Netcastle: Network Infrastructure Testing At Scale

Rob Sherwood [⋆], *Jinghao Shi* [†], *Ying Zhang* [†], *Neil Spring* [†], *Srikanth Sundaresan* [†], *Jasmeet Bagga* [†]
Prathyusha Peddi [†], *Vineela Kukkadapu* [†], *Rashmi Shrivastava* [†], *Manikantan KR* [†], *Pavan Patil* [†]
Srikrishna Gopu [†], *Varun Varadan* [†], *Ethan Shi* [†], *Hany Morsy* [†], *Yuting Bu* [†], *Renjie Yang* [†]
Rasmus Jönsson [†], *Wei Zhang* [†], *Jesus Jussepen Arredondo* [†], *Diana Saha* [†], *Sean Choi* [‡]

[⋆]*NetDebug.com* [†]*Meta Platforms Inc.* [‡]*Santa Clara University*

Abstract

Network operators have long struggled to achieve reliability. Increased complexity risks surprising interactions, increased downtime, and lost person-hours trying to debug correctness and performance problems in large systems. For these reasons, network operators have also long pushed back on deploying promising network research, fearing the unexpected consequences of increased network complexity. Despite the changes' potential benefits, the corresponding increase in complexity may result in a net loss.

The method to build reliability despite complexity in Software Engineering is testing. In this paper, we use statistics from a large-scale network to identify unique challenges in network testing. To tackle the challenges, we develop Netcastle: a system that provides continuous integration/continuous deployment (CI/CD) *network testing as a service* for 11 different networking teams, across 68 different use-cases, and O(1k) of test devices. Netcastle supports comprehensive network testing, including device-level firmware, datacenter distributed control planes, and backbone centralized controllers, and runs 500K+ network tests per day, a scale and depth of test coverage previously unpublished. We share five years of experiences in building and running Netcastle at Meta.

1 Introduction

Decades of accumulated network deployment experience can be pragmatically summarized as: seemingly simple changes to the network can break things in ways that are both catastrophic and non-obvious even to experts. Other times the breakage is subtle and thus hard to detect and root cause [31]. This unfortunate state of affairs is due to the complex reality of modern networks. Many aspects contribute to complexity including the number and variety of devices, workloads, device types and manufacturers, as well as typical distributed systems problems of state distribution, race conditions, consensus, and propagation delays. Compounding these issues, each of these dimensions of complexity seem to be individually increasing (more devices, more workloads, more device types, etc.) for a multiplicative increase in total complexity.

Any change to the system risks non-obvious performance impact or even outages; increased complexity only exacerbates this danger. Thus, it is natural for network operators to want to limit changes to the network. However, this is not practical in today's world. Many changes are initiated by network operators out of necessity, e.g., for network growth or to replace obsolete, inefficient, or failed devices. Moreover, many changes happen without consultation or control of network operators: new users join the system, usage patterns change, and new services are regularly deployed—all assuming that the network will "just work". Thus, increased complexity coupled with regular, sometimes unvetted, change requires constant vigilance of operators to keep networks reliable.

Modern software engineering promises us that complexity can be managed with increased testing and test automation [2]. This is particularly true in software-as-a-service ("SaaS") models where large distributed systems with thousands of active developers continuously change under the covers, unnoticed by end-users, and rarely introducing bugs. These systems rely heavily on various levels of unit, regression, performance, stress, and end-to-end testing to continuously ensure that the proposed changes do not break desired functionality. Only changes that do not break tests are allowed to land ("continuous integration" - CI) and automated tooling regularly applies these changes to the production systems and monitors the state of new deployments ("continuous deployment" - CD) for regressions. With sufficient test coverage, even non-trivial changes (e.g., a large refactoring) can be performed safely.

It should, in theory be possible to run a network with both high-reliability and a high rate of change, using SaaS principles. However, this assumes that it is possible to verify by exhaustion that the network works correctly before and after a proposed change, which in turn requires writing an offline non-production test for every conceivable situation. This simple assumption is challenging to meet in practice. In this paper, we first use the well-known testing matrix [21] concept to quantify the complexity of network testing in production. In

addition to the theoretical analysis, we reveal the real-world challenges of testing large network systems. Motivated by them, we share the development and operation of a single automated, centralized network testing system: Netcastle, which has been in production for five years at Meta. This paper provides the following contributions:

First, we analyze the complexity of network testing formally using the concept of the testing matrix and quantify testing complexity in a production network. Further, we review the limitations of our original test methodologies and use them to derive the requirements for an idealized *network testing as a service* system (§2) including the need for fungible shared hardware resources, dynamic L1 topologies, and a testing pyramid hierarchy that trade-off control vs. coverage. This is the first work that provides systematic and quantitative analysis of the network testing problem.

Second, we present Netcastle, a collection of software and hardware systems that provide network testing as a service (§4). It hides low-level resource management details with high-level abstractions that ease test development. Its management layer automatically handles test scheduling to maximize resource utilization and minimize noise. In addition, it leverages advanced hardware modules such as optical switch and smart power distribution units (PDUs) to provide flexible topology and isolation in a common physical infrastructure.

Third, We share five concrete testing use cases built on top of Netcastle infrastructure, including data center network testing, wide-area Backbone network testing, FBOSS white-box switch testing, and OpenBMC firmware testing (§5). They not only provide details of real-world network tests but also demonstrate how Netcastle enables them flexibly.

Fourth, We evaluate Netcastle's scalability and usability in §6. It comprises over 500 racks with 3MW power and thousands of switches. It supports 171 test scenarios across multiple teams in Meta. It manages 87K assets and receives 495K reservations per day, which is around 6 reservations per second. The services handle 37M requests daily at an average of 428 queries per second. Moreover, we show its impact with five severe bugs that were caught by Netcastle which would otherwise cause catastrophic outages.

Fifth, as the first to propose and share the *network testing as a service* framework with academia, we discuss the testing implications for network complexity and possible future research directions in §7.

2 Motivation

We first motivate the problem by leveraging a well-known metric in software engineering, the test matrix [21]. Intuitively, complexity is a function of the building blocks of the network: e.g., features, components, and device types. In a well-run system, all individual elements will have corresponding tests to ensure they are functioning correctly; as will the interaction between the elements. Thus, the more complex a system, the more tests needed to validate correctness, and thus the larger the test matrix. For the largest networks, such as at Meta, the testing system complexity itself can limit the rate of change of the network and is therefore worth studying on its own.

To illustrate the above, in this section, we quantify the test matrix for a single software system and show how it can grow multiplicatively over features, tests, etc. We then describe how the composition of systems in a network make the overall test matrix grow exponentially. This untrammelled growth serves as the primary motivation for Netcastle; the bigger the test matrix, the more engineering that is required to tame it. More discussions of test matrix are in §A.

2.1 A Single Project's Testing Matrix

FBOSS [5, 29] is the software that manages the switch hardware of Meta's data centers. Its many functions include translating high-level messages from the routing stack to the underlying hardware, exporting statistics, and sourcing network alerts (e.g., port down). FBOSS is updated across the global fleet at least monthly. With each new code or configuration change, the FBOSS binary is tested against all elements of its test matrix to ensure none of the new changes have (a) broken other features (e.g., does ARP expiration still work?), (b) caused a performance problem (e.g., did the route insertion rate drop below a threshold), or (c) caused a regression in a critical scaling dimension (e.g., maximum routes supported). The FBOSS test matrix has the following dimensions:

- Each switch hardware platform deployed in production [1]: e.g., Wedge40/Wedge100/Wedge400 (rack switches), Minipack/Minipack2 (100G/200G fabric switch)
- Each Vendor SDK type/version: e.g., Broadcom SDK 6.5.17, 6.5.18, SAI
- Each boot mode: cold (empty ASIC memory), warm (pre-populated ASIC memory)
- Each feature test: 800+ different features – see "fboss/fboss/agent/hw/bcm/tests/*" in [29]

Validating that a given change has not caused a regression requires all combinations of (6 hardware types) $\times$ (3 SDK versions) $\times$ (2 boot modes) $\times$ (800 feature tests) $=$ 28,800 tests. Each test can take from a few seconds to tens of minutes to run; a serial run of all FBOSS tests would take (impractically) days to finish. Developers may launch several test runs daily and there are dozens of developers, resulting in hundreds of thousands of test runs per day from this single software project (more data in §6.1.1 and Figure 8).

While it might be tempting to take short cuts in this test matrix, e.g., test more intelligently than the brute-force combination of all dimensions, experience has shown that this can lead to bugs being deployed into production. For example, it is not uncommon for a bug to only affect a single feature on one SDK version and one switch type. Predicting these unlikely interactions in advance is hard and predicting incorrectly can

cause an outage, so our hard-won lesson is that brute-force testing all combinations is necessary.

Software engineering best practices dictate that every new feature or change should add its own tests, thus adding complexity to the system. How much complexity a change adds depends exactly on how it affects the test matrix:

- *New element to an existing dimension*: When the FBOSS team introduced a new switch type, Minipack, they increased the "switch hardware" testing dimension from 5 to 6 devices, resulting in a corresponding 20% increase in tests from 24,000 per run to 28,800 tests per run. This is the least complex type of change.
- *Adding a new test matrix dimension*: This is more complex—even a simple dimension with a binary element doubles the size of the test matrix; more elements would result in a correspondingly larger matrix. For example, the FBOSS team experimented with the Algorithmic Longest Prefix Match (ALPM) route memory storage algorithm which increases route capacity. Since ALPM potentially affected all tests and was not initially deployed pervasively, it required a new binary dimension to the test matrix, ALPM off *vs*. ALPM on, or a 100% increase in test matrix size.
- *Adding multiple new dimensions to the test matrix*: An example of a dimension with potentially more elements is the parameter values for Explicit Congestion Notification (ECN) support. ECN has a threshold parameter for when to mark packets and every queue has a different allocation of the shared buffer—forming two independent testing dimensions. In practice, we explored 4 different ECN threshold values and 3 different queue allocations for a net 1200% increase in test matrix size.

2.2 Multiple Project Combinatorial Explosion

Above, we showed that a single network component can have a very large number of tests; testing the network as a whole across different software components results in a combinatorial explosion of tests, e.g., does feature A in version B of FBOSS correctly inter-operate with feature X in version Y of BGP? For the number of software systems and features that interact, this testing matrix quickly become intractable. To see why, we note that FBOSS updates the global data center fleet software every two weeks, and the Backbone controller and switch firmware (details in §5) every 3-6 weeks. Such frequent large-scale releases are only possible if we can have safe, efficient, and extensive testing.

3 Evolution and Challenges

Before Netcastle, every network team ran their own independent test lab and processes. As a result, we have a number of different physical facilities for data center network testing, new hardware testing, backbone network testing, etc. Each of them is set up manually with a small number of switches under a fixed wiring, and hard-coded with static configurations. Making changes to the testbed often requires physical access, together with manual and ad-hoc efforts.

We term these disparate labs "traditional" and in this section, describe key problems with that approach that motivated the Netcastle design. Network testing labs must be isolated from the production network, but this isolation and non-standard testing configurations lead to maintenance challenges. Thus a common theme of traditional test lab problems is "lab rot"— the tendency for labs to accumulate old configurations, forgotten topologies, unmanaged devices, and unclear ownership. Lab rot increases uncertainty in the test signal (is it my code or the lab that is broken?), risks security problems (e.g., missing patches), and wastes resources that users hoard to protect against the misconfigurations of others. This generation of testing exposed several challenges that leads to the Netcastle design, which is summarized in Table 1.

3.1 Physical resource limits

Limited Lab Scale, Unbounded Device Heterogeneity. Test labs are an extra expense, so are typically much smaller than production networks. When a test lab is allocated to each individual team, they are particularly limited in size, making it difficult to find problems that only appear at production scale (e.g., table exhaustion) or infrequently (e.g., race conditions).

At the same time, network testing is meant to confirm that device A will interoperate with device B, across all deployed device types and all relevant configurations. Compounding this interoperability problem is that meeting the relevant specification or RFC alone is not necessarily sufficient; real interoperability requires "bug-level compatibility" testing which requires all n^2 pairs. Further, more device types are added to the network faster than old device types are decommissioned, resulting in unbounded device heterogeneity, and high demand for testing. Because individual teams maintain small scale test labs, this results in compromises to test coverage.

Device, Power, and Space Fragmentation. Physical realities mean that at lab construction time, each row and each rack within that row is allocated an amount of space (rack units) and electrical power (kilowatts) that is cost prohibitive to change. However, many test setups require locality in the form of direct physical connection. This means that when constructing topologies of nearby resources that can be directly connected, fragmentation is possible: devices in a row that are not used by one test setup may be too few to be usable by another. This fragmentation can be aggravated if test device use is of unbounded duration or devices cannot be reclaimed.

3.2 Automating test configurations

Static, Manually Maintained Topologies. A hardware test network must support many topologies: physical cables that di-

Network Testing Challenges		Addressed By Netcastle
Physical resource limits	Disparate Testing Labs Limited Scale Device, Power, Space Fragmentation	Automated Device Enumeration + Lab Syncer (§4.5) Fungible Centralized Resource Pool (§4.1) Centralized Fiber Infrastructure (§4.2, §4.3)
Automate config	Static, Manual Topologies	Any-To-Any Optical Interconnects (§4.3)
Manage and schedule tests	Large Testing Matrices Test Harness Hording Testing Conflicts	Netcastle Test Runner (§4.4) parallelization Finite Reservations + Usage Monitoring (§4.1) Reservations (§4.1, §7.7)
Reduce test noise	Broken Testing Devices Test devices not maintained	Lab Doctor (§4.6) Lab Doctor (§4.6) + Finite Reservations (§4.1, §7.5)
Debug tests	Network Testing Distributed System	Respect the Testing Pyramid (§7.3)

Table 1: Summary of Problems Resolved By Netcastle Design

rectly connect the devices under test. It is not practical to manually rearrange cables at the time scales of when tests run: cable moves can take up to a day to schedule from onsite personnel, while tests run many times per second. Instead, available devices are partitioned into different manually configured topologies, e.g., Clos, star, or snake. With this partitioning, it is not sufficient to have a device of type A available; it must also be cabled appropriately. Ultimately, this means more device types are needed for tests with different topologies.

At the same time, the manual maintenance that supports standard topologies makes it more difficult to manage one-off, experimental topologies. "Innovative" cabling decisions (i.e., "who ran this fiber through the ceiling!? and what will break if I unplug it?") meant to support these topologies can become difficult to manage and contribute to lab rot.

3.3 Managing and scheduling tests

Test Equipment Hoarding. Test lab users tend to hoard equipment, not through malice, but as a practical reaction to the difficulty of acquiring reliable test gear. Once a user has gone through the significant effort to get a test harness installed, debugged, and working correctly, they are rightly hesitant to release ownership because the cost of setting it all back up again is high. Hoarding aggravates the problem of limited lab scale (fewer available resources of specific types) and fragmentation (fewer available resources in specific racks).
Test conflicts. Unfortunately, one test may adversely impact another. Although labs are designed for test isolation, due to hidden dependencies on external services, unclear ownership, and raw complexity it is inevitable that some amount of test breakage occurs due to stepping on toes. A simple example is mistakenly believing a device/cable was not in use and changing it in the middle of someone else's test . More subtly, networking devices often have non-obvious inter-dependencies: the Wedge series of switches have a main CPU complex running Linux (the "microserver") and a completely separate baseboard management controller (BMC [10]). This architecture allowed a different developer to mistakenly log into each system on the same switch and run tests in parallel with apparent but incomplete isolation. In most cases a test on the microserver would not affect a test on the BMC, but under certain test workloads, e.g., PCI controller reset, it would cause subtle and hard to debug test breakage. Also, we discuss in § 7.7 a hard-to-fix architectural issue in traditional labs where test equipment often depended on other test equipment.

3.4 Reducing testing signal noise

Tests leave devices in complex, broken states. Tests intentionally run on questionable code, configurations, and topologies. As a result, they can break devices in obscure and hard to revert ways. For example, a test configuration may disconnect a switch from its console server and management interfaces, making it unreachable without onsite intervention. In extreme cases, e.g., bad firmware, it is possible to cause permanent physical damage to the hardware. This need to verify and debug test equipment before use both justifies the desire to hoard test equipment and also takes resources offline, compounding problems of scale and fragmentation.
Test devices are not maintained like production devices. Large scale production networks are maintained with ruthless homogeneity and pervasive automation. Because lab networks are heterogeneous and transient, they are both hard to automate and cannot be easily directly managed with existing production management tooling. As a result, software upgrades, security patches, system problem detection ("did the disk fill up?", "did the optic stop working?"), hardware fault detection, etc., typically lag behind the production network, leaving devices in various states of disrepair.

3.5 Debugging complex tests

Network testing is distributed systems. While a test setup is simple to sketch on paper, e.g., "n_1 nodes from vendor X, n_2 nodes from vendor Y, running software of a certain version and a certain topology, send messages $m_1 \ldots m_n$, and a link goes down at time t", actually deploying and validating that these parameters are setup and timed correctly is a complex distributed systems problem. Test devices can be unavailable,

temporarily lose connectivity, have bugs independent from the target of the test (e.g., from previous tests), etc. Similarly, correctly sequencing the control signals that cause test events, e.g., "send a BGP HELLO and then the link goes down", and read their results so that the test runner's understanding of the current state correctly reflects the physical reality ("did the link actually go down? Before or after the HELLO?") reduces directly to standard distributed systems consensus problems. And while one might expect that test labs do not run in a byzantine adversarial model and thus consensus should be easy, practical experience shows that significant time is spent debugging the tests themselves; and the bigger and more complicated the test, the more things that can go wrong. Ultimately, tests need to be highly repeatable and problems in the test, e.g., 1 in 1000 race conditions in the test code adds unacceptable noise if it is supposed to catch even less frequent race conditions in the systems under test.

4 Netcastle Testing Infrastructure

The Netcastle team provides "network test infrastructure as a service". The team does not write the tests, instead they provide and maintain the backend software and lab resources, and document best practices so that partner teams (in § 5) can focus on testing their own code instead of on the challenges in § 2. Figure 1 shows this interface. In this section, we describe how Netcastle addresses the problems described in § 3. The relationship between the problems and features to address them is summarized in Table 1. By using these complementary solutions, more and larger tests are made possible, often by avoiding "lab rot".

The goal of Netcastle is to combine independent, smaller test labs from different teams into a communal pool of test resources, eliminating fragmentation and scale limitations. The project aims to replicate the cloud IaaS experience where users can reserve virtual machines via API without worrying about the physical location or available resources. Netcastle offers the same experience for reserving test equipment with strong test isolation, freeing developers from worrying about backend infrastructure maintenance and allowing them to focus on writing tests. To accomplish this vision, Netcastle provides many tools, processes, and points of integration to cover in detail, so we summarize the most innovative elements here and illustrate how they are used in Figure 3.

4.1 Centralized Reservations Guard Resources

In Netcastle, all test equipment must be reserved for finite time through a centralized system; users can query and update this reservation system via CLI, web interface, or a remote API that supports automation. Resources are annotated with key/-value pair attributes such as CPU/device type, kernel version, etc., enabling queries that find resources by these attributes. Individual resources can be grouped into larger atomically reservable units ("ensembles"), e.g., reserving an entire Clos topology rather than individual devices. This simplifies the user interaction and avoids reservation deadlock.

The centralized reservation system also has less obvious benefits. If the requested resources are unavailable, the caller can queue to wait until they become available, which improves resource utilization, particularly for automated testing (see § 7.5 for details and where this can go wrong). Keeping records of these wait times helps to identify which resource types are in high enough demand to warrant adding more, and instances when a component has failed when a queue is unexpectedly stalled. Automated tools can programmatically determine whether a device is participating in a test and act appropriately, for example, alerts, software updates, and other maintenance activities are suppressed for lab devices that are are participating in a test. Requiring each reservation to be of limited duration reduces resource hoarding and allows better sharing of finite resources. Similarly the reservation system provides instrumentation to monitor end-to-end test system performance (e.g., "the queue for routers of type Y is now 45 minutes long—something must be wrong," e.g., like § 7.5).

Given the importance of the centralized reservation system for Netcastle, it needs to have high performance (130k+ reservations a day, see § 6.1.1) and high availability (downtime impacts developer productivity). While intuitively trivial to build, the reservation system had to be redesigned/replaced three times as load increased over time.

4.2 Centralized Fiber Infrastructure

To remove physical locality as a constraint, and reduce resource waste through fragmentation, we deploy centralized passive fiber infrastructure to all test racks. That is, from each rack in the lab, we pull many pairs of fiber (96 or 192 pairs, depending on rack type) to a single central passive "fiber row" that is shared by and sized for the whole lab. By attaching a passive fiber patch cable from the corresponding ports in the central fiber row, any two devices anywhere in the lab can be directly L1 connected. This enables lab devices to be placed wherever there is physical space and power, making the devices interchangeable and reducing fragmentation.

The centralized fiber row is also an ideal place to deploy test equipment shared between teams, including packet generators and passive optical switches. Decoupling physical locality also reduces the number of variations of device deployments, which simplifies automation and reduces lab rot.

4.3 Any-To-Any Optical Interconnects

In addition to manually connecting patch cables in the central fiber row, Netcastle also supports the ability to dynamically, programmatically, create L1 optical topologies. Despite the investment in the central fiber row and the existence of commercial-off-the-shelf programmable passive optical

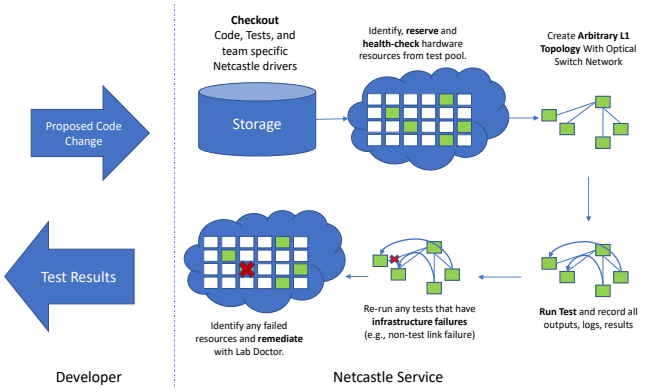

Figure 1: Netcastle testing as a service model.

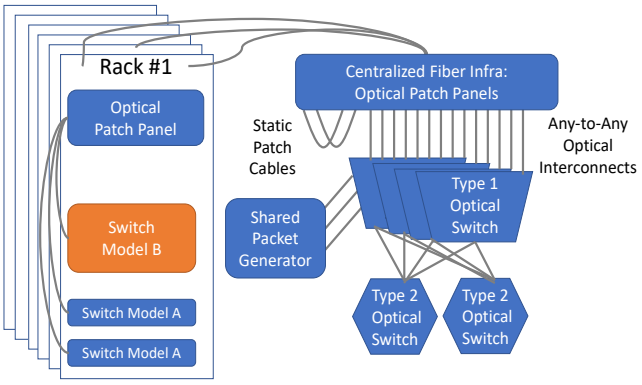

Figure 2: Physical Infrastructure of a Netcastle Lab

switches [14, 15, 33], this is non-trivial due to scale and link setup latencies. Specifically, in order for the link to come up between two devices, the signal to noise ratio, as measured by "light loss level", must be lower than manufacturer specifications. For example, MSA, a common 100Gbps optical standard, allows for a light loss budget of up to 5dB total noise [35] between two devices, including passive connectors (∼0.15–0.5 dB per connector) and the fiber itself (0.5 dB/KM for single mode or 1.0 dB/KM for multimode), must be less than 5dB in order to achieve a high quality connection [24].

Existing commercial passive optical switches fall in to two broad groups. The first use Micro-Electromechanical Systems (MEMS) mirrors or crystals to bend light [14,33]; establishing a link relatively fast(∼10ms) but adding significant light loss (2.5–3dB). They have medium port densities (200–400 ports per device). The second use robotic arms to manage physical passive fabric cables [15]; these can take minutes to establish each link but have light loss equivalent to passive gear (0.5dB per connector), and support high port densities of over 1000 ports per device. Our challenge in making these pieces scale is twofold: first, developer time is important, so it would be prohibitive to have to wait hours to setup a large network topology using only robotic arm based switches. Second, for the number of devices in our labs, there is not a single device big enough to support any-to-any connectivity. We cannot use a "multi-hop" passive optical network with chained optical switches, because it would exceed our light loss budget (e.g., 2 devices each at 2.5dB loss, plus optical connectors, exceeds 5.0 dB). Complicating matters, our data centers prefer OCP [32] standard optics which are cost optimized and have a smaller light loss budget of 3.5dB. We could use different optics (e.g., Multi-Source Agreement or other standards) in our lab, but then we risk testing on equipment that are not used in production, which could undermine our test results.

Faced with these multi-dimensional problems, we developed a hybrid solution that allows limited any-to-any connectivity in our labs (Figure 2). The first insight is that the light loss budget for a given standard (3.5dB for OCP, 5.0 for MSA) is really a *minimum* guarantee and, due to manufacturing variance, a device may support somewhat higher light loss. In practice, we found that most of our nominally 3.5dB OCP optics in fact supported a light loss budget of 5dB. Second, after discussion with multiple MEMS manufacturers, we found that stated light loss for an optical switch (e.g., 2.5–3.0 dB) is a conservative *maximum*; real light loss varies depending on the distance across the device from the input port to the output port. In practice, by carefully picking input and output fiber ports that are physically close to each other, we can significantly reduce light loss. Last, to balance the trade-off between high density, but slow, and and medium density, but fast, connectivity switches, we build a hybrid leaf-spine fabric of optical switches. The leaf nodes which connect directly to the devices under test use the medium density fast connectivity optical switches. We use the high density with slow connectivity switches as spine switches. Then, similar to leaf-spine topologies with electrical-optical switches in the DC, we connect every leaf optical switch to every spine optical switch. As a result, we can achieve any-to-any connectivity across our leaf-spine optical switch fabric with a centralized scheduler: when a developer wants to create an L1 connection between two devices, the centralized scheduler looks up which leaf switches and ports the devices are connected to and allocates a leaf-to-spine-to-leaf optical circuit that is already set up and physically close to the input ports. Thus with this design, we get all three of our desired properties of scale, meeting our light loss budget, and quick link setup.

This any-to-any topology-on-demand feature has enabled a number of network-wide test use cases. For example, examining different generations of data center networks (§5, §6) require creating arbitrary topologies quickly, running tests, and tearing the topologies down. However, topology dynamics introduce additional complexity and potential failures (§7).

4.4 Netcastle Test Runner

Netcastle test "runner" is a unified interface for launching tests and parsing results. The runner combines an array of workflows, including one-off interactive manual testing on

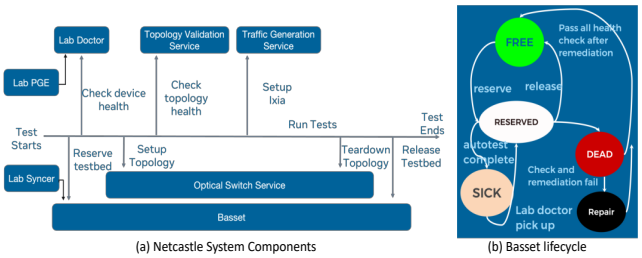

Figure 3: Netcastle Components.

single devices, batch parallel processing across thousands of devices ("test this code with all feature combinations on all types of devices"), to long running stress runs ("run this test 10k times and count failed runs"). The runner automates calls to the reservation system and lab doctor, copies test code and configurations into place, parses the test output, and reports the results. In addition to its own internal accounting databases, the runner must integrate with the graphical tools that developers use to review code, the company-wide test result database that monitors the test results history (e.g., to differentiate between flaky tests vs. tests that are newly consistently failing), and the CI/CD systems that decide if a given build is stable enough to move to the next step in the deployment cycle. For large test runs, the runner also handles sharding across multiple assets in parallel and retrying tests when assets fail for infrastructure reasons. It parses the test output and decides if a given test failure is legitimate (e.g., a bug) vs. a failure in the underlying test infrastructure (§ 7).

Like the centralized reservation system, Netcastle runner is intuitively simple; however, the combinations of use-cases, systems that it must integrate with, and the distributed systems problems associated with setting up and running multi-asset tests, has made it a non-trivial piece of software (§ 7.5).

4.5 Lab Syncer: Asset Verification

To combat lab rot, Netcastle deploys many pieces of automated tooling. *Lab syncer* compares dynamically inferred data (e.g., from connecting to every port on a console server, and from screen scraping management switches) to a centralized configuration to discover devices and report changes to the assets under management. Lab Syncer also verifies that all assets have a common collection of addresses and metadata, e.g., that the reservation tags (device type, kernel version, etc.) match reality and that every device has a proper DNS name.

4.6 Lab Doctor: Device Remediation

The *lab doctor* performs health checks on devices as they exit a reservation to prevent broken resources from returning to the resource pool. It applies a well-known set of remediations to unhealthy devices; it also runs periodically on unreserved assets to ensure that they are healthy. Unhealthy assets that cannot be automatically fixed are marked *DEAD* until manual investigation. Figure 3(b) shows the complete asset life cycle.

```
# 1. Reserve testbed
testbed = Basset.reserve("dc_test_pool", purpose="Testing")
# 2. Check individual device health
is_healty, details = LabDoctor.check_health(testbed)
if not is_healthy:
  # this will be classified as infra error
  raise TestbedError(f"Testbed is not healthy: {details}")
# 3. Reconfigure connections to achieve desired topology
OpticalSwitchService.setUp(testbed, desired_topology)
# 4. Check topology level healthiness
TopologyValidationService.validate(testbed)
# 5. Connect to Ixia traffic generator
TrafficGenerationService.setup(testbed)
# 6. Run tests, example: warmboot without packet loss
# 6.1 Start traffic
TrafficGenerationService.start_traffic(testbed, TRAFFIC_SPEC)
# 6.2 Perform warmboot
forwarding_stack_service.clear_counter("pkt_loss")
forwarding_stack_service.perform_warmboot()
# 6.3 Assert no packet loss
pkt_loss = forwarding_stack_service.get_counter("pkt_loss")
TrafficGenerationService.stop_traffic(testbed)
assertEqual(pkt_loss, 0)
# 7. Tear down topology and release testbed
OpticalSwitchService.tearDown(testbed)
Basset.release(testbed)
```

Figure 4: Example Netcastle Test Case pseudocode.

4.7 Putting it together

Figure 3(a) shows how each component interacts with the test life cycle. Figure 4 is a pseudocode example illustrating how a test developer utilizes Netcastle. The process begins by reserving a test pool of resources and calling on Lab Doctor to check device health status. Then, the optical switch is configured to set up the topology and the traffic generation service (step 5), followed by a measurement to verify its accuracy. Next, in step 6, the test developer runs a set of tests, which we will explain further in §5. Finally, the topology is dismantled, and all assets are returned to the pool. This process ensures that resources are efficiently used and that the test environment is well-maintained throughout the process.

5 Netcastle Use Cases

In this section, we present five test scenarios that use Netcastle and how they relate to experiences from § 7.

5.1 Facebook Open Switch Software Test

FBOSS test aims to achieve rapid, automatic test and deployment for switch software [5, 29]. As we discuss in § 3, every FBOSS code change must be tested across all features, device types, and environments before commit and deployment. FBOSS updates the global data center fleet software every two weeks; this cadence is only possible and safe through extensive testing using Netcastle in the following aspects.

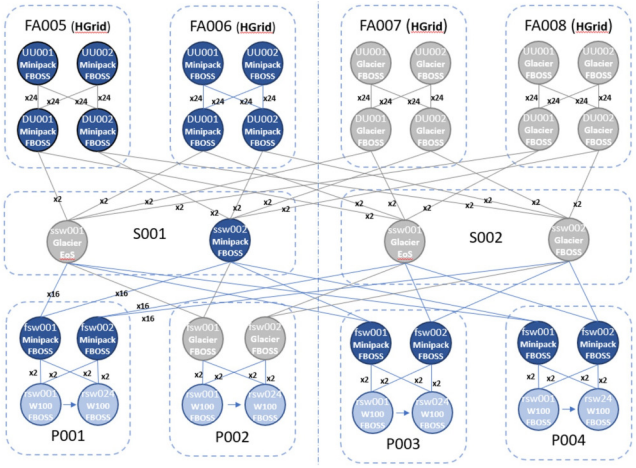

Figure 5: DC Network Testing Usecase.

1. **Hardware functional tests** verify that FBOSS works seamlessly on multiple hardware on a single switch box. About 800 test cases per switch ASIC and SDK combination verify core functions (e.g., QoS, load balancing), they run on every code commit and complete in about an hour.
2. **Hardware benchmark tests** ensure that the current version of FBOSS meets a certain performance threshold (e.g., warmboot time, ECMP shrink time, TX/RX rates). About 20 test cases per switch hardware type and SDK combinations verify that changes meet performance criteria, running once per day.
3. **Integration tests** verify FBOSS switches work seamlessly with other entities in the data center. These tests run on every code commit and completes in about 10 minutes.

Netcastle enables the FBOSS team to focus on writing tests for the software functionality, as Netcastle is able to obtain quality test signals that strictly distinguishes test failures between the software, hardware and the test infrastructure. Furthermore, the FBOSS team is able to have a more predictable deployment cycle, as they can predict how long a test cycle will take and how much coverage the the test signals provide.

5.2 Board Management Controller Test

OpenBMC [10] is an open-source software for managing board management controllers (BMCs) that are embedded in servers and switches to control hardware such as fan speed and remote server access. As the code is shared among switches and servers, thorough testing of OpenBMC is needed to prevent bad code from causing extensive damage. OpenBMC releases new versions 4-6 times per platform per year. The main challenges of testing OpenBMC are the diverse hardware it supports—over 19 different types at Meta—and the need to provide test signals publicly as it is open-source.

Netcastle solves these challenges as follows. For every pull request, Netcastle runs OpenBMC tests on variety of hardware types to catch regressions, and makes the test signals available to the open-source repository, facilitating debugging

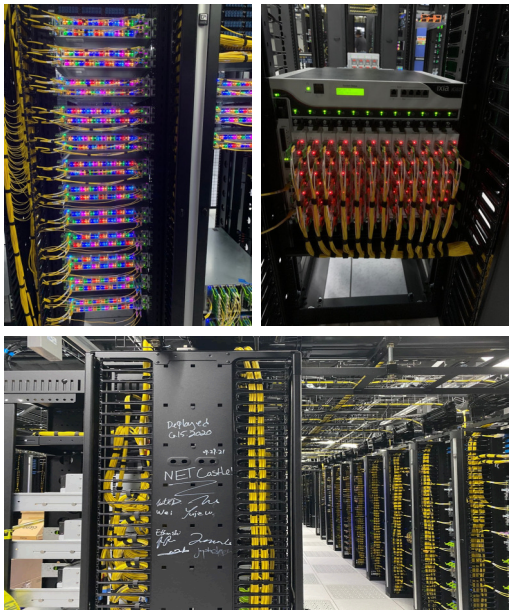

Figure 6: Netcastle Hardware Platform.

efforts by the community. Finally, Netcastle is used in validating OpenBMC image binaries. Thus, by utilizing Netcastle, OpenBMC is able to avoid building another set of testing infrastructure and a build pipeline, and is simultaneously able to reap the benefits of obtaining high quality test signals.

5.3 DC Network Testing

The Data Center Network Engineering team conducts multi-device testing on FBOSS switches and other vendor switches, focusing on network-wide objectives and configuration correctness. This includes designing and validating data center topology, managing network capacity, and more. Figure 5 shows a common set up for DC network testing. It faithfully mimics the production fabric topology in a smaller scale [5]: four pods are set up with two ToRs, connecting to two fabric switches, interconnected by two spine switches at the aggregation layer, and four aggregation switches at the core layer. The tests mainly consists of the following steps: test configuration setup, inducing disruptive events (e.g., software/hardware reboot/crashes, switch drain/undrain, link flaps), and recording the results (e.g., resource utilization and packet loss during the disruptive event). In the example in Figure 4, step 6 tests the packet loss during switch warmboot operations.

Testing new features presents challenges such as isolating the correct topology, and filtering false test signals from manually induced test failures. Netcastle addresses these challenges by enabling easy isolation of the testing topology through simple scripts for controlling switch interfaces, allowing configuration to reserve and use specific switches for testing, and filtering bad signals that are manually induced.

5.4 Express Backbone Controller

The EBB Controller [17] is the SDN WAN controller that manages Meta's inter-region global backbone network. The team has integrated Netcastle into its CI/CD pipeline since 2019 and ships a new release every 3-4 weeks. They required external traffic generators for their test cases; this pushed the Netcastle team to automate reservations and access to third party traffic generation tools. In this topology, software VLANs mimic edge data centers. Five lab express backbone (leb) routers are interconnected with parallel 200G physical links. This setting is used to test new controller software, traffic engineering algorithms, new EB router platforms, and SDKs, among other operations.

5.5 Hardware NPI

New Product Introduction (NPI) validates prototype networking gear and verifies that the hardware works correctly. This includes, among others, environmental tests, sensor and power checks, and data plane performance tests. Data plane testing is simplified by connecting ports in a "snake" topology: Port i is connected to port $i+1 \; \forall \; i \mod 2 = 0$, and traffic sent out of one port is received by the next. This reduces the traffic generator requirements from the full number of ports to only two; one source and one sink.

6 Evaluation

In this section, we demonstrate the impact of Netcastle with both real-world statistics and incidents caught.

6.1 System Evaluation

6.1.1 Scale: Tests, Daily Tests, and Reservations

Netcastle has grown to be the primary testing infrastructure for the Meta network. Figure 7 shows the cumulative number of tests developed on Netcastle over three years. From §2.1, the FBOSS team alone has 800+ different tests before considering different hardware and software configurations. In total, there are close to 4500 total tests developed across 20+ teams. The significant increase from week 40-80 represents Netcastle's largest adoption period. The steady increase after week 80 shows its continuous benefit as the network grows.

Another metric of scale is the number of test runs per day. Figure 8 shows the amount of daily tests in the course of three months, ranging from 300K to as high as 700K. The average daily test runs is around 500K, demonstrating its scale.

We further demonstrate the lab reservation's usage in Figure 9 for a period of recent 4 months. The reservation system can scale to as large as 130K reservations per day. The noticeable drop around day 90 is caused by the ramping down of development activity close to holidays.

6.1.2 Effectiveness: Failed Tests and Errors

The purpose of Netcastle is to catch test failures before bugs manifest in production. Because Netcastle is integrated in both code review stages and continuous deployment stages, we can identify test failures at each stage. Figure 10 shows that in the past four months, Netcastle caught over 500K test failures just during code commit time. Further, the dashed curve represents the necessity of testing during deployment, as it captured issues with other dependent systems. Such failures are orders of magnitude fewer than the failures that manifest at code commit time. Yet, the volume is high—100K.

Not only are the test failures caught by Netcastle abundant but they are also high-quality. Tests in Netcastle are grouped into jobs for efficiency. A Netcastle test job can have 3 outcomes: succeed (all tests passed), test failure, or infra error. Jobs with infra errors are automatically retried and reported as "Infra Error - No Signal" when retry limit is exhausted. This helps developers focus on legitimate test failures that indicate code bugs, rather than noise caused by infra issues.

Figure 11 shows the fraction of cumulative number of Netcastle jobs that fail because of infra error and other reasons (true failures). We observed the fraction of jobs with infra errors is higher than jobs with legitimate test failures. Infra errors increase as the testing framework scales and becomes more complex, while the legitimate test failures grows flatter, demonstrating both the necessity and effectiveness of distinguishing legitimate test failures from infra issues, especially as the physical infrastructure of Netcastle grows.

6.1.3 Optical Switch Reconfiguration

Finally, we present the need for topology reconfiguration in Figure 12. The number of reconfigurations per day ranges from 40 to 160. The high frequency is due to multiple teams co-sharing the testing facility, especially the traffic generator which relies on the optical reconfiguration to inject high volume of traffic at different parts of the topology.

6.2 Five Bugs Kept Out of Production

Netcastle tests integrate with the Meta code commit process. It generates 150K test failures yearly, some of which have prevented catastrophic failures. We share five examples below.

6.2.1 OpenBMC Watchdog Didn't Reset

In January 2021, two changes from the open source OpenBMC upstream were pulled and submitted for review. Each change was tested and landed successfully. However, the two changes were not tested together. With both changes, OpenBMC failed to reset the on-board watch dog timer, and rebooted every 5 minutes. This failure resulted in a lab outage, as all lab devices were rebooting repeatedly and all OpenBMC

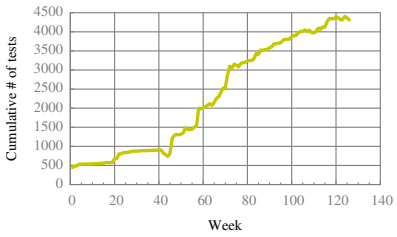

Figure 7: Tests increase weekly.

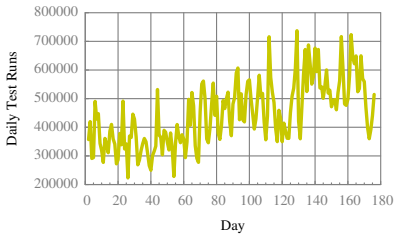

Figure 8: Netcastle test runs per day.

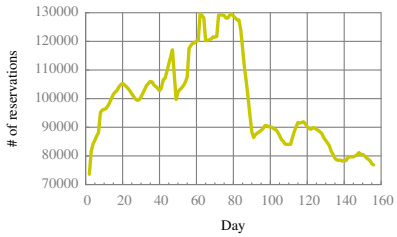

Figure 9: Daily lab reservations.

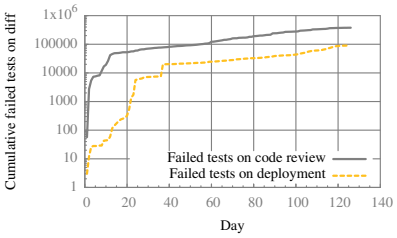

Figure 10: Failed tests by stage.

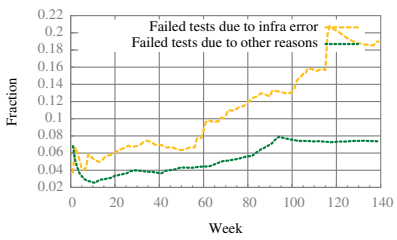

Figure 11: Infra error and test failures.

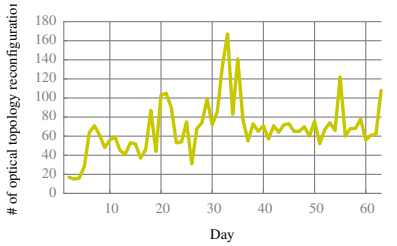

Figure 12: Optical reconfigurations.

tests failed. This bug was caught well before reaching production, and since the test failures were immediate and substantial, it took less than an hour to root cause. Before Netcastle, a similar bug took months to root cause.

6.2.2 FBOSS RIBv2 Redesign Missed Features

The Routing Information Base (RIB) is a critical data structure in FBOSS that stores routes learned from routing protocols and prepares the Forwarding Information Base (FIB) for programming the switch ASICs. Over time, FBOSS has added many use cases and optimizations to the original RIB, including multi-protocol support, fast lookups via radix trees, UCMP, and MPLS. However, the implementation of these features increased the difficulty of adding new use cases and scaling needs. To address this challenge, the FBOSS team initiated a complete redesign of this core data structure. Despite being reimplemented by experienced engineers, massive errors were found in the code: MPLS, mirroring, and traffic engineering did not work as expected. With tests in place, issues were quickly fixed.

6.2.3 Warmboot Packet Loss due to Agent Race

Warmboot is a switch ASIC feature that allows forwarding packets while we we update the FBOSS agent software. Netcastle has the ability to run tests after warmboot from one commit to another, allowing explicit testing of upgrades and rollbacks. In one upgrade, we discovered that a subset of prefixes were experiencing high input discards, and further investigation revealed that a drop rule had been installed in the ASIC but not in the control plane. We also observed that changing the traffic pattern caused previously functioning destination prefixes to fail. We traced the issue to a race condition in the agent software that occurred when adding and removing FDB entries during warmboot with traffic running. A hidden drop rule caused by a race condition would have been extremely disruptive. This issue was challenging to reproduce even in a controlled lab setup; in production, without control over the traffic, it would have been nearly impossible.

6.2.4 Micro-Loop Causing Packet Loss During Drains

"Draining" redirects traffic away from a switch that is being taken offline for maintenance. Our tests check that packet loss during drains lasts less than 30ms. However, during one incident, packet loss continued for 300ms. The root cause of the issue was a micro-loop that formed during routing convergence. When the switch is drained, its transit neighbors are expected to send BGP withdraw messages to the spine to stop attracting traffic. However, the routes were unprogrammed in the Forwarding Information Base (FIB) before the BGP withdrawal was sent to the spine. As a result, the traffic coming from the spine was sent back by the transit switches. Packets were discarded when their time-to-live (TTL) expired.

If this issue had not been detected in the lab, it could have caused significant problems during drain operations, which are performed hundreds of times a day. This black-hole of traffic for 300ms will be handled differently by applications, with responses ranging from normal retransmission and congestion response to abandoning connections and failing queries. The small scale and duration makes such events difficult to detect and troubleshoot in a production environment. Fortunately, using the granular metrics provided by Netcastle, we were able to consistently reproduce and test various hypotheses.

6.2.5 "Factory Reset" Backbone Lab Devices

"Provisioning" is the process of configuring devices (switches or servers) to an operational state. Recently, a faulty provision-

ing workflow led to network devices entering a mode called zero-touch-provisioning: essentially a factory reset state. This state lacked the necessary boot configuration required for operation within our backbone network. This problem was first observed in the Netcastle infrastructure since we were testing the new workflow in the lab. However, we promptly blocked the code change responsible for the misconfiguration to prevent any impact on production. If this issue had not been discovered in the Netcastle and the new provisioning workflow had been rolled out, it would have resulted in a complex and lengthy recovery process. Currently, in EBB, we perform per-plane rebuilds, which involves rebuilding a network shard of 42 devices in parallel. In the worst-case scenario, the bug would have brought down 42 backbone devices, resulting in the loss of hundreds of Terabytes of capacity.

These examples illustrate Netcastle's capability to identify both production tooling issues and device software bugs.

7 Experiences

7.1 Test Signal Quality Is Paramount

Obvious to state but infuriating in the details, the testing signal, e.g., did a given test 'pass' or 'fail', as relayed to the developers/automated tooling must have extremely low false positive and false negative rates. A false negative, say due to a failure in the infrastructure rather than a regression in the code, is often hard to distinguish from a rare bug which can cause developers to waste significant time. A false positive, say when the test runner fails to catch that a test is actually failing but reports as "pass", allows buggy code to be pushed deeper into the continuous deployment ("CD") pipeline. Deeper in the CD pipeline has more noise and variance and so bugs there are harder to replicate, isolate, and debug, and ultimately risk being pushed into production. Worse, any noise in the test signal degrades developer confidence and can cause them to ignore test results or make them less inclined to write more tests in the future. The key insight is, intuitively similar to the famous Nyquist-Shannon sampling theorem, if one wants to catch a code that fails at a certain rate, e.g., 1 in 1000 run race condition, then the corresponding test signal must have proportionately much higher sampling rate/stability (e.g., 1 in 100k test runs result in a false positive). Getting the test infrastructure to this level of engineering confidence is no small effort. Most of the experiences shared below derive from this main point about test signal quality.

7.2 The Test Infrastructure Itself Must Be Testable

The potential for test signal noise is highest when there are bugs in the Netcastle software itself, such as in critical components like the runner. To prevent this, Netcastle has 100% unit test coverage and a complex system of tests that run with each release. In one incident, a bug in the centralized resource reservation system caused conflicting tests to produce a large amount of noise and risked the project's reputation in its early phases. This testing bug resulted in a significant loss of developers' trust, which put the project at risk.

7.3 Respect the Testing Pyramid

The Testing Pyramid [36] is a widely accepted testing abstraction that emphasizes the importance of writing simple tests with fewer moving parts. The number of tests for each type (e.g., unit, component, system, end-to-end) should be inversely proportional to their complexity, creating a pyramid shape. This principle has significant implications for Netcastle and network testing. Despite networking being a distributed system, writing distributed tests directly, such as designing a test with 20 different routers to verify BGP convergence properties, introduces significant testing noise. Therefore, Netcastle and developer teams constantly simplify tests to reduce noise and test the same logic with fewer moving parts, further down the pyramid. An example of simplifying tests is the FBOSS hardware test, where instead of using external devices to source/sink packets for data center switch functional testing, individual switch ports are put into a temporary loopback mode. This replaces an earlier setup where switches were connected to physical servers and test packets were sourced/sinked via remote procedure call (RPC). The move from a distributed system with complex RPC calls to a single machine with only local processing reduced testing noise and eliminated the need to maintain servers.

7.4 Infrastructure vs. Code Errors

When infrastructure errors occur in Netcastle, effort is put into distinguishing them from code errors to avoid disrupting developers. Skipped tests can be retried on different devices to produce a more reliable result, and infrastructure errors are handled by the Netcastle oncall team while code errors are sent to individual developer teams. Lab doctor tests are used to check device health before and after each run, and any changes in device health can help determine whether an error is due to infrastructure or code issues. However, there is no perfect way to separate infrastructure from code errors, and manual triage is often required to determine the root cause.

7.5 Do Not Under Estimate/Engineer Complexity of Testing Workloads

Many of the initial false steps in the project can be attributed to under-estimating and ultimately under-engineering the Netcastle software systems. The team initially underestimated the need for production-level performance, monitoring, and alerting in the testing system, assuming that running tests were not

production workloads. However, the team realized the importance of testing after encountering bugs in the Netcastle code that delayed test signal for code needed to fix a bug in production. The root cause was the lack of a service-level agreement (SLA) for test result return time. Now, Netcastle has a firm SLA with each developer team, and violations trigger SEV alerts [9] and standard production system management procedures to prevent future incidents.

7.6 Manage Latency vs. Completeness

As test coverage grew, we found that not all tests could be run and reported to developers in a timely manner, such as within 30 minutes while they waited for test results to see if their code could be deployed. To compromise, we created a hierarchy of tests that ran at different time scales and with varying levels of coverage. This model mirrors the testing pyramid, with tests categorized as quick and low coverage, periodic and medium coverage, and comprehensive and high coverage. This classification evolved over time due to the need for faster test signal and the growing number of tests, making manual classification impractical. Had we used hierarchies from the beginning, we could have avoided the need for the clunky one-test-per-class heuristic.

7.7 Test Equipment Should Not Depend On Other Test Equipment

In our earlier test labs, we discovered that managing a collection of switches in a standard Clos/leaf-spine topology as individual test equipment can cause interference between tests running on different switches. For instance, tests running on spine switches could unexpectedly disrupt tests on leaf switches and vice versa, leading to incorrect assumptions about network connectivity and causing test failures. Additionally, some leaf nodes had servers attached to them that were only accessible if the leaf switch was running all necessary software, which was not always the case.

To address this issue, we took two steps. First, we designated upstream devices, such as spines connected to leaves, as "production" test gear, meaning they were not reservable and stayed functional during adjacent testing. Second, for developers who wanted to test pairs of gear, we used the concept of an "ensemble" to create atomic units of the desired gear. This prevented a single test device from being reserved, which depended on a test device in someone else's reservation.

8 Related Work

Network Testing Frameworks Testing new networking research ideas is complex and requires consideration of various circumstances such as network size, hardware specifications, and protocols used. Due to the high cost and proprietary nature of network testing frameworks built by owners of large networks, publicly available network testing frameworks based on emulators are becoming more popular. Emulators such as Flexplane [25], NIST Net [4], Mininet [20], and NS3 [28] are widely used tools to emulate large scale networks and test new networking concepts.

In addition to these publicly available tools, there are numerous firms with the business model of providing network testing tools [13]. Some notable firms include, Ixia (Keysight) [34], Forward Networks [11] and Fluke Networks [8]. However, while these tools can be used to detect anomalies in the current network, they are not built with the purpose of testing newly added features, thus they do not naturally support incremental testing of new features.

Network Testing/Verification Methodology. First of all, there are numerous works on verifying correctness of the network [16, 18, 19, 26, 30]. For example, Reitblatt et al. [27] discusses formal method of verifying the correctness network transitions. Similarly, Veriflow [18] is built to verify invariants of the network in real-time. CrystalNet [22] builds an emulation platform for production. However, these methods only check for issues in the software states, making it hard to detect and report any hardware related issues.

Some articles discuss industry methods for network testing, such as Spirent's best practices outlined in [6]. These practices include Vendor Performance Testing, Network Failure Threshold Testing, Configuration Defect Testing, and PASS Methodology Testing. While these practices offer practical guidance, they do not provide a formal theoretical approach for testing networks, which can make it difficult to quantify testing difficulties.

9 Conclusion

Large network operators have real, but hard to quantify, concerns with managing complexity in their networks. In this paper, we first quantify network testing complexity and share the production challenges. We then present the architecture from the Netcastle "test infrastructure as a service" model as well as lessons learned from over three years of growth. Looking forward, we hope that the research community will leverage our "testing matrix as complexity" model (§2.1) and use it in their research to understand which changes operators are more or less likely to deploy in production. We hope that researchers consider these effects when considering new schemes/research as their choices have real impact on network complexity: the people testing it will have to try to tame that complexity.

This work does not raise any ethical issues.

References

[1] Facebook datacenter network. https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/.

[2] K. Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002.

[3] R. Bush. Into the future with the internet vendor task force a very curmudgeonly view or testing spaghetti: A wall's point of view. *SIGCOMM Comput. Commun. Rev.*, 35(5):67–68, Oct. 2005.

[4] M. Carson and D. Santay. Nist net: A linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, July 2003.

[5] S. Choi, B. Burkov, A. Eckert, T. Fang, S. Kazemkhani, R. Sherwood, Y. Zhang, and H. Zeng. Fboss: Building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 342–356, New York, NY, USA, 2018. Association for Computing Machinery.

[6] S. Communications. Testing the data center network: Best practices. https://www.infopoint-security.de/medien/testing_the_data_center_network-best_practices_whitepaper.pdf, November 2013.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[8] F. Corporation. Fluke networks. https://www.flukenetworks.com/, Jan. 2021.

[9] G. Eason. Incident response @ FB, facebook's SEV process, July 2016.

[10] T. Fang. Introducing openbmc: an open software framework for next-generation system management. https://code.facebook.com/posts/1601610310055392, Mar. 2015.

[11] I. Forward Networks. Forward networks. https://forwardnetworks.com/, 2021.

[12] T. Griffin. RFC4264 BGP wedgies. https://tools.ietf.org/html/rfc4264, Nov. 2005.

[13] S. T. Help. Top 30 network testing tools (network performance diagnostic tools). https://www.softwaretestinghelp.com/network-testing-tools/, Jan. 2021.

[14] HUBER+SUHNER. Polatis series 7000 optical switch. https://www.polatis.com/series-7000-384x384-port-software-controlled-optical-circuit-switch-sdn-enabled.asp, Jan. 2021.

[15] T. Inc. Telescent physical layer switching. https://www.telescent.com/, 2021.

[16] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[17] M. Jimenez and H. Kwok. Building express backbone: Facebook's new long-haul network. https://engineering.fb.com/2017/05/01/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/, May 2017.

[18] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, Apr. 2013. USENIX Association.

[19] D. Kim, J. Nelson, D. R. K. Ports, V. Sekar, and S. Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery.

[20] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, 2010. Association for Computing Machinery.

[21] W. E. Lewis. *Software Testing and Continuous Quality Improvement, Third Edition*. Auerbach Publications, USA, 2nd edition, 2008.

[22] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz. Route flap damping exacerbates internet routing convergence. *SIGCOMM Comput. Commun. Rev.*, 32(4):221–233, Aug. 2002.

[24] J. Networks. How to calculate power margin for fiber-optic cable. https://www.juniper.net/documentation/en_US/release-independent/junos/topics/task/installation/fiber-optic-cable-budget-margin-calculating.html, July 2020.

[25] A. Ousterhout, J. Perry, H. Balakrishnan, and P. Lapukhov. Flexplane: An experimentation platform for resource management in datacenters. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 438–451, Boston, MA, Mar. 2017. USENIX Association.

[26] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Feb. 2020.

[27] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334, 2012.

[28] G. F. Riley and T. R. Henderson. The ns-3 network simulator. In *Modeling and tools for network simulation*, pages 15–34. Springer, 2010.

[29] F. O. Source. Fboss open source. https://github.com/facebook/fboss, 2021.

[30] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu. Debugging p4 programs with vera. SIGCOMM '18, page 518–532, New York, NY, USA, 2018. Association for Computing Machinery.

[31] J. Stone and C. Partridge. When the crc and tcp checksum disagree. *SIGCOMM Comput. Commun. Rev.*, 30(4):309–319, Aug. 2000.

[32] J. Taylor. Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond. In *Optical Fiber Communication Conference*, page W1D.5. Optical Society of America, 2015.

[33] C. Technologies. Calient s series photonic switch. https://www.calient.net/products/s-series-photonic-switch/, 2021.

[34] K. Technologies. Connect and secure your network with keysight. https://www.keysight.com/us/en/cmp/2020/network-visibility-network-test.html, Jan. 2021.

[35] Wikipedia. Multiple source agreement optics. https://en.wikipedia.org/wiki/Optical_module, July 2017.

[36] J. Willet. Evolution of the testing pyramid. https://www.james-willett.com/the-evolution-of-the-testing-pyramid/, Sept. 2016.

A Appendix: Network Testing Is Hard In Theory

In this section, we try to quantify why network testing is hard. Specifically, we define a formal metric for system complexity based on the number of states and allowed state transitions. We then argue that in non-trivial systems, the only way to enumerate the actually allowed state transitions (including bugs, surprising interactions, the halting problem, etc.) is through testing. A testing matrix is a common software engineering term for the multi-dimensional space of all possible interactions in a system. We then introduce the concept of a theoretically complete testing matrix: that is, a conservative/worst case set of conditions/states that if one were to theoretically test all elements of the matrix, the system could be proven by exhaustion to be bug free. We then combine these two definitions to show that network testing is the combinatorial combination of software and hardware systems testing and that by our metrics is uncountably infinite in all but the most trivial of cases.

A.1 Quantifying System Complexity

From our literature review, there does not appear to be a commonly agreed upon method to quantify the complexity of a general system, so we define our own. From computational complexity theory, formally the complexity of an algorithm is the minimum number of bits needed to represent the algorithm and its inputs [7]. Inspired by this definition, we quantify the complexity of a system as:

Definition A.1 ***Measure of System Complexity****: the minimum number bits required to describe all possible states and the allowed state transitions of a system.*

Consider a system with n different states where the allowable state transitions can be (possibly inefficiently) represented as an $n \times n$ matrix of bits denoting "state transition $i \rightarrow j$ is allowed". By this definition, naively this system would require $\log(n)$ bits to describe the possible states plus n^2 bits to describe the state transitions or $\log(n)+n^2$ bits to capture this system's complexity. However, if this system's allowed state transition matrix was sparse, summarizable with few rules or otherwise compressible, then the minimum number of bits required to represent it could be much smaller than n^2. From this definition, and matching our intuition, a system gets more complex as it adds more states or as the allowed state transitions become harder to describe.

This definition captures a critical combinatorial property of system complexity: when two or more smaller systems ("components") are coupled into a larger system, the complexity

increases multiplicatively. That is, if one couples two systems that have m and n states respectively, then the resulting system C has $n \times m$ different possible states and thus $(n+m)^2$ possible state transitions. As above, the state transition matrix of C may still be compressible and thus the minimum bits needed to describe them may be smaller than $(n+m)^2$ in practice. That said, as the number of coupled components increase, the ability to succinctly summarize the allowed state transitions of the larger becomes harder, e.g., it requires a combinatorial multiplication of rules.

Note crucially that "allowed state transitions" are the ones allowed by the actual deployed running implementation and not limited to the transitions intended by the original design intent. A network architect may believe they have designed a simple system with few possible states and limited state transition rules ("no persistent loops", "control plane processes never reach the segmentation fault state", etc.). However, the implementation of the actual system including bugs, hardware failure, environmental effects, unforeseen vendor incompatibilities, etc. may in fact be much more complex. Thus, for any large, practical system, there may be unintended and surprising state transitions: i.e., the allowed state transitions an emergent property of the coupling of many systems. More so, the "halt" state is important to any system, so enumerating the states that can transition to 'halt' is directly equivalent to the well-known and undecidable halting problem. As a result, we claim that the only way to even approximately enumerate possible state transitions, and thus quantify the system's complexity, is through providing specific inputs and testing which state transitions are and are not actually allowed.

A.2 Theoretically Complete Testing Matrix

We define a *theoretically complete testing matrix* as a test matrix where if all possible tests were evaluated, then the components/system under test would be proven to be correct by exhaustion. As we will show, this is often not a practical tool as many of the dimensions of the theoretically complete testing matrix will be infinite in practice, we believe it is still a useful vehicle for quantifying network complexity.

There are two properties of a theoretically complete test matrix:

1. It makes no simplifying assumptions about the implementation of the components under test. As a result:

2. Similar to Definition A.1, each dimension or component added to the system increases the size of the testing matrix multiplicatively.

Quantifying Software Systems Testing First, let's consider constructing the simplest possible theoretically complete test matrix. Consider a toy, single threaded user-space process (the "process under test") that has n different possible states and the only information it stores is the value of the current state. The test matrix for such a software system is $O(n^2)$, as the programmer needs to test all possible state transitions to make sure each works as expected. Note that it is important to test not only the valid state transitions (functional tests) but also test the invalid transitions (negative testing) to verify that they are correctly prevented. The current state and the next state form the first two dimensions of our testing matrix and each cell in the matrix is a single bit representing the pass/fail result of running the test.

Next, let us relax our toy application assumption by assuming that it keeps non-trivial information across state transitions (e.g., allocates memory, opens a file, etc.) such that we now need to consider the history of all state transitions - another dimension to our matrix. This new dimension is potentially infinite in size (all permutations of all possible paths through the state diagram) and practically infinite for long running programs. While seemingly abstract, the history of all possible state transitions is where we catch the most common types of bugs: memory corruption, resource leaks, privilege elevation, etc. Then, let us further relax our single-threaded assumption by allowing the process to use threads, asynchronous I/O, or other form of scheduling non-determinism. This adds scheduling order as another dimension to our test matrix which is exponential in the number of threads for each state change. Now relax the single process assumption and allow the process under test to have non-trivial communication with both the kernel and other processes (local or remote). Thus, the theoretically complete testing matrix for a distributed system of software processes is the multiplicative product of each software component in the system. In other words, if one wants to be absolutely certain to prevent bugs in a non-trivial multi-threaded distributed system, they need to write sufficient tests that cover the multiplicative product of: all possible state transitions $\times$ all possible state transitions histories $\times$ all possible schedule orderings $\times$ all possible states of the other components in the system.

Quantifying Hardware Systems Testing Hardware systems have a superset of possible bugs as compared to software systems and thus their testing matrices are correspondingly larger. Consider the logical design of a single integrated circuit or component. Often before construction, such components are modeled in a software simulator, and similar to software testing, are tested across all possible inputs/outputs and state transitions. As a result, all of the above analysis for software test matrices applies to hardware testing matrices as well. Additionally, hardware systems must test for manufacturing repeatability (e.g., if a component requires a physical size tolerance of $\pm X\%$, what is the distribution in practice) and manufacturing process changes (e.g., different vendors of solder, electronics components). Hardware systems must test under different operating conditions (e.g., temperature, humidity, vibrations, electromagnetic interference) and typically ship with specified operating tolerances (e.g., run between

50-85 degrees *F*). All of these aspects add new dimensions to the theoretically complete testing matrix.

A.3 Testing Network Systems

Computer networks are, by definition, transitively coupled hardware and software systems. That is, state from any one node could in theory be transferred to any/every other node in the system. As a result, the complexity (Definition A.1) and theoretically complete testing matrix of a network is the combinatorial explosion of all possible testing matrices of all connected hardware and software systems. More than just a theoretical concept, large network outages happen in practice because of rapid distribution of state from a single failed component. A typical example is a single faulty electrical component (e.g., a capacitor) in one optical interface on a single switch can cause a shared link to flap up and down, which various control plane software systems (e.g., spanning tree protocol in L2, BGP in L3) then propagates the new forwarding state to other nodes in the system, which each try to adapt to the new state based on local policy. The result of these transitive interactions, despite decades of experience and armies of network engineering experts, are often surprising and detrimental [12, 23].

Unique to networking, practical deployments are often the worst case for many software and hardware testing scenarios. For example, device up-time is typically long (over a year in some routers) so the set of all possible paths through the state machine is in practice large. Networking gear is expected to operate in a variety of environments on the edges of their operating tolerances from (in theory) climate controlled data centers to road-side optical devices to the WiFi access point under your desk that collects dust bunnies. Despite formal protocol standards which should in theory ensure interoperability, in practice all-to-all interoperability tests are still required between different implementations and many deployments effectively require "bug-level" compatibility. Last, for reasons of compatibility as well as practical software engineering and business realities [3], networking devices tend to monotonically accumulate features over time (e.g., Apple talk support, RIP, etc.). All of these elements add to the complexity of network systems and thus increase the complexity of testing these systems.