Automating Failure Testing Research at Internet Scale

Peter Alvaro[1] Kolton Andrus[2] Ali Basiri[3] Lorin Hochstein[3] Casey Rosenthal[3] Chris Sanden[3]

[1]*UC Santa Cruz* [2]*Gremlin, Inc. (formerly Netflix, Inc)* [3]*Netflix, Inc*

Submission Type: Experience

Abstract

Large-scale distributed systems must be built to anticipate and mitigate a variety of hardware and software failures. In order to build confidence that fault-tolerant systems are correctly implemented, Netflix (and similar enterprises) regularly run "failure drills" in which faults are deliberately injected in their production system. Existing failure testing approaches either explore the space of potential failures randomly or exploit the "hunches" of domain experts to guide the search—the combinatorial space of failure scenarios is too large to explore exhaustively. Random strategies waste resources testing "uninteresting" faults, while programmer-guided approaches are only as good as the intuition of a programmer and only scale with human effort.

In this paper, we describe how we adapted and implemented a research prototype called lineage-driven fault injection (LDFI) to automate failure testing at Netflix. Along the way, we describe the challenges that arose adapting the LDFI model to the complex and dynamic realities of the Netflix architecture. We show how we implemented the adapted algorithm as a service atop the existing tracing and fault injection infrastructure, and present preliminary results.

1 Introduction

Netflix and similar enterprises operate at a scale at which failures such as machine crashes and network partitions are the rule. In order to provide an "always on" experience to customers, software for Internet-scale companies must be be written to anticipate and work around a variety of error conditions that are unlikely to arise except at large scale. It is difficult to ensure that such fault-tolerant code is adequately tested, because there are so many ways that a large-scale distributed system can fail.

The practice of Chaos Engineering [6]—which emerged as a discipline to tackle resilience of these large-scale distributed systems—is on the rise [29, 23]. Formally, Chaos Engineering is defined as "experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production." Engineers create frameworks that automate failure injection, usually on live traffic. Emerging workshops such as Chaos Community Day [1]—in which top Internet companies share experience designing and implementing chaos experiments to improve the resilience of their systems at scale—underscore the growth of this trend.

Chaos engineering reflects a cultural trend within the software industry away from coordinated design and architecture, monolithic applications, and top-down engineering toward coordination of API boundaries, microservice architectures, and flattened engineering hierarchies. As the complexity of these loosely coupled architectures increases, reliance on automated tooling to provide end-to-end tests for business-critical assumptions about the system becomes unavoidable. Confidence in the system's behavior should be manufactured by experimenting with worst-case failure scenarios in the production, scaled-out system.

Building a production fault injection infrastructure is only the first step towards maintaining fault-tolerant systems. The space of distinct *failure scenarios*—combinations of faults across a distributed system—that such an infrastructure can test is exponential in the number of potential faults. Exhaustive search is intractable; the fault injection infrastructure must choose a search strategy to explore this massive space of possible executions. To the best of our knowledge, all current failure testing solutions use one or both of the following strategies:

1. Random search, in which the fault injection infrastructure chooses failure scenarios arbitrarily. The principal advantages of random search are its simplicity and generality. However, random strategies are unlikely to discover "deep" failures involving combinations of different instances and kinds of faults. They also waste resources and time by exploring failure scenarios that are redundant, or that could be proven to be incapable of triggering a user-visible error.

2. Programmer-guided search, which leverages the in-

tuition of domain experts to guide the search through failure scenarios. In a microservice architecture like that of Netflix, individual services are owned by small engineering teams. Within each team, component-specific domain expertise can be exploited to generate local heuristics. This approach has the advantage that by prioritizing certain "deep" paths through the space of failures, it can drive the search into unlikely but severe corner cases for the given component. Unfortunately, programmer-guided search is fundamentally unscalable, because every component requires a domain expert to invest their time to encode their expertise in a search heuristic.

Imagine if we had perfect information that allowed us to understand exactly how systems like the Netflix backend produce "good outcomes" (for example, by providing a satisfactory response to user requests). We could then transform the very open-ended question "could a bad thing ever happen?" into a set of narrower and more targeted questions: "how did *this* good thing happen (and what could have gone wrong along the way)?" Because (by assumption) fault-tolerant systems employ *redundancy* in various forms to guard against failures, the answer to this "how" question will often reveal a variety of alternative computations that can produce the good outcome. These in turn can help us to prune the space of fault injection executions that we need to consider. This approach to automatically driving a fault injection infrastructure is called lineage-driven fault injection (LDFI) [5].

In this paper, we explain how we implemented LDFI as a snap-in microservice in the Netflix infrastructure, leveraging the existing tracing and fault injection services. We describe the challenges that arose adapting the idealized model of the research prototype to the rigid and imperfect realities of a large-scale distributed system, and how we overcame them.

The paper is organized in the following way. Section 2 presents the LDFI approach and its embodiment in the initial research prototype. Section 3 describes the failure testing infrastructure at Netflix as it existed when we began the project. Section 4 details the challenges that arose implementing a production version of LDFI; Section 5 takes each problem in turn and shows how we solved it. We present some preliminary results in Section 6, and close with a discussion of related work (Section 7) and lessons learned (Section 8).

2 Lineage-driven fault injection

Lineage-driven fault injection (LDFI) is a technique for guiding the search through possible fault injection scenarios [5]. The LDFI prototype system (called Molly) takes as input a distributed program written in an Dedalus [4] (an executable specification language based on Datalog), a correctness specification, program inputs and bounds on execution length, and simulates the program's distributed executions under a variety of faults. Execution of Molly terminates in one of two cases:

1. A violation of the invariants described in the specification is found. Molly then returns a trace visualization of the execution, along with the faults that drove the system into an invalid state. Programmers can use this visualization to identify the root cause of the bug.

2. Molly exhausts the execution bounds without discovering an invariant violation. In this case, the submitted program is "certified" as free from fault-tolerance bugs given the execution bounds and program inputs.

The LDFI approach is based on two key insights. The first is that *fault-tolerance is redundancy*—a program or system is fault-tolerant precisely if it provides enough alternative ways to obtain an expected outcome that it is resilient to some pre-defined set of fault conditions. If we had perfect information about all of the alternative computations a system provides, we could determine what faults it can tolerate—or conversely, identify failure scenarios that could prevent it from succeeding. The second insight is that instead of starting from initial states and exhaustively searching the space of possible executions, a better strategy for quickly identifying fault-tolerance bugs is to *start with successful outcomes and reason backwards*, from effects to causes, in order to understand whether some combination of faults could have prevented the outcome.

2.1 Lineage

LDFI uses data lineage [7, 9] to simultaneously exploit both insights. It begins with a correct outcome, and asks *why* the system produced it. This recursive process of asking why questions (which we illustrate below) yields a lineage graph that characterizes all of the computations and data that contributed to the outcome. By doing so, it reveals the system's implicit redundancy by capturing the various alternative computations that could produce the good result.

For example, a correctness property for a distributed storage system might require that "all acknowledged writes are durably stored." Hence an execution in which a write is durably stored is a witness to the correctness property. Working backwards from this good outcome, we ask *why* the write was durable. Since the storage system used replication to guard against node failure, there

are multiple reasons why the write is durable: namely, because it is stored on some replica $RepA$ *and* because it is stored on $RepB$. We continue to recursively ask why questions: e.g., the write is stored on $RepA$ for multiple reasons, because the client made multiple broadcast attempts. When we finish "unrolling" the explanation of why the write is durable, we end up with a lineage graph like the one shown in Figure 1.

LDFI must now reason about whether any combination of faults could prevent that good outcome from occurring. A naive pass of the lineage graph tells us that the set of contributing events that could fail is $E \equiv \{RepA, RepB, Bcast1, Bcast2\}$. Hence the space of possible combinations of faults that we could explore is 2^E—the power set of E, containing 16 elements. However, closer inspection of the graph reveals that not every element of 2^E is *interesting*. For example, an execution in which faults are injected in $RepA$ and $Bcast1$ is uninteresting, because we already know that even under those faults the system can produce the good outcome via $Bcast2$ and $RepB$.

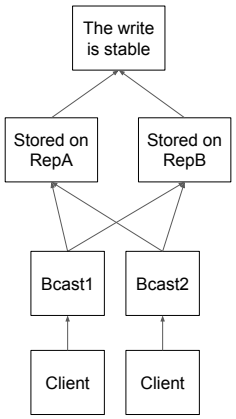

Figure 1: A simple lineage graph

2.2 Boolean encoding and solving

We can make this reasoning more formal by extracting from the lineage graph the set of distinct *paths* from leaves to root; each path corresponds to an alternative computation that is sufficient to produce the outcome. Within each path, the failure of *any* node invalidates the whole path. Consider the path from the $Client$ to the root that transits $RepA$ and $Bcast1$. It required $RepA$ *and* $Bcast1$ to succeed; hence failing *either* is sufficient to invalidate the computation. We can encode the conditions under which this computation could fail as a disjunction of propositional variables:

$$(RepA \vee Bcast1)$$

Preventing the good outcome, however, would require us to find a combination of failures that invalidates *all* of the alternative computations. We can encode this as a conjunction of disjunctions of propositional variables (i.e., a formula in conjunctive normal form):

$$\begin{aligned}&(RepA \vee Bcast1)\\ \wedge\ &(RepA \vee Bcast2)\\ \wedge\ &(RepB \vee Bcast1)\\ \wedge\ &(RepB \vee Bcast2)\end{aligned}$$

The solutions to this boolean formula represent sets of faults that we should test via fault injection. In particular, we are interested in the minimal solutions—those that do not contain other solutions. These are:

$$\{RepA, RepB\},$$
$$\{Bcast1, Bcast2\}$$

2.3 Alternating execution

A fault-tolerant program should, in the presence of faults such as component failure and message loss, attempt to achieve its expected outcome via some other means. Hence a solution to the formula described above does not necessarily indicate a bug, but rather a *hypothesis* that must be tested via fault injection.

By injecting a set of faults corresponding to a solution to the Boolean formula, the fault injection framework will produce one of two outcomes:

1. The system fails to produce the expected outcome, indicating a fault-tolerance bug. Execution terminates, and the user is presented with a visualization of the lineage graphs.

2. The system succeeds in producing the expected outcome. This indicates that it has revealed an alternative strategy for obtaining the outcome. The lineage graph from this execution should be merged with the current one, and a new formula should be extracted and solved.

In this way, LDFI alternates between concrete executions and symbolic solving until either a bug is discovered or the formula is determined to be unsatisfiable. In the latter case, LDFI "certifies" the system as free from fault-tolerance bugs for the given configuration and input.

The basic system architecture of LDFI is depicted in Figure 2. The system is seeded with a *successful outcome*.

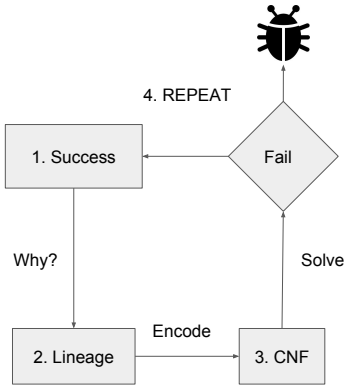

Figure 2: Overview of LDFI.

Recursively asking "why" questions about how the system obtained that outcome yields a *lineage graph*. This lineage graph is encoded into a Boolean formula that is then *solved* to generate failure hypotheses, each of which is tested via a sequence of *replays* of the initial execution in which faults corresponding to the hypothesis are injected. This process is repeated until either a fault tolerance bug is identified or the system exhausts its hypotheses.

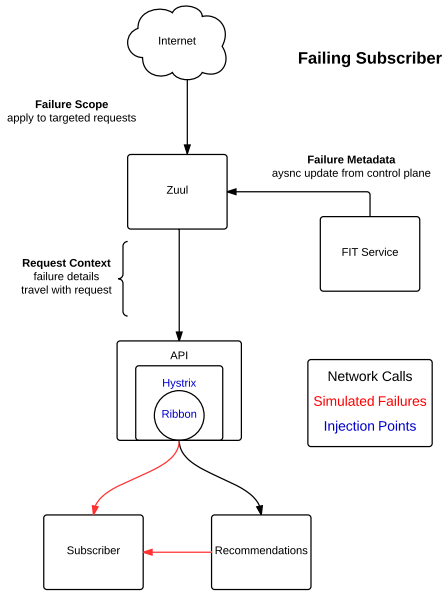

Figure 3: A a simulated failure, demonstrating inflections points where failure can be injected.

3 Failure Testing at Netflix

FIT, "Failure Injection Testing," is a platform within Netflix that simplifies fault-injection with a high degree of precision for what component is failed and which user requests will be impacted. FIT also allows the propagation of failures across the Netflix microservice architecture in a consistent and controlled manner.

3.1 Failure Scope

The Failure Scope is the potential impact that a failure test, or experiment, could have. This is measured in terms of customer impact, from a single customer to all incoming traffic; the service impact, from a single host or service to all hosts; or other attributes of the incoming request. Deliberately causing failures in production is a potentially risky strategy; the failure scope of an experiment keeps users of FIT mindful of the external consequences of a failure drill by understanding its potential "blast radius."

Simulating failure begins when the FIT service pushes failure simulation metadata to the proxy server, Zuul, as shown in Figure 3. Requests matching the "failure scope" are decorated with metadata describing a particular failure scenario. The metadata may describe an added delay to a service call, or remote service call failure. Each "injection point" with which a request interacts checks the request to determine if there is a failure for that specific component. If found, the injection point simulates that failure appropriately.

3.2 Injection Points

The Netflix infrastructure provides developers with several key "building block" components:

1. **Hystrix** is used to isolate failures and to define fallbacks.

2. **Ribbon** is used to communicate with remote services.

3. **EVCache** provides access to cached data stored in Memcached [14].

4. **Astyanax** provides interfaces for durable storage of data in Cassandra [21].

Each of these layers defines an inflection point at which is is possible to inject simulated failures. These layers interface with the FIT context to determine if a given request should be impacted. The failure behavior is provided to the appropriate layer, which then individually determines

how to emulate that failure in a realistic fashion: e.g. sleep for a delay period, return a 500, throw an exception, etc.

3.3 Failure Scenarios

Whether recreating a past outage or proactively testing the loss of a dependency, the failure testing service needs to know what could fail. Netflix leverages an internal tracing system (similar to Dapper [28]) that provides the ability to trace requests through their interactions with distributed services and find all of the injection points along the path. These are then used to create failure scenarios—sets of injection points into which faults can be injected in the same execution. A commonly exercised scenario is the set of "critical services" representing the minimum set of services required to stream video.

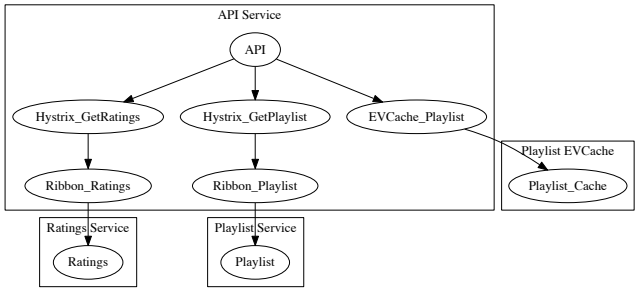

Figure 4: A decorated call graph generated describing the services that participated in servicing a user request. The boxes represent Netflix services, while the oval nodes represent potential fault injection points identified by FIT.

4 Challenges

LDFI has been shown to be effective at discovering bugs related to fault-tolerance in low-level, "classic" protocols (e.g. three-phase commit) as well as modern infrastructure (e.g. Kafka), using orders of magnitude fewer executions than random fault injection [5]. This made it an attractive alternative to the brute-force and engineer driven methods used to drive the failure testing infrastructure at Netflix. Unfortunately, several limitations of the prototype made it unsuitable for use in its current form.

In this section, we describe the assumptions and requirements of the LDFI prototype that were practical barriers to its deployment. In the subsequent section, we detail how we overcame these limitations in the production implementation.

4.1 Language

The Molly prototype requires that distributed systems be implemented in the Dedalus language. The reasons for this restriction are largely historical. The research team that developed LDFI intended for it to provide tool support for Bloom [3], whose semantics are based on Dedalus. Rule-based languages such as Dedalus also make it trivial to collect fine-grained data lineage during execution.

While it is not uncommon for verification systems to require that programs be specified in a custom language [17, 33], this requirement was not acceptable at Netflix. First, there were simply too many applications to port to Dedalus. Moreover, because Netflix executes sandboxed code provided by partners, the source code of applications is sometimes unavailable.

As a consequence, we had to look for another source of lineage data beyond the program text.

4.2 Lineage: granularity and redundancy

Fortunately, the tracing service described in Section 3.2 records service interactions as *call graphs*. A call graph is shown in Figure 4. At a high level (and at a glance), such a trace resembles a lineage graph. It describes how the API tier (the root of the graph) responded to a client request; every service that was required to satisfy that request is represented in the graph. To a first approximation, call graphs differ from the rich lineage structures that could be generated from Dedalus executions principally in their *granularity*. That is to say, a call graph characterizes how a collection of services contributed to a system outcome, while LDFI's data lineage characterized how individual data elements and fine-grained computation steps contributed to an outcome. Because we were willing to sacrifice some precision in order to improve the performance of automated failure testing, it seemed that call graphs could stand in for lineage, freeing us from the dependency on Dedalus as an implementation language.

Recall, however, that LDFI (as described in Section 2) aggressively pruned the space of executions it needed to explore using fault injection by considering explicit *redundancy*, which was revealed in individual graphs like the one shown in Figure 1. Call graphs such as the one shown in Figure 4 capture no redundancy. In fact, if we were to apply the LDFI approach naively to that call graph, we would need to explore the power set of {`API,Ratings,Playlist,Playlist Cache`}, yielding the brute force search strategy described in Section 3.

Of course, a site such as Netflix cannot provide high availability and low latency for user requests without utilizing redundancy in a variety of forms, including caching, active replication, and automatic as well as operator-assisted failover. These forms of *implicit* re-

dundancy are not explicitly revealed in the call graph structure—hence our challenge was to identify and reify them into our model.

4.3 Identifying successful outcomes

As we discussed in Section 2, LDFI works backwards from a *successful system outcome* in order to explore the space of faults that could have prevented it. In the prototype, a successful outcome was merely a data item; a record in a table.

The aforementioned heterogeneity of the applications running in the Netflix ecosystem made it challenging to identify an appropriately general measure of success for individual service interactions. Because Netflix presents a REST API to external clients, we explored the possibility of using the HTTP response code as an indication of the success of the overall interaction. Unfortunately, different applications use the return code inconsistently. It was not uncommon for API calls that produce a client-visible error to nevertheless return a successful status code ("200 OK").

4.4 Replayability

As Figure 1 indicates, LDFI requires individual interactions with a distributed system to be *replayable*. This is unavoidable; not all hypotheses generated by the solver will produce a user visible error, so to ensure that bug identification is sound each hypothesis must be tested via fault injection.

Netflix does not have a production-scale testing environment in which to carry out speculative replay of service interactions. As we described in Section 3, all failure testing is carried out in production using real user traffic.

We explored the possibility of recording user-generated interactions and their traces, and later replaying them to test different failure hypotheses directly in the production environment. Unfortunately, the dynamic nature of the microservice architecture at Netflix made this approach impossible. First, the internal state of individual services is constantly changing as a consequence of both asynchronous updates and user interactions. Worse still, due to software releases the *versions* of various services can change at any time—there is no notion of a "consistent cut" of system-wide software versions, as each microservice evolves independently. Finally, because user interactions with the services are not necessarily idempotent, replay itself may effect the internal state of services.

5 Solutions

In this section we address each of the challenges enumerated in Section 4 in turn. While we sought to preserve the spirit and simplicity of the LDFI approach as much as possible, we had limited flexibility with respect to changing details of the Netflix production infrastructure. As a consequence, we often found ourselves adapting the idealized model of LDFI to the "ugly" realities of the existing system, rather than trying to "boil the ocean" by making the system more closely resemble the model. Luckily, with a bit of creativity and a great deal of flexibility we succeeded in overcoming each of the challenges. We hope that the solutions we describe will prove useful for future adaptations of LDFI to production infrastructure.

5.1 Measuring Success

What does "success" mean in real life? In a production system, it has a real and specific meaning: did the system work correctly for the customer? For the typical customer interaction with Netflix, this means the ability to browse the catalog, view video details, and stream the video. These interactions occur both within web browsers and on devices (such as the Xbox, Playstation, etc). We capture and report back metrics about these (and other) key customer interactions in order to gain insight into the customer experience. From these "Device-Reported Metrics" we are able to determine whether the customer saw an error or had a successful interaction.

We tap into a stream of these device-reported metrics in order to capture the error information related to customer requests that were decorated with failure. We either store "success", or meta-information about the error (useful for later debugging). For each experiment run, we impact several requests so that we can filter out potential false positives. We only mark an experiment as finding a bug if greater than 75 (percent) of the requests result in a failure. Additionally, we ensure that the desired failure actually occurred (as in our non-deterministic world, not every interaction will execute in the same manner).

Lastly, we encountered some scenarios in which no data would be captured from the device-reported metrics for a given experiment. After some debugging, we found that certain injected failures broke not just their request, but the mechanism for reporting errors as well. Therefore we treat a lack of device reported metrics as a failure. This may not be a true failure (i.e. if someone pushes bad code which prevents capturing device reported metrics, etc), but we would rather investigate it as if it were a real failure than ignore it.

5.2 Replay

The LDFI methodology requires that requests be replayed in order to test different failure hypotheses, but the Netflix infrastructure provided no practical replay capacity. To circumvent this obstacle, we *simulate* replay by treating

all user requests that cause the same back-end behavior as if they were replays of a single, canonical request. This required us to define a finite set of equivalence classes representing distinct interactions with the services provided by Netflix, and, for each user request, to *predict* the class to which it belongs. This section describes how we formulated and solved these problems.

5.2.1 Request Classes

All user requests are unique, but intuitively there are only a finite set of distinct service interactions that a request could stimulate. Before we could tackle the problem of predicting class membership for user requests, we had to enumerate this set of abstract "interactions." More formally, user traffic provides an infinite set of requests R, and Netflix's tracing infrastructure provides a function $\texttt{trace} : R \rightarrow T$, where T is a (possibly infinite) set of concrete traces. We needed to define an equivalence relation $\sim$ that gives rise to a set of *request classes*

$$C \equiv \{ \ \{x \in T | x \sim t\} \ | t \in T\} \tag{3}$$

We want to choose $\sim$ in such a way that the cardinality of C is finite and manageable, but large enough that it captures the variety of interactions that are possible for users of the system. Critically, we had to ensure that for any two requests r and r', $\texttt{trace}(r) \sim \texttt{trace}(r')$ if and only if any combination of failures that prevents r from returning a valid response also prevents r'. Informally, two requests are similar if the interactions they cause are sensitive to the same faults.

As we discussed in Section 4.2, call graphs can be easily generated from system traces. Call graphs provide a useful abstraction over the "interactions" that occur as a consequence of user requests: they record which services (drawn from a finite set S) participated in providing a response to the user. The structure of a call graph also records dependency information (i.e., which services were clients and which were servers in directed acyclic graph of API calls) that wasn't required for our purposes. As a trivial example, the services $\{A, B, C\}$ could participate in a variety of different graphs, but the distinctions between these graphs are uninteresting because any faults that affect A, B or C could affect the user-visible response generated by any of the interactions. We write $\texttt{callgraph}(r)$ to denote the call graph produced by request r in a fault-free execution, which we can ascertain directly from system traces because they record both requests and the generated call graphs. Given a call graph $g = (V, E)$, we write `nodes(g)` to denote the set of graph nodes $V \subseteq S$.

Define a function

$$\texttt{interaction} \equiv nodes \circ callgraph \circ trace \tag{4}$$

Note that $\texttt{interaction} : R \rightarrow 2^S$—that is, `interaction` is a function from requests to sets of services. Then we define $\sim$ as

$$\forall r, r' \in R \ \ r \sim r' \iff \texttt{interaction}(r) = \texttt{interaction}(r') \tag{5}$$

Intuitively, two requests belong in the same class if they "light up" the same set of services. Note that we assumed `trace` to be a deterministic function. This is not always the case in practice: call graphs can record a number of non-deterministic effects such as cache misses. We describe below how we work around this difficulty.

5.2.2 Learning Mappings

Because the function `trace` is effectively implemented by the Netflix production infrastructure, it is not available at the time a request arrives at Zuul. Hence as a surrogate for `interaction` (which depends upon `trace`) we must learn a function $f : R \rightarrow 2^S$—that is, a function which, given information known at the time a user request is admitted into the system, predicts the unique set of services that will participate in serving the request. Given a request $r \in R$, we write $\texttt{attrs}(r)$ to denote its set of attributes, such as its URI, device type and query string parameters. Due to the possible run-time nondeterminism of `trace` described above, we model f as a partial function: it produces a defined value only when the classifier predicts a request class with high confidence.

To learn f, we pose a supervised learning problem. Our training input is drawn from the set of production traces, each entry of which associates a request $r \in R$ with a trace $t \in T$ (essentially providing information about how the infrastructure implements `trace` via input/output examples). The features are drawn from request attributes: $F \subseteq attrs(r)$.

We explored two formulations of the classification problem. First, we canonicalized elements of 2^S by sorting them lexicographically and combining them into a large string, and posing a single-label classification problem to predict that string for each request. We also investigated posing a multi-label classification problem [30], in which each element $e \in 2^S$ is treated as a label and the classifier attempts to predict sets of labels for each input. In the end, we used the single-label classifier for the first release of the LDFI service, but are continuing to investigate the multi-label formulation.

5.3 Lineage

By definition, a fault tolerant system provides multiple *alternative* computations that can achieve its expected outcomes—it is precisely by doing so that it can mask

faults that occur during execution. LDFI uses data lineage to reason explicitly about the redundancy provided by a system, in order to aggressively prune the space of executions that it must consider. However, as we saw in Section 4.2, the tracing infrastructure used by Netflix does not directly expose this redundancy in call graphs. Intuitively, however, we know that redundancy exists in the Netflix backend; the challenge was to make it explicit.

Our solution was to have LDFI learn about the various alternative computations provided by the site for a particular class of user requests over time. Recall that Section 5.2 described a mechanism for simulating the replay of user-generated interactions by grouping requests into equivalence classes and treating all requests that map into a particular class as though they are replays of a single user interaction. It occurred to us that we could take advantage of this replay mechanism to incrementally build a model of the variety of alternative ways that a particular request class can provide a satisfactory response.

For example, Figure 4 (as we discussed in Section 4.2) reveals no redundancy: it simply describes the interaction between the services that were used to generate a response to a particular user request that mapped into that class. Based on an analysis of this callgraph alone, it might appear that if the node hosting the ratings service were to crash, the entire request might fail. A Netflix site reliability engineer, on the other hand, might know that in such an event, the API service would fail over to a `RatingsFallback` service that provides (possibly state) ratings information in order to allow the client response to be generated with acceptable defaults. This alternative computation is missing from the graph, for an obvious reason: the fallback code was not executed servicing the original user request.

However, if our automated failure testing system were to generate a hypothesis that failing `RatingsService` could cause a user-visible error, and then tested that hypothesis via fault injection, in a subsequent "replay" (i.e., the next user request that maps to the same class) the system would succeed and generate a slight different request graph (e.g., one in which the `RatingsService` node was replaced by a `RatingsFallback` node). *Collectively*, these graphs explicitly capture the redundancy in this system, which (continuing this process) will be incrementally revealed over time.

By maintaining long-lived models of these alternative computations provided by each request class, we were able to capture both the *dependencies* within individual computations and the *redundancy* across them. We used this structure to stand in for the lineage graph described in Section 2.

6 Results

In this section, we begin by briefly describing the LDFI service that we implemented at Netflix. We then present a case study in which we describe details of how the LDFI service explored a particular request class (called App Boot), and the fault tolerance bugs it identified.

6.1 Implementation

Having addressed the challenges described in Section 4, we implemented LDFI as a service that interposes between the tracing service and FIT.

For each request class (as described in Section 5.2), the LDFI service maintains a model of the alternative computations that are sufficient to produce a satisfactory response for requests that fall within the class. This model—the substitute for fine-grained lineage that we presented in Section 5.3—is enriched over time as described below; at any given time, the model can be thought of as the conjunction of the various call graphs that were produced by experimenting with different fault scenarios. Given such a model, LDFI can produce a set of hypotheses by representing the known alternatives as a Boolean formula and solving, as described in Section 2.

The LDFI service is driven by a daemon that periodically spawns three types of jobs:

1. Training: the service collects production traces from the tracing infrastructure and uses them to build a classifier (as described in Section 5.2) that determines, given a user request, to which request class it belongs. We found that the most predictive features include service URI, device type, and a variety of query string parameters including parameters passed to the Falcor [11] data platform.

2. Model enrichment: the service also uses production traces generated by experiments (fault injection exercises that test prior hypotheses produced by LDFI) to update its internal model of alternatives. Intuitively, if an experiment failed to produce a user-visible error, then the call graph generated by that execution is evidence of an alternative computation not yet represented in the model, so it must be added. Doing so will effectively prune the space of future hypotheses.

3. Experiments: finally, the service occasionally "installs" a new set of experiments on Zuul. This requires providing Zuul with an up-to-date classifier and the current set of failure hypotheses for each active request class. Zuul will then (for a small fraction of user requests) consult the classifier and appropriately decorate user requests with fault metadata.

At any time, administrators may query the LDFI service to obtain information about user-visible failures that have been uncovered, the current state of lineage models for the active request classes, and the current failure hypotheses that will be tested in the next experiment.

6.2 Case study: App Boot

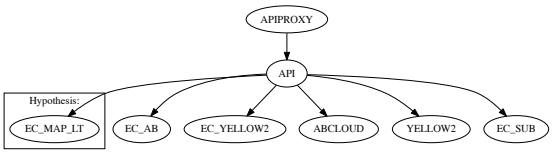

Figure 5: A simplified example of a failure-free run of App Boot. Most of the data required to satisfy the request is obtained from the `EVCache` tier (nodes with the prefix "`EC_`"). A failure hypothesis (crash `EC_MAP_LT`) generated by LDFI after processing the graph is highlighted.

In this section we focus on one of the most critical interactions users have with the Netflix site: a particular request class that Netflix developers colloquially refer to as "App Boot." This interaction loads the metadata needed to run the Netflix application and load the initial list of videos for a member. App Boot represents a moment of truth that, as a company, we want to win by providing a reliable experience from the very start.

It is also a very complex request, touching dozens of internal services and hundreds of potential failure points. A complete (but too dense to read) call graph for App Boot is shown in Figure 6. Brute force exploration of this space would require roughly 2^{100} experiments.

Figure 5 shows a simplified call graph produced by a failure-free initial run of the App Boot request class. Note that in order to make the graphs readable, we display them at a courser grain than the example graph we showed in Figure 4. In Figures 5 and 7, services (boxes in Figure 4) are shown as graph nodes (ovals), while individual fault injection points (ovals in Figure 4) are not shown.

Figure 5 also highlights a failure hypothesis generated by LDFI after processing the graph: causing a fault in the `EC_MAP_LT` service could prevent the App Boot interaction from providing a useful response to the user. At some future time, the LDFI service propagated the details necessary to test this hypothesis via FIT. Some time after that, a user request arrived that the classifier mapped into the App Boot request class. Zuul decorated the user request with the appropriate failure scenario metadata, triggering the appropriate downstream faults.

However, that experiment failed to produce a user-visible error. Instead, it logged a new call graph, shown in Figure 7. Note first that the `EC_MAP_LT` service does not appear as a node in this call graph, because the service was crashed by the FIT infrastructure. In its place, however, a new subgraph has "grown" (shown in a box). The `EC_MAP_LT` cache service materializes the results from a collection of service calls that are required to compile the "LOLOMO" (or list-of-lists-of-movies) that is presented to users after logging in to the Netflix service. Because the cache was unavailable, the API tier called the `MAPLOLOMO` service as a fallback. It, in turn, called the services required to compile the LOLOMO, including the `GROUP_SERVICE`, `GPS_SERVICE` and several lower-level caches.

The `EC_MAP_LT` service and the subgraph that replaced it in Figure 7 represent alternative computations provided by the Netflix backend that are individually sufficient to produce a satisfactory user response for the App Boot request class. Armed with this additional knowledge about fallbacks, LDFI will never again explore scenarios in which faults are injected in `EC_MAP_LT` without also injecting faults in some node in the new subgraph (e.g. `GPS_FRONTEND`). Every time the service spends resources and time running a failure experiment, the results of the experiment either uncover a bug or prune the space of future experiments.

LDFI covered the space of failure scenarios for App Boot after running just under 200 experiments—a miniscule subset of the 2^{100} potential failure scenarios into which random strategy would "stab." Along the way, it discovered 11 new critical failures that could prevent users from streaming content, several of which involved "deep" failure scenarios involving a combination of service fault events.

7 Related Work

Fault injection is a relatively mature subject in the fault tolerance literature [2, 10, 16, 18]. "Failure testing as a service" was first proposed by Gunawi et al [15]. More recently, large-scale infrastructures supporting fault injection in production systems have emerged [6, 13, 29, 23]. In this work, we took advantage of an existing fault injection service and focused on the problem of intelligently searching the combinatorial space of possible failures. To the best of our knowledge, our system is the first large-scale failure testing service to automate the search using techniques more sophisticated than simple random fault injection.

Formal methods [19, 12, 17, 24, 32, 25, 33] have always been available to programmers, but most require significant expertise in a modeling or annotation language. De-

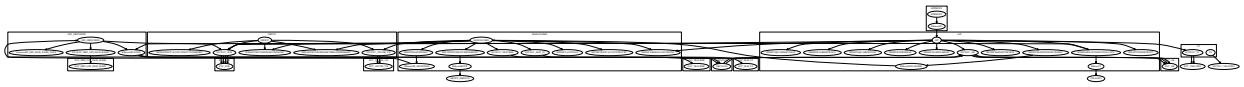

Figure 6: The App Boot call graph.

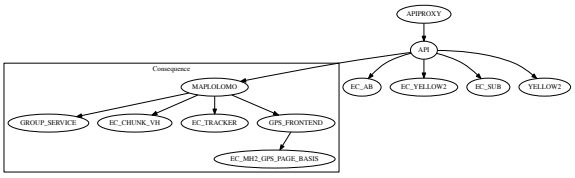

Figure 7: A simplified trace for the request class shown in Figure 5, after injecting a crash failure in the `EVCache` Map service (`EC_MAP_LT` cache service). In this execution, the LOLOMO (List-of-lists-of movies) must be compiled via a deeper subgraph involving calls to several services (e.g. `GROUP_SERVICE`, `GPS_SERVICE`).

spite the difficulty of mastering these tools, large scale Internet companies have reported some recent success using formal methods [26]. In contrast to most of these approaches, our project focused on finding bugs on unmodified (and indeed already deployed) systems. Rather than exhaustively verifying the behavior of individual components, as would a strategy such model checking, FIT and LDFI test end-to-end properties of complete systems.

LDFI uses data lineage to reason about the underlying redundancy provided by a fault tolerant distributed system. Lineage [22, 8, 20, 27, 31] is a mature research area in the data management systems community. While the LDFI prototype uses classic lineage collection and analysis techniques, as we described in Section 4.2 fine-grained data lineage was not available at Netflix. Our approximation—the conjunction of the history of call graphs for a given request class—resembles a (positive) lineage graph.

8 Conclusions and Future Work

The results presented in Section 6 are preliminary, as LDFI has been running in production at Netflix for a short time. As it continues to run on a fraction of our production user traffic, we expect that it will continue to uncover both "fresh" bugs in new software releases and existing deep bugs lurking within new request classes that we have not yet explored. We also anticipate that as the Chaos Engineering discipline proliferates among Internet companies, approaches such as LDFI will become increasingly relevant.

As we saw, the challenges in applying the research to a "real-world" system largely arose from an impedance mismatch between the idealized model of reality of the research prototype and the rigid and often messy realities of production systems. It was impractical to modify the existing systems to more closely resemble the model—in addition to the bare complexity of such a task, there are too many live processes dependent on them to make such changes safe. At the same time, it was undesirable to complicate the models to make them resemble reality—after all, the strength of models is in their simplicity and abstraction.

In the end, the solutions that we devised all involved a principled *mapping* between concrete systems and structures and the idealized model. We were able to preserve the fundamental facets of the LDFI approach—successful outcomes, lineage and replay—by approximating them as "views" over existing services and data structures.

We hope that lessons have emerged from our experiences on two levels. First, we showed how to implement LDFI as a microservice that "snaps in" to existing tracing and fault injection infrastructures. We hope that this experience can be a guide for future integrations efforts. Second, and perhaps more importantly, we share important evidence that it is possible (and indeed, profitable!) to push distributed system research prototypes into production usage.

References

[1] Chaos Community Day. `http://chaos.community`.

[2] The Netflix Simian Army. `http://techblog.netflix.com/2011/07/netflix-simian-army.html`, 2011.

[3] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. CIDR'12.

[4] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in Time and Space. Datalog'10.

[5] P. Alvaro, J. Rosen, and J. M. Hellerstein. Lineage-driven fault injection. In *SIGMOD*, 2015.

[6] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, May 2016.

[7] P. Buneman, S. Khanna, and W.-c. Tan. Why and Where: A Characterization of Data Provenance. ICDT'01.

[8] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Found. Trends databases*, April 2009.

[9] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, June 2000.

[10] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A Fault Injection Environment for Distributed Systems. Technical report, FTCS, 1996.

[11] What is Falcor? `https://netflix.github.io/falcor/starter/what-is-falcor.html`.

[12] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*. Springer Berlin Heidelberg, 2008.

[13] FIT : Failure Injection Testing. `http://techblog.netflix.com/2014/10/fit-failure-injection-testing.html`.

[14] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*

[15] H. S. Gunawi, T. Do, J. M. Hellerstein, I. Stoica, D. Borthakur, and J. Robbins. Failure as a service (FaaS): A cloud service for large-scale, online failure drills. Technical report, EECS Department, University of California, Berkeley, 2011.

[16] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. NSDI'11.

[17] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[18] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Trans. Comput.*, Feb 1995.

[19] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. NSDI'07.

[20] S. Köhler, B. Ludäscher, and D. Zinn. First-Order Provenance Games. In *In Search of Elegance in the Theory and Practice of Computation*, volume 8000 of *LNCS*. Springer, 2013.

[21] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*

[22] A. Meliou and D. Suciu. Tiresias: The Database Oracle for How-to Queries. SIGMOD '12.

[23] Introduction to the Fault Analysis Service. `https://azure.microsoft.com/en-us/documentation/articles/service-fabric-testability-overview/`.

[24] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 2002.

[25] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI'08.

[26] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. Use of Formal Methods at Amazon Web Services. Technical report, 2014.

[27] S. Riddle, S. Köhler, and B. Ludäscher. Towards Constraint Provenance Games. TaPP'14.

[28] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[29] A Deep Dive into Simoorg: Our Open Source Failure Induction Framework.

[30] G. Tsoumakas and I. Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining*, 3(3):1–13, 2007.

[31] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering Why-not Queries in Software-defined Networks with Negative Provenance. HotNets'13.

[32] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. NSDI'09.

[33] Y. Yu, P. Manolios, and L. Lamport. Model checking tla+ specifications. CHARME '99.