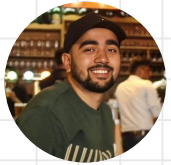
From Autotest to **Avocado**

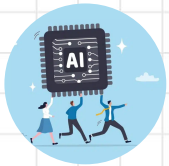
Modernizing System Testing on Ubuntu

Authors Introduction

Nirjal Bhurtel
Engineering @ Yatri Motorcycle | nirjalbhurtel.com.np



Sangam Ghimire
Engineering @ Khalti



AI and LLMS
Engineering @ Everywhere

Motivation

My Introduction to Testing

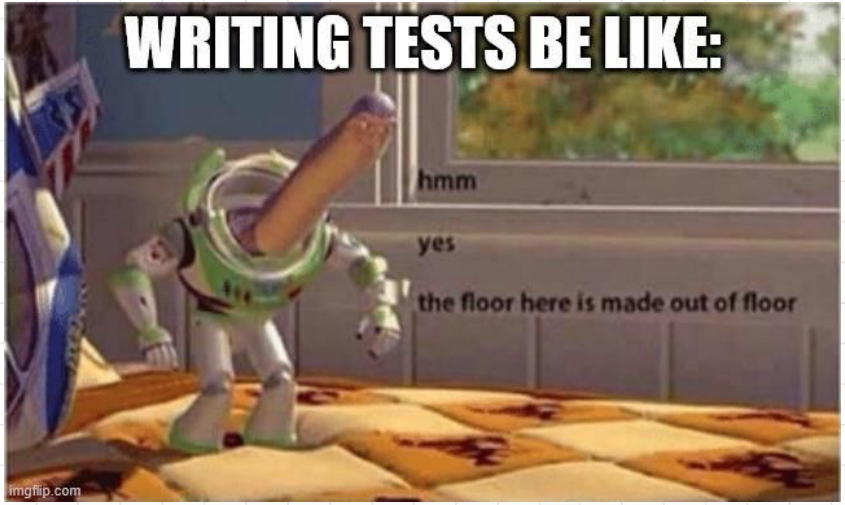
- Started with basic unittest
- Mostly unit and simple integration tests
- Little exposure to system testing

Why I disliked writing tests

- Repetitive code
- Poor structure
- Limited reuse
- Hard to maintain
- Often violated **DRY**

Test code should still be good software

Tests Always Felt Like a burden



1. Create an environment
2. Configure and set it up
3. Make sure every dependency is running
4. Prepare or simulate the required conditions
5. Run the test
6. Debug the environment when the test fails
7. Clean everything up
8. Do it all again for the next test

When Testing Finally Got Interesting

- Joined Intra2Net (I2N) as a **Software Engineering Intern**
- Worked with a large, mature Linux system
- Saw testing at a completely different scale

What caught my attention in QC

- Tests were structured, not just scripts
- Common logic was reused
- Environments and dependencies were managed
- Tests could be parameterized and composed
- Execution and results were handled systematically
- Testing felt like an engineering system of its own

This was the first time I actually enjoyed exploring a testing framework.

The framework behind it?

Avocado 🥑

A Python-based test framework for organizing, executing, parameterizing, and reporting software and system tests.

Avocado-VT

A virtualization-focused extension of Avocado for testing KVM, QEMU, libvirt, and complex virtual machine scenarios.

Avocado-I2N

An Intra2Net-developed extension around Avocado/Avocado-VT for large-scale, reusable system and integration testing across complex Linux environments.

Before We Go Further...

- I first met this ecosystem as an intern
- I’m not an Avocado / Avocado-VT developer
- I haven’t seen every use case or every corner of the framework
- What I did do was use it, explore it, and ask “why is it built this way?”

So this talk is...

- Why we got from Autotest → Avocado
- What changed — and why it matters
- A deeper look at Avocado’s architecture
- The features that make Avocado interesting
- How all of this fits into modern system testing

What Does a System Test Actually Look Like?

1. **Prepare the environment**
   e.g. install packages, create VM/container
2. **Configure the system**
   e.g. copy config files, set kernel/service options
3. **Start required services / machines**
   e.g. start Nginx, boot a VM
4. **Simulate real conditions**
   e.g. send network traffic, generate load, disconnect/reconnect
5. **Run the test**
   e.g. `curl`, benchmark, functional command

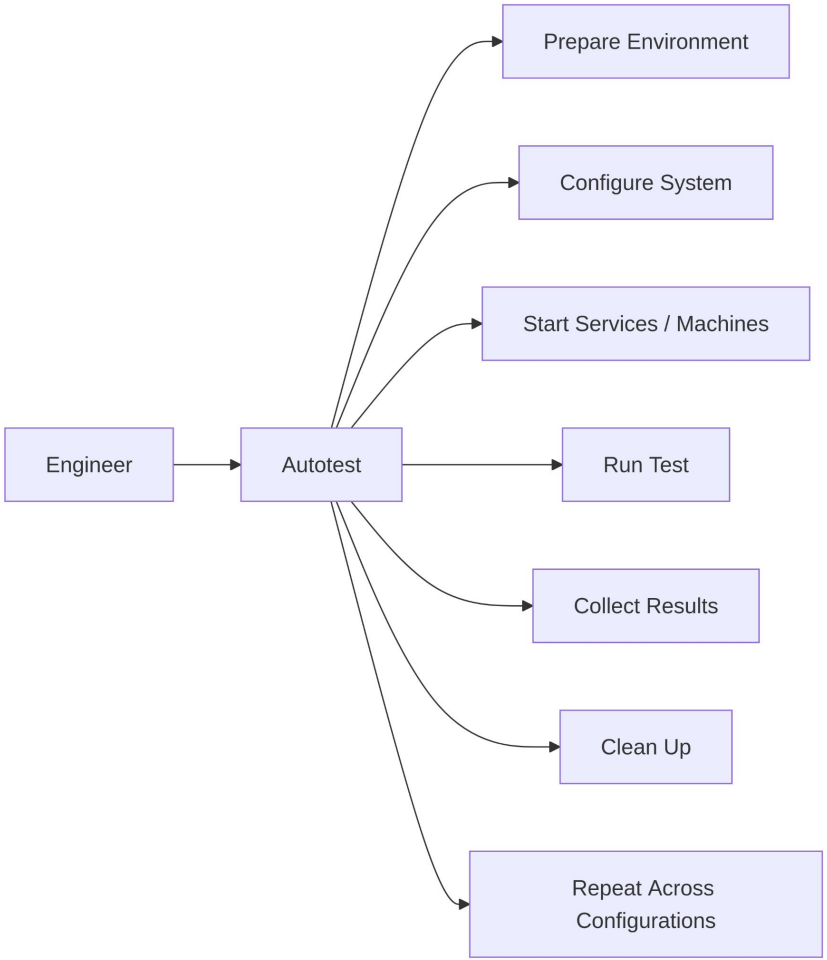
Continued…

6. **Collect logs and results**
   e.g. `journalctl`, test output, performance metrics
7. **Clean up**
   e.g. stop VM, remove temporary files, reset configuration
8. **Repeat across configurations**
   Ubuntu versions • kernels • architectures • networks • storage

The actual test is often the easy part — managing the environment is the hard part.

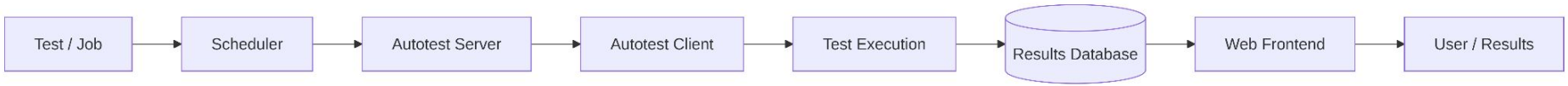
What does Autotest do?

Autotest takes that whole system-testing workflow and **automates the boring parts around the test.**



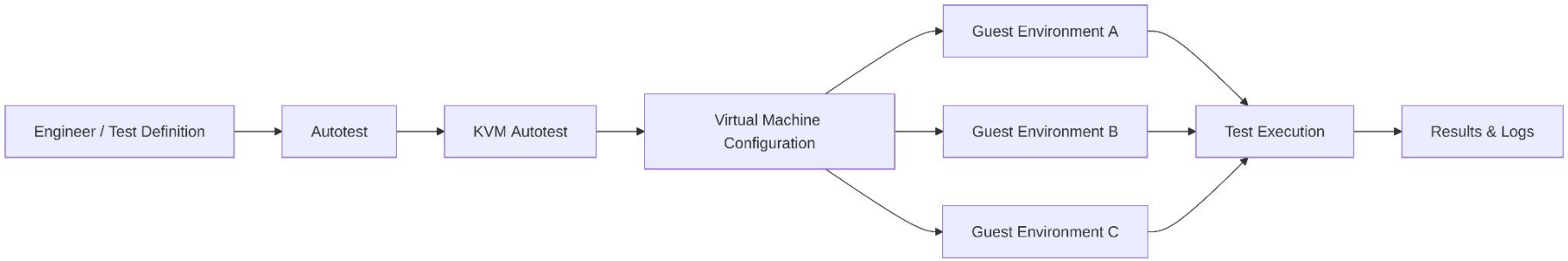
How Autotest Works

- Scheduler selects the test job
- Server coordinates the target machine
- Client executes the test
- Database stores logs and results
- Web UI / CLI presents the results



And talking about KVM Autotest

KVM Autotest extended the Autotest ecosystem specifically for **KVM/QEMU** virtualization testing.



Starting a project in Autotest

1. **Clone Autotest**

   ```
   git clone https://github.com/autotest/autotest.git
   ```

2. **Create a test folder**

   ```
   client/tests/my_test/
   ```

3. **Write the test**

   ```
   class my_test(test.test):
       def run_once(self): ...
   ```

4. **Create a control file**

   ```
   job.run_test("my_test")
   ```

5. **Run it**

   ```
   client/autotest-local client/tests/my_test/control
   ```

The test is built around Autotest's own structure and APIs.

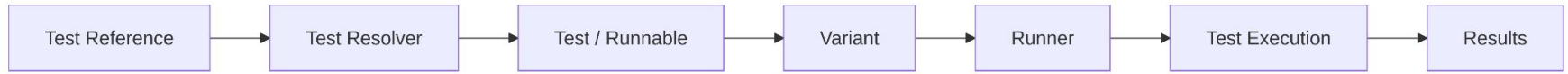
Problems with Autotest

- Tightly coupled architecture
- Client, server, scheduler, database, and frontend added operational complexity
- Difficult to maintain as test infrastructure grew
- Virtualization support made configuration even more complex
- Less natural fit for modern CI/CD and plugin-based workflows

It is more of a **framework** than **modular plugin.**

Now getting into avocado

Test Reference — defines what you want to run.
Resolver — finds and identifies the actual test.
Varianter — generates different parameter sets for the same test.
Runner — knows how that test type should be executed.
Spawner — decides where and how the test process is launched.
Result infrastructure — collects status, logs, and execution results.



Starting a project in avocado

1. **Install Avocado**

   ```
   pip install avocado-framework
   ```

2. **Create a test file**

   ```
   touch my_test.py
   ```

3. **Write an Avocado test**

   ```
   from avocado import Test

   class MyTest(Test):
       def test(self):
           self.log.info("Running test")
   ```

4. **Run the test**

   ```
   avocado run my_test.py
   ```

5. **Avocado resolves the test**

   ```
   Reference → Resolver → Test
   ```

6. **Runner executes it**

   Avocado selects the appropriate runner for the test type.

7. **Results are generated**

   ```
   PASS / FAIL / ERROR / SKIP
   ```

Why Avocado Stood Out to Me

1. Variants / Variators

Let you run one test across many configurations by changing parameters instead of duplicating test code.

```
arch: !mux
  x86:
    arch: "x86"
  arm:
    arch: "arm"
```

Why Avocado Stood Out to Me

2. Plugin-based architecture

Extend resolvers, runners, spawners, result formats, CLI behavior, and more.

For eg:

- **Avocado-VT** adds virtualization-specific testing on top of Avocado
- **Avocado-I2N** builds additional orchestration and system-testing capabilities around Avocado/Avocado-VT for more complex test environments

Why Avocado Stood Out to Me

3. Multiple test types

Run different kinds of tests in the same framework.

For example:

1. **Executable scripts** — `.sh`, binaries
2. **Python `unittest` tests**
3. **Avocado instrumented tests**
4. **TAP-producing tests**
5. **Plugin-defined custom test types**

Different test formats, one execution framework.

Why Avocado Stood Out to Me

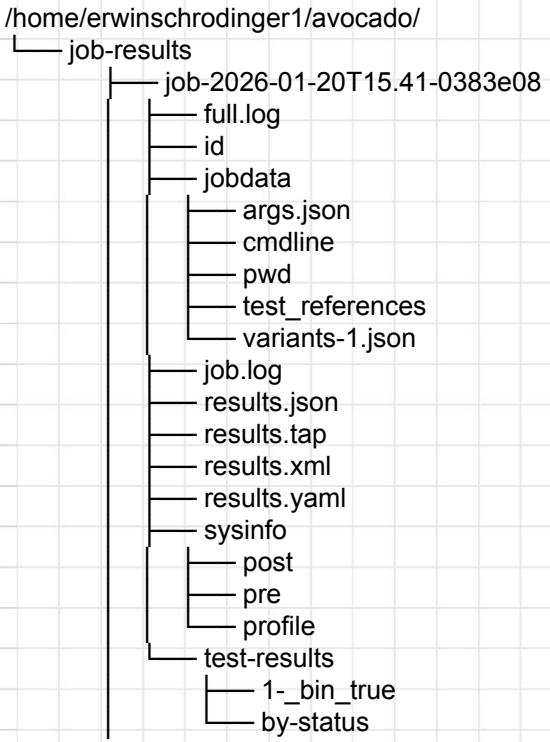
4. CI/CD-friendly workflow

- Simple CLI execution — avocado run tests/
- Structured results and exit codes
- Machine-readable output — JSON, TAP, JUnit/XML
- Easy artifact and log collection
- Works well with GitHub Actions, GitLab CI, Jenkins, etc.
- Same test command locally and in CI

Why Avocado Stood Out to Me

5. Structured Test Outputs

- Avocado produces consistent, machine-readable test results, making it easier to understand failures and integrate testing into automation.
- Clear statuses — PASS, FAIL, ERROR, SKIP, WARN
- Logs and metadata grouped per job/test
- Machine-readable formats such as JSON and JUnit/XML
- Easy to archive as CI artifacts
- Simple for dashboards, scripts, and pipelines to consume



Why Avocado Stood Out to Me

6. Better Separation of Concerns

- Avocado keeps test logic, parameters, execution, environment handling, and reporting as distinct parts of the framework, so each can evolve independently without turning the test itself into infrastructure code.

```
project/
├── tests/
│   └── network_test.py
├── configs/
│   └── variants.yaml
└── plugins/
    └── custom_runner.py
```

Open To Work as Software Engineering Jobs, academics Research on computational efficiency, energy minimization and making technology accessible

References

https://autotest.readthedocs.io/en/latest/

https://avocado-framework.readthedocs.io/en/latest/

https://github.com/avocado-framework/avocado-vt

Any Queries?
Thank You

I might not know everything — thankfully, GPT exists 😉