Strategic Vision: Multi-Layered Observable QA Model

This architecture blends **Test Strategy Maturity** with **Operational Observability**. Here's the framework

🧩 Layered Execution Strategy

Level	Description	Example Tools	Frequency
L0	Developer-local unit + component tests	Jest, Mocha, pytest	On commit
L1	Quick smoke/integration test pipelines	Cypress, Playwright	On push / PR
L2	Full platform automation (UI/API/Mobile)	Appium, REST-assured	Nightly
L3	System-level regression/perf/security	Fortify, ZAP, k6, Artillery	Weekly / release
L4	Manual exploratory + NFRs	Human QA + heuristics	Release cycle

Tests are **promoted/demoted** between levels based on execution time, flakiness, and coverage return. Manual test cycles are treated as **curated exploration**, not redundancy.

Tooling Recommendations & Cost-Security Enhancements

Current Tool	Issue / Risk	Recommendation
BrowserStack	Security concerns, scalability	Evaluate SauceLabs, LambdaTest, or HeadSpin (better session control + enterprise-grade audit trails)
Cypress/Appium	Excellent but siloed	Adopt shared test orchestration layer (e.g., TestContainers, Docker Grid, or Sauce Testrunner Toolkit)
GitHub Actions	Central, good	Add AI-based test prioritization plugin (e.g., Launchable) to reduce CI noise

Current Tool	Issue / Risk	Recommendation
SonarQube/Fortify	Good base	Integrate directly into CI gates with enforced break conditions and dashboards
Datadog/Grafana	Performance & observability	Expand to QA dashboards (test success rate, flake index, slowest tests, test debt)

Scaling with AI & Empowering Manual QA Teams

Most QA orgs fail to scale automation because they treat it like a tool, not a culture. My approach:

- **Upskill Manual QAs** into **"AI-assisted explorers"**—pair them with AI tools like **Testim**, **Mabl**, or **Autify** for visual test generation.
- Use **GenAI for test case suggestions**, edge scenario discovery, and root cause analysis summaries.
- Implement **shared intelligence models** (model-based testing) where automation is self-healing and shared across teams.

This reduces dependency on hard-coding and promotes **modular, maintainable test assets**.

Next Steps If I'm Brought Onboard

1. Audit current pipeline → identify flake-prone, slow ROI areas.
2. Create pilot multi-layer test structure across one squad.
3. Implement observability dashboards & define test maturity model.
4. Run quarterly reviews: test coverage ROI, velocity impact, bug-to-prod reduction.

This isn't about "adding more testing." It's about **amplifying confidence without slowing teams down**, reducing quality risk at scale, and shifting your QA practice into a **real-time, insight-driven, AI-augmented organization**.

Let's move quality from a phase to a **business differentiator**.

Multi-Level test architecture explained, layer by layer

🧪 L0 – Developer-local Unit + Component Tests

These are fast, isolated tests run directly on the developer's machine. No networks, no services, just code punching code.

Tool	What It Is	Why It Belongs Here
Jest	JavaScript test framework (used with React, Node.js)	Blazing fast, mock-friendly, runs in-memory, great for frontend units
Mocha	Minimalist JS test framework (Node.js backend use mostly)	Easy to plug in, used in older projects and API test setups
pytest	Python testing framework with plugins and fixtures galore	Flexible, expressive, great for microservices and logic testing

How it works: Devs run tests locally or via pre-commit hooks. If it's broken here, it never even gets to CI. The "bouncer" layer of your test club.

⚙️ L1 – Quick Smoke / Integration Pipelines

These are automated tests that validate basic functionality (smoke tests) and simple interactions (integration) with minimal dependencies. They run early in the CI/CD cycle.

Tool	What It Is	Why It Belongs Here
Cypress	JS-based frontend E2E test runner with real browser automation	Fast, flaky-resistant, CI-friendly, great for UI tests
Playwright	Microsoft's competitor to Cypress, supports multiple browsers + languages	Better for cross-browser and headless testing, great with CI

How it works: These tools simulate user flows or API flows in a lightweight CI environment—GitHub Actions, GitLab CI, etc. They catch things like "Login form is dead" *before* you even merge the PR.

📱 L2 – Full Platform Automation (UI/API/Mobile)

Heavier tests that hit the full stack. Run overnight, on real devices or simulators, across multiple platforms and environments.

Tool	What It Is	Why It Belongs Here
Appium	Mobile UI test framework (Android, iOS)	Cross-platform, works on real devices and emulators
REST-assured	Java DSL for API testing	Powerful, expressive, widely used in backend testing pipelines

How it works: These are usually orchestrated in Docker containers or cloud-based device farms (e.g., BrowserStack, SauceLabs). Slower than smoke tests but more thorough. This is where you catch the "Oops we broke the signup API in Spanish on iOS 14" bugs.

🔐 L3 – System-Level Regression / Performance / Security

Expensive, long-running tests. Often post-merge or pre-release. Focus here is stability, speed, and security under load.

Tool	What It Is	Why It Belongs Here
Fortify	Static application security testing (SAST)	Scans code for vulnerabilities—early detection of big security fails
OWASP ZAP	Dynamic security testing (DAST) tool	Pen-test-like fuzzing on running apps—good for staging environments
k6	Modern performance testing tool	Scriptable load testing, integrated with CI/CD, DevOps friendly
Artillery	Lightweight performance/load testing tool	Node.js-based, quick to spin up for API & load scenarios

How it works: These tests run on environments that mimic production. They're often part of release gates. Here you find "The payment system crashes if 200 users log in at once" kind of bugs. Fun.

🎩 L4 – Manual Exploratory + NFRs

Human-driven tests that explore edge cases, behavior nuances, and non-functional requirements like accessibility, usability, etc.

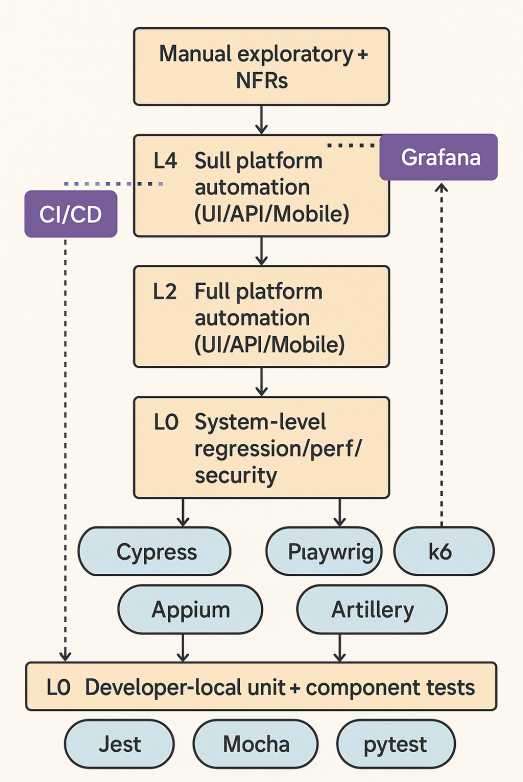
Methodology	What It Is	Why It Belongs Here
Heuristic-based exploratory testing	Skilled testers use experience, personas, and scenarios to break stuff in creative ways	Finds bugs that automation misses: workflow friction, hidden edge cases

Methodology	What It Is	Why It Belongs Here
NFR reviews (e.g. a11y, performance, UX)	Accessibility, internationalization, visual accuracy, etc.	These are best verified by humans, or hybrid AI+human sessions

How it works: Manual testers use environments instrumented with logs/telemetry. These tests provide insights automation simply can't—and often become the basis for new automated scripts.

TYING IT ALL TOGETHER

- All these tools can be tied together via CI tools like **GitHub Actions**, **Jenkins**, or **CircleCI**.
- Tests can generate data for **Grafana dashboards**, **Datadog alerts**, or **Slack reports**.
- You can tie AI tools like **Launchable** to predict which tests matter most per commit, and **Mabl/Testim** to generate low-code automation to help bridge the manual–automated gap.



MULTI-LAYER TEST ARCHITECTURE DEMO

Github repo: https://github.com/josemathias/qa-lab

PREREQUISITES

- **Node.js** (>= 18) and **npm**
- **Python 3** (for pytest)
- **Docker** + **Docker Compose**
- **GitHub account** (for GitHub Actions pipeline)
- Optional: A free **Grafana Cloud** account or local instance
- Optional: Access to **BrowserStack**, **LambdaTest**, or **SauceLabs**

Copy and paste this into your terminal:

```
echo "Checking prerequisites..."

echo -n "Node.js: "; node -v
echo -n "npm: "; npm -v
echo -n "Python 3: "; python3 --version
echo -n "Docker: "; docker --version
echo -n "Git: "; git --version
echo -n "npm packages globally installed: "
npm list -g --depth=0

echo "Done."
```

IF SOMETHING'S MISSING

Here's the installation recipe in case any of those pieces are absent or cursed:

NODE.JS + NPM

Install with Homebrew (recommended):

```
brew install node
```

PYTHON 3

You probably have this already, but if not:

```
brew install python
```

Then install pytest:

```
pip3 install pytest
```

🐋 Docker

If docker --version fails, install Docker Desktop from:

👉 https://www.docker.com/products/docker-desktop/

Then run:

```
docker run hello-world
```

🧙‍♂️ Git

If you don't have Git:

```
brew install git
```

🧪 Create the directory structure

It will look like this:

```
qa-lab/
├── L0-unit-tests/
│   ├── package.json
│   └── tests/
│       └── math.test.js          # Jest example
│
├── L1-integration-tests/
│   ├── cypress/
│   │   └── e2e/sample.cy.js      # Cypress example
│   └── cypress.config.js
│
├── L2-api-mobile-tests/
│   ├── appium/
│   │   └── basic-appium-test.js  # Connects to localhost:4723
│   └── rest-assured/
│       └── basic-api-test.java   # Java REST-assured example
│
├── L3-performance-security/
│   ├── k6/
│   │   └── sample-load-test.js   # k6 test script
│   └── zap/
│       └── config/               # ZAP baseline config (optional)
│
├── .github/
│   └── workflows/
│       └── ci.yml                # GitHub Actions running tests
│
└── README.md                     # The legend of your QA empire
```

1. Create the GitHub repo

Go to GitHub and create a new public repo called qa-lab. Don't initialize it with a README—we've got our own.

Or just run:

```
gh repo create qa-lab --public –confirm
```

2. Scaffold the Directory Locally (you can choose your own naming here)

```
mkdir -p qa-lab/L0-unit-tests/tests
mkdir -p qa-lab/L1-integration-tests/cypress/e2e
mkdir -p qa-lab/L2-api-mobile-tests/appium
mkdir -p qa-lab/L2-api-mobile-tests/rest-assured
mkdir -p qa-lab/L3-performance-security/k6
mkdir -p qa-lab/.github/workflows
```

3. Create placeholder files to avoid Git complaints

Git doesn't track empty directories. You need a dummy file in each folder so Git sees them.

```
find . -type d -empty -exec touch {}/.gitkeep \;
```

4. Setup your GitHub SSH keys in case you haven't already

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

5. Add your public key to GitHub

```
pbcopy < ~/.ssh/id_ed25519.pub

Then go to GitHub → SSH keys → "New SSH Key" → Paste.

ssh -T git@github.com
```

If it says "Hi [username]! You've successfully authenticated," you're golden.

6. Initialize and Push

```
cd qa-lab
git init
git add .
git commit -m "Initial QA Lab structure"
git branch -M main
git remote add origin git@github.com:YOUR_USERNAME/qa-lab.git
git push -u origin main
```

Now, let's get each stage functional and populated

🔧 L0: Developer-local Unit + Component Tests

🛠️ Tools

- Jest (Frontend/React)
- Mocha (Node backend)
- pytest (Python backend)

⚙️ Setup

```
# cd to L0 directory
cd qa-lab/L0-unit-tests/tests

# For Jest
npm init -y
npm install --save-dev jest
npx jest --init

# For Mocha
npm install --save-dev mocha chai

# For pytest
pip install pytest
pytest --maxfail=1 --disable-warnings -q
```

🧪 Add L0 Unit Test (Jest)

File: L0-unit-tests/tests/math.test.js

```
function sum(a, b) {
  return a + b;
}

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

⚙️ L1: Quick Integration Tests (Headless Browser)

🛠️ Tools

- Cypress

⚙️ Setup

```
# cd to L0 directory
cd qa-lab/L1-integration-tests/

npm install cypress --save-dev
```

```
npx cypress open  # Opens the GUI once, creates `cypress/` folder
```

⚙️ Add L1 Integration Test (Cypress)

File: L1-integration-tests/cypress/e2e/sample.cy.js

```
describe('Basic UI Test', () => {
  it('visits Google and checks the title', () => {
    cy.visit('https://google.com');
    cy.title().should('include', 'Google');
  });
});
```

File: L1-integration-tests/cypress.config.js

```
const { defineConfig } = require('cypress');

module.exports = defineConfig({
  e2e: {
    baseUrl: 'https://google.com',
  },
});
```

To run headless (for CI):

```
npx cypress run
```

📱 L2: Full Stack Automation (Mobile, API)

🛠️ Tools

- **Appium** for mobile (use simulators/emulators or Docker)
- **REST-assured** (Java-based)

⚙️ Setup

Appium Setup (Docker-based):

```
# cd to L0 directory
cd qa-lab/L2-api-mobile-tests/appium

docker pull appium/appium
docker run -d -p 4723:4723 appium/appium
```

Use an emulator or install Android Studio to spin up devices. You'll need test scripts using WebdriverIO or Java bindings.

Alternative (easier): Use **BrowserStack Automate** or **LambdaTest** to test mobile without local devices.

📱 Add L2 Appium Test

File: L2-api-mobile-tests/appium/basic-appium-test.js

```
const wd = require("wd");

async function runTest() {
  const driver = wd.promiseRemote("http://localhost:4723/wd/hub");

  const desiredCaps = {
    platformName: "Android",
    deviceName: "emulator-5554",
    browserName: "Chrome"
  };

  await driver.init(desiredCaps);
  await driver.get("https://example.com");
  const title = await driver.title();
  console.log("Page title is:", title);

  await driver.quit();
}

runTest().catch(console.error);
```

🔒 L3: System-level Regression / Perf / Security

🛠️ Tools

- **k6** (for load testing)
- **OWASP ZAP** (DAST scanner)
- **Artillery** (Node-based load test tool)

⚙️ Setup

k6 Setup:

```
# cd to L0 directory
cd qa-lab/L3-performance-security/k6

brew install k6  # or use Docker
k6 run sample-script.js
```

📈 Add L3 Load Test (k6)

File: L3-performance-security/k6/sample-load-test.js

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  vus: 10,
  duration: '10s',
};

export default function () {
  const res = http.get('https://test.k6.io');
  check(res, { 'status is 200': (r) => r.status === 200 });
  sleep(1);
}
```

ZAP:

- Download and run the ZAP GUI
- Point it at your local app or staging environment
- Enable spider + active scan for basic coverage

🧠 L4: Manual + Exploratory + Reporting

Write user personas, test charters, and exploratory notes. Simulate manual testing with tools like:

- **Postman** (for API poking)
- **BugMagnet** (Chrome extension for exploratory payloads)
- **Lighthouse / axe** (for accessibility & performance)

📊 OBSERVABILITY + CI/CD INTEGRATION

🛠️ GitHub Actions (Sample)

.github/workflows/test.yml:

```
name: CI Test

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Install dependencies
        run: npm install
      - name: Run Cypress tests
        run: npx cypress run
```

🔁 Add GitHub Actions Workflow

File: .github/workflows/ci.yml

```yaml
name: Run L0 and L1 Tests

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [18.x]

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}

      - name: Run Unit Tests (Jest)
        working-directory: L0-unit-tests
        run: |
          npm init -y
          npm install --save-dev jest
          echo '{ "scripts": { "test": "jest" } }' > package.json
          npm test

      - name: Run Cypress Tests
        working-directory: L1-integration-tests
        run: |
          npm init -y
          npm install cypress --save-dev
          echo '{ "scripts": { "test": "cypress run" } }' > package.json
          npx cypress install
          npm test
```

ADD README.MD

File: README.md

```
# QA Lab

This repository demonstrates a multi-layered QA testing strategy, including:

- **L0**: Unit tests using Jest
- **L1**: Integration tests using Cypress
- **L2**: Mobile/browser automation using Appium
- **L3**: Load testing using k6
- **CI**: GitHub Actions workflow for L0 and L1 tests
```

```
---

## How to run tests

- Unit tests: `cd L0-unit-tests && npm install && npm test`
- Cypress tests: `cd L1-integration-tests && npm install && npm test`
- Appium test: `cd L2-api-mobile-tests/appium && node basic-appium-test.js`
- k6 test: `cd L3-performance-security/k6 && k6 run sample-load-test.js`
```

🛠️ Grafana Dashboards (optional)

- Set up Grafana + Loki for log ingestion or use k6 Cloud + Grafana plugin
- Or use Allure Reports or MochaAwesome for test result reporting in HTML

🪄 OPTIONAL: WRAP IT UP WITH DOCKER

Put your app + tests + Appium + ZAP + Grafana all into a docker-compose.yml and run it as a mini fake-QA-lab.

🧪 WHAT TO DEMO

7. Show L0 unit test running and failing on commit.
8. Push code to GitHub → trigger GitHub Actions → Cypress test runs.
9. Show load test with k6 + reports.
10. Optionally run ZAP scan against a dummy app.
11. Show test dashboard if you went all-in.

Running the DEMO

✅ Step 1: Check Prerequisites

Copy and paste this into your CLI:

```
echo "Checking prerequisites..."

echo -n "Node.js: "; node -v
echo -n "npm: "; npm -v
echo -n "Python 3: "; python3 --version
echo -n "Docker: "; docker --version
echo -n "Git: "; git --version
echo -n "npm packages globally installed: "
npm list -g --depth=0

echo "Done."
```

This will spit out version info for:

- Node.js
- npm
- Python 3
- Docker
- Git
- and any globally installed npm tools (like cypress, eslint, etc.)

🔁 Git Command Sequence

From your repo root (~/qa-lab/qa-lab):

```
git add .
git commit -m "Add QA test layers, configs, and workflow"
git push origin main
```

✅ Immediate Aftermath: CI Pipeline Runs

💥 GitHub Actions should automatically:

1. Detect the push to main
2. Trigger the Run L0 and L1 Tests workflow
3. Set up Node.js (18.x)
4. Run:
    a. Your **Jest unit test** from L0-unit-tests
    b. Your **Cypress integration test** from L1-integration-tests

If all is right with the world (and your YAML is not possessed), you'll see a green check ✅ next to your latest commit in GitHub.

Go to your repo:

👉 https://github.com/[username]/qa-lab/actions

That's your live workflow dashboard. It'll show:

- Success/failure status
- Logs for each step
- Any embarrassing typos in the test runner output