Pursuing Ideal Regression Testing for HPC: A Comparative Framework Analysis

Seminar Thesis
Louis Wellmeyer

Chair for High Performance Computing, IT Center,
RWTH Aachen, Seffenter Weg 23,
52074 Aachen, Germany
Supervisor: Felix Tomski

Regression testing involves running tests after system updates to ensure that functionalities continue to work as intended and that no new errors have been introduced. High-performance computing (HPC) systems are prone to errors due to their complexity, resource intensity, and frequent distribution across multiple nodes, making regression testing a crucial instrument in these systems. To enhance the efficiency of the testing workflow, this paper discusses modern regression testing frameworks, namely ReFrame, BuildTest, and Testpilot. A comparison reveals similar architectures while offering distinct functionalities, highlighting unique qualities. This research explores the trade-offs and drawbacks emerging when employing specific functionalities in those frameworks. It proposes solutions by combining frameworks and integrating alternative tools for data logging, management, and automation features. Consequently, an ideal regression testing environment for HPC systems is outlined in this research.

Keywords: HPC, High Performance Computing, Regression Testing, Testing Frameworks, ReFrame, BuildTest, Testpilot

1 Introduction

High-performance computing (HPC) systems are highly complex and often distributed in terms of architecture and software stack. Consequently, failures are not uncommon when introducing small changes [4]. For that reason, tests and performance checks must be executed during each update process to detect and prevent errors.

Testing a system for regression is easier said than done. On the one hand, tests ranging from simple unit tests to extensive benchmarks on the whole cluster have to be implemented and employed. On the other hand, managing and maintaining them is even more challenging as every small change can demand adjustments in many tests, the number of which grows further as the system scales.

Therefore, strategic approaches are desired to minimize time consumption and human action when writing and executing tests. Testing frameworks offer exactly those approaches. Commonly, test frameworks provide a structure for organizing and defining tests. Most test frameworks provide features to help write extensive tests with high coverage.

In this paper, three commonly employed modern testing frameworks for HPC systems, ReFrame [11], BuildTest [1], and Testpilot [4] are compared in detail in Section 2, from their infrastructure, test definition, up to essential quality attributes like efficiency, ease-of-use, maintainability, performance, and more. Subsequently, Section 3 explains what an ideal regression testing framework looks like and introduces approaches to create an optimal testing environment. Section 4 summarizes the accomplishments of this research, followed by a presentation of future research in Section 5.

2 Comparison

On the quest to find an ideal testing framework for HPC systems, the following section compares three commonly employed yet differently approached testing frameworks: ReFrame, BuildTest, and Testpilot. This comparison examines these modern frameworks' unique features and functionalities, offering insights into their applicability and effectiveness in high-performance computing.

Initially publicly released in May 2017, ReFrame is an open-source framework for writing system

regression tests and benchmarks targeted to HPC systems. ReFrame's primary aim is to abstract away the system-related details so that users must solely focus on writing the test's functionalities. [11]

BuildTest was founded in 2017 as a single master script to run several dozen test scripts targeted at core HPC components at Pfizer [12]. It is now being developed at the National Energy Research Scientific Computing Center (NERSC). The framework is community-driven, intended to provide the ability to share test configurations among the community [13].

Not open source, Testpilot is a testing framework to monitor an HPC cluster's health continuously [4]. Additionally, it supports various testing scenarios, including application testing and operating system updates.

For a detailed comparison, key categories of characteristics are established to assess and contrast the strengths and weaknesses of ReFrame, BuildTest, and Testpilot.

```
site_configuration = {
    'systems': [
        {
            'name': 'daint',
            'descr': 'Piz Daint Supercomputer',
            'hostnames': ['daint'],
            'modules_system': 'tmod32',
            'partitions': [
                {
                    ...
                },
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C gpu', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'nvidia', 'cray'],
                    'max_jobs': 100,
                },
                ...
            ]
        }
    ],
    'environments': [
        {
            'name': 'gnu',
            'modules': ['PrgEnv-gnu'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        ...
    ]  # end of environments
    'logging': [
        {
            ...
        }
    ]  # end of logging
}
```

Figure 1: Snippet of an example Python configuration file in ReFrame found in [11]

2.1 Portability

HPC infrastructures vary widely and undergo continuous evolution. To mitigate the risk of faults introduced by a small hardware change and make the framework more future proof, it is crucial for the framework to provide a high degree of independence from the underlying system. Ideally, a framework is desired that functions seamlessly on a spectrum of environments, requiring minimal or no configuration of system specifications. Furthermore, existing tests should not be affected by system configurations, and a new system should support those with minimal changes.

[11] explains that ReFrame can easily be set up on any cluster. All the system interaction mechanisms are implemented as backends and are not exposed directly to writers of tests. Consequently, the same test can be run on different systems. Tests can be run in the same partition and environment as ReFrame without needing any additional configuration. Other partitions and environments must be specified in a YAML or Python configuration file (example in Figure 1). To configure a system partition, the minimum information includes the job scheduler, the parallel job launcher, and the programming environments. Symbolic reference names are assigned to programming environments for tests. Redefinition is possible, enhancing test maintainability.

[13] unveils that BuildTest can be run on any Linux and MacOS system. The framework supports IBM Spectrum LSF, Slurm, PBS, and Cobald batch schedulers [1], all GNU and Intel compilers, in addition to OpenMPI and MPICH [12]. Unlike the other frameworks, Karakasis et al. [9] states additional decoupling from the underlying system in BuildTest by having all information, including job-scheduler and environment modules, in one distinct YAML configuration file. As explained in [1], the configuration file is provided by default and supports Linux and Mac. It can be freely adjusted to fit the user's system. It is also possible to define multiple clusters. The approach of locating all system details to one configuration file significantly eases maintainability across diverse systems by minimizing the required test modifications.

Also Testpilot can be seamlessly configured on a variety of clusters, though it is important to note that Testpilot requires the use of PBS Schedulers [4]. Nevertheless, there are effective workarounds available to address this limitation. Refer to Section 3 for insights into overcoming this requirement. One disadvantage of Testpilot, compared to the other frameworks, is that job scheduler-specific details are defined in each test and are not handled by the framework. This requires tests to be adjusted by users when moving to a different system.

2.2 Architecture

The core of any framework is its underlying architecture. The architecture comprises the framework's pipeline from test creation to execution to everything after that. It is also essential to discover the programming languages utilized and design choices

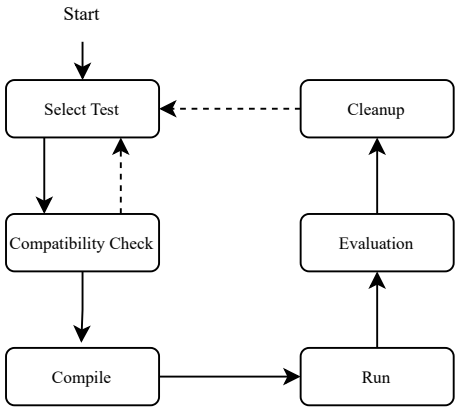

Figure 2: Generic regression pipeline for every test

that influence the framework's functionality.

The test pipeline is the heart of all test frameworks. Each test in any framework follows a similar set of phases in order, which is depicted in Figure 2. Detailed definitions of each phase for the individual frameworks are discussed in the following.

ReFrame is written in Python3. In ReFrame, tests are loaded as test cases, represented as tuples. Each tuple consists of a clone of the original test and each combination of the system partition and programming environment specified in the body of the test. The clone ensures that no test state is reused. The pipeline in Figure 2 in conjunction with ReFrame's official documentation [11] is defined as follows: According to [9], in the first phase, each test is selected and tried for all current system's partitions and each programming environment supported by the partitions during the compatibility check. In the compile phase, a job script for the test compilation is created and submitted for execution. During the run phase, a job script for the test case is submitted for execution and executed asynchronously via a batch scheduler or locally. After execution, the test output is stored, logged, and checked against specified patterns or compared to reference values. ReFrame assists in determining the output's sanity and performance but does not make assumptions on its own. Both phases can be skipped. Lastly, job scripts and their outputs are copied to ReFrame's output directory, and other files created during testing are deleted.

BuildTest, implemented in Python3, introduces a YAML interface known as "buildspec" for writing tests. Buildspecs are validated by the parser using a globally defined JSON schema. Each test is then individually validated using another sub-schema depending on the specified `type` field inside the test. Following the structure of Figure 2, BuildTest's test pipeline is defined by [1] as follows: First, the framework finds buildspecs based on command line arguments to build by file, directory, scheduler, or tag. Subsequently, compatibility is checked by validating the buildspecs and tests using the JSON schemas. Invalid buildspecs are ignored, and if an error occurs during compilation, an exception is raised and the buildspec is also ignored. During the compile phase, executable shell scripts are generated from the valid YAML buildspecs. The framework then executes the test scripts locally or through a batch scheduler, retrieving output and error files. Following the execution, BuildTest runs sanity checks to ascertain whether a test has failed or passed. BuildTest supports several status and performance checks. For details, refer to the official documentation [1]. Finally, during the cleanup phase, the report file is updated with the test results, including metadata.

As presented by [4], at the heart of Testpilot is the Testpilotunit, enabling individual application testing through the job scheduler. It includes test definitions, a job submit engine incorporating progress tracking, and a verification engine. Complementing this core are two fundamental modes: Testpilotwing, a primary regression testing tool for updates or downtime, which receives special attention in this paper, and Testpilotpatrol, actively monitoring cluster health by periodically executing unit tests and reporting errors. Each test passes through a pipeline with a similar structure as Figure 2; after being read from the personal test profiles, where tests along with run options like number of tests and pass threshold are defined, the framework compiles the tests during the compatibility check together with the compile phase. This is done before submission to ensure the test script is compatible with the system components. Once done, the scheduler job is executed and continuously monitored with the output files to guarantee they are synced in case of an error or delay. After that, during the evaluation phase, the framework compares the output to reference outputs and standard errors to assess the test as failed or passed. Afterward, the framework automatically removes the temporary files created in the cleanup phase.

2.3 Test Implementation

Each testing framework employs unique programming languages for creating tests. Moreover, regression tests are handled differently across frameworks and present their own requirements to write them. Consequently, a broad rundown is given on how to implement a simple test in each framework. To illustrate, we will cover a simple test that checks if a Hello-World C-Program, as in Figure 3, produces the expected output. It is important to note that

```c
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

Figure 3: Hello World program written in C

```python
import reframe as rfm
import reframe.utility.sanity as sn

@rfm.simple_test
class HelloTest(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    sourcepath = 'hello.c'

    @sanity_function
    def assert_hello(self):
        return sn.assert_found(r'Hello, World\!', self.stdout)
```

Figure 4: Example hello world test script in ReFrame [11]

this paper is not intended as a tutorial on how to write tests in each framework. Instead, the aim is to provide a visual aid of the structure of tests, highlighting the requirements for writing tests. For a detailed breakdown of how to write complex performance tests, the reader is referred to the official documentation of each framework.

In ReFrame, a regression test is a Python class that derives from the `RegressionTest` base class provided by the framework. Variables in ReFrame are attributes that can be defined directly in the Python class body. One optional attribute provided by ReFrame that can be set is `build_system` which allows the user to set the build system for their regression tests inside each test [11]. If no build system is specified, ReFrame will automatically pick one. Along many other optional attributes, there are two attributes each test must always set. The first one is the `valid_systems` attribute. It is a list that defines systems by their name and, optionally, a specific partition a test can run on. In the given example (Figure 4), the test is defined to run everywhere, given `['*']`. The test can be defined to only run for the gpu partition of a system named "mysystem" through `valid_systems = ['mysystem:gpu']`. The second mandatory attribute is `valid_prog_environs`, similarly defining for which programming environments the test is valid. Systems and their partitions, in addition to the programming environment names, are configured in ReFrame's configuration file, which is further covered in Section 2.1. When inspecting the example script (Figure 4), it can be observed that the C-Program is nowhere directly compiled and executed. According to the official documentation of

```yaml
buildspecs:
    hello_test:
        executor: generic.local.bash
        type: script
        description: Test for Hello, World output
        run: |
            #!/bin/bash
            sourcepath="hello.c"

            module purge
            module load gcc
            gcc -o hello $sourcepath

            output=$(./hello)

            if grep -q "Hello, World!" <<< "$output"; then
                echo "Test passed"
                exit 0
            fi
            echo "Test failed"
            exit 1
```

Figure 5: Example buildspec file including one test `hello_test`

ReFrame [11], each test must define an executable or a source file to be compiled. The optional attribute `sourcepath` defines the source file to be compiled and automatically uses the produced executable to run the test.

As visible by the partition attribute, for example, writers of tests only interact with the system interaction mechanics via an API at the highest level. Therefore, regression tests become easier to write, and readability is enhanced, as explained by Karakasis et al. [9].

As mentioned in Section 2.2, BuildTest defines tests in a YAML file called buildspec. The buildspec file holds multiple tests. For each test, attributes to alter execution and help define complex tests are directly set inside the buildspec file. All attributes are not covered here but are explained in great detail in the official BuildTest documentation [1].

A possible simple Hello-World-Test can be seen in Figure 5. The test uses the script schema defined by `type: script`. The `executor` property defines the scheduler specified in the BuildTest configuration. In this case, the local executor with a name "bash" inside the BuildTest settings is used. The `description` field documents the test. The `run` property defines the script content, which can be a shell script (bash, csh) or a Python script.

As explained in [4], in Testpilot, each test has its own directory that holds a file for presubmission, the actual test file for submission, expected outputs and error files, and additional source codes and data files. Scripts for Testpilot are written in Perl and Shell. Figure 6, shows the `test.pre` file for presubmission and the `test.sub` file for submission of the "Hello, World"-Test. In this case, the `test.err` file defines standard errors, but it is empty. The `test.out` files define the expected outcome, and its content is thus "Hello, World!". Testpilot works without any workarounds for PBS schedulers only. PBS directives are used to configure job submission. In

```
#!/ bin / bash
module purge
module load gcc
gcc -o hello hello . c
```

(a) `test.pre`

```
# PBS -S / bin / bash
# PBS -l nodes =1
# PBS -l walltime =5:00
cd $PBS_O_WORKDIR
module purge
module load gcc
./ hello
```

(b) `test.sub`

Figure 6: Example hello world test scripts for compilation 6a and execution phase 6b in Testpilot [4]

this example, these directives are used:

- **PBS -S /bin/bash**: Specifies the shell to be used, in this case, Bash.
- **PBS -l nodes=1**: Requests one node for the job.
- **PBS -l walltime=5:00**: Requests a time limit of 5 minutes for the job.

Users can customize test execution by adding a `test.range` file, where reference values and deviations in percentage can be specified. Additionally, the importance of variables can be weighted.

In comparison, using YAML in BuildTest for test management enhances readability by separating configuration from the test script within the same file. YAML's compatibility with many programming languages [10] is advantageous, allowing the framework to cover modules or components in various languages. This is beneficial for integrating tests written in different languages. However, BuildTest may encounter limitations in extensive analysis due to its reliance on YAML [12]. While YAML simplifies test reading and definition as a data serialization format, it lacks the expressive power of programming languages like Python. Languages with rich libraries enable the creation of more complicated tests and features.

Another significant benefit of Python in ReFrame is inheritance. This feature allows users to write similar tests efficiently with minimal effort. In contrast, YAML, Perl, and Shell do not have inheritance support, potentially demanding users of BuildTest and Testpilot to redefine test parts multiple times.

Based on the given examples (Figure 4, 5, 6), it is evident that users of ReFrame do not need to make adjustments to the environment directly inside the tests for the test to run. This is configured inside the configuration file under `'environments'` (see Figure 1) and afterwards taken care of by the framework. In contrast, BuildTest and Testpilot require users to manually set up the environment for each test, leading to redundant efforts and an increased risk of unnecessary mistakes.

In ReFrame, a single test script can run on multiple systems with their partitions, schedulers, and with various compilers. To do so, compilers must be configured inside the programming environment inside ReFrame's configuration file (see Figure 1) with systems and their details. Afterwards, multiple specifications can be defined for a single test by referencing them inside each test. Similarly, BuildTest allows users to use a single test for multiple schedulers and compilers. In both cases, users can define multiple specifications inside tests using wildcard patterns. For example, defining `executor: 'generic.local.(bash|sh)'` inside a test will run it for the generic.local.bash and generic.local.sh scheduler [1]. To run a single test for various combinations of compilers and schedulers in Testpilot, new test scripts for each combination must be redefined.

All of the three frameworks provide tests and benchmarks out-of-the-box. ReFrame has its own test library, a collection of popular benchmarks to employ or build upon. As described by [11], among these highly configurable benchmarks and tests are some to test the efficiency of GPUs and ones to check if commonly employed frameworks and packages are functioning as intended. They do not require any configuration before executable. In the case of BuildTest, NERSC openly shares all tests used, accessible at [5], and the HPC community contributes further BuildTest tests. Siddiqui et al. [13], including founders of BuildTest, encourage users to share their test repositories with the HPC community. Naturally, users potentially need to adjust shared test attributes before they are applicable on their systems. As for Testpilot, it integrates numerous tests, as proposed by [4]. Users have the option to overwrite these defaults with custom tests if desired.

This general approach leverages the advantages of reusing proven tests, eliminating the need to develop them from scratch and ensuring their reliability. Moreover, these built-in tests function as practical templates, simplifying the process for newcomers to create tests within these frameworks.

2.4 Usage

Not only differentiate BuildTest, Testpilot, and Reframe when it comes to testing implementation, but they all have their own way of test execution and making use of them. A framework should provide a high degree of user-friendliness, aiming to manage as much of the test process as possible. This streamlines the testing procedure, reduces the learning curve, and enables a larger audience to adopt its capabilities.

As introduced in Section 2.2, test profiles can be

```
#
# |Mode|Test Name|Pass Threshold|Total Tests|
#
# PBS Tests
C pbs_cpusets 9 10
C pbs_16cores_on_1node 9 10
C pbs_20cores_on_1node 9 10
C pbs_24cores_on_1node 9 10
C pbs_48cores_on_1node 9 10
#
# Compiler Tests
C intel 9 10
C pgi 9 10
#
# Parallel Library Tests
C mpich2 9 10
C mvapich2 9 10
C openmpi 8 10
C openmp 9 10
#
# Specific Application Tests
C abaqus 9 10
C gaussian 9 10
C maple 9 10
C matlab 9 10
C octave 9 10
C r 9 10
```

Figure 7: Example Testpilotwing profile file. The first column defines the execution policy, the second specifies the test name, the third specifies how many tests need to succeed to be considered passed, and the fourth sets the number of runs. Non-existing tests on the underlying system are simply skipped. Source: [4]

defined in Testpilot. These profiles allow users to specify the specific tests to be executed, the number of runs allotted for each test, the success criteria for individual tests, and the option to choose between concurrent or sequential test execution. All of these configurations are conveniently managed through a straightforward text file. An illustrative example of a test profile is presented in Figure 7. As previously mentioned in Section 2.2, each test can have a `.range` file defined, where variable references can be specified, and tolerances, as well as weights, are given. Testpilotwing delivers continuous feedback to users by providing a summary of the progress of each test. This enables users to make informed decisions during test execution and empowers them to intervene in the event of problems [4]. Individual tests can be run with the `testpilotunit` command and express the test to run after. The regression test mechanism Testpilotwing is executed when issuing `testpilotwing`, producing a continuous output like Figure 8. It is not required to state the profile to run. If not specified, the profile "`default`" is selected for execution. With `-o`, additional scheduler options can be passed to the job scheduler, `-l` lists all tests, `-k` preserves temporary raw output for debugging purposes, and `-n` displays hostnames of nodes which fail tests of the profile.

Similar to the execution options in Testpilotwings profile and range files, ReFrame offers a rich set of command-line options for fine-tuning its execution. Users can customize the execution order based on various attributes, modify the execution policy, set test duration, specify the number of runs, filter by name or tag, and more. For further details, refer to the documentation [11]. To run ReFrame, `-c` is used to search inside the given path for a directory or single test file. Appending `-r` executes the tests at this path. After execution, ReFrame provides a console output stating the test's success. An example output can be observed in Figure 9.

To run tests with BuildTest, the `buildtest build` command is used. BuildTest will produce some console output alongside listed build information, as seen in Figure 10. Also BuildTest provides command-line options to tweak test execution. The most essential one is `-b`, which is used to specify a path to the build spec or directory of build specs that are supposed to be executed. Similarly, `-x` can exclude specified buildspecs. Moreover, very similar to ReFrame, a timeout for all tests of the buildspec can be specified, tests can be run by tag, tags can be excluded, they can be run by executor, name, be filtered, and more. For details and other command-line options, the reader is referred to the documentation [1]. Further adjustments, like the number of tests that can run concurrently, can be specified inside the BuildTest configuration file.

Both ReFrame [9] and BuildTest [1] incorporate a test grouping feature. By assigning tags to tests using the `tags` attribute in both, users can form groups of tests. Testpilot indirectly employs a similar feature where multiple test profiles can be created in which various tests can be specified to be executed together during a run [4]. The benefit of a grouping feature lies in its efficient ability to invoke specific groups of tests. Consequently, sets of relevant tests can be executed instead of requiring to run the entire test suite. A further benefit lies in better maintenance, where users can clearly navigate and update groups of tests that may need to be modified without overseeing certain test cases or possibly tweaking the wrong ones.

To capture the performance and output of tests, ReFrame logs each test case in detail. The work of [9] describes that ReFrame records their values and references. The authors highlight that by default, the framework creates log files for each test, system, and partition, appending data to them each time a test is run. These logs include timestamps, test descriptions, job IDs, and performance details. Additionally, ReFrame can be configured to log for each defined performance variable, according to [11]. Furthermore, it is mentioned that inheriting the Python logging framework enriches ReFrame with a strong set of base functionality. Contrary to ReFrame, the main goal of Testpilot, as presented in [4], is to test complicated systems extensively,

yet reducing this to simple PASS/FAIL outcomes. Additionally, output files for each test are stored which are compared with defined standard outputs for test evaluation. While this simple PASS/FAIL outcome allows even inexperienced users to validate a cluster after updates or incidents, it does not help directly investigate the cause of those fails. Nonetheless, when a particular test requires a more complex analysis, this analysis can be integrated into the test script directly [4]. This, however, comes with a tradeoff, requiring additional workload for the users. Testpilot continuous monitoring mode Testpilotpatrol logs successful test results in syslog. The official BuildTest documentation [1] unveils that the BuildTest framework stores their test results in a JSON file for post-processing. These results can be displayed as a table in the console via `buildtest report` or `buildtest inspect`. The ability to easily format and query those results is further mentioned.

Furthermore, a drawback of ReFrame's in-depth logging is that creating log files for each test, system, and partition demands a substantial amount of files to be managed. This may pose challenges regarding data organization and result retrieval, especially when the framework is employed in isolation. Potential solutions to address this disadvantage are introduced in Section 3.3.1. While tests are kept in mainly one file for ReFrame and BuildTest, Testpilot deviates from the other frameworks by each test having its own directory, which encompasses a presubmit-script that manages compilation, the actual test file, a file for expected outputs, one for expected errors and additional source codes and data files, as stated by Colby et al. [4]. One benefit of this approach is that complex tests can be better understood and thus easier maintained, as the directory structure is a form of documentation. A potential drawback is the increased complexity of managing tests, especially for simple ones. Managing multiple tests can increase additional overhead in terms of file operations.

2.5 Performance and Scalability

Many tests are required to effectively test if a system fulfills all desired roles, ranging from unit tests of small functions on specific components to integrated simulations on the whole HPC cluster. Naturally, a large quantity of tests take time to implement and run. Preferably, tests should be executed efficiently to minimize these times. Furthermore, It is not uncommon for HPC systems to face intensive workloads. Therefore, it is mandatory for the test framework to operate efficiently under these conditions. For that reason, a good framework should

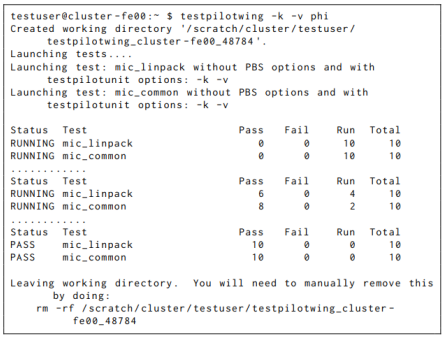

Figure 8: Example output of Testwing. Source: [4]

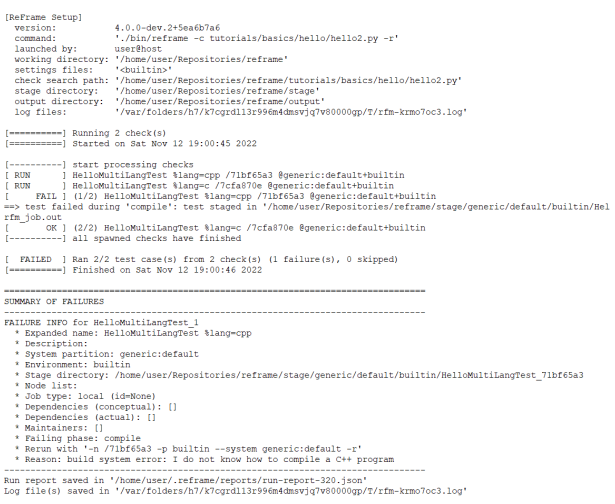

Figure 9: Example output in ReFrame. Source: [11]

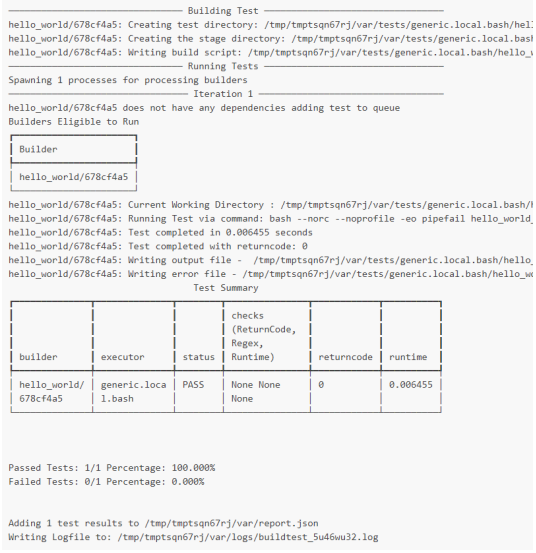

Figure 10: Example output in BuildTest found in [1]. This image excludes printed buildtest summary, discovered buildspces, and printed parsing

provide the ability to create test cases with sufficient coverage and, as the volume of test cases increases, function as a scalable tool that does not slow down execution under heavy load to prevent delayed development.

ReFrame defaults to processing test cases in parallel, but users have the flexibility to switch to sequential execution if desired [11]. While Testpilot does not fully support parallel test execution according to [12], it offers the option to run tests synchronously for manual testing or asynchronously for automated testing [4]. Within BuildTest's configuration file, users can specify the number of jobs that run concurrently, with the default being all jobs running in parallel [1]. In general, concurrent tests accelerate the overall testing process, help identify performance bottlenecks, and assess the scalability of the system, providing insights into how well the system scales under varying workloads.

Resulting in a more efficient development workflow, ReFrame automatically handles and resolves conflicts that emerge when changing the programming environment of a test. According to [9], ReFrame also handles conflicts between required modules and errors when writing tests. This eliminates the need for manual intervention and debugging.

Compared to ReFrame, Colby et al. [4] states one clear downside, which is that concurrent testing in Testpilot is vulnerable to resource conflicts, which may lead to false negatives. BuildTest's official documentation [1] does not mention the existence of a solution for resource conflicts, so, similar to Testpilot, false negatives can be expected when testing concurrently. That means that some tests need to be run manually, which can become very time-consuming and burdensome, especially at a large scale. However, an exception is raised if a failure occurs during the build, and the failed buildspec is ignored.

Automation is a feature that increases consistency and provides an efficient testing process. Moreover, it enables easy historical analysis, meaning performance changes over time can be identified. Among the three frameworks, only Testpilot, with its continuous monitoring abilities, has an automated testing function. Referring to the research conducted in [4], the Testpilotpatrol mode is described to execute a batch script every 30 minutes, distributing tests asynchronously to the job scheduler in a way where the job queues do not become fully clogged. Furthermore, Testpilot accounts for various errors, including scheduler errors and job crashes, which may extend the overall test duration. If a test does not run by its next submission interval, the previous test is said to be canceled, and administrators are alerted. The results are logged in syslog, while errors are communicated via email. In contrast, both ReFrame and BuildTest lack native automated testing features. However, they offer seamless integration with continuous integration, delivery, and deployment (CICD) tools for test automation, a topic discussed in Section 3.

2.6 Extendibility

Extendibility defines the framework's capacity to be altered by users, transforming it into a dynamic and scalable tool. This promotes versatility and ensures the framework's longevity, as users can expand its features to meet evolving needs.

As highlighted in Section 2.2, ReFrame guides every test through a structured pipeline of various stages. If desired, the user can intervene between those stages and customize their behavior, explained by Karakasis et al. [9].

Outlined in [9], in addition to ReFrame's backend API for writing regression tests, the framework defines several internal APIs that interface job schedulers, parallel job launchers, build systems, and module systems. This enables the expansion of their functionality by implementing diverse, independent backends that do not require change to the core infrastructure of the framework.

As mentioned in Section 2.2, BuildTest is validated using JSON schemas. These schemas can be adjusted or custom-implemented. Consequently, they can be extended to validate additional features, expanding test functionalities.

As Buildtest has an integrated CDash functionality, which allows the framework to easily push test logs to a CDash server for log data management and further data analysis. This feature is further discussed in Section 3.3.1.

Section 2.1 mentioned Testpilots restriction to PBS schedulers. Colby et al. [4] explains that support for other schedulers can be easily added by translating scheduler-specific definitions inside tests to other schedulers. However, this feature is not directly provided by the framework and requires manual setup.

One advantageous feature of ReFrame is the ability to define any build system for utilization. Users can set up and configure any build system directly within each test using provided attributes, as detailed in [11]. BuildTest's documentation [1] does not name the specific build system the framework supports. However, its CDash integration suggests the primary utilization of CMake. It is unclear which build systems Testpilot supports. Nevertheless, it is evident that Testpilot only supports PBS schedulers and if the user wants to utilize different

compilers, users have to define the according shell commands inside each test manually.

2.7 Summary

This paper compares three modern testing frameworks for regression testing in HPC systems, aiming to find an optimal test framework. This comparative analysis revealed trends and clear differences between ReFrame, Testpilot, and BuildTest. Table 1 summarizes key aspects of that comparison.

As explained in Section 2.2, though exhibiting a very similar regression test pipeline, all frameworks use different approaches for defining tests. Tests in ReFrame are single Python classes and the system is configured in a configuration file. BuildTest defines tests inside a YAML file together with system specifications, and Testpilot defines a system inside test scripts directly and distributes tests into multiple files for different test phases.

The comparison unveiled that all three frameworks provide the ability to run tests in parallel. This is an essential feature for any scalable and efficient testing framework. While in Section 2.5 the ability to execute tests concurrently in BuildTest and Testpilot was stated, it was later in that section revealed that tests in those frameworks can be vulnerable to resource conflicts. This can require some tests to be run manually in those frameworks, which is a clear downside compared to ReFrame.

A disadvantage of Testpilot is the absence of a tagging feature. In ReFrame and BuildTest, tests can be tagged, which allows specific test groups to be directly invoked by their tag. An example of why this can be beneficial is: suppose that a system has compatibility tests of software for different hardware components, like GPU, CPU, and memory. When introducing a new GPU, users might want to skip the tests for CPU and memory, speeding up the testing process and quickly assessing only the relevant change. Section 2.5 additionally mentions that through tagging, groups of tests can be updated easily without oversight.

The only of the three frameworks with a test automation feature is Testpilot with its Testpilot-patrol mode, making testing vastly more efficient and consistent.

Testpilot and BuildTest excel in user-friendliness and maintainability, which is attributed to their clear test structure. ReFrame's class structure allows all components to reside inside a single file, which can be more difficult to maintain. However, naturally, test files can be split as needed. This flexibility can prove advantageous because using a single file for each test can ease management, particularly when exposed to many tests and files. Moreover, as done in Testpilot, splitting files can add overhead due to file operations.

Section 2.4 introduces ReFrames intensive logging. This can become overwhelming, but it provides in-depth details, and its logging system can be customized. This further proves the tradeoff between user-friendliness and advanced capabilities.

ReFrame is highly adjustable by allowing to intervene and tweak its pipeline and define independent backends using provided internal APIs. BuildTest JSON schemas can be adjusted by developers expanding test functionality.

While ReFrame may be viewed as more challenging to set up and use, it compensates for this drawback by offering seamless configuration across diverse environments. Additionally, this approach minimizes changes required for existing tests to run on different systems. Other frameworks, though relatively more straightforward to set up due to their reduced system information requirements, exhibit limitations in supporting various systems.

In conclusion, ReFrame generally proves more capable than the other frameworks. Its key characteristics are a high degree of customization, advanced scalability, and detailed logging. As a tradeoff, ReFrame can become overwhelming to maintain extensive tests, and it exhibits a steeper learning curve, making it less user-friendly when compared to the alternative frameworks. Completely missing in ReFrame and BuildTest is test automation.

Ultimately, the optimal choice among ReFrame, Testpilot, and BuildTest depends on the specific needs and preferences of the testing environments. All frameworks are fairly similar in their base functions but excel in certain aspects.

3 Synergistic Approach

Previously, it was concluded that none of the frameworks addressed in this paper can be considered ideal. This section covers what an optimal regression test framework looks like. Furthermore, the combination of different frameworks and tools is introduced as a valid approach towards achieving the optimum.

3.1 The Ideal Regression Framework for HPC systems

By looking at the modern frameworks of this paper, features, and attributes, every framework should ideally cover are unveiled. Additionally, Colby et al. [4] expresses specific design goals aimed at optimizing ease of use and enhancing the generalizability of

	ReFrame	BuildTest	Testpilot
Portability 2.1			
Operating Systems	any	Linux and MacOS	any
Scheduler	any	IBM Spectrum LSF, Slurm, PBS, Cobalt	PBS
Build Systems	any	primarily CMake	
Other Systems/Tools		Intel, GNU, OpenMPI, MPICH	
Configuration	default adjustable config file for partitions and environments	adjustable default configuration + inside tests	directly inside tests
Architecture 2.2			
Test Structure	Python Class	buildspec YAML file	test directories referenced in profiles
Implementation 2.3			
Script Language	Python	YAML + Shell or Python	Perl + Shell
Single Test Multiple Specifications	✓	✓	✗
Provide Tests	✓	✓	✓
Usage 2.4			
Grouping	✓	✓	indirectly via profiles
Logging	JSON file for run + detailed log file for each test, system, partition	single JSON Result File	Reduced PASS/FAIL output + output files for each test; Tppatrol.: syslog
Performance and Scalability 2.5			
Concurrency	✓	(✓)	(✓)
Automation	✗	✗	✓
Error Handling	test exceptions, conflicts between modules, environments	environment conflicts	
Extendibility 2.6			
Author Rating	++	+	-

Table 1: Comparison summary of features in ReFrame, BuildTest, and Testpilot. Empty cells mean 'unknown'.

Testpilot. This subsection delves into those features and attributes.

One design goal mentioned is that new tests should not require changing the framework's core components. This is a crucial key requirement for any regression test framework. Without it, deploying a framework primarily for regression testing, yet potentially introducing flaws when incorporating new elements, is paradoxical. Additionally, this principle eases maintenance, as already working functionalities are left untouched after a change. Furthermore, a system benefits in terms of scalability, as it allows the framework to grow without becoming difficult to manage.

Also supporting systems to scale well is a high degree of customization and the ability to extend the framework in many aspects beyond its base functionalities. Incorporating these abilities preserves an agile framework, allowing it to meet changes and advancements as the system grows and remain up to date with the latest developments in the HPC domain.

The authors of [4] state: "The verification framework must be generic such that all test outputs may be verified with a single script", meaning a test framework should be adaptable and applicable to a wide range of test outputs. No matter the output and its format, the framework should be able to verify the results to effectively scale as the number and complexity of tests grow and provide consistent results.

The ability to capture unusual testing scenarios is mentioned in [4]. Unusual testing scenarios include running tests against only a subset of components or nodes or setting test parameters. Adjusting parameters helps discover bottlenecks and address edge cases. Testing only for a subset of components or nodes can be beneficial in terms of efficiency.

[9] outlines ReFrames attribute not to require developers to learn complex libraries or custom syntaxes to write advanced tests, a characteristic shared with the other presented frameworks. This feature contributes to keeping the learning curve low.

A trade-off expressed in Section 2.4 is ease-of-use versus detail regarding logging. In-depth logging, like in ReFrame, can result in information overload, making critical details harder to identify, but it enhances debug efficiency by helping diagnose issues quicker. However, less detailed outputs are potentially easier to understand and work with while providing less help to discover root problems.

A well-defined structure and documentation generally ease maintainability. This structure can be accomplished by splitting different test aspects into distinct files. While this improves navigation and is easier to maintain because large single files can become overwhelming, it can be less efficient as explained in 2.5. In contrast to the latter aspect, managing fewer files is easier, and poorly organized large quantities of files can become overwhelming as well.

An automation feature in the form of continuous testing can achieve consistent and efficient testing without requiring human intervention. Note that this has to be done in a controlled way that does not overload the cluster or influence its performance negatively.

A feature essential for any test framework is the ability to test in parallel. This significantly increases the overall efficiency of the testing process. Especially when exposed to many tests, testing time can be kept minimal.

As concluded in Section 2.7, resource conflicts need to be handled by the framework to guarantee flawless concurrency. Without such a feature, false negatives can occur, potentially resulting in needless debugging or manual reruns of tests, consequently wasting a lot of time and labor.

Optimally, a framework should be universally applicable for any system, including its scheduler, compiler, and operating system.

On top of that, when setting up the framework or transitioning to a different system, configuration requirements should be minimal. To ensure test maintainability, porting to another system should not demand changes to existing tests.

Having outlined a blueprint that could guide to an ideal testing framework, a question may arise: "If these goals are so evident, why does framework X not employ feature Y?". It is easy to notice that many features that are beneficial for one attribute may conflict with another. Conflicts arise in areas such as ease-of-use versus detail in logging, trade-offs in file structures, the potential drawbacks of uncontrolled automation versus its controlled counterpart, the consequences of concurrency when conflicts are unaddressed, and the balance between minimal configuration and universal applicability. The rest of this Section 3 explains strategies to deal with these challenges and create an ideal testing environment.

3.2 Framework Combinations

One approach to maximize attributes like efficiency and scalability is applying multiple frameworks simultaneously. This allows the system to make up for shortcomings in certain frameworks by leveraging the strengths of others. Notably, while this strategy provides many benefits, it introduces new drawbacks.

In particular, integrating Testpilot with ReFrame, for example, can have significant advantages. Test-

pilot can continuously check a system for bugs after changes, while ReFrame, with its ability to write extensive performance tests within Python, measures system efficiency after updates. This addresses ReFrame's lack of automation and leverages its extensive execution options and test features along with Python's libraries and functions to define complex performance tests. The results of these performance tests are logged in detail with ReFrame's extensive logging for further analysis. Concurrently, simple checks can be reduced into uncomplicated outputs, as intended by Testpilot.

Additional advantages of integrating frameworks are that it can encompass expanded test coverage, thanks to different test specializations of employed frameworks, it can enhance reliability by validating results across multiple frameworks, and it has the potential to lower the entry level for new developers. They can comfortably utilize user-friendly frameworks without completely resigning to the strengths of more advanced frameworks, perceived as more challenging.

Utilizing multiple frameworks concurrently brings forth not only benefits but unveils some disadvantages. A consequence of employing multiple frameworks is increased complexity in the testing workflow. Coordinating the execution of multiple frameworks poses a challenge, potentially leading to increased overhead in time and effort. Moreover, the requirements of different frameworks can be conflicting or incompatible. Furthermore, most frameworks introduce their own test syntax, resulting in the inability to reuse the same tests within other frameworks. This can demand tests to be written in multiple languages and frameworks. As the number of frameworks employed increases, managing files becomes more difficult, with each framework organizing files uniquely. Consequently, maintenance and navigation can become more challenging.

3.3 Supplementary Tools

As mentioned earlier, combining multiple testing frameworks makes up for the weaknesses of the individual frameworks but introduces new challenges. The integration of supplementary tools emerges as a more targeted approach against specific weaknesses while introducing significantly fewer challenges in terms of configuration and management.

This section provides an overview of how the drawbacks introduced through trade-offs identified in our comparison (Section 2), in particular, logging and automation, can be mitigated by integrating distinct tools with test frameworks.

In the following, tools and their requirements will be named. However, this paper does not delve into the specifics of configuring these tools or explaining how they operate. Readers are referred to the provided resources next to their introduction for detailed information on these matters.

3.3.1 Logging

As explained in Section 2.4, ReFrame generates detailed log files for each test, their performance variables, system details, and partition information. Through comparison with frameworks that prioritize user-friendly, simplified logging, such as Testpilot [4], it was revealed that frameworks with detailed logging can be overwhelming due to the amount of files and detailed information. Simpler log files enable inexperienced users to analyze test results but offer less support for debugging and identifying root causes.

In the case of ReFrame, [9] mentions that thanks to the Python logging framework, data can be sent to Syslog or Graylog servers. Syslog and Graylog address two downsides of ReFrame: challenging management of extensive log data and files, together with potentially resulting overwhelm. In particular, Graylog [7] enables efficient management and offers powerful query capabilities to retrieve data. Furthermore, Graylog supports data visualization and summaries via configurable dashboards, vastly simplifying data analysis in ReFrame. To set up Graylog for ReFrame, the user has to define the `graylog` header in ReFrame's logging configuration file. In case log processing using handlers that send test records to log servers (including Graylog) break, ReFrame has to be configured to log for each performance variable by setting the `perflog_compat` configuration parameter. Consequently, all relevant attributes, encompassing performance thresholds, references, measurement units, names, and actual values, are logged. While not explicitly documented for Testpilot and Buildtest, Graylog can be utilized by those. The general steps to use Graylog for any test framework are to install Graylog on a dedicated server, define input types (e.g., Syslog, GELF, Beats, AWS) within Graylog, and send logs formatted in the specified type to the server. It is crucial to ensure that log files are parsed into the correct format if not handled by the framework. Testpilots Testpilotpartrol monitoring provides logs in fitting syslog format, making Graylog easily applicable to this framework too.

Similarly, Siddiqui et al. [13] explains using the test server CDash [2] to display results. In addition to the previously recognized benefit of quickly identifying test results, a boost in users' confidence is stated in [13] by displaying results publicly accessible. As expressed in BuildTests documentation [1],

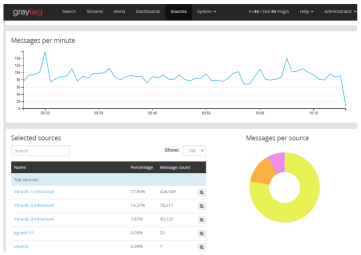

Figure 11: General data visualization example inside Graylog interface. Source: [6]

running `buildtest cdash upload` and appending a buildname after will upload test results to the cdash server specified in the configuration file. After running this command, a URL to a web interface is returned where outputs can be analyzed (see example **??**). Inside the web interface, results can be analyzed by viewing test histories and inspecting further test details. An example is depicted in Figure 12

Thanks to the integrated CDash functionality within BuildTest, setting up CDash in BuildTest becomes straightforward. The only requirements are the URL of the CDash server, a project name for the server, a site name (which should be the system name, as specified in [1]), and a freely selectable build name. While configuring CDash for other frameworks is possible, it can pose challenges, especially when not using CMake for building and CTest for testing. CDash is tightly coupled with CMake and CTest. Running CDash without these components requires extensive manual configurations, as detailed in [3].

It is important to note that many comparable tools exist for comprehensive logging, visualization of test data, and post-processing analysis. This paper, however, does not conduct a detailed comparison of these tools. Instead, readers are encouraged to research additional options independently. Among notable alternatives to the tools mentioned earlier include Ganglia and Splunk.

3.3.2 Automation

The comparison concluded that automation, especially in the form of continuous test execution, offers significant advantages. Among the three frameworks of this paper, Testpilot stands out as the only framework with an automated testing feature. This capability can monitor cluster health by running tests and benchmarks at set intervals without manual involvement. This raises Testpilot's test consistency since tests at specified times and regular intervals create reliable and comparable results.

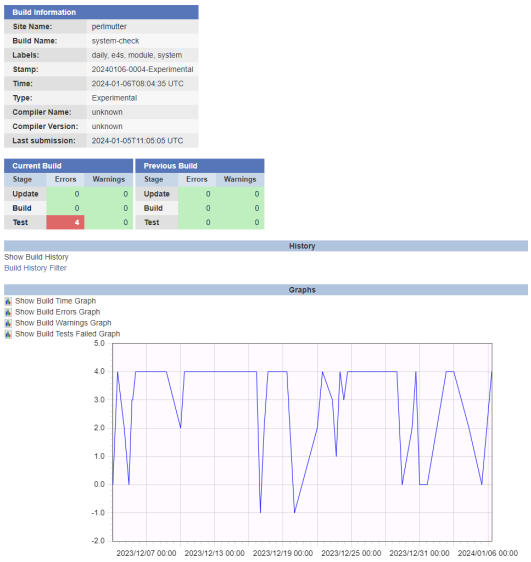

Figure 12: Build information and failed test graph for daily system-check at NERSC from the 6th January 2024 NERSC CDash project `https://my.cdash.org/index.php?project=buildtest-nersc&date=2024-01-06` `https://my.cdash.org/build/2467647`

Furthermore, users and developers can minimize downtime and focus on tasks other than executing tests and monitoring performance.

To make up for this deficiency in the other proposed frameworks, CICD tools can be integrated, leveraging their strengths of automated testing. Alternatively, tests can be run periodically using the cron job scheduler in Linux systems, for example. The benefit of modern CICD tools is the common integration with version control systems, allowing tests to run automatically in the case of changes, which is particularly interesting for testing regression. Moreover, common CI tools are designed to handle complex pipelines, respecting dependencies and supporting parallelism, which can be challenging to employ without these tools.

While not directly employing an automation feature, ReFrame can be effortlessly integrated with popular CICD frameworks like Jenkins or Gitlab. These enable the framework to be employed for ongoing automated tests and even for the continuous deployment of changes to the system [9].

According to [11], for GitLab CI integration, ReFrame provides the functionality to generate a child pipeline, encompassing each ReFrame test while respecting their dependencies. It will generate a CI job for each test to use concurrency. A runner

```yaml
stages:
  - generate
  - test

generate-pipeline:
  stage: generate
  script:
    - reframe --ci-generate=${CI_PROJECT_DIR}/pipeline.yml
        -c ${CI_PROJECT_DIR}/path/to/tests
  artifacts:
    paths:
      - ${CI_PROJECT_DIR}/pipeline.yml

test-jobs:
  stage: test
  trigger:
    include:
      - artifact: pipeline.yml
        job: generate-pipeline
    strategy: depend
```

Figure 13: `.gitlab-ci.yml` for generation of all ReFrame tests defined at `${CI_PROJECT_DIR}/path/to/tests` to CI jobs in GitLab. Credit: [11]

has to be set up, as previously explained. Afterwards, a `.gitlab-ci.yml` file, as illustrated in 13, will generate CI jobs for each test. The "generate" stage will cause ReFrame to handle the specified tests as usual, but instead of executing, each test is defined as a CI job in a child pipeline. The second phase called "test", uses the generated pipeline as an artifact, retrieving the newly defined CI jobs.

BuildTest can be integrated with a variety of CICD tools. However, no pipeline generation as in ReFrame exists in BuildTest. As stated in [13], multiple pipelines must be created to consider queue policies and dependencies.

Alternatively, Jenkins [8], an open-source automation platform widely used for building and deploying software, can be utilized instead. Jenkins, known for its extensive extensibility through plugins, provides an automation pipeline similar to GitLab. Configuration of Jenkins pipeline jobs involves a specially formatted Groovy script (depicted in Figure 14) that defines each step of the build process and where the step should be executed. While slightly limited in comparison, new jobs can be defined directly within the Jenkins GUI. Inside the GUI, jobs can be specified to run periodically. Similarly, periodic execution can be defined within a scripted Jenkinsfile using the `trigger` directive and a cron-style string to specify a regular interval for triggering the pipeline. For an example, see Figure 15.

Other applicable continuous integration tools are CircleCI, TravisCI, and many more.

4 Conclusions

This paper presented an overview of modern test frameworks for regression testing in HPC systems. Regression testing is pivotal in HPC environments, as systems can become error-prone even with minor

```
pipeline {
    agent any
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') {
            steps {
                sh 'make'
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml'
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
```

Figure 14: Example of a Jenkinsfile Groovy script to define CI job pipeline [8]

```
pipeline {
    agent any
    triggers {
        cron('H */4 * * 1-5')
    }
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

Figure 15: Example of 'Hello World' printed every 4 hours on weekdays [8]

changes, given their complexity, resource intensity, and commonly distributed nature. By comparing the features and design goals of ReFrame, BuildTest, and Testpilot, the strengths and weaknesses of each framework have been identified and summarized. In consequence, an ideal testing framework was unveiled in theory. However, such an ideal framework does not exist due to contrasting attributes among framework features.

The primary goals identified for an optimal testing framework include achieving efficient scalability without performance degradation as the scale of both the system's hardware and software expands. Additionally, the framework should provide great resource management through parallel execution while respecting dependencies, provide comprehensive debugging details, require minimal to no configuration for deployment on any system, have easy extensibility without limitations to accommodate evolving needs, and offer user-friendly interfaces for defining and executing tests. The framework should also be easily manageable and maintainable.

In reality, many features that enhance one attribute decrease another. Understanding these conflicts and trade-offs within individual frameworks, it was concluded that combining multiple frameworks can be beneficial. While allowing them to leverage each other's strengths, it can introduce new chal-

lenges, including higher maintenance complexity, management issues, and an overload of configurations.

To address the identified drawbacks more directly, additional tools such as CDash and Graylog can be used for functionalities that are not directly integrated into the framework. These tools contribute to better organization, navigation, and comprehension of test results, mitigating challenges related to file structure and result interpretation. Additionally, the incorporation of Jenkins and GitLab was proposed to address the absence of automated test execution in BuildTest and ReFrame, resulting in a more consistent testing process with comparable results and minimizing human intervention.

In conclusion, none of the addressed frameworks can universally be considered "ideal". Each framework excels under different circumstances depending on various aspects, encompassing the use case, system specifications, and more. An optimal testing framework can not exist in practice due to conflicting attributes. Utilizing additional tools can minimize drawbacks and solve trade-offs when employing frameworks.

5 Future Research

Future work involves extending the comparison of regression testing frameworks to include additional frameworks such as Pavilion2, the Automated Testing System, JUBE, JACE, DART, HPCSWTEST, and OpenHPC. This expansion can help identify approaches for developing an optimal regression testing process. Moreover, well-working functionalities and features can be adapted by other frameworks. Furthermore, combining frameworks in practice deserves detailed exploration to find synergies among them and potentially advance towards ideal regression testing. While this paper compared three frameworks in theory, analyzing by employing each of these frameworks on the same specific HPC systems will enable the direct evaluation of attributes like performance, scalability, and portability.

References

[1] *BuildTest Documentation version 1.7*. 2023. URL: https://buildtest.readthedocs.io/en/v1.7/.

[2] *CDash*. URL: https://www.cdash.org/.

[3] *CDash Documentation*. URL: https://public.kitware.com/Wiki/CDash.

[4] Kevin Colby et al. "Testpilot: A Flexible Framework for User-Centric Testing of HPC Clusters". In: *Proceedings of the Fourth International Workshop on HPC User Support Tools*. HUST'17. Denver, CO, USA: Association for Computing Machinery, 2017. ISBN: 9781450351300. DOI: 10.1145/3152493.3152555. URL: https://doi.org/10.1145/3152493.3152555.

[5] BuildTest HPC community. *NERSC BuildTest tests for Cori and Perlmutter*. https://github.com/buildtesters/buildtest-nersc. 2023.

[6] GÜL Emre and Ercan Nurcan YILMAZ. "LOG MANAGEMENT WITH OPEN SOURCE TOOLS". In: ().

[7] *Graylog Documentation*.

[8] *Jenkins*. URL: https://www.jenkins.io/.

[9] Vasileios Karakasis et al. "Enabling Continuous Testing of HPC Systems Using ReFrame". In: Mar. 2020, pp. 49–68. ISBN: 978-3-030-44727-4. DOI: 10.1007/978-3-030-44728-1_3.

[10] Aditya Kavalur and Shahzeb Siddiqui. *HPC System and Software Testing via Buildtest*. 2021 ECP Annual Meeting. Apr. 2021. URL: https://www.exascaleproject.org/event/buildtest/.

[11] *ReFrame Documentation 4.4.1*. 2022. URL: https://reframe-hpc.readthedocs.io/en/v4.4.1/.

[12] Shahzeb Siddiqui. "Buildtest: A Software Testing Framework with Module Operations for HPC Systems". In: Mar. 2020, pp. 3–27. ISBN: 978-3-030-44727-4. DOI: 10.1007/978-3-030-44728-1_1.

[13] Shahzeb Siddiqui et al. "An Automated Approach to Continuous Acceptance Testing of HPC Systems at NERSC". In: Dallas, Texas USA: SIGHPC Systems Professionals Workshop, 2022. URL: https://github.com/HPCSYSPROS/Workshop22/blob/main/An_Automated_Approach_to_Continuous_Acceptance_Testing_of_HPC_Systems_at_NERSC/ws_hpcsysp105s2-file1.pdf.