SINCE 1994

Managing *complex* test environments.

A LOGISTICS PROBLEM, NOT A SPREADSHEET PROBLEM · REX BLACK, INC.

Some bugs are *shy.*

They only reveal themselves in **field-like conditions:** server crashes under load, data corruption under concurrent write, regressions specific to a driver or firmware.

Testing for them needs an environment **complex enough to provoke them.**
That environment becomes its own system of record.

WHAT THIS IS ABOUT

In plain English.

If your product only works in a production-like mesh — many services, many versions, many labs — then your **test environment itself** becomes a logistics operation. You need a system of record.

This talk answers:

- **Why** test in production-like conditions at all.
- **What to model:** hardware, software, people, locations, schedules.
- **How to model it** with entities and relationships — the same frame that underwrites Airtable, Terraform, and every inventory tool.
- **What reports fall out** that management will actually read.

Shy bugs.

These bug classes **hide** in trivial environments and **surface catastrophically** in production:

- **Performance** at scale.
- **Volume and capacity** limits.
- **Data quality** failures.
- **Reliability / MTBF** regressions.
- **Driver- or firmware-specific** failures.
- **Model drift** on specific dataset slices.

WHAT A PRODUCTION-LIKE ENVIRONMENT ACTUALLY IS

Nodes · versions · states · schedules.

ENTERPRISE

Dozens of services across cloud regions, on-prem data centers, and third-party APIs.

EMBEDDED / IOT

Hardware, firmware, radios, calibration equipment in specific physical locations.

AI / ML

Model versions, dataset versions, GPU SKUs, cache states, prompt templates, evaluator configs.

THE COMMON STRUCTURE

Nodes. Relationships. States. Versions. Schedules.
Everything else is details.

SYSTEM OF RECORD · SEVEN THINGS MINIMUM

Track these or guess forever.

- **Hardware installation, locations, relocations.**
- **Hardware configurations** — current, historical, planned.
- **Interconnections and networking.**
- **Test locations.**
- **Test infrastructure** — harnesses, simulators, stubs.
- **Engineer assignments and locations.**
- **Human resource deployment.**

Answers the three questions every program gets at 2 PM on release day:
Where is it · who has it · what's on it?

THE FRAME

Entity-relationship *thinking.*

THE E-R RECIPE

Same frame. Different tool.

Airtable. Terraform state plus a harness layer. JSON/YAML in git. Bespoke internal app. All fine. **The method below doesn't change.**

- **Identify entities** — hardware, software, testers, tests, locations.
- **List their properties** — serial, build hash, username, test ID, site code.
- **Pick key properties** that uniquely identify each.
- **Identify relationships** — one-to-one, one-to-many, many-to-many.
- **Note relationship properties** — date, cycle, pass/fail, time-on-task.

The E-R model is the **conceptual backbone.** The tool stores it.

FIVE ENTITIES AT A GLANCE

The minimum model.

HARDWARE

Key: serial / asset ID.
Properties: type, spec, state, location, install-date, owner.
Relationships: installed at *Location*; runs *Software*; allocated to *Test*.

SOFTWARE

Key: revision identifier (e.g. *v4.2.1-gpu-a100*).
Properties: phase, cycle, platform, build-date, artifact URL, checksum.
Relationships: installed on *Hardware* for *Cycle*; under-test by *Tester*.

TESTER

Key: username.
Properties: role, shift, home location.

TEST

Key: test ID.
Properties: suite, duration, hardware requirement.

LOCATION

Key: site code.
Properties: type (lab, datacenter, customer site, remote), capacity, timezone.

WHAT REPORTS FALL OUT

Reports are the product.

The model is infrastructure.

- **Per-tester weekly plan** — tests × days.
- **Per-test schedule** — execution window.
- **Per-location capacity** — hardware × assignment × date.
- **Per-cycle coverage matrix** — platform × phase × cycle.
- **Per-build history** — software × platform × tester × pass/fail.
- **Environment-drift diff** — what changed between cycle *N-1* and cycle *N*.

These are what management reads. The entity model exists to produce them cheaply.

LOCATIONS MATTER MORE THAN TEAMS ADMIT

Capture location *over time*.

Equipment is installed in locations — test labs, colos, customer sites.
People work at locations — labs, cubicles, home offices, on the road.

The location graph **changes.** Equipment relocates. Labs reconfigure. People move.

A model that doesn't track locations **over time** cannot explain why a test ran differently this cycle than last.

Capture location. Capture dates on location.

SOFTWARE IS A STACK, NOT A VERSION

Drift *is* the regression.

The model has to hold BIOS / firmware · OS · applications · VMs / interpreters · utilities · test tools and scripts.

The release plan ties **specific revisions to specific cycles on specific platforms at specific dates.**

When a regression appears, the question *"what changed in the stack between cycle N-1 and cycle N on this platform"* has to be answerable from the model in **a single query** — not reconstructed from memory.

Five platforms. Three phases. *Three cycles.*

CASE STUDY

A 5 × 3 × 3 lattice.

Forty-five distinct slots. Each with its own build revision, assigned tester, hardware allocation, and date.

- **Revision grammar:** *C.1.Mac* = Component, Cycle 1, Mac build. A single naming convention makes the model legible.
- **Release dates live on the revision,** not the plan.
- **Tested configurations are a query,** not a static report.

Replace the historical platforms (Mac / Win / Solaris) with today's — *four cloud regions × three runtime versions × three model revisions* — and **the structure holds.**

WHAT GOES WRONG WITHOUT IT

Three failure modes.

- **Materiel misunderstandings.** Two teams assume two labs are the same, or two firmware versions are the same. Test fails or succeeds for reasons *no one can reconstruct.*
- **Missing coverage** against customer-impacting bugs. The customer's environment **wasn't in the matrix** — because no one was counting matrices.
- **Bug irreproducibility.** The build that triggered the bug can't be re-created because **no one captured the stack** at the moment of failure.

All three **go away** when the environment is modeled instead of remembered.

Model the graph.

Write the queries.

TAKEAWAYS

Four to hold against.

- **Treat the test environment as data.** If it lives in heads and hallways, your complex-system testing is guesswork.
- **Entities and relationships, not tools.** Pick the tool the team can maintain *without a DBA.*
- **Reports are the product.** Model is infrastructure. Per-tester plan, per-cycle coverage, per-build history — these are what management consumes.
- **Dates and locations are first-class.** Track everything with a date and a location. Regression triage stops being forensic work.

> *Test results in less-complex settings rarely extrapolate, because software is not linear. If your customer runs it in a mesh, you have to test it in a mesh.*

SINCE 1994

Thank you.

REX BLACK, INC. · REXBLACK.COM/RESOURCES/TALKS/MANAGING-COMPLEX-TEST-ENVIRONMENTS