Automation Strategy

1. **Overall Automation Approach:** The preferred strategy would develop a combination of API and UI testing to ensure comprehensive coverage of system functionalities.

   Developing a unified codebase for API and UI will ensure reusable utilities across test types and less maintenance.

   For UI testing, efficiency would be increased as test perquisites or entry criteria can be replaced with API utilities which are faster and less flaky.

   Setting up CI pipelines would be less complex as one setup can be used to trigger both UI and API tests (preferably as separate jobs) - This will ensure high reliability.

2. **Test Pyramid (API vs UI):** As per the definition of Test pyramid, we have unit testing at the base, integration tests in middle, E2E tests at the top

   1. Since E2E simulates the user workflow, we can have UI automation tests here. However, due to flaky nature of UI tests, these tests should be less in number.

   2. Integration tests can be mostly API tests, since integration tests involve more than one module, tests could be time consuming, having these tests as API test cases will ensure less execution time. These API tests can also be employed for performance testing.

      e.g., Integration between Alerts and Dashboards can be written as an API test under integration test suite.

   3. Unit test can be used for quick validation of individual components.

3. **What to Automate at API and UI Level:**

   - **API Level:** Business logic, data validation & consistency in workflows, and integration between different components or their microservices.
   - **UI Level:** User workflows (E2E), responsiveness on various devices (mobile, desktop and tablets), and cross browser testing (chrome, edge, and safari).

4. **What Should Not Be Automated and Why:** Complex cognitive tasks that require human judgment, like visual design assessment should not be automated. E.g., graphs derived from metrics.

   Additionally, rarely occurring edge case scenarios may not justify the effort of automation (Ad hoc test cases).

5. **Handling Shared Environment and Flakiness:**

   Docker setups can be used to ensure isolation between testing teams in a shared environment.

   Using CI pipelines, required environment can be created before test commence and can be torn down, this will ensure uniqueness of environment.

   Documentation and defining guidelines with schedules can help prevent environment being used in parallel between multiple teams.

   Adding retry mechanisms can help minimize flakiness. Regular monitoring and analysis of consistent script failures can help overcome flakiness sometimes.

6. **Scalability in Real Product Team:** Employing modular test framework for script development, and integrating test automation into the development workflow within a sprint cycle (ins-sprint automation).

Test Data Strategy

1. **Test Data Challenges with Grafana:** Since Grafana is a visualization tool which mainly involves graphical data sources we may require complex datasets at least for automation scripts.
2. **Deterministic Testing Approach:** Using predefined dataset inputs that yield predictable outputs for consistency across test cycle. This is crucial for performance benchmarking and regression testing.
3. **Data Seeding, Isolation, and Cleanup Principles:**
   - Data Seeding: Prepopulate databases with known data for testing.
   - Isolation: Using dedicated test environments where we have complete control of what data is created and destroyed.
   - Cleanup: Automate the removal of test data to maintain a clean slate for next test cycle.
4. **Strategy for Shared and Controlled/Local Environments:**
   - For shared environments, employ strict data access controls.
   - In controlled/local environments, we have more flexibility of creating and destroying datasets, it is recommended to include data driven testing in these environments.