Actionable and Interpretable Fault Localization for Recurring Failures in Online Service Systems

Zeyan Li, Nengwen Zhao, Mingjie Li, Xianglin Lu, Lixin Wang, Dongdong Chang, Xiaohui Nie, Li Cao, Wenzhi Zhang, Kaixin Sui, Yanhua Wang, Xu Du, Guoqiang Duan, Dan Pei

ESEC/FSE 2022

Table of Contents

Background

Search engine

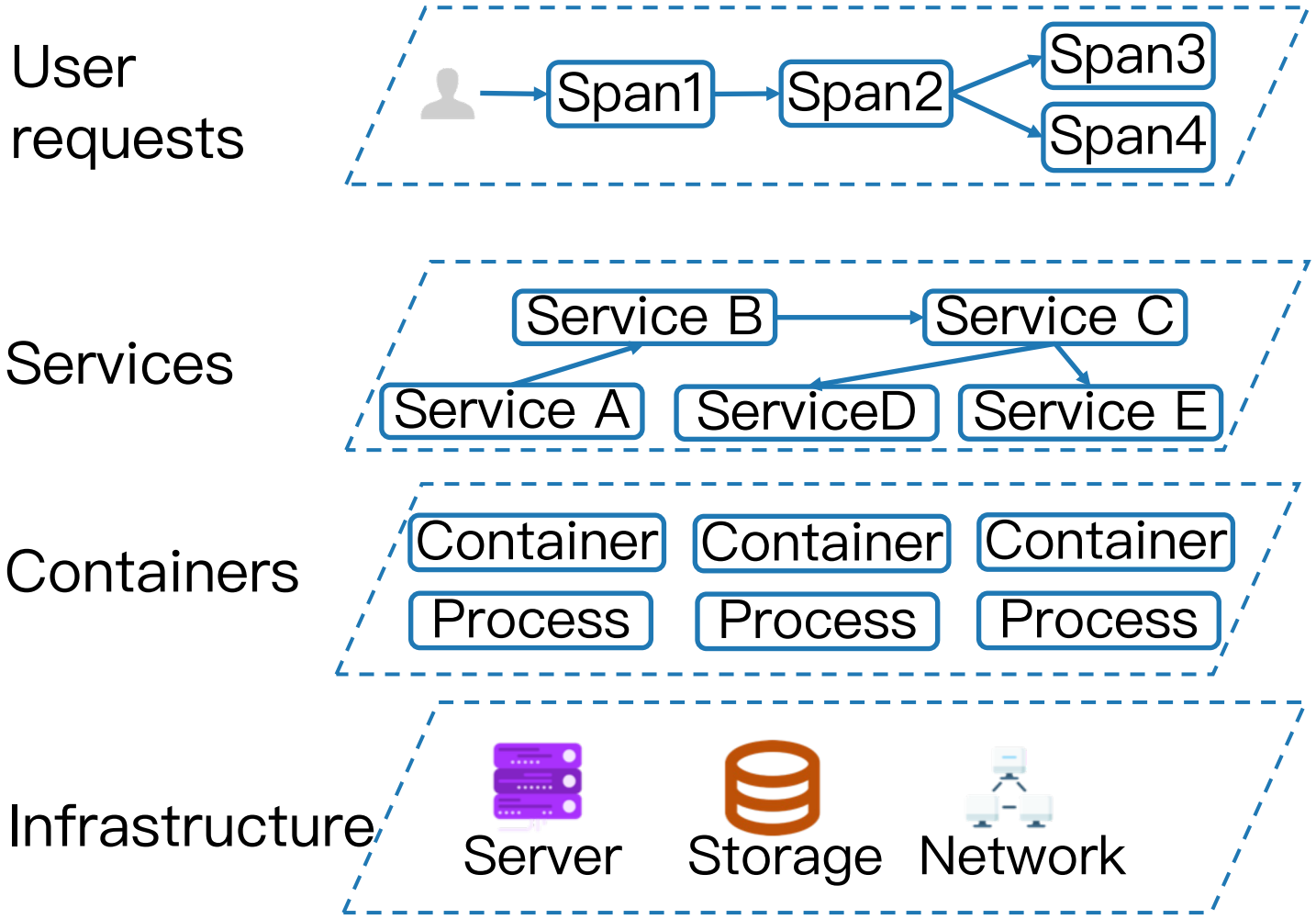
Online shopping

Social network

Videos

Online service systems have become an indispensable part of our daily life.

Background

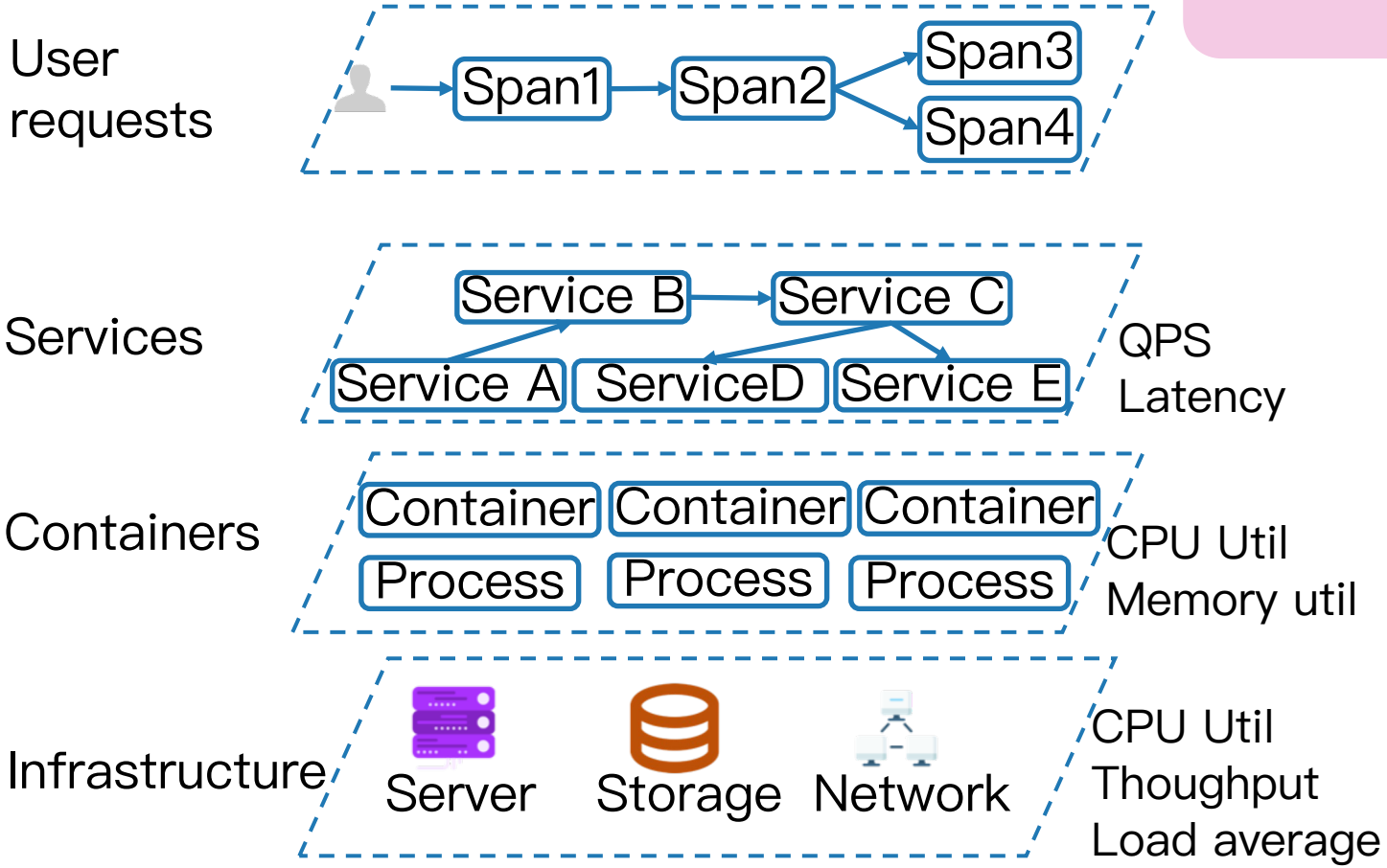


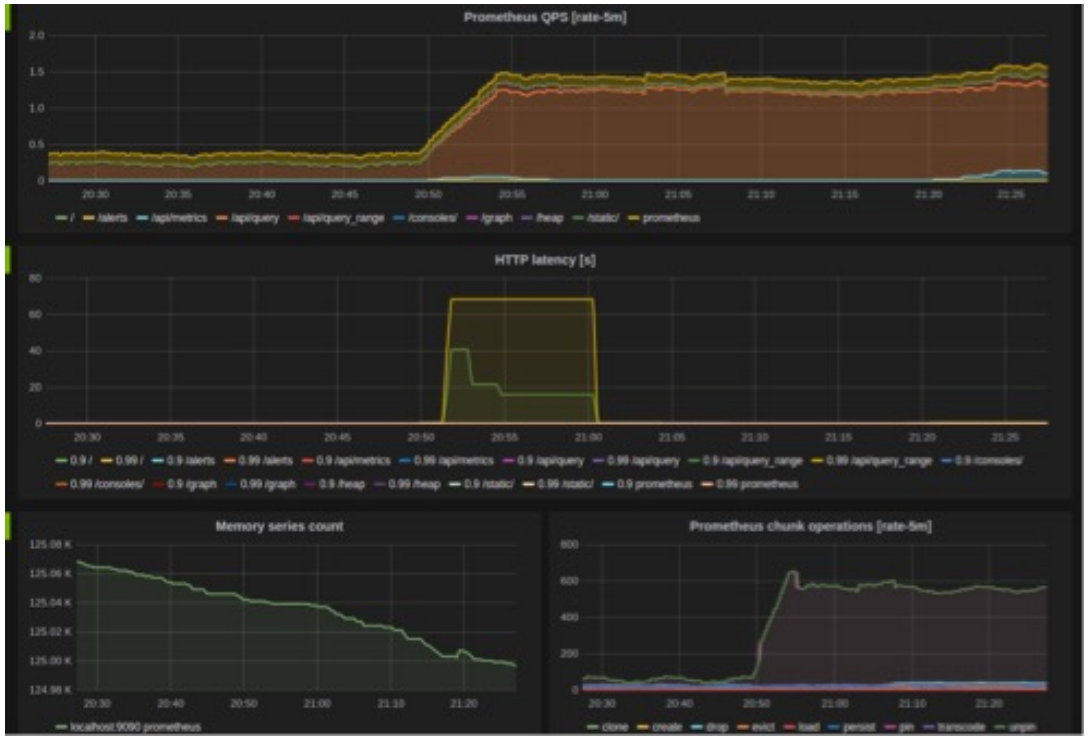
Online service systems comprise many different components (e.g., services, containers and servers).

The large scale and complex dependencies among components make system failures inevitable and hard to diagnose.

Monitoring Metrics

Various metrics are closely monitored on a 24×7 basis. They serve as the most direct signals to the underlying failures





Recurring Failures

Recurring failures are repeated failures of the same kinds at different locations.

Reasons of Recurring

- Wrong fix
- Delayed fix
- High utilization
- Hardware issues
- Third-party services

How to determine recurring failures? Recurring failures should be discovered, mitigated and fixed in similar ways in engineers' perspective.

Recurring Failures

We analyzed 576 failure tickets, spanning one year, of a production banking information system.

Typical recurring failures:

- Failed disks
- Unavailable third-party services
- Missing database indexes
- Slow SQLs

Typical non-recurring failures:

- Design flaw
- Data inconsistency
- Code defects

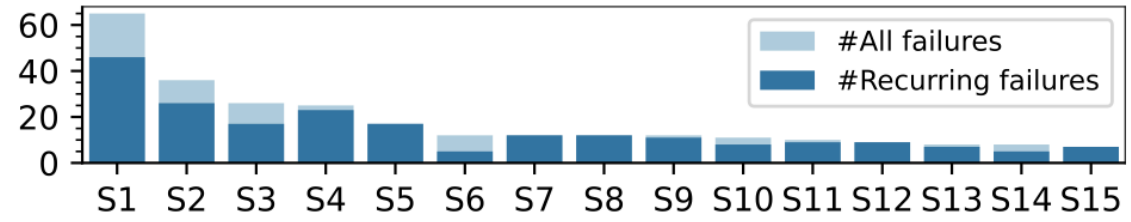
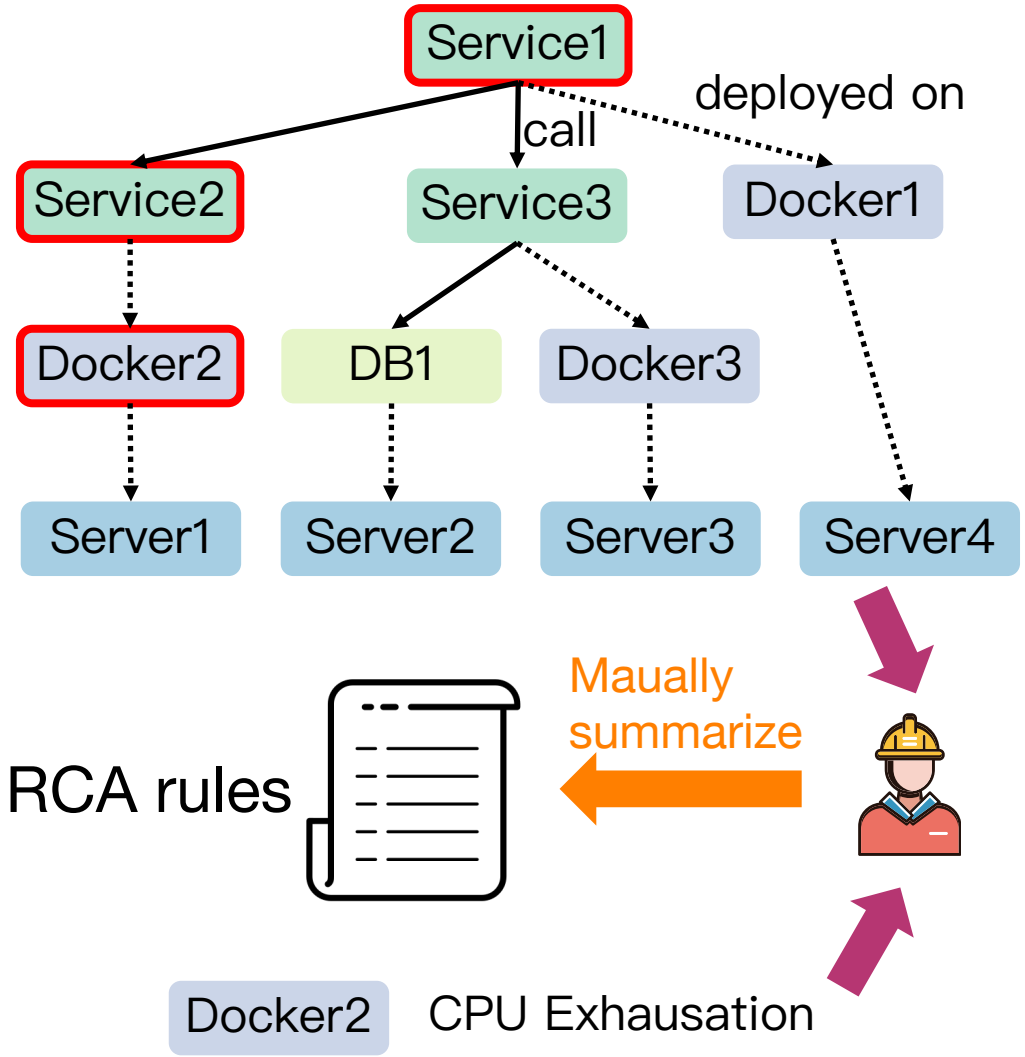
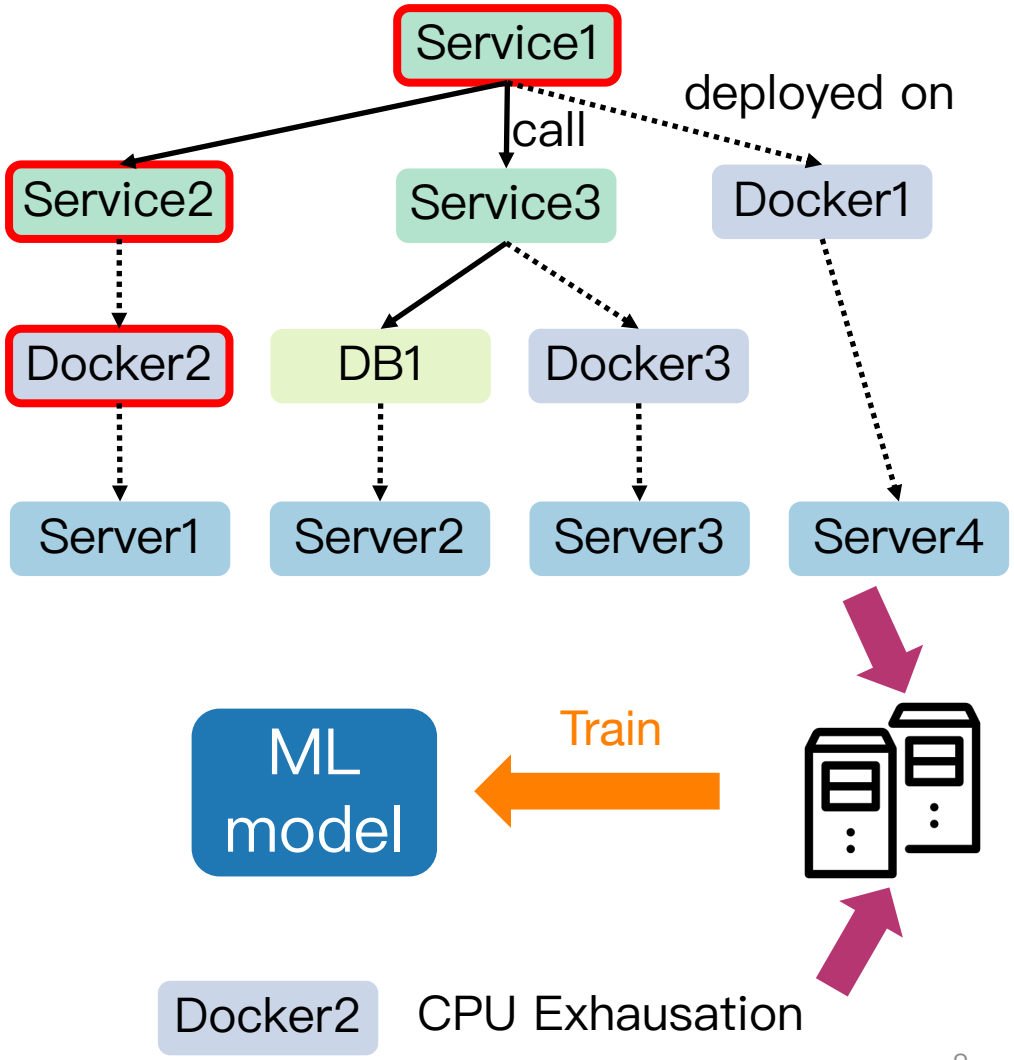


Figure 1: The number of recurring failures at *compA*.

Recurring failures are largely prevalent in practice.

Core Ideas

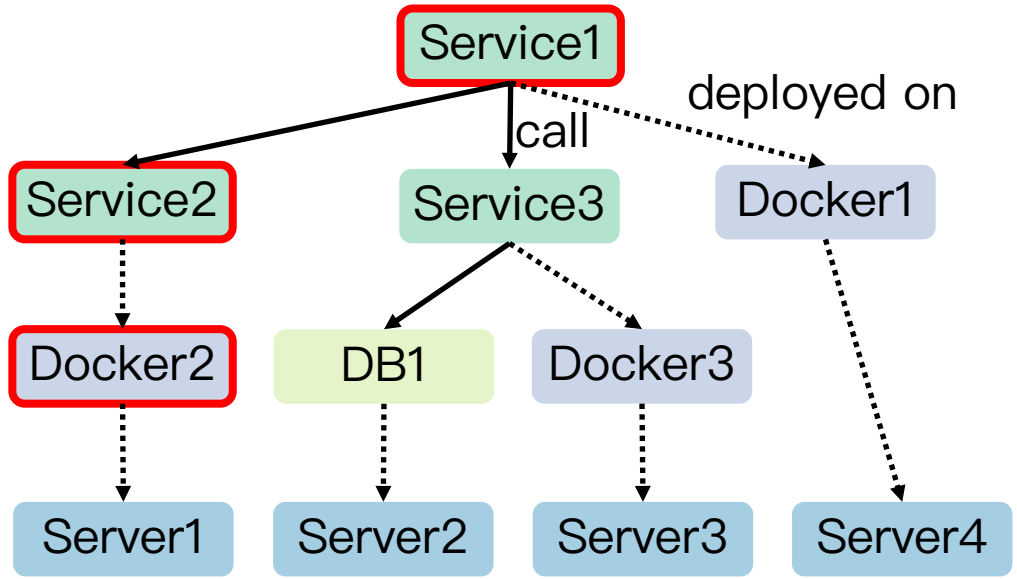

Service1
call
deployed on
Service2
Service3
Docker1
Docker2
DB1
Docker3
Server1
Server2
Server3
Server4
Maually
summarize
RCA rules
Docker2
CPU Exhausation

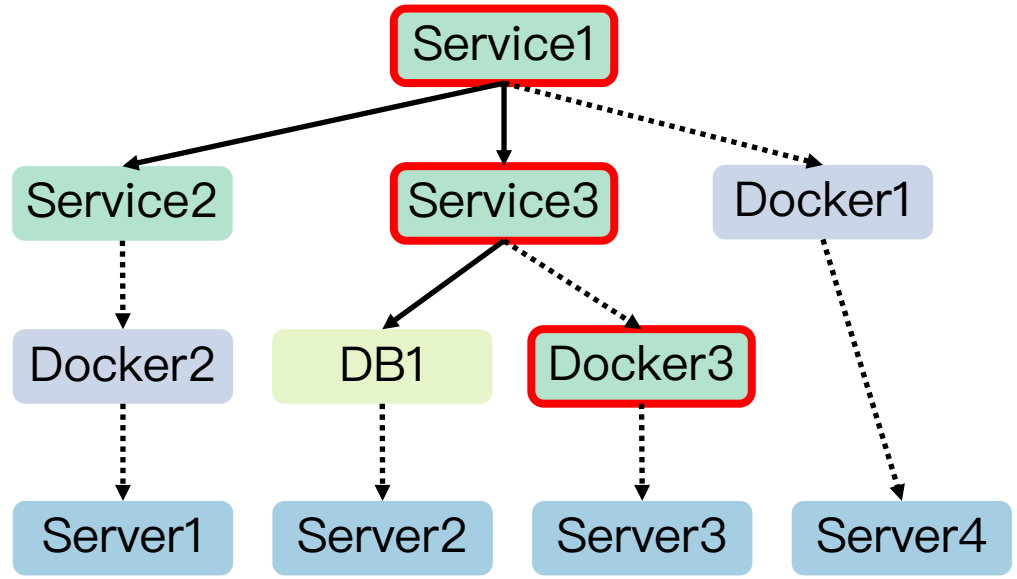

Service1
call
deployed on
Service2
Service3
Docker1
Docker2
DB1
Docker3
Server1
Server2
Server3
Server4
Train
ML
model
Docker2
CPU Exhausation

Core Ideas

Monitoring metrics and the dependency graph

Failure symptoms





ML model

Root causes

Faulty Failure Unit

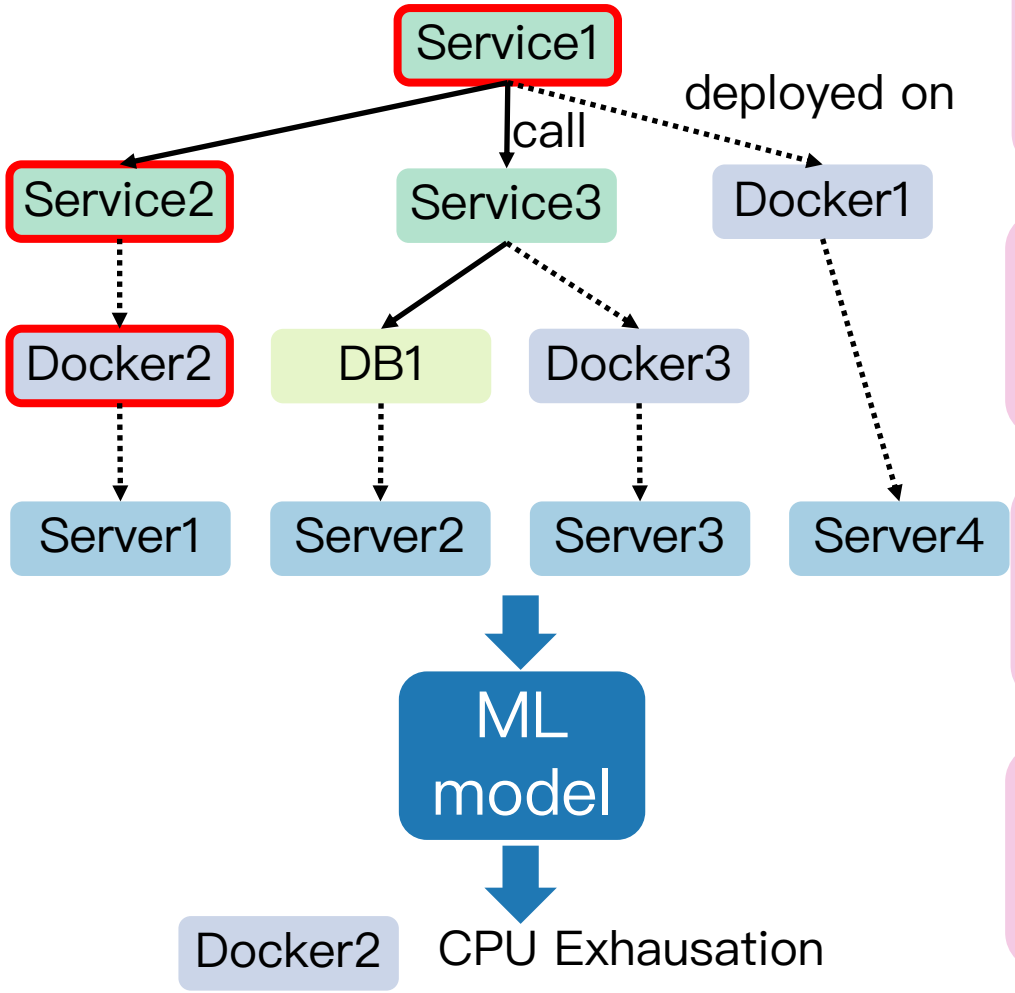
Docker2 CPU Exhausation

Fault location + fault type

Docker3 CPU Exhausation

Fault location + fault type

Challenges



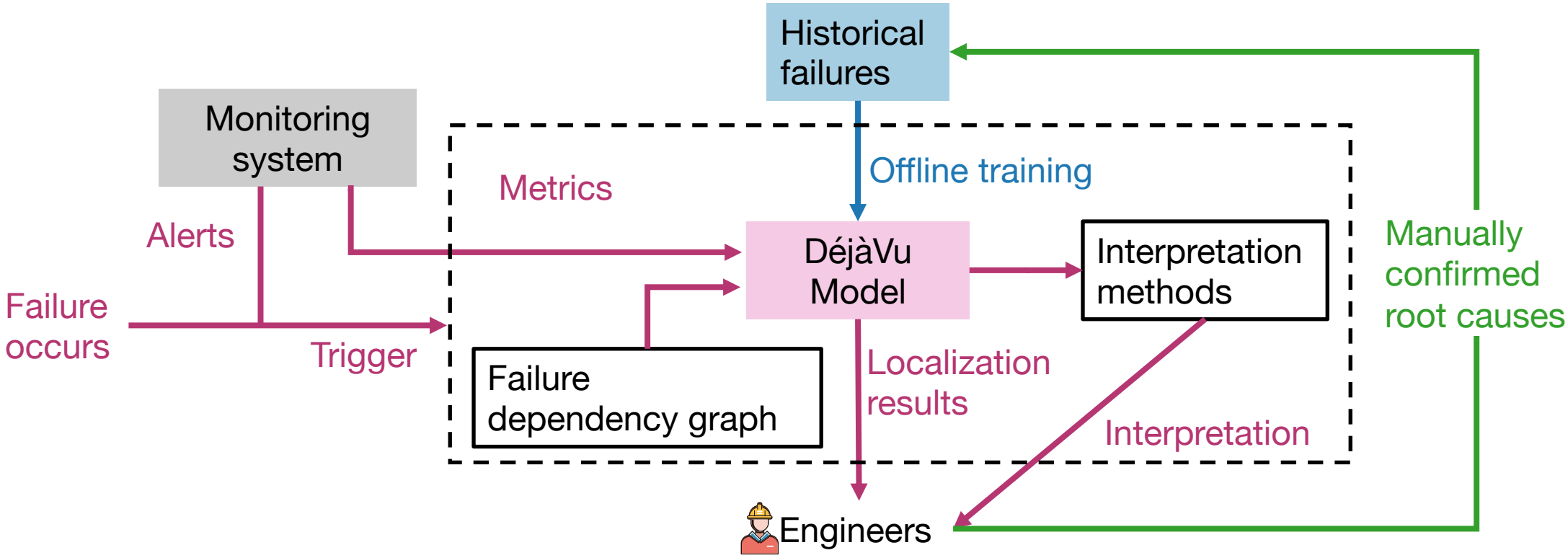
Various metrics are monitored on different components.

Failure propagates from faulty components due to the complex dependencies among components.

Generalizing to previously unseen failures (no failures of the same kinds have occurred at the same locations).

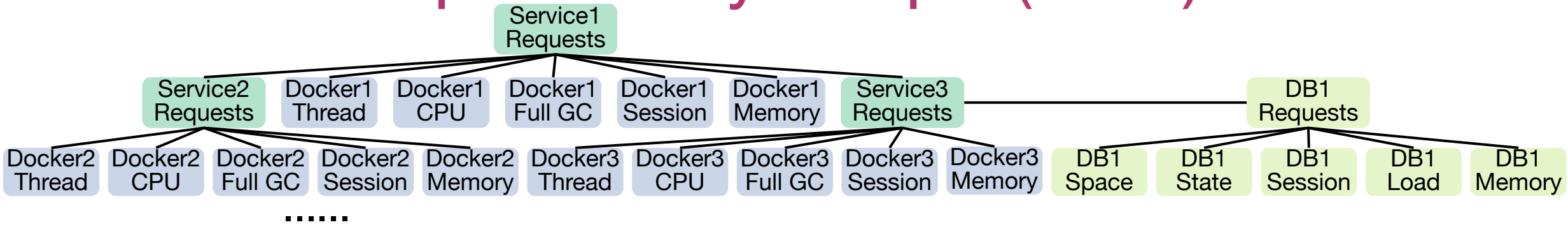
It is hard for engineers to trust the localization results from black-box models without interpretation.

DéjàVu Overview



The name DéjàVu comes from a French phrase (Déjà vu) which means“already seen”.

Failure Dependency Graph (FDG)



The vertices of an FDG is all the candidate failure units of an online service system

Service

DB

Docker

Server

CPU Exhaustion : CPU utilization, I/O wait...

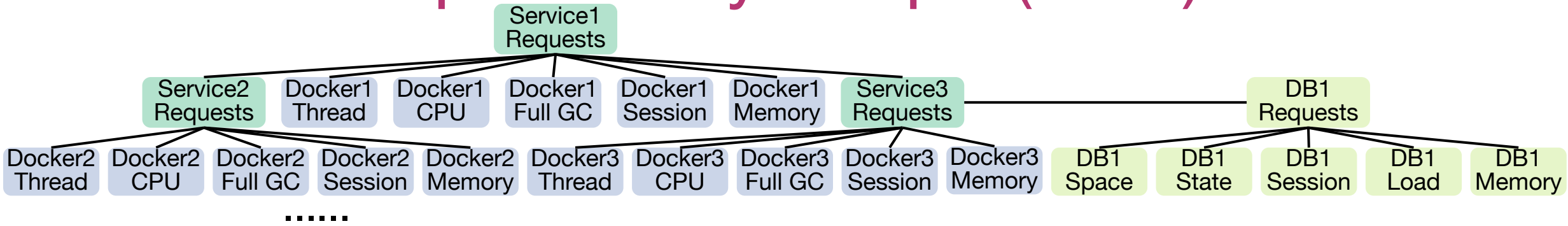
Packet loss： Recv/Sent packets， Recv/Sent bytes.....

Out of memry: free bytes, cached bytes, total bytes.....

Engineers are supposed to summarize the actionable fault types of historical failures and the corresponding input metrics

We can generate all the candidate failure units by combining all components and the corresponding fault types.

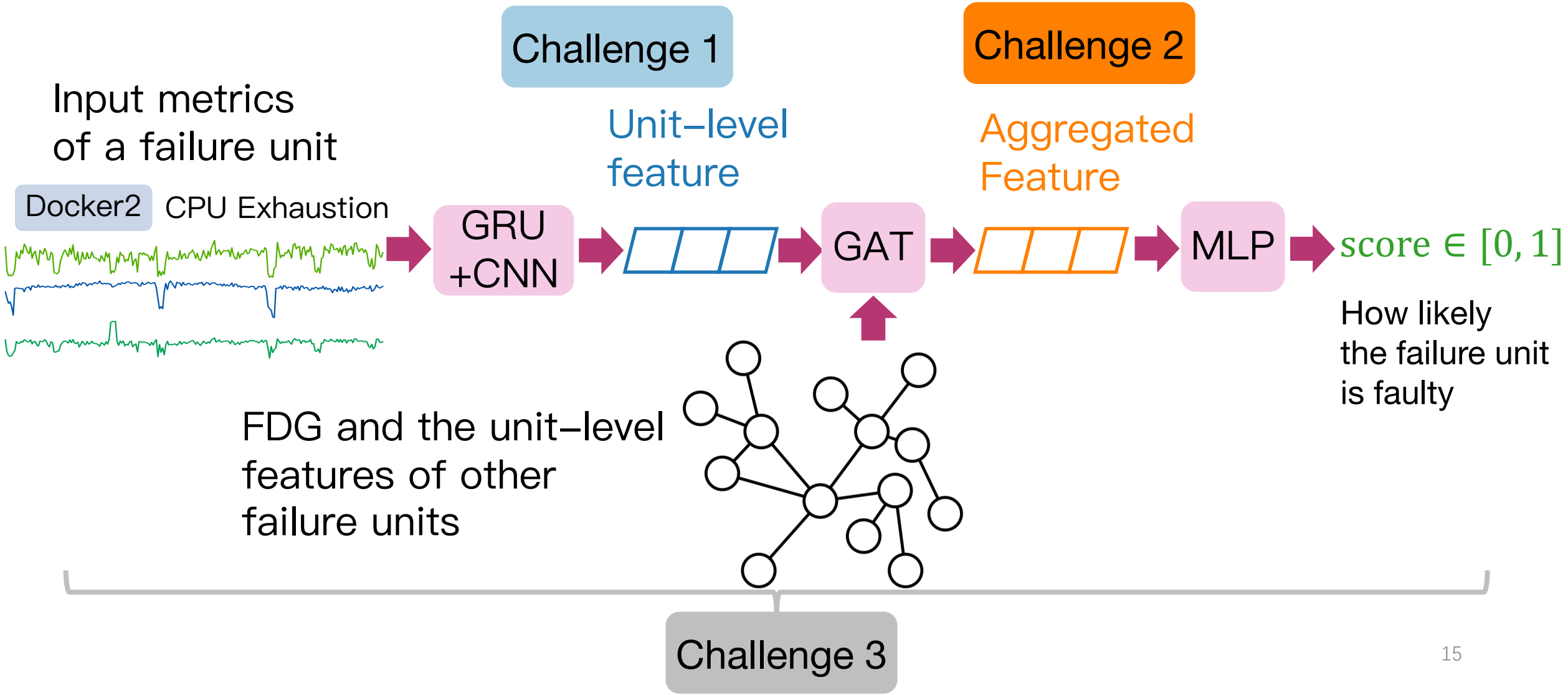
Failure Dependency Graph (FDG)



We connect the candidate failure units on an FDG by the call or deployment relationships of the components

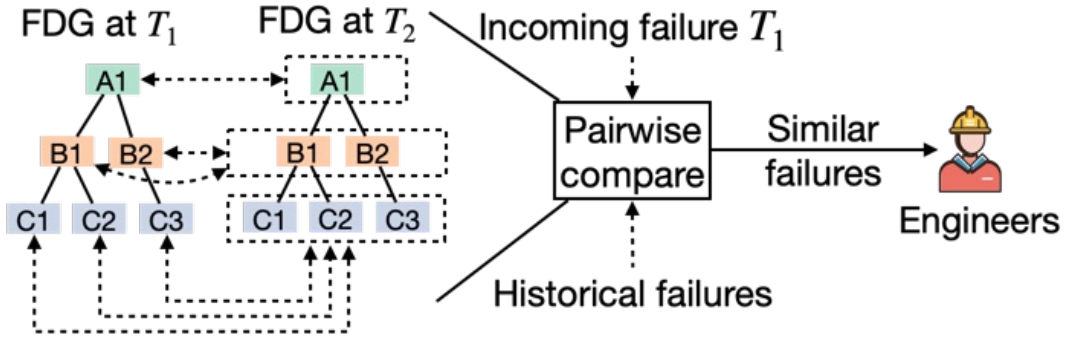
FDGs are automatically constructed for different failures since FDGs are dynamic due to frequent changes

Model Architecture



Interpretation Methods

Local interpretation: finding similar historical failures based on the aggregated features



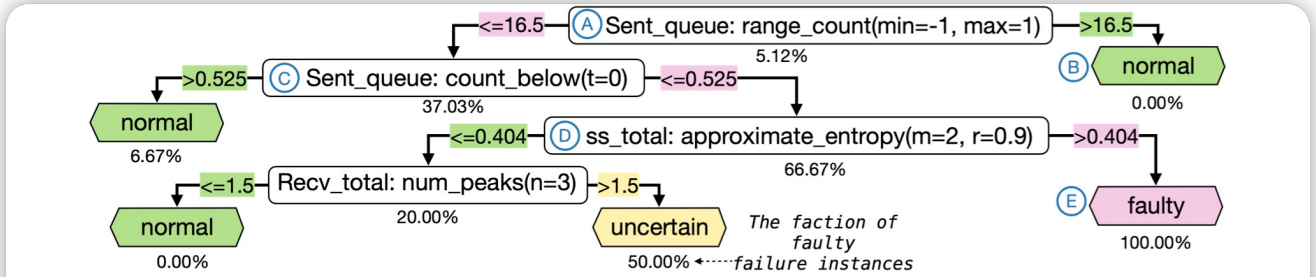


Figure 15: A decision tree trained for the failure class *OS Network* of $\mathcal{A}$. Note that the metrics values are normalized.

Global interpretation: using simpler but interpretable models to approximate (rather than outperform) it as accurately as possible.

Evaluation: Datasets

Dataset	#Failures	#Metrics	#Failure Units	#Failure Classes	System	Failures
$\mathcal{A}$	188	710	102	18	A production system	Injected
$\mathcal{B}$	158	2419	189	20	A production system	Injected
$\mathcal{C}$	99	2594	41	41	A production system	Real-world failures
$\mathcal{D}$	156	5724	1044	23	An open-source benchmark	Injected

Our experiments are conducted on four systems and 601 failures, including three production systems and 99 real-world failures.

The typical failure injection types include CPU/memory stress, database session limit, and network packet delay/loss.

Evaluation: Overall Performance

Category	Approach	Mean Average Rank (MAR)	Top-3 Accuracy	p-value (MAR, t-test)
Supervised	**DéjàVu**	**2.82**	**87.45%**	**---**
Similar failure matching	JSS'20	103.56	30.63%	1.1e-16
	iSUQAD	80.29	34.34%	1.8e-14
Traditional supervised ML	Decision Tree	49.25	63.90%	7.3e-17
	Gradient Boosting	12.83	71.59%	1.5e-10
	Random Forest	6.20	88.78%	2.1e-08
	SVM	33.34	26.48%	3.1e-11
Unsupervised heuristic	RandomWalk@Metric	36.19	19.13%	7.4e-17
	RandomWalk@FI	135.44	20.49%	8.7e-18

The localization performance of DéjàVu significantly outperforms the baselines.

Limitations and Future Work

Non-recurring failures: the localization models should be able to identify potential non-recurring failures and notify engineers.

Frequent changes: the localization models should be updated after every change without relying on new failures.

Thank you

https://github.com/NetManAIOps/DejaVu