International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 4, Issue 4, 163-166, 2023
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V4I4P118

Original Article

CI/CD-Triggered Functional Testing of Billing Engines Using Parameterized Xml Payloads and SQL Checks

Udayan Verma
Denver, USA.

Abstract *- Automating testing of functional billing engine testing by allowing for test authoring in CI/CD generated pipelines, is a standard for billing applications. By using parameterized XML and SQL database validation, companies can test and validate both transaction accuracy and long term data accuracy. In this paper we highlight our work with Charter Communications of creating Jenkins pipelines to automatically test for functional testing of rating, invoicing and adjustment transactions. We created dynamic parameterized XML template which allowed us to create near any billing transaction with SQL checks to verify both ledger balance, account balances, audit log balance. We implemented double barrier validations, front end validations and back end persistence validations to create two check points for errors as early as possible in the build/release cycle. Dashboards and reporting offered transparency, Jenkins offered the tests, every time a change is made, as regression tests with continuous delivery. Together, we realized efficiencies in process, and reduced human error, and decreased revenue leakage and compliance violations in a high transaction volume billing applications.*

Keywords *- CI/CD, Jenkins Pipeline, Billing Engine Testing, Xml Payloads, SQL Validation, Test Automation, Financial Systems, Regression Testing.*

1. Introduction

The billing engine is critical in telecommunications and financial systems. Billing engines process millions of transactions per day and affect customer experience and revenue assurance. These engines perform rating, billing, adjustments, and ledger postings and must provide accurate results despite the complexity of: . . . code changes, new code additions, ever-changing business rules. If there is any flaw in these processes, the bill might be wrong, which can result in customer disputes about bills, revenue loss, and possible violations of regulations. This is where robust functional testing is indispensable. Functional testing previously involved executing manual tests in billing environments; not only is it labor intensive, it involved error prone human activity, especially when conducting many cycles/ customer scenarios. Test coverage was minimal, execution cycles were lengthy, and blindfolded humans added variability in the results. The advent of continuous integration/continuous deployment (CI/CD) at organizations certainly made performing manual tests in the delivery pipeline unmanageable compared to the pace of the delivery pipeline. To modernize, Charter Communications had a CI/CD pipeline testing framework that included the functional validations into the Jenkins pipeline. With XML payloads, templates, and SQL-based back-end checks, the framework used continuous automated tests to validate billing transactions. The testing framework reduces prioritization in finding defects, maintain transactional integrity, and enhance durable financial systems implementations.

2. Background and Rationale

Automated testing was pursued due to the sheer size of the billing and financial systems, and the critical nature of risk control and risk mitigation. Billing systems have very complex processes with prorating, discounting, late fees, and multi-cycle adjustments. Even a small error in the calculation can create major financial discrepancies that will result in customer disputes, loss of revenue, and possibly regulatory issues. As of now, most Billing and finance departments were still utilizing slow manual testing practices, with coverage limitations. Manual testing techniques cannot absorb (or process) large volumes of transactional data, PLUS layers of complexity with ever changing billing rules [1].

Early direction of the industry was shifting towards automated testing integrated with CI/CD pipelines to eliminate the inefficiencies of manual testing. Regression and functional checks that are run automatically with each release cycle offered reliability and faster feedback loops. Parameterized XML payloads became the method of choice as they offered reusable templates that could be dynamically changed using account identifiers, billing dates, rate plans, or adjustment codes. While this was happening, SQL validations were conducted at the database level to check ledger accuracy, calculated account balances and audit consistency. By connecting XML-driven triggers with SQL-based assertions, Charter created a Jenkins-enabled solution that elevated functional testing to a continuous, scalable, auditable process.

3. System Architecture & Pipeline Design

The architecture of the automated testing framework was based upon Jenkins pipelines designed to invoke functional validations at each code integration, deployment, or scheduled billing cycle. The system was two-layered: the first and primary layer was to invoke XML payloads to each API or message queue representing billing transactions; then execute SQL checks on the database for data persistence and integrity. Parameterized XML Payloads served as the first layer [2]. This reduced duplication and improved the speed of designing the tests.

The next level of SQL Database Validation was to verify transactional incremental accuracy. SQL was ran on the billing databases to validate ledgers were accurate, account balances were accurate, and reporting/transactions were accurate. This was to validate the transaction confirmed success, as well as compliance. All layers of validation were executed in a Jenkins pipelined approach declaratively using a pipeline-as-code. The stages were enumerated submit xml, database validation, results amalgamation, and automated reporting. When any of the stages failed, notifications notified that app defect, before the pipeline advanced [3].

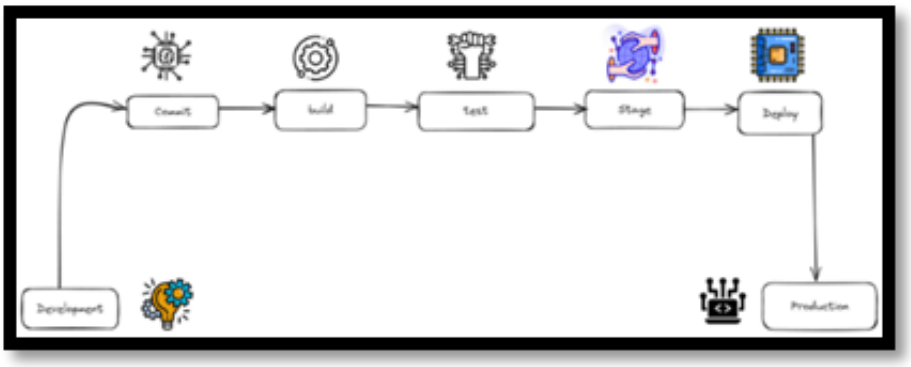

Fig 1: Jenkins-Driven Two-Layer Validation Framework

Ultimately, it provided confidence in the CI/CD by removing manual dependencies and enabling auditable replicable execution of tests.

4. Integration & Maintenance Strategy

To promote sustainment of future use, testing assets had to be integrated into CI/CD executions and test environments were routinely maintained. Charter's solution focused on modularity, interoperability and maintainability across their testing stack. This made it necessary for any changes to billing logic to reflect changes in the test payloads. The Jenkins pipelines would fetch the most current templates during a test run and kept the verification aligned with changing business rules. Likewise, the SQL scripts associated with back-end checks were packaged into modules, allow for reuse across various workflows while reducing duplication [4].

On the maintenance side, there were certain risk factors to consider involving schema changes, pipeline outage, or outdated templates. For example, updates to the database schema were initiated based on a migration script, while the automated tests would verify that the SQL queries still performed backward compatibility. The plugins in Jenkins created pipeline resilience, allowing for re-running in case of short-lived failures, while providing logging details for future diagnosis [5]. Table 1 provides examples of both parameterized test payloads and SQL checks for integration.

Table 1: Example Mapping of XML Test Payloads with SQL Validations

Scenario	XML Parameter Example	SQL Validation Check
Invoice Generation	<Invoice cycle="2023-11" />	SELECT COUNT(*) FROM invoices WHERE cycle='2023-11'
Rate Application	<Rate plan="PREMIUM" />	SELECT amount FROM ledger WHERE plan='PREMIUM'
Adjustment Posting	<Adjustment code="DISC10" />	SELECT * FROM adjustments WHERE code='DISC10'

By establishing this mapping technique, the traced between testing inputs and realizing results was maintained by the framework. Updating XML templates and SQL assertions in the framework facilitated continuous system improvement, which aligned with product developments, so the framework remained valid through releases.

5. Scalability & Optimization

Billing engines process transactions at an enterprise level, so a test framework must achieve production level throughput with no slowdown in speed. Scalability was achieved by architecting the Jenkins pipeline to balance workloads to different executors and utilize contained environments. Through the Dockerized test agents, XML payloads and SQL validation could

run in parallel. The contained environment had its own dedicated redistributable from every execution, such as the database driver, XML libraries and Jenkins agents. Integrating deductibles ensured the same execution was replicated to execute over nodes [5][6].

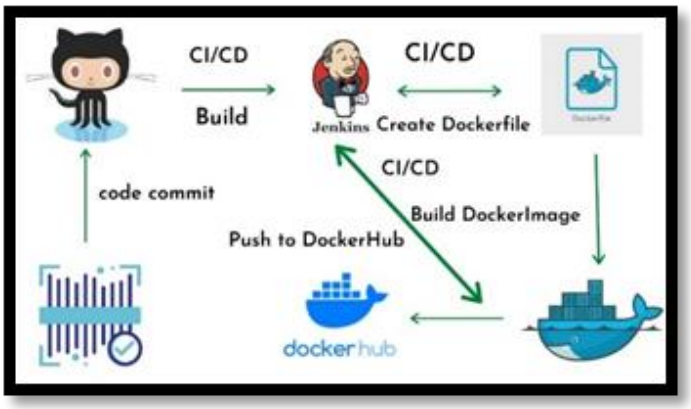

Fig 2: Scalable Containerized Execution of Billing Test Pipelines

The optimization allowed speedy loop feedback on stakeholders where test teams could see to billing inaccuracies within minutes rather than hours. Then it was enterprise-ready because it was open to continual deployments and rapid release cadences.

6. Reporting & Monitoring

The reports provided with a new approach for responding to activity reporting in real-time through stakeholder feedback. Visuals via Jenkins dashboards demonstrated live trends of bold/fail results along with timing the execution each of the scripts and failure confirmation in the environments for deployment. The reporting base operated like PCI or less frequently but likely at least differently different KU applications serving the logs, visuals, functional metrics etc, for whatever environments support data almost real-time.

Table 2: Reporting Metrics before Vs. After CI/CD Integration

Metric	Pre-CI/CD (Manual)	Post-CI/CD (Automated)
Average Execution Time	10 hours	2 hours
Test Coverage	~60% workflows	~95% workflows
Defect Detection Speed	Days	Within same cycle
Reporting Transparency	Limited logs	Full dashboards

Continuous monitoring achieved validation of the processes that were ensured safe and accountable, and we determined if our billing is validated, or if the data above solutions are unvalidated, i.e. invalid.

7. Conclusion

Functional testing of the billing engine completed within CI/CD with parameterized XML payloads and appropriate SQL validation has produced a globally scalable, auditable model. Both the XML transaction triggers and SQL persistence produced a compliant rating, invoicing, and adjustments provided for Charter Communications. Jenkins automation had to replace the original manual dependency and reduced defect/data detection time, therefore providing financial accountability. Containerized environments and dashboards are scalable for enterprise oversight, and are much more transparent. As developing the templates and queries were very labor intensive, this part was unavoidable proactive maintenance model, to validate we're testing. Our next opportunity is AI anomaly detection for continuous regression testing.

Reference

[1] Pratama, M.R. and Kusumo, D.S., 2021, August. Implementation of continuous integration and continuous delivery (ci/cd) on automatic performance testing. In 2021 9th International Conference on Information and Communication Technology (ICoICT) (pp. 230-235). IEEE.

[2] OWASP Foundation, "*XML External Entity (XXE) Prevention Cheat Sheet*," *OWASP Cheat Sheet Series*, 2017 [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html.

[3] Jenkins Project, "*Pipeline as Code — Jenkins User Documentation*," Jenkins.io, 2023 [Online]. Available: https://www.jenkins.io/doc/book/pipeline/

[4] M. Shahin, M. A. Babar and L. Zhu, "*Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*," *IEEE Access*, vol. 5, pp. 3909–3943, 2017, doi: 10.1109/ACCESS.2017.2685629. Available: https://ieeexplore.ieee.org/document/7926944

[5] John, M.M., Olsson, H.H. and Bosch, J., 2021, September. Towards mlops: A framework and maturity model. In 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 1-8). IEEE.

[6] J. Castelein, G. De Vos, R. Hoeben and Y. Vanrompay, "*Search-Based Test Data Generation for SQL Queries*," in *Proc. 33rd ACM Symp. Appl. Comput. (SAC 2018)*, Pau, France, pp. 1543–1550, Apr. 2018, doi: 10.1145/3167132.3167283. Available: https://dl.acm.org/doi/10.1145/3180155.3180202.