarzalność pomiarów pozostała porównywalna z trybem sekwencyjnym, a wskaźniki *pass rate* zgodne z jego wynikami. Wykonanie równoległe nie wprowadziło więc dodatkowej niestabilności. Odnotowano przy tym jedną ciekawą różnicę jakościową. Test paginacji z kategorii 6, deterministycznie odrzucany przez parę Playwright + Chrome w trybie sekwencyjnym, w trybie równoległym przechodził we wszystkich uruchomieniach.



Rysunek 5. Przyspieszenie wykonania przy 4 procesach roboczych względem trybu sekwencyjnego. Źródło: opracowanie własne.

6.1.3. Porównanie między przeglądarkami

Wpływ silnika przeglądarki na czas wykonania wyrażono jako względny narzut w odniesieniu do Chrome, stanowiącego konfigurację bazową każdego frameworka (Tabela 24).

Tabela 24. Względny narzut czasu wykonania przy zmianie przeglądarki (tryb sekwencyjny, Chrome = 0%). Źródło: opracowanie własne.

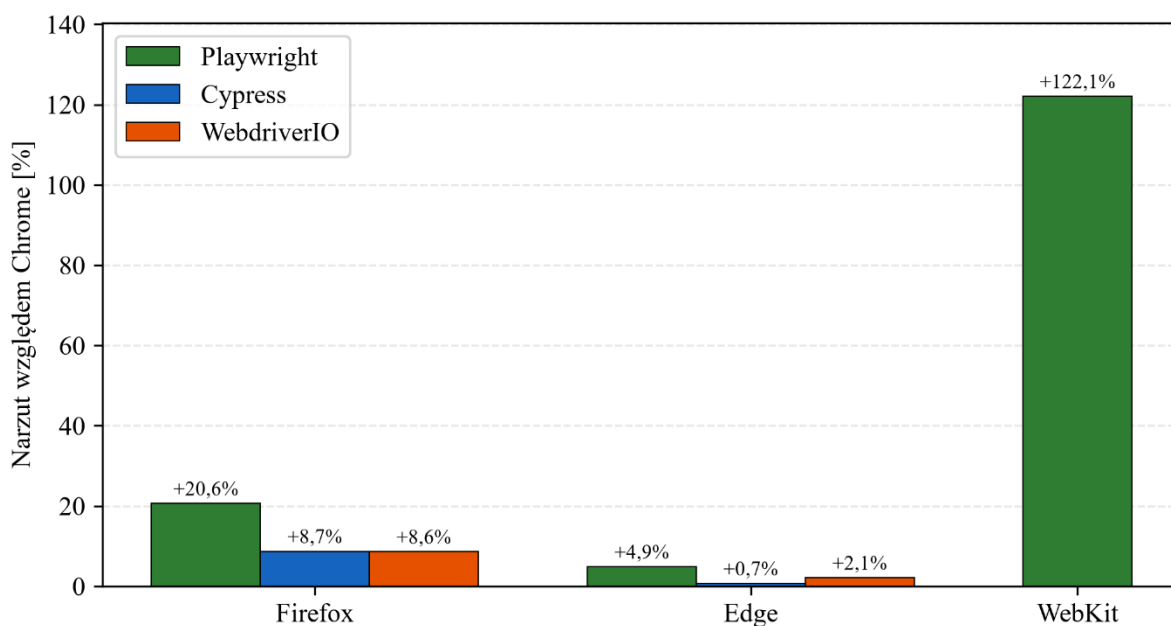
Framework	Firefox[%]	Edge[%]	WebKit[%]
Playwright	+20,6	+4,9	+122,1
Cypress	+8,7	+0,7	n.d.
WebdriverIO	+8,6	+2,1	n.d.

Najmniej kosztuje zamiana Chrome na Microsoft Edge. W żadnym z analizowanych frameworków nie przekroczył on 4,9%. Ponieważ obie przeglądarki korzystają z tego samego silnika, jej zmiana ma jedynie marginalny wpływ na czas wykonania testów. Najmniejszą różnicę odnotowano dla Cypress, co można wiązać z jego architekturą. Framework działa wewnątrz przeglądarki więc silniej zależy od samego silnika renderującego niż od otaczającej go aplikacji.

Inaczej wygląda sytuacja w Firefox. Narzut jest niemal identyczny dla Cypress i WebdriverIO, ale ponad dwukrotnie wyższy dla Playwright. Różnicę tę tłumaczy architektura komunikacji. W przeglądarkach opartych na Chromium, Playwright korzysta natywnie z protokołu Chrome DevTools, natomiast w Firefox posługuje się własnym protokołem automatyzacji, utrzymywanym we własnej kompilacji przeglądarki. Cypress i WebdriverIO sterują oboma silnikami tymi samymi mechanizmami, dzięki czemu proporcja kosztów pozostaje stała.

Najwyższą cenę ma silnik WebKit. Kombinacja Playwright + WebKit potrzebuje na wykonanie testów o 122,1% więcej czasu niż Playwright + Chrome, a więc przeszło dwukrotnie, przy wyraźnie obniżonym *pass rate* (por. w podrozdziale 6.2). Zarazem jest to jedyna w badanym zestawie możliwość uruchomienia testów na silniku przeglądarki Safari w systemie Windows. WebdriverIO w ogóle nie wspiera tego silnika, a Cypress oferuje jedynie wsparcie eksperymentalne [21]. Wysoki koszt tego przeniesienia, należy zatem interpretować jako cenę funkcjonalności, a nie jako słabość frameworka. Zróżnicowanie narzutów przedstawiono na Rysunku 6.

Dekompozycja czasu według kategorii pokazuje, że narzut ten jest w znacznej mierze wynikiem testów deterministycznie odrzucanych. Zdecydowana większość z nich, skoncentrowana w kategorii 8, kończy się w tym silniku wyczerpaniem limitu czasu, a oczekiwanie kumuluje się w całkowitym czasie uruchomienia. Po wyłączeniu kategorii 8 z sumy czasów testów obu porównywanych kombinacji, narzut silnika WebKit spada do około +33,5%. Rzeczywisty koszt wykonania scenariuszy zakończonych powodzeniem jest zatem znacznie niższy, niż sugeruje surowa wartość odniesiona do Chrome.



Rysunek 6. Względny narzut czasu wykonania przy zmianie przeglądarki (tryb sekwencyjny, Chrome = 0%). Źródło: opracowanie własne.

6.2. Stabilność testów

Stabilność oceniono klasyfikując każdy unikatowy test w ramach danej kombinacji na podstawie wyników wszystkich uruchomień. Test stabilny (ang. *pass*) kończy się w każdym uruchomieniu wynikiem pozytywnym. Test niestabilny (ang. *flaky*), zgodnie z definicją [7], kończy się co najmniej jednym wynikiem pozytywnym i negatywnym w identycznych warunkach. Test deterministycznie odrzucany (ang. *always fail*) nie przechodzi w żadnym uruchomieniu. Wskaźnik *flakiness rate*, zdefiniowany w podrozdziale 4.3.2, oznacza odsetek testów niestabilnych w danej kombinacji.

Skala zjawiska niestabilności okazała się bardzo mała. W trybie sekwencyjnym, obejmującym dokładnie 50 000 wykonań testów, zidentyfikowano zaledwie 5 testów niestabilnych. Ich pełną listę zawiera Tabela 25.

Wskaźnik *flakiness rate* w żadnej z analizowanych kombinacji nie przekroczył 0,44%. Niestabilność wykazało jedynie 5 z blisko 5 000 unikatowych par test × kombinacja, co odpowiada około 0,02% wszystkich 50 000 wykonań, w których uzyskano wynik odbiegający od dominującego. Trzy kombinacje Cypress + Chrome, Cypress + Edge oraz WebdriverIO + Chrome nie wykazały ani jednego testu niestabilnego, ani deterministycznie odrzucanego. Zbiorcze wyniki klasyfikacji dla wszystkich kombinacji zestawiono w Tabeli 26.

Tabela 25. Testy niestabilne zidentyfikowane w trybie sekwencyjnym.
Źródło: opracowanie własne.

Kombinacja	Kat.	Test	Przeszedł
Playwright Edge	12	should close new window with back button	9/10
Playwright WebKit	10	should not duplicate images with same name	5/10

Kombinacja	Kat.	Test	Przeszedł
Cypress Firefox	8	should maintain session when navigating back and forth	8/10
WebdriverIO Edge	4	should show alert when submitting without confirmation checkbox	9/10
WebdriverIO Edge	12	should display advertisement overlay on page load	8/10

Tabela 26. Klasyfikacja stabilności testów per kombinacja (tryb sekwencyjny).
Źródło: opracowanie własne.

Kombinacja	Testy	Stabilne	Niestabilne	Odrzucone	Flakiness rate [%]
Playwright Chrome	524	523	0	1	0,00
Playwright Firefox	524	523	0	1	0,00
Playwright Edge	524	523	1	0	0,19
Playwright WebKit	524	480	1	43	0,21
Cypress Chrome	518	518	0	0	0,00
Cypress Firefox	518	517	1	0	0,19
Cypress Edge	518	518	0	0	0,00
WebdriverIO Chrome	450	450	0	0	0,00
WebdriverIO Firefox	450	418	0	32	0,00
WebdriverIO Edge	450	448	2	0	0,44

Odrębną kategorię pokazuje 77 testów deterministycznie odrzucanych. Koncentrują się one przede wszystkim w dwóch kombinacjach: 43 z nich dotyczy pary Playwright + WebKit, a 32 pary WebdriverIO + Firefox. Po jednym odrzucanym teście odnotowano również w parach Playwright + Chrome i Playwright + Firefox. Ich rozkład nie jest przypadkowy, a klasyfikację przyczyn zestawiono w Tabeli 27. W kombinacji Playwright + WebKit niemal wszystkie odrzucone testy koncentrują się w kategorii 8, obejmującej autoryzację i zarządzanie sesją, co wskazuje na restrykcyjną politykę silnika WebKit wobec ciasteczek i lokalnego magazynu danych. W kombinacji WebdriverIO + Firefox przyczyny są bardziej zróżnicowane i wynikają z zabezpieczeń samej przeglądarki oraz ograniczeń sterownika geckodriver. Osobny charakter mają pojedyncze testy specyficzne dla danej pary framework × przeglądarka. Żaden test nie był odrzucany w więcej niż jednym frameworku, co wyklucza defekt aplikacji testowej jako przyczynę. Szczegółową interpretację tych ograniczeń przedstawiono w podrozdziałach 7.1 i 7.2.

Tabela 27. Klasyfikacja przyczyn testów deterministycznie odrzucanych (tryb sekwencyjny). Źródło: opracowanie własne.

Kombinacja	Kat.	Liczba	Przyczyna
Playwright WebKit	8	41	Restrykcyjna polityka silnika wobec ciasteczek i lokalnego magazynu danych skutkująca niepowodzeniem asercji zmiany adresu URL po zalogowaniu
Playwright WebKit	2	2	Obsługa elementów dynamicznych w silniku WebKit

Kombinacja	Kat.	Liczba	Przyczyna
WebdriverIO Firefox	8	18	Blokada uwierzytelniania HTTP Basic poprzez adres URL z osadzonymi poświadczeniami, wprowadzona w przeglądarce ze względów bezpieczeństwa
WebdriverIO Firefox	4	9	Brak implementacji endpoint przesyłania plików geckodriver
WebdriverIO Firefox	11	4	Mechanizm ograniczania warunków sieciowych dostępny wyłącznie w rodzinie przeglądarek Chromium
WebdriverIO Firefox	9	1	Pojedynczy przypadek specyficzny dla silnika przeglądarki
Playwright Chrome	6	1	Wykrywanie przejściowego stanu DOM podczas tranzycji paginacji
Playwright Firefox	7	1	Pomiar czasu żądania specyficzny dla pary framework × silnik
Razem		77	

Tryb równoległy, przedstawiony w Tabeli 28, obejmuje siedem kombinacji i 17 230 wykonań testowych. Zestawienie z wynikami trybu sekwencyjnego z Tabeli 26 wykazuje wyższą stabilność. Liczba testów niestabilnych spadła z pięciu do dwóch. W trybie równoległym pozostały jeden test w parze Playwright + WebKit (kategoria 11) i jeden w parze WebdriverIO + Firefox (kategoria 8). Zbiór testów deterministycznie odrzucanych pozostał niemal identyczny (76 wobec 77 w kombinacjach objętych oboma trybami). Jediną różnicą było przejście testu paginacji w parze Playwright + Chrome, omówione w podrozdziale 6.1.2. Wynik ten ma znaczenie praktyczne. Równoległe wykonywanie testów na jednej maszynie bywa wskazywane jako źródło niestabilności, jednak w badanym środowisku z izolacją stanu nie podniosło ono poziomu *flakiness*, a wręcz je obniżyło.

Tabela 28. Klasyfikacja stabilności testów per kombinacja (tryb równoległy).
Źródło: opracowanie własne.

Kombinacja	Testy	Stabilne	Niestabilne	Odrzucane	Flakiness rate [%]
Playwright Chrome	524	524	0	0	0,00
Playwright Firefox	524	523	0	1	0,00
Playwright Edge	524	524	0	0	0,00
Playwright WebKit	524	480	1	43	0,21
WebdriverIO Chrome	450	450	0	0	0,00
WebdriverIO Firefox	450	418	1	32	0,24
WebdriverIO Edge	450	448	0	0	0,00

6.3. Objętość kodu

Objętość kodu zmierzono narzędziem cloc [44] zgodnie z metodologią opisaną w podrozdziale 4.3.3, analizując wyłącznie linie kodu właściwego (kategoria *code*), z pominięciem linii pustych i komentarzy. Pomiar wykonano jednorazowo, oddzielnie dla każdego z dwunastu folderów kategorii

w trzech repozytoriach testowych. Wyniki w podziale na kategorie, wraz z wartościami znormalizowanymi do liczby testów, przedstawiono w Tabeli 29.

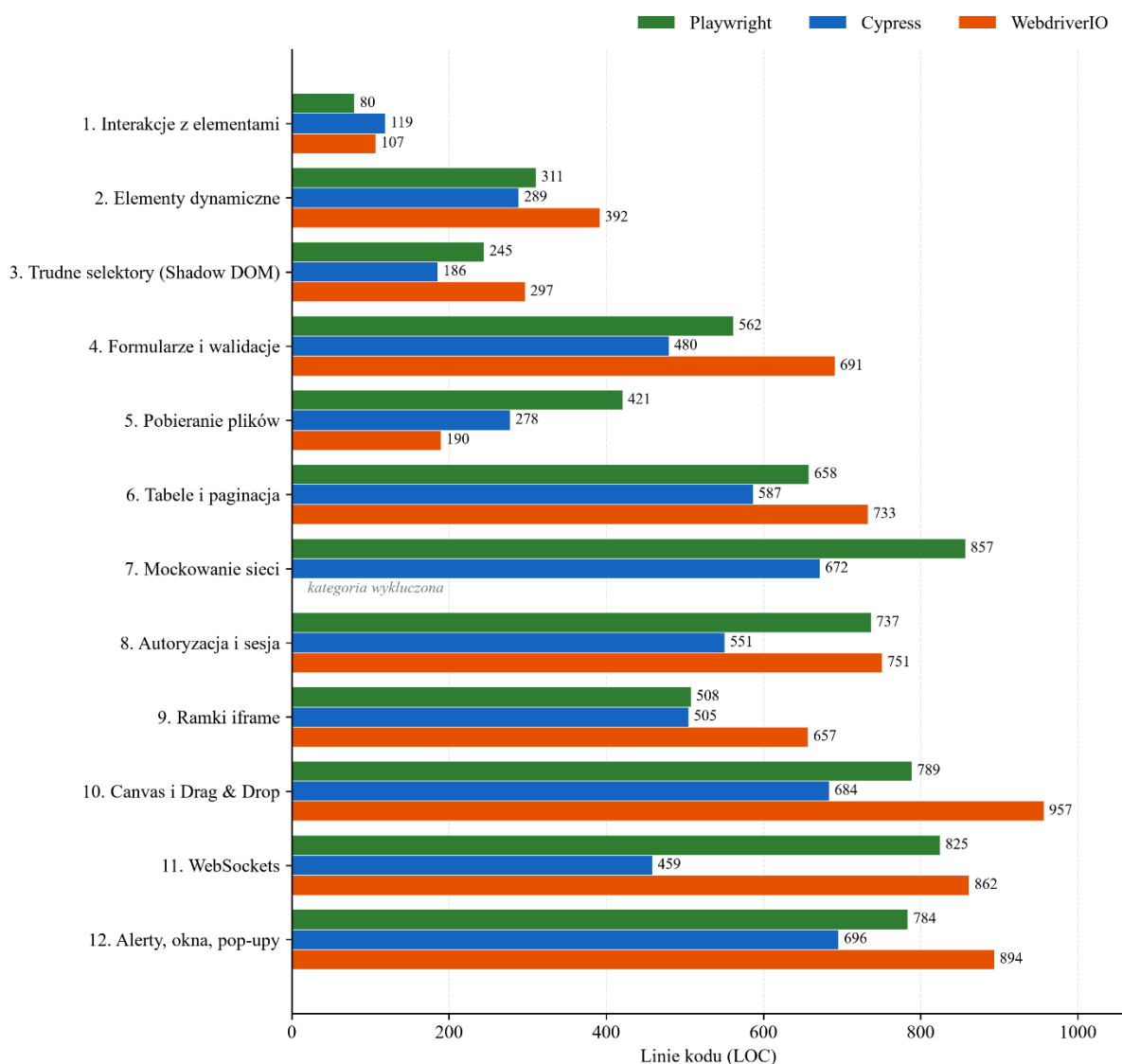
Tabela 29. Objętość kodu [LOC] w podziale na kategorie testowe.
Źródło: opracowanie własne.

Kategoria	Playwright	Cypress	WebdriverIO
1. Podstawowe interakcje z elementami	80	119	107
2. Elementy dynamiczne i mechanizmy czekania	311	289	392
3. Zaawansowane selektory i Shadow DOM	245	186	297
4. Formularze i walidacje	562	480	691
5. Pobieranie plików	421	278	190
6. Tabele i paginacja	658	587	733
7. Przechwytywanie sieci i mockowanie API	857	672	–
8. Autoryzacja i zarządzanie sesją	737	551	751
9. Iframes i zagnieżdżone konteksty	508	505	657
10. Canvas i Drag & Drop	789	684	957
11. WebSockets i komunikacja real-time	825	459	862
12. Alerty, okna i pop-upy	784	696	894
Suma	6777	5506	6531
LOC/test (pełne zestawy)	12,9	10,6	14,5
LOC/test (bez kategorii 7)	12,7	10,5	14,5

Najniższą łączną objętość kodu odnotowano dla Cypress. Suma dla WebdriverIO jest niższa niż dla Playwright, jednak wartości bezwzględne dotyczą zestawów o różnej liczebności testów, na którą składa się przede wszystkim wykluczona kategoria 7 oraz węższe pokrycie kategorii 5 (por. w podrozdziale 4.2). Miarodajne porównanie wymaga zatem normalizacji do liczby testów. Wartości znormalizowane zestawiono na dole Tabeli 29. Jest tam dostępny wariant dla pełnych zestawów oraz dla pojedynczego testu z wyłączeniem kategorii 7. Niezależnie od wariantu uporządkowanie jest identyczne. Implementacja analogicznych scenariuszy w WebdriverIO wymaga o około 14% więcej kodu niż w Playwright i o około 38% więcej niż w Cypress.

W ujęciu frameworków na kategorię, Cypress osiągnął najmniejszą objętość kodu w 10 z 12 kategorii, co można wiązać ze zwężnością łańcuchowego API poleceń. WebdriverIO wykazał największą objętość w 9 kategoriach, przy czym różnica jest najbardziej widoczna w kategoriach wymagających jawnej synchronizacji i zarządzania kontekstami. Najwyraźniej widać to w kategorii 12, gdzie na dodatkowy kod składa się między innymi ręczne zarządzanie uchwytami okien oraz sekwencje jawnych oczekiwań opisane w podrozdziale 5.12. Odstępstwem od tego wzorca jest kategoria 5, w której WebdriverIO ma najmniej kodu. Wartość ta jest konsekwencją węższego zakresu zaimplementowanych scenariuszy, a nie zwężłość składni. Analogicznie interpretować należy kategorię 1, w której najniższą wartość uzyskał Playwright. Jest to zarazem jedyna kategoria, w której Cypress ma najwięcej kodu, przy identycznej liczbie testów. Porównania frameworków na kategorię należy zatem zawsze interpretować z uwzględnieniem różnic w pokryciu testowym.

Rozkład objętości kodu w poszczególnych kategoriach przedstawiono na Rysunku 7.



Rysunek 7. Liczba linii kodu z podziałem na kategorie testowe. Źródło: opracowanie własne.

6.4. Zużycie zasobów systemowych

Zużycie zasobów mierzono zgodnie z metodyką opisaną w podrozdziale 4.3.4. Monitor próbował z częstotliwością 1 Hz zużycie pamięci RAM i obciążenie procesora przez drzewo procesów uczestniczących w wykonywaniu testów, obejmujące proces *test runner*, sterowniki oraz procesy przeglądarki. Rejestrował również liczebność tego drzewa. Wartości CPU wyrażono jako odsetek łącznej mocy obliczeniowej procesora (12 wątków logicznych). Dla każdej kombinacji agregowano wartości średnie oraz szczytowe z poszczególnych uruchomień. Monitor nie obejmował pełnego drzewa procesów przeglądarek uruchamianych przez zewnętrzne sterowniki (ograniczenie opisane w podrozdziale 4.4). Z tego powodu wartości należy traktować jako dolne oszacowania rzeczywistego zużycia, a porównania interpretować przede wszystkim w ujęciu względnym.

Wyniki dla trybu sekwencyjnego zestawiono w Tabeli 30. Najoszczędniej gospodaruje pamięcią Playwright, który utrzymuje przy tym najmniejszą liczbę procesów. WebdriverIO zajmuje

pod tym względem pozycję pośrednią. Najwięcej pamięci potrzebuje Cypress – w Chrome około 2,6 razy więcej niż w Playwright. Wynik ten dobrze koresponduje z architekturą frameworka, który utrzymuje w pamięci aplikację Electron, osadzającą *test runner* oraz testowaną aplikację. We wszystkich analizowanych frameworkach widoczny jest również wyraźny wpływ silnika przeglądarki. W Firefox średnie zużycie pamięci jest większe względem Chrome o 25% w Cypress, o 45% w WebdriverIO i aż o 117% w Playwright. Najmniejsze zużycie pamięci w całym analizowanym zbiorze odnotowano dla silnika WebKit.

Tabela 30. Zużycie zasobów systemowych w trybie sekwencyjnym.
Źródło: opracowanie własne.

Kombinacja	Śr. RAM [MB]	Szczyt RAM – mediana [MB]	Śr. CPU [%]	Szczyt CPU – śr. [%]	Procesy – śr. szczyt
Playwright Chrome	571	843	2,9	13,4	9,0
Playwright Firefox	1236	1597	5,9	17,3	12,0
Playwright Edge	836	1021	3,7	16,7	13,0
Playwright WebKit	557	805	3,6	16,3	9,0
Cypress Chrome	1468	1798	4,4	19,9	17,4
Cypress Firefox	1840	2124	5,1	19,6	17,1
Cypress Edge	1492	1822	4,4	16,8	16,7
WebdriverIO Chrome	713	1180	1,9	14,2	14,5
WebdriverIO Firefox	1034	1538	3,5	25,7	16,6
WebdriverIO Edge	946	1373	2,5	22,2	18,1

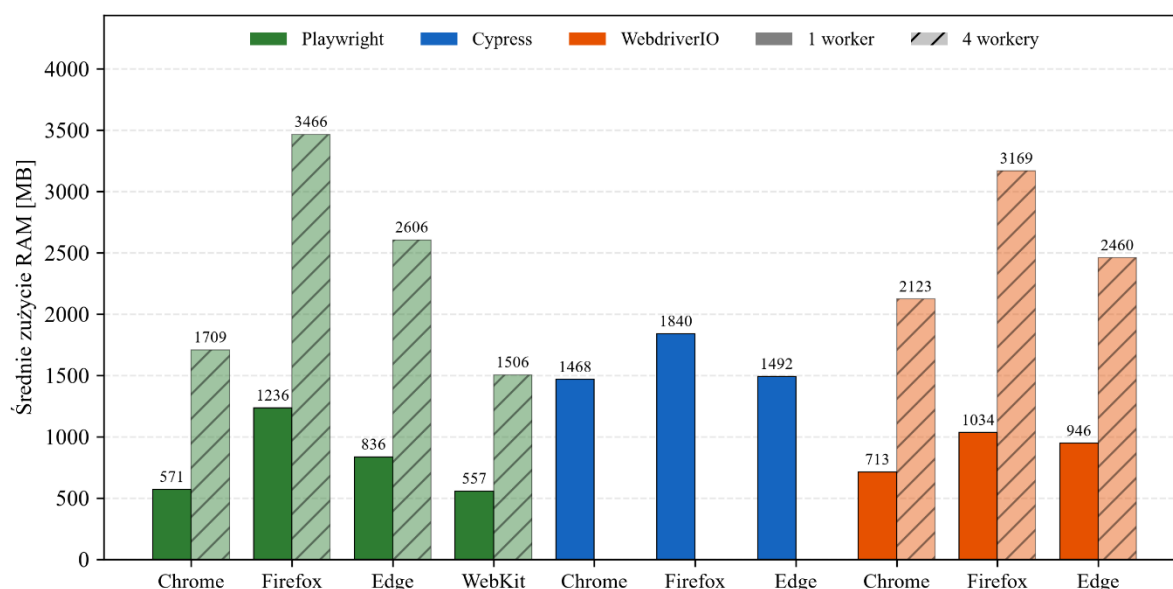
W wartościach szczytowych RAM pojawiły się pojedyncze obserwacje odstające. W jednym z uruchomień Cypress + Chrome szczytowe zużycie ponad trzykrotnie przekroczyło medianę szczytów tej kombinacji, sięgając około 6 GB, a dwa uruchomienia WebdriverIO + Edge wyraźnie odbiegły od pozostałych. Z tego powodu w Tabeli 30 i Tabeli 31 jako miarę szczytową RAM raportowana jest mediana szczytów (mniej wrażliwa na obserwacje odstające), natomiast dla CPU i liczby procesów – średnia szczytów z uruchomień. Średnie obciążenie procesora w trybie sekwencyjnym było niskie i w ujęciu ciągłym nie przekraczało jednego wątku logicznego. Potwierdza to, że tempo sekwencyjnego wykonania testów *E2E* ogranicza oczekiwanie na aplikację i przeglądarkę, a nie moc obliczeniowa. Zwraca przy tym uwagę systematycznie niższe raportowane obciążenie CPU dla WebdriverIO. Wartość tę należy częściowo przypisać wspomnianemu ograniczeniu pokrycia drzewa procesów przez monitor.

Przejsięcie na 4 procesy robocze zwiększyło średnie zużycie pamięci od 2,6 do 3,1 razy (Tabela 31), a więc mniej niż czterokrotnie. Wskazuje to, że część zasobów jest współdzielona między procesami. Najwyższe wartości odnotowano dla kombinacji z Firefox, których mediany szczytów przekroczyły 4 GB. Nawet w najbardziej wymagającej konfiguracji zużycie nie przekroczyło jednak około 14% pamięci dostępnej w środowisku badawczym. Średnie obciążenie procesora dla Playwright wzrosło do poziomu odpowiadającego około 1,2–1,6 wątku logicznego, z wyraźnie wyższymi wartościami szczytowymi. Wartości zużycia CPU raportowane dla WebdriverIO pozostały natomiast na poziomie trybu sekwencyjnego. Wobec osiągniętego przyspieszenia około 3,7× ponownie należy je uznać za wynik ograniczenia monitora zasobów, a nie rzeczywistą charakterystykę frameworka.

Potwierdzają to pozostałe metryki tych samych kombinacji, które skalują się zgodnie z oczekiwaniami. Porównanie zużycia pamięci między trybami przedstawiono na Rysunku 8.

Tabela 31. Zużycie zasobów systemowych w trybie równoległym (4 procesy robocze, średnie z 5 uruchomień). Źródło: opracowanie własne.

Kombinacja	Śr. RAM [MB]	Szczyt RAM – mediana [MB]	Śr. CPU [%]	Szczyt CPU – śr. [%]	Procesy – śr. szczyt
Playwright Chrome	1709	2342	9,7	49,5	25,4
Playwright Firefox	3466	4135	13,1	55,7	35,4
Playwright Edge	2606	2935	12,0	65,7	40,2
Playwright WebKit	1506	2091	13,0	48,3	26,0
WebdriverIO Chrome	2123	3036	2,6	14,8	41,6
WebdriverIO Firefox	3169	4390	1,6	27,4	55,6
WebdriverIO Edge	2460	3563	2,2	16,5	49,8



Rysunek 8. Średnie zużycie pamięci RAM w trybie sekwencyjnym i równoległym. Źródło: opracowanie własne.

6.5. Raportowanie

Zdolności raportowania oceniono jakościowo. Podstawą oceny były doświadczenia z implementacji scenariuszy (rozdział 5) oraz kampanii pomiarowej (podrozdział 4.4), w której wszystkie trzy frameworki raportowały wyniki do wspólnego formatu Allure, pełniącego rolę wspólnego mianownika porównania. Konfiguracje natywnych mechanizmów raportowania zestawiono w podrozdziale 3.4. Dalsza część podrozdziału koncentruje się na praktycznej ocenie poszczególnych aspektów diagnostycznych.

Najbogatszy natywny zestaw narzędzi diagnostycznych oferuje Playwright. Generowany raport HTML [45] łączy wyniki testów ze zrzutami ekranu, nagraniami wideo [48] oraz zapisem *trace* [50]. Ten ostatni odtwarza stan drzewa DOM, żądania sieciowe oraz logi konsoli po zakończeniu testu. W kampanii pomiarowej mechanizmy te nie wymagały instalowania dodatkowych zależności. Cypress automatycznie wykonuje zrzuty ekranu przy niepowodzeniach oraz nagrywa wideo z przebiegu testów [49]. Jego interaktywny tryb graficzny udostępnia podgląd migawek DOM przed i po wykonaniu każdego polecenia (mechanizm *time travel*). Raportowanie zbiorcze opiera się na reporterach zgodnych z Mocha [46], a rozszerzona analityka historyczna dostępna jest w płatnej usłudze chmurowej Cypress Cloud [29]. WebdriverIO w konfiguracji domyślnej ogranicza się do tekstowego reportera *spec* [47]. Zrzuty ekranu, nagrania i raporty HTML wymagają instalacji pakietów z ekosystemu wtyczek. Wykorzystana w pomiarach integracja z Allure zapewniła porównywalny zakres raportowanych wyników, jednak wymagała najwięcej konfiguracji spośród trzech analizowanych frameworków.

Syntetyczną ocenę przedstawiono w Tabeli 32. Zastosowano w niej trzystopniową skalę:

- „natywne” (funkcja wbudowana w framework dostępna bez dodatkowej konfiguracji),
- „wtyczka” (wymaga dodatkowego pakietu i konfiguracji),
- „–” (brak wsparcia).

Warto dodać, że wyniki diagnostyczne mają zauważalny koszt dyskowy. W kampanii pomiarowej pojedyncze uruchomienie generowało około 50 MB danych Allure, co przy 135 uruchomieniach złożyło się na kilkugigabajtowy wolumen danych diagnostycznych.

Tabela 32. Ocena wsparcia aspektów raportowania i diagnostyki.

Źródło: opracowanie własne.

Aspekt	Playwright	Cypress	WebdriverIO
Raport HTML	natywny	wtyczka	wtyczka
Zrzuty ekranu przy niepowodzeniach	natywne	natywne	wtyczka
Nagrania wideo	natywne	natywne	wtyczka
Zapis <i>trace</i>	natywny	natywny	–
<i>time travel</i> / podgląd migawek	natywny	natywny (w trybie interaktywnym)	–
Integracja z Allure	wtyczka	wtyczka	wtyczka
Analityka chmurowa	–	wtyczka	–

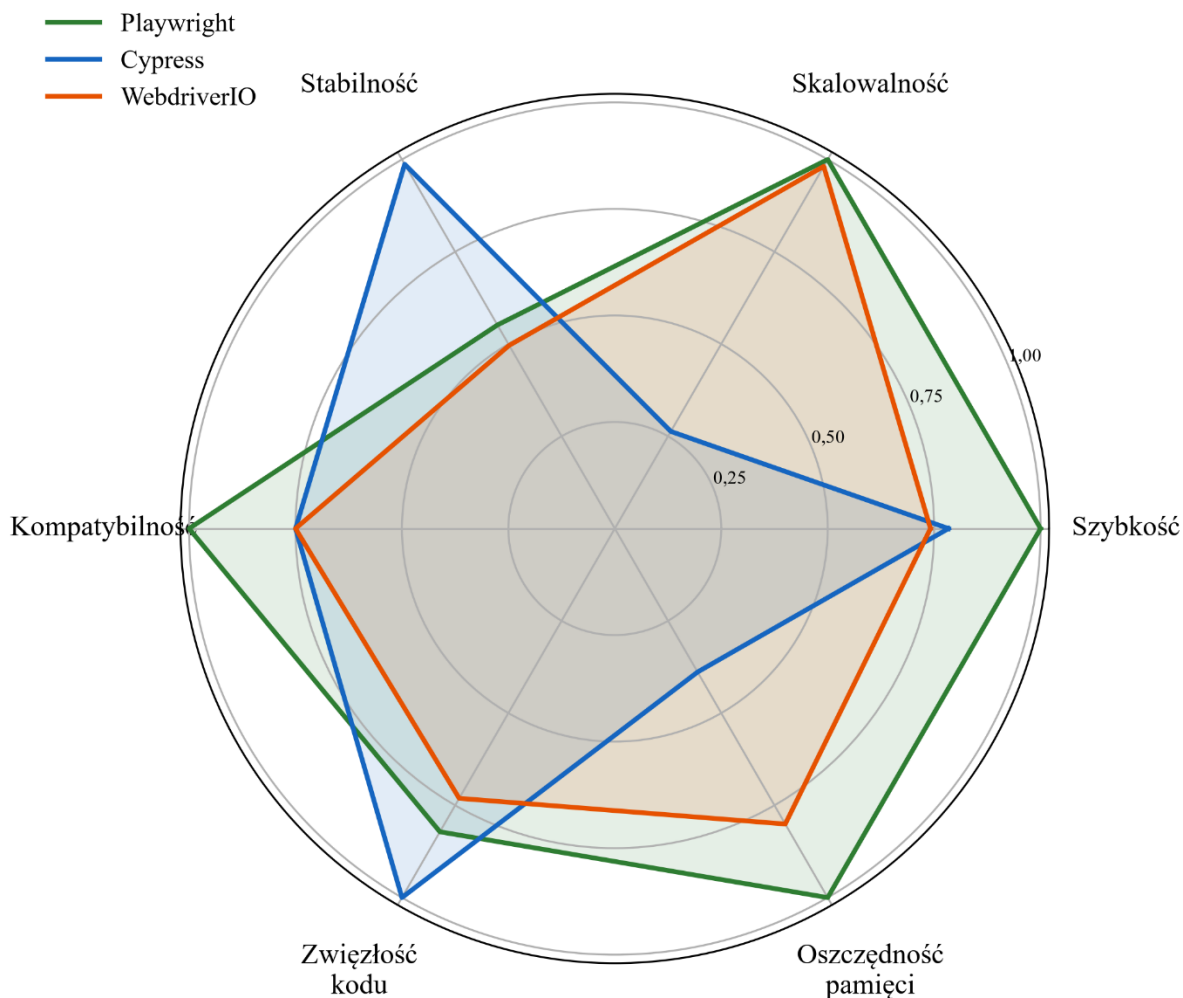
6.6. Macierz porównawcza

Tabela 33 agreguje wyniki metryk opisanych w rozdziale 6, w podziale na poszczególne frameworki. Najlepszy wynik w każdym wierszu wyróżniono pogrubieniem. Dla metryk zależnych od przeglądarki przyjęto konfigurację bazową z przeglądarką Chrome, a wartości dotyczące objętości kodu pochodzą z podrozdziału 6.3.

Tabela 33. Macierz porównawcza prezentująca zbiorcze zestawienie metryk.
Źródło: opracowanie własne.

Metryka	Playwright	Cypress	WebdriverIO
Liczba testów	524	518	450
Śr. czas sekwencyjny (Chrome) [s]	486,6	613,8	563,7
Czas na test (Chrome) [ms]	929	1185	1253
Przyspieszenie przy 4 procesach roboczych (Chrome)	3,80×	brak	3,73×
Narzut Firefox / Edge [%]	+20,6 / +4,9	+8,7 / +0,7	+8,6 / +2,1
Wsparcie WebKit (Windows)	tak (+122,1%)	eksperymentalne	nie
Zakres pass rate, tryb sekwencyjny [%]	91,51–99,98	99,96–100	92,89–100
Testy niestabilne (tryb sekwencyjny)	2	1	2
Testy deterministycznie odrzucane (tryb sekwencyjny)	45	0	32
LOC łącznie / na test	6777 / 12,9	5506 / 10,6	6531 / 14,5
Śr. RAM, Chrome, 1 proces roboczy [MB]	571	1468	713
Liczba kategorii z najkrótszym czasem	7	1	4

Profil każdego frameworka zwizualizowano na Rysunku 9 w postaci wykresu radarowego o sześciu osiach: szybkość wykonania, skalowalność, stabilność, kompatybilność przeglądarkowa, zwiężłość kodu oraz oszczędność pamięci. Wartości znormalizowano liniowo w obrębie każdej osi, tak aby 1 oznaczało najlepszy wynik w badanym zestawie.



Rysunek 9. Profil porównawczy frameworków na wykresie radarowym na podstawie sześciu metryk. Źródło: opracowanie własne.

W ujęciu zbiorczym żaden framework nie dominuje we wszystkich wymiarach. Playwright uzyskał najlepszy wynik w największej liczbie metryk. Prowadzi w szybkości wykonania, skalowalności, oszczędności pamięci oraz kompatybilności – jako jedyny oferuje pełne wsparcie silnika WebKit. Okupuje to jednak najwyższą liczbą testów deterministycznie odrzucanych, skoncentrowaną w skrajnej konfiguracji WebKit. Cypress osiągnął najlepszą stabilność, bez ani jednego testu deterministycznie odrzucanego w jakiegokolwiek kombinacji, a przy tym najniższe narzuty *cross-browser* i najmniejszą objętość kodu. Płaci za to zwiększonym czasem wykonania, najwyższym zużyciem pamięci i brakiem natywnej równoległości lokalnej. WebdriverIO zajmuje pozycję pośrednią pod względem skalowalności i zużycia pamięci. Oferuje niemal stałą efektywność zrównoleglenia i pełną stabilność w przeglądarce Chrome. Dziedziczy jednak ograniczenia sterowników Firefox oraz wymaga największej objętości kodu na test. Interpretację tych profili i wynikające z nich rekomendacje doboru frameworka przedstawiono w rozdziale 7.

7. Interpretacja wyników

W rozdziale 6 przedstawiono wyniki pomiarów w ujęciu ilościowym, celowo powstrzymując się od ich wartościowania. Niniejszy rozdział przechodzi od danych do wniosków. Najpierw zinterpretowano zaobserwowane różnice w kontekście architektury poszczególnych narzędzi, a następnie usystematyzowano ograniczenia każdego z nich i sformułowano rekomendacje doboru frameworka dla różnych typów projektów. Rozdział zamyka omówienie ograniczeń samego badania oraz kierunków jego kontynuacji.

7.1. Kluczowe różnice między frameworkami

Przeprowadzone badanie pozwoliło wskazać sześć obszarów, w których porównywane frameworki różnią się w sposób odczuwalny w codziennej pracy. Należą do nich model synchronizacji z aplikacją, przechwytywanie ruchu sieciowego, obsługa Shadow DOM, praca z wieloma kartami i oknami, równoległe uruchamianie testów oraz zakres wspieranych silników przeglądarek.

Najgłębsza z tych różnic dotyczy synchronizacji, czyli sposobu w jaki narzędzie czeka na gotowość aplikacji. Playwright i Cypress realizują odpowiednio automatyczne oczekiwanie (ang. *auto-wait*) i automatyczne ponawianie (ang. *retry-ability*), więc frameworki samodzielnie czekają na interakcję z elementem. Synchronizacja WebdriverIO jest częściowo automatyczna z dodatkiem jawnego oczekiwania. Podstawowe akcje m.in. kliknięcie czy wpisanie tekstu, mają wbudowane oczekiwanie, jednak przy dynamicznie zmieniającej się stronie, autor testu musi sam dopisać instrukcje czekania na gotowość elementu lub na spełnienie warunku. Przykłady takich konstrukcji przedstawiono w podrozdziale 5.2.

Różnica ta jest widoczna już w objętości kodu. W porównaniu symetrycznym (podrozdział 6.3) test w WebdriverIO zajmuje średnio 14,5 linii, w Playwright 12,7, a w Cypress 10,5. Za znaczną część tej nadwyżki są odpowiedzialne jawne oczekiwania oraz ponowne pobranie kolekcji elementów po zmianach w DOM wpływających na jej zawartość. To drugie wynika z faktu, że WebdriverIO zwraca zmaterializowaną tablicę elementów (ang. *materialized collection*), czyli statyczną migawkę stanu strony, która po przebudowie DOM staje się nieaktualna. Sama liczba linii nie oddaje przy tym pełnego kosztu. Część złożoności została w badanej implementacji przeniesiona do funkcji pomocniczych, między innymi do obsługi okien i alertów w kategorii 12. Funkcje te programista musi samodzielnie zbudować i utrzymywać, podczas gdy Playwright analogiczną warstwę dostarcza natywnie.

Model synchronizacji odbija się także na precyzji wykrywania elementów na stronie. Omówiony w podrozdziale 6.2 test paginacji był deterministycznie odrzucany wyłącznie przez Playwright w przeglądarce Chrome. Okazuje się, że agresywne automatyczne oczekiwanie w połączeniu z ponawianymi asercjami dostrzega krótkotrwałe stany przejściowe interfejsu, które umykają wolniejszym mechanizmom odpytywania Cypress i WebdriverIO. Ten sam test przechodził jednak w trybie równoległym, co potwierdza czasowy charakter zjawiska. Taka czujność jest zaletą diagnostyczną, ale w pracy zespołowej może oznaczać dodatkowy nakład na stabilizację testów.

Najostrzej frameworki dzieli przechwytywanie ruchu sieciowego. Playwright [53], [54] i Cypress [58] mają natywne, stabilne mechanizmy mockowania. Obie implementacje kategorii 7 przechodziły komplet testów we wszystkich przeglądarkach. W WebdriverIO w wersji 9 wbudowany mechanizm mockowania [43] okazał się niefunkcjonalny, przez co całą kategorię 7, liczącą 57 testów,

trzeba było wykluczyć z porównania. Dla zespołów, których strategia testowa opiera się na izolacji od *backend*, jest to różnica dyskwalifikująca, niezależnie od pozostałych metryk.

Jeszcze wyraźniej architektura narzędzi ujawnia się przy komponentach z ukrytą strukturą Shadow DOM. Playwright przenika ją automatycznie i nie wymaga żadnej dodatkowej składni. W Cypress [53] trzeba jawnie zaznaczyć przejście do ukrytego drzewa, a w WebdriverIO [54] używać przeznaczonych do tego selektorów. Ta z pozoru drobna różnica dała mierzalny efekt. W kategorii 3 test w Playwright trwał średnio 144 ms, wobec 245 ms w Cypress i 313 ms w WebdriverIO, a jego implementacja pozostała najprostsza składniowo. Wraz z rosnącą popularnością komponentów webowych (ang. *Web Components*) znaczenie tej różnicy rośnie.

Praca z wieloma kartami i oknami dzieli narzędzia w sposób zero-jedynkowy. Cypress, wykonujący testy wewnątrz przeglądarki, w ogóle nie obsługuje scenariuszy *multi-tab* [27], dlatego w kategorii 12 zastosowano obejście wymuszające nawigację w bieżącej karcie. Playwright otwiera kolejne strony naturalnie, w ramach kontekstów przeglądarki, natomiast WebdriverIO wymaga ręcznego zarządzania uchwytami okien. Ten ręczny nadzór ma swoją cenę, bo implementacja kategorii 12 w WebdriverIO rozrosła się do 894 linii wobec 784 w Playwright i 696 w Cypress.

Osobne miejsce należy się równoległości lokalnej, którą w pełni zmierzono dopiero w niniejszym badaniu. Playwright i WebdriverIO uzyskały przy czterech procesach roboczych przyspieszenie od 3,05 do 3,80 razy bez wzrostu niestabilności (podrozdziały 6.1.2 i 6.2). Dzięki temu pełny cykl regresji, trwający sekwencyjnie od ośmiu do dziesięciu minut, zamyka się w dwie–trzy minuty. Cypress nie oferuje takiej możliwości lokalnie, ponieważ jego paralelizacja wymaga usługi Cypress Cloud [29], [39] i rozdziela testy między osobne maszyny agenta CI. W projektach o dużym wolumenie testów wpływa to na koszt i czas informacji zwrotnej.

Ostatni obszar to zakres środowisk przeglądarkowych. Playwright jako jedyny uruchamia testy na silniku WebKit w systemie Windows, choć wiąże się to z ponad dwukrotnie dłuższym czasem wykonania (+122,1%) oraz ograniczeniami w scenariuszach autoryzacji, które omówiono w podrozdziale 7.2. Cypress i WebdriverIO ograniczają się w praktyce do przeglądarek Chrome, Edge oraz Firefox, a ich zmiana powoduje niewielki narzut, który waha się od +0,7% do +8,7% czasu.

7.2. Ograniczenia poszczególnych narzędzi

Pomiary ujawniły dwa typy systematycznych ograniczeń. Część z nich framework dziedziczy po silniku przeglądarki lub sterowniku, co przejawia się występowaniem testów deterministycznie odrzucanych. Pozostałe wynikają z architektury samego narzędzia. Ograniczenia zestawiono w podziale na frameworki.

Playwright. Najpoważniejszy problem dotyczy kombinacji z silnikiem WebKit, w której 43 testy, czyli 8,2% zestawu, były deterministycznie odrzucane, z czego aż 41 w kategorii 8. Wszystkie kończą się niepowodzeniem asercji zmiany adresu po zalogowaniu, co wskazuje na restrykcyjną politykę WebKit wobec ciasteczek i lokalnego magazynu danych, najsurowszą wśród badanych środowisk przeglądarkowych. Jeśli dodać do tego narzut czasowy +122,1%, WebKit pod Playwright w systemie Windows trudno uznać za konfigurację w pełni produkcyjną dla scenariuszy stanowej autoryzacji. Dla pozostałych 91,8% testów pozostaje on natomiast użyteczny. Zdarzały się też pojedyncze testy *fragile*, związane z konkretną parą framework × przeglądarka. Chodzi o test paginacji w Chrome, który przechodził w Cypress, w WebdriverIO, a nawet w trybie równoległym samego Playwright oraz o test pomiaru czasu żądania w Firefox. Do tego dochodzi podwyższony narzut Firefox,

+20,6% sekwencyjnie i +50,5% równolegle, będący kosztem własnego protokołu automatyzacji. Warto również pamiętać, że Playwright [15] wykonuje testy na własnych, dostosowanych kompilacjach przeglądarek, a nie na wersjach produkcyjnych instalowanych u użytkowników.

Cypress. Jego ograniczenia mają charakter architektoniczny i są udokumentowane przez producenta. Narzędzie nie obsługuje wielu kart i okien ani natywnej równoległości, przez co jako jedyne nie uczestniczyło w pomiarach trybu równoległego. W obrębie pojedynczego testu obowiązują też ograniczenia polityki *same-origin*. Pomiary dodały do tej listy wymierny koszt zasobowy. Cypress zużywał najwięcej pamięci spośród badanych narzędzi, średnio od 1468 MB do 1840 MB, czyli około 2,6 razy więcej niż Playwright w Chrome. Odpowiada za to aplikacja Electron, w której działa *test runner* wraz z testowaną aplikacją. Cypress odnotował także najdłuższe czasy wykonania, sięgające 1185 ms na test w Chrome. Po stronie zalet, pomiary potwierdziły natomiast wyjątkową stabilność, bo jako jedyny z frameworków nie miał ani jednego testu deterministycznie odrzucanego w żadnej kombinacji.

WebdriverIO. Najpoważniejszym ograniczeniem jest niedziałające w wersji 9 mockowanie sieci. Metoda `browser.mock()` kończy się błędem komunikacji z procesem renderującym, co wykluczyło kategorię 7 z porównania. Druga grupa problemów pochodzi ze sterowników oraz polityk przeglądarek i uwidoczniła się w kombinacji z Firefox, gdzie deterministycznie odrzucane były 32 testy, czyli 7,1% zestawu. Złożyły się na to blokada uwierzytelniania HTTP Basic poprzez adres z osadzonymi poświadczeniami, brak obsługi przesyłania plików w sterowniku geckodriver oraz dostępność symulacji warunków sieciowych wyłącznie dla rodziny przeglądarek Chromium [68]. Trzecim ograniczeniem jest praca na zmaterializowanych tablicach elementów, które po zmianach w DOM stają się nieaktualne i wymagają ponownego pobierania kolekcji. Te słabości równoważą atuty potwierdzone pomiarami. WebdriverIO okazał się w pełni stabilny w trybie równoległym w przeglądarkach Chrome i Edge, ponieważ przechodził pełny komplet testów. Framework utrzymał niemal stałą efektywność zrównoleglenia na poziomie 91%–93% i jako jedyny gwarantuje zgodność ze standardem W3C WebDriver, a więc testowanie na produkcyjnych wersjach przeglądarek.

7.3. Rekomendacje doboru frameworka

Zestawienie wyników ilościowych z rozdziału 6 z opisanymi wyżej różnicami architektonicznymi pozwala sformułować rekomendacje dla typowych sytuacji decyzyjnych. Żaden z badanych frameworków nie dominuje we wszystkich wymiarach, dlatego odpowiedź na pytanie o wybór narzędzia zależy od priorytetów projektu.

Projekt wymagający testowania *cross-browser*, w tym Safari. Tutaj wybór sprowadza się do Playwright, bo żaden inny badany framework nie uruchomi testów na silniku WebKit poza systemem macOS. Zespół powinien jednak zaplanować budżet czasowy na ponad dwukrotnie dłuższe wykonania testów w WebKit oraz sprawdzić, czy scenariusze autoryzacji oparte na ciasteczkach mogą być w ogóle zrealizowane w danym środowisku przeglądarkowym dla danej aplikacji.

Projekt intensywnie korzystający z mockowania API. Do tej grupy należą projekty wymagające izolacji od *backend* i testowania stanów błędów oraz aplikacje SPA z rozbudowaną komunikacją sieciową. W tym przypadku należy rozważyć frameworki Playwright lub Cypress, natomiast WebdriverIO w wersji 9 powinien zostać wykluczony do czasu ustabilizowania obsługi protokołu WebDriver BiDi (podrozdział 7.5). Warto wspomnieć, że w siódmej kategorii porównującej

wyłącznie te dwa frameworki, Cypress osiągnął krótszy czas testu, a jego mechanizm przechwytywania żądań okazał się rozwiązaniem dojrzałym i ergonomicznym.

Mały zespół lub startup, dla którego liczy się szybkie wdrożenie. W takim projekcie najlepiej sprawdzi się Cypress. Pozwala napisać ten sam test najmniejszą ilością kodu, a interaktywny tryb z podglądem wykonania i w pełni automatyczna synchronizacja obniżają próg wejścia. Potwierdzona pomiarami stabilność, w tym zero testów deterministycznie odrzucanych, ogranicza koszt utrzymania. Braki tego narzędzia jak *multi-tab*, równoległość lokalna, wsparcie silnika WebKit oraz wyższe zużycie pamięci, rzadko bywają krytyczne w małych projektach webowych. Powinny jednak zostać świadomie zaakceptowane na etapie wyboru narzędzia.

Projekt enterprise z dużym wolumenem testów i infrastrukturą CI/CD. W tym scenariuszu najwięcej argumentów przemawia za Playwright. Wykonuje testy najszybciej w Chrome, a tryb równoległy pozwala zwiększyć szybkość wykonywania testów. Do tego dochodzi najbogatsza natywna diagnostyka z raportem HTML i przeglądarką śladów wykonania (ang. *trace viewer*). Wszystko to wpływa na koszt utrzymania rozbudowanych ciągów testowych (ang. *pipeline*) i czas informacji zwrotnej dla zespołu.

Projekt z istniejącą infrastrukturą Selenium lub wymogiem zgodności ze standardem W3C. Naturalnym kandydatem jest WebDriverIO, jedyny z badanych frameworków oparty na otwartym standardzie protokołu. Dzięki temu można wykorzystać istniejące siatki Selenium Grid i usługi chmurowe zgodne z protokołem WebDriver oraz testować projekt na produkcyjnych wersjach przeglądarek. Pomiary pokazały przy tym konkurencyjną wydajność, w tym drugą pozycję czasową w Chrome, i stabilną efektywność zrównoleglenia. Trzeba jednak doliczyć wyższy koszt utrzymania kodu, wynikający z jawnej synchronizacji, zarządzania referencjami i własnoręcznie utrzymywanej warstwy funkcji pomocniczych, a także pamiętać o braku działającego mockowania sieci w wersji 9.

Omówione zależności podsumowano w Tabeli 34.

Tabela 34. Scenariusze decyzyjne doboru frameworka E2E. Źródło: opracowanie własne.

Scenariusz	Rekomendacja	Kluczowy argument	Główne ryzyko
<i>Cross-browser</i> z Safari	Playwright	jedyne wsparcie WebKit poza systemem macOS	narzut i ograniczenia autoryzacji w WebKit
Intensywne mockowanie API	Playwright / Cypress	natywne, stabilne API przechwytywania	brak
Mały zespół / startup	Cypress	najmniej kodu na test, pełny <i>auto-wait</i> , zero <i>always fail</i>	brak <i>multi-tab</i> i równoległości lokalnej
Enterprise z CI/CD	Playwright	szybkie testy w Chrome, natywna diagnostyka	niestandardowe kompilacje przeglądarek
Infrastruktura Selenium / standard W3C	WebDriverIO	zgodność z protokołemm W3C WebDriver, przeglądarki produkcyjne	brak mockowania sieci, wyższy koszt utrzymania kodu

7.4. Ograniczenia przeprowadzonego badania

Rzetelna interpretacja wyników wymaga wskazania słabych stron samego badania. Zidentyfikowano ich pięć.

Pierwsza wynika z wyboru aplikacji testowej. Wszystkie pomiary przeprowadzono na jednej aplikacji (GAD), uruchamianej lokalnie, co zapewniło pełną kontrolę środowiska i powtarzalność, ale oznacza minimalne opóźnienia sieciowe. W warunkach produkcyjnych, gdzie występują opóźnienia sieciowe, proporcje czasów mogą się zmienić. Przewaga automatycznego oczekiwania może wówczas zyskać na znaczeniu, a bezwzględne różnice czasowe między frameworkami częściowo zniwelować się przez opóźnienia związane z komunikacją siecią. Wyniki nie muszą się też przenosić na aplikacje o innej charakterystyce, np. na ciężkie aplikacje SPA z intensywnym renderowaniem.

Druga dotyczy środowiska. Badanie przeprowadzono lokalnie, w systemie Windows 11, na pojedynczej konfiguracji sprzętowej i poza infrastrukturą CI/CD. Bezwzględne wartości czasów i zasobów są więc specyficzne dla tej maszyny, a przenośne pozostają przede wszystkim relacje między frameworkami. Replikację pomiarów w środowisku CI wskazano jako kierunek dalszych badań w rozdziale 7.5.

Trzecia wiąże się z pomiarami trybu równoległego, które objęły tylko 4 procesy robocze oraz mniejszą próbę, bo 5 uruchomień na kombinację wobec 10 w trybie sekwencyjnym. Zmierzone przyspieszenia opisują więc jeden punkt krzywej skalowania, a nie jej przebieg. Dodatkowo pomiar zużycia zasobów napotkał ograniczenie metodyczne. Pierwotny monitor, budujący drzewo procesów po identyfikatorze procesu nadrzędnego, gubił procesy renderujące przeglądark, które system operacyjny przepinał do innych procesów macierzystych. Z tego powodu przeprojektowano go na porównywanie migawek listy procesów (podrozdział 4.4). Mimo to wartości obciążenia procesora dla WebdriverIO pozostały zaniżone (podrozdział 6.4), więc wszystkie wartości zasobowe należy traktować jako dolne oszacowania, a porównania CPU między frameworkami interpretować ostrożnie.

Czwarta słabość to pominięcie czynnika ludzkiego, w tym krzywej uczenia się i ergonomii pracy zespołowej, a także koszty licencji usług towarzyszących np. Cypress Cloud. Ocena raportowania (podrozdział 6.5) ma charakter jakościowy i pokazuje perspektywę tylko jednego autora. Wiąże się z tym pokrewny problem. Wszystkie trzy zestawy testów zaimplementowała ta sama osoba, co eliminuje zmienność stylu między frameworkami, ale oznacza, że nierówna biegłość autora w poszczególnych narzędziach mogła wpłynąć na jakość implementacji.

Ostatnie zastrzeżenie dotyczy momentu badania. Porównanie pokazuje stan ekosystemu wyznaczony przez konkretne wersje frameworków, szczególnie w kontekście wykluczenia kategorii 7 dla WebdriverIO. To ograniczenie jest właściwością bieżącej wersji frameworka i może zniknąć wraz z rozwojem protokołu WebDriver BiDi. Liczba testów różni się także między frameworkami, co skorygowano poprzez normalizację czasu do wykonania pojedynczego testu. Porównując jednak bezwzględne czasy wykonania całych zestawów, należy uwzględnić różnice w ich liczebności.

7.5. Kierunki dalszych badań

Rozwój protokołu WebDriver BiDi. Najpoważniejsze ograniczenie WebdriverIO zidentyfikowane w badaniu, czyli niedziałające mockowanie sieci, wynika z wykorzystania trybu WebDriver Classic, a nie docelowego kształtu ekosystemu. Standard WebDriver BiDi [37] definiuje dwukierunkowy kanał komunikacji z przeglądarką i przewiduje zestandaryzowane polecenia

przechwytywania żądań (**`network.addIntercept`**) oraz emulacji warunków sieciowych, które w protokole klasycznym wymagały niestandardowych rozszerzeń CDP. W wersji 9 badanego frameworka implementacja tego trybu okazała się jednak zbyt niestabilna do zastosowania w kampanii pomiarowej, co opisano w podrozdziale 3.3.3. Naturalnym rozwinięciem pracy byłoby zatem powtórzenie porównania po ustabilizowaniu obsługi BiDi w WebdriverIO oraz w sterownikach przeglądarek. Pozwoliłoby to przywrócić pełną symetrię zestawów testowych dzięki włączeniu 57 testów kategorii 7. Umożliwiłoby to także zweryfikować, czy nowy protokół eliminuje pozostałe niedogodności zaobserwowane w trybie klasycznym, w tym jawne łańcuchy przejść przez granice Shadow DOM (podrozdział 5.3) oraz ograniczenie emulacji trybu offline do przeglądarek Chrome oraz Edge (podrozdział 5.11). Osobnym pytaniem badawczym pozostaje narzut wydajnościowy dwukierunkowej komunikacji, który można byłoby zmierzyć tą samą procedurą pomiarową, porównując czasy wykonania identycznych scenariuszy w obu trybach.

Charakterystyka skalowania równoległości. Pomiary z czterema procesami roboczymi wykazały wysoką efektywność zrównoleglenia, od 76 do 95%, jednakże opisują pojedynczy punkt krzywej skalowania. Serie z 2, 6 i 8 procesami pozwoliłyby wyznaczyć punkt nasycenia, ograniczony liczbą rdzeni, pamięcią i przepustowością aplikacji testowej. O tym, że taki punkt istnieje, pośrednio świadczy fakt, że kombinacje z Firefox już przy czterech procesach roboczych przekraczały 4 GB pamięci. Można byłoby też sprawdzić, czy słabsza skalowalność pary Playwright + Firefox pogłębia się wraz z liczbą procesów. Osobnym wątkiem pozostaje paralelizacja Cypress w jej natywnym modelu, czyli w rozproszeniu na wiele maszyn agenta CI, której nie da się wprost porównać z równoległością lokalną.

Replikacja w środowisku CI/CD. Powtórzenie kampanii pomiarowej w infrastrukturze ciągłej integracji, z kontenerami Linux i współdzielonymi zasobami, zweryfikowałoby przenośność wyników poza lokalne środowisko Windows. Pozwoliłoby też zbadać, jak ograniczenia zasobowe wpływają na *flakiness*, który w izolowanym środowisku okazał się pomijalny i wyniósł 0,01%.

Rozszerzenie zakresu narzędzi i aplikacji. Porównanie można poszerzyć o kolejne narzędzia, m.in. Selenium, TestCafe czy Puppeteer oraz o aplikacje o odmiennej charakterystyce – od ciężkich aplikacji SPA po strony generowane serwerowo. Uzupełnieniem byłoby kontrolowane wprowadzanie opóźnień sieciowych (ang. *throttling*), które pokazałoby, jak przewagi mechanizmów synchronizacji zachowują się w warunkach zbliżonych do produkcyjnych.

Czynnik ludzki. Metryki techniczne warto byłoby uzupełnić badaniem z udziałem programistów. Badanie mogłoby obejmować pomiar krzywej uczenia się, subiektywną ocenę ergonomii oraz czas diagnozowania celowo wprowadzonych defektów w każdym z frameworków. Pozwoliłoby to zobiektywizować aspekty, które w niniejszej pracy oceniono jakościowo z perspektywy jednego autora.

8. Podsumowanie

Efektem pracy jest analiza porównawcza trzech frameworków do automatyzacji testów *End-to-End* aplikacji internetowych: Playwright, Cypress oraz WebdriverIO. Sformułowane wnioski odnoszą się do wyboru narzędzia pod konkretne wymagania projektu.

Na potrzeby badania zaimplementowano analogiczne zestawy testów w każdym z narzędzi, obejmujące dwanaście kategorii tematycznych, od podstawowych interakcji z elementami strony po scenariusze zaawansowane. Testy wykonywano wielokrotnie na lokalnie uruchamianej aplikacji GAD, mierząc ich czas wykonania, stabilność, objętość kodu, wsparcie raportowania oraz zużycie zasobów systemowych.

Wyniki wykazały, że żadne z badanych narzędzi nie osiąga najlepszych rezultatów we wszystkich analizowanych kryteriach. Playwright zaoferował najszerzy zakres możliwości i najlepiej poradził sobie ze scenariuszami zaawansowanymi. Cypress pozwolił napisać najzwęższy kod oraz wyróżnił się największym powodzeniem przeprowadzonych testów, jednak jego ograniczenia architektoniczne zawęziły zakres zastosowań. WebdriverIO okazał się rozwiązaniem najbliższym standardom sieciowym, ale wymagał największego nakładu pracy i nie obsłużył jednej z kategorii. Wszystkie narzędzia zapewniły przy tym wysoką stabilność testów.

Wyniki zebrano w macierzy porównawczej oraz przedstawiono pięć scenariuszy decyzyjnych, które wiążą priorytety projektu z konkretnym narzędziem.

Do ograniczeń pracy należą przeprowadzenie badania na jednej maszynie i jednej aplikacji testowej, a także brak biegłości autora w posługiwaniu się narzędziami. Ponadto limitacją badania jest wykorzystanie ówczesnych wersji frameworków, których dostępne funkcjonalności wpłynęły na uzyskane wyniki.

Pracę można rozwinąć w przyszłości poprzez powtórzenie pomiarów w infrastrukturze ciągłej integracji, rozszerzenie badania o kolejne narzędzia oraz przeprowadzenie ocen z udziałem większej grupy testerów.

Bibliografia

- [1] P. Vorbach, „npm-stat”. Dostęp: 2 kwietnia 2026. [Online]. Dostępne na: <https://npm-stat.com/>
- [2] jaktestowac, *GAD Github*. Dostęp: 3 maja 2026. [Online]. Dostępne na: <https://github.com/jaktestowac/gad-gui-api-demo>
- [3] „International Software Testing Qualifications Board (ISTQB)”. Dostęp: 2 lipca 2026. [Online]. Dostępne na: <https://istqb.org/>
- [4] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, 2009. ISBN: 978-0-321-57936-2.
- [5] H. Vocke, „The Practical Test Pyramid”. Dostęp: 3 lipca 2026. [Online]. Dostępne na: <https://martinfowler.com/articles/practical-test-pyramid.html>
- [6] M. Trzaska, *Modelowanie i implementacja systemów informatycznych 2.0*. 2025. Stron 487. ISBN 978-83-976442-0-5.
- [7] Q. Luo, F. Hariri, L. Eloussi, i D. Marinov, „An Empirical Analysis of Flaky Tests”, zaprezentowano na Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), ACM, lis. 2014, s. 643–653. doi: 10.1145/2635868.2635920.
- [8] J. Humble i D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. ISBN 978-0-321-60191-9.
- [9] M. Moń i B. Pańczyk, „A comparative analysis of web application test automation tools”, *Journal of Computer Sciences Institute*, t. 35, s. 159–165, cze. 2025, doi: 10.35784/jcsi.7119.
- [10] A. Tymoshchuk, „The Evolution of Test Automation: From Selenium to Playwright. A Comparison of Automation Tools: Selenium vs. Playwright vs. Cypress”, *International Journal of Computer*, t. 54, s. 37–45, kwi. 2025.
- [11] P. Gosik i M. Miłosz, „Comparative analysis of Cypress and Playwright frameworks in end-to-end testing for applications based on Angular”, *Journal of Computer Sciences Institute*, t. 36, s. 320–327, wrz. 2025, doi: 10.35784/jcsi.7662.
- [12] A. Pasik, „Analiza narzędzi służących do wykonywania testów wydajnościowych aplikacji internetowych stworzonych w różnorodnych technologiach”, PJATK, Warszawa, 2020.
- [13] N. Chotza, „Automatyczne testowanie aplikacji webowych w podejściu End-to-End: Przegląd technologii i studium przypadku”, PJATK, Warszawa, 2025.
- [14] „Documentation | Playwright”. Dostęp: 4 kwietnia 2026. [Online]. Dostępne na: <https://playwright.dev/docs/intro>
- [15] „Browsers | Playwright”. Dostęp: 5 kwietnia 2026. [Online]. Dostępne na: <https://playwright.dev/docs/browsers>
- [16] „Auto-waiting | Playwright”. Dostęp: 7 kwietnia 2026. [Online]. Dostępne na: <https://playwright.dev/docs/actionability>
- [17] „Timeouts | Playwright”. Dostęp: 5 maja 2026. [Online]. Dostępne na: <https://playwright.dev/docs/test-timeouts>
- [18] „Assertions | Playwright”. Dostęp: 5 maja 2026. [Online]. Dostępne na: <https://playwright.dev/docs/test-assertions>
- [19] „Locators | Playwright”. Dostęp: 8 kwietnia 2026. [Online]. Dostępne na: <https://playwright.dev/docs/locators>
- [20] „Documentation | Cypress”. Dostęp: 8 maja 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/get-started/why-cypress>

- [21] „WebKit (Experimental) | Cypress”. Dostęp: 8 maja 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/references/launching-browsers#WebKit-Experimental>
- [22] „Architecture | Cypress”. Dostęp: 9 maja 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/get-started/why-cypress#Architecture>
- [23] „Command Queue | Cypress”. Dostęp: 9 maja 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/core-concepts/introduction-to-cypress#The-Cypress-Command-Queue>
- [24] M. Fowler, „Fluent Interface”, martinowler.com. Dostęp: 5 lipca 2026. [Online]. Dostępne na: <https://martinfowler.com/bliki/FluentInterface.html>
- [25] „Retry-ability | Cypress”. Dostęp: 11 maja 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/core-concepts/retry-ability>
- [26] „Variables and aliases | Cypress”. Dostęp: 12 maja 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/core-concepts/variables-and-aliases>
- [27] „Trade-offs | Cypress”. Dostęp: 10 kwietnia 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/references/trade-offs>
- [28] „Migrate from Playwright to Cypress”. Dostęp: 13 maja 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/guides/migration/playwright-to-cypress>
- [29] „Cypress Cloud”. Dostęp: 14 maja 2026. [Online]. Dostępne na: <https://docs.cypress.io/cloud/get-started/introduction>
- [30] „Documentation | WebdriverIO”. Dostęp: 19 maja 2026. [Online]. Dostępne na: <https://webdriver.io/docs/gettingstarted/>
- [31] „WebdriverIO v9 | WebdriverIO”. Dostęp: 22 maja 2026. [Online]. Dostępne na: <https://webdriver.io/blog/2024/08/15/webdriverio-v9-release/>
- [32] „Appium Setup | WebdriverIO”. Dostęp: 23 maja 2026. [Online]. Dostępne na: <https://webdriver.io/docs/appium/>
- [33] „WebDriver”. Dostęp: 18 maja 2026. [Online]. Dostępne na: <https://www.w3.org/TR/webdriver2/>
- [34] „Setup Types | WebdriverIO”. Dostęp: 24 maja 2026. [Online]. Dostępne na: <https://webdriver.io/docs/setuptypes/>
- [35] „Automation Protocols | WebdriverIO”. Dostęp: 24 maja 2026. [Online]. Dostępne na: <https://webdriver.io/docs/automationProtocols/>
- [36] „Auto-waiting | WebdriverIO”. Dostęp: 20 maja 2026. [Online]. Dostępne na: <https://webdriver.io/docs/await/>
- [37] „WebDriver BiDi”. Dostęp: 23 maja 2026. [Online]. Dostępne na: <https://www.w3.org/TR/webdriver-bidi/>
- [38] „Parallelism | Playwright”. Dostęp: 12 kwietnia 2026. [Online]. Dostępne na: <https://playwright.dev/docs/test-parallel>
- [39] „Parallelize tests | Cypress Cloud”. Dostęp: 12 kwietnia 2026. [Online]. Dostępne na: <https://docs.cypress.io/cloud/features/smart-orchestration/parallelization>
- [40] „Parallelization | WebdriverIO”. Dostęp: 14 kwietnia 2026. [Online]. Dostępne na: <https://webdriver.io/docs/organizingSuites/>
- [41] jaktestowac, „GAD README”, GitHub. Dostęp: 26 maja 2026. [Online]. Dostępne na: <https://github.com/jaktestowac/gad-gui-api-demo/blob/main/README.pl.md>
- [42] „jaktestowac.pl”, jaktestowac.pl. Dostęp: 27 maja 2026. [Online]. Dostępne na: <https://jaktestowac.pl/>

- [43] „Browser mock | WebdriverIO”. Dostęp: 15 kwietnia 2026. [Online]. Dostępne na: <https://webdriver.io/docs/api/browser/mock/>
- [44] AIDanial, *cloc*. Perl. Dostęp: 15 kwietnia 2026. [Online]. Dostępne na: <https://github.com/AIDanial/cloc>
- [45] „Reporters | Playwright”. Dostęp: 16 kwietnia 2026. [Online]. Dostępne na: <https://playwright.dev/docs/test-reporters>
- [46] „Reporters | Cypress”. Dostęp: 18 kwietnia 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/tooling/reporters>
- [47] „Spec Reporter | WebdriverIO”. Dostęp: 19 kwietnia 2026. [Online]. Dostępne na: <https://webdriver.io/docs/spec-reporter/>
- [48] „Videos | Playwright”. Dostęp: 21 kwietnia 2026. [Online]. Dostępne na: <https://playwright.dev/docs/videos>
- [49] „Screenshots and Videos | Cypress”. Dostęp: 22 kwietnia 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/guides/screenshots-and-videos>
- [50] „Trace viewer | Playwright”. Dostęp: 24 kwietnia 2026. [Online]. Dostępne na: <https://playwright.dev/docs/trace-viewer>
- [51] „Page object models | Playwright”. Dostęp: 25 kwietnia 2026. [Online]. Dostępne na: <https://playwright.dev/docs/pom>
- [52] „State of JavaScript 2025”. Dostęp: 27 kwietnia 2026. [Online]. Dostępne na: <https://2025.stateofjs.com/en-US/usage/>
- [53] „shadow() | Cypress”. Dostęp: 10 czerwca 2026. [Online]. Dostępne na: <https://docs.cypress.io/api/commands/shadow>
- [54] „shadow\$ | WebdriverIO”. Dostęp: 11 czerwca 2026. [Online]. Dostępne na: [https://webdriver.io/docs/api/element/shadow\\$/](https://webdriver.io/docs/api/element/shadow$/)
- [55] „Downloads | Playwright”. Dostęp: 8 lipca 2026. [Online]. Dostępne na: <https://playwright.dev/docs/downloads>
- [56] „Network | Playwright”. Dostęp: 9 lipca 2026. [Online]. Dostępne na: <https://playwright.dev/docs/network>
- [57] „Mock APIs | Playwright”. Dostęp: 9 lipca 2026. [Online]. Dostępne na: <https://playwright.dev/docs/mock>
- [58] „intercept() | Cypress”. Dostęp: 10 lipca 2026. [Online]. Dostępne na: <https://docs.cypress.io/api/commands/intercept>
- [59] „Test isolation | Cypress”. Dostęp: 11 lipca 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/core-concepts/test-isolation>
- [60] „Authentication | Playwright”. Dostęp: 12 lipca 2026. [Online]. Dostępne na: <https://playwright.dev/docs/auth>
- [61] „session() | Cypress”. Dostęp: 14 lipca 2026. [Online]. Dostępne na: <https://docs.cypress.io/api/commands/session>
- [62] „Dialogs | Playwright”. Dostęp: 15 lipca 2026. [Online]. Dostępne na: <https://playwright.dev/docs/dialogs>
- [63] „Frames | Playwright”. Dostęp: 16 lipca 2026. [Online]. Dostępne na: <https://playwright.dev/docs/frames>
- [64] „Frequently asked questions | Cypress”. Dostęp: 18 lipca 2026. [Online]. Dostępne na: <https://docs.cypress.io/app/faq>

- [65] „switchFrame | WebdriverIO”. Dostęp: 19 lipca 2026. [Online]. Dostępne na:
<https://webdriver.io/docs/api/browser/switchFrame/>
- [66] „trigger() | Cypress”. Dostęp: 21 lipca 2026. [Online]. Dostępne na:
<https://docs.cypress.io/api/commands/trigger>
- [67] „Animations | Chart.js”. Dostęp: 22 lipca 2026. [Online]. Dostępne na:
<https://www.chartjs.org/docs/latest/configuration/animations.html>
- [68] „Chromium | WebdriverIO”. Dostęp: 24 lipca 2026. [Online]. Dostępne na:
<https://webdriver.io/docs/api/chromium/>
- [69] „Pages | Playwright”. Dostęp: 25 lipca 2026. [Online]. Dostępne na:
<https://playwright.dev/docs/pages>

Wykaz rysunków

Rysunek 1. Dynamika pobrań rocznych narzędzi <i>E2E</i> z serwisu npm w latach 2022–2025.....	20
Rysunek 2. Średni czas wykonania zestawu testów w trybie sekwencyjnym	62
Rysunek 3. Rozkład czasów wykonania w poszczególnych uruchomieniach (tryb sekwencyjny)	63
Rysunek 4. Średni czas pojedynczego testu w podziale na kategorie testowe (Chrome, tryb sekwencyjny)	65
Rysunek 5. Przyspieszenie wykonania przy 4 procesach roboczych względem trybu sekwencyjnego	66
Rysunek 6. Względny narzut czasu wykonania przy zmianie przeglądarki (tryb sekwencyjny, Chrome = 0%).....	68
Rysunek 7. Liczba linii kodu z podziałem na kategorie testowe	72
Rysunek 8. Średnie zużycie pamięci RAM w trybie sekwencyjnym i równoległym	74
Rysunek 9. Profil porównawczy frameworków na wykresie radarowym na podstawie sześciu metryk	77

Wykaz tabel

Tabela 1. Porównanie konfiguracji i ekosystemu trzech frameworków <i>E2E</i> (stan na maj 2026)	17
Tabela 2. Specyfikacja środowiska testowego	18
Tabela 3. Różnice w liczbie przygotowanych testów między frameworkami	20
Tabela 4. Kategorie testowe – uzasadnienie różnic i liczba testów	21
Tabela 5. Porównanie natywnych mechanizmów raportowania	23
Tabela 6. Konwencja kolorowania składni stosowana w listingach kodu	27
Tabela 7. Mechanizmy selekcji i indeksowania elementów w pierwszej kategorii testowej	28
Tabela 8. Mechanizmy oczekiwania w drugiej kategorii testowej	30
Tabela 9. Obsługa trudnych selektorów w trzeciej kategorii testowej	33
Tabela 10. Obsługa formularzy i walidacji w czwartej kategorii testowej	36
Tabela 11. Weryfikacja pobierania plików w piątej kategorii testowej	38
Tabela 12. Obsługa tabel, paginacji i nieskończonego przewijania w szóstej kategorii testowej	41
Tabela 13. Mechanizmy przechwytywania żądań i mockowania API w siódmej kategorii testowej ...	42
Tabela 14. Obsługa autoryzacji i sesji w analizowanych frameworkach	45
Tabela 15. Porównanie obsługi ramek w trzech frameworkach	49
Tabela 16. Obsługa Canvas i Drag & Drop w analizowanych frameworkach	53
Tabela 17. Obsługa komunikacji WebSocket w jedenastej kategorii testowej	56
Tabela 18. Obsługa alertów, okien i pop-upów w analizowanych frameworkach	59
Tabela 19. Zakres pomiarowy	61
Tabela 20. Czas wykonania testów w trybie sekwencyjnym (1 proces roboczy, 10 uruchomień na kombinację)	62
Tabela 21. Normalizacja czasu pojedynczego testu w trybie sekwencyjnym	63
Tabela 22. Średni czas pojedynczego testu w podziale na kategorie (Chrome, tryb sekwencyjny)	64
Tabela 23. Czas wykonania testów w trybie równoległym (4 procesy robocze, 5 uruchomień na kombinację)	65
Tabela 24. Względny narzut czasu wykonania przy zmianie przeglądarki (tryb sekwencyjny, Chrome = 0%)	67
Tabela 25. Testy niestabilne zidentyfikowane w trybie sekwencyjnym	68
Tabela 26. Klasyfikacja stabilności testów per kombinacja (tryb sekwencyjny)	69
Tabela 27. Klasyfikacja przyczyn testów deterministycznie odrzucanych (tryb sekwencyjny)	69
Tabela 28. Klasyfikacja stabilności testów per kombinacja (tryb równoległy)	70
Tabela 29. Objętość kodu [LOC] w podziale na kategorie testowe	71
Tabela 30. Zużycie zasobów systemowych w trybie sekwencyjnym	73
Tabela 31. Zużycie zasobów systemowych w trybie równoległym (4 procesy robocze, średnie z 5 uruchomień)	74
Tabela 32. Ocena wsparcia aspektów raportowania i diagnostyki	75
Tabela 33. Macierz porównawcza prezentująca zbiorcze zestawienie metryk	76

Tabela 34. Scenariusze decyzyjne doboru frameworka <i>E2E</i>	81
---	----

Wykaz listingów kodu

Listing 1. Lokalizacja i kliknięcie elementu w Playwright.....	27
Listing 2. Lokalizacja i kliknięcie elementu w Cypress.	28
Listing 3. Lokalizacja i kliknięcie elementu w WebdriverIO.	28
Listing 4. Ponowne pobranie tablicy elementów po modyfikacji w DOM w WebdriverIO.....	28
Listing 5. Interakcja z opóźnionym elementem w Playwright.....	29
Listing 6. Interakcja z opóźnionym elementem w Cypress.	29
Listing 7. Interakcja z opóźnionym elementem w WebdriverIO.	29
Listing 8. Oczekiwanie na zniknięcie wskaźnika ładowania w Playwright.....	30
Listing 9. Oczekiwanie na zniknięcie wskaźnika ładowania w Cypress.	30
Listing 10. Oczekiwanie na zniknięcie wskaźnika ładowania w WebdriverIO.	30
Listing 11. Selekcja wiersza po treści przy dynamicznych ID w Playwright.	31
Listing 12. Selekcja wiersza po treści przy dynamicznych ID w Cypress.....	31
Listing 13. Selekcja wiersza po treści przy dynamicznych ID w WebdriverIO.	31
Listing 14. Dostęp do Shadow DOM w Playwright.	32
Listing 15. Dostęp do Shadow DOM w Cypress.	32
Listing 16. Dostęp do Shadow DOM w WebdriverIO.....	32
Listing 17. Dostęp do zagnieżdżonego Shadow DOM w Cypress.	32
Listing 18. Dostęp do zagnieżdżonego Shadow DOM w WebdriverIO.	32
Listing 19. Wprowadzenie wartości do pola w Playwright.	33
Listing 20. Wprowadzenie wartości do pola w Cypress.	33
Listing 21. Wprowadzenie wartości do pola w WebdriverIO.....	34
Listing 22. Wyzwolenie zdarzenia <i>blur</i> w Playwright.....	34
Listing 23. Wyzwolenie zdarzenia <i>blur</i> w Cypress.	34
Listing 24. Wyzwolenie zdarzenia <i>blur</i> przez kliknięcie poza polem w WebdriverIO.	34
Listing 25. Weryfikacja stanu walidacji w Playwright.	34
Listing 26. Weryfikacja stanu walidacji w Cypress.....	34
Listing 27. Jawne oczekiwanie na zmianę klasy walidacji w WebdriverIO.....	35
Listing 28. Weryfikacja natywnych atrybutów walidacji HTML5 w Playwright.....	35
Listing 29. Weryfikacja natywnych atrybutów walidacji HTML5 w Cypress.	35
Listing 30. Weryfikacja natywnych atrybutów walidacji HTML5 w WebdriverIO.....	35
Listing 31. Przesłanie pojedynczego pliku w Playwright.	35
Listing 32. Przesłanie pojedynczego pliku w Cypress.....	35
Listing 33. Przesłanie pojedynczego pliku w WebdriverIO.	36
Listing 34. Obejście przesyłania wielu plików przez wstrzyknięcie <i>DataTransfer</i> w WebdriverIO. ...	36
Listing 35. Pobranie i weryfikacja treści pliku w Playwright.....	37
Listing 36. Pośrednia weryfikacja pobrania przez przechwycenie API w Cypress.	37

Listing 37. Weryfikacja stanu interfejsu po pobraniu w WebdriverIO.....	38
Listing 38. Zbioreczy odczyt kolumny w Playwright.....	39
Listing 39. Odczyt kolumny w Cypress.....	39
Listing 40. Odczyt kolumny w WebdriverIO.	39
Listing 41. Synchronizacja zmiany strony w Playwright.....	39
Listing 42. Synchronizacja zmiany strony aliasem żądania w Cypress.....	39
Listing 43. Synchronizacja zmiany strony w WebdriverIO.....	40
Listing 44. Nieskończone przewijanie w Playwright.....	40
Listing 45. Nieskończone przewijanie w Cypress.	40
Listing 46. Nieskończone przewijanie w WebdriverIO.....	40
Listing 47. Mockowanie odpowiedzi API ze statusem 200 w Playwright.....	41
Listing 48. Mockowanie odpowiedzi API ze statusem 200 w Cypress.	42
Listing 49. Pobranie i weryfikacja ciasteczka z tokenem w Playwright.	43
Listing 50. Pobranie i weryfikacja ciasteczka z tokenem w Cypress.....	43
Listing 51. Pobranie i weryfikacja ciasteczka z tokenem w WebdriverIO.	43
Listing 52. Odczyt stanu sesji metodą storageState() w Playwright.	44
Listing 53. Buforowanie sesji prymitywem cy.session() w Cypress.	44
Listing 54. Obsługa natywnego dialogu prompt() w Playwright.	45
Listing 55. Podmiana funkcji prompt zaślepką w Cypress.	45
Listing 56. Obsługa natywnego dialogu przez most CDP/Puppeteer w WebdriverIO.	45
Listing 57. Dostęp do elementu w ramce poziomym pierwszego w Playwright.	46
Listing 58. Dostęp do elementu w ramce poziomym pierwszego w Cypress.....	47
Listing 59. Dostęp do elementu w ramce poziomym pierwszego w WebdriverIO.	47
Listing 60. Dostęp do ramki zagnieżdżonej (poziom drugi) w Playwright.	47
Listing 61. Dostęp do ramki zagnieżdżonej (poziom drugi) w Cypress.	47
Listing 62. Dostęp do ramki zagnieżdżonej (poziom drugi) w WebdriverIO.....	47
Listing 63. Powrót do strony głównej w WebdriverIO.....	48
Listing 64. Selekcja zagnieżdżonej ramki po atrybucie data-testid w Playwright.....	48
Listing 65. Selekcja ramki bez identyfikatora przez pozycję w Cypress.....	48
Listing 66. Selekcja zagnieżdżonej ramki bez identyfikatora w WebdriverIO.....	49
Listing 67. Wysłanie syntetycznego zdarzenia dragenter w Playwright.....	50
Listing 68. Wysłanie syntetycznego zdarzenia dragenter w Cypress.	50
Listing 69. Wysłanie syntetycznego zdarzenia dragenter w WebdriverIO.....	50
Listing 70. Przesłanie pliku przez ukryty element input w Playwright.	50
Listing 71. Przesłanie pliku przez ukryty element input w Cypress.	51
Listing 72. Ręczna budowa DataTransfer i przesłanie pliku w WebdriverIO.	51
Listing 73. Odczyt etykiet wykresu z instancji Chart.js w Playwright.	51

Listing 74. Odczyt etykiet wykresu z instancji Chart.js w Cypress.....	52
Listing 75. Odczyt etykiet wykresu z instancji Chart.js w WebdriverIO.	52
Listing 76. Reaktywne oczekiwanie na zakończenie animacji wykresu w Playwright.	52
Listing 77. Sterowanie klawiaturą i weryfikacja stanu gry w Playwright.	52
Listing 78. Sterowanie klawiaturą i weryfikacja stanu gry w Cypress.	53
Listing 79. Sterowanie klawiaturą i weryfikacja stanu gry w WebdriverIO.....	53
Listing 80. Nasłuchiwanie ramek połączenia WebSocket w Playwright.....	54
Listing 81. Wysyłanie i weryfikacja wiadomości w interfejsie w Cypress.	54
Listing 82. Wysyłanie wiadomości i ustabilizowanie połączenia w WebdriverIO.....	54
Listing 83. Symulacja dwóch klientów w odrębnych kontekstach w Playwright.....	55
Listing 84. Obejście jednorzytkownikowe z dostępem do okna przez cy.window() w Cypress.....	55
Listing 85. Symulacja dwóch klientów w odrębnych kartach w WebdriverIO.	55
Listing 86. Wymuszenie trybu offline w Playwright.	55
Listing 87. Odzworowanie rozłączenia przeładowaniem i nawigacją wstecz w Cypress.....	56
Listing 88. Wymuszenie trybu offline przez CDP w WebdriverIO.....	56
Listing 89. Przechwycenie i akceptacja natywnego dialogu w Playwright.	57
Listing 90. Przechwycenie treści alertu przez zaślepkę w Cypress.	57
Listing 91. Obsługa natywnego dialogu metodami sterownika w WebdriverIO.....	57
Listing 92. Przechwycenie dodatkowego okna jako obiektu Page w Playwright.....	58
Listing 93. Obejście braku obsługi wielu kart w Cypress.....	58
Listing 94. Ręczne zarządzanie uchwytami okien w WebdriverIO.....	58
Listing 95. Obsługa modala DOM w Playwright.	59
Listing 96. Obsługa modala DOM w Cypress.	59
Listing 97. Obsługa modala DOM w WebdriverIO.....	59