

Spring 2023

## Automated Evaluation for Distributed System Assignments

Nimesh Nischal  
*San Jose State University*

Follow this and additional works at: [https://scholarworks.sjsu.edu/etd\\_projects](https://scholarworks.sjsu.edu/etd_projects)



Part of the [OS and Networks Commons](#), and the [Other Computer Sciences Commons](#)

---

### Recommended Citation

Nischal, Nimesh, "Automated Evaluation for Distributed System Assignments" (2023). *Master's Projects*. 1213.

DOI: <https://doi.org/10.31979/etd.gpew-2xbn>

[https://scholarworks.sjsu.edu/etd\\_projects/1213](https://scholarworks.sjsu.edu/etd_projects/1213)

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact [scholarworks@sjsu.edu](mailto:scholarworks@sjsu.edu).

Automated Evaluation for Distributed System Assignments

A Project Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Nimesh Nischal

May 2023

© 2023

Nimesh Nischal

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Automated Evaluation for Distributed System Assignments

by

Nimesh Nischal

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

Dr. Ben Reed      Department of Computer Science

Dr. Genya Ishigaki      Department of Computer Science

Dr. Navrati Saxena      Department of Computer Science

## ABSTRACT

A distributed system can exist in numerous states, including many erroneous permutations that could have been addressed in the code. As distributed systems such as cloud computing and microservices gain popularity, involving distributed computing assignments is becoming increasingly crucial in Computer Science and related fields. However, designing such systems poses various challenges, such as considering parallel executions, error-inducing edge cases, and interactions with external systems. Typically, distributed assignments require students to implement a system and run multiple instances of the same code to behave as distributed. However, such assignments do not encourage students to consider the potential edge cases that external systems may induce when interacting with their code. Assignments that execute a combination of student submissions as a single system promote high-quality design discussions before and during code writing and encourage students to consider how to handle faults generated by other systems. Testing such assignments is labor-intensive and involves repetitive tasks of setting up and destroying a virtual environment in which to test the system. In some cases, inducing a specific type of fault may require modifying the submitted source code, which is strongly discouraged. This research project explores the necessity, design, and implementation of Distributed CodEval, a tool that enables course instructors to define test cases for automating the evaluation of distributed system assignments.

## ACKNOWLEDGMENTS

I sincerely thank Dr. Ben Reed for his exceptional guidance, unwavering motivation, and unwavering support throughout this research undertaking. Working under his supervision has taught me the importance of creating straightforward yet effective tools to enhance user experience. His encouragement of critical thinking has been my significant motivation and focus throughout this project.

In addition, I would like to extend my sincere gratitude to Aditi Agrawal and Archit Jain, the creators of CodEval, whose work on the CodEval tool has inspired and influenced my project.

Furthermore, I would like to acknowledge the invaluable support of my defense committee members, Dr. Genya Ishigaki and Dr. Navrati Saxena, and all the faculty members in the Computer Science department, who have provided me with crucial guidance and encouragement throughout my degree program.

Lastly, I thank my family, friends, and colleagues for their unwavering support and motivation during my academic journey.

## TABLE OF CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>History and Background</b>	<b>3</b>
2.1	Distributed System Assignments Automation	3
2.2	Distributed System Testbeds	7
2.3	Test Scenarios Specification	11
2.4	CodEval	15
<b>3</b>	<b>Design Methodology</b>	<b>18</b>
3.1	Orchestration	18
3.2	Division of Tests	18
3.3	Heterogeneous Tests	20
3.4	Heterogeneous Test Submissions Pool	22
3.5	Specification File	23
3.5.1	Declaration Tags	24
3.5.2	Action Tags	26
3.5.3	Command Placeholders	28
3.6	Configuration Changes	29
<b>4</b>	<b>Using Distributed CodEval</b>	<b>31</b>
4.1	Adding Configuration	31
4.2	Preparing Checker Program and Supporting Files	31
4.3	Writing a Test Specification	34
4.4	Running Distributed CodEval	38

<b>5</b>	<b>Difficulties and Challenges</b>	40
5.1	No Support for Maven Projects on CodEval	40
5.2	Stale Docker Containers	40
5.3	Factorial Growth of Unique Student Submission Combinations	40
<b>6</b>	<b>Future Work and Improvements</b>	42
6.1	Optimize Test Duration for Complex Assignments	42
6.2	Optimizing Initial Heterogeneous Combinations	42
6.3	Support for Range in ICMD and ICMDT Commands	43
6.4	Configurable Heterogeneous Tests Timeout	43
6.5	No Orphan Docker Containers	43
6.6	Network Faults Between Containers	44
6.7	Clearing Data From MongoDB	44
6.8	Configurable Wait Time for ECMDT ASYNC and ICMDT ASYNC Tag Commands	44
6.9	Parallel Execution of ICMD SYNC and ICMDT SYNC Tag Commands	45
<b>7</b>	<b>Conclusion</b>	46
	<b>REFERENCES</b>	47



## LIST OF TABLES

1	CodEval specification tags . . . . .	17
2	Declaration tags used in Distributed CodEval specifications . . .	24
3	Action tags used in Distributed CodEval specifications . . . . .	27

## LIST OF FIGURES

1	Architecture of a remote distributed systems lab using Docker containers . . . . .	4
2	A conceptual network of Docker containers running student program on the left and the respective actual connectivity using a router node on the right . . . . .	6
3	A breakdown of 198 random catastrophic failures analyzed by Yuan et al. [1] in Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis . . . . .	8
4	An example functional test in NESSEE checking two APIs of the Software Under Test (SUT) . . . . .	10
5	Sample specification of a Docker network . . . . .	12
6	Sample specification of dispatch actions to be sent to each listed Docker container along with a delay in between . . . . .	12
7	Example of TDL modules, such as NetworkCapabilities, Topology, and a Server, used by a NESSEE server . . . . .	14
8	Distributed CodEval architecture . . . . .	19
9	A sample heterogeneous test submission pool document . . . . .	22
10	Sample Distributed CodEval configuration . . . . .	31
11	A sample checker program in Java using PICOCLI library . . . . .	33
12	Execution instructions provided by Java's PICOCLI library . . . . .	34
13	Contents of a sample helper zip file . . . . .	34
14	Contents of a sample Distributed CodEval specification file . . . . .	35
15	Location of a Distributed CodEval specification file in a Canvas course . . . . .	36

16	Graph showing a growth rate of possible heterogeneous test combinations of 4 submissions versus the number of submissions on Canvas . . . . .	41
----	---	----

## 1. INTRODUCTION

A distributed system refers to a group of physical or logical machines that work together to solve a computational problem as a single system [2]. Such systems provide high reliability, scalability, availability, and cost-effectiveness [2]. Parallel and Distributed Computing (PDC) are necessary topics for students in Computer Science or a related field. M. Paprzycki [3], M. Gowanlock, and B. Gallet [4] emphasize the importance of including such courses early in the university curriculum. E. Saule [5] states the benefits of teaching parallel executions through assignments. He compares teaching through assignments with the classical way of introducing the theories and challenges of parallel and distributed computing. An important mindset while developing distributed systems is to expect failures and faults as an integral part of the system. Necessary checks must be present in the program to handle such faults.

There are multiple tools to auto-evaluate assignments, some of which are described and compared in [6, 7]. Most of them are used to provide immediate feedback to the student submissions. Some of these tools support testing programming assignments, but none can test the complexities of distributed systems, like concurrency, consistency, availability, and so forth [2, 8]. An auto-evaluator built specifically for distributed algorithms [9] has the ability to orchestrate the execution and termination of instances running such algorithms to generate fault scenarios.

Students may make assumptions about particular system behaviors that can differ from other students' assumptions. If so, even if the system works using a single implementation, it may fail with different implementations. To emphasize the importance of design discussions before and during writing the code, assignment submissions must be evaluated both as a homogeneous system, running multiple instances of a single submission, and a heterogeneous system of multiple submissions. These heterogeneous systems can be translated into industrial software systems running

different versions of the same program. Typically distributed program auto-graders, such as [9], and testing tools, as mentioned in [10, 11, 12], do not support deployment and orchestration of multiple versions of the same program.

A new tool was needed to assist instructors in testing, auto-evaluation, and providing feedback for the distributed assignments by executing a combination of student submissions as one system and introducing faults in a deterministic manner. This research project aimed to develop an auto-evaluation tool for distributed assignments that support interactions between a combination of submissions. Distributed CodEval expands the functionalities of CodEval [13], a code evaluation tool, by executing student submissions in isolation from other submission executions using Docker containers. It reads a configuration of test cases in a format that supports orchestrating executions of student submissions as a single distributed system, controlling interactions between the submissions, and introducing possible faults in the system. Distributed CodEval provides feedback to student submissions based on the test cases.

## 2. HISTORY AND BACKGROUND

A distributed system is an architecture where components run on different machines and communicate with each other over a network. This architecture is preferred over monolithic systems when scalability and high availability are prioritized above lower latency. Such systems are more prone to faults caused by the developers' negligence as it's not intuitive to consider all the possible cases where the system can enter an error state. The difficulty in covering edge cases increases more when the system enters an inconsistent state without raising any red flags. Building instincts in students through related assignments is necessary to create a habit of thinking about such cases.

As discussed in this chapter, several tools are available to automate different aspects of assignments. Some of them are used for distributed assignments as well. This project, Distributed CodEval, gets inspiration from such tools and distributed testbeds to provide a user-friendly interface by expanding the features provided by CodEval.

### 2.1 Distributed System Assignments Automation

GitHub Classroom [14] provides an online interface for assignments where students get a description of the task. The instructor can also upload a default template which the student can clone in their local machine and start working. Templates help the students as they do not have to start from scratch and help the instructor by creating a similar architecture of the submissions, which can be evaluated using the same tools. The instructors can also provide feedback as Pull Requests. GitHub Classroom provides autograding [15] through input/output tests and command tests. An input/output test provides a standard input to a test command and evaluates the output against an expected result. A command test runs a test command and checks the exit status of the command. Any exit code apart from 0 results in failure.

Command tests can be used by instructors to introduce scripts [16] that can run the submissions and check particular scenarios. GitHub Classroom uses GitHub Actions to run command tests which have an allocated limit of 3000 minutes per month, which can be extended to 50k minutes per month by a free enrollment process. This limitation can be a roadblock if the number of submissions and time needed to test each submission increases.

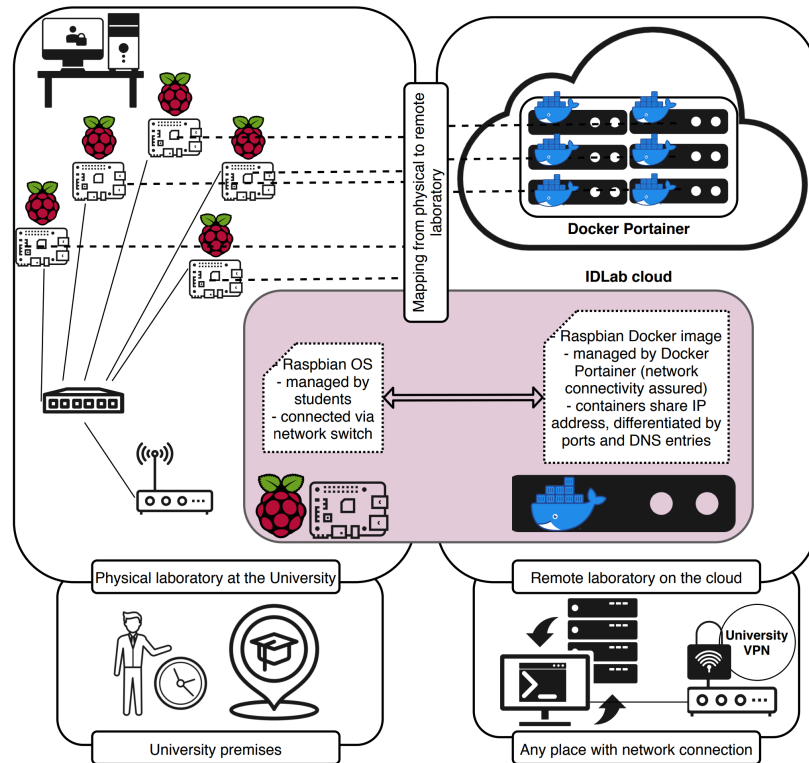


Figure 1: Architecture of a remote distributed systems lab using Docker containers  
 Source: [17]

CodeGrade [18] provides an instant feedback mechanism for students submitting their code. It can be integrated with a Learning Management System (LMS), a GitHub account, or a GitLab account to fetch submissions and run tests for feedback. Instructors can create an AutoTest [19] and start it on their online platform, which will run on previously submitted submissions (if any) and run on each new submission.

An AutoTest can include Unit Tests, Program Tests that execute the submission and check the return code or exit status, Capture Point Tests that run custom testing scripts and utilizes the expected floating point number output between 0 and 1 for grading, Code Quality Tests which run a static code analysis tool, such as linter, using a config file or a custom program that create comments on the submission, and Checkpoints that specify a minimum percentage of points needed to continue with subsequent test categories [20]. CodeGrade cannot be used to test distributed systems assignments because an AutoTest has the context of a single submission and cannot execute a submission in a combination of other submissions. The categorization of different types of tests, along with Checkpoints, were taken as an inspiration while designing Distributed CodEval.

Kriještorac, Resende, and Marquez-Barja [17] introduce their method of conducting remote experiments for distributed systems lab work. They use Docker containers [21] on the cloud and assign each container to a student to work on remotely, providing Laboratory as a Service (LaaS) [22]. The architecture of this approach is shown in Figure 1. The Docker containers replace the Raspberry Pi devices in the physical laboratory. It includes a survey of 45 students to demonstrate the effectiveness of such platforms. Providing a cloud laboratory to the students, which was accessible 24/7, proved to be a favorable practice. Inspired by such an architecture, Distributed CodEval uses Docker containers to execute student submissions in isolation and provide immediate feedback and part of logs in case of failure. This is beneficial for the students to check the execution logs and modify their implementation for faster testing.

Vivar and Magna [23] present the architecture and benefits of using a remote network lab to teach computer networks. The platform consists of a server using Java Communications API [24] and a web application using Java Applet on the



client side. The students can use the client-side application to interact remotely with commercial network devices like switches, routers, and firewalls. The paper mentions the availability of the lab to the students increased without the need for instructor supervision, which is necessary for a physical lab. The paper evaluates the effectiveness of the remote lab with a group of students and notes a great similarity between face-to-face and remote lab assignments' results. This architecture is an inspiration to build an automated feedback system for distributed system assignments so that users can get better access to testing infrastructure and prompt feedback with minimal manual intervention by the course instructor.

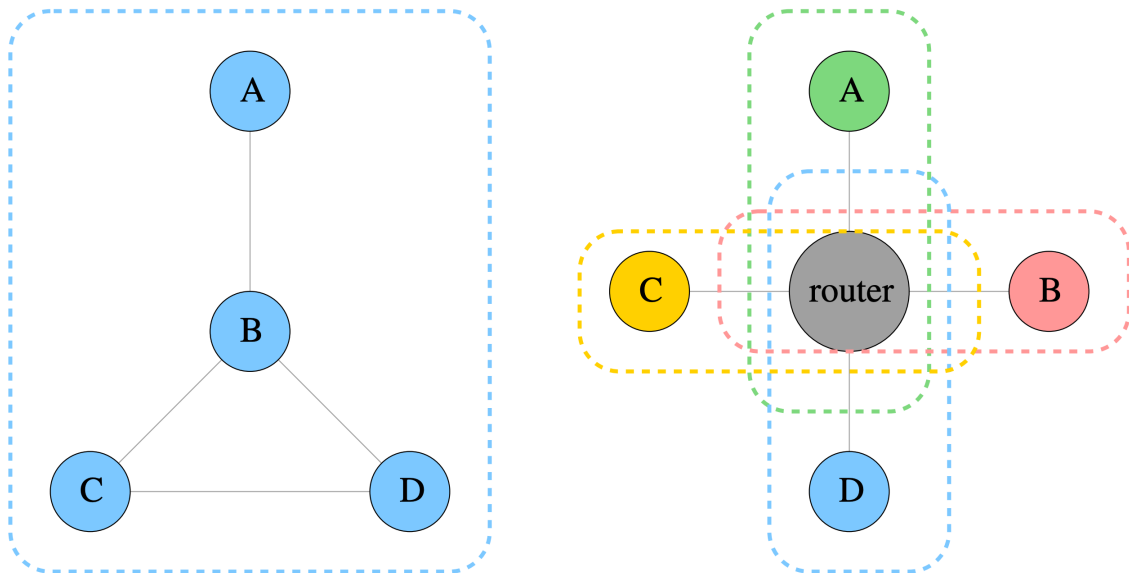


Figure 2: A conceptual network of Docker containers running student program on the left and the respective actual connectivity using a router node on the right  
Source: [9]

Maicus et al. [9] present a platform to auto-grade distributed system assignments using Docker containers. The authors emphasize the importance of fast and automated feedback to student submissions. The tool provides a way to control network traffic by delaying, dropping, and reordering filters. It does so with the help of a node

invisible to the user submissions, called a *router*. The router node intercepts and logs all messages in the network and can delay, drop, and reorder messages. The router can be instructed to do so through rules defined for each host-port pair in the network. The author remarks that although such an approach is more intrusive than other solutions [25, 26], it provides greater control over the ability to generate deterministic network faults and other benefits, such as allowing students access to a detailed message log. Figure 2 presents the structure of a conceptual network containing Docker containers running a student submission on the left. The figure’s right side contains the connectivity of the nodes with the router node. The router node is responsible for routing the respective data to all the other nodes and, thus, is able to control the flow of messages. The router is implemented in Python and can be customized according to the test case. A test case can also run without a network router node. The nodes get the knowledge of other host-ports in the network by parsing the *knownhosts.txt* file, which is generated by a network configuration discussed in Section 2.3.

## 2.2 Distributed System Testbeds

Distributed system failures are common, and handling every fault is difficult, as faults can occur in numerous ways. Current large-scale systems like Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis still fail periodically and sometimes catastrophically [1]. Yuan et al. [1] discuss the severity of not properly handling exceptions and analyze 198 random unique failures extracted from issue-tracking databases of some of the popular distributed systems. They provide statistics of failures, such as 88% of failures occur where a specific order of events is necessary, 98% occur in just three nodes, 74% are deterministic that can be reproduced given the right input events, and 92% arise due to incorrect handling of

non-fatal errors mentioned explicitly in the software. Figure 3 presents a break-down of 198 catastrophic failures analyzed by Yuan et al. [1] in Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis. These statistics reinforce the need for Distributed CodeVal to help the students get familiar with the complexities of implementing a distributed system.

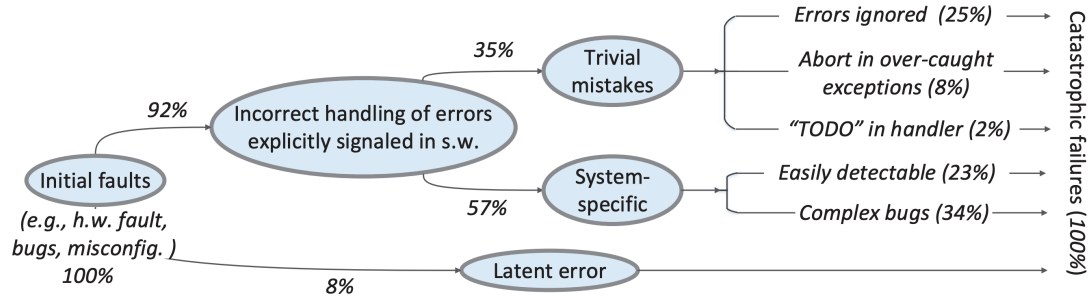


Figure 3: A breakdown of 198 random catastrophic failures analyzed by Yuan et al. [1] in Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis  
Source: [1]

Gunawi et al. [27] introduce a framework to test the recoverability of cloud services: Failure Testing Service (FATE) and Declarative Testing Specifications (DESTINI). FATE is used to push cloud systems into specific failure states. DESTINI provides specifications of the system’s expected behavior under test in a faulty state. The specifications are written in a relational logic language called Datalog [28]. The introduction of faults requires the availability of the system’s source code and leverages interposition technology like AspectJ [29] to force the system into specific states. Similarly, Joshi et al. [30] present SETSUDO, a perturbation-based testing framework, which leverages AspectJ to intercept execution points of interest. RemoteTest [10] uses a Java Reflection [31] and Aspect Oriented Programming [32] to provide a distributed testbed. Using AspectJ and similar frameworks to introduce faults did not fit the

requirements of this project as it required understanding individual source codes of student submissions and implementing aspects that might be needed during testing. The goal of Distributed CodEval is built to be highly decoupled from the source code of user submissions.

NESSEE [33] introduces three types of testing supported by the platform:

1. Functional testing that tests implementations individually by providing inputs and comparing outputs.
2. Scalability testing in which the system's expected behavior is scripted under varying loads. The expected behavior is specified using XML-based Test Description Language (TDL), explored briefly in Section 2.3
3. Network testing where testers can manually configure the behavior using NESSEE's web front end or automatically using test scripts.

Figure 4 gives an example of a functional test in the NESSEE platform. First, the Software Under Test (SUT) functions are identified. Then the test creator creates a test script and a helper object that calls the corresponding SUT functions. The test script contains the expected output that is compared to the actual output during the test execution. The test fails if the outputs do not match.

Distributed CodEval gets inspiration from this structure and uses checker functions in an executable to test the correctness of the system's state, as explained in Section 4.2. It is a good idea to populate a specification file, Section 4.3, with tests in increasing order of complexity, similar to NESSEE, so that the students get incremental feedback on their implementation.

Jepsen-io [34] provides a tool to test large-scale distributed systems as a Clojure library. A Jepsen test is written as a Clojure program. In a test, a distributed

system is initialized, a number of operations are executed against the system, and then the history of the operations is verified. Jepsen has been used to analyze several databases, coordination services, and queues [35]. The test runs on a control node that coordinates all the other nodes of the system being tested. The control node uses SSH [36] to log into a set of nodes and set up the distributed system to be tested. After the system has been initialized, Jepsen uses multiple single-threaded processes to execute test operations specified in the test program. A history of the start and end of these operations is maintained by Jepsen to analyze for correctness and to generate reports, graphs, etc.

Distributed CodEval uses a similar idea of a controller container, Section 3.1, to initiate external services with which the student submissions interact.

Jepsen uses a special nemesis process [37] to generate faults in the system. The nemesis is not bound to any node and acts as a special client. Jepsen uses *iptables* [38] and *qdisc* [39] to create partitions between nodes and filter packets [40].

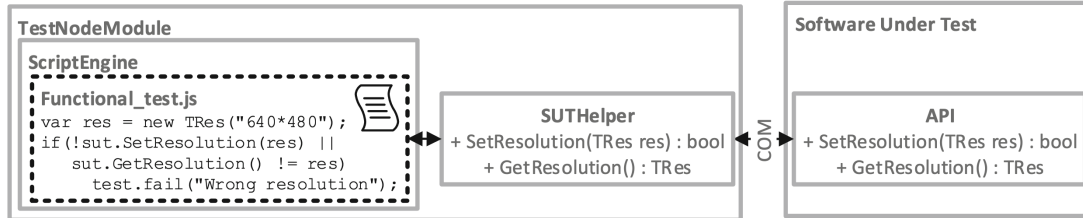


Figure 4: An example functional test in NESSEE checking two APIs of the Software Under Test (SUT)  
Source: [33]

ZooKeeper [41], a distributed and reliable coordination service, has been benchmarked and tested using various tools [42], such as Jepsen and Byteman. Byteman [43] is used to test the trace, monitor, and test the behavior of applications written in Java. It does so by injecting Java code into the application methods or runtime

methods during JVM startup or even when the application is running. Using Byteman does not require recompiling, repackaging, or redeploying the application.

### 2.3 Test Scenarios Specification

Lima [44] talks about the problem of test case explosion and the lack of integration testing support for distributed systems. The paper describes the use of UML Sequence Diagrams (SDs) [45] to create scenario-based models. These models describe the interactions in key scenarios between the users and components of a distributed system. The tester describes the participants and the behavior of the system with UML SDs and maps the information between the model and implementation. The SDs are automatically translated to a formal notation for generating test input efficiently and checking conformance at runtime. The approach runs test cases without a global clock, with and without local time constraints. The paper describes using VDM++ formal specification language [46], supported by the Overture tool [47], for checking distributed conformance in the absence of time constraints.

Yu and Patil [48] describe using workflow management [49] UI to create a collection of tests or a test session. The paper focuses on automating tests to save time. Users can launch tests using a workflow-based test session and monitor their status on the fly. All the test data is uploaded to a centralized database in real time. Tests can be specified to run in a parallel manner to save time. The paper describes a generic Workflow Manager and Workflow Agent implemented using .Net Remoting Objects running [50] as Windows services [51].

Maicus et al. [9] present a way to define network configurations using JSON. The reason behind this decision was to provide instructors with a simple network configuration that should be short and human-readable. Figure 5 demonstrates a specification of a Docker network. Similar configuration JSON objects are defined for

```

1  {
2  "use_router" : true,
3  "containers" : [
4    {
5      "container_name" : "container0",
6      "container_image" : "python:3.6",
7      "outgoing_connections" : ["container1"],
8      "commands" : ["python3 server.py"]
9    },
10   {
11     "container_name" : "container1",
12     "container_image" : "python:3.6",
13     "outgoing_connections" : ["container0"],
14     "commands" : ["python3 client.py"]
15   }
16 ]
17 }

```

Figure 5: Sample specification of a Docker network  
Source: [9]

```

1  {
2  "dispatcher_actions" :
3  [
4    {
5      "action" : "stdin",
6      "containers" : ["container0"],
7      "string" : "This will be piped to container0\n"
8    },
9    {
10     "action" : "delay",
11     "seconds" : 2
12   },
13   {
14     "action" : "stdin",
15     "containers" : ["container1"],
16     "string" : "This is piped to container1 2 sec later\n"
17   }
18 ]
19 }

```

Figure 6: Sample specification of dispatch actions to be sent to each listed Docker container along with a delay in between  
Source: [9]

each test case and presented as a JSON array to the system. Each test case creates new docker containers, networks, and output files. A network configuration consists

of 5 parameters:

- **use\_router** states if a router node, described in Section 2.1, needs to be inserted in the network.
- **container\_name** provides a name to the container. The container name is used when defining the network connections and as a directory name to store the student output.
- **container\_image** describes the name of the Docker image to be used.
- **outgoing\_connections** describes one-way connection channels between Docker containers in the network. By default, the connection channel is open to all the other containers. The values of `outgoing_connections` across all containers are combined to generate "knownhosts.txt", which is used as described in Section 2.1.
- **commands** provide the commands that need to be executed sequentially in the container.

The authors also present a configuration to dispatch actions to the containers in a network. Figure 6 displays a sample actions list. The actions are executed by the instructor node running in the network, discussed in Section 2.1. Using the current implementation of the framework, an instructor can provide the input strings to the standard input of Docker containers. The actions are performed sequentially without any subsequent delay by default. The "action" key in the action object can have one of the four possible values:

- **delay**: Used to introduce a delay between 2 actions in the list.
- **stdin**: Used to broadcast a string message to all one or more containers listed under "containers". The message is broadcast in parallel to all the containers within that action.
- **start**: Starts a specific container.



- **stop**: Stops a running container.

```

<NetworkCapabilities id="DSL">
  <delay direction="both" variation="7"
    distribution="laplace">12</delay>
  <loss direction="up">0.1</loss>
  <loss direction="down">0.09</loss>
  <reordering direction="both">0.3</reordering>
  <duplication direction="both">0.001</duplication>
  <disconnect start="24h" duration="5s" />
</NetworkCapabilities>
<NetworkCapabilities id="DSL6000" BasedOn="DSL">
  <bandwidth direction="down">6000</bandwidth>
  <bandwidth direction="up">1000</bandwidth>
</NetworkCapabilities>
<!-- ... -->
<ClientTopology>
  <NetworkNode id="DSLRouter" NetworkCapabilitiesId="DSL6000">
    <NetworkEndpoint id="UserPC" />
    <NetworkNode NetworkCapabilitiesId="WiFiRouter">
      <NetworkEndpoint id="UserNotebook"
        NetworkCapabilitiesId="WiFiEndpoint"/>
    </NetworkNode>
  </NetworkNode>
</ClientTopology>
<ServerComponents>
  <Datacenter id="datacenter_europe">
    <ISP id="isp01" NetworkCapabilitiesId="ISPCaps01"/>
    <ISP id="isp02" NetworkCapabilitiesId="ISPCaps02"/>
    <ServerCluster id="cluster01" defaultISPId="isp01"/>
  </Datacenter>
</ServerComponents>

```

Figure 7: Example of TDL modules, such as NetworkCapabilities, Topology, and a Server, used by a NESSEE server

Source: [52]

NESSEE [52] uses Test Description Language, an XML-based language for describing test cases to be executed by a NESSEE server. TDL provides a modular design where the modules are independent of each other. A few of the components supported by TDL are:

- **NetworkTopology module**: Used to define different sets of network param-

ters and network topologies.

- **Behavior module:** Used to define actions that can be arranged in TDL flows.
- **TestCaseDescription:** Used to link other modules and represent all the elements as a single test case.

A part of a sample TDL specification is shown in Figure 7, containing the capability definition of networks. A client topology and a server component are also defined using the defined network capabilities.

Distributed CodEval takes inspiration from the above specification formats and presents simple and easy-to-use tags in specification files, Section 3.5 that do not require familiarity with any particular programming language or testing tool.

## 2.4 CodEval

CodEval [13] is an automation tool to test student submissions on Canvas [53]. It is executed by providing the name of a course and several optional flags as command line arguments. CodEval searches for the course on Canvas and checks for assignments that have a related CodEval specification file uploaded on Canvas. For each such assignment, CodEval downloads the specification file and other prerequisite files if specified in the specification file. Then it checks for student submissions that have not been evaluated. CodEval then downloads each of such submissions and executes each test in the specification within a docker container, supplying specified text inputs or inputs from files to the student's program. It then verifies the specified expected exit code and output.

At the end of testing each submission, CodEval adds a comment to the student submission. If the tests pass successfully, the comment contains a success message with the number of test cases passed. If a test fails, the comment contains the failed test case number, the executed test command, a part of the failure logs, or a related

message to help the student debug the issue. Table 1 lists the tags supported in a CodEval specification file.

CodEval specification files are designed to be easy to write and read and flexible enough to support simple as well as complex programming assignments. Distributed CodEval expands the functionalities of CodEval to test distributed system assignments. It also takes inspiration from the tags supported by CodEval to introduce new tags pertaining to the complexities of distributed system assignments. The new tags are listed and explained in Section 3.5.

CodEval is designed to be executed for a particular course in regular intervals, such as 10 minutes or every hour, depending on the complexity of testing the assignments and how quickly the students expect feedback on their submissions. CodEval runs tests on a single docker container executing a student's submission by comparing the output of a program. This functionality is adequate to evaluate standalone programming assignments but cannot test the complexities of distributed assignments. Distributed assignments need to execute independently of each other, coordinated by an external factor, and their correctness is verified by checking the system's state. Coordination is a critical factor in inducing faults in the system to check the guarantees the system should provide. Distributed CodEval works on these principles to provide a distributed system assignments evaluation platform.

Tag	Meaning	Function
RUN	Run Script	Specifies the script to evaluate the specification file. Defaults to evaluate.sh.
Z	Download Zip	Will be followed by zip files to download from Canvas to use when running the test cases.
CF	Check Function	Will be followed by a function name and a list of files to check to ensure that the function is used by one of those files.
CMD/TCMD	Run Command	Will be followed by a command to run. The TCMD will cause the evaluation to fail if the command exits with an error.
CMP T/HT	Compare Test Case	Will be followed by two files to compare. Will be followed by the command to run to test the submission.
I/IF	Supply Input	Specifies the input for a test case. The IF version will read the input from a file.
O/OF	Check Output	Specifies the expected output for a test case. The OF version will read from a file.
E	Check Error	Specifies the expected error output for a test case.
TO	Timeout	Specifies the time limit in seconds for a test case to run. It defaults to 20 seconds.
X	Exit Code	Specifies the expected exit code for a test case. It defaults to zero.

Table 1: CodEval specification tags  
Source: [13]

### 3. DESIGN METHODOLOGY

Distributed CodEval expands the functionalities of CodEval [13] to support the automatic evaluation of distributed system assignment submissions. It introduces new tags in the specification file and coordinates the execution of submitted code in multiple Docker containers [21].

#### 3.1 Orchestration

Distributed CodEval executes student submissions in Docker containers [21] to isolate the submission from the host machine and other submissions. CodEval interacts with the submissions through ports exposed in each container. Student submissions may need to interact with an external service, such as ZooKeeper [41] or a central database. These external services, declared in the specification file as explained in Section 3.5.2, run in a special container CodEval called the *controller* container. The controller container is started at the beginning and terminated at the end of each round of tests. Each round of tests is either Homogeneous or Heterogeneous, described in the next section. In each round, a fixed number of containers containing student submissions is started and terminated once or repeatedly, depending on the structure of the tests, as described in Section 3.5.1. Each round of tests has an associated timeout declared in the specification file, as explained in Section 3.5.1. Figure 8 displays a high-level architecture of Distributed CodEval.

#### 3.2 Division of Tests

Distributed assignment tests can be divided into two parts:

1. **Homogeneous tests:** A student's submitted program is executed in a number of containers as the distributed system's individual instances. CodEval executes tests listed in the specification file on these containers running the program.
2. **Heterogeneous tests:** A student's submission is executed in a container,

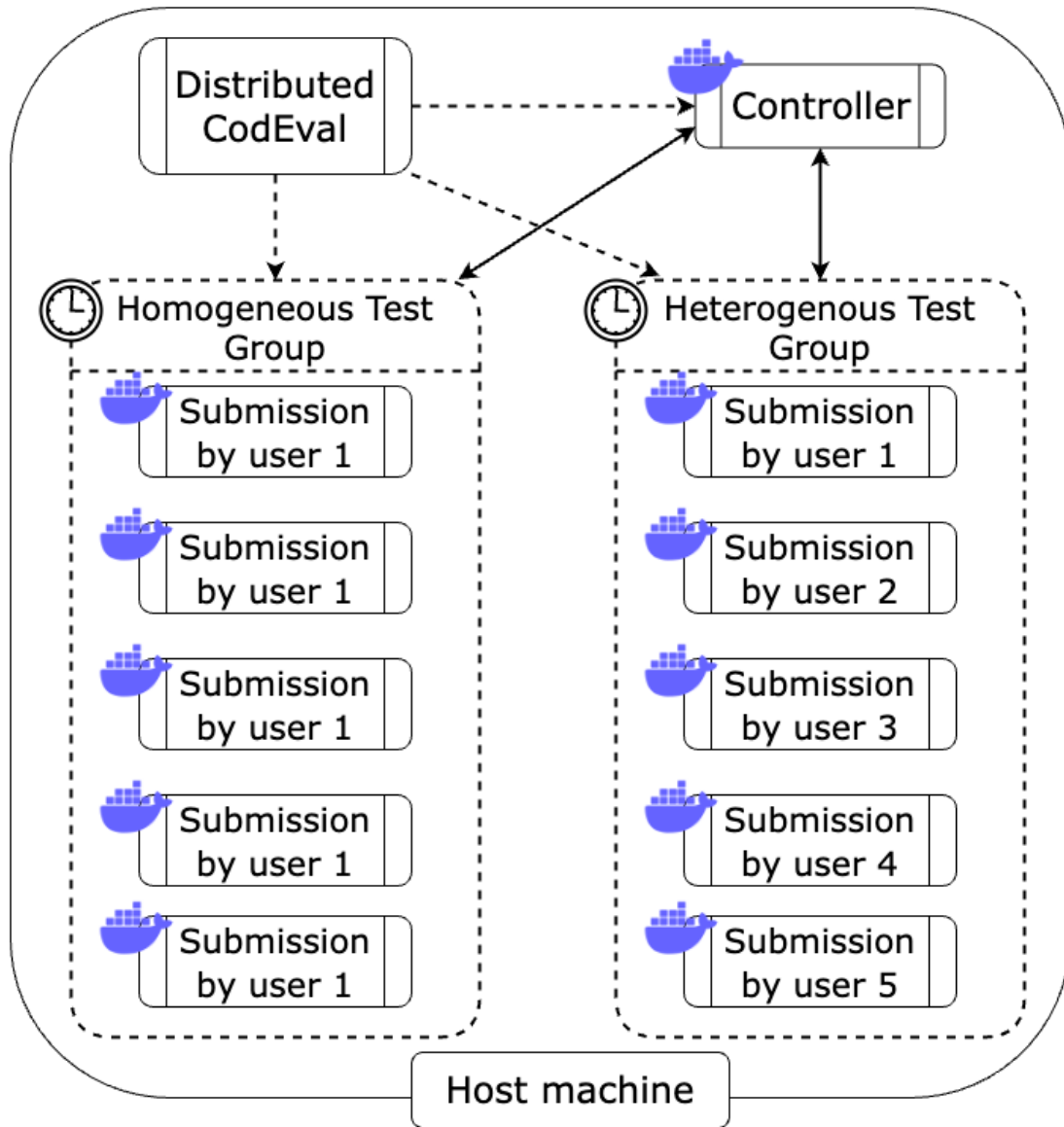


Figure 8: Distributed CodEval architecture

and a fixed number of other students' submitted programs run in separate containers. All the running programs behave as a single distributed system on which CodEval runs tests from a specification file.

Statistically, homogeneous tests have a higher chance of succeeding than hetero-

geneous tests. A primary reason behind this axiom is that students make assumptions when implementing distributed systems, and their program expects the system to behave in a particular way. When the system consists of instances running the same implementation, the assumptions remain valid, and the tests have a higher chance of passing.

Heterogeneous tests execute different implementations in each of its instances, with each implementation built on different assumptions by the students. When these assumptions vary greatly, the same tests that passed in a homogeneous system tend to fail in a heterogeneous system. This intended failure is a necessary part of these assignments, as students are expected to get involved in quality discussions about the possible edge cases in the distributed system. The architecture of heterogeneous tests is explained in the next section.

Logically, CodEval first runs all the distributed tests from a specification file in a homogeneous system. If all the tests pass, then the tests are evaluated in a heterogeneous system. This is based on the reasoning that a failed test in a homogeneous system implies that the student's implementation has a bug and thus has a high probability of failing in a heterogeneous system.

### **3.3 Heterogeneous Tests**

Heterogeneous tests are essential to test the compatibility of student submissions with other students' implementations. A primary challenge is finding the right combinations of student submissions that can pass the test conditions. To overcome this challenge, CodEval uses standalone tests and homogeneous tests as screenings before adding a student submission to a heterogeneous test pool. The implementation of the pool is explained later in this section. On a high level, the tests are grouped into three consecutive levels: standalone, homogeneous, and heterogeneous. Failure

on a level leads to the submission not being evaluated on the next level.

In the pool of submissions eligible for heterogeneous tests, CodEval assigns a score to each submission. The score is used in creating a combination of student submissions to give higher preference to the submissions that have proven to be more resilient in other tests. This gives a new student submission a higher chance of passing heterogeneous tests compared to getting a random combination of submissions from the pool. As CodEval executes tests on each student submission, it is guaranteed that the submission being evaluated will be included and evaluated in all possible combinations.

CodEval generates a combination of  $n$  submissions using the current student's submission and  $n - 1$  submissions from the pool. This  $n$  value is read from the specification file, as described in Section 3.5.1. The first step in the process is to get the data of all submissions from the pool, excluding the entry by the current student, if present. If the number of submissions fetched from the pool is less than  $n - 1$ , the current student submission is added, or replaced if already present, to the pool, and the student is notified of the same.

If at least  $n - 1$  submissions are read from the pool, all the possible combinations having  $n - 1$  submissions are generated using the method `COMBINATIONS` from Python's library `ITERTOOLS` [54]. Then the combinations are sorted in descending order of the sum of the constituent submissions' scores. In the case where two or more combinations have an equal sum of scores, the sum of epochs of submission times of each submission is compared. A higher preference is given to the combination having a lower sum of epochs, which translates to a higher preference for earlier submissions. A primary reason behind this decision was to reward early submissions.

The decision to give higher preference to early submissions in case of equal scores assists in the initial cases when the scores of all submissions in the pool are 0. This



prevents early submissions from starvation of getting evaluated in a combination when new submissions are included by default in their own evaluation.

### 3.4 Heterogeneous Test Submissions Pool

The pool of submissions for heterogeneous tests is implemented in persistent storage instead of in memory. This design helps CodEval to access and manage the pool across multiple executions. The choice of storage used by CodEval is MongoDB [55], as it provides the flexibility of modifying schema on the fly and does not have the overhead of managing relations and transactions. CodEval uses a well-maintained Python library with great community support, *pymongo* [56], to interact with MongoDB.

```
> db.a_6599544.findOne()
{
  "_id" : ObjectId("6452dcd510e13b110470c4b0"),
  "student_id" : "12345678",
  "student_name" : "Test User",
  "submitted_at" : ISODate("2023-05-03T16:17:50Z"),
  "attachments" : [
    {
      "display_name" : "assignment_submission.zip",
      "url" : "https://test.instructure.com/files/123/download?download_frd=1&verifier=testtoken"
    }
  ],
  "score" : 1,
  "active" : true
}
```

Figure 9: A sample heterogeneous test submission pool document

The structure of each document in the pool is fairly simple. Each assignment has its own collection, and each document in the collection represents a student's latest submission in the heterogeneous pool. Figure 9 shows a sample document in the pool. The fields in a document are required to download the student submission, add comments on the submission, and compare scores. The boolean field *active* is used to denote the last eligibility status of student submission to be included in heterogeneous test combinations.

If a student submission passes standalone and homogeneous tests, it gets added to the heterogeneous test pool in MongoDB. If the student submits an updated program that fails in either standalone or homogeneous tests, the respective entry of the student in the heterogeneous test pool is marked as not active. This was preferred over deleting the entry to prevent the loss of scores of past submissions. If a submission is marked as not active in the pool, it does not get included when creating combinations. The submission is marked active once it has passed all the standalone and homogeneous tests again.

The value of scores is only incremented through CodEval. Once a combination of submissions passes all the heterogeneous tests, 1 point is added to each submission in the combination. This helps future submissions get a combination of past submissions that have passed the tests, increasing their own chance of passing the tests. If a student submits a program that passes all the heterogeneous tests and then submits again with a change in the program that leads to failure in any of the tests, the submission in the pool is marked as not active and the score remains the same. This prevents the latest incorrect submission from getting involved in new combinations for other user submissions. Once the bug is fixed and the new submission passes all the tests, the entry in the pool is marked as active again. This lets the new submission participate in future heterogeneous rounds of test. The new successful submission inherits the score from the last successful submission, recovering its priority in the combination pool. This design choice was based on the belief that students aim to improve the functionalities with subsequent submissions.

### **3.5 Specification File**

Distributed CodEval introduces new tags in the specification files. The new tags were designed to keep specification files easy to understand and modify. The

new tags introduced by Distributed CodEval are flexible enough to support simple tests as well as complex testing strategies. These new distributed test cases can be appended to an existing specification file to add distributed test cases without the need to create another file. The student submissions are expected to be in a specific programming language or using a specific framework and should be specified in the assignment description. The specification file contains commands to compile the student submissions and needs to use specific commands related to the language or framework used.

The new tags in Distributed CodEval are divided into two high-level groups: declaration tags and action tags.

### 3.5.1 Declaration Tags

The declaration tags are used to group commands together, mark the beginning of a section of commands, signal the start and end of distributed tests in the specification file, and run commands to verify the state of the system. Unlike action tags, these tags do not modify the state of the distributed system except by booting up or terminating the system as a whole, as explained later. Table 2 lists the declaration tags.

Tag	Meaning
--DT--	Distributed Tests Milestone
GTO	Global Timeout
PORTS	Exposed Ports Count
DTC <int> [HOM] [HET]	Distributed Test Config Group
HINT	Hint message
TESTCMD	Test Command
--DTCLEAN--	Cleanup Commands

Table 2: Declaration tags used in Distributed CodEval specifications

A sample specification file with the usage of each of the declaration tags is

explained in Section 4.3.

The tag `--DT--` marks the beginning of distributed tests in a specification file. It is also used by CodEval to determine if the specification file has distributed tests. It is interpreted by CodEval as a milestone to stop standalone tests and start running distributed tests.

The **GTO** tag specifies a timeout in seconds for each round of distributed tests. Each round consists of either one round of homogeneous tests or a round of heterogeneous tests with one or more combinations of submissions until the tests are successful or there are no more available submissions. Homogeneous tests get a timeout equal to the GTO value. Heterogeneous tests get a timeout of  $2 \times$  GTO value. This is to let a student submission run with at least two combinations or more without getting timed out.

The maximum number of ports to expose per docker container, specified using the **PORTS** tag, should consider the maximum number of ports needed for each submission container and the controller container. This value is used to expose the ports when running the containers, even before the ports are assigned to each process inside the containers.

The **DTC** tag Denotes the start of a new group of distributed tests. `<int>` is the number of containers that need to be running for this test group. **HOM** denotes this group has homogeneous tests. **HET** indicates heterogeneous tests. **DTC** tag groups the tags in the subsequent lines together until the next "DTC" or "--DTCLEAN--" is encountered. The group can contain a combination of Action Tags, described in Section 3.5.2, and **TESTCMD** tags. On reading the "DTC" tag, CodEval starts a new set of submission containers and stops any running containers belonging to the previous group. This is used to provide a test group with a new state of the distributed system to operate upon, preventing any changes from action tags of the previous group

from leaking into the current group. The optional tags "HOM" and "HET" indicate that the test group will be run in homogeneous, heterogeneous, or both rounds of tests. CodEval waits until the specified number of containers is running before executing the commands in the test group.

The tag **TESTCMD** executes the provided command in the controller container to test the correctness of the system. It expects the command to have an exit code of 0 if the test is successful or an exit code of more than 0 if the test fails. A suggested architecture to design a command for checking the state of the system is explained in Section 4.2. If the command fails, the executed command is shared with the student to help them debug the issue.

If **HINT** is declared before **TESTCMD**, the provided hint message is shared with the student instead of the test command. This can be used to define a hidden test case.

The beginning of a section providing cleanup commands is denoted by **--DTCLEAN--**. This section should be at the end of the specification file. Only the "ECMD" and "ECMDT" action tags are supported in this section. These tags modify the state of the controller container and can be used to clean up any effects that might linger on the host machine. The cleanup section is executed after each round of homogeneous or heterogeneous tests, independent of the success or failure of the tests.

### 3.5.2 Action Tags

Action tags are used to modify the state of the system. These tags have additional supportive tags to control the behavior and targets of the respective command. Table 3 lists the action tags.

A sample specification file with the usage of each of the action tags is explained

in Section 4.3.

Tag	Extra Parameters	Meaning
ECMD	SYNC/ASYNC	Non-critical External Command
ECMDT	SYNC/ASYNC	Critical External Command
ICMD	SYNC/ASYNC $*/n_1, n_2, n_3\dots$	Non-critical Internal Command
ICMDT	SYNC/ASYNC $*/n_1, n_2, n_3\dots$	Critical Internal Command

Table 3: Action tags used in Distributed CodEval specifications

All the action tags require an additional supportive tag of **SYNC** or **ASYNC**. CodEval waits for a command marked with SYNC to execute and then proceeds to the next step. For a command marked with the ASYNC tag, CodEval does not wait for its execution to complete. ASYNC commands can be used to run commands that normally do not terminate, such as starting a service.

The tag **ECMD** provides a command to be executed in the controller container. CodEval does not fail the test if the provided command fails to execute or exits with an error. With ASYNC, ECMD is used to execute a non-terminating command, such as starting a service in the controller container that student submissions will interact with. With SYNC, ECMD is used to execute a terminating command, such as updating the state of a service running in the controller container. Section 4.3 provides an example of this command.

The tag **ECMDT** is similar to ECMD, except that CodEval fails the test if the provided command is unable to execute or exits with an error. With SYNC, CodEval waits for a successful termination of the command. With ASYNC, CodEval executes the command and waits 3 seconds for any failures.

The tag **ICMD** provides a command to be executed in the submission containers. Similar to ECMD, CodEval does not fail the test if the command fails. ICMD requires

an argument denoting the containers in which the command needs to be executed. The value of this argument can be either "\*", instructing CodEval to run the command in all the submission containers, or comma-separated integers denoting the container indexes to run the command in. The indexes start from 0 and should have a value less than the number of containers specified with the "DTC" tag. In heterogeneous tests, container index 0 represents the container running the student's own submission. Similar to ECMD, ICMD uses SYNC and ASYNC tags to denote a terminating or non-terminating command, respectively. With SYNC, CodEval executes the command linearly in each container. This can be optimized to run in parallel, as Section 6.9 describes in brief.

The tag ICMDT works similarly to ICMD, except it fails the test if the command fails to execute or terminates with an error in any of the specified containers. With the ASYNC tag, ICMDT instructs CodEval to wait 3 seconds after each execution in the containers for any failure, similar to ECMDT ASYNC.

### 3.5.3 Command Placeholders

CodEval uses placeholders in the command used to run a docker container which is read from a configuration file. The placeholders are used to inject the path to a temporary directory containing student submission and downloaded helper files and to inject a sub-command for executing tests. Using placeholders is beneficial as CodEval is able to take advantage of the execution context and provide configurable commands.

Distributed CodEval extends the usage of placeholders to the commands in specification files. Following are the new placeholders currently used by Distributed CodEval:

- **TEMP\_DIR** is used in commands with ECMD and ECMDT tags. It is

replaced by the temporary directory generated by CodEval containing a student submission and other related files.

- **HOST\_IP** is used in commands with ECMD, ECMDT, ICMD, and ICMDT tags. It is replaced by the IP address provided in the configuration file, explained in Section 3.6. The IP address is generally that of the host machine where CodEval is running.
- **USERNAME** is used in ICMD and ICMDT commands to be replaced with the student's username.
- **PORT\_<int>** is used in ICMD and ICMDT commands. <int> is the index of the assigned ports, which starts from 0 and should be less than the value specified with the PORTS tag. This is used as a placeholder to use dynamically assigned ports by CodEval in the submission containers.
- **H\_PORT\_<int>** is used with ECMD, ECMDT, ICMD, and ICMDT commands to be replaced by a port, indexed at <int>, assigned to the controller container. <int> starts from 0 and should be less than the value specified with the PORTS tag.
- **PEER\_HP[<int>]** is used with TESTCMD commands to be replaced by <int> number of comma separated host-ports of submission containers. The value of the host is the HOST\_IP provided in the configuration file, as shown in Section 3.6, and the port gets the value of the first port assigned to each container.

### 3.6 Configuration Changes

Distributed CodEval requires two new keys under the RUN section, as shown in Figure 10:

- **dist\_command**: Provides a modified form of CodEval command to start a



docker container with exposed ports and without the execution of "evaluatesh".

The command contains two new placeholders:

- NAME is replaced by the name of the container. This is used by CodEval to provide the name "controller" to the controller container and "replica-  
<int>" to submission containers. This helps to recognize the running containers externally and to stop and remove any stale containers from past tests.
- PORTS is replaced by a number of *-p port:port* sub-commands, depending on the value provided with the PORTS tag of specification.
- **host\_ip**: Provides host IP of the machine where CodEval is supposed to execute. The value of host\_ip is used by commands in the specification file, explained in Section 3.5.3.

Distributed CodEval also introduces a new section **MONGO** to provide the details to connect to a MongoDB database. This section expects two keys:

- **url** provides the URL of a running MongoDB instance.
- **db** states the database name CodEval should connect to.

An example of the updated configuration is presented in Section 4.1.

## 4. USING DISTRIBUTED CODEVAL

To use Distributed CodEval, the following steps need to be completed in sequence:

### 4.1 Adding Configuration

CodEval [13] requires a configuration file named "codeval.ini". The location of this file depends on the OS of the machine on which CodEval is supposed to run. [57] provides a command line tool to assist with the creation of a configuration file.

Distributed CodEval requires new keys within the configuration file, listed in Section 3.6. The configuration file looks similar to Figure 10 on adding the required keys.

```
1  [SERVER]
2  url=https://sjsu.instructure.com
3  token=XXXXXXXXXXXX
4  [RUN]
5  precommand=
6  command=docker run -i -v SUBMISSIONS:/submissions -v ~/.m2:/root/.m2
   autograder-java bash -c "cd /submissions; EVALUATE"
7  dist_command=docker run --name NAME -dt -v SUBMISSIONS:/submissions
   -v ~/.m2:/root/.m2 PORTS autograder-java
8  host_ip=172.27.24.40
9  [MONGO]
10 url=mongodb://localhost:27017/
11 db=codeval_submissions
12
```

Figure 10: Sample Distributed CodEval configuration

In the sample configuration, the key *dist\_command* contains a sub-command "-v ~/.m2:/root/.m2". This sub-command is added to optimize maven [58] builds.

### 4.2 Preparing Checker Program and Supporting Files

Distributed CodEval uses a checker command to test the expected state of the system, specified with the TESTCMD tag explained in Section 3.5.1. Checking the state of a system can be difficult to implement in a single standalone bash command

[59]. A program can be written to accomplish the task. The program's executable file and supporting files, if any, can be provided to CodEval. The process of providing such files to CodEval is discussed later. The command to execute the executable file can be specified with the TESTCMD tag, which Distributed CodEval will execute.

Printing logs and error messages to stdout [60] in the checker program is advised as a good practice for using Distributed CodEval. Distributed CodEval reads stdout and stderr [61] of the checker program execution and shares a part of the combined logs in a comment to the student submission if the check fails. This helps students debug the issues with their submissions.

Distributed CodEval expects the checker program to use exit codes [62] for differentiating between a passed and a failed test. Section 4.2 presents a simple checker program architecture that exits with code '1' if the test fails.

It is advised to provide multiple sub-commands in the checker program to check different parts of the system. Multiple sub-commands can be used with different TESTCMD tags in CodEval specification to help the students better understand which part of the system failed the test. Java's PICOCLI [63] and Python's CLICK [64] libraries provide a straightforward interface to expose command-line sub-commands from a program. An example of test cases using Java's PICOCLI library is shown in Figure 11. The library provides instructions to run the sub-commands on executing the generated jar file without any parameters, as shown in Figure 12.

The executable file of the checker program and other files required to run tests on Distributed CodEval can be bundled in a zip file [65] and uploaded to Canvas. The structure of the zip file may look similar to Figure 13. The zip file can be uploaded to Canvas under the related assignment's course. The name of the zip file should be included under the tag "Z" in the test specification file, similar to the sample shown in Section 4.3. Distributed CodEval downloads the zip file and extracts the contents

```

import picocli.CommandLine;
import picocli.CommandLine.Command;
import picocli.CommandLine.Option;

public class SampleChecker {

    public static void main(String[] args) {
        System.exit(new CommandLine(new Cli()).execute(args));
    }

    @Command(mixinStandardHelpOptions = true)
    static class Cli {

        @Option(names = "--zk")
        String zookeeperHostPort;

        @Command
        public void test1() {
            System.out.println("Checking the internal parts");
            boolean passed = checkInternalParts();
            if (!passed) {
                System.out.println("Failed due to internal parts not in expected state");
                System.exit(1);
            }
            System.out.println("Passed the test!");
        }

        @Command
        public void test2() {
            System.out.println("Checking the value of key 'Foo'");
            boolean passed = checkVariableEqualTo("Foo", 1);
            if (!passed) {
                System.out.println("Failed due to 'Foo' not equal to 1");
                System.exit(1);
            }
            System.out.println("Passed the test!");
        }

        private boolean checkInternalParts() {
            // Check some parts of the distributed system here
            return true;
        }

        private boolean checkVariableEqualTo(String key, int expectedValue) {
            // Check the value of key in the distributed system
            return false;
        }
    }
}

```

Figure 11: A sample checker program in Java using PICOCLI library

to the root of the temporary directory created for the student submission, where the student submission is also available. The contents of the temporary directory can be

```

→ SampleChecker java -jar target/SampleChecker-1.0-SNAPSHOT-spring-boot.jar
Missing required subcommand
Usage: <main class> [-hV] [--zk=<zookeeperHostPort>] [COMMAND]
  -h, --help      Show this help message and exit.
  -V, --version   Print version information and exit.
  --zk=<zookeeperHostPort>

```

Commands:

```

test1
test2

```

Figure 12: Execution instructions provided by Java’s PICOCLI library

directly accessed in the action tag commands, as shown in Section 4.3.

```

→ temp ls
assignment_helper.zip
→ temp zipinfo -1 assignment_helper.zip
assignment.proto
checker.jar
zookeeper.jar

```

Figure 13: Contents of a sample helper zip file

### 4.3 Writing a Test Specification

CodEval searches for a test specification file on Canvas for each assignment in a course. The test specification file must be directly inside a folder named *CodEval*. The name of the specification file must be *<assignment name>.codeval*. For example, if the assignment name is "Distributed Hashtable" the respective CodEval specification file must be named as "*Distributed Hashtable.codeval*". A sample specification file used by Distributed CodEval is presented in Figure 14. The expected location of the specification file is shown in Figure 15.

In the example specification file, lines 1 to 11 contain CodEval commands [13]. Line 1 instructs CodEval to download the file *assignment\_helper.zip* and unzip it. Line 3 creates a directory if not present in the student submission directory and copies

```

assignment.codeval
1  Z assignment_helper.zip
2
3  CMD mkdir -p src/main/proto && cp assignment.proto src/main/proto
4
5  CTO 60
6  C mvn -T 1C clean package
7
8  HT test -f ./target/assignment-1.0-spring-boot.jar && echo "File exists" || echo
   "File does not exist"
9  X 0
10 0 File exists
11 HINT Is your code compiling properly? Is it generating a jar with the name
   "assignment-1.0-spring-boot.jar"?
12
13
14 --DT--
15 GTO 300
16 PORTS 1
17
18 ECMD ASYNC java -jar ./submissions/zookeeper.jar server H_PORT_0 datadir
19 ECMD SYNC sleep 5
20 ECMD SYNC java -jar ./submissions/zookeeper.jar client -server HOST_IP:H_PORT_0
   create /test
21 ECMDT SYNC java -jar ./submissions/zookeeper-dev-fatjar.jar client -server
   HOST_IP:H_PORT_0 get /test
22
23 DTC 5 HOM HET
24 ICMD SYNC * mvn -f ./submissions -T 1C clean package
25 ICMDT ASYNC * java -jar ./submissions/target/assignment-1.0-spring-boot.jar USERNAME
   HOST_IP:PORT_0 HOST_IP:H_PORT_0 /test
26 ECMD SYNC sleep 5
27 TESTCMD java -jar ./submissions/checker.jar --zk=HOST_IP:H_PORT_0 --znode=/test test1
28
29 DTC 4 HET
30 ICMD SYNC * mvn -f ./submissions -T 1C clean package
31 ICMDT ASYNC 0,1,3 java -jar ./submissions/target/assignment-1.0-spring-boot.jar
   USERNAME HOST_IP:PORT_0 HOST_IP:H_PORT_0 /test
32 ECMD SYNC sleep 5
33 TESTCMD java -jar ./submissions/checker.jar --zk=HOST_IP:H_PORT_0 --znode=/test test2
34
35 DTCLEAN
36 ECMD SYNC java -jar ./submissions/zookeeper.jar client -server HOST_IP:H_PORT_0
   delete /test

```

Figure 14: Contents of a sample Distributed CodEval specification file

a file from the unzipped helper file to the created directory. Lines 5 and 6 specify a compile timeout and command to compile the student submission. Lines 8 to 11



Name	Date Created	Date Modified	Modified By	Size
 assignment_helper.zip	May 7, 2023	May 7, 2023	Nimesh Nischal	27.3 MB
 assignment.codeval	May 7, 2023	May 7, 2023	Nimesh Nischal	1 KB

Figure 15: Location of a Distributed CodEval specification file in a Canvas course

include a hidden test command that checks for the compilation output and prints a message to the student submission if the test fails.

Line 14 marks the beginning of Distributed CodEval section of the specification file. Line 15 provides a global timeout of 300 seconds for each round of tests, as explained in Section 3.5.1. Line 16 declares only one port will be exposed in the controller container and each submission container. Here, a controller container is started with access to the unzipped files and an exposed port chosen by CodEval during runtime.

Lines 18 to 22 contain ECMD and ECMDT commands before any DTC tag has been defined. This section is used to initialize the controller container and can contain only ECMD and ECMDT tags. Line 18 starts a ZooKeeper server in the controller container asynchronously using one of the files provided in the zip file. The ZooKeeper server listens on a port specified by the placeholder  $H\_PORT\_0$ , which is replaced by the first port number assigned to the controller container during its initialization. Line 19 instructs CodEval to sleep for 5 seconds to let the ASYNC command in line 18 accomplish its task. Line 20 synchronously creates a znode `"/test"` to be used for tests. Line 21 verifies that the znode `"/test"` exists. CodEval fails the test due to ECMDT if the znode is not present or the command fails for any other reason.

Line 23 declares a distributed test config group that needs five containers to run, and the test group must be included in both homogeneous and heterogeneous rounds of tests. CodEval starts five containers here and exposes a unique port on each container. If the current round of tests is homogeneous, all five containers contain the student's program whose submission is being evaluated. If a heterogeneous round of tests is running, the first container contains the current student's submission, and the other four containers contain four other students' submissions, according to a combination generated by Distributed CodEval, described in Section 3.3. Line 24 runs the command synchronously in all the submission containers to compile the programs. Line 25 contains the command to run asynchronously in all the submission containers. The placeholders, described in Section 3.5.3, are replaced with the respective values. ICMDT fails the test round if the command in any container exits with a failure. The example command executes the submission programs inside their respective containers. CodEval waits five seconds due to line 26, an adequate time to wait for the programs started in line 25 to finish initializing. Line 27 specifies a test command, using the sub-command "test1", that is executed in the controller container using the ".jar" file obtained from the zip file.

Line 29 defines another distributed test config group that needs four containers to run. Distributed CodEval stops, removes, and starts all the submission containers again. This test group is marked as heterogeneous only. Line 30 is similar to line 24, where it compiles the submission in all the containers. Line 31 specifies an example of an ICMDT command running only on containers with indexes 0, 1, and 3, respectively. This command is presented as an example of a selective ICMDT tag and does not conduct any significant action. Lines 32 and 33 are similar to lines 26 and 27, where CodEval sleeps for 5 seconds and then executes the sub-command "test2" to test the system. If the checker program expects all four instances of the program to run, it



will fail the test.

Line 35 denotes the tests' end and the clean-up zone's start. This zone is optional and supports only ECMD and ECMDT tags, as shown in line 36.

This specification file must be uploaded to Canvas under a folder with the name "CodEval".

#### 4.4 Running Distributed CodEval

Distributed CodEval is executed using the same command as CodEval [13]:

```
$ python3 codeval.py grade-submissions "<course-name>"
```

The course name can also contain a part of it and should be able to identify the course on Canvas uniquely. The above command can be run with the following optional flags:

- **--dry-run / --no-dry-run**: The dry run flag prevents CodEval from making any updates on Canvas and prevents Distributed CodEval from updating MongoDB collections.
- **--verbose / --no-verbose**: The verbose flag enables CodEval to print additional logs to the console, which can help debug any issues with CodEval or specification files.
- **--force / --no-force**: The flag allows CodEval to evaluate a student's already evaluated submission.
- **--copytmpdir / --no-copytmpdir**: The "copytmpdir" flag copies the temporary directory, containing the student submission and related files extracted from the zip, to the current directory for debugging purposes after CodEval's execution.

CodEval is generally used with crontab [66] to execute in intervals for a Canvas course. The cron job should be scheduled with an interval of at least  $CTO + 3 \times GTO$

seconds. This provides CodEval enough time to execute a round of standalone tests, a round of homogeneous tests, and at least two rounds of heterogeneous tests, after which the tests timeout. This recommended interval prevents subsequent executions of CodEval from interfering with each other's progress.

## 5. DIFFICULTIES AND CHALLENGES

Designing and developing Distributed CodEval came with several challenges that needed to be resolved to make progress and improve the user experience of the tool. Some of the challenges are discussed here briefly.

### 5.1 No Support for Maven Projects on CodEval

CodEval [13] does not support testing maven projects as its Docker image does not contain maven dependencies. Extra commands had to be added in the Dockerfile [67] definition to install and use maven in the CodEval docker containers. CodEval uses jdk version 17 [68] to evaluate Java projects. A specific version of maven (v3.8.6) [69] was installed on the Docker image for compatibility with Java 17.

### 5.2 Stale Docker Containers

Docker containers kept running submission programs and/or controller logic when the Distributed CodEval process got interrupted or crashed due to errors. These stale Docker containers had to be manually stopped and removed as they were occupying host machine resources and could have intervened with future executions of CodEval.

To solve this issue, running containers are given the name "replica- $\langle$ int $\rangle$ " where "int" is the container index, and before starting new containers, any running container with the same name is killed and removed by Distributed CodEval. This also prevents the unbounded growth of stale docker containers.

### 5.3 Factorial Growth of Unique Student Submission Combinations

In heterogeneous testing, the number of unique possible student submission combinations has a factorial growth rate with respect to the submission count. For example, selecting 4 out of 15 submissions will have 1365 unique possibilities, 4845 possibilities when selecting from 20 submissions, and 12650 possible combinations with 25 submissions. Figure 16 shows this exponential rate of growth. Executing heterogeneous tests on all the possible combinations will take a huge amount of time

and resources, and CodEval will not be able to provide immediate feedback to the students.

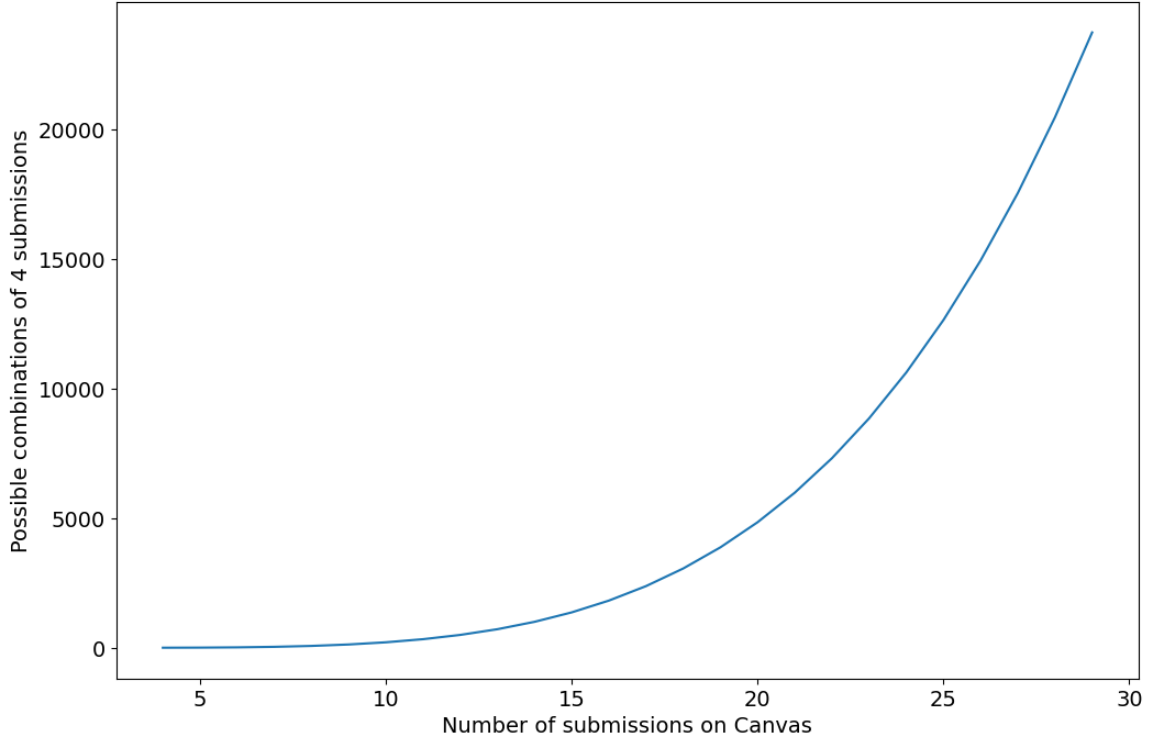


Figure 16: Graph showing a growth rate of possible heterogeneous test combinations of 4 submissions versus the number of submissions on Canvas

To overcome this problem, student submissions are ranked with the highest possibility of passing heterogeneous tests on past tests performance by maintaining a score. Combinations are formed by giving preference to high-scoring submissions so that a new submission will have the highest chance of passing the tests. Also, heterogeneous tests execute within a time bound within which at least two combinations can be tested.

## 6. FUTURE WORK AND IMPROVEMENTS

Distributed CodEval is built to provide instructors with automatic evaluation capabilities and faster student feedback. This tool can be optimized further to provide faster feedback, use fewer computing resources, provide new ways to test submissions and handle edge cases better, as discussed below.

### 6.1 Optimize Test Duration for Complex Assignments

Complex distributed system assignments can take a long time to run tests on multiple submissions, particularly when there are numerous test cases and the timeouts defined in the specification file are more than 5 minutes. This will affect the interval between each execution of CodEval as a cron job. This results in students waiting for extended periods of time between submissions and getting feedback on their submissions.

This problem of extended execution time can be reduced by performing parallel tests on each student submission, either using multiple threads or running tests on different servers.

### 6.2 Optimizing Initial Heterogeneous Combinations

There can be a case when there are a large number of submissions in the heterogeneous test pool. One possible way this can happen is if all the submissions pass the standalone and homogeneous tests but are unable to pass the heterogeneous tests. A new submission will get a large number of submissions to form a heterogeneous combination from where all the other submissions have the same score, i.e., 0. In this case, the submissions that were uploaded earliest will be chosen. If the first combination fails, Distributed CodEval will keep replacing one of the submissions with a newer one until the tests pass or get timed out.

These initial combinations from a large pool can have a better chance of success if the combinations vary by a greater degree after each round of failed tests. If the

oldest submission in the combination of size  $n$  has a fault due to which the tests are failing, the current implementation of Distributed CodEval will not replace the oldest submission until  $n$  rounds of tests have been executed. If instead of 1, 2 submissions are replaced, then  $n/2$  rounds will replace the oldest submission. This level of variation can be optimized through further research.

### 6.3 Support for Range in ICMD and ICMDT Commands

ICMD and ICMDT tags require an argument providing the container indexes on which the command needs to be executed. The current format of this argument is either "\*" or a comma-separated list of whole numbers, such as "0,2,4". If the list of numbers is large and consecutive, the instruction looks bloated. Support for another format can be added to provide a better user experience in defining specification files. This format can take a range of whole numbers. Following is a comparison of the existing and the new suggested instruction:

Current:

```
ICMD SYNC 2,3,4,5,6,7,8,9 java -jar submission.jar
```

Suggested new format:

```
ICMD SYNC 2-9 java -jar submission.jar
```

### 6.4 Configurable Heterogeneous Tests Timeout

Currently, heterogeneous tests have a fixed timeout of  $2 \times \text{GTO}$  value. This can be made configurable in the specification file to either a multiple of the GTO value or a separate timeout value in seconds.

### 6.5 No Orphan Docker Containers

In the current implementation of Distributed CodEval, if the executing process is interrupted between tests, the docker containers started by distributed tests may remain in the running state. These orphan container processes will be stopped in

the subsequent execution of Distributed CodEval, but they use precious computing resources until then. Distributed CodEval can be extended to handle interruptions by killing spawned docker processes.

## **6.6 Network Faults Between Containers**

Distributed CodEval can be extended by supporting the introduction and removal of network faults in the system. A suggested way of doing so is by supporting new tags in specification files. The tag and the respective command can control network faults between pairs of submission containers and between a submission container and the controller container. The network faults can be implemented using the Linux *iptables* tool [38].

## **6.7 Clearing Data From MongoDB**

New execution commands can be introduced in Distributed CodEval to list and clear stale assignment data from MongoDB. A suggested way to achieve this is to store the respective metadata for an assignment, such as assignment name and deadline date-time, in a separate collection for each assignment present in the database. A new command can print the assignment details present in MongoDB, and another command can delete the related data for a given assignment name or id.

## **6.8 Configurable Wait Time for ECMDT ASYNC and ICMDT ASYNC Tag Commands**

The current implementation of Distributed CodEval waits 3 seconds after executing the command provided with ECMDT ASYNC or ICMDT ASYNC tags to check for any failures. The wait time can be made configurable in the specification files to provide more control to the instructor.

## 6.9 Parallel Execution of ICMD SYNC and ICMDT SYNC Tag Commands

Distributed CodEval executed ICMD SYNC and ICMDT SYNC tag commands linearly within each submission container, waiting for the execution to successfully complete before starting the execution in the next container. This can be optimized to improve the total test time by executing these commands in parallel.

This suggestion presents a set of challenges as it introduces non-determinism in the state of the distributed system that depends on the order of execution of such commands. Parallelism can be made configurable in the specification file, or Distributed CodEval can provide a false sense of parallelism by introducing small delays between each subsequent parallel execution.



## 7. CONCLUSION

In summary, understanding the challenges of distributed systems is a difficult task and the best way to learn is by building a resilient distributed system that works with other implementations. Bake-off styled assignments have been effective to introduce students to such situations early.

Distributed CodEval reduces the instructor's effort to evaluate bake-off style assignments. The instructor needs to create a program to check the different states of the system, create a test specification file, and upload all the supporting files to Canvas. The instructor can automate the evaluation of distributed assignment submissions by setting up Distributed CodEval to run at regular intervals for a course.

The student submissions are automatically evaluated based on the test cases defined by the instructor. The results of the tests are notified to the students as a comment on their submission on Canvas. Distributed CodEval also tries to find a combination of submissions that can pass the test cases and notifies the involved students of the result. This helps the students to get prompt feedback on their submissions. Distributed systems is still a complex topic, but Distributed CodEval makes understanding it a bit easier.

## REFERENCES

- [1] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 249–265.
- [2] J. S. Ng, W. Y. B. Lim, N. C. Luong, Z. Xiong, A. Asheralieva, D. Niyato, C. Leung, and C. Miao, "A comprehensive survey on coded distributed computing: Fundamentals, challenges, and networking applications," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 3, pp. 1800–1837, 2021.
- [3] M. Paprzycki, "Education: Integrating parallel and distributed computing in computer science curricula," *IEEE Distributed Systems Online*, vol. 7, no. 2, pp. 6–6, 2006.
- [4] M. Gowanlock and B. Gallet, "Data-intensive computing modules for teaching parallel and distributed computing," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 350–357.
- [5] E. Saule, "Experiences on teaching parallel and distributed computing for undergraduates," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 361–368.
- [6] J. DeNero, S. Sridhara, M. Pérez-Quñones, A. Nayak, and B. Leong, "Beyond autograding: Advances in student feedback platforms," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 651–652. [Online]. Available: <https://doi.org/10.1145/3017680.3017686>
- [7] C. Wilcox, "Testing strategies for the automated grading of student programs," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 437–442. [Online]. Available: <https://doi-org.libaccess.sjlibrary.org/10.1145/2839509.2844616>
- [8] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.
- [9] E. Maicus, M. Peveler, S. Patterson, and B. Cutler, "Autograding distributed algorithms in networked containers," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '19. New York,

- NY, USA: Association for Computing Machinery, 2019, p. 133–138. [Online]. Available: <https://doi-org.libaccess.sjlibrary.org/10.1145/3287324.3287505>
- [10] C. Torens and L. Ebrecht, “Remotetest: A framework for testing distributed systems,” in *2010 Fifth International Conference on Software Engineering Advances*, 2010, pp. 441–446.
- [11] A. Marroquin, D. Gonzalez, and S. Maag, “Testing distributed systems with test cases dependencies architecture,” in *2015 7th IEEE Latin-American Conference on Communications (LATINCOM)*, 2015, pp. 1–6.
- [12] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, “Debugging distributed systems: Challenges and options for validation and debugging,” *Queue*, vol. 14, no. 2, p. 91–110, mar 2016. [Online]. Available: <https://doi.org/10.1145/2927299.2940294>
- [13] A. Agrawal, A. Jain, and B. Reed, “Codeval: Improving student success in programming assignments,” in *EDULEARN Proceedings*. IATED, jul 2022. [Online]. Available: <https://doi.org/10.21125%2Fedulearn.2022.1767>
- [14] “Manage coursework with github classroom - github docs,” <https://docs.github.com/en/education/manage-coursework-with-github-classroom>, (Accessed on 12/04/2022).
- [15] “Use autograding - github docs,” <https://docs.github.com/en/education/manage-coursework-with-github-classroom/teach-with-github-classroom/use-autograding>, (Accessed on 12/04/2022).
- [16] “Simple autograding with github classroom + github actions + cml container | matsui-lab blog,” <https://mti-lab.github.io/blog/2021/12/15/autograding.html>, (Accessed on 12/04/2022).
- [17] N. Slamnik-Kriještorec, H. C. C. de Resende, and J. M. Marquez-Barja, “Practical teaching of distributed systems: A scalable environment for on-demand remote experimentation,” in *Proceedings of the 6th EAI International Conference on Smart Objects and Technologies for Social Good*, ser. GoodTechs ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 120–125. [Online]. Available: <https://doi.org/10.1145/3411170.3411230>
- [18] “Codegrade - virtual assistant for your coding classroom,” <https://www.codegrade.com/>, (Accessed on 12/04/2022).
- [19] “Starting your autotest - codegrade help,” <https://help.codegrade.com/faq/starting-your-autotest>, (Accessed on 12/04/2022).

- [20] “Tests — codegrade quietstorm.1 documentation,” <https://docs.codegra.de/user/autotest/tests.html>, (Accessed on 12/04/2022).
- [21] “What is a container? - docker,” <https://www.docker.com/resources/what-container/>, (Accessed on 12/04/2022).
- [22] L. Tobarra, S. Ros, R. Hernández, A. Marcos-Barreiro, A. Robles-Gómez, A. C. Caminero, R. Pastor, and M. Castro, “Creation of customized remote laboratories using deconstruction,” *IEEE Revista Iberoamericana de Tecnologías del Aprendizaje*, vol. 10, no. 2, pp. 69--76, 2015.
- [23] M. A. Vivar and A. R. Magna, “Design, implementation and use of a remote network lab as an aid to support teaching computer network,” in *2008 Third International Conference on Digital Information Management*, 2008, pp. 905--909.
- [24] “Java communications api,” <https://www.oracle.com/java/technologies/java-communications-api.html>, (Accessed on 12/04/2022).
- [25] K. Alnawasreh, P. Pelliccione, Z. Hao, M. Rånge, and A. Bertolino, “Online robustness testing of distributed embedded systems: An industrial approach,” in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 133--142.
- [26] “Terra nullius,” [https://alexei-led.github.io/post/pumba\\_docker\\_chaos\\_testing/](https://alexei-led.github.io/post/pumba_docker_chaos_testing/), (Accessed on 12/07/2022).
- [27] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, “Fate and destini: A framework for cloud recovery testing,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’11. USA: USENIX Association, 2011, p. 238–252.
- [28] “Datalog: Deductive database programming,” <https://docs.racket-lang.org/datalog/>, (Accessed on 12/04/2022).
- [29] “The aspectj project | the eclipse foundation,” <https://www.eclipse.org/aspectj/>, (Accessed on 12/04/2022).
- [30] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou, “Setsudo: Perturbation-based testing framework for scalable distributed systems,” in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, ser. TRIOS ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: <https://doi.org/10.1145/2524211.2524217>

- [31] “Using java reflection,” <https://www.oracle.com/technical-resources/articles/java/javareflection.html>, (Accessed on 12/04/2022).
- [32] “Aspect-oriented programming - wikipedia,” [https://en.wikipedia.org/wiki/Aspect-oriented\\_programming](https://en.wikipedia.org/wiki/Aspect-oriented_programming), (Accessed on 12/04/2022).
- [33] R. Lübke, D. Schuster, and A. Schill, “Nessee: An in-house test platform for large scale tests of multimedia applications including network behavior,” in *Testbeds and Research Infrastructure: Development of Networks and Communities*, V. C. Leung, M. Chen, J. Wan, and Y. Zhang, Eds. Cham: Springer International Publishing, 2014, pp. 229–238.
- [34] “jepson-io/jepson: A framework for distributed systems verification, with fault injection,” <https://github.com/jepson-io/jepson>, (Accessed on 12/10/2022).
- [35] “Jepson analyses,” <https://jepson.io/analyses>, (Accessed on 12/10/2022).
- [36] “ssh command usage, options, and configuration in linux/unix.” <https://www.ssh.com/academy/ssh/command>, (Accessed on 12/10/2022).
- [37] “jepson/05-nemesis.md at main · jepson-io/jepson,” <https://github.com/jepson-io/jepson/blob/main/doc/tutorial/05-nemesis.md>, (Accessed on 12/10/2022).
- [38] “iptables(8) - linux man page,” <https://linux.die.net/man/8/iptables>, (Accessed on 12/10/2022).
- [39] “tc(8) - linux manual page,” <https://man7.org/linux/man-pages/man8/tc.8.html#QDISCS>, (Accessed on 12/10/2022).
- [40] “jepson/net.clj at main · jepson-io/jepson,” <https://github.com/jepson-io/jepson/blob/main/jepson/src/jepson/net.clj>, (Accessed on 12/10/2022).
- [41] “Apache zookeeper,” <https://zookeeper.apache.org/>, (Accessed on 12/10/2022).
- [42] “Zookeeper: Because coordinating distributed systems is a zoo,” [https://zookeeper.apache.org/doc/current/zookeeperTools.html?fbclid=IwAR0eAIQ7Yy-DyTcXT8B10oXlQ3t6v4\\_asJSqLaDqGfMrND0aekyW\\_rC2pGQ](https://zookeeper.apache.org/doc/current/zookeeperTools.html?fbclid=IwAR0eAIQ7Yy-DyTcXT8B10oXlQ3t6v4_asJSqLaDqGfMrND0aekyW_rC2pGQ), (Accessed on 12/10/2022).
- [43] “Byteman homepage · jboss community,” <https://byteman.jboss.org/>, (Accessed on 12/10/2022).
- [44] B. Lima, “Automated scenario-based integration testing of time-constrained distributed systems,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 486–488.

- [45] “Sequence diagram - wikipedia,” [https://en.wikipedia.org/wiki/Sequence\\_diagram](https://en.wikipedia.org/wiki/Sequence_diagram), (Accessed on 12/05/2022).
- [46] E. Durr and J. van Katwijk, “Vdm++, a formal specification language for object-oriented designs,” in *CompEuro 1992 Proceedings Computer Systems and Software Engineering*, 1992, pp. 214--219.
- [47] “Overview,” <https://www.overturetool.org/>, (Accessed on 12/05/2022).
- [48] W. D. Yu and G. Patil, “A workflow-based test automation framework for web based systems,” in *2007 12th IEEE Symposium on Computers and Communications*, 2007, pp. 333--339.
- [49] “Workflow management system - wikipedia,” [https://en.wikipedia.org/wiki/Workflow\\_management\\_system](https://en.wikipedia.org/wiki/Workflow_management_system), (Accessed on 12/05/2022).
- [50] “.net remoting - wikipedia,” [https://en.wikipedia.org/wiki/.NET\\_Remoting](https://en.wikipedia.org/wiki/.NET_Remoting), (Accessed on 12/05/2022).
- [51] “Introduction to windows service applications - .net framework | microsoft learn,” <https://learn.microsoft.com/en-us/dotnet/framework/windows-services/introduction-to-windows-service-applications>, (Accessed on 12/05/2022).
- [52] R. Lübke, R. Lungwitz, D. Schuster, and A. Schill, “Large-scale tests of distributed systems with integrated emulation of advanced network behavior,” *IADIS International Journal on WWW/Internet*, vol. 10, pp. 138--151, 01 2013.
- [53] “Canvas by instructure | world’s #1 teaching and learning software,” <https://www.instructure.com/canvas>, (Accessed on 12/11/2022).
- [54] “itertools — functions creating iterators for efficient looping — python 3.11.3 documentation,” <https://docs.python.org/3/library/itertools.html>, (Accessed on 04/25/2023).
- [55] “MongoDB: The developer data platform,” <https://www.mongodb.com/>, (Accessed on 04/25/2023).
- [56] “Pymongo 4.3.3 documentation — pymongo 4.3.3 documentation,” <https://pymongo.readthedocs.io/en/stable/>, (Accessed on 04/25/2023).
- [57] “Sjsu-cs-systems-group/canvas\_tool: a command-line python based tool for teachers who use canvas.” [https://github.com/SJSU-CS-systems-group/canvas\\_tool](https://github.com/SJSU-CS-systems-group/canvas_tool), (Accessed on 04/25/2023).
- [58] “Maven – introduction,” <https://maven.apache.org/what-is-maven.html>, (Accessed on 12/11/2022).

- [59] “Bash - gnu project - free software foundation,” <https://www.gnu.org/software/bash/>, (Accessed on 04/25/2023).
- [60] “stdout(3): standard i/o streams - linux man page,” <https://linux.die.net/man/3/stdout>, (Accessed on 04/25/2023).
- [61] “stderr(3): standard i/o streams - linux man page,” <https://linux.die.net/man/3/stderr>, (Accessed on 04/25/2023).
- [62] “Exit status - wikipedia,” [https://en.wikipedia.org/wiki/Exit\\_status](https://en.wikipedia.org/wiki/Exit_status), (Accessed on 04/25/2023).
- [63] “picocli - a mighty tiny command line interface,” <https://picocli.info/>, (Accessed on 04/25/2023).
- [64] “Click | the pallets projects,” <https://palletsprojects.com/p/click/>, (Accessed on 04/25/2023).
- [65] “Zip (file format) - wikipedia,” [https://en.wikipedia.org/wiki/ZIP\\_\(file\\_format\)](https://en.wikipedia.org/wiki/ZIP_(file_format)), (Accessed on 04/25/2023).
- [66] “crontab(5) - linux manual page,” <https://man7.org/linux/man-pages/man5/crontab.5.html>, (Accessed on 04/25/2023).
- [67] “Dockerfile reference | docker documentation,” <https://docs.docker.com/engine/reference/builder/>, (Accessed on 04/25/2023).
- [68] “Jdk 17,” <https://openjdk.org/projects/jdk/17/>, (Accessed on 04/25/2023).
- [69] “Maven – release notes – maven 3.8.6,” <https://maven.apache.org/docs/3.8.6/release-notes.html>, (Accessed on 04/25/2023).