

# Fault Injection

Lecture 13: CPEN 400P

Karthik Pattabiraman, UBC

(Some Slides based on Wes Weimer's course at U. Mich. and the Netflix Technical Blog - <https://netflixtechblog.com/>)

# Outline

## **Resilience Engineering**

Fault Injection

Software Fault Injection

LLFI: LLVM-based Fault Injector

Fault Injection in Cloud Applications

# Resilience: Definition

Property of a system to “bounce back” after a failure or disruption

- Treats failures as common occurrence rather than exceptions
- Need to have (sufficient) redundancy to recover from failure
- Closely related to the notion of availability
- Availability - Fraction of time the system is up for service

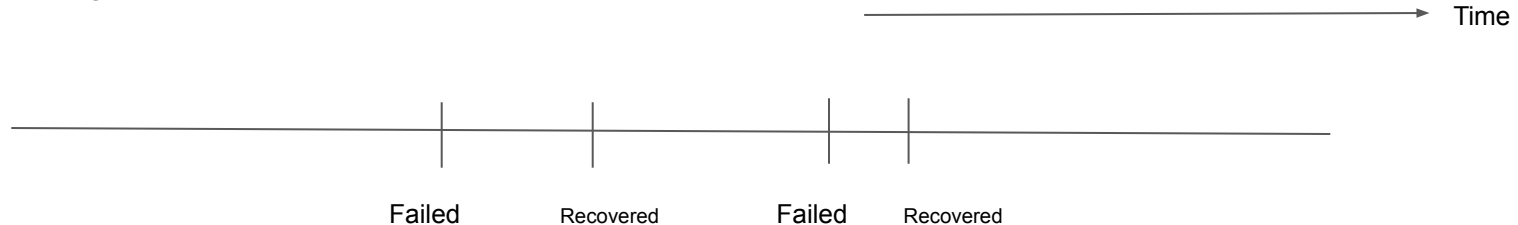
# Availability

Readiness of system for service - probability that the system is up for service

MTTF: Mean Time to Failure

MTTR: Mean Time to Recovery

$$\text{Availability} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$$



# Resilience

Resilience engineering attempts to maximize availability by **decreasing** the MTTR

Bug-finding approaches via static and dynamic analysis attempt to reduce MTTF

Need a systematic way to evaluate the resilience of the system

- Many bugs are often found in error-detection and recovery code [IBM study]
- Hidden dependencies in the system may prevent (Fast) recovery
- Eliminate need for manual problem identification and fixing

# Outline

Resilience Engineering

## **Fault Injection**

Software Fault Injection

LLFI: LLVM-based Fault Injector

Fault Injection in Cloud Applications

# Fault Injection

- Fault-injection (or fault-insertion) is the act of deliberately introducing faults into the system in a controlled and scientific manner, in order to study the system's response to the fault
  - Can be used to estimate resilience (e.g., detection, recovery)
  - Also used to understand inherent fault tolerance of the system
  - To obtain reliability estimates of the system prior to deployment (requires statistical projection)

# Why fault-injection ?

- **Versus Model-based**

- More realistic, as it evaluates actual system
- No need to worry about mathematical feasibility
- No need to supply input parameters

- **Versus operational measurements**

- Failures take a **\*long\*** time to occur and when they do, are often not reproducible or analyzable
- Failures provide limited insight into what **\*can\*** go wrong
- Need to wait until the system is deployed - often too late

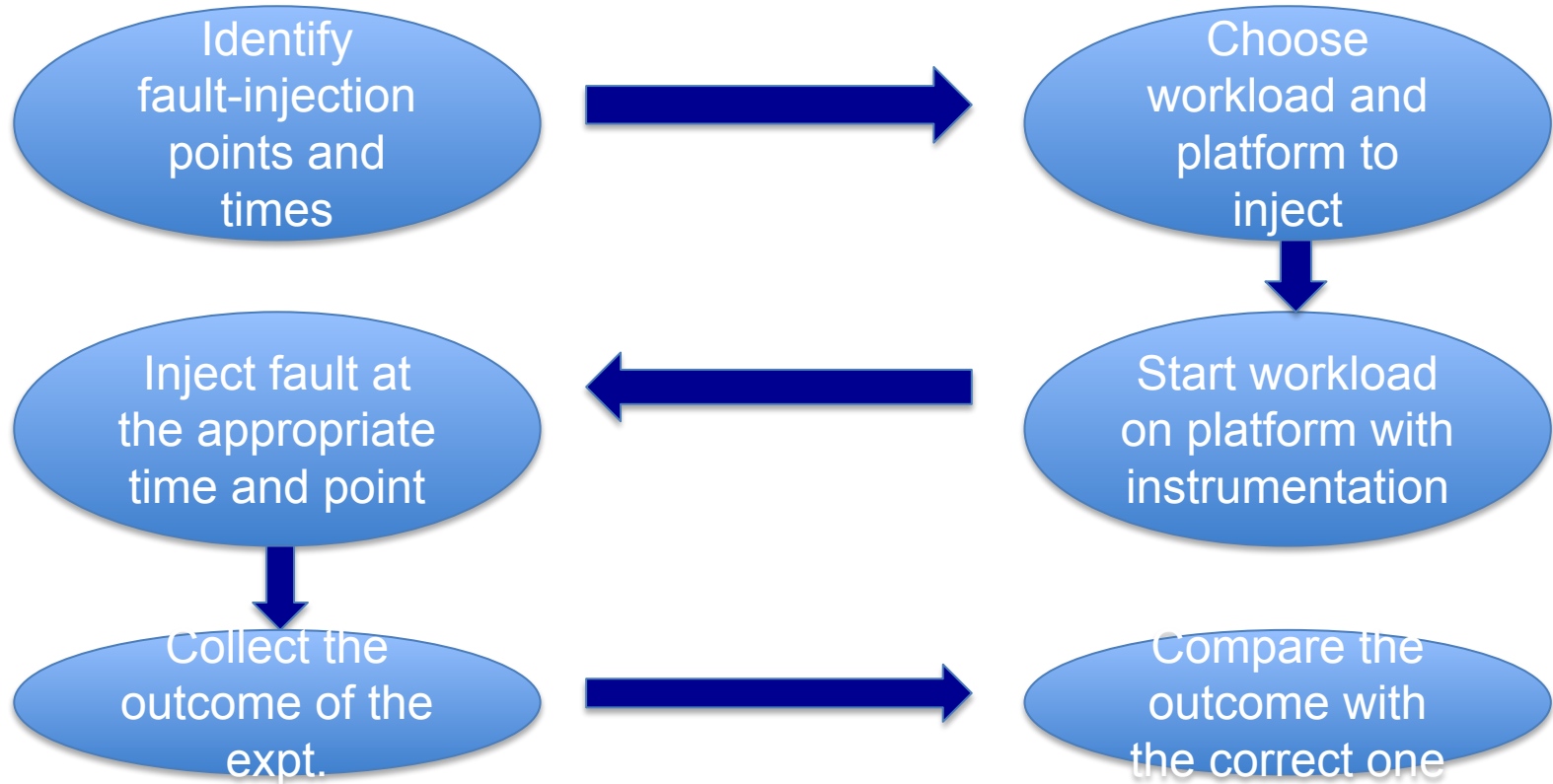


# Example of Real-world Fault-Injection: ChaosMonkey

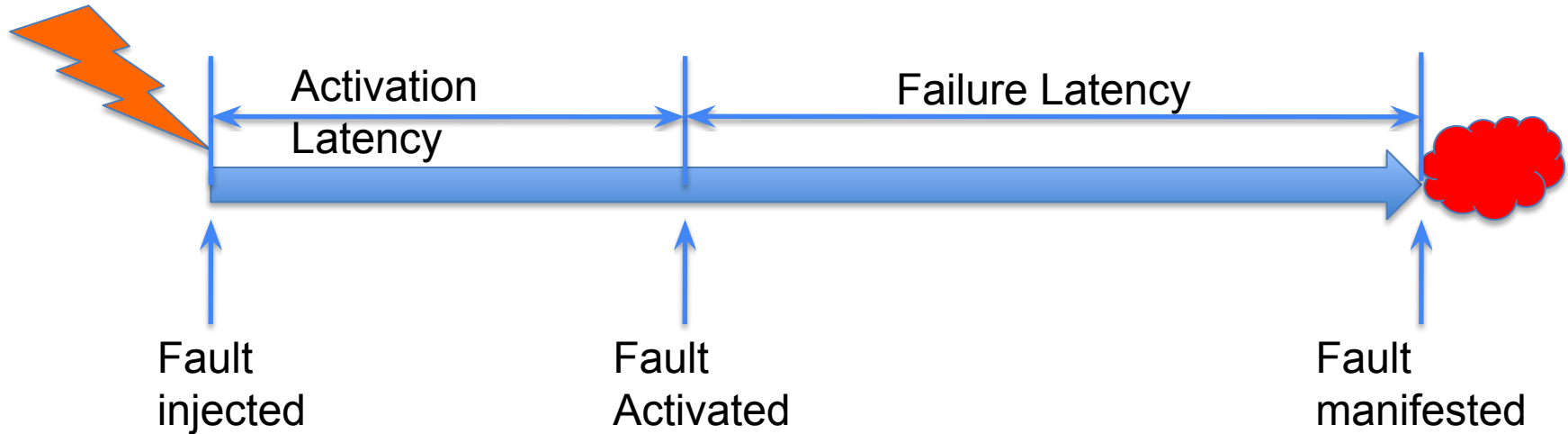
Invented in 2011 by Netflix to test the resilience of its IT infrastructure

*“Imagine a monkey entering a “data center”, these “farms” of servers that host all the critical functions of our online activities. The monkey randomly rips cables, destroys devices and returns everything that passes by the hand. The challenge for IT managers is to design the information system they are responsible for so that it can work despite these monkeys, which no one ever knows when they arrive and what they will destroy.” – Antonio Martinez, Chaos Monkey*

# Fault-injection Steps



# Measures to Compute



- What fraction of injected faults are activated ?
- What fraction of activated faults manifest as failures ?
- What are the average activation and failure latencies ?

# Assumptions/Requirements

- A representative set of faults must be injected
  - Need to include enough faults to give confidence in the measures being studied
- Only one or controlled no. of faults injected
  - Ability to map the outcome to a set of faults
- Need to have a specification of correct behavior to distinguish incorrect outcomes
  - May need to determine golden run ahead of time

# Outline

Resilience Engineering

Fault Injection

**Software Fault Injection**

LLFI: LLVM-based Fault Injector

Fault Injection in Cloud Applications

# Software-based Fault-Injection (SWiFI)

## Pros

- Do not require expensive hardware modifications
- Can target applications and OS errors
- Many hardware faults do not require probes, e.g, register data corruption

## Cons

- Restricted to inject only faults that S/W can see
- May perturb the workload that is running on the system, resulting in missing many heisenbugs
- Coarser-grained time resolution than h/w

# SWiFl: Types

## Compile-time

- Modify source code or machine code of the program prior to execution
- Can be used to model software defects
- Requires going through the compile-run cycle each time

## Runtime

- Modify the program or its data during runtime
- Can be done through the debugger, kernel or with support from compiler
- No need to go through compile-run cycle each time

# Compile-time Injection

- Modify program's code prior to execution
  - Model hardware transient faults in machine code
  - Model software errors that are deterministic (Bohrbugs) on specific code paths
  - Typically only inject into the first dynamic instance of an instruction
- Main advantage: Take advantage of static analysis of the code to customize injection



# Runtime Injection

- Advantages

- Can inject faults without recompiling - speed
- Faults can occur deeper in the execution. e.g., one-millionth iteration of a loop
  - Some of the errors can be non-deterministic (e.g., Mandelbugs)
- Fault can depend on runtime conditions. e.g., if memory usage exceeds a threshold, inject fault (includes aging-related bugs)

# Compile-time Injector Example: GSWFIT

G-SWFIT: Developed at U. Coimbra by Henrique Madiera and others

- First tool to inject **representative** software faults based on Orthogonal Defect classification (ODC)
- Injection into the machine code of the program - no need for source code (for the most part)
- For more details, see the following paper

Duraes, Joao & Madeira, Henrique. (2006).

*Emulation of Software Faults: A Field Data Study and a Practical Approach. Software Engineering*, IEEE Transactions on. 32. 849-867. 10.1109/TSE.2006.113.

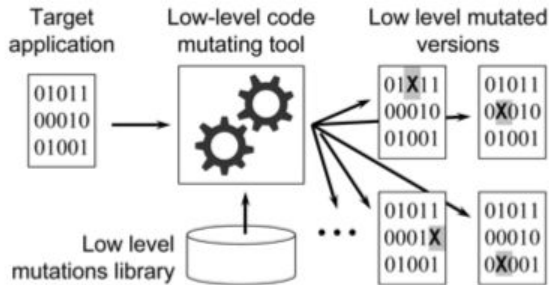
# G-SWFIT: Main Idea

Injects compile-time faults in the machine code of the program under test

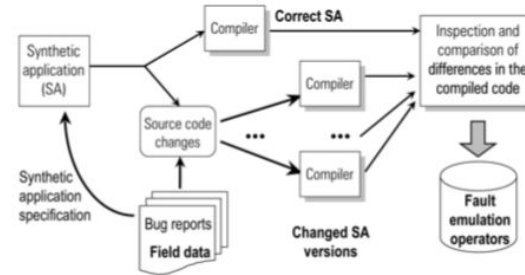
- Search patterns: Patterns of machine code that represent common high-level programming constructs
  - Mutations based on ODC classification
- Low-level Faults: Faults in a single machine code instruction
  - Mutation based on flipping bits of instructions

# G-SWFIT: Approach

Added mutation operators for top 'N' fault types in extended ODC - covers 67.6%



**Overall Methodology of G-SWFIT:**  
**Mutation library based on ODC**



**Identification mutation operators:**  
**Based on bug-reports and field data**

# G-SWFIT: Examples of Faults Injected

## Missing Function Call (search pattern)

- Look through the machine code for pattern corresponding to function call and replace it with No-Ops
- Need to replace return value with prior value

## Missing Variable Initialization (low-level)

- Find the instruction that assigns a constant to a variable and replace it by No-Ops, or randomly perturb its contents

# Runtime Injector Example: NFTAPE

Developed at the University of Illinois at Urbana Champaign (since 2000)

- Successor of FI tools such as FTAPE, DEPEND etc.
- Long line of runtime fault injection tools such as Xception, Ferrari etc.
- Common architecture and interface of multiple fault injectors

Paper describing NFTAPE (Tool has evolved considerably since then):

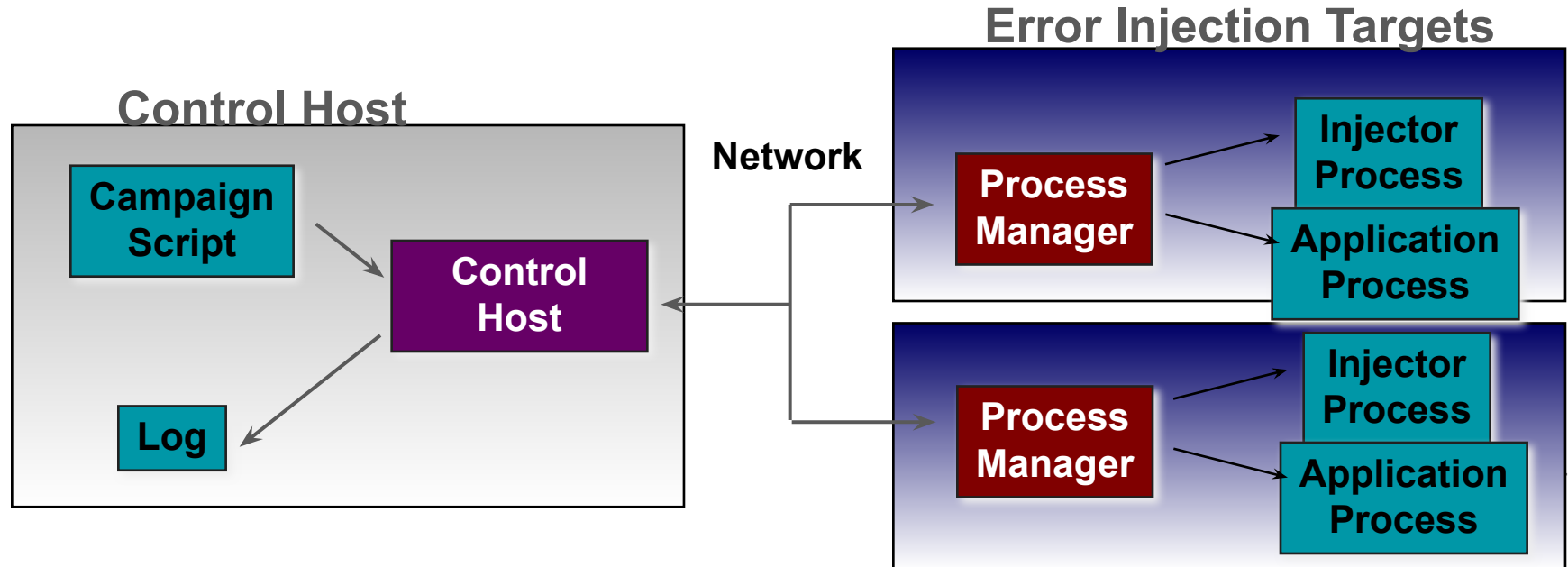
Stott, David & Floering, B. & Burke, Daniel & Iyer, Ravishankar. (2000). NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. Proceedings -IEEE International Computer Performance and Dependability Symposium, IPDS.

# Injection Targets and Outcome Categories

Target	User space
Code	Functions: e.g., main Instructions: any or selected subset (e.g., branch, load, store)
Data	Static data and dynamically allocated memory (heap)
Stack	Data on an application stack
CPU Registers	General purpose registers
Memory range	Any location in application memory space

Outcome Category	Description
Activated	The corrupted instruction/data is executed/used.
Not Manifested	The corrupted instruction/data is executed/used, however it does not cause a visible abnormal impact on the system.
Fail Silence Violation	Either operating system or application erroneously detects the presence of an error or allows incorrect data/response to propagate out. Workload programs are instrumented to detect errors.
Crash	Application/OS stops working, e.g., bad trap or system panic. Crash handlers embedded into OS are enhanced to enable dump of failure data (processor and memory state).
Hang	System resources are exhausted resulting in a non-operational application/system, e.g., deadlock or livelock .

# NFTAPE Framework Configuration





# Outline

Resilience Engineering

Fault Injection

Software Fault Injection

**LLFI: LLVM-based Fault Injector**

Fault Injection in Cloud Applications

# Why yet another fault injector?

- **Difficult to customize existing injectors**

- Inject into specific instructions
- Inject into a specific variable
- Inject into specific code constructs

- **Difficult to understand the results**

- Difficulty in fault injection customization
- Difficult to study the propagation of errors
- Difficult to map result back to source code

# LLFI

- **A fault injector based on LLVM (<http://llvm.org>)**

- Intermediate representation (IR) level injection
- Hybrid compile-time and runtime injection

- **Features**

- Easy to customize the fault injection
- Easy to analyze fault propagation
- Accurate compared to assembly level injection

# LLFI: Hybrid Compile/Runtime Injection

- **Source-level instrumentation of programs**

- Integrated with a compiler framework, LLVM
- Precise targeting of selected code constructs

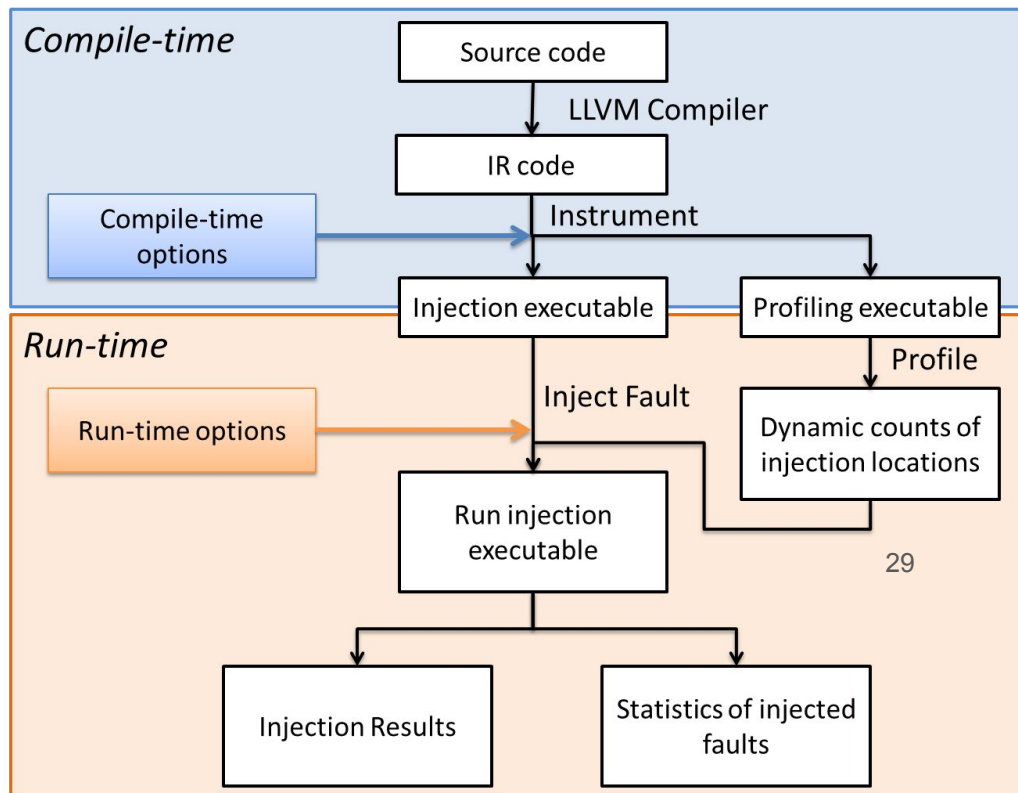
- **Fault-injection is done at program runtime**

- Avoid going through the compile-cycle every time
- Ability to use run-time information to inject faults

- **Tracing of faults after injecting them**

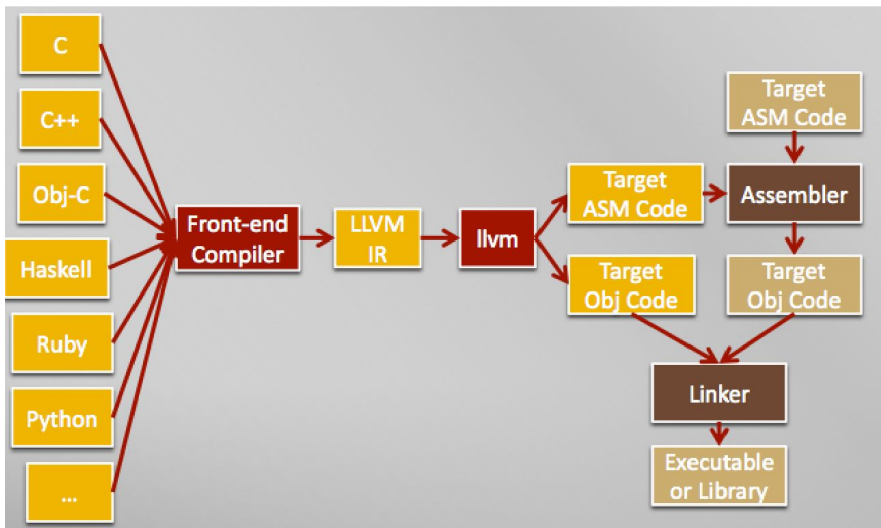
- Map the fault back to the source code and data

# LLFI: Workflow

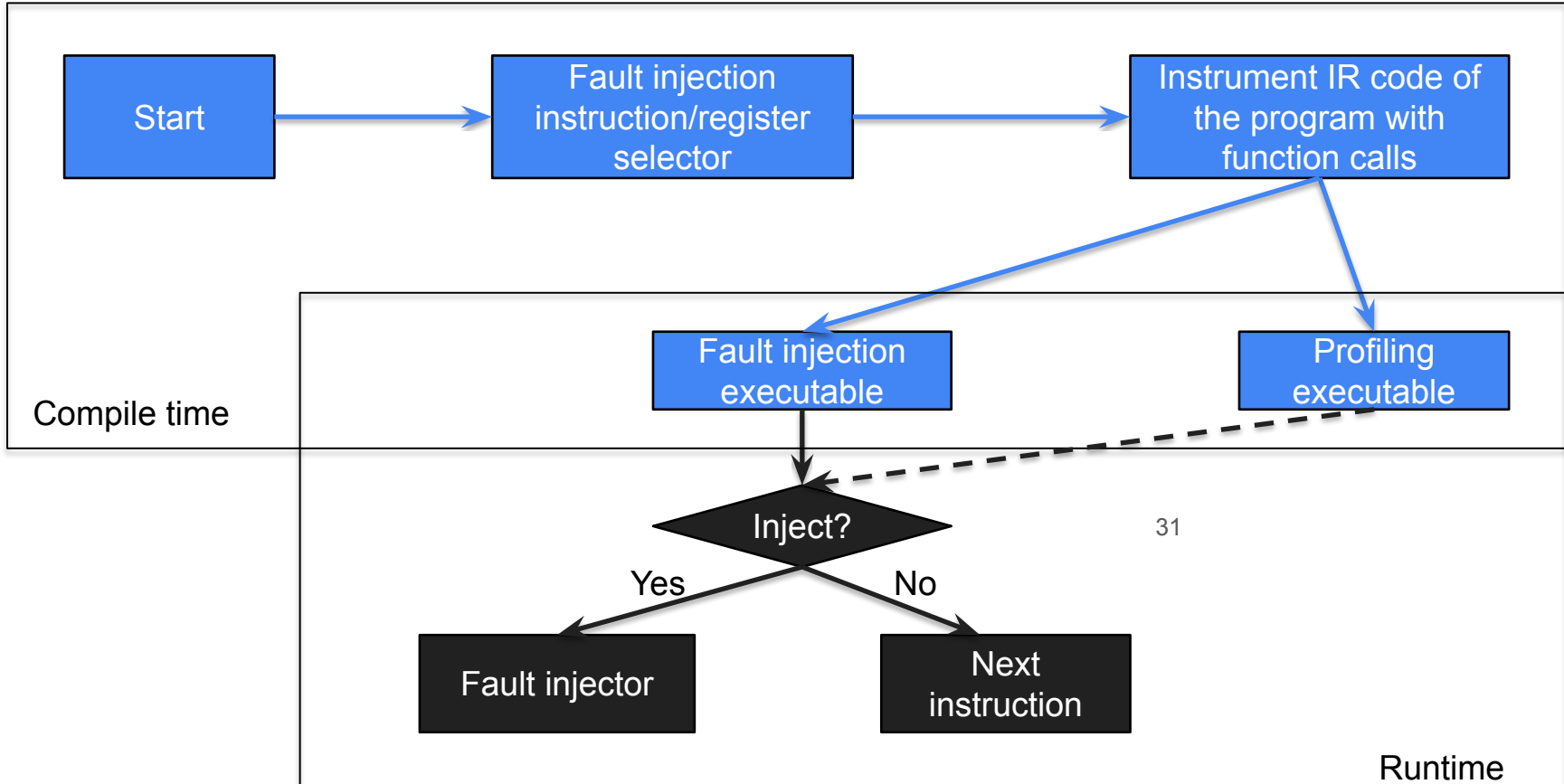


# Why LLVM Compiler ?

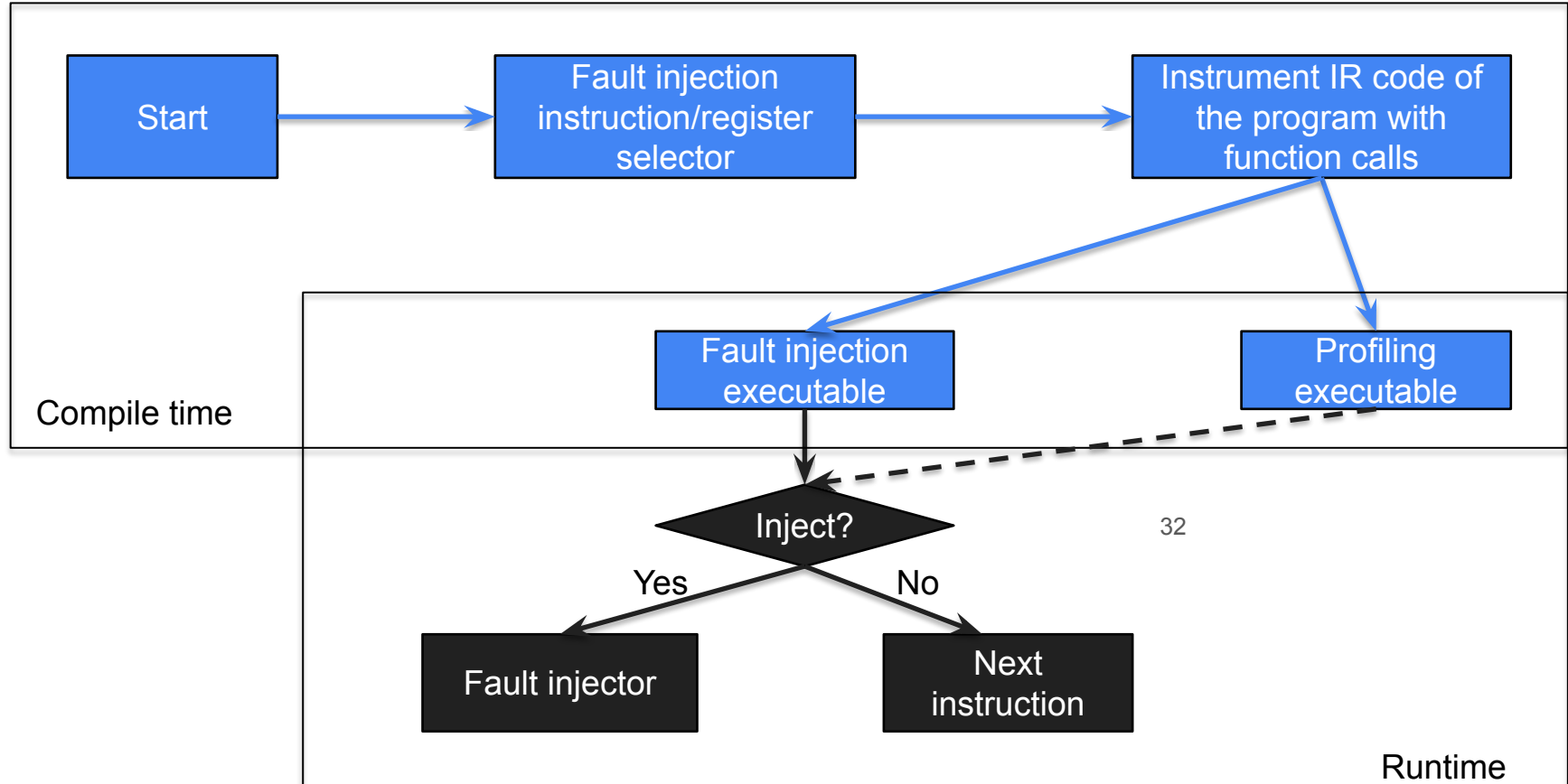
- Supports wide variety of front- and back-ends
- Provides high-level features in the IR code



# How does LLFI work?



# How does LLFI work?





# LLFI: Injection – Example

```
char* buf = "Hello World"; char* p;  
void foo(int size) {  
    p = (char*)malloc(size);  
    memcpy(p, buf, size);  
}  
void goo() {  
    free(p);  
}
```

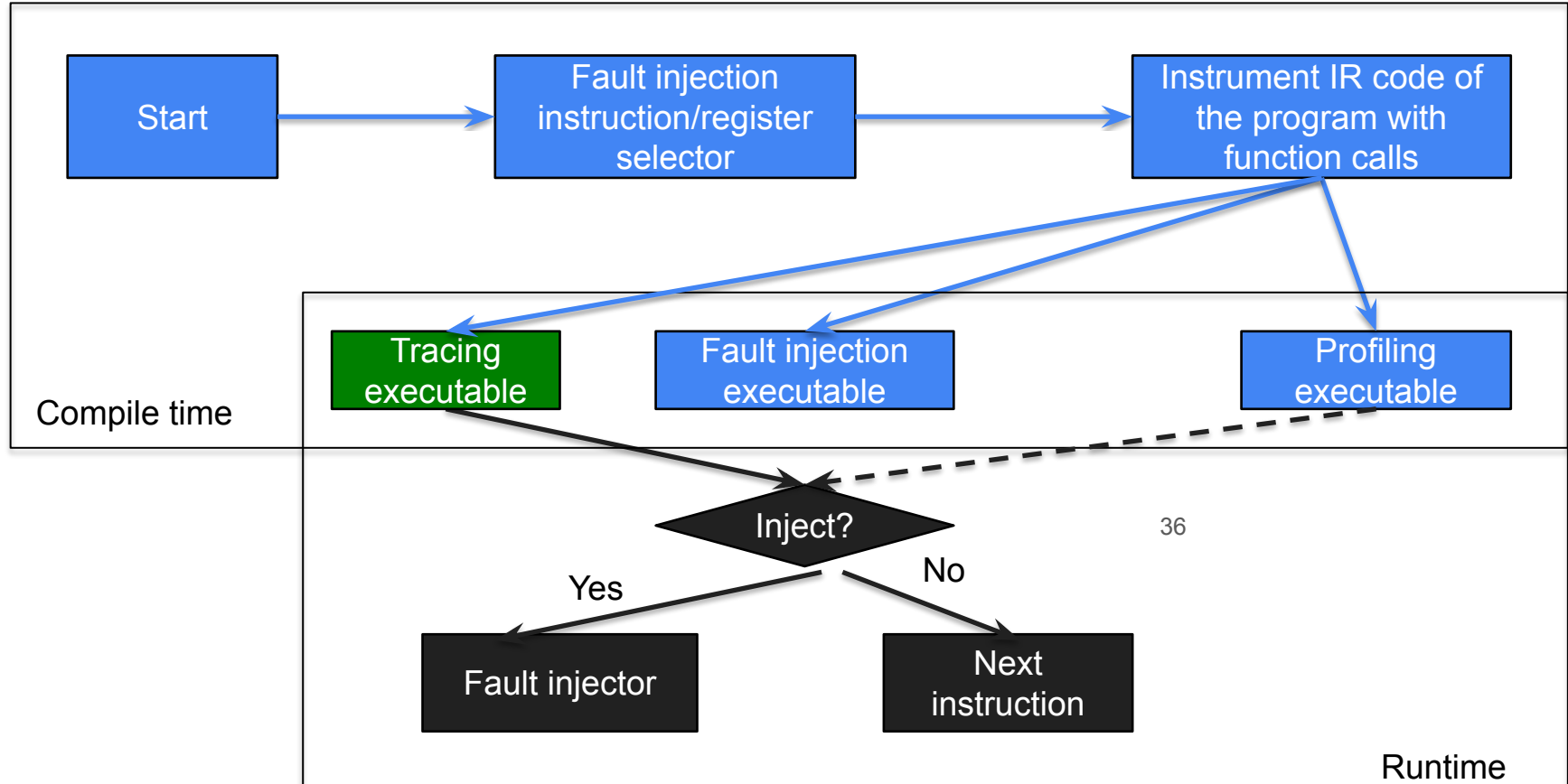
# LLFI: Injection – Buffer Overflow

```
char* buf = "Hello World"; char* p; int count = 0;
void foo(int size) {
    p = (char*)malloc( perturbData(size, count++));
    memcpy(p, buf, size);
}
void goo() {
    free(p);
}
```

# LLFI: More complex cases

- **Can inject into data of specific types/structures**
  - Example: Code that manipulates linked list nodes
  - Example: Arguments of certain function calls
- **Can inject faults at specific execution points in the program – based on the program's state**
  - Example: 100<sup>th</sup> iteration of a loop
  - Example: Call to a function when `arg=some_value`
  - Example: when the size of the heap > 100 KB

# How does LLFI work?



# LLFI: Tracing Example

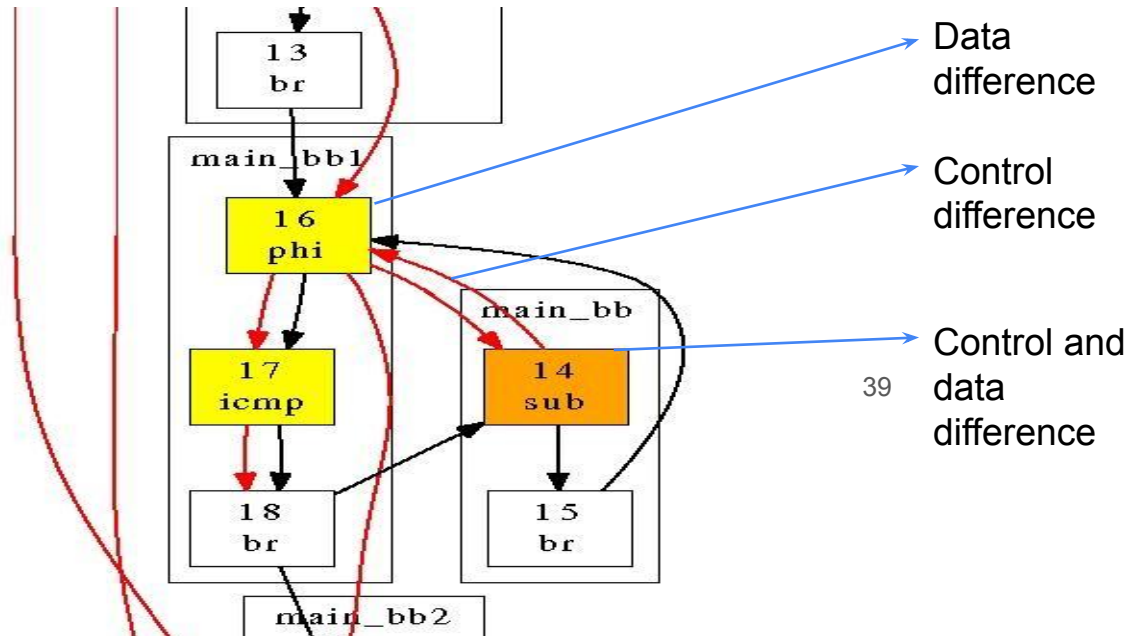
```
char* buf = "Hello World"; char* p; int count = 0;
void foo(int size) {
    p = (char*)malloc( perturbData(size, count++));
    memcpy(p, buf, size);
}
void goo() {
    free(p);
}
```

# LLFI: Tracing Example

```
char* buf = "Hello World"; char* p; int count = 0;
void foo(int size) {
    p = (char*)malloc( perturbData(size, count++));
    memcpy( trace(p), buf, trace(size) );
}
void goo() {
    free( trace(p) );
}
```

# LLFI: Tracing

- Graphical output of trace differences as dot file



# LLFI: Easy to Use (Java GUI)

The screenshot displays the LLFI Java GUI with the following components:

- File List:** A list of files including `factorial.c` and `factorial.ll`.
- Code Editor:** Contains LLVM IR code for a factorial function, including attributes and module flags.
- Tutorial:** A sidebar with instructions for running LLFI, such as uploading a file and configuring options.
- Results Table:** A table showing the status of fault injection attempts, including run options, fault types, indices, lines, cycles, bits, SDC occurrence, status, and results.

Run Option	Fault Injection Type	Index	Line	Cycle	Bit	SDC Occurance	Status	Result	Trace	Select All ...
1	bitflip	31	9	16	20	Occured	Injected	Nil	<input checked="" type="checkbox"/>	
2	bitflip	26	11	13	5	Occured	Injected	Nil	<input checked="" type="checkbox"/>	
3	bitflip	5	N/A	5	21	Not Occured	Injected	Program crashed, terminate...	<input type="checkbox"/>	
4	bitflip	5	N/A	5	42	Not Occured	Injected	Program crashed, terminate...	<input type="checkbox"/>	
5	bitflip	27	11	14	28	Occured	Injected	Nil	<input type="checkbox"/>	
6	bitflip	28	11	31	7	Occured	Injected	Nil	<input type="checkbox"/>	
7	bitflip	28	11	23	14	Occured	Injected	Nil	<input type="checkbox"/>	
8	bitflip	31	9	32	30	Occured	Injected	Nil	<input type="checkbox"/>	
9	bitflip	2	11	2	51	Not Occured	Injected	Program crashed, terminate...	<input type="checkbox"/>	
10	bitflip	31	9	48	0	Occured	Injected	Nil	<input type="checkbox"/>	




# LLFI Architecture

- **Integrated with LLVM Pass Manager**
  - A pass performs an analysis or transformation
  - LLFI passes identify and instrument selected instructions and registers
- **Runtime libraries are simple and portable**
- **Unified Yaml config file for configuring both**

# Example YAML file

```
compileOption:  
  instSelMethod:  
    - customInstselector:  
      include:  
        - BufferOverflowMalloc(Data)  
    - funcname:  
      include:  
        - all  
      exclude:  
        - main
```

Instruction selector (Related to your fault)



```
regSelMethod: customregselector  
customRegSelector: Automatic
```

Function selector



# LLFI Software Faults

- Examples of Supported Faults
  - Data Corruption
  - File I/O Buffer Overflow
  - Buffer Overflow Malloc
  - Function Call Corruption
  - Invalid Pointer
  - Race Condition

# Implementation: LLFI

- ❖ Mapping investigated fault model attributes to LLFI framework

- ❖ Instrumentation pass development

  - A pool of 38 instruction selectors

  - A pool of 38 register selectors

- ❖ Run-time library development

  - More than 20 software fault injectors

44

- ❖ Capabilities added by Software Fault Injection

  - ❖ Automatic register selection

  - ❖ Automatic fault injection

# Outline

Resilience Engineering

Fault Injection

Software Fault Injection

LLFI: LLVM-based Fault Injector

**Fault Injection in Cloud Applications**

# Cloud Applications

Depend on many different components and services - any of these could fail

Distributed across many nodes, and even across data-centers

Failures are the norm, not the exception

Need to introduce failures systematically, in a controlled manner, **in production**

- Get engineers to think about failures from the get go during development
- Practice failure drills and ensure that playbook for failure handling is solid

# ChaosMonkey: Philosophy

Learn about the system via injecting actual failures and watching what happens

Prove or disprove hypothesis about failures by observing what happens

*“We have created Chaos Monkey, a program that randomly chooses a server and disables it during its usual hours of activity. Some will find that crazy, but we could not depend on the random occurrence of an event to test our behavior in the face of the very consequences of this event. Knowing that this would happen frequently has created a strong alignment among engineers to build redundancy and process automation to survive such incidents, without impacting the millions of Netflix users. Chaos Monkey is one of our most effective tools to improve the quality of our services.”*

- Greg Orzell, Netflix Chaos Monkey Upgraded

# ChaosMonkey Arsenal: Simian Army Examples

1. Latency Monkey induces artificial delays in RESTful client-server communication layer to simulate service degradation
2. Conformity Monkey finds instances that don't adhere to best practices and shuts them down (e.g., instances that don't belong to an auto-scaling group)
3. Doctor Monkey taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect unhealthy instances and remove them
4. 10-18 Monkey (short for Localization-Internationalization) detects configuration and run time problems in instances serving customers in multiple geographic regions, using different languages and character sets



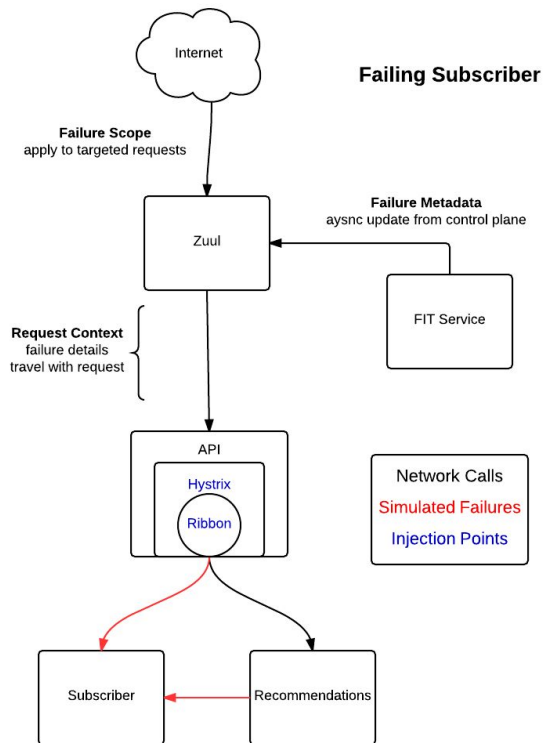
# Challenge - 1

Limit the “blast radius” of the failure, while breaking things in realistic ways

One solution: Treat fault injection as a service and isolate it to a specific set of servers or accounts

- Netflix implemented this via Failure Injection Testing (FIT)
- User writes the Failure Injection service
- Zuul executes the FIT service at the appropriate points and ensures that only the expected accounts/devices are in fact impacted

# Example of FIT



Source:

<https://netflixtechblog.com/fit-failure-injection-testing-35d8e2a9bb2>

Zuul inspects all requests that are sent by the FIT service

- Checks local store of FIT metadata to check if request should be impacted
- If so, decorates the request with a failure context, which is propagated to all dependent services

Use Zuul to isolate impacted requests to only specific accounts or devices, and then gradually dial up the amount of chaos to 100%

Each layer determines how to emulate the failure in a realistic way e.g., sleep for a delay period, return a 500, throw an exception etc.

## Challenge - 2

Characterize the normal behavior of the system via metrics

Compare with behavior after failure is injected to find anomalies

Challenge: How to find normal behavior, especially if it's time varying ?



SPS Metric variation over time at Netflix (Source:  
<https://www.oreilly.com/content/chaos-engineering/>)

# Types of Failures Injected

- Hardware failures
- Functional bugs
- State transmission errors (e.g., inconsistency of states between sender and receiver nodes)
- Network latency and partition
- Large fluctuations in input (up or down) and retry storms
- Resource exhaustion
- Unusual or unpredictable combinations of interservice communication
- Byzantine failures (e.g., a node believing it has the most current data when it actually does not)
- Race conditions
- Downstream dependencies malfunction

Combination of the above events - this is important as many corner cases occur

# Steps in Chaos Engineering

(<https://www.oreilly.com/content/chaos-engineering/>)

1. Pick a hypothesis
2. Choose the scope of the experiment
3. Identify the metrics you're going to watch
4. Notify the organization
5. Run the experiment
6. Analyze the results
7. Increase the scope
8. Automate

# Results of Chaos Engineering

Significant benefits in real systems (see articles on Piazza)

Used by all the big cloud companies to test their infrastructure (Gameday)

AWS introduced AWS Fault Injection Service in 2021

Curated list of resources for fault injection:

<https://github.com/dastergon/awesome-chaos-engineering>

# Outline

Resilience Engineering

Fault Injection

Software Fault Injection

LLFI: LLVM-based Fault Injector

Fault Injection in Cloud Applications