Modern software is multi-faceted. As modern software provides more functionalities, software systems are becoming larger and more complex. They often incorporate a wide range of artifacts, from source code in different languages to various types of non-code artifacts. Our recent work also shows that 97% of a video game repository with over a million files are non-code files [19], highlighting the scale and complexity of modern software. To weave these artifacts into a cohesive whole, the build system is responsible for compiling application code into binaries, managing dependencies, executing test cases, and creating deliverables. These benefits come at the cost of a heavy workload incurring expensive computing resources and energy.

Despite the growing size and complexity of software systems, there is also an increasing demand for faster delivery. Our recent work has shown that the average build time for a video game project at Ubisoft is 20 minutes and 16% of the builds fail [19]. Failures not only block others from validating their work but also necessitate repeated executions, incurring more resource consumption. In addition to the computing resources, energy, and time spent executing the build job, fixing build failures is also time-consuming [5–7, 21]. Currently, build failures are typically resolved manually. In a case study at a large organization, 18% of builds failed, with an average of 56 minutes spent fixing these failures [11]. In accelerating the build process, we aim to reduce inefficiencies that contribute to wasted computing resources and energy and hinder productivity, highlighting the need for automated solutions to address build failures in the software industry.

Our recent work, RavenBuild, predicts the build outcome of a build before its execution [18]. However, even with these predicted outcomes, manual effort to locate and fix the build failure is still required. Extensive work has shown that manually addressing the build failure can be challenging and time-consuming [5–7, 21]. Prior studies have shown that in the open-source community, the top three reasons for build failures are compilation errors, dependency issues, and test execution errors [22].

Although various methods for fixing compilation errors, dependency issues, and test execution errors have been individually studied, there is limited research on an integrated approach to handling build failures as a whole. Existing work typically evaluates fixes at one stage of the build process, but changes at one stage can introduce new errors in later stages due to the inter-dependencies within the build pipeline. For example, updating dependencies may resolve a dependency-induced failure, but also introduce a new failure due to a version incompatibility. Current approaches fall short of addressing these cross-stage issues. To develop a comprehensive automatic build repair system, we propose the following research objectives (ROs):

RO1) Formulate automatic build repair strategies for compilation errors

In open-source projects, 9.1% of build failures are caused by compilation errors [22] which tend to be syntactical in nature and follow recurrent patterns [15]. Previous work has explored learning-based approaches to address this issue [1,3,12,15,17,23]. However, existing work does not validate the generated fixes for compilation errors in the subsequent build stages. A compilable program that behaves in an unexpected way may fail the build at the subsequent testing stage, or worse, allow defective software to be released. Thus, repairing compilation errors alone does not guarantee a successful build. Therefore, when evaluating the required accuracy, we will also consider whether the fix generates new failures in subsequent build stages. To address RO1, we will leverage natural language processing techniques with fine-tuning to accommodate the software engineering domain knowledge to consider test specifications when generating fixes for compilation errors.

RO2) Formulate automatic build repair strategies for dependency errors

Dependency errors account for 7.1% of build failures in open-source projects [22]. Prior efforts have focused on automating the repair of dependency-related failures by injecting or updating third-party libraries [14]. However, these methods can only successfully repair 46% to 54% of the studied broken builds, indicating that there is potential room for further improvement in this area. To address RO2, we

will propose an ensemble approach for repairing dependency errors by incorporating machine learning approaches to complement the existing heuristic approach.

RO3) Formulate automatic build repair strategies for test execution errors

As the most frequent cause of failing builds, test execution failures have been the focus of extensive research. Test execution failures are the leading cause of build failures, responsible for 41.3% of failures in open-source communities [22]. Unlike syntactically incorrect compilation errors, test cases verify if the code behaves as expected (*i.e.,* semantically correct). Additionally, fixing test execution errors can generate fixes that introduce compilation errors [8, 24, 25]. Existing work employs an iterative approach that compiles and tests these candidate fixes until a true fix is found [25]. To address RO3, in addition to the iterative validations, we will develop grammar-constrained methods that take both syntax and semantics into account to reduce the generation of candidate fixes that could not be compiled.

RO4) Formulate integrated strategies for an automated build repair solution

Prior work has focused on fixing compilation errors [1, 3, 12, 15, 17, 23], test cases [8, 9, 13], or specific types of defects in isolation (*e.g.,* build scripts [4, 14]). At the build level, researchers have proposed approaches that provide hints to help developers to fix build failures [20, 21]. To the best of our knowledge, there have been no attempts to integrate these efforts to repair the entire build holistically. To further automate the build-repairing process, a comprehensive approach should address all aspects of build failures and restore the build to a functional state in full automation. To address RO4, our approach will fix build errors sequentially, addressing issues in the order they occur (*i.e.,* compilation, dependency specifications, test executions) while iterating through the stages to ensure a complete repair.

Technical Approaches

We will address build failures with learning-based and template-based approaches. With large language models demonstrating promising performance, research increasingly leverages learning-based techniques for automatic program repair [1, 3, 8, 9, 12, 13, 15, 17, 23], whereas in resource-constrained scenarios, template-based methods can be adopted to balance performance and resource demands.

Data Collection - We will construct datasets that contain all types of build failures to train and evaluate the integrated system. By mining the wealth of open-source repositories hosted on social coding platforms (*e.g.,* GitHub), we can reproduce a large sample of build invocations to construct a dataset containing compilation-related, dependency-induced, and test-related build failures and their corresponding fixes. Off-the-shelf datasets such as DeepFix [2] for compilation errors, and Defects4J [10] for test execution errors can be used to train and test our task-specific components. However, there are limited existing benchmarks for dependency-induced failures and fixes. To address this shortfall of data, we will start studying the dependency-induced build failures by constructing a public dataset to fill in the gap.

Method - Using the collected data, we will conduct an empirical analysis to understand developer-generated fixes and propose models for each type of build failure. These models will consider the interdependencies between different stages of the build process. In resource-constrained scenarios, we can propose template-based fixing approaches that leverage the empirical analysis knowledge.

Validation - Fixes can be validated in several ways. Some approaches compare generated fixes to the ground truth, while others argue that multiple valid fixes exist for a single failure [3]. At the test level, executable datasets such as RunBugRun [16] can be adopted to validate the diverse candidate fixes. At the build level, the ultimate validation is whether the build succeeds after applying the fix.

Evaluation Metrics - In addition to the existing effectiveness metrics in measuring repair strategies, we will propose new metrics at the build level. These include the number of builds fully fixed, builds fixed at each stage, and the median number of test cases repaired in partially fixed builds. Efficiency will be measured by the time taken to fix the entire build and the time taken to resolve failures at each stage.