2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) | 979-8-3503-1192-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/SEAMS59076.2023.00033

CHESS: A Framework for Evaluation of Self-adaptive Systems based on Chaos Engineering

Sehrish Malik* Moeen Ali Naqvi* Leon Moonen*§

*Simula Research Laboratory, Oslo, Norway §BI Norwegian Business School, Oslo, Norway

Email: {sehrish, moeen}@simula.no, leon.moonen@computer.org

***Abstract*—There is an increasing need to assess the correct behavior of self-adaptive and self-healing systems due to their adoption in critical and highly dynamic environments. However, there is a lack of systematic evaluation methods for self-adaptive and self-healing systems. We proposed *CHESS*, a novel approach to address this gap by evaluating self-adaptive and self-healing systems through fault injection based on chaos engineering (CE).**

The artifact presented in this paper provides an extensive overview of the use of *CHESS* through two microservice-based case studies: a smart office case study and an existing demo application called *Yelb*. It comes with a managing system service, a self-monitoring service, as well as five fault injection scenarios covering infrastructure faults and functional faults. Each of these components can be easily extended or replaced to adopt the *CHESS* approach to a new case study, help explore its promises and limitations, and identify directions for future research.

***Index Terms*—self-healing, resilience, chaos engineering, evaluation, artifact.**

I. Introduction

Self-adaptive systems (SAS) and self-healing systems (SHS) are becoming increasingly important in fields such as the Internet of Things, Industry 4.0, and smart cities [1]. These systems are designed to operate in highly dynamic environments and are expected to handle uncertainty and unanticipated behavior, as well as provide fault tolerance and resilience. However, due to the complexity and dynamic nature of these systems, it is challenging to anticipate all possible scenarios that these systems will encounter. This is particularly important for the evaluation of these systems, which requires assessing the correct behavior of these systems.

There has been a growing concern among researchers about the lack of systematic evaluation for SAS and SHS [2–4]. A recent systematic mapping study found that only a small percentage of studies in this field focus on evaluating previously developed applications [3]. In addition, there are limited tools available to support evaluations based on runtime measures, with most studies focusing on evaluating the models used to design the system [4]. These models, however, may not take into account all potential scenarios that a system may encounter during operation. While runtime models offer a solution to these limitations, they also come with their own challenges, such as maintenance and the need for model creation [5]. Hence, there is a need for mechanisms to evaluate SAS and SHS based on runtime measures that consider potential scenarios a system may encounter during operation.

To fill this gap, our earlier work proposed *CHESS*, an approach for the systematic evaluation of self-adaptive and self-healing systems that build on chaos engineering principles [6]. *CHESS* systematically perturbates the system-under-evaluation and records how the system responds to those perturbations. The artifact[1] presented in this paper provides an extensive overview of the use of *CHESS* through two microservice-based case studies: a smart office case study and an existing demo application called *Yelb*. Concretely, the artifact consists of (i) predefined functional and infrastructural level fault injection scenarios, (ii) a self-monitoring service that presents extensive logs for the deployed services' normal and abnormal behaviors, (iii) the managing system service that reacts to the system's abnormal behavior traces and brings the system back to a stable condition, and (iv) a comparison of the service failure and cascading effects with and without deployment of the managing system service.

The remainder of this paper is organized as follows. In Section II, we summarize self-adaptive and self-healing systems for the microservice architecture, evaluation of these systems based on CE, and position *CHESS* in the landscape of self-adaptive system artifacts. Section III presents the design and architecture of *CHESS*, along with the microservice-based case studies and test scenarios considered for the fault injection. Section IV describes how to use the artifact with both of the case studies and presents the results of the fault injection scenarios. We conclude in Section V, including discussions of the artifact's applicability and directions for future work.

II. Background and Positioning of the Artifact

We briefly introduce the basics of SAS and SHS, and their evaluation in the context of microservices architecture, and highlight how *CHESS* complements the existing artifacts for engineering self-adaptive systems.

A. Self-Adaptive System for Microservices Architecture

Self-adaptive software systems are a class of software systems that can adapt to changes in their environment [7]. At a high level, these systems can be seen as comprising a *managed system* that is controlled by a *managing system*, generally realized through a MAPE-K feedback loop [8].

Self-adaptive systems for microservices architecture can bring multiple benefits, such as increased scalability, improved reliability, and reduced maintenance costs. Studies have suggested that there is an overall improvement in

[1] CHESS artifact on Zenodo: https://doi.org/10.5281/zenodo.6817763

DOI 10.1109/SEAMS59076.2023.00033

the management of microservices-based applications, allocation of microservices among the available servers, quality attributes such as performance, scalability, and resilience through the introduction of self-healing, self-management, and self-optimization properties [9]. Furthermore, an architecture-based self-adaptation framework with a MAPE-K feedback loop for a microservice as a *managed system* shows a reduced cost of ownership and faster self-adaptation [10]. On the other hand, the introduction of microservices architecture, as a *managing system*, can improve the self-adaptation capabilities of systems for various measures such as run-time data analysis [11]. Some challenges in developing a self-adaptive system based on microservices include developing monitoring and adaptation mechanisms for ensuring quality attributes; determining the level of distribution, observability, and granularity for deploying *control components*; and determining mechanisms for evaluation of the given quality attributes [12].

B. Evaluation of SAS and SHS based on Chaos Engineering

The evaluation of self-adaptive and self-healing systems is an important aspect of their design and deployment. In our previous work [6], we defined the evaluation of self-adaptive and self-healing systems as "*an approach to determine if a system meets objectives under operation, identify areas in which the system performs as well as desired or predicted, and provide evidence to the value and applicability of the system.*"

Several approaches to the evaluation of SAS and SHS include model-based evaluation [13], metric-based evaluation [14], model checking [15], and runtime testing and verification [16] each with its benefits and limitations. However, none of these evaluation approaches are viable for evaluation covering the execution of the system under real-life failure scenarios. Therefore, to address this gap, we introduce a mechanism that builds on *chaos engineering* principles. Chaos engineering (CE) is the practice of intentionally causing and studying controlled chaos within software systems operating in realistic environments, with the goal of increasing the systems' resilience and ability to handle unforeseen circumstances [17]. The core tenets of CE can be outlined as four main principles, which include formulating hypotheses based on the steady-state behavior of systems, introducing variations to real-world events, conducting experiments in a production environment, and automating these experiments for continuous execution. Our approach evaluates the systems through a systematic process that involves exposing the system to faults and testing its ability to recover from such perturbations.

C. Positioning of the Artifact

Artifacts play a vital role in advancing the field of self-adaptation. They serve as tangible examples of the algorithms and techniques developed by researchers, allowing for their evaluation and assessment. In addition, artifacts provide problematic scenarios and solutions that can inspire further research, as well as facilitate the comparison of results among different studies. The self-adaptive exemplars website[2] gives an overview of re-usable artifacts produced by researchers and engineers in the self-healing and self-adaptive community.

Various existing artifacts focus on web-based systems, microservices architecture, or cloud environments that assist in the evaluation of the *managed systems*. Znn.com [18] is a web-based information system that mimics real-world systems and provides an experimental environment to facilitate the evaluation. The exemplar applies a self-adaptive framework, *Rainbow* and presented an evaluation of the self-adaptive system based on a benchmark. Hogna [19] is a platform for deploying self-managing web applications on the cloud. It automates operations, monitors the health of the applications, extracts metrics, and analyzes performance data based on a model to create and execute an action plan. K8-Scalar [20] is an exemplar that allows the evaluation of different self-adaptive approaches to autoscaling container-orchestrated services. It is based on Docker, and Kubernetes, and extends a generic testbed for scalability evaluation of large-scale systems called Scalar. SEAByTE [21] enhances the automation of continuous A/B testing of a microservice-based system. Furthermore, exemplars such as SWIM [22], DARTSim [23], and RDMSim [24] consist of *managed systems* which can assist in the evaluation of external adaptation managers.

The present artifact represents a departure from existing artifacts in the field, as it prioritizes the *evaluation* of *managing systems* by inducing faults within the *managed system*. Thus, as opposed to primarily focusing on the *creation* of new self-adaptive approaches, it supports one of the critical tasks of software engineering research, i.e., the *systematic evaluation* of novel approaches. This aligns with the growing desire to produce artifacts in self-adaptation that support industry-relevant research [25].

III. CHESS

This section presents the CHESS architecture, the demo applications used for testing, and the scenarios for fault injection.

A. Design and Architecture

Figure 1 presents a detailed architecture for the CHESS approach. It consists of four main modules: containerized deployment cluster (CDC), system monitor, system manager, and fault injection. These modules are presented below.

Containerized Deployment Cluster (CDC): This artifact discusses the process of introducing faults into microservice-based applications deployed in a containerized environment using Kubernetes (K8s). The environment details are shown in Table I. The K8s cluster starts with a total of 20 GB memory and 4 CPUs to ensure that it has sufficient resources to run the demo applications. To make the demo applications accessible from outside the cluster, we have configured the K8s cluster with MetalLB, which is an addon that enables external IP services in K8s. In addition, the Istio service mesh is installed in the cluster to monitor the microservices traffic flow and to provide an easy way to manage the traffic between microservices. In order to visualize the data traffic and services' status, we have installed Grafana, Kiali, and

[2] http://self-adaptive.org/exemplars/

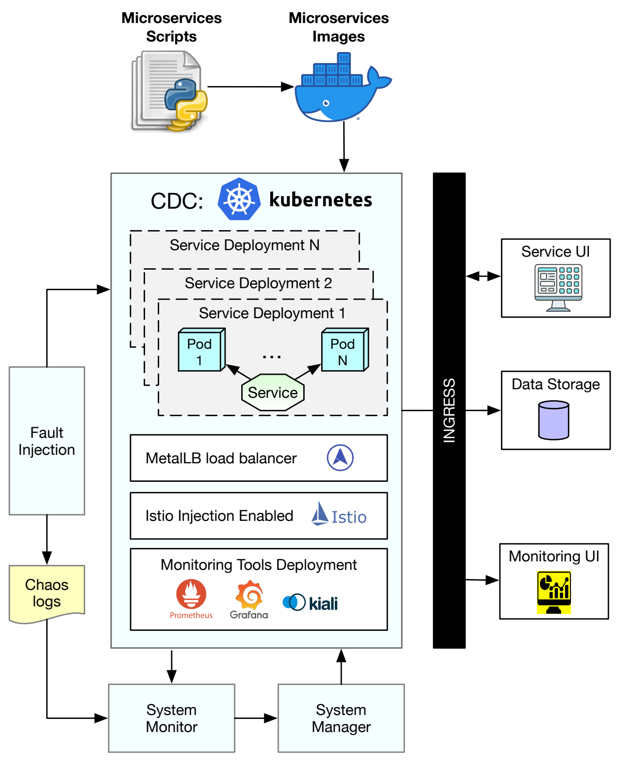


Fig. 1. Implementation Architecture for the applications deployment

Prometheus monitoring tools. Grafana is used to visualize the data traffic, while Kiali is used to visualize the services' status. Prometheus is used to query data metrics from its database and to store the data for later analysis. The combination of these tools provides a complete solution for monitoring microservices-based applications and for evaluating the fault tolerance of these applications.

System Monitor: The system monitor module monitors the status of the running services and performs necessary checks to ensure their validity. Its purpose is to observe the behavior of the system under various conditions, distinguish the system's normal behavior patterns from abnormal ones, and alert the system manager in case of any abnormal behavior.

System Manager: The system manager module plays a crucial role in ensuring the continuous operation of services deployed in the CDC. The primary responsibility of the system manager module is to receive abnormal system behavior alerts from the system monitor module and handle the recovery phase of the services that encounter faults. In this artifact, the system manager adopts a rule-based approach for fault recovery. The recovery process is automated and follows a set of predefined rules, which have been configured for the selected demo applications. The use of the system manager module also enables us to compare the impact of faults, known as the blast radius, for different fault injection scenarios, both with and without the system manager. This helps us to evaluate the effectiveness of the system manager in mitigating the effects of faults in the running services.

TABLE I
K8s Cluster Details

K8s cluster	Minikube
Cluster memory	20 GB
Cluster cpus	4
Load balancer	MetalLB
Service mesh	Istio
Monitoring DB	Prometheus
Monitoring tools	Grafana & Kiali
Chaos tools	chaostoolkit-Kubernetes

TABLE II
Chaos Experiment Template

steady-state-hypothesis:	
type	probe
provider type	python
provider module	chaosk8s.probes
func	{deployment_available_and_healthy}, {battery_charged}, {timely_response}
arguments	service_name namespace
tolerance	true
chaos-method:	
type	action
func	{inject_fault}, {deprecate_battery}, {inject_delay}, {terminate_pods}, {load_service}
arguments	service_name namespace
pause	{before, after} in seconds

Fault Injection: The fault injection module follows the principles of Chaos Engineering (CE) to induce faults in the deployed application services. A set of chaos experiments is carefully designed and scripted, for each fault injection scenario, in order to inject the desired faults into the services. A chaos experiment template is shown in Table II. A chaos experiment first defines the steady state with a probe function and a check against a tolerance. Once the steady state is met, the chaos action method is called to inject the respective fault, using the *service_name* and *namespace* arguments to identify the service. A pause before or after the fault injection can also be added. A virtual environment in Python is prepared with "chaostoolkit" and "chaostoolkit-Kubernetes" libraries to execute the chaos injection scripts. Each chaos injection generates chaos logs, which can be used for system observation purposes.

B. Demo Applications for Testing

The artifact uses the smart office case study from our earlier paper [6] and an open source example application named Yelb[3] as the demo applications for chaos injection and testing.

Smart Office Case Study: The smart office case study consists of nine services, including three input services (temperature sensor, motion sensor, and external weather), two control services (heating control, light control), two actuator services (heating actuator, light actuator), an MQTT broker, and a user interface. The control services retrieve periodic

[3] The Yelb application was reused from https://github.com/mreferre/yelb

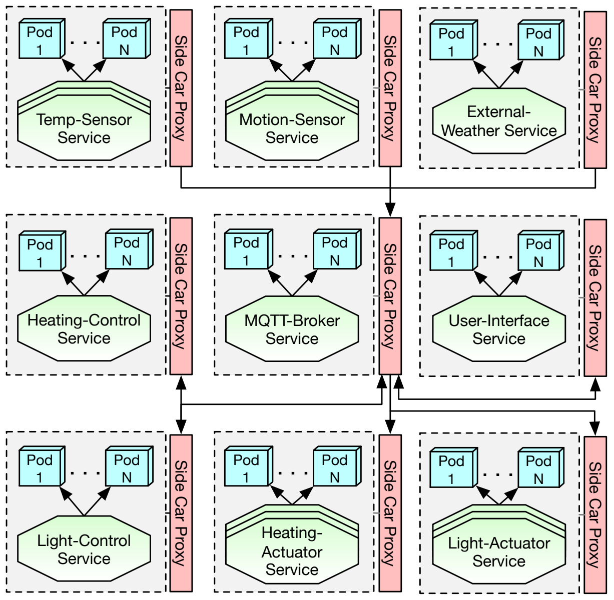


Fig. 2. Service graph for the Smart Office case study

sensing data and weather data, then use rules to control heating and lighting actuators. The service graph for the smart office case study is shown in Figure 2.

Yelb Application: The Yelb application consists of four services: a user interface service, an application server (appserver) service, a redis server service, and a database service. It allows users to vote for their preferred restaurant among given options, and updates a pie chart based on the number of votes received for each option. The Istio view of the service graph for the Yelb application is shown in Figure 3.

C. Test Scenarios for Fault Injection

Chaos Engineering allows the injection of two types of faults in a running application: infrastructure level and functional level. In the context of microservices-based applications deployed in a CDC cluster, the infrastructure level faults include the faults that occur due to issues with CDC configurations and resources. These faults can be mostly injected using predefined functions available within various chaos toolkits. The functional level faults are unique to the application and require some additional knowledge of the application's back-end logic and data flow. Chaos Engineering provides support for functional level faults, which can be injected by writing custom probes and action methods for a specific scenario and calling them in a chaos experiment script.

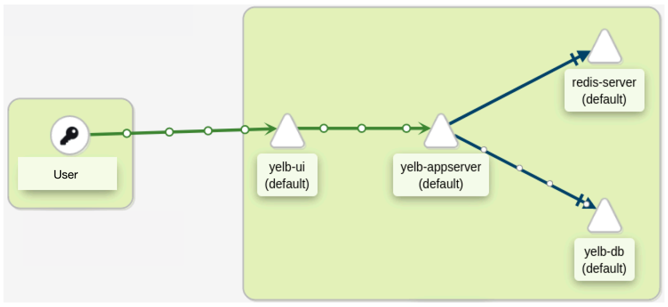


Fig. 3. Istio view of the service graph for the Yelb case study

TABLE III
TEST SCENARIOS FOR FAULT INJECTION

Fault Injection	Fault Level	Target Application
Sensor Down	Infrastructure & Functional	Smart-Office
Sensor Faulty	Infrastructure & Functional	Smart-Office
Service Delayed	Infrastructure	Smart-Office
Service Down	Infrastructure	Smart-Office
High SRR	Infrastructure	Yelb-App

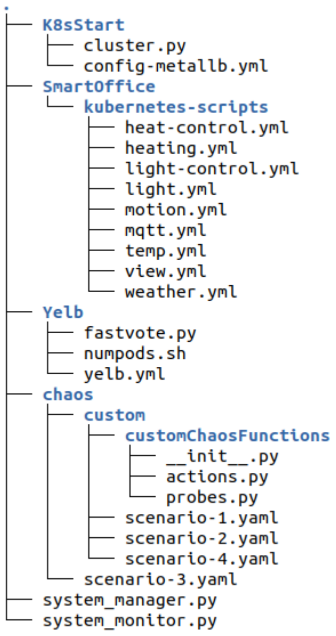
The artifact presents 5 fault injection scenarios, 4 using the smart office case study and 1 using the Yelb app.

FS-1: a deployed sensor is down unexpectedly.
FS-2: a deployed sensor sends erroneous readings.
FS-3: a running service is down abruptly.
FS-4: a running service is delayed.
FS-5: a running service is loaded with a high service request rate (SRR).

Table III presents an overview of faults injected with respect to the fault level and the target application. The infrastructure level fault injection is executed on the system service in a black-box manner. The infrastructure & functional level fault execution is performed in a grey-box manner where some additional access to the service's functional status is required.

IV. EXPERIMENTS

The artifact is available in a virtual machine. The directory structure for artifact implementation is shown in Figure 4.



```
.
├── K8sStart
│   ├── cluster.py
│   └── config-metallb.yml
├── SmartOffice
│   └── kubernetes-scripts
│       ├── heat-control.yml
│       ├── heating.yml
│       ├── light-control.yml
│       ├── light.yml
│       ├── motion.yml
│       ├── mqtt.yml
│       ├── temp.yml
│       ├── view.yml
│       └── weather.yml
├── Yelb
│   ├── fastvote.py
│   ├── numpods.sh
│   └── yelb.yml
├── chaos
│   ├── custom
│   │   ├── customChaosFunctions
│   │   │   ├── __init__.py
│   │   │   ├── actions.py
│   │   │   └── probes.py
│   │   ├── scenario-1.yaml
│   │   ├── scenario-2.yaml
│   │   └── scenario-4.yaml
│   └── scenario-3.yaml
├── system_manager.py
└── system_monitor.py
```

Fig. 4. Artifact's directories' hierarchy

TABLE IV
RUN SMART-OFFICE SCENARIOS

Step 1: Start K8s Cluster	`python3 cluster.py`
Step 2: Run Demo-App Scenario	`python3 system_monitor.py` *X Y*
Step 3: Run Chaos Virtual Env	`~/.venvs/chaosEnv/bin/activate`
Step 4: Run Chaos Script	`chaos run scenario-X.yaml`
Step 5: Kiali visualization [Optional]	`istioctl dashboard kiali`
Step 6: Grafana visualization [Optional]	`istioctl dashboard grafana`

TABLE V
RUN YELB-APP SCENARIO

Step 1: Start K8s Cluster	`python3 cluster.py`
Step 2: Run Demo-App Scenario	`python3 system_monitor.py` *X* `0`
Step 3: Run pods logger	`./numpods.sh`
Step 4: Start overloading yelb-app	`python3 fastvote.py` <url>
Step 5: Kiali visualization [Optional]	`istioctl dashboard kiali`
Step 6: Grafana visualization [Optional]	`istioctl dashboard grafana`

A. How to use the artifact: Smart-Office Scenarios

There are four main steps, along with two optional steps for online data visualization, to run the artifact for the smart-office scenarios (refer to Table IV).

The first step is to create or start the K8s cluster by running the script *cluster.py*. This will create a new minikube-based K8s cluster with 4 CPUs and 2000 MB of memory. The newly created cluster will be configured with metalLB load balancer and have istio mesh installed, with istio-injection enabled and monitoring tools installed. Next, you can deploy the demo application's services for a specified scenario by running the script *system_monitor.py X Y*. The value for X represents the scenario number and can be 1, 2, 3, or 4 to run FS-1, FS-2, FS-3, or FS-4, respectively. The value for Y is either 0 if you want to run the system monitor alone, or 1 if you want to run the system monitor with the system manager to recover the services from failures. To view the running deployments and services in the k8s cluster, run the command: *kubectl get all*. Step three involves activating the virtual chaos environment for chaos injection. Then, run the chaos command *chaos run scenario-X.yml* to inject a failure corresponding to the running scenario. Where, *scenario-X* represents the chaos experiment to be injected, and X can be 1, 2, 3, or 4 for FS-1, FS-2, FS-3, or FS-4, respectively. Finally, you can use online monitoring tools to examine the online monitoring data for service statuses and traffic flow. If desired, the Kiali dashboard and Grafana dashboard can be launched in separate terminals to observe the service data metrics.

B. How to use the artifact: Yelb-App Scenario

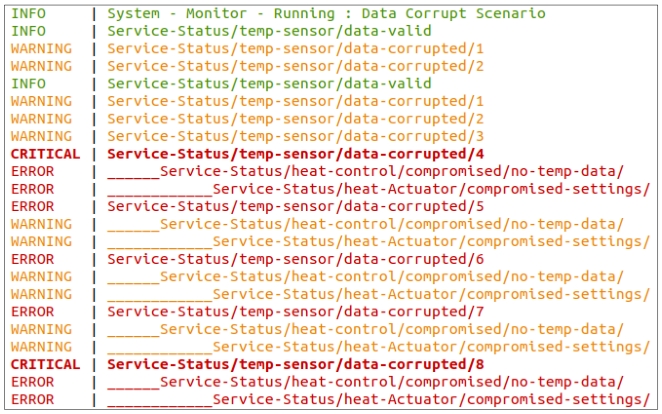
The execution process for the Yelb application scenario involves four main steps, with two additional optional steps for online data visualization, as described in Table V.

The first step is to create the K8s cluster, followed by the deployment of demo application services. The app-server service can be enabled for auto-scaling using the horizontal pod autoscaling (HPA) functionality of K8s by running the following command: *kubectl autoscale deployment yelb-appserver –cpu-percent=10 –min=1 –max=20*. The parameters *cpu-percent*, *min*, and *max* represent the dedicated CPU percentage, the minimum number of replicas to be running, and the maximum number of replicas that can run under heavy load, respectively. These values can be adjusted based on the available resources in the cluster. In the third step, the pods' logger is executed by running the shell script *numpods.sh*. This script records changes in the number of replicas over time and writes the results to a log file, which can later be analyzed to understand the relationship between the number of active users and the usage of CPU resources, and the increase or decrease in replicas. The fourth step is to run the scripted chaos for the fast voting onto the yelb-appserver. Finally, for real-time data observation, the Kiali dashboard and Grafana dashboard can be launched in separate terminals, as an optional step.

C. Results

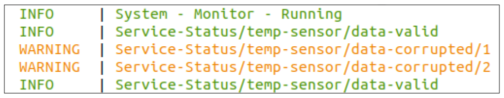
Smart-Office Scenarios: The first four scenarios (FS1, FS2, FS3, and FS4) are designed to evaluate the effect of chaos injection and failures on a running system, both with and without the deployment of a system management service. They provide a comparison of the system's behavior when it fails to recover from an injected fault and when it successfully recovers to evaluate the system's behavior and recovery capabilities. Figure 5 depicts the system monitor output for FS-1 when run without a management service (i.e., running *python3 system_monitor 1 0*). In this scenario, an injected fault in the form of data corruption leads to a cascading failure that the system is unable to recover. On the other hand, Figure 6 shows the system monitor output for FS-1 when run with a management service (i.e., running *python3 system_monitor 1 1*). In this case, the system manager is able to quickly identify and recover from the failed service, resulting in a much smoother and more efficient recovery process.

Yelb-App Scenario: The FS5 evaluates the scalability of services deployed in a Kubernetes cluster by imposing different loads on the service. The resulting log file captures two types



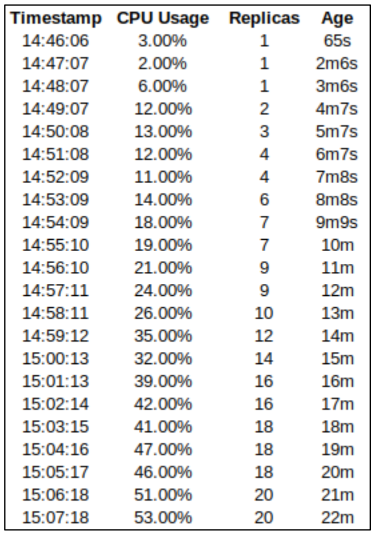
```
INFO      | System - Monitor - Running : Data Corrupt Scenario
INFO      | Service-Status/temp-sensor/data-valid
WARNING   | Service-Status/temp-sensor/data-corrupted/1
WARNING   | Service-Status/temp-sensor/data-corrupted/2
INFO      | Service-Status/temp-sensor/data-valid
WARNING   | Service-Status/temp-sensor/data-corrupted/1
WARNING   | Service-Status/temp-sensor/data-corrupted/2
WARNING   | Service-Status/temp-sensor/data-corrupted/3
CRITICAL  | Service-Status/temp-sensor/data-corrupted/4
ERROR     | _____Service-Status/heat-control/compromised/no-temp-data/
ERROR     | ___________Service-Status/heat-Actuator/compromised-settings/
ERROR     | Service-Status/temp-sensor/data-corrupted/5
WARNING   | _____Service-Status/heat-control/compromised/no-temp-data/
WARNING   | ___________Service-Status/heat-Actuator/compromised-settings/
ERROR     | Service-Status/temp-sensor/data-corrupted/6
WARNING   | _____Service-Status/heat-control/compromised/no-temp-data/
WARNING   | ___________Service-Status/heat-Actuator/compromised-settings/
ERROR     | Service-Status/temp-sensor/data-corrupted/7
WARNING   | _____Service-Status/heat-control/compromised/no-temp-data/
WARNING   | ___________Service-Status/heat-Actuator/compromised-settings/
CRITICAL  | Service-Status/temp-sensor/data-corrupted/8
ERROR     | _____Service-Status/heat-control/compromised/no-temp-data/
ERROR     | ___________Service-Status/heat-Actuator/compromised-settings/
```

Fig. 5. Running scenario 1 without system manager service



```
INFO      | System - Monitor - Running
INFO      | Service-Status/temp-sensor/data-valid
WARNING   | Service-Status/temp-sensor/data-corrupted/1
WARNING   | Service-Status/temp-sensor/data-corrupted/2
INFO      | Service-Status/temp-sensor/data-valid
```

Fig. 6. Running scenario 1 with system manager service



Timestamp	CPU Usage	Replicas	Age
14:46:06	3.00%	1	65s
14:47:07	2.00%	1	2m6s
14:48:07	6.00%	1	3m6s
14:49:07	12.00%	2	4m7s
14:50:08	13.00%	3	5m7s
14:51:08	12.00%	4	6m7s
14:52:09	11.00%	4	7m8s
14:53:09	14.00%	6	8m8s
14:54:09	18.00%	7	9m9s
14:55:10	19.00%	7	10m
14:56:10	21.00%	9	11m
14:57:11	24.00%	9	12m
14:58:11	26.00%	10	13m
14:59:12	35.00%	12	14m
15:00:13	32.00%	14	15m
15:01:13	39.00%	16	16m
15:02:14	42.00%	16	17m
15:03:15	41.00%	18	18m
15:04:16	47.00%	18	19m
15:05:17	46.00%	18	20m
15:06:18	51.00%	20	21m
15:07:18	53.00%	20	22m

Fig. 9. Data extracted from numpods logfile

of data: information about all running pods (obtained through *kubectl get pods*) and data on HPA deployment (obtained through *kubectl get hpa*). The log is updated every 30 seconds and records the name, status, number of restarts, and age of each running pod (as shown in Figure 7), and the name, CPU targets, minimum and maximum number of pods, number of replicas, and age of the HPA deployment (as shown in Figure 8). This log can be analyzed to uncover patterns in the service's performance and resource utilization over time. For example, Figure 9 presents sample data extracted from observing generated log files. The extracted data depicts a dramatic increase in CPU usage and the number of replicas over a 22-minute period.

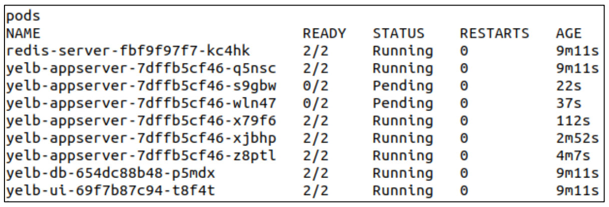
V. Concluding Remarks

Contributions: This paper presents an artifact that provides a detailed overview of how the *CHESS* approach can be used to evaluate a system's resilience and ability to recover from various types of faults. The implemented modules highlight the effectiveness of the system monitoring and system managing services in detecting and mitigating failures in the system. The artifact consists of (i) predefined functional and infrastructural level fault injection scenarios, (ii) a self-monitoring service that presents extensive logs for the deployed services' normal and abnormal behaviors, (iii) the managing system service that reacts to the system abnormal behavior traces and brings the system back to the stable condition, and (iv) a comparison of the service failure and cascading effects with and without deployment of the managing system service. The artifact is available on Zenodo,[1] and a demo video is on YouTube.[4]

Applicability: The artifact provides the SEAMS community with support for one of the critical tasks of software engineering research, i.e., the *systematic evaluation* of novel approaches. It aligns with the growing desire to produce self-adaptation artifacts that support industry-relevant research [25] by using chaos engineering to systematically observe containerized applications in Docker [26]. Artifact components are reusable, extendable, and modifiable for new case studies. Existing fault scenarios can be combined and expanded to create complex ones. The custom fault injection scripts can inspire exploration of grey-box level fault injection and testing.

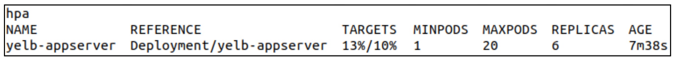
Future work: Directions of interest include exploring the use of observability data and system logs for chaos engineering-controlled data synthesis. In addition, techniques for the automated selection of regions for chaos experiments based on the health and performance status of services are also of interest. Finally, the inclusion of contextual information, such as ontologies or knowledge graphs, can also be considered to enhance fault injection targeting.

Acknowledgments: This research is supported by the Research Council of Norway through the cureIT project (#300461) and used resources provided by the Experimental Infrastructure for Exploration of Exascale Computing (eX3), supported by the Research Council of Norway (#270053).



```
pods
NAME                                 READY   STATUS    RESTARTS   AGE
redis-server-fbf9f97f7-kc4hk         2/2     Running   0          9m11s
yelb-appserver-7dffb5cf46-q5nsc      2/2     Running   0          9m11s
yelb-appserver-7dffb5cf46-s9gbw      0/2     Pending   0          22s
yelb-appserver-7dffb5cf46-wln47      0/2     Pending   0          37s
yelb-appserver-7dffb5cf46-x79f6      2/2     Running   0          112s
yelb-appserver-7dffb5cf46-xjbhp      2/2     Running   0          2m52s
yelb-appserver-7dffb5cf46-z8ptl      2/2     Running   0          4m7s
yelb-db-654dc88b48-p5mdx             2/2     Running   0          9m11s
yelb-ui-69f7b87c94-t8f4t             2/2     Running   0          9m11s
```

Fig. 7. Pods data view in logfile



```
hpa
NAME             REFERENCE                   TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
yelb-appserver   Deployment/yelb-appserver   13%/10%   1         20        6          7m38s
```

Fig. 8. HPA data view in logfile

[4] Artifact demo video: https://youtu.be/CBcaPJgpi-o

References

[1] T. Wong, M. Wagner, and C. Treude. “Self-adaptive systems: A systematic literature review across categories and domains.” In: *Information and Software Technology (IST)* 148 (Aug. 2022), p. 106934. ISSN: 0950-5849. DOI: 10/gp6krm.

[2] I. Gerostathopoulos, T. Vogel, D. Weyns, and P. Lago. “How do we Evaluate Self-adaptive Software Systems?: A Ten-Year Perspective of SEAMS.” In: *Int’l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, May 2021, pp. 59–70. ISBN: 978-1-66540-289-7. DOI: 10/gmvfd2.

[3] W. F. Passini, C. A. Lana, V. Pfeifer, and F. J. Affonso. “Design of frameworks for self-adaptive service-oriented applications: A systematic analysis.” In: *Software: Practice and Experience (SP&E)* 52.1 (2022), pp. 5–38. ISSN: 1097-024X. DOI: 10/gpd4x4.

[4] A. O. de Sousa, C. I. M. Bezerra, R. M. C. Andrade, and J. M. S. M. Filho. “Quality Evaluation of Self-Adaptive Systems: Challenges and Opportunities.” In: *Brazilian Symp. Software Engineering*. ACM, Sept. 2019, pp. 213–218. ISBN: 978-1-4503-7651-8. DOI: 10/gpd4xj.

[5] S. Ghahremani and H. Giese. “Evaluation of Self-Healing Systems: An Analysis of the State-of-the-Art and Required Improvements.” In: *Computers* 9.1 (Mar. 2020), p. 16. DOI: 10/gkgf26.

[6] M. A. Naqvi, S. Malik, M. Astekin, and L. Moonen. “On Evaluating Self-Adaptive and Self-Healing Systems using Chaos Engineering.” In: *IEEE Int’l Conf. Autonomic Computing and Self-Organizing Systems (ACSOS)*. Sept. 2022, pp. 1–10. DOI: 10/grdjz4.

[7] D. Weyns. *An introduction to self-adaptive systems: a contemporary software engineering perspective*. Wiley, 2021. ISBN: 978-1-119-57494-1.

[8] J. Kephart and D. Chess. “The vision of autonomic computing.” In: *Computer* 36.1 (Jan. 2003), pp. 41–50. ISSN: 0018-9162. DOI: 10/c3t3bc.

[9] M. Filho, E. Pimentel, W. Pereira, P. H. M. Maia, and M. I. Cortes. “Self-Adaptive Microservice-based Systems - Landscape and Research Opportunities.” In: *Int’l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, May 2021, pp. 167–178. ISBN: 978-1-66540-289-7. DOI: 10/gp6kr2.

[10] S. R. Boyapati and C. Szabo. “Self-adaptation in Microservice Architectures: A Case Study.” In: *Int’l Conf. Engineering of Complex Computer Systems (ICECCS)*. Mar. 2022, pp. 42–51. DOI: 10/gq35n3.

[11] A. Banijamali, P. Kuvaja, M. Oivo, and P. Jamshidi. “Kuksa*: Self-adaptive Microservices in Automotive Systems.” In: *Product-Focused Software Process Improvement*. Ed. by M. Morisio, M. Torchiano, and A. Jedlitschka. Springer International Publishing, 2020, pp. 367–384. ISBN: 978-3-030-64148-1. DOI: 10/gjpm6d.

[12] N. C. Mendonca, P. Jamshidi, D. Garlan, and C. Pahl. “Developing Self-Adaptive Microservice Systems: Challenges and Directions.” In: *IEEE Software* 38.2 (Mar. 2021), pp. 70–79. ISSN: 0740-7459, 1937-4194. DOI: 10/gp6ksx.

[13] D. M. Barbosa, R. G. d. M. Lima, P. H. M. Maia, and E. C. Junior. “Lotus@Runtime: A Tool for Runtime Monitoring and Verification of Self-adaptive Systems (Artifact).” In: *Dagstuhl Artifacts Series* 3.1 (2017), 7:1–7:5. ISSN: 2509-8195. DOI: 10/gk5ns9.

[14] J. Porter, D. A. Menascé, H. Gomaa, and E. Albassam. “TESS: Automated Performance Evaluation of Self-Healing and Self-Adaptive Distributed Software Systems.” In: *Int’l Conf. Performance Engineering*. ACM, Mar. 2018, pp. 40–47. ISBN: 978-1-4503-5095-2. DOI: 10/gkgf2n.

[15] J. Cámara and R. de Lemos. “Evaluation of resilience in self-adaptive systems using probabilistic model-checking.” In: *Int’l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. June 2012, pp. 53–62. DOI: 10/gp6kr9.

[16] T. M. King, A. A. Allen, Y. Wu, P. J. Clarke, and A. E. Ramirez. “A Comparative Case Study on the Engineering of Self-Testable Autonomic Software.” In: *Int’l Conf. and Ws. Engineering of Autonomic and Autonomous Systems (EASE)*. Apr. 2011, pp. 59–68. DOI: 10/c75gr5.

[17] A. Basiri, A. Blohowiak, L. Hochstein, and C. Rosenthal. “A Platform for Automating Chaos Experiments.” In: *Int’l Symp. Software Reliability Engineering Workshops (ISSREW)*. Oct. 2016, pp. 5–8. DOI: 10/gkf957.

[18] S.-W. Cheng, D. Garlan, and B. Schmerl. “Evaluating the effectiveness of the Rainbow self-adaptive system.” In: *ICSE Ws. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. May 2009, pp. 132–141. DOI: 10/dm9jmh.

[19] C. Barna, H. Ghanbari, M. Litoiu, and M. Shtern. “Hogna: A Platform for Self-Adaptive Applications in Cloud Environments.” In: *Int’l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. May 2015, pp. 83–87. DOI: 10/gk5ntg.

[20] W. Delnat, T. Heyman, W. Joosen, D. Preuveneers, A. Rafique, E. Truyen, and D. V. Landuyt. “K8-Scalar: a workbench to compare autoscalers for container-orchestrated services (Artifact).” In: *Dagstuhl Artifacts Series* 4.1 (2018), 2:1–2:6. ISSN: 2509-8195. DOI: 10/gk5ns7.

[21] F. Quin and D. Weyns. *SEAByTE: A Self-adaptive Microservice System Artifact for Automating A/B Testing*. Apr. 2022.

[22] G. A. Moreno, B. Schmerl, and D. Garlan. “SWIM: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Web Applications.” In: *Int’l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. May 2018, pp. 137–143.

[23] G. Moreno, C. Kinneer, A. Pandey, and D. Garlan. “DARTSim: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Smart Cyber-Physical Systems.” In: *Int’l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. May 2019, pp. 181–187. DOI: 10/gkgfxr.

[24] H. Samin, L. H. G. Paucar, N. Bencomo, C. M. C. Hurtado, and E. M. Fredericks. “RDMSim: An Exemplar for Evaluation and Comparison of Decision-Making Techniques for Self-Adaptation.” In: *Int’l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. May 2021, pp. 238–244. DOI: 10/grqbj6.

[25] D. Weyns et al. “Guidelines for Artifacts to Support Industry-Relevant Research on Self-Adaptation.” In: *ACM SIGSOFT Software Engineering Notes* 47.4 (Sept. 2022), pp. 18–24. ISSN: 0163-5948. DOI: 10/grnqx9.

[26] J. Simonsson, L. Zhang, B. Morin, B. Baudry, and M. Monperrus. “Observability and chaos engineering on system calls for containerized applications in Docker.” In: *Future Generation Computer Systems* 122 (Sept. 2021), pp. 117–129. ISSN: 0167-739X. DOI: 10/gjqvps.