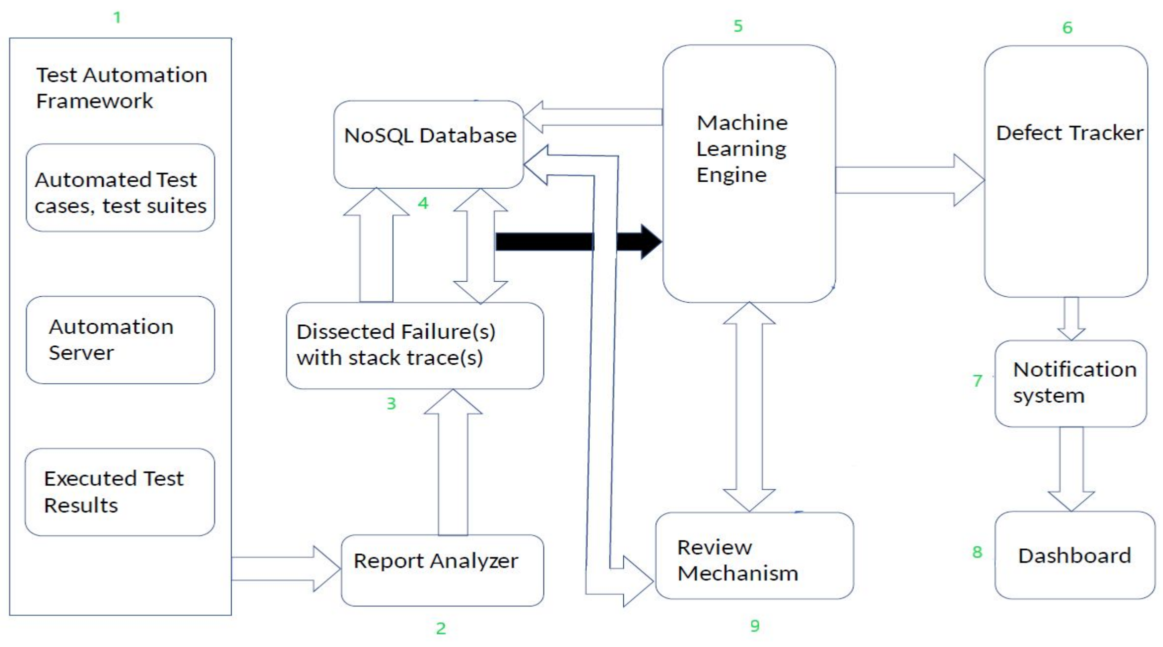
Automated Software Testing with Machine Learning

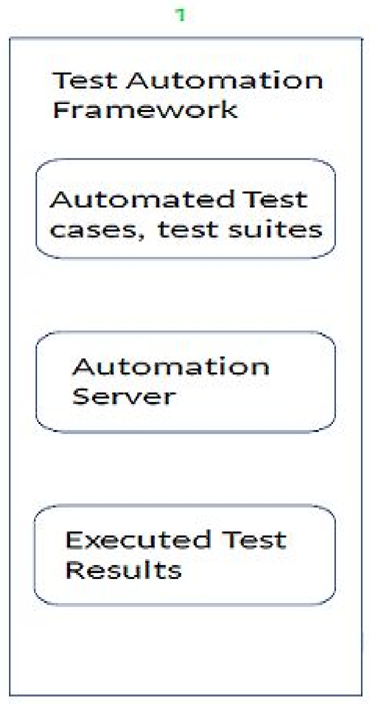
Global Finance IT - Strategic Reporting



Overview

Illustration above presents an approach to assemble test management life-cycle in a fecund and efficient manner, that is brisk and reliable. In this setup within the testing pipeline, machine learning algorithms act as a catalyst for bug triaging, filing and notification systems. Sequences of supervised and unsupervised learning programs enable swift, automated test processes in a quality controlled environment to effectively predict & handle any occurring defects.

Repairing programs by generate-and-validate techniques gain a lot by producing acceptable, well-fitted patches through learning from past history to generate replacement patches from correct code via a probabilistic model. So reduces effort and cost of bug fixing!



Architecture

For better comprehension, we'll dive deeper into this recursive pipeline segregated below in successive steps.

Core Testing Framework [1]

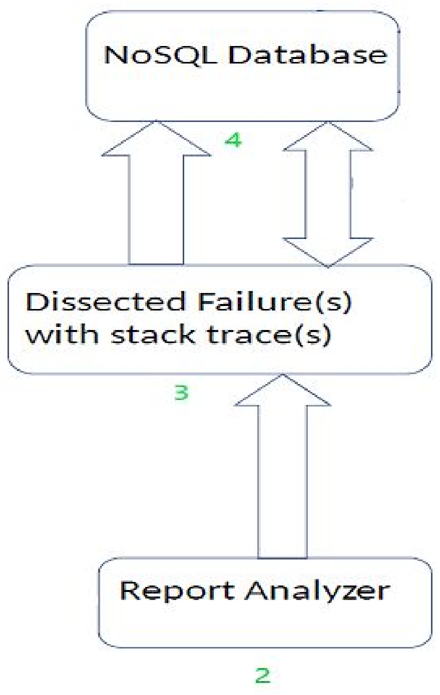
First pillar of our test automation framework contains automated test cases and automated test suites. Easy to interpret feature files created in native English language with business logic for automated test cases are programmatically powered by logic code written in languages like Python or Java for Behavior driven development.

Automated test suite is a collection/pipeline of these test cases, page objects and utility programs processed through an automation server. Such a server like Jenkins acts as an interface to easily configure business requirements & dependencies for desired test results in a predetermined format like XML or JSON by running scheduled jobs at stipulated time.

Report Analyzer & Dissected Failures [2,3]

Report parser infers core framework test results to capture failures or exceptions with their respective stack traces. It is a program written in languages like Python or Java, that goes through the abridged content of the result file and searches for certain attributes or keywords.

Our system picks resultant XML file with records from the automation (say, Jenkins) server location and by using a preferred programming language (say, Python) script, it goes through the length and breadth of the XML file to filter out all the respective failures or exceptions for respective tests with respective stack traces.



NoSQL Database [4]

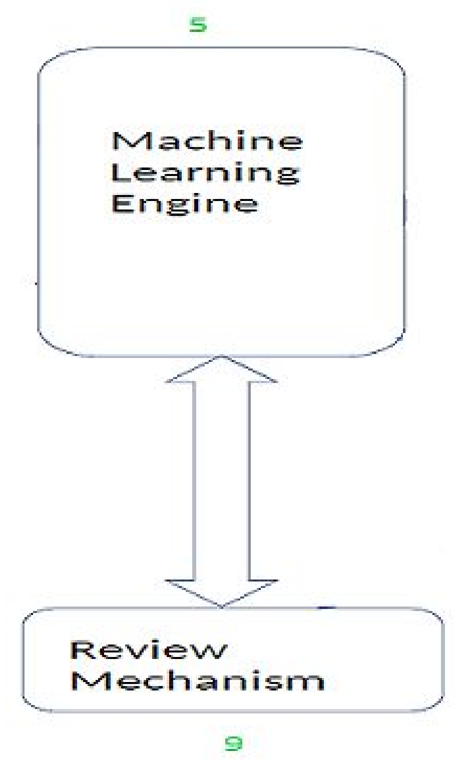
Records of our parsed and dissected results can be placed in a NoSQL database table (or in Hadoop ecosystem) where these will be eventually compared with our historical results for further processing. In a preferred paradigm, this facilitates comparison of the historical result set with the newly encountered failures or exceptions.

Machine Learning Engine [5]

In this embodiment, once the current failure or exception is found among historical records in our NoSQL database, then ML engine kicks in to detect whether a defect ticket was created earlier for this automated test. On positive confirmation of an antecedent, the failure or exception is classified as a 'known issue' with an already recorded stack trace for debugging.

When an ML engine is unable to detect the current failure or exception in the historical data, then it classifies the defect as an 'unknown issue' and accordingly files it. For this, ML engine uses the test management tool (say, JIRA) and its APIs to file the failure as a new defect with all the related stack traces or exceptions in the ticket. Severity of the bug and priority of the failed test case remains same if it is a historically encountered failure.

If this failure is new, then our ML engine gets the priority of our automated test case for which the failure occurred and severity of the defect can be mapped in two ways. In the first way, if our failed test case has high priority then the bug is considered to be of high severity; and if a low priority test case failed, then the bug is of low severity. Thus the system has a mechanism to map the severity of the failure to the defect ticket.



Review Mechanism [9]

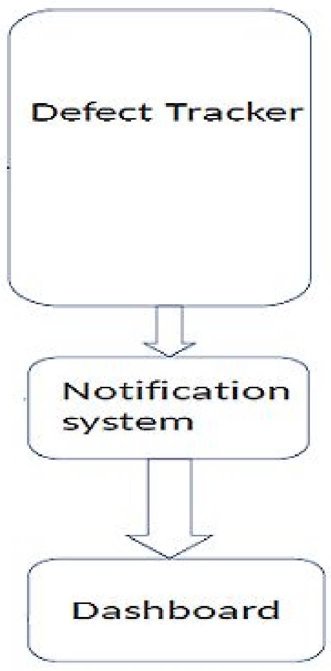
Our second approach of defect severity detection, enables ML engine to learn in an unsupervised manner by looking at the historical failure or exception data along with stack traces.

Argumentatively, in case the severity predicted by ML algorithm is incorrect as per test engineer, he/she can manually modify severity mapping for such types of failure or exception in the NoSQL database mapping structure. Such manual feedback in turn boosts confidence intervals and precision of our engine algorithm.

Intrinsic to note our algorithms are not equipped to deal with all kinds of scenarios and exceptional cases as it is built upon the idea that it would learn based on the data provided, and to learn in a supervised and an unsupervised manner. Thus feedback mechanism helps in fine tuning ML algorithms such that we can make it's learning targeted towards providing sensible outputs. Hence, the algorithms and programs control the quality so any number of defects can be handled and save time for concerned personnel.

Defect Tracker [6]

Another paradigm of our structure is creation of bug tickets in a defect tracking tool like JIRA or Bugzilla, that includes logging the failure or exception as a defect or bug in ticket management tool for our developers to fix. If the failure is new then a new ticket is created and status of the new ticket is kept as 'Open'. If the failure is already present in our tracking tool and the status of that ticket is 'Closed' then status of that ticket gets changed to 'Reopen'. If the failure is already present in our defect tracking tool and status of bug ticket is either 'Known' , 'Deferred' , 'Open' or 'Reopen' then the system will add a comment with current failure details. All this is done through the defect/bug tracking system's APIs.



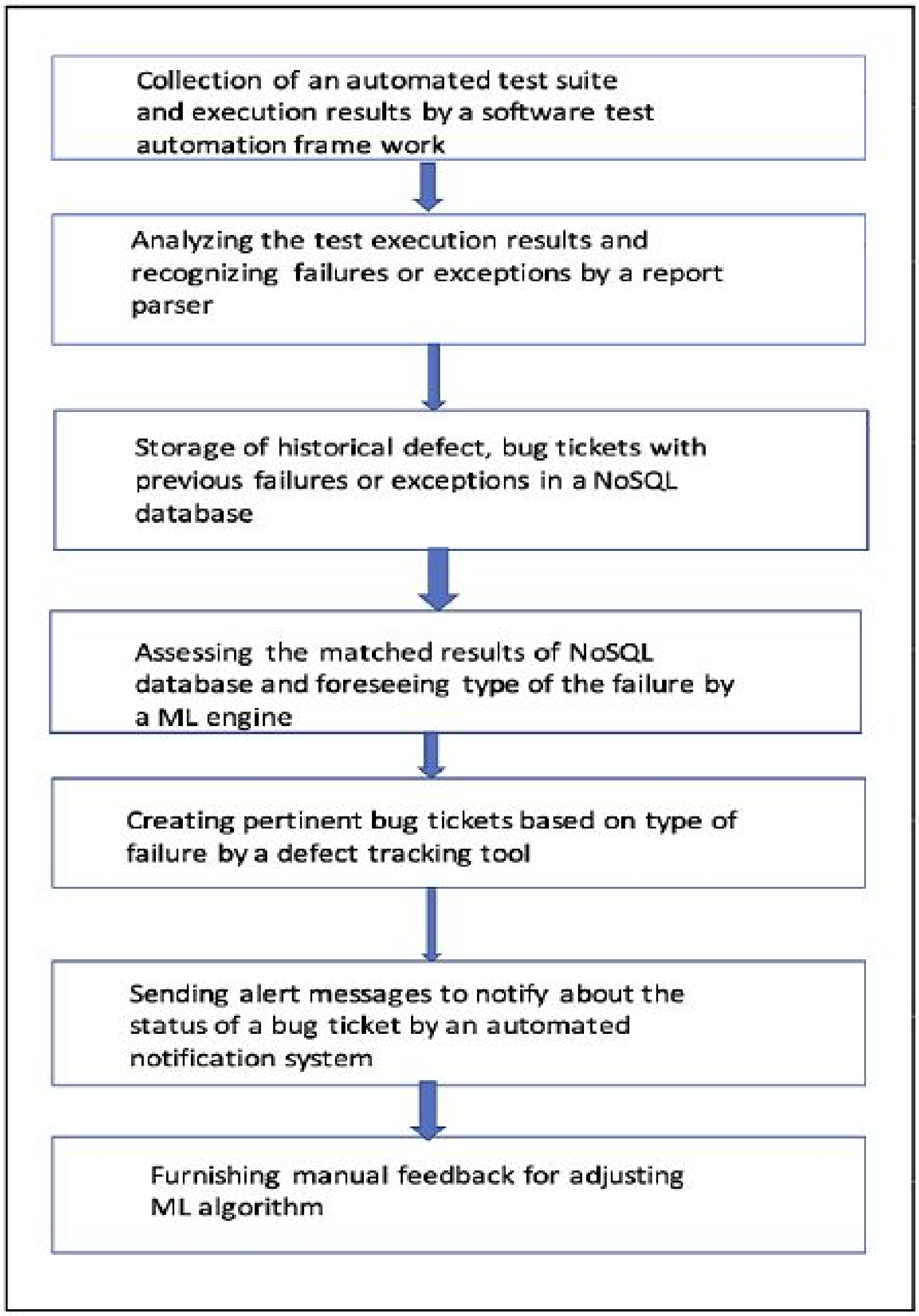
Notification & Dashboard [7,8]

Once a bug ticket is created or reopened, our system will automatically notify stakeholders via email or instant messaging about their status. In comparison to static software testing tools, our architecture scales analysis to find deep and intricate potentially multi-threaded software bugs.

Furthermore by using ML to learn program behavior, our approach provides heuristics to automatically uncover hypothetical program-dependent properties. Ticketing dashboard integrated with added options like applied clustering algorithms shall enhance visibility of automated test suites.

Summary

- Software test automation framework configured to collect automated test suite and test execution results.
- A report parser to infer test execution results generated by test framework and configured to identify the failures or exceptions with their respective stack trace.
- A NoSQL database configured to hold historical defects, bug tickets with prior failures.
- ML engine to identify and evaluate matching results from NoSQL database along with defect type prediction of arising failure or exception.
- Defect-tracking tool configured to create relevant bug tickets based on failure type.
- An automated notification system configured to notify the status of a bug ticket.
- Dashboard to facilitate ease of access to results, logs, failures, key performance indicators etc. in form of histograms, pie graphs etc.
- A manual feedback mechanism for algorithm improvement.


Collection of an automated test suite and execution results by a software test automation frame work
Analyzing the test execution results and recognizing failures or exceptions by a report parser
Storage of historical defect, bug tickets with previous failures or exceptions in a NoSQL database
Assessing the matched results of NoSQL database and foreseeing type of the failure by a ML engine
Creating pertinent bug tickets based on type of failure by a defect tracking tool
Sending alert messages to notify about the status of a bug ticket by an automated notification system
Furnishing manual feedback for adjusting ML algorithm