Test Selection for Unified Regression Testing

Shuai Wang, Xinyu Lian, Darko Marinov, Tianyin Xu
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{swang516, lian7, marinov, tyxu}@illinois.edu

***Abstract*—Today's software failures have two dominating root causes: code bugs and misconfigurations. To combat failure-inducing software changes, unified regression testing (URT) is needed to synergistically test the changed code *and* all changed production configurations (configs) for deployment reliability. However, URT could incur high cost to run a large number of tests under multiple configs. Regression test selection (RTS) can reduce regression testing cost. Unfortunately, no existing RTS technique reasons about code and config changes collectively.**

We introduce Unified Regression Testing Selection (uRTS) to effectively reduce the cost of URT. uRTS supports project changes on 1) code only, 2) configs only, and 3) both code and configs. It selects regular tests and configuration tests with a unified selection algorithm. The uRTS algorithm analyzes code and config dependencies of each test across runs and across configs. uRTS provides the same safety guarantee as the state-of-the-art RTS while selecting fewer tests and, more importantly, reducing the end-to-end testing time.

We implemented uRTS on top of Ekstazi (a RTS tool for code changes) and Ctest (a config testing framework). We evaluate uRTS on hundreds of code revisions and dozens of configs of five large projects. The results show that uRTS reduces the end-to-end testing time, on average, by 3.64X compared to executing all tests and 1.87X compared to a competitive reference solution that directly extends RTS for URT.

I. Introduction

Today's software failures have two dominating root causes: faults in program code (i.e., bugs) and errors in configuration (config) files (i.e., misconfigurations) [1]–[5]. Many software projects include some default config together with the code in the project repository. Modern continuous integration and deployment (CI/CD) [6]–[9] aims to quickly check and release project changes. To combat bugs and misconfigs introduced through project changes, modern CI/CD environments widely use regression testing. Regression testing checks that project changes do not break previously working functionality. Traditional regression testing is mainly applied to (potentially changed) code under the (potentially changed) *default* config [10]–[12].

However, the deployed software uses *production* configs that typically differ from the default config. One limitation of the traditional regression testing is that the code changes are not tested under the production configs; likewise, changes of production configs are not tested with the code. Consequently, many code changes pass the regression tests under the default config but lead to failures in production. In fact, companies such as Google and Meta report misconfigs [1]–[4], [13], [14] as the main cause of failures in production systems, even more frequent than code bugs. While large software organizations have started to treat configs as important as code, e.g., using version control for configs and reviewing config changes manually, much more remains to be done.

Recent research has proposed *configuration testing* to detect erroneous changes to production configs [15]–[18]. A config test is a *parameterized* test [19], [20] with input parameters being config parameters; it runs by instantiating the input parameters with the values from the production configs (§II-A). The key idea is to connect config changes to tests, so that config changes can be tested in the context of code affected by the changes. Config testing can reason about the program behavior under production configs and detect sophisticated misconfigs that are missed by rule-based validation [21]–[25] or data-driven approaches [4], [26]–[31]. However, a major limitation of prior work is that config testing addresses only config changes of the *same* code version—it assumes that the code does *not* change and has only one production config (or if more, each is tested in isolation).

Synergistically testing code and config changes requires *unified regression testing* (URT). URT tests code changes both under the default config and under production configs to increase deployment reliability. For config changes, URT tests the changed configs against the latest code version, which may include code changes since the previous test run. URT applies when project changes are not only to code or configs separately, but also to both code and configs together. In large software organizations, code-config co-changes are common, partially due to increasing popularity of monolithic software repositories driven by the DevOps practice of maintaining both code and production configs [4], [32]–[34]. Inconsistencies between code and config changes constantly result in production failures, e.g., as Microsoft reports [4]. So, the ability to test code-config co-changes is critically important.

However, URT could be very costly because it has to run a large number of tests under multiple configs, including the default config and several production configs. Note that it is a norm that *multiple* production configs co-exist in production deployments; for example, production systems are constantly under staged deployments with multiple versions of code or configs [6], [18]. As a reference, the cost of regression testing under only the default config is already considered high [35]–[39]. If every test needs to be run for every config, the regression testing cost could become unaffordable.

Regression test selection (RTS) [10], [40]–[43] can effectively reduce the cost of regression testing. RTS runs only a subset of the regression tests that are *affected* by the project changes; in other words, RTS does not run the tests whose outcome cannot change due to the recent changes. RTS is

widely used in large software organizations, e.g., Google and Meta publicly report on the practice [35]–[39]. RTS is successful because project changes in CI/CD environment are incremental—they typically change a small part of a large software project, i.e., a small piece of code or a small number of config values. It is well documented that code changes are relatively small [44]–[46], and one study reports that 49.5% of config changes are just two-line revisions [21].

Unfortunately, no existing RTS technique is tailored for URT—existing RTS techniques are designed either for code changes only [10] or for config changes only [15]. The former is done with *regular tests* and the latter with *config tests*. We use the term "test" to generically refer to either a regular test or a config test. No RTS technique reasons about code and config changes collectively, and no technique works with both regular tests and config tests.

We introduce *unified regression test selection* (uRTS), the first RTS technique for URT. uRTS works with project changes on 1) code only, 2) configs only, and 3) both code and configs. uRTS is based on the following observations:

- Config tests can be used to test *code* changes under production configs, in addition to testing *config* changes (its original use case). In essence, a config test exercises code under a specific (default or production) config.
- A project change typically changes a small piece of code or a small number of config values. Therefore, only a subset of tests need to be rerun for any change.
- The production configs are typically largely similar. Therefore, a config test need not be repeatedly run for every production config.

uRTS selects regular tests and config tests with a unified algorithm. The algorithm analyzes code and config dependencies of each test for all configs. uRTS first selects regular tests against the (potentially) changed code under the (potentially) changed default config. Thus, it checks for regression faults in the code under the default config. It then selects config tests against the (potentially) changed code under the (potentially) changed production configs. uRTS uses a *two-dimensional comparison*—comparing dependencies to the previous project revision, and comparing dependencies to the previously run configs—to select fewer config tests and thus speed up testing. A config test is *not* selected iff *both* its code and config dependencies remain unchanged. Not selecting a test could, in general, lead to *unsafe* RTS that misses a test failure.

uRTS provides the same safety guarantee as state-of-the-art RTS for code-only (e.g., Ekstazi [11]) and config-only changes (e.g., Ctest RTS [15]), and uRTS also guarantees safety for code-config co-changes. Meanwhile, uRTS is more effective than state-of-the-art RTS. Compared with traditional RTS for code changes, e.g., Ekstazi, uRTS is aware of changes on the parameter values. The parameter granularity used by uRTS is more precise than the file granularity used by Ekstazi, where any regular test that reads any default config file needs to be rerun even if just one out of hundreds of parameter values in the file is changed. Compared with Ctest RTS for config changes, uRTS selects config tests *across multiple production configs*, while Ctest RTS assumes only one production config.

We implemented uRTS for Java and JUnit on top of Ekstazi [47], [48], a state-of-the-art RTS tool, and openctest [49], a config testing framework. Specifically, our implementation employs Ekstazi to dynamically track file dependencies of code changes and applies the instrumentation techniques of Ctest to dynamically track config dependencies of each test. It also uses openctest to instantiate and run config tests.

We evaluate uRTS on a total of hundreds of code revisions and dozens of config files of five large software projects (HCommon, HDFS, HBase, Alluxio, and ZooKeeper). Some of our experiments are the *largest* RTS experiments performed on open-source projects, e.g., running all regression tests in HDFS for just one project revision takes over 6 hours on a powerful server machine. The results show that uRTS reduces the end-to-end testing time, on average, by 3.64X compared to executing all tests and 1.87X compared to a competitive reference solution that directly extends RTS for URT. Compared to *unsafe* RTS, uRTS increases the testing time by 1.93X when run for *three* configs.

In summary, this paper makes the following contributions:

- **Concept:** We introduce Unified Regression Testing (URT) and motivate the need for RTS in URT.
- **Algorithm:** We develop uRTS, the first RTS for URT. uRTS works with project changes on code, configs, or both. It provides the same safety guarantee as existing RTS but is more effective and applicable in more cases.
- **Implementation:** We implement uRTS on the state-of-the-art RTS tools for regular tests and config tests.
- **Evaluation:** We show the effectiveness of uRTS in reducing the cost of URT with large-scale experiments.
- **Data Availability:** https://github.com/xlab-uiuc/uRTS-ae

II. Background

A. Configuration (Config) Testing

The terms "configuration" and "testing" refer to different concepts in different lines of work. Following Sun et al. [15], we view *config testing* as a technique for detecting erroneous config changes (manifesting as failing tests) early, to prevent them from being deployed to production systems. The original idea of config testing was to connect configs to software tests, so that config changes can be tested in the context of code affected by the changes. Unlike Sun et al. [15] who assumed that code does *not* change, in this work we support the general case where the code, as well as configs, can change.

A config test $\hat{t}(\hat{P})$ is parameterized by a set of config parameters $\hat{P}$. Running a config test instantiates each input parameter from $\hat{P}$ with a concrete value from a production config. Like regular tests, config tests exercise the program and check (via assertions) certain properties (e.g., correctness, performance, security). Fig. 1 illustrates a config test (annotated by @Ctest) from prior work. Note that *when a config test is instantiated by the default config, it is equivalent to a regular test*. This equivalence is the foundation of designing a *unified* solution for both config tests and regular tests.

Fig. 1: A config test instantiated by a changed production config. The test fails as the code under the new production config throws an exception.

Sun et al. [15] showed that config tests can be generated by transforming regular tests, similar to parameterizing existing unit tests [50]. The basic idea is to selectively parameterize a regular test t by the config parameters $\hat{P}$ that are 1) read by the test and 2) generic to the test logic. For a config test $\hat{t}(\hat{P})$, $\hat{P}$ is a subset of all the parameters read by the test t.

Config testing differs from approaches that explore *multiple* configs, e.g., config-aware testing, combinatorial testing, or misconfig-injection testing [51]–[58], which sample representative configs or misconfigs. A config test focuses only on the specific configs to be deployed to the production system.

B. Regression Test Selection (RTS) and Ekstazi

Regression testing is widely used as projects evolve to test whether the recent changes break existing functionality. Regression testing is important but also costly as many tests are run for many changes.

Regression test selection (RTS) [10], [35]–[40] reduces the cost of regression testing by selecting to run only a subset of tests, based on the most recent *code* changes. Traditional regression testing does *not* consider config changes. A typical RTS technique finds dependencies of each test on code parts and selects to run only tests whose execution can reach changed part. Various techniques compute dependencies dynamically or statically; code parts range from statements, basic blocks to methods and classes to entire modules and projects.

Ekstazi. Ekstazi [11] is an open-source RTS tool [47] for Java programs. Ekstazi determines test dependencies dynamically, at the level of files, including code .class[1] files and *optionally* other files. When a test runs, Ekstazi monitors the execution to determine what files the test depends on. The Ekstazi tool provides many options, but one is crucially relevant—what files to track in the dependencies. The tool default, which we call *Ekstazi*$^-$, tracks only .class files, although that option is *unsafe* [11]. We also evaluate an alternative, which we call *Ekstazi*$^+$, that tracks .class files and config files.

For the test in Fig. 1, Ekstazi$^-$ finds that testGetMasterInfoPort depends on the test class HttpServerTest, the HttpServer class directly under test, and all other project and library classes that the execution reaches, including Server that contains the shown doStart method. Ekstazi$^+$ additionally finds that testGetMasterInfoPort depends on hbase-default.xml when run under the default config (or on the production config file when run under it). When the project changes, Ekstazi selects to run testGetMasterInfoPort (or rather, selects the entire class HttpServerTest) if *any* of the dependent files (including HttpServerTest itself) changes in *any* way.

If all the classes (and other tracked files) remain the same, Ekstazi does not select a test because its behavior will be the same as before the changes. Ekstazi operates at the level of classes not methods: 1) it is safer for object-oriented code [11], and 2) it was shown to, somewhat surprisingly, work faster *end-to-end* [11], [60]. A key aspect of RTS is to consider testing time *end-to-end*, from the moment when developers initiate testing (e.g., via mvn test) until they get the result. The time, called $AE(C)$ [11], includes the *analysis* phase (A) that determines what tests to run, the *execution* phase (E) that executes the tests, and (together with the execution or separately) the *collection* phase (C) that collects the dependencies for the next revision. Although class-level RTS selects some more tests than method-level RTS, and thus has a slower E phase, class-level RTS has a much faster (and safer) A phase. Note that in the very first run of a test (e.g., on the first run of Ekstazi, or when a new test is added), Ekstazi has no dependency info, so it always selects to run the test.

An important point is that Ekstazi is *not* config aware and does not handle config files in any special way: Ekstazi$^-$ ignores config files, and Ekstazi$^+$ tracks entire config files.

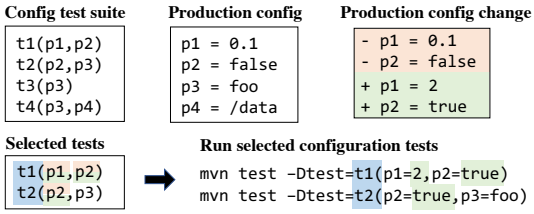

Fig. 2: RTS of config tests for a config change.

C. RTS for Config Tests

Prior work [15] developed a RTS algorithm for config tests but under a restrictive assumption that the *code never changes*. Because typical config changes only update a small number of config parameters [21], not all available config tests need to run. A config test $\hat{t}(\hat{P})$ is selected to run for a given config change if at least one config parameter in $\hat{P}$ is changed. The config change passes if all *selected* config tests pass, and it fails if any selected config test fails. Fig. 2 gives an example.

Note that this prior RTS did *not* consider code changes.

III. Unified Regression Testing

The goal of *unified regression testing* (URT) is to test code changes and config changes *collectively*. We define a project change as a "diff" D that updates the system code S, the

[1]Ekstazi tracks compiled .class files rather than source .java files because different sources can result in the same compiled file (e.g., correcting a misspelling in a comment), and the same source can result in different compiled files (e.g., using a different compiler or linking to a different library [59]). JVM executes the compiled code.

Notation	Description
S	The code (including test code) of the target system
C_{def}	The default config
$\mathbb{C}_{prod}$	The set of production configs
$\hat{t}(\hat{P})$	A config test, where $\hat{P}$ is its input parameter set
$\hat{T}$	The config test suite of all config tests $\hat{T} = \{\hat{t}(\hat{P})\}$
T	The regular test suite $T = \{t\}$, where each t only runs with C_{def}
D	A diff from $(S', C'_{def}, \mathbb{C}'_{prod}, T', \hat{T}')$ to new $(S, C_{def}, \mathbb{C}_{prod}, T, \hat{T})$

TABLE I: Notations used in the paper and their descriptions.

Change Type	Run T	Run $\hat{T}$
$S' \neq S \vee C'_{def} \neq C_{def}$, $\mathbb{C}'_{prod} = \mathbb{C}_{prod}$	Prior work	**Our work**
$S' = S$, $C'_{def} = C_{def}$, $\mathbb{C}'_{prod} \neq \mathbb{C}_{prod}$	Need not run	Recent work [15] for $\lvert\mathbb{C}_{prod}\rvert = 1$ **Our extension** for $\lvert\mathbb{C}_{prod}\rvert > 1$
$S' \neq S \vee C'_{def} \neq C_{def}$, $\mathbb{C}'_{prod} \neq \mathbb{C}_{prod}$	**Our work**	**Our work**

TABLE II: Type of tests run for various types of changes.

default config C_{def}, production configs $\mathbb{C}_{prod}$, the test suite T (for regular tests), or the test suite $\hat{T}$ (for config tests):

$$D : (S', C'_{def}, \mathbb{C}'_{prod}, T', \hat{T}') \rightarrow (S, C_{def}, \mathbb{C}_{prod}, T, \hat{T})$$

Note that D can be one commit or a bundle of several commits, depending how the code/config repositories are maintained and how the project changes are deployed [4], [21], [32], [39], [44]. Table I lists the notation we use.

URT runs regression testing for the following combinations:

- Run regular tests T on the (changed) code S under the (changed) default config C_{def}; and
- Run config tests $\hat{T}$ on the (changed) code S under every (changed) production config $C \in \mathbb{C}_{prod}$.

URT generalizes traditional code-oriented regression testing and config testing. On one hand, it generalizes regression testing [10] by testing code changes under not only the default config but also the production configs (§III-A). On the other hand, it generalizes config testing [15] by testing config changes against the new code (§III-B). Moreover, it handles diffs that co-change both code and configs, be they default or production configs (§III-C). We next discuss these three cases one by one. For now, we assume no regression test selection (RTS)—we are unaware of any prior RTS tailored for URT.

A. Testing Code Changes

For a diff that only changes code or the default config, URT tests the changed code S under 1) the default config C_{def} and 2) every production config $C \in \mathbb{C}_{prod}$. The former is equivalent to traditional regression testing: Basically, URT runs all the tests in the test suite T.

However, traditional regression testing does not test code changes under production configs. URT runs config tests in $\hat{T}$ to check whether the code can be deployed under different production configs. This checking is viable because each config test essentially tests the code under a specific config—a config test $\hat{t}(\hat{P})$ should pass for *any* correct values of parameters in $\hat{P}$. Thus, a config test can check the correctness of not only config changes but also code changes.

Without RTS, for a diff that only changes code or the default config, URT runs $|T| + |\hat{T}| \times |\mathbb{C}_{prod}|$ tests.

B. Testing Production Config Changes

For a diff that only changes production configs, URT tests the code under the changed configs $\mathbb{C}_{prod}$. This checking is done by running config tests $\hat{T}$ against the code $S = S'$ under each $C \in \mathbb{C}_{prod}$. Specifically, URT runs every config test $\hat{t}(\hat{P})$ in $\hat{T}$ by instantiating $\hat{P}$ with values in each production config C. URT passes iff every $\hat{t}(\hat{P})$ under every C passes. This part is equivalent to the original config testing described in §II-A, except that all prior work [15]–[17] assumes there is only one distinct production config, while we allow multiple. Note that multiple production configs are the norm in real-world deployments, e.g., real-world production systems are constantly under staged deployments with multiple production configs in place [6], [18].

Without RTS, for a diff that only changes production configs, URT needs to run $|\hat{T}| \times |\mathbb{C}_{prod}|$ tests.

C. Testing Code-Config Co-Changes

For a diff that co-changes both code and production configs, URT runs tests against the changed code S with all changed configs $\{C_{def}\} \cup \mathbb{C}_{prod}$. URT runs all the tests in T against S under C_{def}. URT also runs all the config tests in $\hat{T}$ against S under every $C \in \mathbb{C}_{prod}$.

Without RTS, for a diff that co-changes both code and production configs, URT runs $|T|$ + $|\hat{T}| \times |\mathbb{C}_{prod}|$ tests.

IV. The uRTS Algorithm

As discussed in §III, unified regression testing (URT) is very expensive without effective *regression test selection* (RTS). However, no existing RTS technique is directly tailored for URT due to not being able to reason about code and config co-changes collectively *and precisely*. Table II presents the types of tests run for different types of project changes, which highlights our contributions over existing RTS.

We develop the *unified regression test selection (uRTS)* to fill this important gap. The goal of uRTS is to reduce the testing time by minimizing the number of tests (both regular tests from T and config tests from $\hat{T}$) to run for a given diff that could change code, configs, or both. We base uRTS on the following key observations:

- A diff typically changes a small piece of code or a small number of config values [21]. Thus, only a subset of T or $\hat{T}$ needs to be run for any diff. The basic assumption of any RTS is that a relatively cheap analysis can select a subset of tests and, thus, save the time that would have been spent running unselected tests.
- A production config typically changes only a small number of config values from the default values [61]. Assuming $\hat{T}$ has been run against C_{def}, a config test $\hat{t}(\hat{P}) \in \hat{T}$ need not be run for any production config that changes none of $p \in \hat{P}$ from the default.
- The n production configs typically only differ in a small number of config values [61]. A config test $\hat{t}(\hat{P})$ need

not be repeatedly run for production configs that share the same values of $\hat{P}$.

For a diff D:$(S', C'_{def}, \mathbb{C}'_{prod}, T', \hat{T}')$→$(S, C_{def}, \mathbb{C}_{prod}, T, \hat{T})$, uRTS employs a two-step test selection algorithm:

1) Select a subset of regular tests from T to check the new code S for the new default config C_{def}.
2) Select a subset of config tests from $\hat{T}$ to check the new code S for each production config in $\mathbb{C}_{prod}$.

A. Step 1: Selecting Regular Tests

If the target diff D changes the code (i.e., $S' \neq S$), uRTS first selects regular tests from T to test the code changes. uRTS applies the traditional RTS to select the tests, i.e., a test t is selected as long as D changes some code exercised by t. Our implementation uses Ekstazi, which checks for each $t \in T$ whether D changes any code file on which t depends.

If the target diff D changes the default config (i.e., $C'_{def} \neq C_{def}$), uRTS further selects from the remaining $t \in T$ every test that exercises at least one config parameter whose values differ in C'_{def} and C_{def}, i.e., the change of $C'_{def} \rightarrow C_{def}$ by D could affect the test results of t. We discuss in §V-B1 how to track config parameters read by each test. Finally, uRTS runs every selected t against S under C_{def}.

Recall that a config test $\hat{t}(\hat{P})$ is parameterized by a set of config parameters $\hat{P}$. Running $\hat{t}(\hat{P})$ against S under C_{def} is equivalent to running a regular test t against S.

B. Step 2: Selecting Config Tests

In this step, uRTS checks the new code S under every new production config $C \in \mathbb{C}_{prod}$. uRTS only selects config tests from $\hat{T}$ because not all tests in T can test production configs.

One can use file-level RTS, like Ekstazi$^+$, to track the (default or production) config but at the level of the *entire file(s)*. Configs are stored in files, usually in a standard format like INI or XML. Ekstazi$^+$ can track for each test what file(s) the test reads. However, tracking config at the file granularity is rather inefficient. A config file could include hundreds of parameters and their values, and be several KBs or even MBs [21]. Even if a diff only changes one config value, all the config tests need to be run because they read the config file. So, uRTS works at the granularity of *config parameters* instead of entire config files.

For each $C \in \mathbb{C}_{prod}$, uRTS selects a $\hat{t}(\hat{P}) \in \hat{T}$ iff no prior test run of $\hat{t}(\hat{P})$ exercises the same code under the same values of all config parameters in $\hat{P}$. In other words, uRTS unselects a config test $\hat{t}(\hat{P})$ as long as it has been run against the same code dependencies and the same values of all parameters from $\hat{P}$ in either 1) Step 1, where a regular test t is equivalent to a config test $\hat{t}$ instantiated with C_{def}; or 2) the last test round, on $(S', C'_{def}, \mathbb{C}'_{prod})$. Consider a config test $\hat{t}(\hat{P})$ for a production config $C \in \mathbb{C}_{prod}$, and let $\mathbb{C}^{<C}_{prod}$ be the set of production configs that run before C. Effectively, the analysis for selecting a test uses a two-dimensional comparison, illustrated in Fig. 3:

- **Horizontal comparison:** if the value of every parameter $p \in \hat{P}$ is the same in C as in another $C^* \in \{C_{def}\} \cup \mathbb{C}^{<C}_{prod}$ (i.e., the test was executed against S under C^*), then $\hat{t}(\hat{P})$ is *not* selected to run for C, because running $\hat{t}$ against S under C is equivalent to a prior test run. Note that this case does *not* compare code dependencies; it is an intentional optimization as code is the same.
- **Vertical comparison:** if the value of every parameter $p \in \hat{P}$ is the same in C as in C' *and* every code dependency of $\hat{t}$ under C' has the same content in S' as in S, then $\hat{t}(\hat{P})$ is *not* selected to run for C, because running $\hat{t}$ against S under C is equivalent to running $\hat{t}$ against S' under C', which was done in the previous test round.

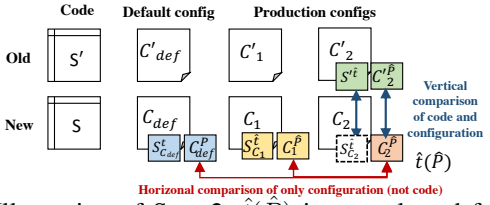

Fig. 3: Illustration of Step 2: $\hat{t}(\hat{P})$ is *not* selected for S and C_2 if either vertical or horizontal comparison matches. One optimization: *not* compare code dependencies horizontally.

C. Correctness

Safety. A critical property of RTS is *safety* [10], [40]—selecting all tests whose results could change due to the project changes. For diffs that only change code or the default config, uRTS provides the same safety as traditional file-based RTS such as Ekstazi [11], in the Ekstazi$^+$ mode, that implicitly considers the default config. uRTS tracks config dependencies at the granularity of config *parameters* instead of config *files*, thus providing more efficient end-to-end testing. For diffs that only change production configs, uRTS provides the same level of safety as the RTS for config testing [15]; uRTS is more efficient than the prior work when multiple production configs exist. Thus, uRTS provides the same safety guarantees when reduced to the special cases. Moreover, uRTS supports the most general case where both code and configs change.

Precision. For a regular test t, uRTS selects t iff the test result could change due to the changed code S or C_{def}. For a config test $\hat{t}(\hat{P})$, uRTS selects $\hat{t}$ iff the code and the config parameters exercised by $\hat{t}$ were not already run in the current test round ("horizontal comparison") or the previous test round ("vertical comparison"). In summary, $\hat{t}(\hat{P})$ runs only *once* for the same code and same set of config values of $\hat{P}$.

V. The uRTS Implementation

We implemented uRTS for Java and JUnit on top of Ekstazi [11] and Ctest [15]. We use Ekstazi to track the code executed by each test, be it regular or config test. We integrate Ctest into Ekstazi to support running config tests. Moreover, we extend Ekstazi's dependency collection and analysis to track config parameters and their values during each test run; We also encode the config parameters and their values in the *dependency file* format that Ekstazi stores for each test [47].

Fig. 4 shows the workflow of our uRTS implementation. uRTS first selects tests (§IV) during the analysis phase (§V-B).

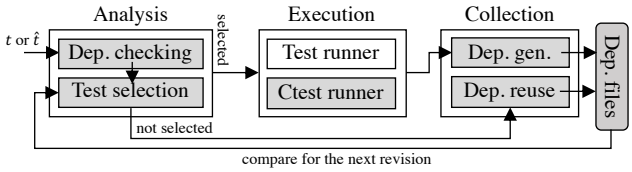

Fig. 4: uRTS implementation uses Ekstazi's integration with JUnit; we added or enhanced the grayed components.

To execute regular tests, uRTS uses the existing Ekstazi integration with JUnit and Maven Surefire. For each test that is not selected, uRTS reuses the test's dependency file generated by an earlier equivalent run; for each test that is selected, uRTS runs the test and generates the new dependency file (§V-C).

A. Dependencies

uRTS maintains two types of dependencies for each test under each config $C \in \{C_{def}\} \cup \mathbb{C}_{prod}$:

- **Code dependency:** the code files that the test depends on, in the Ekstazi form of ⟨URI, checksum⟩ pairs, where the file URI could be a .class file in a directory or in a .jar archive, and the checksum is a hash of the file content. Note that the same test could have different code dependencies when run under different configs.
- **Config dependency:** the names and values of config parameters read by the test, in the form of ⟨parameter, value⟩ pairs. For each config test $\hat{t}(\hat{P})$, its config dependencies only include the parameters in $\hat{P}$.

uRTS maintains a dependency file (with code and config dependencies) for each pair of a test (identified by the test name) and a config (with a separate directory for dependency files for each config). Each dependency file is 1) (re)generated during the test run if the test is selected, or 2) reused from a dependency file of an equivalent run if the test is not selected.

B. Analysis Phase

The analysis phase of uRTS analyzes the latest code and config dependencies for each test (§V-B1), and decides whether or not to select the test to execute (§V-B2).

1) **Analyzing Dependency Changes:** As Ekstazi, uRTS checks whether a file dependency changes between two code versions by comparing the checksums of the file content in S' and S. To analyze config dependencies, uRTS needs to obtain the value of each config parameter in the corresponding config $C \in \{C_{def}\} \cup \mathbb{C}_{prod}$ and to compare the current value with that recorded in a dependency file.

A seemingly easy solution is to simply parse the config files based on their format, which is typically a standard file format such as INI or XML. However, our experience in implementing uRTS for real-world projects shows that configs could have complex representations, making it difficult to understand how they are interpreted by the project. Table III presents some examples from Hadoop, involving config variables and complex dependencies based on config values. It is hard to duplicate such sophisticated logic for each project.

Type	Parameter	Value
Value dep.	**hadoop.tmp.dir** io.seqfile.local.dir	/tmp/hadoop-${**user.name**} ${**hadoop.tmp.dir**}/io/local
Complex dep.	dfs.ha.namenodes.**CID** dfs.namenode.rpc-address.**CID**.**NN1** dfs.namenode.rpc-address.**CID**.**NN2**	**NN1**,**NN2** machine1.example.com:8020 machine2.example.com:8020

TABLE III: Two examples of sophisticated config in the Hadoop project which makes it difficult to analyze config changes by simply parsing a config file.

```
1 public static void recordConfig(String confFile) {
2     Configuration conf = new Configuration(confFile);
3     for(String parameter : conf.getAllKeys()) {
4         String value = conf.get(parameter);
5         ConfigListener.record(parameter, value);
6     }
7 }
```

Fig. 5: The method we added for the Hadoop project to read all the config parameter values of a given file. Hadoop code uses the `Get` API to read config values.

Rather than reverse-engineering sophisticated config logic, our insight is to use a more general, cleaner solution to obtain config parameter values by reusing the project's own config APIs for reading config parameter values.[2] uRTS instruments the same config APIs for the collection phase (§V-C). Using the same interface to collect and compare config dependencies ensures the consistency of the analysis. Specifically, we implement a config reader that invokes the Get API to read the config values of a config parameter. The reader returns all the config parameters and their values from a given config file.

2) **Test Selection:** uRTS implements the test selection algorithm (§IV) based on dependency analysis. If there is no dependency file (e.g., on the first run of uRTS, or when a new test or a new config is added), the test is selected to run.

Step 1 (§IV-A) Selecting regular tests. For each test t against S under C_{def}, uRTS checks whether or not the config dependencies of t (the default parameter values read by t) change. If so, t is selected to run. Otherwise, uRTS further checks whether any code-dependency change. If so, uRTS selects the test t to execute. Otherwise, if neither the config nor the code dependencies change, uRTS does not select t.

Step 2 (§IV-B) Selecting config tests. For each config test $\hat{t}(\hat{P})$ under a given production config file $C \in \mathbb{C}_{prod}$, uRTS first "horizontally" checks whether any value of the $\hat{t}$'s input parameters $\hat{P}$ in C differs from the corresponding value in the config-dependency of $\hat{t}$ previously executed under both C_{def} and other production configs. (Note that $\hat{t}$ could read more parameters than those in $\hat{P}$, but the values of parameters not in $\hat{P}$ are the *same* as in the default config.) If there is no such config difference, uRTS does not select $\hat{t}$.

[2]Modern software projects use uniform APIs for reading configs, which is the basis for recent config analysis techniques [15], [16], [57], [62]–[69]. For example, in many Java projects, the `Get` APIs can be abstracted as a method of the form `String get(String parameter)`. It takes a parameter name as the input and returns its value (which is further typecast by higher level APIs). Many `Get` APIs are declared in wrapper classes on top of `java.util.Properties` for Java projects. Ctest [15] instruments the `Get` APIs to generate config tests from regular tests.

```
1   public String get(String name) {
2 +     String urtParam = name;
3       String[] names = handleDeprecation(
            deprecationContext.get(), name);
4       String result = null;
5       for(String n : names) {
6 +         urtParam = n;
7           result = substituteVars(getProperty(n));
8       }
9 +     ConfigListener.record(urtParam, result);
10      return result;
11  } /* .../hadoop/conf/Configuration.java */
```

Fig. 6: Instrumentation for the `Get` API in Hadoop. The `get` method is the lowest level API used by high-level APIs, e.g., `getInt` and `getBool`. `handleDeprecation` replaces deprecated names.

Otherwise, uRTS further "vertically" compares 1) the current values of $\hat{P}$ in C against the prior config dependency of C' and 2) the current code dependency against the prior code dependency of $\hat{t}$. uRTS selects $\hat{t}$ iff either code or config dependencies change.

C. Collection Phase

1) **Not selected tests:** If a test under $C \in \{C_{def}\} \cup \mathbb{C}_{prod}$ is not selected, it means that uRTS finds an equivalent test run with the same code and config dependencies during horizontal/vertical comparison. In this case, uRTS copies corresponding dependency files to update current dependencies.

2) **Selected tests:** For tests that are selected to run, uRTS executes the test and (re)generates the code and config dependencies. uRTS uses Ekstazi to track code dependencies, as the .class files on which the test execution depends (§II-B). Ekstazi instruments class loading and other class uses (e.g., dereference of static fields) to track the classes, and then maps each class name to the URI file location that stores the class.

uRTS collects config dependencies with the instrumentation of the config APIs. uRTS applies the techniques of Ctest [15] to instrument the config APIs to monitor the config parameters read by each test during the test execution. Fig. 6 shows the instrumentation for the Get API of the Hadoop project, which invokes the same record method as in Figure 5.

D. Optimizations

We apply several optimizations in the uRTS implementation to reduce the analysis time.

First, the horizontal comparison only analyzes config dependencies, not code dependencies. The reason is that the code revision (and, thus, the content/checksum of the .class files) does not change when testing C_{def} and any $C \in \mathbb{C}_{prod}$ (see Fig. 3); hence, if the config dependencies are the same, the test executes the same, and both code and config dependencies are the same. If the config dependencies differ, then the test is selected to rerun anyway, so both its code and config dependencies will be updated. Recall that code dependencies for a config test can differ when running the test under C_{def} or some $C \in \mathbb{C}_{prod}$. This optimization reduces the analysis time, because checking the new code dependencies is expensive due to the need to recompute checksums of all the dependent files.

Moreover, because config dependency comparison is computationally much cheaper than code dependency comparison (the former only involves string comparison without checksum calculation), our implementation always analyzes config dependencies before code dependencies. When config dependencies differ, it saves the overhead of comparing code dependencies (as the test is selected already).

Project	Module	LOC	# Rev.	# Conf. files	# Rev. on $\mathbb{C}_{prod}$	# Test Classes [avg] Regular	Config
HCommon	hadoop-common	256K	50	20	18	510	259
HDFS	hadoop-hdfs	371K	50	20	18	751	751
HBase	hbase-server	427K	50	20	17	295	158
Alluxio	core	154K	50	20	17	247	118
ZooKeeper	zookeeper-server	101K	50	20	19	299	187

TABLE IV: Projects, commits, config files, and tests (regular and config) used in the evaluation.

VI. Experimental Methodology

Evaluating uRTS on open-source projects is challenging, because most projects do not offer a DevOps-based environment with both code and production config changes. RTS research often uses commits from the revision history of open-source projects for code (and default config) changes [11], [12], [70], but it is non-trivial to collect real-world production configs and their revision histories, which are typically proprietary.

We create an evaluation set on top of the Ctest dataset [49] which contains 100 deployed configs collected from public Docker images on DockerHub for five large, widely used open-source projects. We treat those deployed configs as production configs and use multiple deployed configs as different, evolving revisions of one production config. Moreover, we use the most recent compilable commits as our evaluation project changes. Table IV shows the evaluated projects, the statistics of code commits and configs, and the test suites.

Evaluated Projects and Commits. We use the same set of open-source projects as recent work on config testing [15], [16]: HCommon, HDFS, HBase, Alluxio, and ZooKeeper. As all these projects use JUnit as their testing framework,[3] we integrated Ekstazi and uRTS into their build and test systems. All the projects use Apache Maven for building source code and the Maven Surefire plugin for running tests.

For each project, our evaluation uses 50 recent commits. Specifically, we used the newest released version as of 1/2022 of each project as the last commit and checked out 49 prior compilable commits that each modifies relevant code or config. Among these commits, some in HCommon, HDFS and Alluxio change default config included as a part of the codebase. No commit in HBase and ZooKeeper changes default config.

Production Configs. We use real-world configs of each project from public Docker images as the production configs in our evaluation. Those configs were collected in the Ctest dataset [49]. We treat each project as having two production configs in the evolution, to represent the simplest case of multiple configs. In general, the more production configs a system deploys, the *more* benefits uRTS brings.

[3]The Ekstazi tool that we obtained [47] supported only JUnit 4 (and 3); we extended Ekstazi to support JUnit 5 to evaluate ZooKeeper.

Project	ReTestAll		Ekstazi$^+$		uRTS		$T_{+R}\%$	$N_{+R}\%$	$T_{UR}\%$	$N_{UR}\%$	$T_{U+}\%$	$N_{U+}\%$	Ekstazi$^-$		$T_{U-}\%$	$N_{U-}\%$
	time [sec]	# classes	time [sec]	# classes	time [sec]	# classes							time [sec]	# classes		
HCommon	4467.06	1030.50	1303.06	133.40	542.01	51.12	29.17%	12.94%	12.13%	4.96%	41.60%	38.32%	220.67	14.82	245.62%	345.03%
HDFS	65283.01	2049.00	45346.87	759.27	30614.97	478.44	69.46%	37.06%	46.90%	23.35%	67.51%	63.01%	13818.32	199.58	221.55%	239.72%
HBase	2844.60	611.00	1303.90	105.76	652.03	45.27	45.84%	17.31%	22.92%	7.41%	50.01%	42.81%	381.32	31.38	170.99%	144.27%
Alluxio	1831.13	485.50	1401.36	212.22	998.18	97.53	76.53%	43.71%	54.51%	20.09%	71.23%	45.96%	529.78	73.65	188.42%	132.43%
ZooKeeper	3168.50	677.00	1624.30	208.54	701.71	69.21	51.26%	30.80%	22.15%	10.22%	43.20%	33.19%	458.92	46.59	152.91%	148.53%
Σ/avg	5451.09	842.33	2810.57	216.35	1499.39	94.34	51.56%	25.69%	27.51%	11.20%	**53.35%**	**43.61%**	776.72	50.19	193.04%	187.98%

TABLE V: Test run results of ReTestAll, Ekstazi$^+$, uRTS, and Ekstazi$^-$. The average reduction of testing time and the number of selected test classes are denoted by $T_{+R}\%$, $N_{+R}\%$ when comparing Ekstazi$^+$ to ReTestAll; $T_{UR}\%$, $N_{UR}\%$ when comparing uRTS to ReTestAll; $T_{U+}\%$, $N_{U+}\%$ when comparing uRTS to Ekstazi$^+$; and $T_{U-}\%$, $N_{U-}\%$ when comparing uRTS to Ekstazi$^-$.

Unfortunately, Docker images do not provide a revision history of the deployed configs. Therefore, we use the configs to simulate changes of the production configs. For each project, we divide all the configs into two groups to represent two production configs. For each group with n configs, we assume they are n revisions of one config.

We associate each config change with randomly chosen project commits; each commit could change either code (with default config) or both code (with default config) and some production config(s). For each of the two groups of n configs, we independently choose commits, so some commits change both config files. The number of commits with at least one production config change is 18 for HCommon, 18 for HDFS, 17 for HBase, 17 for Alluxio, and 19 for ZooKeeper (Table IV). We have no commit that changes only a production config; for such commits, uRTS would bring even *more* benefits.

Config Tests. The evaluated projects do not come with explicit config test suites. We use Ctest [49] to transform existing tests into config tests by parameterizing those tests with config parameters (§II-A). While the Ctest dataset [49] provides the config tests for the five evaluated projects, those provided config tests were generated for only one version of each project (as the Ctest work did not consider code evolution [15]), and the version is older than all the versions used in our evaluation. Thus, we generate all the config tests for each commit in our evaluation following Sun et al. [15].

Hardware and System Settings. All the experiments are run on Azure VMs with dual-core CPU and 14GB of RAM [71], Ubuntu 20.04.2, and Java 64-bit 1.8.0. The experiments spent 2000+ hours of machine time in total.

Baseline. We compare uRTS with the ReTestAll baseline that conducts URT without RTS (§III). ReTestAll is a common baseline used in RTS research [40]. ReTestAll runs all the regular tests in T if the target commit changes the code or the default config; it also runs all the config tests in $\hat{T}$ for each production config. With the setup of two production configs, ReTestAll runs $|T| + 2 \times |\hat{T}|$ tests for every commit. To limit the machine time used in our experiments, we do not measure ReTestAll time for all 50 commits but only run the first and last commits in the range and use their average time for all the commits. The difference between the time for the first and last commits is, on average, just 1.0%. For the number of test classes, we extract the precise number for every commit, but the difference is again small, on average, just 0.6% between the min and max.

References. We compare uRTS with two reference solutions, Ekstazi$^+$ and Ekstazi$^-$ (§ II-B). For Ekstazi$^+$, we integrated Ctest with Ekstazi, and configured Ekstazi to track for each config test its *file dependencies* that include both code and config files. If a config test depends on a config file that had *any* change from a previous run, then the test is selected. We expect Ekstazi$^+$ to be less effective than uRTS because *file granularity* is rather coarse for config dependencies—a config file could include hundreds of config values [61], while a config change typically modifies only a small number of parameters [21]. In contrast, uRTS tracks config dependencies at the *parameter granularity*.

We also evaluate Ekstazi$^-$, the Ekstazi default that does not track config files and only concerns code changes. The goal is to understand the cost of URT and uRTS over *unsafe*, code-driven regression testing.

Metrics. We use two main RTS metrics: 1) the end-to-end testing time, and 2) the number of selected test classes. For the testing time, we measure the time to *execute the build command* that developers use to execute tests, specifically mvn test for all the evaluated projects. For the commonly running tests, we did *not* modify any build config in any project's build files; hence, the speedup that we observe in our experiments reflects what developers would have experienced.

VII. Evaluation Results

Our evaluation aims to answer the following questions:

1) How effective is uRTS?
2) What is the overhead to support URT?
3) How much does uRTS save for config changes?
4) Are both of the two-dimensional comparisons needed?

A. RQ1: Savings of Testing Time and The Number of Tests

End-to-end testing time is the key metric to measure the effectiveness of RTS in reducing the testing cost; as an additional metric, we use the number of selected test classes. Table V shows the RTS results, in terms of the two metrics, of uRTS, compared with ReTestAll and Ekstazi$^+$. (uRTS provides the *same* safety guarantees as ReTestAll and Ekstazi$^+$.)

For each project, Table V shows the average number over all revisions; lower numbers are better. We compare uRTS over ReTestAll and Ekstazi$^+$ in terms of testing time ($T_{UR}\%$, $T_{U+}\%$) and the number of test classes ($N_{UR}\%$, $N_{U+}\%$), respectively. We compute the ratio for each commit, average the ratios via geometric mean over all commits, and then

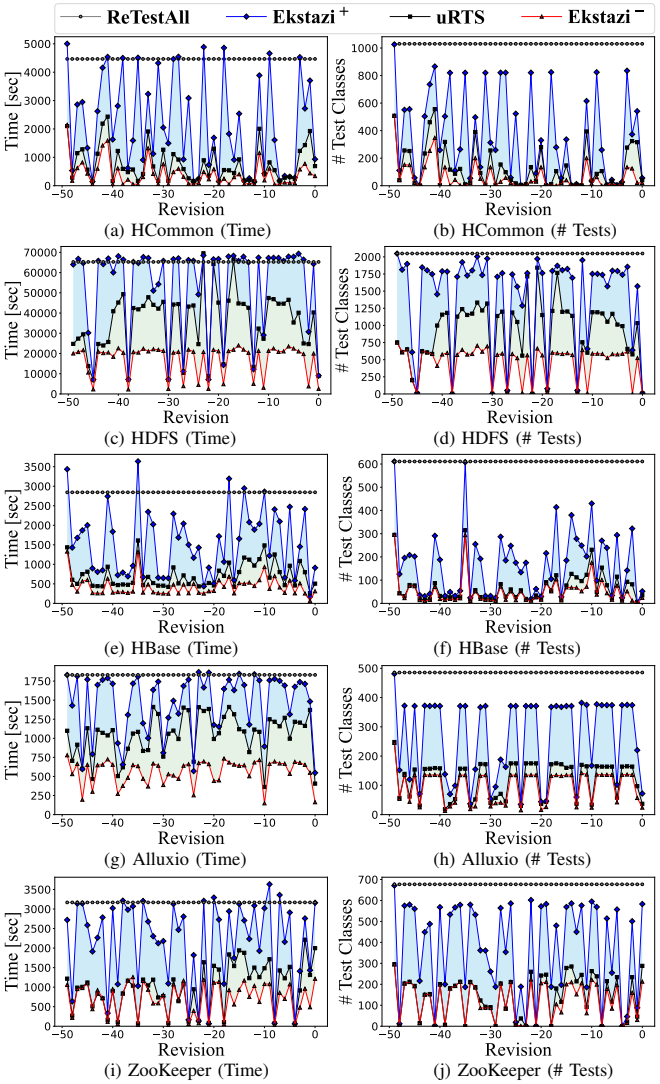

Fig. 7: End-to-end testing time in seconds (a,c,e,g,i) and number of selected test classes (b,d,f,h,j) for all five projects.

average the results across all projects, obtaining an *unweighted* average that equally treats all projects (so the results from the largest project do not dominate the overall average).

Main Results. On average, uRTS only takes 53.35% of testing time compared with Ekstazi$^+$ and 27.51% of testing time compared with ReTestAll. Stated differently, *uRTS reduces the end-to-end testing time by 1.87X compared with Ekstazi$^+$ and 3.64X compared with ReTestAll.*

In terms of the number of test classes, uRTS only selects 43.61% and 11.20% of test classes compared with Ekstazi$^+$ and ReTestAll, respectively. It is not uncommon for RTS to select a smaller percentage of tests than the percentage of the overall time savings [11], [12], for two reasons: 1) the overall time includes not just the test execution time but also the build overhead that is the same both with and without RTS; and 2) the test execution time itself is not proportional to the number of selected tests because the selected tests are typically *longer-running* and *larger*, with more dependencies, than the unselected tests.

For example, for Alluxio, the test class FileSystemFactoryTest runs significantly longer than others. If it is selected, its running time dominates the overall testing time. It is selected often in our evaluation (104 times by uRTS and 135 times by Ekstazi$^+$, out of 150=50×3 cases), which explains the gap between $T_{U+}\%$ of 71.23% and $N_{U+}\%$ of 45.96%. For HDFS, both $T_{U+}\%$ and $N_{U+}\%$ are higher than for the other projects due to a config parameter, dfs.namenode.datanode.registration.ip-hostname-check. 584 test classes depend on this parameter, so when it is changed, all 584 tests are selected.

The $T_{+R}\%$, $N_{+R}\%$ results of Ekstazi$^+$ show the effectiveness of RTS in reducing the test cost of URT. We can see that even a coarse-grained RTS technique like Ekstazi$^+$ can effectively reduce the testing time to 51.56%, on average, and the number of test classes to 25.69%, compared with the baseline (ReTestAll). The $T_{U+}\%$, $N_{U+}\%$ results of uRTS show the further effectiveness of config-aware RTS for URT—tracking config dependencies at the parameter granularity can further reduce the testing time to 53.35%, on average, and the number of test classes to 43.61%, compared with the RTS at the file granularity (Ekstazi$^+$). In brief, *Ekstazi$^+$ almost halves the time of ReTestAll*, and *uRTS further almost halves the time of Ekstazi$^+$*. The results indicate that *with uRTS, URT is substantially cheaper*.

Fig. 7 visualizes the RTS results per commit across the evaluated projects. Ekstazi$^+$ has an even higher testing cost than ReTestAll in the initial commits, because Ekstazi$^+$ needs to run the entire test suites (both T and $\hat{T}$) and collect test dependencies. Collecting dependencies is also the reason why the testing time of Ekstazi$^+$ is sometimes higher than of ReTestAll for later commits (e.g., Fig. 7e). In contrast, uRTS can reduce the testing time effectively compared with Ekstazi$^+$ and ReTestAll from the very first commit by avoiding redundant test runs via the "horizontal" selection across the configs (more discussion in §VII-D).

B. RQ2: Overhead of URT

To understand the overhead of URT over *unsafe* regression testing that ignores config changes, we compare the end-to-end testing time ($T_{U-}\%$) and the number of selected test classes ($T_{U-}\%$) of uRTS over the *unsafe* Ekstazi$^-$. Note that Ekstazi$^-$ only tests code changes under the default config, while uRTS tests three different configs (one default and two production).

Table V and Fig. 7 show the results. On average, uRTS for three different configs takes 1.93X of testing time and 1.88X test classes over Ekstazi$^-$. With uRTS, testing three configs (the default config and two production configs) does not even double the testing cost. The results indicate that *with uRTS, the cost of URT is close to traditional regression testing*.

C. RQ3: Focus on Config Changes

uRTS is expected to reduce more testing cost over Ekstazi$^+$ when config changes are more frequent. For example, Meta reported that config changes are even more frequent than source-code changes [21]. We next focus on RTS results with more frequent production configs. To better understand how

Project	Ekstazi$^+$ [sec]		uRTS [sec]		T_{U+}%		
	C_1	C_2	C_1	C_2	C_1	C_2	$C_1 + C_2$
HCommon	1555.20	1471.42	260.01	255.73	16.72%	17.38%	17.04%
HDFS	22215.74	19965.39	13748.08	3978.08	61.88%	19.92%	42.02%
HBase	1154.25	1127.30	183.58	88.16	15.90%	7.82%	11.91%
Alluxio	555.61	535.49	276.40	206.85	49.75%	38.63%	44.29%
ZooKeeper	1011.87	985.09	61.51	52.98	6.08%	5.38%	5.73%
Σ/avg	1862.64	1771.97	406.92	250.32	21.85%	14.13%	**18.50%**

TABLE VI: Test time for Ekstazi$^+$ and uRTS, with production config change in every commit.

uRTS works for config changes, we focus on the subset of commits with at least one production config change. Table VI shows the RTS results in terms of the testing time but broken down for each of the two production configs separately (we omit the ReTestAll results due to space limit). uRTS takes 21.85% and 14.13% of the time of Ekstazi$^+$ for production configs C_1 and C_2, respectively. The testing time of uRTS is $\approx 1/4$ for C_1 and $\approx 1/7$ for C_2 of the time spent by Ekstazi$^+$, across all projects. If we consider both C_1 and C_2, uRTS takes only 18.50%. We can contrast the overall average for these config-related commits, 18.50% (Table VI), vs. the overall average for all commits, 53.35% (Table V); the percentage is substantially lower, as expected.

D. RQ4: Effectiveness of Two-Dimensional Comparison

For each config test, uRTS makes the selection decision by two-dimensional comparisons (§IV-B). It "horizontally" compares the dependencies with early test run of the same code version but different config(s), while "vertically" compares with the previous code version but (potentially) the same config. We analyzed uRTS performance with only horizontal and only vertical comparison; the results show that the comparisons in both dimensions are necessary—neither dimension subsumes the other. On average, uRTS selects 330 test classes by vertical comparison, 164 by horizontal comparison, and *only* 108 by two-dimensional comparisons. We omit detailed per-project information due to space limit.

VIII. Threats to Validity

The threats to external validity mainly lie in the evaluated projects and dataset. To reduce such threats, we use recent releases of real-world projects and deployed config files from the Ctest dataset [15]. However, we synthesized config changes using different configs files as different versions. Future work should consider more diverse datasets.

The threats to internal validity mainly lie in the potential bugs in our implementations and experimental scripts. We extensively review the code and carefully check the results.

The threats to construct validity mainly lie in the metrics. We consider not only the number of selected tests but chiefly the end-to-end testing time. The time that the developers wait, from initiating a test-suite run for a new code revision until all the test outcomes are available, is the most relevant for RTS.

IX. Discussion

Generality. Our uRTS implementation builds on Ekstazi with dynamic RTS. We chose Ekstazi because it is open source [47], robust, and was used in several studies [11], [60], [72]–[74]. However, the key principles of uRTS—1) tracking config dependencies at the level of config parameters rather than config files, 2) performing both horizontal and vertical comparisons, 3) comparing parameter values across multiple production configs not just against the default—can be applied to other dynamic RTS, whether at a finer granularity of code dependencies (e.g., method) or coarser (e.g., modules).

In fact, we can view the idea of tracking config parameters rather than config files as an application of a general idea to track dependencies at a finer rather than coarser granularity whenever it provides a benefit, i.e., the somewhat higher cost of collection and analysis provides an even higher savings by unselecting tests. Nanda et al. [75] discuss tracking config files, like Ekstazi$^+$, but not the granularity of tracking config parameters instead of files. For any novel application of RTS, it is important to evaluate what granularity level provides a better *end-to-end* time, even if it selects *more* tests to run (e.g., class-level RTS is better than method-level RTS despite selecting more tests). Our experiments show that uRTS provides a better *end-to-end* time than Ekstazi$^+$ and also selects *fewer* tests.

Selection granularity. Our evaluation uses test class as the selection granularity. uRTS supports other selection granularity such as test *method*—both Ekstazi and Ctest can select tests at the method granularity. Although prior studies [11], [60], [76] found that RTS based on class dependencies was more effective, recent work shows that RTS can potentially benefit from a hybrid approach that uses different granularities. We leave the hybrid approach for uRTS as future work.

Nondeterministic tests. One concern, especially for large codebases, is that tests may be nondeterministic and have a different outcome even for the same code and config dependencies. An RTS technique is still safe if it unselects a test when its executions *observed* in prior runs do not change, even if the test may have other executions that could change. If developers want to check more executions of a test, they need to run the test multiple times.

X. Other Related Work

RTS has been studied for 25+ years since seminal work in late 1990's [40], [77]. Several surveys [10], [41]–[43] provide a broad overview. Early research focused on selecting as few tests as possible from the regression test suites, but later work focused on reducing the end-to-end regression testing time. While early techniques tracked test dependencies at fine granularity levels (e.g., basic blocks [40], [78]), over time the dependencies got coarser (e.g., methods [79], classes [11], [60], [74], [76], [80], and modules [12], [81]). RTS is widely used in practice [35]–[39], [82].

RTS for configuration-aware regression testing [53], [83] focuses on *generation* of configs and *prioritization of configs* during regression testing, while uRTS focuses on *production* configs and *selection of tests*. The prior work implicitly

assumes that all regular tests can be used as config tests (i.e., $\hat{T} = T$) and aims to generate (or "select") configs from the large config space. However, bugs that manifest for generated configs may not manifest for real production configs. Likewise, bugs (and misconfigs) that manifest for production configs may be missed with generated configs.

XI. Concluding Remarks

We have presented test selection uRTS for unified regression testing (URT) for both code and config changes. uRTS selects a subset of tests to run and provides the safety guarantees as traditional RTS. uRTS reduces the end-to-end testing time by 1.87X, on average, compared to Ekstazi$^+$, and by 3.64X compared to executing all tests. uRTS is a step toward making URT practical and widely adopted.

Data Availability: https://github.com/xlab-uiuc/uRTS-ae

Acknowledgments

We thank Runxiang Cheng and Xudong Sun for their continuous help on Ctest. This work was partially supported by NSF grants CCF-1763788 and CCF-1956374. We acknowledge support for research on regression testing from Microsoft and Qualcomm.

References

[1] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines*, 2nd ed. Morgan and Claypool Publishers, 2013.

[2] B. Maurer, "Fail at Scale: Reliability in the Face of Rapid Change," *Communications of the ACM*, vol. 58, no. 11, 2015.

[3] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" in *USITS*, 2003.

[4] S. Mehta, R. Bhagwan, R. Kumar, B. Ashok, C. Bansal, C. Maddila, C. Bird, S. Asthana, and A. Kumar, "Rex: Preventing Bugs and Misconfiguration in Large Services using Correlated Change Analysis," in *NSDI*, 2020.

[5] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages," in *SoCC*, 2016.

[6] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous Deployment at Facebook and OANDA," in *ICSE*, 2016.

[7] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams, "The Top 10 Adages in Continuous Deployment," *IEEE Software*, vol. 34, no. 3, 2017.

[8] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, and M. Stumm, "Continuous Deployment of Mobile Software at Facebook," in *FSE*, 2016.

[9] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects," in *ASE*, 2016.

[10] S. Yoo and M. Harman, "Regression Testing Minimisation, Selection and Prioritization: A Survey," *Software Testing, Verification, and Reliability*, vol. 22, no. 2, 2012.

[11] M. Gligoric, L. Eloussi, and D. Marinov, "Practical Regression Test Selection with Dynamic File Dependencies," in *ISSTA*, 2015.

[12] A. Shi, P. Zhao, and D. Marinov, "Understanding and Improving Regression Test Selection in Continuous Integration," in *ISSRE*, 2019.

[13] S. Kendrick, "What Takes Us Down?" *USENIX ;login:*, vol. 37, no. 5, 2012.

[14] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An Empirical Study on Configuration Errors in Commercial and Open Source Systems," in *SOSP*, 2011.

[15] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing Configuration Changes in Context to Prevent Production Failures," in *OSDI*, 2020.

[16] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-Case Prioritization for Configuration Testing," in *ISSTA*, 2021.

[17] T. Xu and O. Legunsen, "Configuration Testing: Testing Configuration Values as Code and with Code," *arXiv*, 2019.

[18] S. Ma, F. Zhou, M. D. Bond, and Y. Wang, "Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems," in *EuroSys*, 2021.

[19] N. Tillmann and W. Schulte, "Parameterized Unit Tests," in *ESEC/FSE*, 2005.

[20] N. Tillmann and W. Schulte, "Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution," *IEEE Software*, vol. 23, no. 4, 2006.

[21] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic Configuration Management at Facebook," in *SOSP*, 2015.

[22] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci, "Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud," in *Middleware*, 2017.

[23] P. Huang, W. J. Bolosky, A. Sigh, and Y. Zhou, "ConfValley: A Systematic Configuration Validation Framework for Cloud Services," in *EuroSys*, 2015.

[24] R. Potharaju, J. Chan, L. Hu, C. Nita-Rotaru, M. Wang, L. Zhang, and N. Jain, "ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection," in *VLDB*, 2015.

[25] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems," in *ESEC/FSE*, 2020.

[26] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff, "Mining for Misconfigured Machines in Grid Systems," in *KDD*, 2006.

[27] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing Configuration File Specifications with Association Rule Learning," in *OOPSLA*, 2017.

[28] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic Automated Language Learning for Configuration Files," in *CAV*, 2016.

[29] O. Tuncer, N. Bila, C. Isci, and A. K. Coskun, "ConfEx: An Analytics Framework for Text-Based Software Configurations in the Cloud," IBM Research, Tech. Rep. RC25675 (WAT1803-107), 2018.

[30] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based Online Configuration Error Detection," in *ATC*, 2011.

[31] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection," in *ASPLOS*, 2014.

[32] R. Potvin and J. Levenberg, "Why Google Stores Billions of Lines of Code in a Single Repository," *Communications of the ACM*, vol. 59, no. 7, 2016.

[33] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software Configuration Engineering in Practice: Interviews, Surveys, and Systematic Literature Review," *TSE*, vol. 46, no. 6, 2020.

[34] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter, and E. Murphy-Hill, "Advantages and Disadvantages of a Monolithic Repository: A case study at Google," in *ICSE*, 2018.

[35] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for Improving Regression Testing in Continuous Integration Development Environments," in *FSE*, 2014.

[36] Z. Mi, "Mobile performance: Tooling infrastructure at Facebook," https://engineering.fb.com/2015/04/10/developer-tools/mobile-performance-tooling-infrastructure-at-facebook, 2015.

[37] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-Scale Continuous Testing," in *ICSE-SEIP*, 2017.

[38] C. Leong, A. Singh, J. Micco, M. Papadakis, and Y. le Traon, "Assessing Transition-based Test Selection Algorithms at Google," in *ICSE-SEIP*, 2019.

[39] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive Test Selection," in *ICSE-SEIP*, 2019.

[40] G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, 1997.

[41] E. Engström, P. Runeson, and M. Skoglund, "A Systematic Review on Regression Test Selection Techniques," *Information and Software Technology*, vol. 52, no. 1, 2010.

[42] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression Test Selection Techniques: A Survey," *Informatica*, vol. 35, 2011.

[43] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective Regression Test Case Selection: A Systematic Literature Review," *ACM Comput. Surv.*, vol. 50, no. 2, 2017.

[44] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How Do Centralized and Distributed Version Control Systems Impact Software Changes?" in *ICSE*, 2014.

[45] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study," in *ASE*, 2017.

[46] H. L. Nguyen and C.-L. Ignat, "An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects," *Comput. Supported Coop. Work*, vol. 27, no. 3–6, 2018.

[47] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight Test Selection," http://www.ekstazi.org, 2022.

[48] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight Test Selection," in *ICSE (Demo)*, 2015.

[49] "openctest," https://github.com/xlab-uiuc/openctest, 2022.

[50] S. Thummalapenta, M. R. Marri, T. Xie, N. Tillmann, and J. de Halleux, "Retrofitting Unit Tests for Parameterized Unit Testing," in *FASE*, 2011.

[51] H. Srikanth, M. B. Cohen, and X. Qu, "Reducing Field Failures in System Configurable Software: Cost-Based Prioritization," in *ISSRE*, 2009.

[52] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D'Amorim, "SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems," in *ESEC/FSE*, 2013.

[53] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization," in *ISSTA*, 2008.

[54] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines," *TSE*, vol. 40, no. 7, 2014.

[55] M. Mukelabai, D. Nešić, S. Maro, T. Berger, and J.-P. Steghöfer, "Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems," in *ASE*, 2018.

[56] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A Comparison of 10 Sampling Algorithms for Configurable Systems," in *ICSE*, 2016.

[57] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do Not Blame Users for Misconfigurations," in *SOSP*, 2013.

[58] L. Keller, P. Upadhyaya, and G. Candea, "ConfErr: A Tool for Assessing Resilience to Human Configuration Errors," 2008.

[59] J. Dietrich, K. Jezek, and P. Brada, "What Java Developers Know About Compatibility, And Why This Matters," *Empirical Software Engineering*, vol. 21, no. 3, 2016.

[60] L. Zhang, "Hybrid Regression Test Selection," in *ICSE*, 2018.

[61] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software," in *ESEC/FSE*, 2015.

[73] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, "A Framework for Checking Regression Test Selection Tools," in *ICSE*, 2019.

[62] M. Lillack, C. Kästner, and E. Bodden, "Tracking Load-time Configuration Options," *TSE*, vol. 44, no. 12, 2018.

[63] M. Lillack, C. Kästner, and E. Bodden, "Tracking Load-time Configuration Options," in *ASE*, 2014.

[64] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early Detection of Configuration Errors to Reduce Failure Damage," in *OSDI*, 2016.

[65] A. Rabkin and R. Katz, "Static Extraction of Program Configuration Options," in *ICSE*, 2011.

[66] A. Rabkin and R. Katz, "Precomputing Possible Configuration Error Diagnosis," in *ASE*, 2011.

[67] F. Behrang, M. B. Cohen, and A. Orso, "Users Beware: Preference Inconsistencies Ahead," in *ESEC/FSE*, 2015.

[68] S. Zhang and M. D. Ernst, "Automated Diagnosis of Software Configuration Errors," in *ICSE*, 2013.

[69] S. Zhang and M. D. Ernst, "Which Configuration Option Should I Change?" in *ICSE*, 2014.

[70] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration," in *ISSTA*, 2021.

[71] "Azure Linux Virtual Machines Pricing," https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/#pricing.

[72] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression Test Selection Across JVM Boundaries," in *ESEC/FSE*, 2017.

[74] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-Aware Static Regression Test Selection," in *OOPSLA*, 2019.

[75] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso, "Regression Testing in the Presence of Non-Code Changes," in *ICST*, 2011.

[76] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An Extensive Study of Static Regression Test Selection in Modern Software Evolution," in *FSE*, 2016.

[77] G. Rothermel and M. Harrold, "Analyzing Regression Test Selection Techniques," *TSE*, vol. 22, no. 8, 1996.

[78] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression Test Selection for Java Software," in *OOPSLA*, 2001.

[79] L. Zhang, M. Kim, and S. Khurshid, "FaultTracer: A Change Impact and Regression Fault Analysis Tool for Evolving Java Programs," in *FSE*, 2012.

[80] A. Orso, N. Shi, and M. J. Harrold, "Scaling Regression Testing to Large Software Systems," in *FSE*, 2004.

[81] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. Module-level Regression Test Selection for .NET," in *ESEC/FSE*, 2017.

[82] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka, "Optimizing Test Placement for Module-level Regression Testing," in *ICSE*, 2017.

[83] M. B. Cohen, J. Snyder, and G. Rothermel, "Testing across Configurations: Implications for Combinatorial Testing," in *Proceedings of the 2nd Workshop on Advances in Model Based Testing (A-MOST'06)*, 2006.