Automation Testing ALM – Project Summary

1. Project Overview

This project is a **custom-built Automation Testing Application Lifecycle Management (ALM)** system designed from scratch with a strong focus on **control, scalability, and transparency**.

Unlike traditional frameworks that tightly couple execution logic, locators, and reporting, this ALM deliberately **separates responsibilities** and gives **maximum control to the tester** while remaining **execution-engine agnostic**.

The current implementation uses **Playwright (JavaScript/Node.js)** as the execution engine, with a clear path to integrate **BrowserStack, Sauce Labs, and native Playwright Cloud** in parallel execution mode.

---

2. Core Design Principles

1. **Tester-first control**
2. No forced locator engine
3. No opinionated action abstraction
4. Testers decide *what* to do with a locator
5. **Stateless execution**
6. Page objects, URLs, and test data are not stored locally
7. Everything is fetched dynamically via HTTP APIs
8. **Feature-level execution**
9. One feature = one scenario
10. One feature = one execution unit
11. One feature = one log file
12. **Parallel-safe by design**
13. No shared mutable state
14. Each feature execution has its own isolated runtime context
15. **ALM-ready architecture**
16. Logs and execution metadata are structured JSON
17. Easy ingestion into future ALM dashboards

---

3. High-Level Architecture

3A. Framework Concepts Used

This framework intentionally applies a small set of **core engineering concepts**, kept explicit and understandable. Below are the key concepts and how they are used.

1. Feature-as-a-Unit Concept

Definition: A feature file represents a single, complete execution unit.

How it is used: - One feature = one scenario - One feature = one runtime context - One feature = one log file

Why: - Simplifies parallel execution - Avoids cross-scenario state leaks - Makes reporting and ALM ingestion trivial

---

2. Runtime Context Pattern

Definition: A short-lived, per-feature execution container that holds everything needed during execution.

Contains: - Feature metadata (name, wording) - Execution timestamps (start/end) - Environment & application - API-fetched URLs - API-fetched locators - API-fetched test data - Playwright `page`

Why: - No globals - Parallel-safe - Clear lifecycle: created → used → destroyed

---

3. Tag-Driven Configuration

Definition: Feature tags act as declarative configuration, not logic.

Examples: - `@env='QA'` - `@App='AeriaLink'` - `@data={'AeriaLink.MCH.Niche.00002'}`

How it is used: - Parsed once in `Before` hook - Converted into runtime context - Never re-read inside steps

Why: - Business-readable configuration - Zero code changes across environments

---

4. Externalized State (API-First Design)

Definition: All mutable test assets live outside the framework.

Externalized Assets: - Locators - URLs - Test data

Why: - No redeploys for locator changes - Centralized governance - Enables ALM ownership of test assets

---

5. Tester-Controlled Actions (No Locator Engine)

Definition: The framework does not decide how locators are used.

What the framework does: - Fetches locators - Exposes them as plain strings

What the tester does: - Chooses Playwright APIs - Chooses validations - Chooses flow control

Why: - Maximum flexibility - No abstraction leaks - Easy debugging

---

6. Explicit Step Execution

Definition: Steps are simple functions with explicit intent.

No: - Auto-retries - Implicit waits hidden by the framework - Magic wrappers

Why: - Predictable behavior - Transparent failures - Easier onboarding for experienced testers

---

7. Feature-Level Logging Concept

Definition: Logging is scoped to a feature execution, not the whole run.

What is logged: - Feature start/end - Step actions and results - Errors with stack trace

Why: - Parallel-safe - Easy ALM ingestion - Simple troubleshooting

---

8. Time-Based Execution Identity

Definition: Each feature execution is uniquely identified by a timestamp.

Example:

```
Login.feature.20260129T183045.json
```

Why: - No file collisions - Traceable executions - Supports retries and history

---

9. Playwright-Native Execution

Definition: The framework does not fight Playwright.

Practices: - Uses Playwright lifecycle hooks correctly - Avoids internal APIs - Respects worker-based parallelism

Why: - Stability - Forward compatibility

---

10. ALM-Ready Data Contracts

Definition: Logs and execution metadata are structured as data, not text.

Why: - Direct ingestion into dashboards - Analytics-ready - Supports trend analysis and traceability

---

3B. Architectural Outcome

By applying these concepts, the framework achieves:

- Strong separation of concerns
- High parallel reliability
- Minimal framework code
- Maximum tester autonomy

This makes the system suitable not only as a test framework, but as a **long-term execution backbone for an ALM platform**.

---

4. Dynamic Data Model (No Local Storage)

```
┌──────────────┐
│ Feature File│   (Intent)
└─────┬────────┘
      │
┌─────▼──────────────┐
│ Step Definitions│   (Tester-controlled actions)
└─────┬──────────────┘
      │
┌─────▼──────────────┐
│ Runtime Context │   (Page, Locators, URLs, Data)
└─────┬──────────────┘
      │
┌─────▼──────────────┐
│ Execution + Logs│   (Feature-level reporting)
└────────────────────┘
```

---

4. Dynamic Data Model (No Local Storage)

4.1 Locators

Locators are fetched dynamically at runtime:

- **Endpoint**

```
GET /api/exports/locators/{Application}
```

- **Example Response**

```json
{
  "application": "AerialLink",
  "count": 4,
  "locators": {
    "BTN-Login.default.Landing": "//button[text()='Login']",
    "INP-Email.default.Login": "//input[@id='signInFormUsername']"
  }
}
```

The framework **does not interpret or wrap locators**. They are passed as-is to the tester.

4.2 Application URLs

Environment-specific URLs are fetched dynamically:

```
GET /service/url/{Application}/{Environment}
```

This allows the same test to run across QA, UAT, and PROD without code changes.

4.3 Test Data

Test data is fetched using logical tags:

```
GET /data/{Application}.{Domain}.{Module}.{RecordId}
```

Supports multiple data records (comma-separated IDs) for data-driven execution in the future.

---

5. Runtime Execution Model

Each feature execution creates its own **Feature Runtime Context**, which includes:

- Feature name
- Start time
- End time
- Execution status (PASSED / FAILED)
- Step-level execution details
- Browser runtime (Playwright page)
- Dynamic locators, URLs, and test data

This context is **never shared** across parallel executions.

---

6. Step Execution Philosophy

- Steps are **explicit function calls**
- No hidden magic, no interception of Playwright internals
- Logging happens **at step boundaries**, not inside the framework

Example Step

```
Navigate to Login Page
```

- Tester explicitly decides:
- Which locator to use
- Which Playwright API to call
- What validation to perform

This ensures full flexibility while still capturing execution metadata.

---

7. Logging & Reporting Strategy

7.1 Feature-Level Logs

- One log file per feature

- Naming convention:

  ```
  <FeatureName>.<RunTimestamp>.json
  ```

- Parallel-safe (no file collisions)

7.2 Logged Metadata

Each log captures:

- Feature name
- Start time / End time
- Total execution duration
- Overall status
- Step-wise details:
- Step text
- Start & end time
- Duration
- Status
- Error message (if failed)

---

8. Parallel Execution Support

The framework is designed to work seamlessly with:

- Playwright workers
- BrowserStack parallel sessions
- Sauce Labs parallel sessions
- Native Playwright Cloud

Parallelism is achieved **without any special handling** because:

- Each feature has its own runtime context
- Logs are isolated per execution
- No global mutable state is shared

---

9. Error Handling & Stability Improvements

During development, issues were encountered due to:

- Attempting to hook into Playwright internal APIs
- Assuming runtime hooks in `playwright-bdd`

These were resolved by:

- Removing all internal hooks
- Avoiding overrides of `Given / When / Then`
- Implementing a clean, explicit execution wrapper

This resulted in a **more stable, predictable, and maintainable system**.

---

10. Current State vs Future ALM Integration

Current State

- JSON-based execution logs
- Local file storage
- CLI-based execution

Future Roadmap

- Push logs to ALM backend
- Execution dashboards
- Trend analysis
- Flakiness detection
- Retry orchestration
- Test history & traceability

---

11. Key Takeaway

This ALM is **not just a test framework**.

It is a **foundational execution platform** designed to:

- Scale to enterprise needs
- Support millions of executions
- Remain flexible for testers
- Integrate cleanly with future ALM tooling

The current implementation deliberately favors **clarity and control over abstraction**, making it robust, debuggable, and ALM-ready from day one.

Status: Stable and extensible **Execution Engine:** Playwright (current) **ALM Integration:** Planned (next phase)