Unlocking the Power of CI/CD for Data Pipelines in Distributed Data Warehouses

Hongtao Yang
Google LLC
Mountain View, California, USA

Zhichen Xu
Google LLC
Mountain View, California, USA

Sergey Yudin
Google LLC
Mountain View, California, USA

Andrew Davidson
Google LLC
Mountain View, California, USA

ABSTRACT

Ensuring the reliability of data pipelines is critical for modern data-driven organizations, yet building robust Continuous Integration (CI) in large, distributed data warehouses remains a significant challenge. Complexities arising from distributed ownership, the high cost of replicating production environments, and the rapid evolution of business logic lead to fragile pipelines and costly failures. This paper introduces a novel CI framework designed to conquer these challenges, achieving 94.5% pre-production issue detection in YouTube's data warehouse while dramatically reducing resource consumption. Our key innovation lies in a production-configuration-driven testing methodology, that enables scalable, isolated testing directly within the production environment. This approach reduces testing overhead and ensures high test fidelity. Furthermore, we present a lineage-aware impact analysis framework that automatically propagates data quality checks across distributed pipeline components based on an algebraic dependency model, ensuring data consistency and promoting cross-team collaboration. This production-proven solution provides a practical blueprint for CI/CD in complex, large-scale environments.

PVLDB Reference Format:
Hongtao Yang, Zhichen Xu, Sergey Yudin, and Andrew Davidson. Unlocking the Power of CI/CD for Data Pipelines in Distributed Data Warehouses. PVLDB, 18(12): 4887 - 4895, 2025.
doi:10.14778/3750601.3750613

1 INTRODUCTION

Data pipelines are the lifeblood of modern data-driven organizations, powering everything from machine learning to business intelligence. They are the essential arteries that enable efficient data transformation, data cleansing, and data movement for analytical and operational needs. However, the inherent complexity of these pipelines, particularly within large, distributed data warehouse environments, makes robust Continuous Integration (CI) notoriously difficult. Distributed ownership, the dynamic nature of data (schemas, data quality, business logic), the high cost of production replication, deployment bottlenecks, and the need for cross-team collaboration all contribute to pipeline fragility and increased risk of costly production incidents.

Consider the typical scenario in a large organization: multiple teams own and maintain different segments of a complex data pipeline. Each team might use different technologies, have varying levels of expertise in data quality, and operate on different release cycles. Additionally, some data may be sourced from third parties. This distributed ownership makes it difficult to maintain a consistent understanding of data flow, enforce data quality standards across the entire pipeline, and ensure that changes in one component do not have unintended consequences downstream. This fragmented knowledge often results in data silos, broken dependencies, and a significant increase in the time and effort required to identify and resolve data-related issues.

In the realm of large-scale data pipelines, the establishment and upkeep of dedicated testing environments present a significant economic and logistical challenge. Replicating the complexity of production infrastructure, including extensive data storage, computational resources, and intricate inter-component dependencies, incurs substantial capital expenditure and operational overhead. Maintaining parity between the testing and production environments requires continuous updates and synchronization, consuming valuable engineering resources. Furthermore, the sheer volume of data processed by modern pipelines necessitates equally expansive test datasets, compounding storage costs and data management complexities. The inherent dynamic nature of production pipelines, with frequent updates and evolving dependencies, demands a similarly agile and adaptable testing infrastructure, often resulting in a resource-intensive and error-prone process.

This paper critically examines these CI hurdles and introduces a novel CI framework specifically designed to overcome them. The unique contributions of this paper are:

- **A novel configuration rewriting architecture** that isolates test environments within production infrastructure by rewriting configurations to manage inputs, outputs, and external dependencies, thereby drastically reducing overhead compared to full environment replication.
- **Lineage-Aware Impact Analysis with Automated Data Quality Assurance** which leverages explicit dependency tracking and combines data lineage with automated data quality checks (anomaly detection, schema validation, data integrity) to ensure data quality upon schema or logic changes.
- **Built-in Data Downsampling with User-Supplied Customization** that maintains data representativeness while

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750613

significantly reducing test data volume for faster, more efficient testing. This is achieved through a unique combination of algorithmic stratified sampling and a mechanism for user-supplied, manually crafted downsamplers.
- The framework also supports **pre-submission testing** for fast feedback on change impacts, offering a detailed understanding of global interdependencies through static analysis, dry runs, and isolated test runs.

2 CHALLENGES UNIQUE TO DATA PIPELINE

This section describes the general challenges for introducing CI for data pipelines.

2.1 Data's Inherent Complexity

Unlike code, data is inherently messier and more unpredictable. In a distributed data warehouse environment, collaboration between different teams often hinges on data dependencies. However, the contract for data, especially when exchanged, is often more implicit than explicit. The structure, semantics, and quality expectations may not be clearly defined or enforced, leading to misunderstandings and integration problems. For instance, an upstream team might be unaware of how downstream teams utilize specific columns within a table they produce. Consequently, removing or altering the data distribution of a column could disrupt critical assumptions in downstream business logic. The situation is further complicated by the fact that different teams may have independent product development cycles and release schedules. Schema changes, data quality issues, and evolving business requirements can create cascading effects throughout the data pipeline.

Big data's massive volume, high velocity, and diverse variety further complicate testing [8]. Additionally, defining expected results (oracles) for big data tests can be difficult [2]. The absence of predefined correct outputs for many big data scenarios makes it challenging to verify the accuracy of results. Traditional CI, focusing on code correctness and unit tests, doesn't fully capture the nuances of data validation [3]. Ensuring data correctness, completeness, and consistency across vast datasets necessitates new approaches like data diffing, integration tests, and statistical validation [5, 8]. CI/CD for big data requires specialized tools and techniques to handle these challenges effectively.

2.2 Testing Infrastructure Hard to Replicate

Big data applications often depend on intricate networks of distributed systems, cloud services, and even specialized hardware. Recreating these production environments for testing is a significant challenge. It can be incredibly costly to duplicate the infrastructure, and even then, subtle differences can lead to unreliable test results [1, 10, 11]. This complexity demands dedicated expertise and resources to ensure test environments accurately mirror production, making thorough testing a hurdle in big data development.

2.3 Deployment Bottleneck

Deploying a data pipeline is more than just pushing code. It often involves provisioning infrastructure, configuring data sources, and managing dependencies on external systems. This complexity can hinder rapid iteration and continuous delivery. We need better tools and abstractions to automate these processes and streamline deployments.

2.4 Observability Need

Big data environments are often distributed and complex, which makes identifying and isolating faults difficult and time-consuming. When something goes wrong, pinpointing the root cause can be like finding a needle in a haystack. We need better observability tools – think distributed tracing (like Dapper from OSDI 2010 [12]), data lineage tracking, and real-time performance monitoring – to gain insights into the inner workings of our pipelines.

2.5 Integration Challenge and Collaboration Need

Data pipelines, typically developed and maintained by diverse teams of data engineers, data scientists, and analysts, necessitate rigorous testing of integration points between components [14, 16]. This is crucial to ensure seamless data flow and identify compatibility issues or unexpected data transformations that may arise from integrating independently developed components. Effective CI/CD in this context hinges on collaboration and shared understanding across these disciplines. We need to bridge the gap between these disciplines and foster a culture of shared responsibility for data quality and pipeline reliability.

2.6 Test Data Management

Managing test data for data pipelines presents a unique set of challenges:

- **Volume and Variety**: Data pipelines often handle massive and diverse datasets, making it difficult to create and manage representative test data. Traditional approaches like copying production data may not be feasible or compliant with privacy regulations.
- **Data Evolution**: Data schemas and characteristics can change over time, requiring test data to be updated and maintained accordingly.
- **Environment Fidelity**: Test environments need to accurately reflect production environments, including data sources, infrastructure, and processing capabilities, to ensure reliable testing.
- **Data Privacy and Security**: Test data must be properly anonymized and secured to protect sensitive information and comply with regulations.

2.7 Comparison with Traditional Software CI

Humble and Farley's seminal work, "Continuous Delivery" [8], provides a foundational understanding of CI/CD, emphasizing the importance of automated testing, frequent integration, and rapid feedback cycles. While these principles remain highly relevant for data pipelines, Densmore [6] highlights the unique aspects of this domain that necessitate specialized CI practices. These include the inherent complexities of handling large data volumes, ensuring data validity, and orchestrating intricate processing workflows. To illustrate the key differences between CI for traditional software

Table 1: Traditional CI vs CI for data pipelines.

Feature	Traditional Software CI	Data Pipeline CI
Primary Focus	Code changes and application logic	Code changes, data validation, and pipeline orchestration
Testing Emphasis	Unit tests, integration tests, functional tests	Unit tests for code, data quality tests, integration tests for pipeline stages, end-to-end pipeline validation
Build Artifacts	Executable files, libraries, deployment packages	Data transformations, processed datasets, machine learning models
Deployment	Deploying application code to servers	Deploying data pipelines to processing frameworks cloud services
Version Control	Primarily for code	For code, data schemas, and potentially data itself (using techniques like data versioning or lineage tracking)
Environment Replication	Focus on replicating server environments and dependencies	Focus on replicating data sources, data volumes, and processing infrastructure
Monitoring	Application performance, error rates, resource usage	Data quality metrics, pipeline throughput, data drift, model accuracy
Challenges	Managing dependencies, environment consistency, complex build processes	Data volume and variety, test data management, infrastructure scalability, reproducibility of data transformations

and for data pipelines, consider Table 1, which summarizes the contrasting characteristics across various aspects of the CI process.

Key Differences: Traditional software CI primarily focuses on code correctness through unit tests, while data pipeline CI requires data quality tests, pipeline stage integration tests, and end-to-end data validation to ensure correctness, as well as proper orchestration of the entire pipeline process.

3 A NOVEL CI FRAMEWORK

This section describes a novel framework designed to tackle obstacles in implementing Continuous Integration for data pipelines within distributed data warehouses. This framework has been implemented to facilitate pipeline CI for data production teams within the YouTube data warehouse, empowering data engineers, software developers, and data scientists from different product areas to efficiently conduct development testing and release testing with robust isolation and comprehensive analysis.

3.1 Background

Youtube's data warehouse system is central to YouTube's vast, ever-growing, and dynamic data ecosystem. Each day, the system orchestrates thousands of complex data pipelines, ensuring the timely processing and availability of critical information [4, 18]. These pipelines collectively manage an astonishing volume of data, currently exceeding multiple exabytes and continuing to grow rapidly.

The system serves as the foundational platform for thousands of engineers and data scientists across hundreds of teams within YouTube. These diverse professionals leverage the system daily to access and analyze YouTube data, collaborate on complex projects, develop innovative features, and derive actionable insights that drive product development and user experience improvements.

The scale and dynamic nature of the data necessitate robust CI/CD practices to ensure data reliability, maintainability, and agility in the face of rapid changes. CI/CD in data warehousing is crucial for automating testing, deployment, and validation of data pipelines, which minimizes errors and ensures data quality. This section outlines the critical characteristics of YouTube's data pipeline.

3.1.1 Data Model. The core data model of YouTube's data warehouse is characterized by partitioning and independent versioning. Data is partitioned by time granularity (e.g., daily, weekly), providing an efficient way to manage and process subsets of data. Each partition is versioned independently, enabling incremental updates and independent rollbacks. This approach adds complexity to the data processing pipeline and requires careful management of data dependencies and versions. Successful continuous integration demands a clear understanding of these complex dependencies to avoid conflicts and maintain consistency within the data warehouse.

3.1.2 Rich Dependency Types. The dependencies between upstream base tables and downstream materialized views are explicitly captured through configurations. This system employs a rich set of dependency types. An algebraic approach models these relationships

Table 2: Dependency examples

Example	Descriptions

Job("moving_average_calculator",
partition = DAY,
deps = [table("day_stats", partition = DAY-1),
table("day_stats", partition = DAY-2),
table("day_stats", partition = DAY-3)],
outputs = ["three_day_moving_average"],
cadence = ["input events"])
	Range dependency.

Job("accounting",
partition = WEEK,
deps = [table("daily_spending", partition = WEEK.days())],
outputs = ["weekly_spending"],
cadence = ["1 am every monday"])
	A weekly job rolls up a daily upstream.

between dependencies, and enforces static checks on configurations. This model has a key role in enabling CI capabilities for data pipelines as it ensures the correct flow of data through complex systems.

Table 2 illustrates the dependency algebra with a few examples. The main building block of the algebra is the notion of partition variables which allow time arithmetic over abstract notion of time and time ranges. For example, for the second row in Table 2, the upstream is table "day_stats" with day partition. "DAY" is a partition variable. Job "moving_average_calculator" describes updates to a downstream view "three_day_moving_average". Each "day" partition of "three_day_moving_average" is computed from 3 partitions of "day_stats" for the past 3 days. {YYYYMMDD} partition of "three_day_moving_average" depends on the {YYYYMMDD-1}, {YYYYMMDD-2} and {YYYYMMDD-3} partitions of "day_stats". Similarly, for row 3, "WEEK" is a partition variable, and "WEEK.days()" returns the abstract DAYs in the WEEK. Job "accounting" computes "weekly_spending" from "daily_spending" for the days in each week.

The cadence directive specifies when jobs should be triggered. For example, the "three_day_moving_average" job is triggered each time a new version of the upstream table is available, whereas the accounting job is triggered at 1am on every Monday.

In addition to the dependencies illustrated in Table 2, the system also incorporates proximity constraints that dictate the distance between data and compute resources, ensuring efficient processing by minimizing data movement across geographical regions or availability zones. Moreover, SLO (Service Level Objective) constraints are integrated into the system. These constraints define performance targets for data pipelines (e.g., maximum latency, throughput requirements), which guide job scheduling and resource allocation decisions. By considering these additional constraints, the system can optimize job execution and resource utilization while meeting the specific performance requirements of each data pipeline.

3.1.3 Key data pipeline components. YouTube's data pipelines are defined by the following interacting components provided by the client teams:

- **Job Scheduling Configurations**: These manage the timing and execution of data production jobs, including dependencies illustrated in the previous section, and data and computing proximity constraints.
- **Data Management Configurations**: These control the storage, access, and replication of data within the pipeline.
- **Data Production Job Configurations**: These provide the detailed settings for each data production job, defining resources and specific computations.
- **Business Logic Implementation**: SQL or similar languages handle transformations and data calculations.

These components are highly interconnected, and to ensure data quality and consistency, they require a comprehensive CI framework that can support pre-submit development testing, release A/B testing, integration testing, and other CI features.

3.2 CI Framework Overview

Figure 1 presents the architecture of our CI/CD framework, fully implemented for the YouTube Data Warehouse. Blue boxes denote components from the production environment, while white boxes represent the key elements of our CI/CD framework. The system incorporates the following key components:

- **Test configuration**: The test configuration is designed with the following key principles:
  1. **Targeted Subgraph Testing**: It enables focused testing by allowing specific sections (subgraphs) of the data pipeline to be isolated and tested independently.
  2. **Flexible Parameter Overrides**: It supports fine-grained control over experimental conditions through customizable parameter overrides.
  3. **Minimal Redundancy**: It avoids duplication by leveraging the existing production configuration wherever possible.

  In addition, the test setup supports the integration of specialized utility jobs, such as data downsampling, data diffs, and data quality checks. Given that YouTube data pipelines are dependency-driven, the architecture allows for A/B versions of the pipeline to have different topologies while still

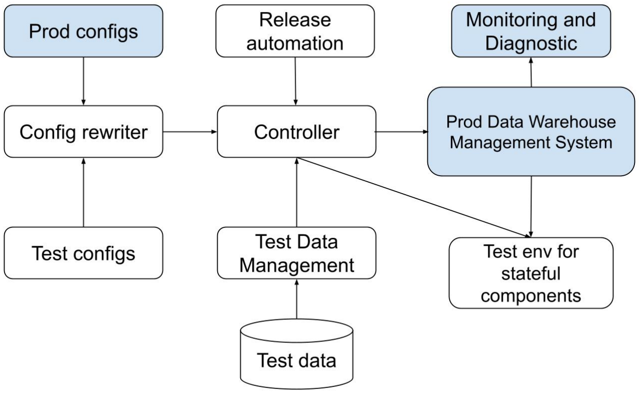

Figure 1: Main Components of our CI Framework Implementation.

processing the same dataset. This design enables robust comparison and makes it possible to test and validate topology changes effectively.

- **Configuration Rewriter**: Enabling Isolated Reproducible Tests:
  The Configuration Rewriter is a crucial component that facilitates isolated and reproducible test runs by transforming job configurations. Unlike ad hoc cloning, it uses the data pipeline's dependency graph to perform targeted and scalable rewriting of configurations within a specified subgraph. This graph-centric approach efficiently propagates changes, impacting only the nodes directly affected by the test scenario.
  A key differentiator is its semantic understanding of data lineage, allowing it to automatically adjust configurations while preserving correctness and ensuring full isolation. This includes intelligently renaming input/output data objects to prevent collisions and dynamically reconfiguring stateful services to route them to isolated instances.
  This component ensures both logical correctness and operational safety by preserving data dependency semantics and minimizing disruption to the production environment. By reusing and adapting production configurations, the system maintains fidelity with live deployments and reduces setup overhead, making it a generalizable framework for safe experimentation in complex data ecosystems.
- **Test Data Management**: The framework provides efficient mechanisms for:

(1) Subsetting and masking production data to create realistic test datasets while protecting sensitive information. This allows developers to test with representative data without exposing sensitive information.
(2) Managing synthetic data tailored to the specific needs of test scenarios. This enables the creation of datasets that cover various scenarios and edge cases, particularly valuable for teams dealing with complicated business logic.
(3) Versioning test data to enable reproducibility and facilitate analysis of different test iterations. This ensures that tests can be consistently re-run with the same data, allowing for accurate comparison of results over time.

- **Controller Module**: This module orchestrates the entire A/B testing process, providing the following functionalities:

(1) Provisioning and managing isolated test environments, including spinning up dedicated instances of stateful components like databases.
(2) Seeding test databases with relevant data, ensuring the test environment accurately reflects the desired conditions.
(3) Scheduling, launching, and monitoring the execution of jobs within the test run.
(4) Analyzing the output of diff jobs and data quality checks to assess the impact of the changes being tested. By systematically comparing intermediate and final results from pipelines A and B to isolate discrepancies and pinpoint the impact of specific changes.
(5) Implementing a "fail-fast" mechanism to automatically terminate the test pipeline if critical errors or unexpected deviations are detected, saving time and resources.
(6) Implementing real-time monitoring and alerting mechanisms to promptly identify and isolate failures, enabling immediate troubleshooting and corrective action.

- **Diagnostics and Reporting**: To enhance understanding and enable efficient problem-solving, the system incorporates tools for:

(1) Gathering comprehensive statistics and metrics from the A/B tests, providing insights into performance, data quality, and potential issues.
(2) Presenting findings in a clear and actionable format, including visualizations and detailed reports.
(3) Facilitating root cause analysis by pinpointing the source of failures or anomalies, enabling rapid debugging and iteration.

3.3 Key Capabilities

This CI framework enables several key capabilities that directly address the challenges outlined in Section 2, significantly improving data pipeline reliability, efficiency, and collaboration. These capabilities are realized through novel architectural components and innovative algorithms, as detailed below:

3.3.1 Lineage-Aware Impact Analysis. This capability leverages the algebraic dependency model (described in Section 4.1.2) to build very fine-grained impact analysis. The framework constructs a dependency graph that accurately represents data lineage, proximity constraints, external dependencies, SLO constraints, to have a complete picture of impact for a change.

To optimize the efficiency of checks, our framework employs a range of techniques spanning static analysis (e.g., to detect dependency breakage, such as reduced retention periods for tables or changes in upstream table permissions) to dry runs (e.g., to validate job configurations) and to actual code execution over downsampled data (for exercising business logic).

These analyses are organized using an extensible framework, where analysis modules are implemented as plug-ins categorized by the type of metadata they require. For instance, dependency checks access the internal representation of the inputs and outputs of affected pipeline components, while modules for business logic analysis access the abstract syntax tree (AST) of the code.

3.3.2 Automated Data Quality Checks. For each affected job, the framework dynamically generates and executes data quality checks, which includes sanity checks such as schema modification detection, data presence, data distribution shift; auto derived and manually crafted data invariants and user-specified SQL assertions.

The generation of data quality checks is guided by metadata about data types, constraints, and historical data distributions to create appropriate validation rules. This algorithm achieves high coverage of potential data quality issues while minimizing the overhead of rule execution. The system performs a dynamic data profiling of the data as part of the CI process that compares the incoming data with historical trends. Furthermore, the framework incorporates anomaly detection algorithms to identify unexpected data patterns that may indicate data quality problems. This enables us to detect data drifts early on in the CI. The system also identifies potential performance degradations by monitoring and comparing to historical performance statistics.

Automated impact analysis is a well known field, however, integrating automated data quality assurance into the analysis is not. This provides unprecedented data quality assurance coverage for complex pipelines, reducing the engineering hours spent in debugging data pipeline issues.

3.3.3 Scalable and Isolated Testing with Production-Configuration-Driven Environment Generation. In the novel configuration rewriting architecture, a user can provide declarative specifications of the scope. For each test run, the framework automatically creates an isolated test environment by cloning and modifying production job configurations. The cloning and modification process is guided by the dependency graph, ensuring that only the necessary configurations are modified, minimizing the overhead and side effects.

For dependencies on stateful components, such as database systems, we leverage Google's testing infrastructure to spin up test instances on demand and leverage the orchestration framework to seed these components with appropriate test data, utilizing our test data management system and downsampling capabilities.

This approach enables high-fidelity testing without the cost and complexity of replicating the entire production environment. Consequently, user onboarding time is significantly reduced from days or weeks for complex pipelines to merely minutes.

While configuration management techniques exist, their application to automated CI/CD testing within the context of complex data pipelines, to our knowledge, is novel. Before using our system, teams would have to spend weeks provisioning and maintaining a separate, costly infrastructure. Now, this is a fully automated process.

3.3.4 Efficient Test Execution through Data Downsampling with Representative Data Preservation. To enable rapid and cost-effective testing, the framework incorporates data downsampling techniques that significantly reduce the volume of test data while preserving key data characteristics. The data downsampling process uses a combination of stratified sampling and data synthesis to create representative test datasets. Stratified sampling preserves the distribution of categorical variables, while data synthesis techniques are used to generate realistic values for numerical variables.

A key challenge in achieving representativeness lies in the intricacies of sophisticated joins and complex business logic inherent in data pipelines. Often, client teams possess the most nuanced understanding of these intricacies and are best positioned to design downsamplers that strike an optimal balance between representativeness and efficiency. Our framework acknowledges this expertise by incorporating a mechanism for user-supplied, manually crafted downsamplers. These custom downsamplers are centrally managed and seamlessly integrated into the testing process, further enhancing the framework's adaptability and effectiveness.

3.3.5 Enhanced Collaboration and Knowledge Sharing through a Centralized Metadata Hub. The framework fosters collaboration through a centralized metadata hub that provides a comprehensive view of the data pipeline, including data lineage, data quality metrics, and job configurations. This centralized hub serves as a single source of truth for all information related to the data pipeline, enabling data engineers, data scientists, and analysts to collaborate more effectively. The metadata hub exposes a set of APIs and datasets that allow users to programmatically query and update metadata, enabling the automation of data governance and data quality tasks.

3.3.6 Access to Production Capabilities. Test pipelines, seamlessly integrated with the production environment, can access a full range of production capabilities, including the UI and monitoring alerts. This enhances the realism and effectiveness of the testing process, ensuring that the behavior observed during testing closely mirrors what can be expected in production.

3.4 Experience and Learnings

The implementation of this framework has significantly enhanced our data production teams' ability to proactively identify and rectify potential issues.

3.4.1 ***Case Study 1****.* : A client team responsible for a data pipeline with over 100 components used the framework's downsampling capability, reducing the test data volume by 99.9% while maintaining representative data distributions. The downsampling technique involved stratified sampling based on key data characteristics, ensuring that the test data accurately reflected the production data. This reduced the testing time from over a day to approximately 1 hour, enabling them to validate new metrics against production data within a strict Service Level Objective (SLO) of 60 minutes, ensuring both pipeline integrity and customer satisfaction.

3.4.2 ***Case Study 2****.* : Teams leveraged the framework to establish shared sandbox environments, reducing the average time to diagnose integration issues by 50%. The shared environment allowed data engineers and data scientists to collaborate more effectively. The sandbox environments provided a common platform for testing and debugging code, reducing the time and effort required to identify and resolve integration issues. The mean-time-to-resolution was reduced from a mean of 1 day to a mean of 4 hours with the new CI/CD framework deployed.

3.4.3 ***Case Study 3***. : Schema changes over hundreds of components were rolled out in O(weeks) instead of O(months) due to the framework's automated impact analysis, which identifies downstream dependencies, and automated data quality checks, which flag inconsistencies and potential data integrity issues. The impact analysis used the algebraic dependency model to identify all downstream components affected by the schema change, allowing engineers to focus their testing efforts on the most critical areas. The automated data quality checks flagged several potential data integrity issues before they reached production, preventing costly data errors. By leveraging the automatic impact analysis, data teams were able to avoid manually updating downstream dependencies, which often leads to human error and inconsistencies.

3.4.4 ***Overall Synthesis and Core Learnings***. The experience gained from designing, implementing, and deploying this CI/CD framework for data pipelines within the YouTube Data Warehouse has yielded several invaluable insights, directly addressing many of the historical pain points and inefficiencies.

Configuration Consistency and Isolation: Reusing production configurations across all environments (development, testing, and staging) critically increases development velocity and reduces inconsistencies. This approach simplifies configuration management and enables teams to isolate and test any data pipeline subgraphs using reliable run isolation through adapted production configurations. This ensures what's tested is truly reflective of production, minimizing unexpected issues.

Integrated Data Quality as a First-Class Citizen: Directly integrating data quality checks into the pipeline workflow significantly improves overall data quality, fosters proactive collaboration between upstream and downstream teams, and enhances observability by offering immediate, actionable insights into potential data quality issues before they propagate. This shifts data quality from an afterthought to an integral part of the development process.

High-Fidelity, Reproducible Testing: The ability to modify production configurations to create isolated, hermetic testing environments has proven transformative. This capability not only reduces infrastructure overhead but, more importantly, improves test fidelity by mirroring the production environment. It provides access to the full range of production capabilities, including the user interface and monitoring alerts, further enhancing realism and effectiveness, and ultimately reducing the dreaded "production-only bug".

Beyond these specific benefits, the modular design of our framework has empowered our users to creatively employ its core functionalities to construct custom solutions tailored to their specific needs. For instance, teams have leveraged the robust diffing and data quality check features to validate the output of extensive backfills against frontfill data before releasing it, showcasing the adaptability and effectiveness of our framework in addressing diverse and evolving data pipeline challenges. Our experience confirms that by proactively addressing the foundational issues of testing, collaboration, and change management, we can achieve unparalleled levels of data integrity and development agility.

3.5 Guiding Principles

This section outlines key guiding principles that we have used to guide our design and implementation of CI for YouTube's Data Warehouse. We believe these principles will be useful for teams facing similar challenges.

3.5.1 Embrace Data's Dynamic Nature: We should embrace the dynamic nature of data — schemas change, data quality fluctuates, and new formats emerge. In a distributed data warehouse environment, where the ownership of data pipeline components are distributed, data contracts are often implicit and not enforced. To prevent skew in data quality over time, teams must implement a combination of robust validation techniques:

- **Schema Validation**: Enforce data types, constraints, and relationships to catch inconsistencies early. For example, use schema validation tools to ensure data types are consistent across all pipeline components and that data relationships are well-defined.
- **SQL Assertions**: Embed data quality checks directly into your pipelines to verify expected conditions. For example, include SQL assertions that ensure the number of null values for a certain column remains below a specific threshold.
- **Anomaly Detection**: Utilize statistical methods or machine learning to identify outliers and unexpected patterns. For example, use anomaly detection techniques to detect when a column distribution changes unexpectedly.
- **Data Profiling**: Regularly analyze data distributions and characteristics to understand changes over time. For example, implement data profiling tools to track shifts in data size over time.
- **Statistical Validation**: Apply statistical tests to verify data integrity and identify potential biases. For example, apply statistical hypothesis tests to validate whether data quality changes are statistically significant.

3.5.2 Comprehensive Testing Strategy: Unit tests are essential, but they're just the beginning. Adopt a multi-layered testing approach:

- **Data Diffing is particularly useful when data contracts and invariants are not explicitly captured**: Compare datasets before and after transformations to pinpoint discrepancies.
- **Integration Tests**: Validate interactions between pipeline components and external systems.
- **End-to-End Tests**: Simulate real-world scenarios to ensure the entire pipeline functions correctly. For example, simulate real-world data flow and usage scenarios to ensure the system functions correctly under different load and usage patterns.

3.5.3 Streamline Deployment Processes: Manual deployments are error-prone and time-consuming. Invest in automation and tooling:

- **Infrastructure as Code (IaC)**: Manage your infrastructure (servers, databases, etc.) using code, enabling version control and reproducibility.
- **Configuration Management**: Centralize and automate the configuration of data sources, pipeline parameters, and environment variables.

- **Continuous Integration/Continuous Deployment**: Implement pipelines to automatically build, test, and deploy your data pipelines.
- **Dependency Management**: Use tools like package managers to track and manage dependencies, ensuring consistent environments across development, testing, and production.

3.5.4 Prioritize Observability: Gain deep insights into your pipeline's behavior to detect issues early and resolve them quickly:

- **Distributed Tracing**: Track data flow across pipeline stages to pinpoint bottlenecks and performance issues.
- **Data Lineage Tracking**: Understand the origin and transformations of your data to ensure accuracy and compliance.
- **Real-Time Monitoring**: Collect and visualize metrics on data throughput, latency, error rates, and resource utilization.
- **Alerting and Anomaly Detection**: Set up alerts to notify you of critical issues and leverage machine learning to identify unusual patterns.
- **Log Aggregation and Analysis**: Centralize logs from various pipeline components for easier troubleshooting and root cause analysis.

3.5.5 Foster Collaboration: Data pipelines are a team effort. Break down silos and promote shared ownership:

- **Shared metadata**: Implement common metadata and reporting mechanisms that span across teams.
- **Shared Responsibility**: Encourage collaboration between data engineers, data scientists, and analysts to ensure data quality and pipeline reliability.

3.5.6 Proactive Test Data Management: Test data is crucial for ensuring pipeline quality, but it can be challenging to manage:

- **Evolving Test Data**: Develop strategies to keep test data in sync with schema changes and production data distributions.
- **Synthetic Data Generation**: Use techniques like data masking and synthetic data generation to create realistic test data while protecting sensitive information.
- **Data Subsetting**: Extract representative subsets of production data for testing, balancing realism with privacy and performance considerations.
- **Test Data Versioning**: Maintain version control for test data to enable reproducibility and traceability.

Data pipelines are inherently dynamic and require continuous monitoring, improvement, and adaptation to meet the evolving needs of the organization. Adherence to these guiding principles facilitates the development of robust, scalable, and maintainable data pipelines that consistently deliver high-quality data.

4 RELATED WORK

This work builds upon established software engineering and release practices, drawing inspiration from Nygard [11] and Humble et al. [8].These works emphasize the importance of designing systems for production readiness, incorporating strategies for stability, and automating the build, test, and deployment processes to achieve reliable software releases. Several notable efforts have explored CI/CD for data pipelines and related areas. For example, Caveness [3] describes TFDV, a tool that helps to analyze and validate data for machine learning. It identifies inconsistencies, schema issues, and data drift within ML pipelines. Vadavalasa [17] proposed a CI/CD framework for machine learning, encompassing data ingestion, feature engineering, model training, evaluation, and deployment, and highlighted tools like Git, Jenkins, and TensorFlow to improve automation and reliability in ML pipelines. Kanstrén [9] explores the challenges and best practices for testing data-intensive systems. It shares real-world experiences and lessons learned, highlighting the need for comprehensive testing strategies that cover functional correctness,performance, and data integrity in such systems. Additionally, researchers like Zhang et al. [19], Gao et al. [7], and Staegemann et al.[13–16] have conducted surveys and reviews of quality assurance techniques and challenges in Big Data applications, providing valuable insights into the current landscape of CIand CD practices for Big Data pipelines. While these existing works offer valuable insights and solutions, our framework distinguishes itself by providing a more holistic and user-centric approach to CI in data pipeline management. We address a broader range of challenges, including data complexity, testing intricacies, deployment bottlenecks, observability needs, and collaboration aspects. Furthermore, our framework emphasizes ease of use, efficiency, and seamless integration with existing data infrastructure, making it more accessible and adaptable to diverse data environments. By combining the strengths of previous research with our own innovations, we have developed a comprehensive and practical solution that empowers YouTube to build and manage robust, reliable, and high-quality data pipelines.

5 CONCLUSIONS

This paper examined the key challenges of implementing Continuous Integration (CI) for data pipelines, particularly in the context of large-scale, distributed data warehouses. Our framework prioritizes user-friendliness and minimizes the infrastructure setup and maintenance overhead required to support various testing environments and phases. By leveraging and adapting production configurations, and integrating with Google's robust testing infrastructure, we ensure that tests accurately reflect the actual production environment. This approach promotes consistency, reduces manual effort, and increases the reliability of test results. Furthermore, our framework fosters collaboration and enhances observability by incorporating data quality checks as integral components of the CI/CD pipeline. These checks serve as a shared understanding of key data warehouse table properties, promoting a unified approach to data quality across different teams. Deployed within YouTube's data warehouse, our framework has demonstrably improved data pipeline reliability, preventing 94.5% of production issues and reducing resource consumption by as much as 10,000x. This framework offers a practical solution for efficient CI/CD in complex data environments, accelerating development and improving data reliability across our data warehouse infrastructure, and we believe it can be applied to many other organizations facing similar challenges. By providing a comprehensive and automated solution for CI in data pipelines, our framework empowers data-driven organizations to build and maintain reliable, scalable, and high-quality data pipelines that can support their most critical business needs. Future work will focus on leveraging advanced machine learning techniques to automate the generation of even more sophisticated and relevant data quality checks, and developing more dynamic and adaptive test data sampling strategies that can automatically adjust to evolving data characteristics. We also plan to explore the integration of CI/CD frameworks with emerging data governance standards and practices to further enhance data quality and compliance in large-scale data pipelines.

ACKNOWLEDGMENTS

We would like to thank Mike Lin, Chioma Ezete, and Malcolm Milanovich for their invaluable contributions to this project. We also thank Fred Faber for his support and insightful comments and suggestions. Finally, thank you to our customers who suggested core features and capabilities.

REFERENCES

[1] Iram Arshad, Saeed Hamood Alsamhi, and Wasif Afzal. 2021. Big Data testing techniques: taxonomy, challenges and future trends. *arXiv preprint arXiv:2111.02853* (2021).

[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[3] Emily Caveness, Paul Suganthan GC, Zhuo Peng, Neoklis Polyzotis, Sudip Roy, and Martin Zinkevich. 2020. Tensorflow data validation: Data analysis and validation in continuous ml pipelines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2793–2796.

[4] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, et al. 2019. Procella: Unifying serving and analytical data at YouTube. PVLDB 12 (12)(2019), 2022–2034.

[5] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.

[6] James Densmore. 2021. *Data pipelines pocket reference.* O'Reilly Media.

[7] Jerry Gao, Chunli Xie, and Chuanqi Tao. 2016. Big data validation and quality assurance–issuses, challenges, and needs. In *2016 IEEE symposium on service-oriented system engineering (SOSE)*. IEEE, 433–441.

[8] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education.

[9] Teemu Kanstrén. 2017. Experiences in testing and analysing data intensive systems. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 589–590.

[10] Neelum Nasir, Salma Imtiaz, Saima Imtiaz, and Muhammad Nabeel. 2022. Testing Framework for Big Data: A Case Study of Telecom Sector of Pakistan. (2022).

[11] Michael Nygard. 2018. Release it!: design and deploy production-ready software. (2018).

[12] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[13] Daniel Staegemann, Pouya Ataei, Erik Lautenschläger, Matthias Pohl, Christian Haertel, Christian Daase, Matthias Volk, Mohammad Abdallah, and Klaus Turowski. 2024. An Overview on Testing Big Data Applications. In *International Congress on Information and Communication Technology*. Springer, 303–315.

[14] Daniel Staegemann, Johannes Hintsch, and Klaus Turowski. 2019. Testing in Big Data: An Architecture Pattern for a Development Environment for Innovative, Integrated and Robust Applications. (2019).

[15] Daniel Staegemann, Matthias Volk, Mohammad Abdallah, and Klaus Turowski. 2023. On the Challenges of Applying Test Driven Development to the Engineering of Big Data Applications.. In *ICSBT*. 129–135.

[16] Daniel Staegemann, Matthias Volk, Abdulrahman Nahhas, Mohammad Abdallah, and Klaus Turowski. 2019. Exploring the specificities and challenges of testing big data systems. In *2019 15th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*. IEEE, 289–295.

[17] Ram Mohan Vadavalasa. 2020. End to end CI/CD pipeline for Machine Learning. *International Journal of Advance Research, Ideas and Innovation in Technology* 6 (2020), 906–913.

[18] Zhichen Xu, Ying Gao, and Andrew Davidson. 2023. Keep Your Distributed Data Warehouse Consistent at a Minimal Cost. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[19] Pengcheng Zhang, Xuewu Zhou, Wenrui Li, and Jerry Gao. 2017. A survey on quality assurance techniques for big data applications. In *2017 IEEE Third International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 313–319.