

# Software Reliability Engineering: from Software Reliability Models to Software Resilience

Bruno Rossi

[brossi@mail.muni.cz](mailto:brossi@mail.muni.cz)

*Lasaris (Lab of Software Architectures and Information Systems)  
Faculty of Informatics  
Masaryk University, Brno, Czech Republic*



MUNI  
FI

# Structure

The presentation will be covering three parts:

Part 1: Software Reliability & Software Reliability Growth Models (SRGMs)

Part 2: Quality Models as Proxies for Failure Detection

Part 3: Moving towards Software Systems Resilience & Self-\* Capabilities

# Motivation – C4e Project - Critical Infrastructure

- **Critical Infrastructure** provide mission critical services - typically implemented as connected **Cyberphysical systems (CPS)**
- P1. Critical infrastructure protection:
  - P1.1 Simulation and predictive analysis of critical infrastructures
  - P1.2 Formal verification of critical infrastructures
  - P1.3 Recommendations for critical infrastructure realization
- Need to get a cohesive view, including cybersecurity and aspects related to cyber-law

C4e

# Software Reliability

**Software Reliability** is defined as the **probability of failure-free software operation** for a **specified period of time** in a **specific environment**

Probably one **one the most important qualities** of software systems as it can make a system inoperative

# ISO/IEC 25010 Standard – key terms

- ISO/IEC 25010 places four key terms under reliability:

**Maturity:** *“how well a system is able to meet the needs of reliability”*

**Availability:** *“how much a system is operational and accessible”*

**Fault Tolerance:** *“how well a system can operate despite hardware and/or software faults”*

**Recoverability:** *“how well a system can recover data in the event of a failure”*

# Software Reliability Engineering (SRE)

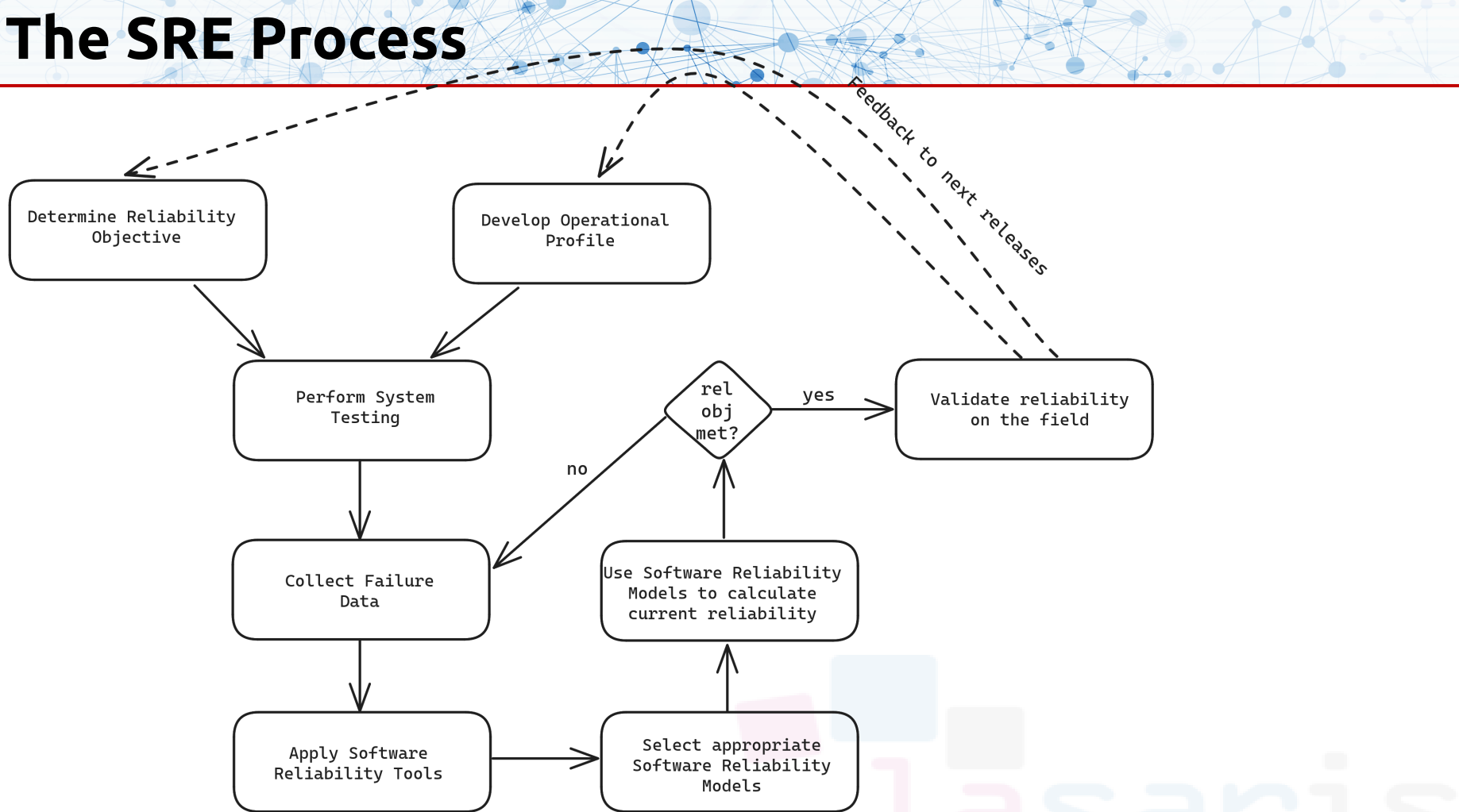
**Software Reliability Engineering (SRE)** is defined as the **quantitative** study of **operational behaviour** of **software-based systems** with respect to **user requirements concerning reliability**

**SRE includes:**

1. **Software reliability measurement** – estimation and prediction
2. **attributes and metrics of software design, development process, architecture and their impact on reliability**
3. usage of the **acquired knowledge** to guide the **design of software systems and development processes**

Lyu, Michael R. Handbook of software reliability engineering. Vol. 222. Los Alamitos: IEEE computer society press, 1996.

# The SRE Process



# Failure Rate & Hazard Rate

- **Reliability** is defined as the probability that a software system will **not fail** during the next  $x$  time units in a **specific environment**

**Failure Rate:** probability that a failure per unit of time occurs in the interval  $[t, t + \Delta t]$  given that a failure has not occurred before  $t$

$$\text{failure rate}(\lambda) = \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)}$$

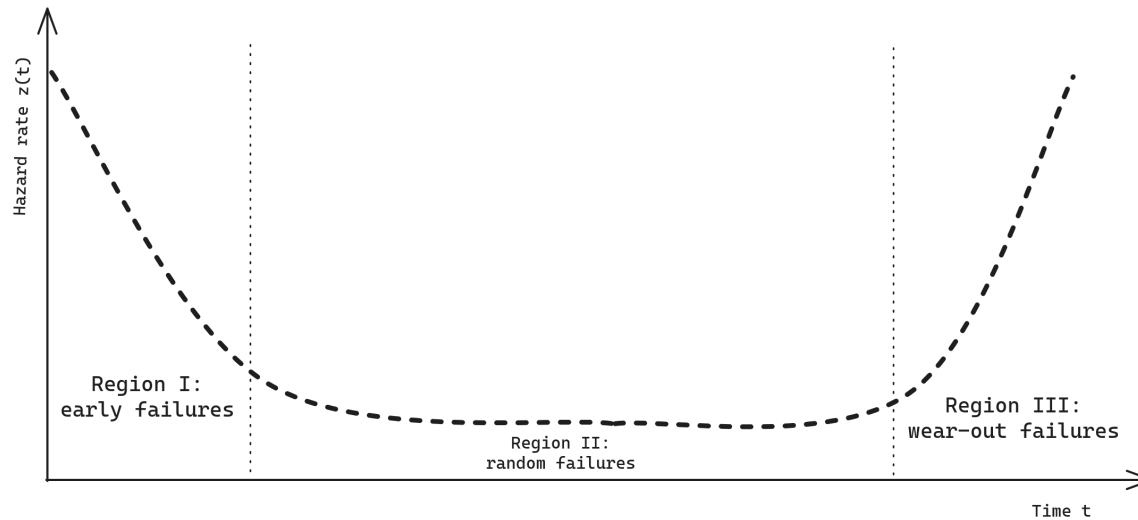
**Hazard rate**  $z(t)$ : the probability that the component fails in a short interval of time given it has survived up till that moment per unit interval of time

$$z(t) = \lim_{\Delta t \rightarrow 0} \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)} = \frac{f(t)}{R(t)}$$



# Main difference Hardware vs Software Reliability

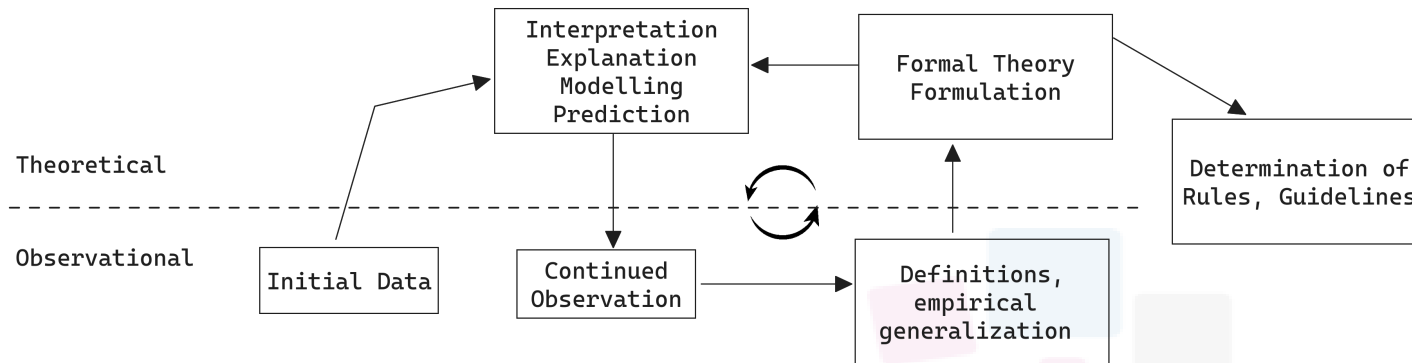
- Software **does not have a wear-out region** as in the hardware domain (in which hardware becomes *obsolete* and can lead to an increase in failures)



Typical hazard rate  $z(t)$  of a system / component

# Work of Lehman and Belady (1/3)

- Lehman started by a nine month study in **1968** to evaluate the IBM programming process, focusing on the OS/360 system
- After that experience Lehman and Belady joined in successive studies – laws of software evolution were defined in a time range from **1974 to 1996**
- The aim was to capture different **growth trends** of software systems and their **long term evolution**
- The laws apply to **E-type systems**: *“programs that mechanize a human or social activity”* (Lehman, 1980)



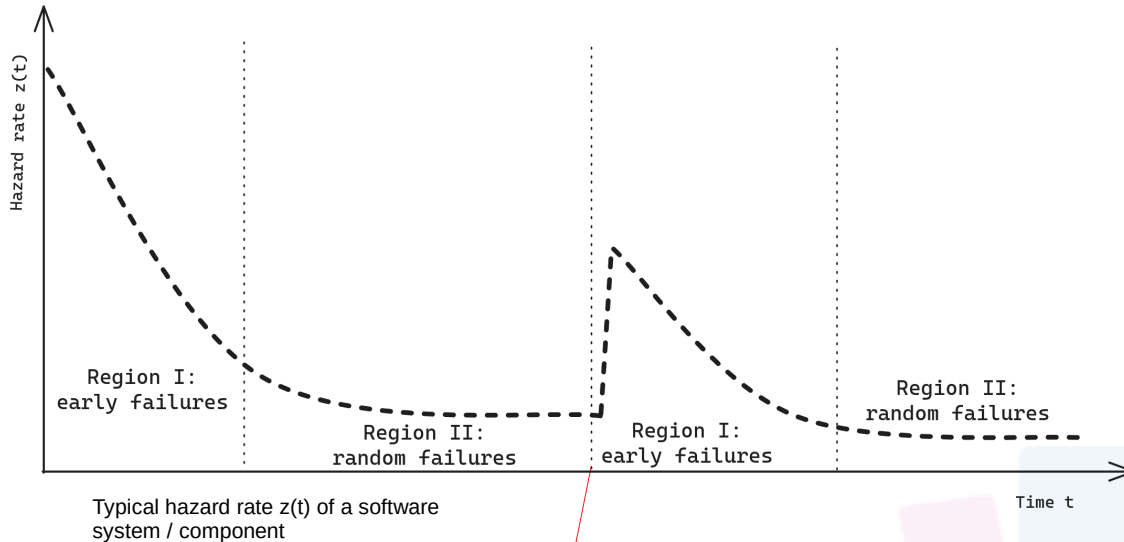
Source: Lehman, M., M., Ramil, J.,F. “Software evolution--Background, theory, practice,” Information Processing Letters, vol. 88, Ott. 2003, pagg. 33-44.

# What Lehman's Laws of Software Evolution tell us

L1. Continuing change	Systems must <b>continuously change</b> as otherwise they will become less useful
L2. Increasing complexity	Changes lead to <b>more complex structure</b> : activities need to be done to reduce complexity over time
L3. Large program evolution	Attributes such as size, errors are <b>invariant</b> for each system release
L4. Organizational stability	Rate of development is <b>approximately constant</b> over the lifecycle
L5. Conservation of familiarity	Incremental changes in each releases are <b>constant</b> over the lifecycle
L6. Continuing Growth	Functionality of the system has to <b>increase</b> to maintain satisfaction from the users
L7. Declining Quality	The quality of systems will <b>decline over time</b> if the system is not adapted to changes to the operation environment
L8. Feedback System	System improvements evolution should be considered as a <b>multi-loop feedback system</b> (e.g., integrating changing requirements from users)

# Some implications for SRE

- A model in Region I will **not work well** for Region II
- If in Software there is no Region III then the same model as in Region III could be applied – however, according to Lehman's laws quality of the systems decreases over time



Typical hazard rate  $z(t)$  of a software system / component

New software release



# Software Reliability Growth Models (SRGMs)



# Software Reliability Growth Modelling

## SRGM

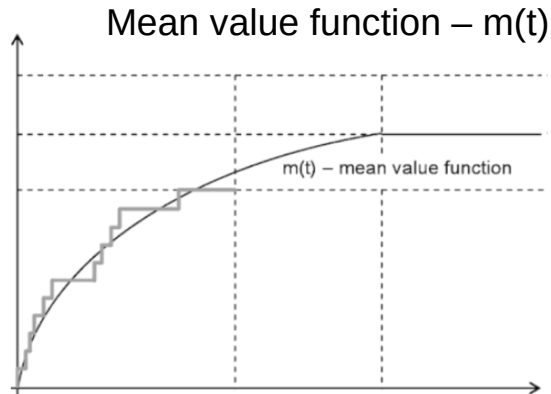
”If the history of fault detection and removal follows a particular recognizable pattern, it is possible to describe the mathematical form of the pattern“

# Software Reliability Growth Modelling

## SRGM

"If the history of fault detection and removal follows a particular recognizable pattern, it is possible to describe the mathematical form of the pattern"

Fitting the cumulative failures over time

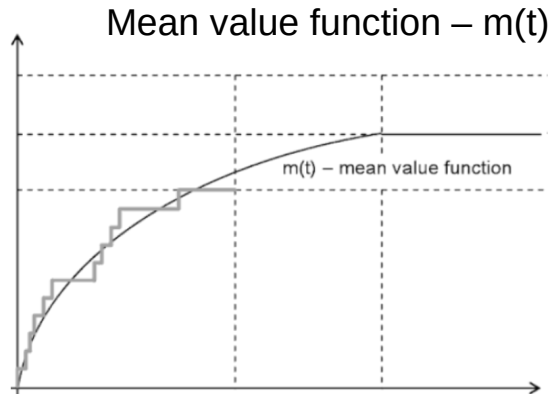


# Software Reliability Growth Modelling

## SRGM

”If the history of fault detection and removal follows a particular recognizable pattern, it is possible to describe the mathematical form of the pattern“

Fitting the cumulative failures over time

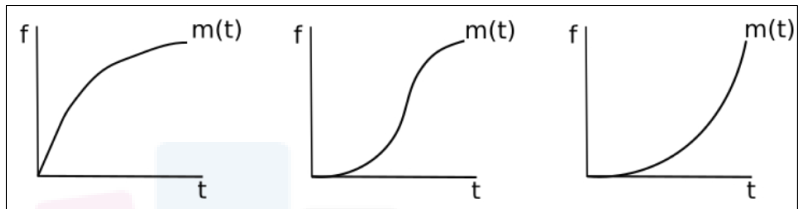


Types of models

**Concave models** – assume the total number of faults in software is finite, and that it is possible to achieve fault-free software in finite time

**S-shaped models** – they also assume that the total number of faults is finite. Early testing is not as effective in fault discovery as the testing in the later stages. Therefore, there is a period in which the number of faults is increasing

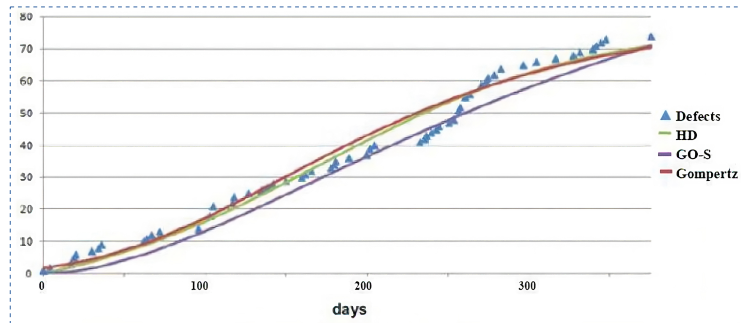
**Infinite models** – assume that it is not possible to develop fault-free software because during fault removal we can introduce new ones





# One of the earliest studies...

- One of the first papers\* to apply **SRGMs** to **Open Source software projects**
- Comparing several models, like Weibull, Hossain Dahiya (HD), Goel Okumoto S-shaped (GO-S), Gompertz
- Three projects analyzed: Mozilla Firefox, LibreOffice, OpenSuse
- Generally the **Weibull model** was found to be the best in terms of **Goodness of Fit (GoF)**
- However, no model was generally good for **predictive capability**



\* Rossi, B., Russo, B., & Succi, G. (2010). Modelling failures occurrences of open source software with reliability growth. In Open Source Software: New Horizons: 6th International IFIP WG 2.13 Conference on Open Source Systems, OSS 2010, Notre Dame, IN, USA, May 30–June 2, 2010 (pp. 268-280). Springer Berlin Heidelberg.

# ...and how SRGMs have been used so far

1. Large heterogeneity of results in terms of the best models
2. Generally small samples of projects analyzed

Ref	Models	Best Models	#OSS	GoF
Rahmain et al. 2010 [19]	GOS, SCH, WE	WE	5	R <sup>2</sup>
Mohamed et al. 2008 [18]	GO, GOS	--	2	R <sup>2</sup>
Zhou et al. 2005 [20]	WE	WE	8	R <sup>2</sup>
Rossi et al. 2010 [21]	WE, WES, GO, GOM GOS, HD, YE	WE	3	R <sup>2</sup> , AIC
Tamura et al. 2005 [24]	GO, HD, LP	LP	1	AIC, MSE
Ullah et al. 2012 [23]	MO, HD, GO, GOS, WE, GOM, LOG, YE	GOM, HD	6	R <sup>2</sup>
Wang et al. 2021 [17]	GO, GOS, ISS, PNZ, PZ, WM, L, W	W	3	R <sup>2</sup> , MSE

\*SCH - Schneidewind model [19], WES - Weibull S-Shaped model [25], GOM - Gompertz model [26], LOG - Logistic model [27], LP - Logarithmic Poisson Execution Time model [24], MSE - mean squared error, ISS - Inflection S-Shaped model [28], PNZ - Pham-Nordmann-Zhang model [29], PZ - Pham-Zhang model [30], WM - Wang-Mi model [31], L - Li model [32], W - Wang model [17]

# STRAIT Tool (1/3)

## STRAIT

A tool to mine failure data from software repositories, build SRGM based on projects' snapshots\*

### Typical process

1. Getting issue reports from data sources
2. Creation of snapshots and persistence storage
3. Data processing & filtering
4. Building of pluggable SRGM models (trend test, parameters estimation, GoF metrics)
5. Outputting module

# STRAIT Tool (2/3)

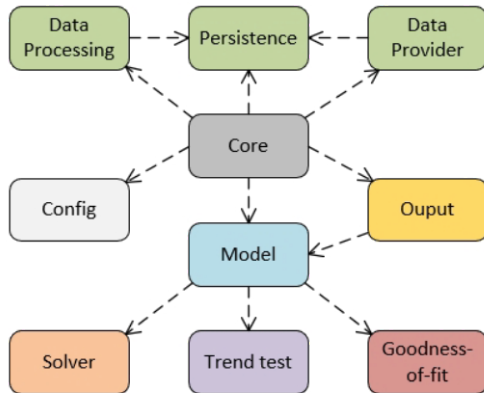
## STRAIT

A tool to mine failure data from software repositories, build SRGM based on projects' snapshots\*

### Typical process

1. Getting issue reports from data sources
2. Creation of snapshots and persistence storage
3. Data processing & filtering
4. Building of pluggable SRGM models (trend test, parameters estimation, GoF metrics)
5. Outputting module

### Components



# STRAIT Tool (3/3)

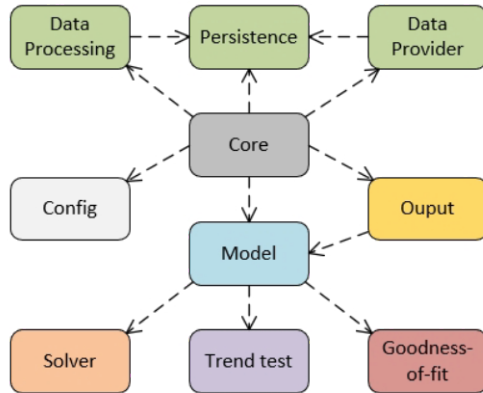
## STRAIT

A tool to mine failure data from software repositories, build SRGM based on projects' snapshots\*

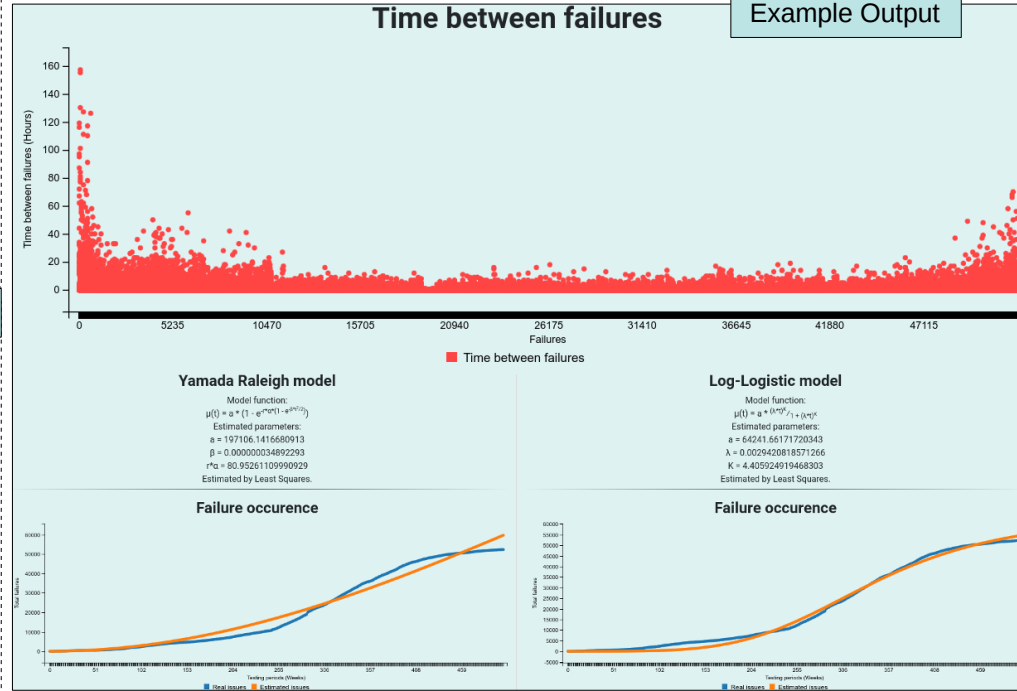
### Typical process

1. Getting issue reports from data sources
2. Creation of snapshots and persistence storage
3. Data processing & filtering
4. Building of pluggable SRGM models (trend test, parameters estimation, GoF metrics)
5. Outputting module

### Components



### Example Output



# Experimental Evaluation

1. We adapted and used **STRAIT** to mine data from **GitHub bug tracking repositories**: top ten projects from different topics of the "Topic Lists" and combined them with ten more projects from the "Trending List"
2. We run STRAIT in a **Cloud environment** (16 threads / 128GB RAM) for increased performance
3. We fitted 792 SRGMs (**88 projects x 9 models**) with 383K software defects for RQ1, RQ2, and additionally 261 SRGMs for software releases (29 releases x 9 models) in RQ3

Model	Type	$\mu(t)$	Implemented models
Goel-Okumoto (GO)	Concave	$a(1 - e^{-bt})$	
Goel-Okumoto S-Shaped (GOS)	S-Shaped	$a(1 - (1 + bt)e^{-bt})$	
Hossain-Dahiya (HD)	Concave	$a(1 - e^{-bt}) / (1 + ce^{-bt})$	
Musa-Okumoto (MO)	Infinite	$\alpha \ln(\beta t + 1)$	
Duane (DU)	Infinite	$\alpha t^\beta$	
Weibull (WE)	Concave	$a(1 - e^{-bt^c})$	
Yamada Exponential (YE)	Concave	$a(1 - e^{-r\alpha(1 - e^{-\beta t})})$	
Yamada Raleigh (YR)	S-Shaped	$a(1 - e^{-r\alpha(1 - e^{-\beta t^2/2})})$	
Log-Logistic (LL)	S-Shaped	$a(\lambda t)^\kappa / (1 + (\lambda t)^\kappa)$	

# Aims of the experimental evaluation

## Research Questions

RQ1. What is the ranking of the models based on GoF?

RQ2. How does the project's domain affect the GoF of models?

RQ3. Does project division into OSS releases change the applicability of SRGMs?

# Used Metrics (GoF)

$R^2$  (coefficient of determination)

$$R^2 = 1 - \frac{\text{sum squared regression (SSR)}}{\text{total sum of squares (SST)}},$$
$$= 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}.$$

How well the model fits the outputs  
(range 0-1)

Akaike Information Criterion (AIC)

$$AIC = 2K - 2 \ln(L)$$

Indicators about the quality of the models, penalizing  
models with higher nr of parameters

where:

$K$  – the number of estimated parameters in the model,  
 $L$  – the likelihood of the model given the data,  
 $n$  – the size of the dataset.

Bayesian Information Criterion (BIC)

$$BIC = K \ln(n) - 2 \ln(L)$$

Residual Standard Error (RSE)

$$RSE = \left( \sum_{i=1}^n (y_i - f(x_i))^2 / (n - 2) \right)^{1/2}$$

How well the model fits the outputs (in  
the unit of dependent variable)

where:

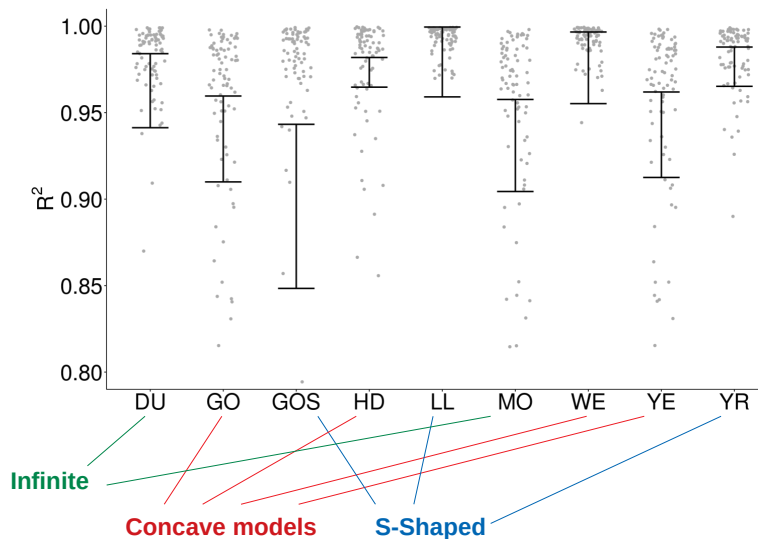
$y_i$  – the  $i^{th}$  observed value,  
 $f(x_i)$  – the  $i^{th}$  predicted value,  
 $n$  – the size of the dataset.



# RQ1 – Ranking of Models

To answer this RQ, we considered 792 SRGMs fitted on the whole dataset with 383 788 software defects.

Model	R <sup>2</sup>		AIC		RSE	
	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$
LL	0.979	0.094	3,374	1771.464	83.907	188.828
YR	0.977	0.052	3,806	1847.782	175.468	445.723
WE	0.976	0.096	3,411	1782.277	81.158	164.656
HD	0.973	0.039	3,769	1890.365	145.866	380.665
DU	0.963	0.099	3,708	1947.489	147.956	395.543
YE	0.937	0.115	3,889	2087.951	233.414	604.541
GO	0.935	0.115	3,896	2078.743	234.270	604.028
MO	0.931	0.123	3,883	2073.052	230.064	606.065
GOS	0.896	0.220	3,767	1942.052	180.096	438.825



## RQ1 Findings

Based on 792 fitted SRGMs, considering the R<sup>2</sup> metric LL, YR, WE, HD, DU are the best models. GO, GOS, MO, YE show the highest variance than other models. GOS is in general the worse model in terms of R<sup>2</sup>.

# RQ2 – Project Domain

To answer this RQ, we considered 792 SRGMs fitted on the whole dataset with 383 788 software defects and segmented by categories

Category	#	Defects (AVG.)	Category	#	Defects (AVG.)
C1 Admin/monitoring	14	10012	C5 SW Development	16	1876
C2 Cryptocurrency	9	2527	C6 System/OS tools	10	8671
C3 DB and data analysis	10	3327	C7 Text processing	7	833
C4 Multimedia	10	5286	C8 Web/Networking	12	1017

Model	C1		C2		C3		C4		C5		C6		C7		C8	
	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$
DU	0.989	0.008	0.982	0.016	0.850	0.269	0.976	0.027	0.982	0.016	0.976	0.020	0.952	0.076	0.960	0.080
GO	0.949	0.056	0.971	0.018	0.766	0.283	0.952	0.061	0.958	0.040	0.970	0.027	0.934	0.050	0.945	0.079
GOS	0.842	0.331	0.949	0.114	0.774	0.255	0.845	0.259	0.872	0.256	0.987	0.008	0.969	0.026	0.980	0.017
HD	0.970	0.043	0.977	0.021	0.988	0.011	0.968	0.072	0.966	0.033	0.982	0.028	0.954	0.041	0.988	0.008
LL	0.996	0.002	0.993	0.009	0.871	0.277	0.993	0.007	0.989	0.009	0.994	0.009	0.984	0.009	0.994	0.055
MO	0.947	0.058	0.970	0.018	0.757	0.301	0.947	0.059	0.957	0.040	0.967	0.026	0.910	0.103	0.952	0.021
WE	0.995	0.003	0.993	0.009	0.865	0.274	0.993	0.008	0.987	0.011	0.994	0.008	0.958	0.077	0.993	0.008
YE	0.950	0.056	0.971	0.018	0.762	0.286	0.952	0.061	0.958	0.040	0.970	0.027	0.935	0.050	0.969	0.028
YR	0.985	0.017	0.986	0.014	0.921	0.165	0.984	0.018	0.975	0.030	0.987	0.009	0.970	0.023	0.985	0.012

## RQ2 Findings (shortened)

The GOS model, while statistically worse than all other models when considering the whole dataset, has some domains in which it has low variance and good  $R^2$  rankings.

# RQ3 – Impact of Project Releases

To answer this RQ, we used 63 SRGMs (7 projects with releases fitted by 9 models) and 261 SRGMs (29 releases, 9 models, 6 800 defects)

Rankings of models based on R2 considering Releases (R) and whole projects (NR)

	R <sup>2</sup>	
	R	NR
DU	3	3
GO	7	8
GOS	9	9
HD	4	4
LL	1	1
MO	6	7
WE	2	2
YE	5	6
YR	8	5

## RQ3 Findings (shortened)

Considering projects as a whole or releases mostly does not have an impact on the rankings of the models in terms of R<sup>2</sup> (the top-3 models remain the same).

# Major Challenges for SRGMs

**“Faults are repaired immediately when they are discovered”** → this might not be the case – however, filtering of issues can help (e.g., *duplicate reports*)

**“Failures are independent”** → One of the strongest assumptions of the models are that each failure is independent and does not cause additional failures

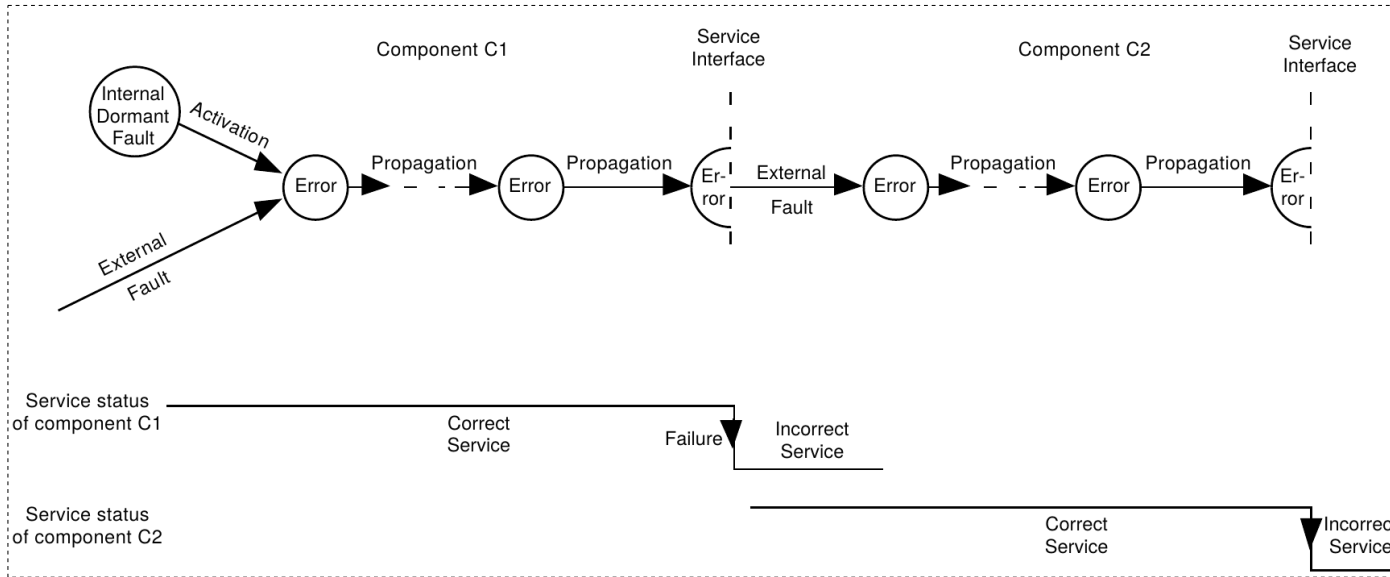
**“Fault repairs are perfect”** → fault repairs likely introduce new faults, so this can have an impact on the model

**“Tests represent operational profile”** → the models expect that the tests are representative of the usage of the system. Nowadays there are workload generators and test case generators, so there can be a high similarity about the two processes

**“No new code is introduced during testing”** → new code is frequently introduced throughout the entire test period (faults repair and new features), especially for agile development. This is accounted for in parameter estimation since real faults discoveries are used. However, the shape of the curve may be changed (i.e., make it less concave)

# Alternative methods

They look more at the propagation and chaining of faults and failures



Source diagram: Avizienis, A., Laprie, J. C., & Randell, B. (2001). Fundamental concepts of computer system dependability. In Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments (pp. 1-16).

- Bayesian networks
- Fault trees and Markov chains
- Stochastic Petri Nets and Markov chains

# Major Takeaways

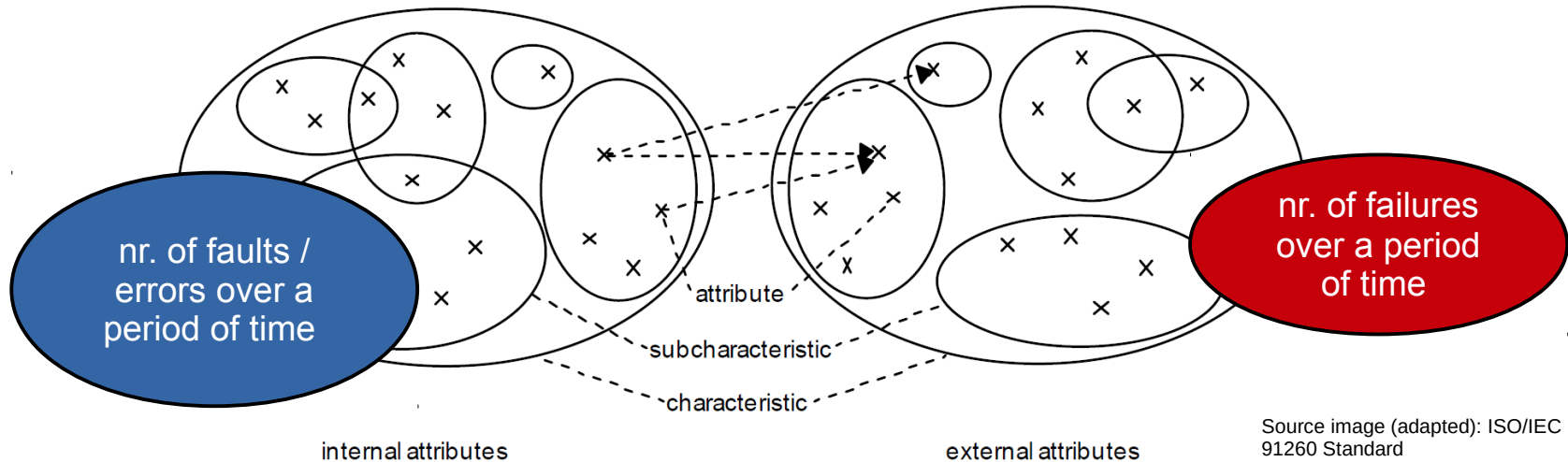
- **SRE** deals with Software Reliability measurement, estimation and prediction.
- **SRGMs** are useful to model the cumulative reliability of software systems and are part of the SRE process.
- **SRGMs** can be evaluated according to different metrics, both looking at the *goodness of fit* and at the *predictive capabilities*. Being an *inductive model*, is not possible to establish a model that will work best apriori.



# Quality Models as Proxies for Failures Detection

# How IEEE maps Failures to Quality

The mapping of internal attributes to external ones is a key aspect in software reliability



**“When failure data is not available, metrics from the software development process can be used to estimate the reliability of the software”** Lyu, Michael R. Handbook of software reliability engineering. Vol. 222. Los Alamitos: IEEE computer society press 1996.



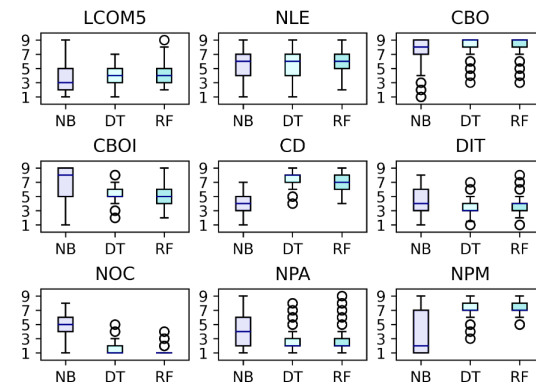
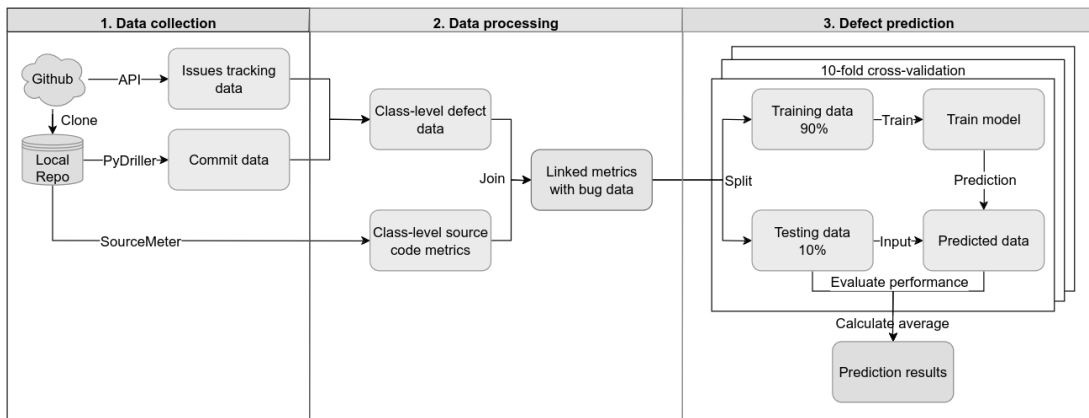
# ...indeed many metrics were used in the prediction models

Year	Metrics	Prediction model
2020	<b>Source code metrics:</b> 60 class-level complexity, size and object-oriented metrics	Deep neural networks
2019	<b>Source code metrics:</b> ATFD, ATLD, CC, CDISP, CINT, CM, FanOut, LOC, LOCNAMM, MaMCL, MAXNESTING, MeMCL, NMCS, NOAM, NOLV, NOMNAMM, NPA, TCC, WOC <b>Code smell metric:</b> Code smell intensity	Simple Logistic, Decision Tree Majority, Naive Bayes, Logistic Regression
2013	<b>Source code metrics:</b> LOC, MLOC, PAR, NOF, NOM, NOC, CC, DIT, LCOM, NOT, WMC <b>Process metrics:</b> PRE, Churn <b>Antipattern metrics:</b> ANA, ACM, ARL, and ACPD	Various intra-system and cross-system
2011	<b>Explored metrics:</b> log(KLOC), Prior faults, Prior changes, Prior changed, Prior developers, Prior lines added/deleted/modified	Binomial regression
2010	<b>Source code metrics:</b> WMC, DIT, NOC, RFC, LCOM, CBO, LCOM3, NPM, DAM, MOA, MFA, CAM, IC, CBM, AMC, Ca, Ce, CC, LOC	Kohonen's neural network
2010	<b>Source code metrics:</b> WMC, DIT, NOC, RFC, LCOM, CBO, LOC, FanIn, FanOut, NOA, NPA, NOPRA, NOAI, NOM, NPM, NOPRM, NOMI <b>Change metrics:</b> EDHCM, LDHCM, LGDHCM	Generalized linear regression
2009	<b>Change complexity metrics:</b> BCC, ECC, HCM	Statistical linear regression
2001	<b>Inheritance metrics:</b> DIT, NOC <b>Coupling metrics:</b> ACAIC, ACMIC, DCAEC, DCMEC, OCAIC, OCAEC, OCMIC, OCMEC	Calibrated logistic regression

Rebro, D. A., Chren, S., & Rossi, B. (2023). Source Code Metrics for Software Defects Prediction. In Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (pp. 1469-1472).

# Software Defects Prediction Process

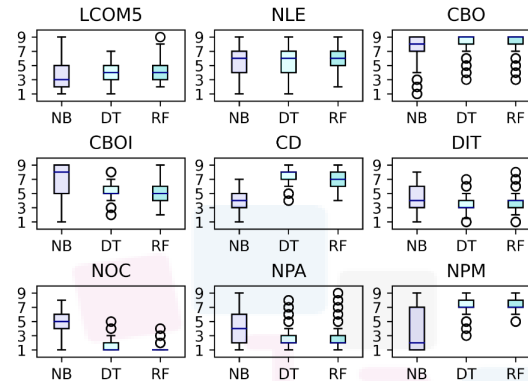
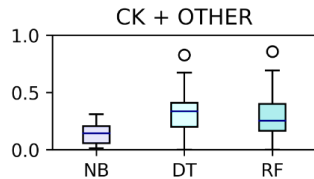
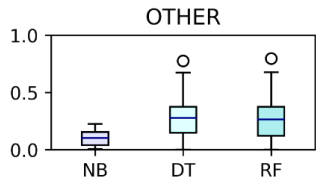
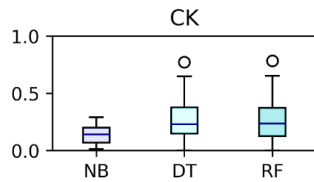
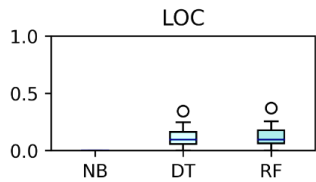
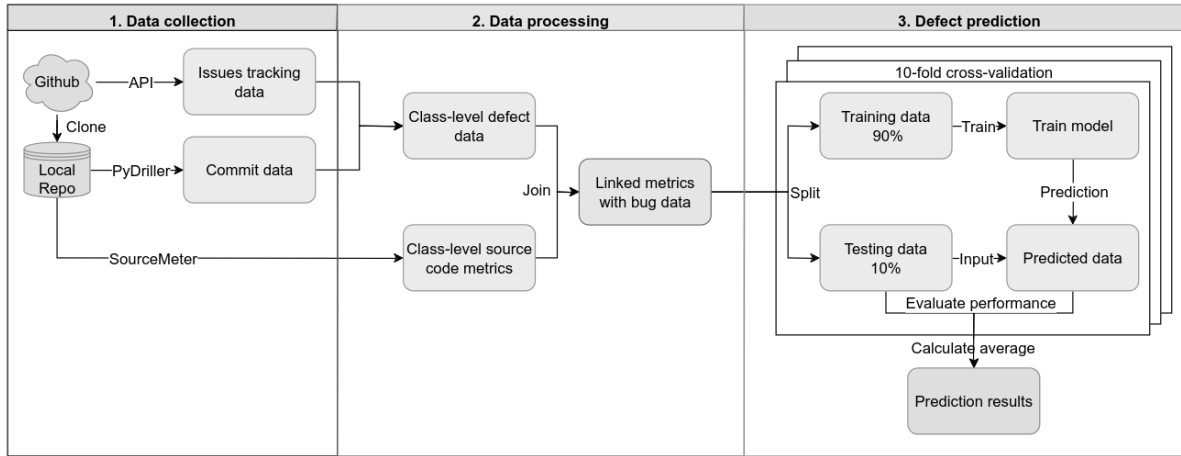
**Software Defects Prediction:** area that attempts to build models to predict parts of software more prone to software defects based on previous history. Likely one of the most studied area in Software Engineering.



Example: ranking of metrics in the prediction model (the lower the better)

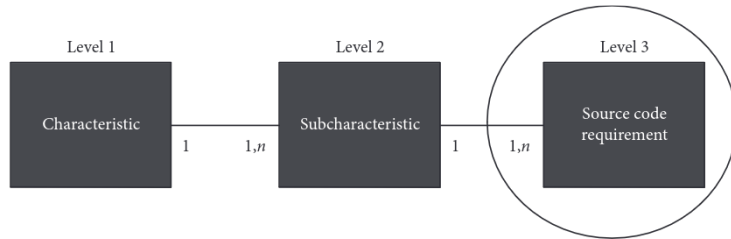
Rebro, D. A., Chren, S., & Rossi, B. (2023). Source Code Metrics for Software Defects Prediction. In Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC)

# Generic process of defects prediction



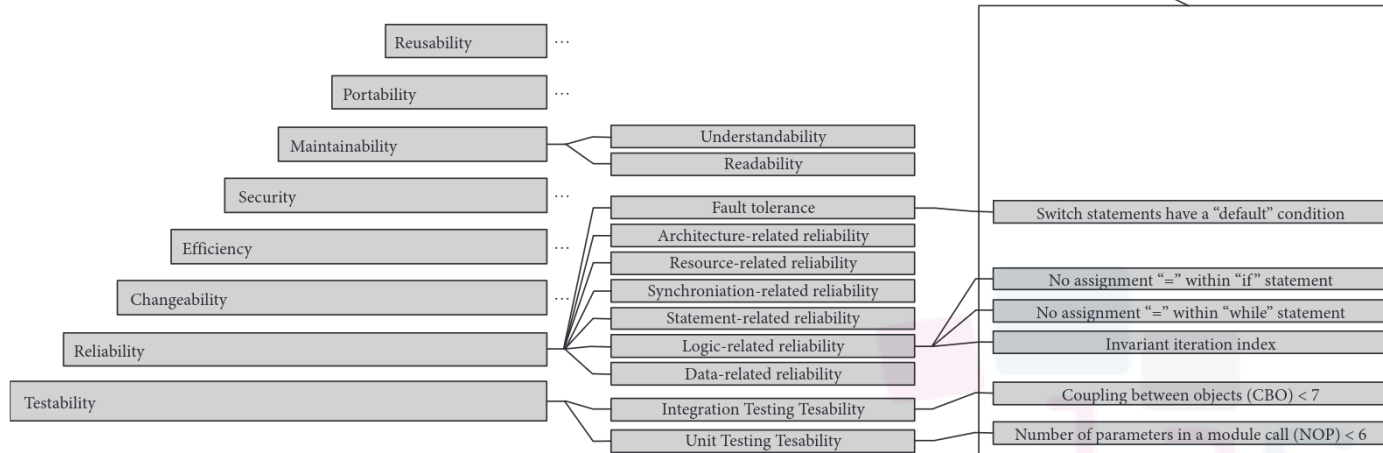
# Quality Models (example: SQALE)

- Many quality models were developed over time – the assumption is: **control the quality and you will control the failures**



In short, the model defines a Remediation Cost (RC) to fix all the violations:

$$RC = \frac{\sum_{rule} effortToFix(violations_{rule})}{8[hr/day]}$$



# Major Challenges


- Identifying which **modules** are **more defects prone**
- Identifying the importance of **features** for the prediction
- Considering **changes in history of a project** (*drifts*)
- Dealing with **imbalanced data**
- **Associating** defects to implementations activities
- **Integration of the models** into *running systems*
- **Mining** representative datasets (e.g., *NASA dataset* has been used for long time in SE)

# My Research Focus in this Area

- Evaluating the impact of **several metrics** on the **defect prediction of models**
  - Rebro, D. A., Chren, S., & Rossi, B. (2023). Source Code Metrics for Software Defects Prediction. In Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (pp. 1469-1472).
- Evaluating the **comparability of different maintainability indexes** (SQALE, MI, SIG-TD)
  - Strečanský, P., Chren, S., & Rossi, B. (2020). Comparing maintainability index, SIG method, and SQALE for technical debt identification. In Proceedings of the 35th Annual ACM Symposium on Applied Computing (pp. 121-124).
- Studying **bug triaging** in both Open Source Software and one company involved
  - Dedík, V., & Rossi, B. (2016). Automated bug triaging in an industrial context. In 2016 42th Euromicro conference on software engineering and advanced applications (SEAA) (pp. 363-367). IEEE.
- Evaluating the applicability of **Mutation Testing** in a industrial context
  - J. Možucha and B. Rossi. Is mutation testing ready to be adopted industry-wide? In P. Abrahamsson, A. Jedlitschka, A. Nguyen Duc, M. Felderer, S. Amasaki, and T. Mikkonen, editors, Product-Focused Software Process Improvement, pages 217–232, Cham, 2016. Springer International Publishing

# Major Takeaways

- **Software Defect Prediction** can be considered as a **proxy** of failure prediction when failure data is not available.
- **Software Quality models** were developed over time to model different aspects of software systems, mainly based on different set of metrics.
- **Software metrics & Software Quality Models** have been used successfully in prediction models to predict software defects.



# Software Systems Resilience & Self-\* capabilities

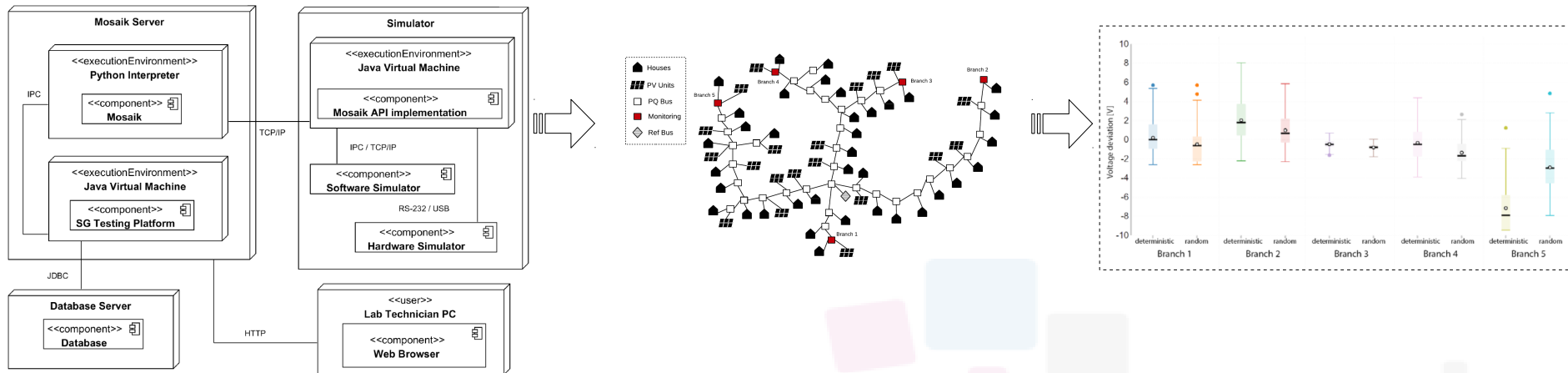




# Motivation (1/2)

- In previous work we created a **testing management platform** for Smart Grids based on the **Mosaik** framework for **co-simulations**
- We extended Mosaik with the disconnect method to remove edges from the dataflow graph and the entity graph → A simple way to simulate node failures
- This can be useful to understand the patterns of failures

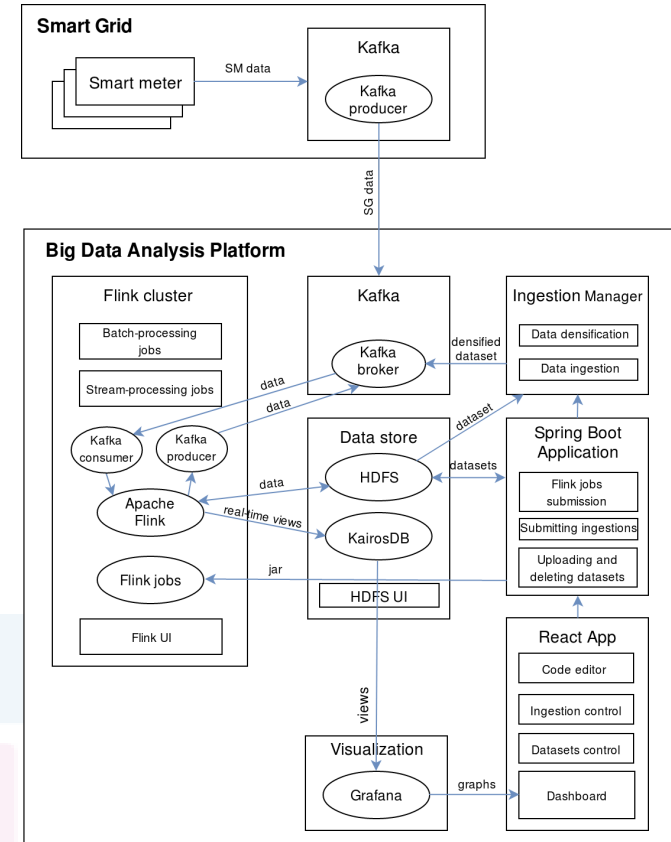
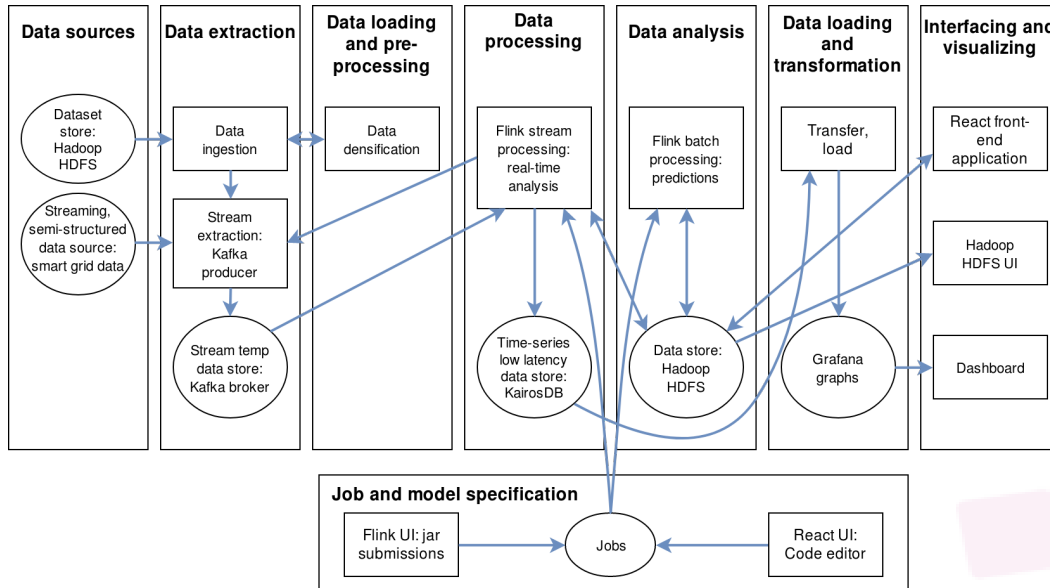
## Smart Grids Testing Processes



- Mihal, P., Schvarcbacher, M., Rossi, B., & Pitner, T. (2022). Smart grids co-simulations: Survey & research directions. Sustainable Computing: Informatics and Systems.
- Schvarcbacher, M., Hrabovská, K., Rossi, B., & Pitner, T. (2018). Smart grid testing management platform (sgtmp). Applied Sciences, 8(11), 2278.
- Gryga, L., & Rossi, B. (2021). Co-simulation of Smart Grids: Dynamically Changing Topologies in Failure Scenarios. In Complexis.

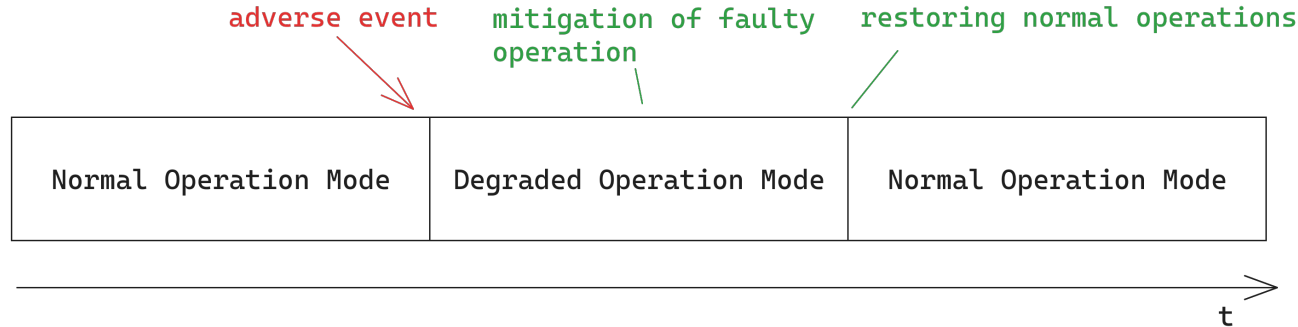
# Motivation (2/2)

- In the **CERIT-SC Big Data** project we looked into anomalies for power consumption data
- Built a Big Data platform based on Apache Flink that could integrate anomaly detection algorithms



# Software Systems Resilience

**Resilience:** ability of a system to **self-heal**, **recover**, and **continue operating** after encountering failure, outage, security incidents

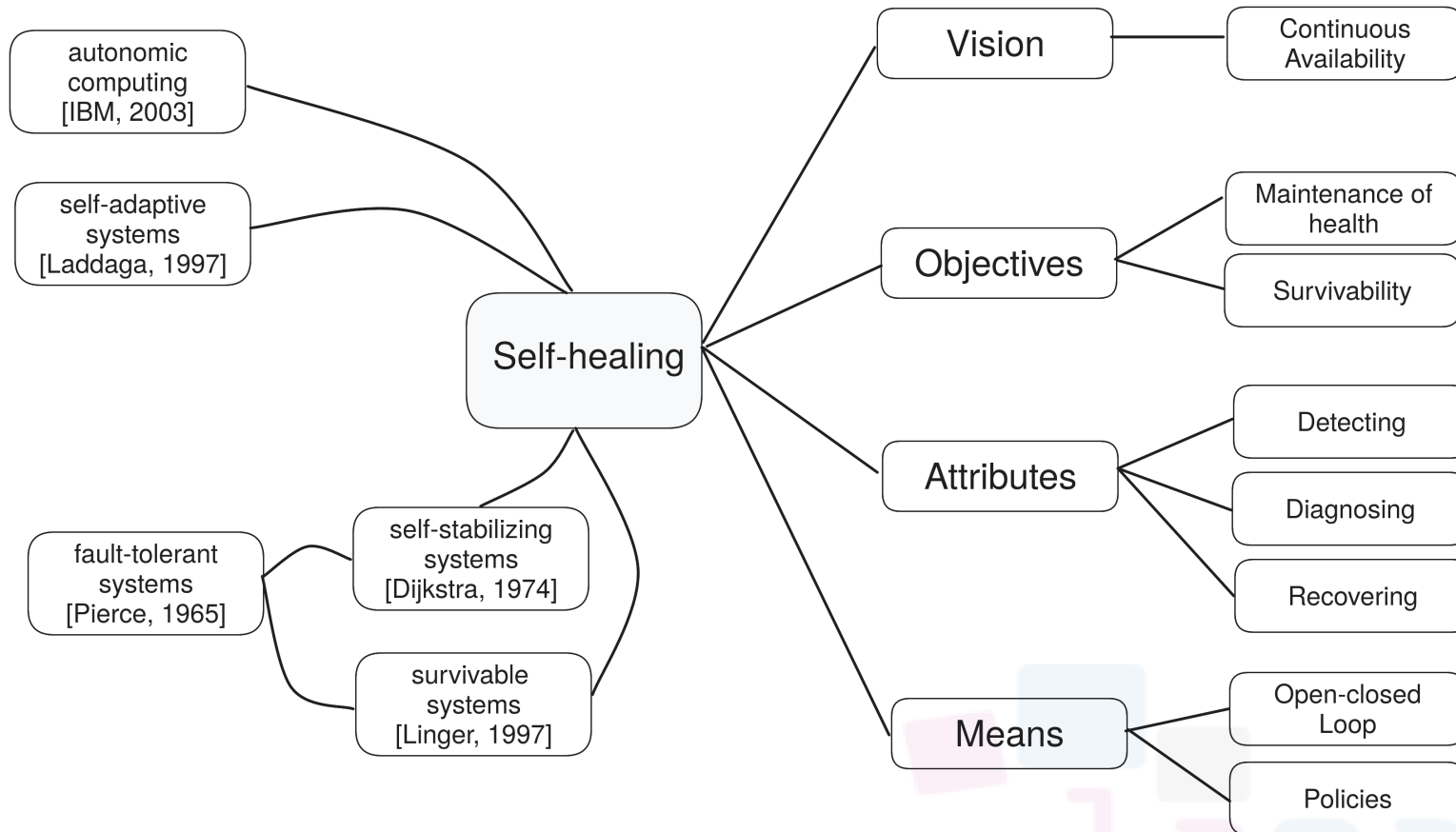


# Software Systems Resilience – Self-\* Systems

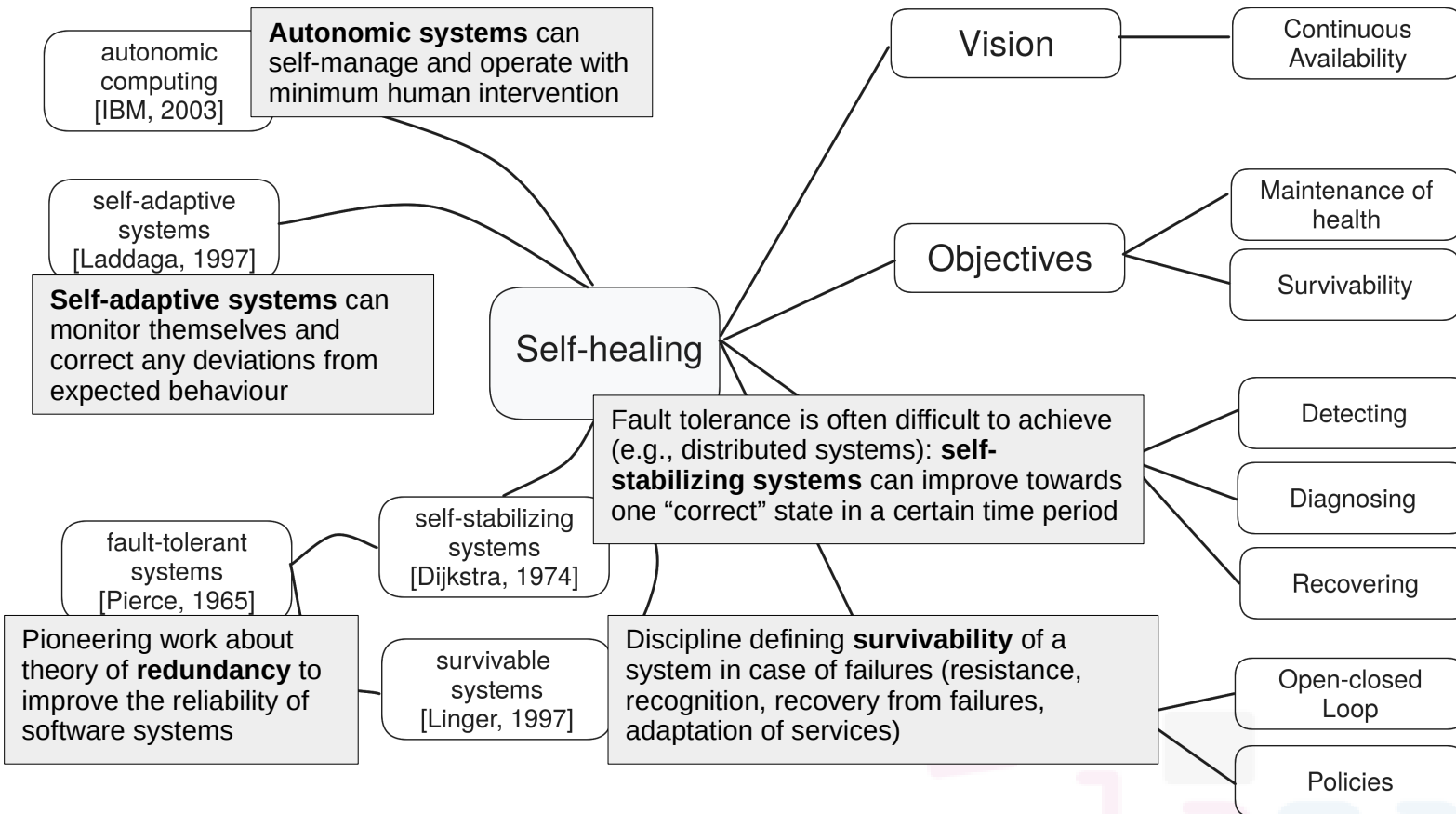
- Software Resilience is often associated with the following concepts (4S)

- \* **Self-Configuring:** The ability to readjust in real time without redeployment.
- \* **Self-Healing:** capacity to diagnose issues and take countermeasures returning to an operative state.
- \* **Self-Optimization:** optimization of the resources allocation based on the real time requirements.
- \* **Self-Protection:** capacity to anticipate, detect, identify, and defend from failures.

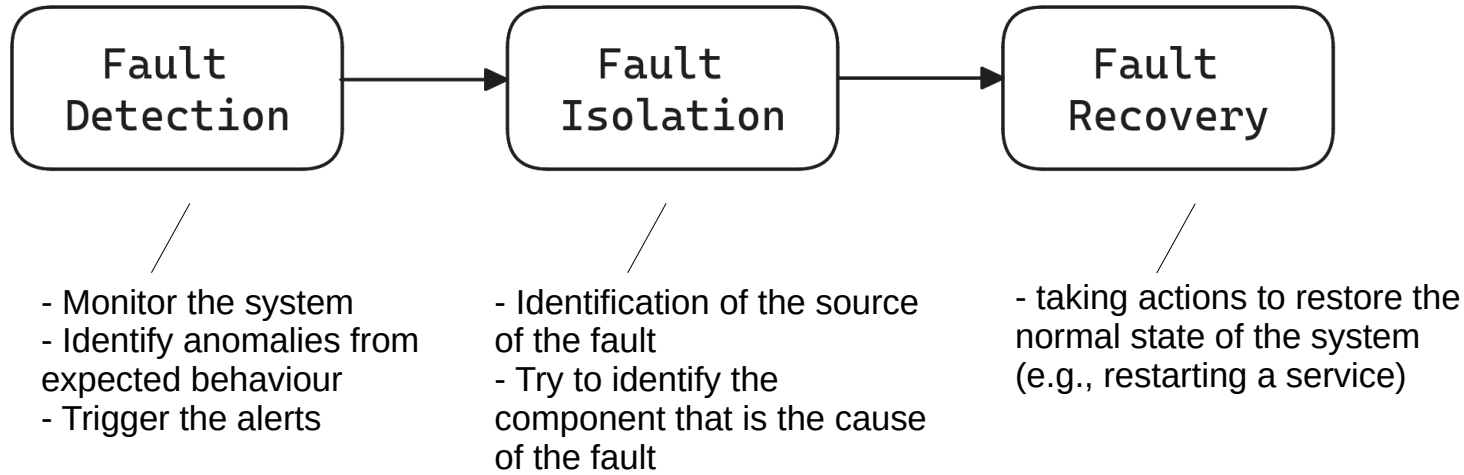
# Self Healing Research



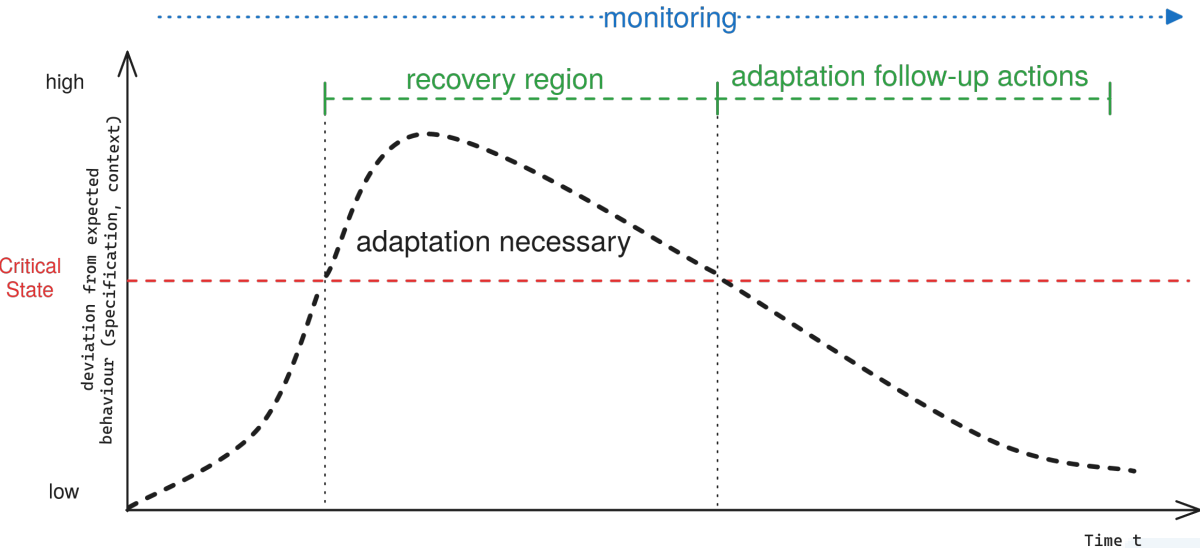
# Self Healing Research



# Typical Aspects of Self-Healing Systems



# Self-healing System Challenges

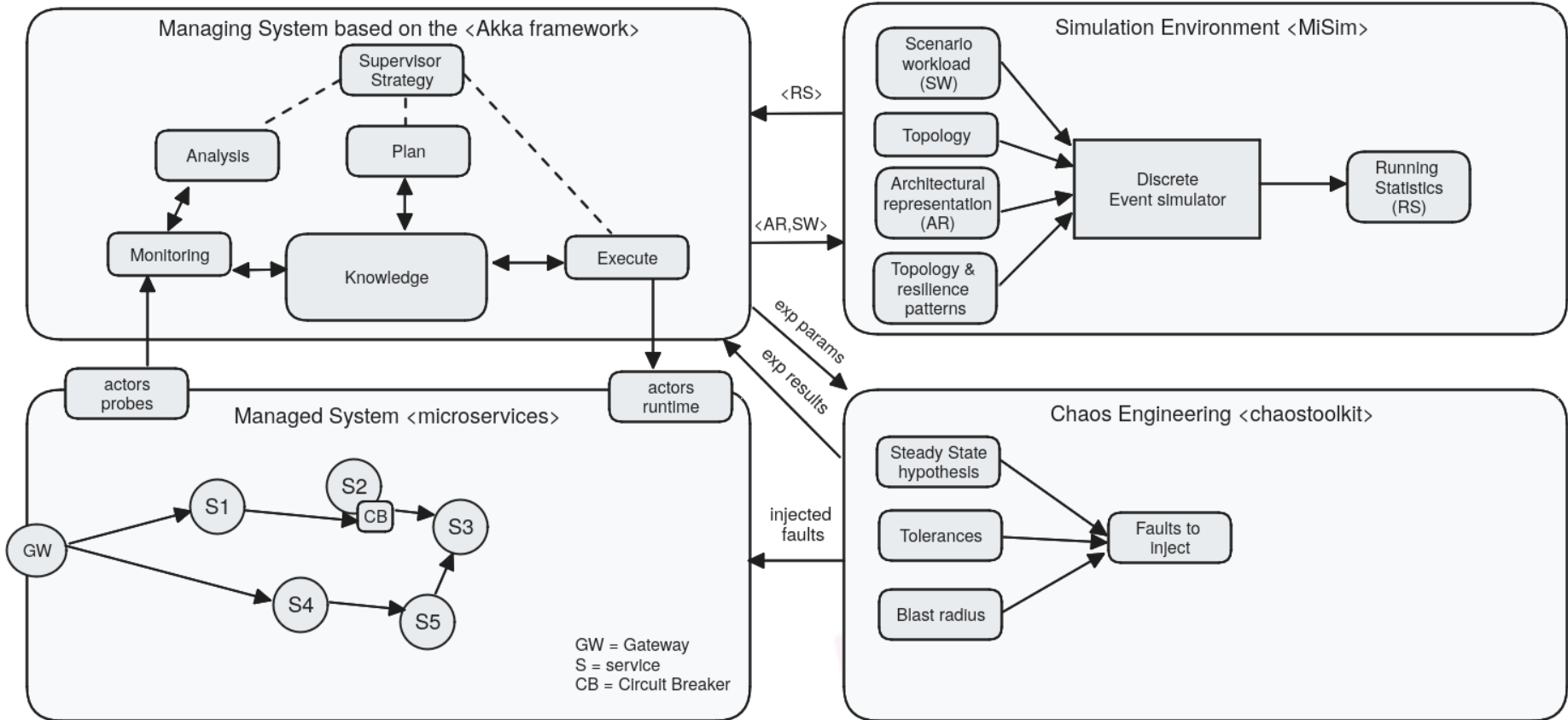


## Major Challenges:

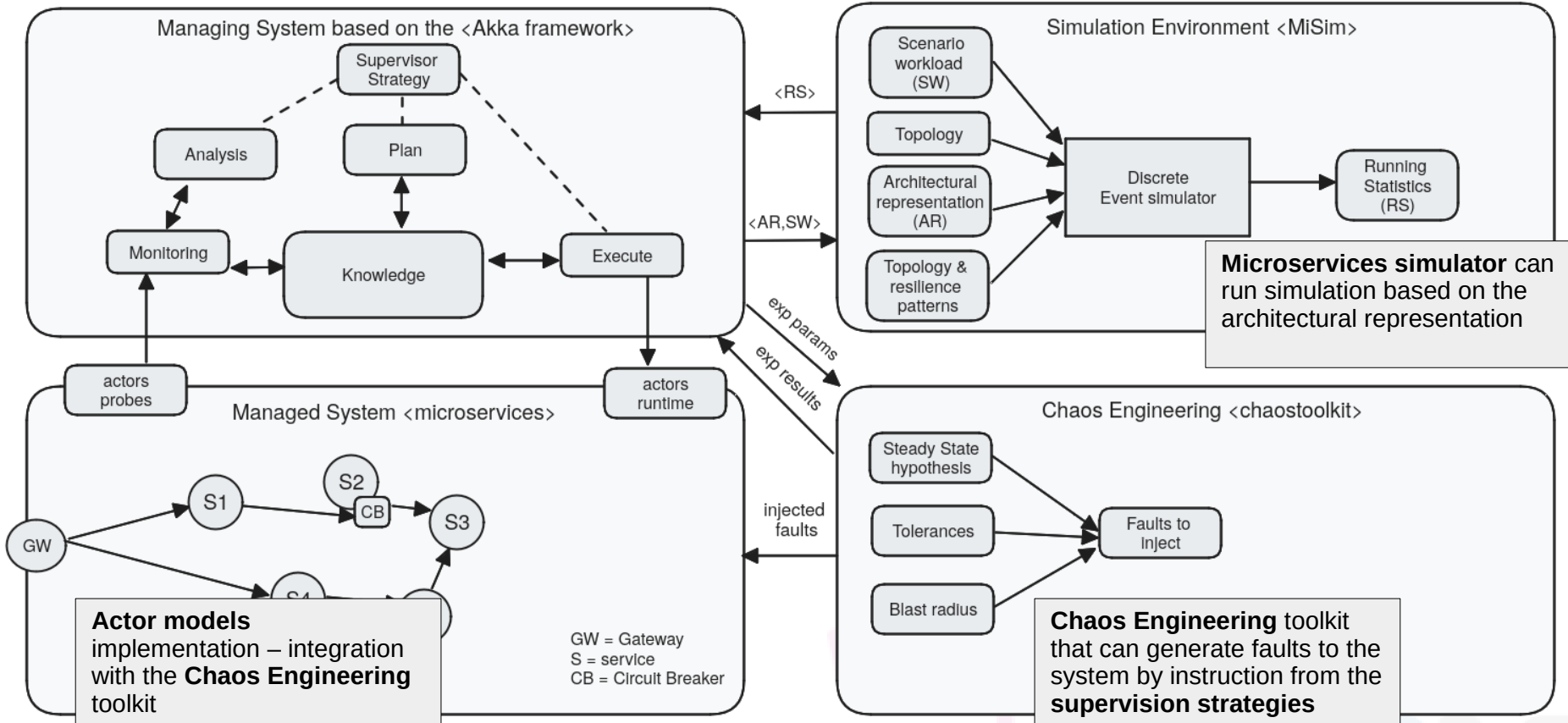
- How to define the **expected behaviour**? Both in the sense of **specifications** but also **anomalies**
- Defining **situational / context** awareness
- **Fault analysis**: when and which recovery actions to take? What is “enough” of a recovery action to restore the state?
- Can the system “**learn**” based on the actions performed?
- Are **predictive capabilities** needed? Taking preventive actions based on some signals
- How to deal with **uncertainty** of such systems
- **Openness of the self-healing system**: how open/close is the system in terms of adaptive actions



# Proposed Software Architecture to reach the "4S"

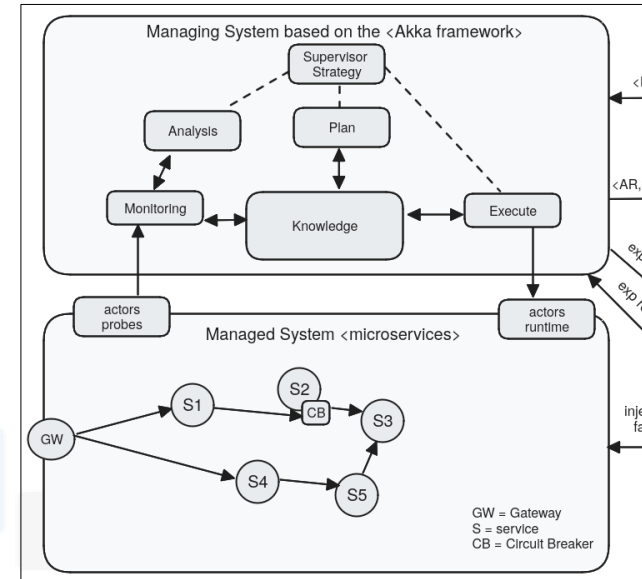
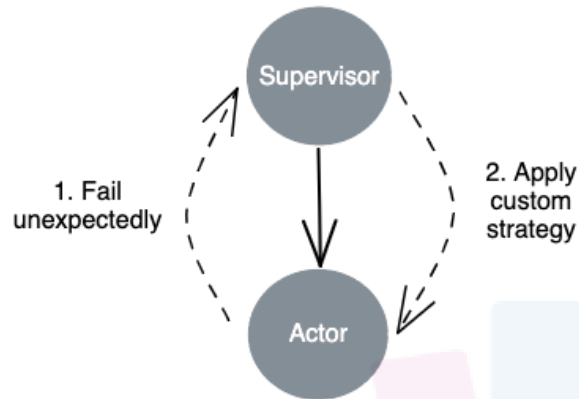
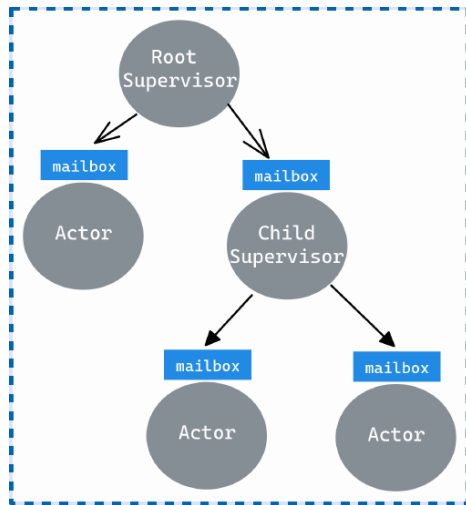


# Proposed Software Architecture to reach the "4S"



# Using the Actor Models to reach the “4S” (1/2)

- Our proposal is to use the **Actor Models**. The actor model is a mathematical model of concurrent computation with roots dating back to 1973. It was introduced by Hewitt et al. in 1973
- The system using an actor model consists of location-transparent actors, seen in the model as the universal primitives of concurrent computations. Each actor receives input and responds by
  - sending a **finite number of messages** to the other actors
  - creating a **finite number of child actors**
  - **modifying its internal state**



# Using the Actor Models to reach the “4S” (2/2)

- Implementation of the **Actor Model** with the **AKKA** framework
- Creation of a framework for the integration of the **supervision strategies**
- Integration in a **Spring Boot** microservice system
- Integration of **resilience patterns** like the **circuit breaker**

**Adopting the Actor Model for Antifragile Serverless Architectures**  
Marcel Mraz, Hind Bangui, Bruno Rossi, and Barbara Buhnova  
Faculty of Informatics, Masaryk University  
Brno, Czech Republic

**Antifragility**

**Main Concept**

- Antifragility is an improved resilience approach introduced in 2012 by Nassim Nicholas Taleb in his book entitled: "Antifragile: Things That Gain from Disorder". The idea behind Antifragility is to appreciate some level of stressors and perturbations and actively "employ" them to get better performance over a longer time horizon.

**The problem**

- Software systems are **inherently fragile** and cannot currently withstand high levels of stress without **becoming failures** - we are far from systems that can self-heal and improve.

**Our Vision**

- Seek to **inject volatility** in smart environments to **expose their fragilities**.
- Enable critical infrastructure systems to make **autonomous decisions** to move from stable to unstable conditions.
- Enable smart environments to **survive shocks** and dynamically determine **self-healing actions**.

**Service-Oriented Systems**

- Code is executed in **stateless containers** triggered by events and structured as **Functions as a Service (FaaS)**.
- In **FaaS**, each function can represent a small part of the application.
- **Differently from a microservice environment**, the functions have a **limited time span** when they are instantiated **on-demand**.


**Actor Models**

**Description**

- The actor model is a mathematical model of concurrent computation with roots dating back to 1973. It was introduced by Hewitt et al. (1973).

**Modeling**

- The system using an actor model consists of location-transparent actors, seen in the model as the universal primitive of concurrent computations. Each actor receives input and responds by:
  - 1) sending a finite number of messages to the other actors
  - 2) creating a finite number of child actors
  - 3) modifying its internal state



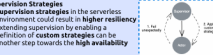
**Antifragile Solution**

**Supervision Strategies**

- **Supervision strategies** in the serverless environment could result in **higher resiliency**.
- Extending supervision by enabling a definition of **custom strategies** can be another step towards the **high availability**.

**Antifragile Strategy**

- **Stressor** selects an Actor to stress, on top of which it will try to generate errors.
- **Autonomous Learner**, as a machine learning component, is responsible for analyzing the generated errors outputting a list of system fragilities.
- **Antifragile builder**, as a component for analyzing the fragilities and building a list of antifragile improvements.
- **Supervisor** is the actor responsible for managing the lifecycle of its child actors throughout the implemented **supervision strategies**.
- **Actor** is the selected stressed child actor, which is managed by its supervisor.




**Antifragile Solution**

1. Generated errors
2. Detected errors
3. Applied improvements
4. Strategy change
5. System improvements

**Antifragile Solution**

1. We described the importance of actor models for antifragility of software systems - with the adoption of supervision trees.
2. We proposed a predictive strategy based on the concept of stressors, which actors or a hierarchy of actors can be selected for injecting failures. Other components (Autonomous Learner and Antifragile builder) generate a list of system's improvements. The supervisor component is responsible for managing the lifecycle of the child actors.
3. The solution adopting robust actor model based systems can improve the system's resiliency towards antifragility.

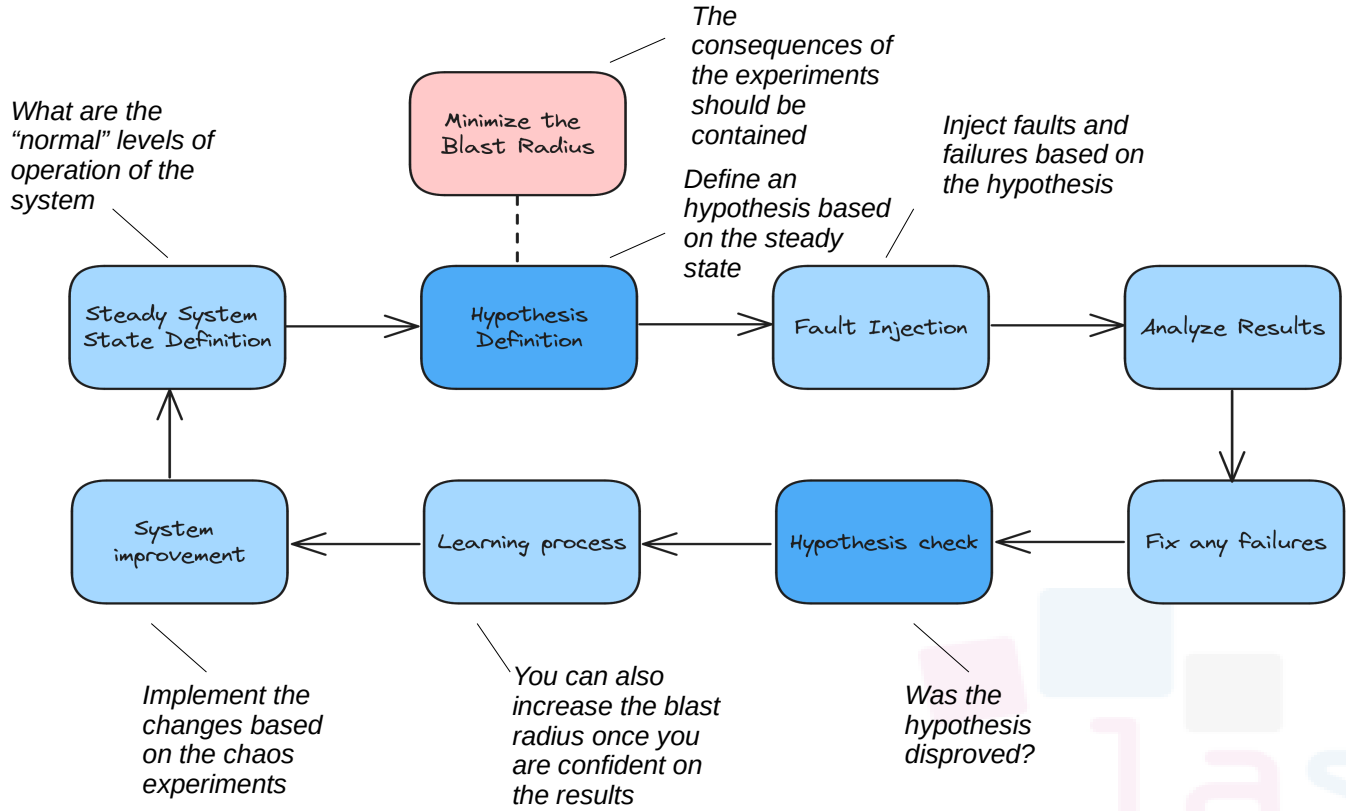
**Antifragile Solution**



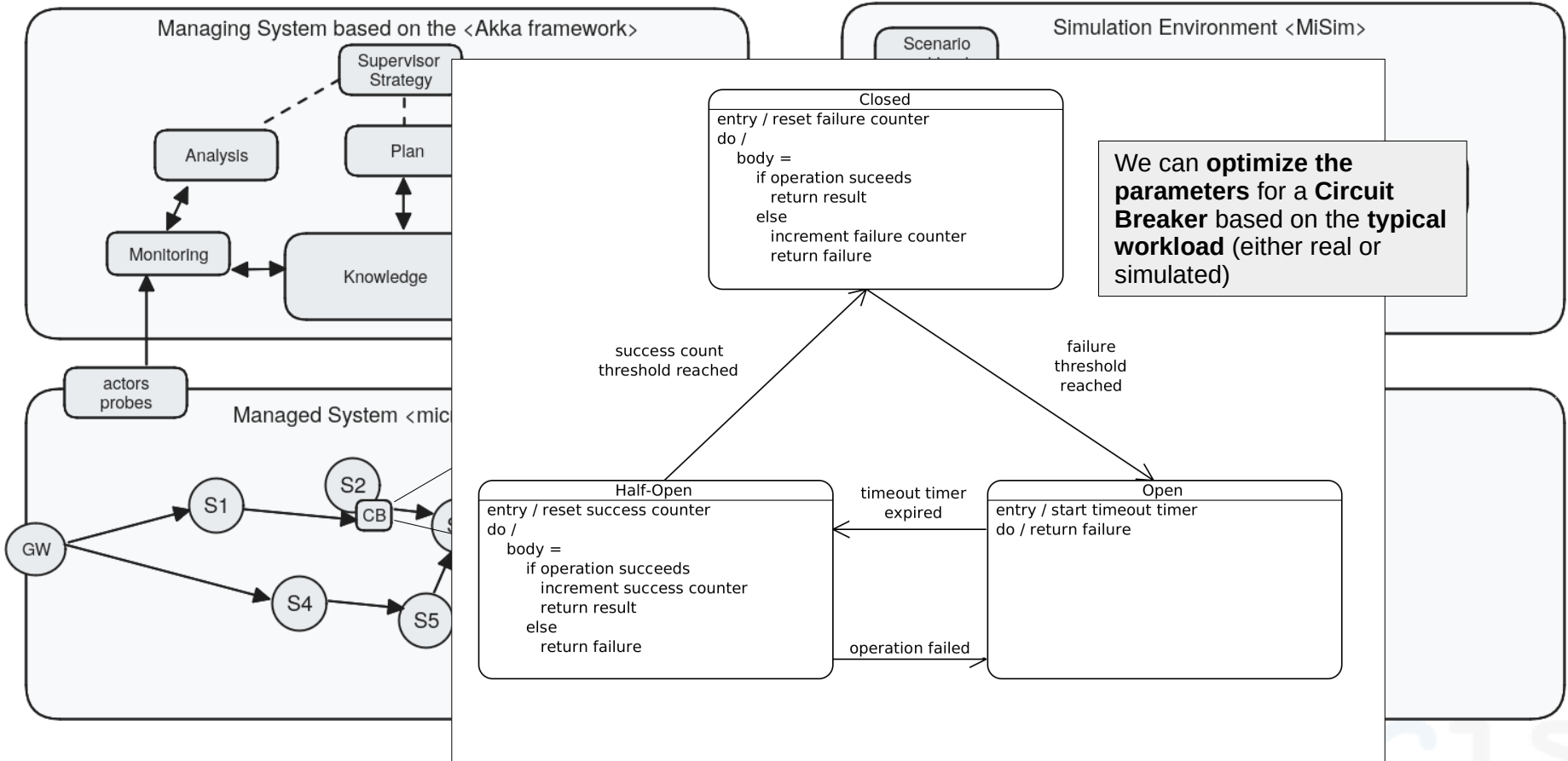
Mraz, M., Bangui, H., Rossi, B., & Buhnova, B. (2023). Adopting the Actor Model for Antifragile Serverless Architectures. Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)

# Using Chaos Engineering to reach the "4S"

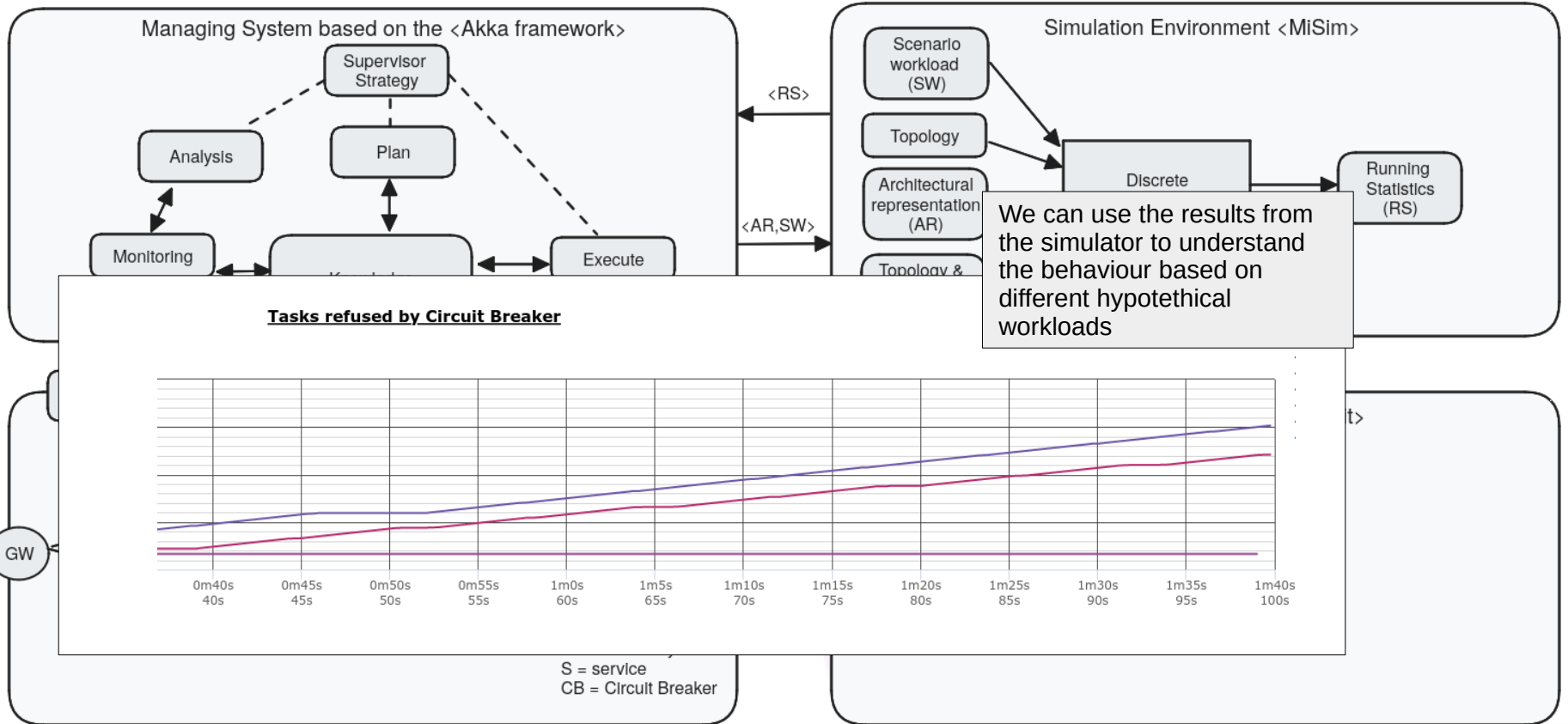
*"Chaos Engineering can help to understand how emergent behavior from component interactions could result in a system drifting into an unsafe, chaotic state"* From Miles, R. (2019). Learning Chaos engineering: discovering and overcoming system weaknesses through experimentation. O'Reilly Media.



# Integrating a Simulation Environment to reach the “4S”



# Integrating a Simulation Environment to reach the "4S"



# Main Challenges

- Modelling **expected behaviour** and how to **verify it**
- **Modelling unknown unknowns** and **uncertainty in the models**
- Which **anomaly detection algorithms** to integrate into the system
- Modelling **stress functions** of components
- Integration of **ML models** for all the **phases of Fault Detection, Isolation, Recovery**
- **Accuracy** of the simulator and capability of transferring the whole architectural representation
- **Automation of the design of chaos engineering** experiments and integration of ML models



# Main Takeaways

- **Software Resilience** means the capability of a software system to continue operations withstanding failures.
- **Self-\*** are class of software systems with properties of self-healing, self-configuration, self-optimization, self-protection.
- Our vision is that we cannot have **self-healing** systems without considering **self-adaptive** properties.
- One proposal is to adopt the **Actor Models** for the implementation of supervision strategies integrating with **Chaos Engineering** and a **microservices simulator**.

# Thank you a lot! Q&A

## Software Reliability Growth Modelling

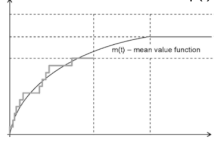
### SRGM

"If the history of fault detection and removal follows a particular recognizable pattern, it is possible to describe the mathematical form of the pattern"

### Types of models

### Fitting the cumulative failures over time

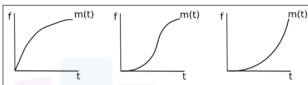
Mean value function –  $\mu(t)$



**Concave models** – assume the total number of faults in software is finite, and that it is possible to achieve fault-free software in finite time

**S-shaped models** – they also assume that the total number of faults is finite. Early testing is not as effective in fault discovery as the testing in the later stages. Therefore, there is a period in which the number of faults is increasing

**Infinite models** – assume that it is not possible to develop fault-free software because during fault removal we can introduce new ones

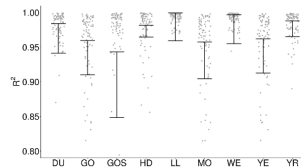


## RQ1 – Ranking of Models

To answer this RQ, we considered 792 SRGMs fitted on the whole dataset with 383 788 software defects.

Model	R <sup>2</sup>		AIC		RSE	
	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$
LL	0.979	0.094	3,374	1771.464	83.907	188.828
YR	0.977	0.052	3,806	1847.782	175.468	445.723
WE	0.976	0.096	3,411	1782.277	81.158	164.056
HD	0.973	0.039	3,769	1890.365	145.866	380.665
DU	0.963	0.099	3,708	1947.489	147.956	395.543
YE	0.937	0.115	3,889	2087.951	235.414	604.541
GO	0.935	0.115	3,896	2078.743	234.270	604.028
MO	0.931	0.123	3,883	2073.052	230.064	606.065
GOS	0.896	0.220	3,767	1942.052	180.096	438.825

The Kruskal-Wallis rank sum test for R<sup>2</sup> p-value < 0.05 (large effect size  $\eta^2[\eta^2] = 0.191$ ) indicates statistically significant differences between two or more groups



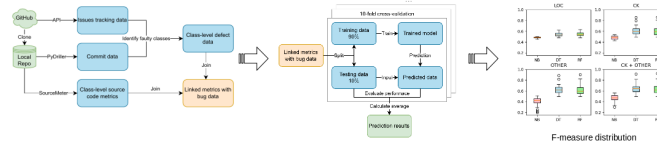
Infinite  
Concave models  
S-Shaped

### RQ1 Findings

Based on 792 fitted SRGMs, considering the R<sup>2</sup> metric LL, YR, WE, HD, DU are the best models. GO, GOS, MO, YE show the highest variance than other models. GOS is in general the worse model in terms of R<sup>2</sup>.

## Defect Prediction as proxies for failures

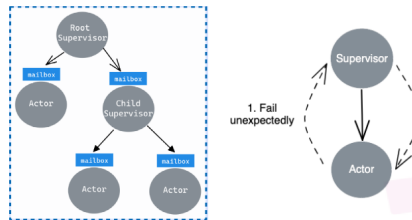
- We can have prediction models telling us about the prediction of defects in code



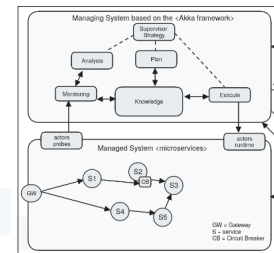
- It is assumed that the **more defects** → **the more the failures**
- look into code and improve to avoid future failures (for e.g., to see which modules require more attention)

## Using the Actor Models to reach the "4S" (1/2)

- Our proposal is to use the **Actor Models**. The actor model is a mathematical model of concurrent computation with roots dating back to 1973. It was introduced by Hewitt et al. in 1973
- The system using an actor model consists of location-transparent actors, seen in the model as the universal primitives of concurrent computations. Each actor receives input and responds by
  - sending a **finite number of messages** to the other actors
  - creating a **finite number of child actors**
  - modifying its internal state**



- Fail unexpectedly
- Apply custom strategy



## Adopting the Actor Model for Antifragile Serverless Architectures

Marcel Mráz, Hind Bangui, Bruno Rossi, and Barbara Buhnova  
Faculty of Informatics, Masaryk University  
Brno, Czech Republic

**Antifragility**  
Main Concept  
Antifragility is an improved resilience approach introduced in 2012 by Nassim Taleb. Unlike the basic resilience "strategize" (to get good despite) or "robustness" (to get ready to respond to stress), antifragile systems benefit from stressors and perturbations and actively "employ" them to get better performance over a longer time horizon.

**Proposed Solution**  
Supervisor Strategies in the operating system  
Supervisor strategies in the operating system  
Supervisor strategies in the operating system

**Actor Models**  
Description  
The actor model is a mathematical model of concurrent computation with roots dating back to 1973. It was introduced by Hewitt et al. in 1973.

**Antifragile Strategy**  
"Stressor" selects an Actor to stress, on top of which it will try to generate errors.  
"Autonomous Learner" as a machine learning component, is responsible for analyzing the generated errors and suggesting a list of system fragilities.

**Results**  
1) We demonstrated the importance of actor models for antifragility of software systems...  
2) We demonstrated the importance of actor models for antifragility of software systems...  
3) The solution adopting a robust actor model-based ecosystem can improve the system's resilience towards antifragility.

Thank you to the many colleagues and students that collaborated to the research: Radoslav Mičko, Dominik Arne Rebro, Stanislav Chren, Michael Schvarcbacher, K. Hrabovská, Barbora Buhnova, Tomas Pitner, Martin Macak, Marcel Mráz, Hind Bangui and many more