# Teracloud Streams Automated Cross-DC Failover Testing Framework: Implementation and User Guide

Document Version: 1.0

Date: October 26, 2023

Framework Version: 1.0 (Targeting Teracloud Streams 7.2.0 and later)

**Table of Contents**

1. Introduction

\* 1.1. Purpose of the Document

\* 1.2. Overview of the Automated Cross-DC Failover Testing Framework

\* 1.3. Target Audience

\* 1.4. Key Technologies and Teracloud Streams Components Involved

\* 1.5. Document Conventions

2. Framework Architecture and Design

\* 2.1. High-Level Architecture Diagram

\* 2.2. Core Components of the Python Automation Framework

\* 2.3. Key Design Principles and Decisions

\* 2.4. Interaction Model with Teracloud Streams

\* 2.5. Assumptions Made During Design and Implementation

3. Teracloud Streams Interface Deep Dive (Based on Research & Assumptions)

\* 3.1. REST Management API (Teracloud Streams 7.2+)

\* 3.2. Data Exchange Service (Teracloud Streams 7.2+)

\* 3.3. Cross-DC Failover Toolkit (Teracloud Streams 7.2+)

4. Implementation Guide (For Senior Engineers & LLM)

\* 4.1. Development Environment Setup

\* 4.2. Core Python Modules and Libraries

\* 4.3. Implementing Test Lifecycle Automation

\* 4.4. Code Structure and Modularity

\* 4.5. Generating Test Reports (JUnit XML)

5. User Guide (For Deployment and Operations)

\* 5.1. Prerequisites and System Requirements

\* 5.2. Deployment of the Failover Testing Framework

\* 5.3. Configuring Test Scenarios

\* 5.4. Running Failover Tests

\* 5.5. Interpreting Test Results

\* 5.6. Troubleshooting Common Issues

6. Advanced Topics & Extensibility

\* 6.1. Integrating with CI/CD Pipelines

\* 6.2. Customizing Probes and Actions

\* 6.3. Extending the Framework for New Test Cases

7. Points Requiring Clarification/Confirmation from Teracloud SMEs

\* 7.1. REST Management API (Teracloud Streams 7.2+)

\* 7.2. Data Exchange Service (Teracloud Streams 7.2+)

\* 7.3. Cross-DC Failover Toolkit (com.teracloud.streams.crossdcfailover)

\* 7.4. Python Client/SDK

\* 7.5. Monitoring and Metrics

8. Appendix

\* 8.1. Example Configuration Files

\* 8.2. Sample Teracloud Streams Application (Conceptual)

\* 8.3. Glossary of Terms

\* 8.4. References

## 1. Introduction

### 1.1. Purpose of the Document

This document is based on the official Teracloud Streams documentation for version 7.2.0, with special attention to the REST Management API, Data Exchange service, and Cross-DC Failover Toolkit features. Section 7 lists specific points where additional clarification from Teracloud Subject Matter Experts (SMEs) would enhance our implementation. As a living document, it will continue to evolve with updates and refinements as we gain more experience with Teracloud Streams environments and gather feedback from users.

It is intended for two primary audiences:

* **Senior Software Engineers and LLM-based code generation tools:** This audience will utilize the document for understanding the framework's internal workings, API interactions, code structure, and extension points. The level of detail provided aims to be sufficient for direct use by Large Language Model (LLM)-based code generation tools, facilitating rapid development and extension of the framework.
* **Operational Users (e.g., Site Reliability Engineers (SREs), Quality Assurance (QA) Teams):** This audience will use the document as a comprehensive manual for deploying the framework, configuring test environments, executing predefined or custom test scenarios, and interpreting the generated reports and logs.

The dual-purpose nature of this document necessitates a balanced approach. Technical sections are designed to be exhaustive, providing granular details on components, methods, data structures, and algorithms suitable for developers and LLMs. User-facing sections, conversely, prioritize clarity and step-by-step instructions, abstracting away internal complexities to ensure ease of use.

### 1.2. Overview of the Automated Cross-DC Failover Testing Framework

The Automated Cross-DC Failover Testing Framework is a Python-based solution designed to rigorously and automatically test cross-datacenter (DC) failover scenarios for applications deployed on Teracloud Streams. The core objective of this framework is to provide a reliable, repeatable, and automated method to verify the resilience and recovery mechanisms of Teracloud Streams applications in the event of a simulated DC outage or a controlled DC switchover.

Key functionalities of the framework include:

* **Automated Application Deployment:** Programmatic deployment of Teracloud Streams Application Bundle (SAB) files to the target Streams instance.
* **State Management and Data Verification:** Injection of deterministic and identifiable test data into applications before failover, and retrieval and validation of this data post-failover to assess data integrity and consistency.
* **Fault Injection:** Simulation of various failure conditions, such as network partitions between DCs, increased network latency, packet loss, or targeted service disruptions.
* **Automated Failover Orchestration:** Triggering or observing application failover processes, particularly those managed by the Teracloud Streams Cross-DC Failover Toolkit.
* **Monitoring and Metrics Collection:** Gathering relevant performance metrics and application health indicators during test execution.
* **Results Analysis and Reporting:** Generation of comprehensive test reports detailing the outcomes, including success/failure status, recovery times, and any observed anomalies.

The framework is architected to interact primarily with Teracloud Streams Version 7.2.0 and later, as this version introduces critical features like the enhanced REST Management API, the Data Exchange service, and the Cross-DC Failover Toolkit, which are foundational to the framework's design.9

### 1.3. Target Audience

The framework and this accompanying document are tailored to meet the needs of distinct groups:

* **Senior Software Engineers & LLM Tools:** This group requires a deep understanding of the framework's architecture, the nuances of its interaction with Teracloud Streams APIs, the internal logic of its components, and the design patterns employed. The detailed specifications, class and module descriptions, and API interaction protocols provided are intended to be directly consumable by LLM-based code generation systems to accelerate development, create extensions, or generate client code for new test scenarios. For human engineers, this detail supports maintenance, debugging, and advanced customization.
* **Operational Users (e.g., SREs, QA Teams, Operations Staff):** This group focuses on the practical application of the framework. They need clear, concise instructions for deploying the framework in their test environments, configuring it for specific Teracloud Streams applications and infrastructure, defining and executing failover test scenarios, and interpreting the resulting data to assess system resilience. The user-facing sections of this guide are written to be accessible without requiring an in-depth knowledge of the framework's Python codebase.

### 1.4. Key Technologies and Teracloud Streams Components Involved

The framework leverages a combination of Python technologies and specific Teracloud Streams components:

* **Python:** The core programming language for the automation framework. Version 3.9 or later is recommended, aligning with dependencies like tcconfig 11 and chaostoolkit.12
* **Teracloud Streams Version 7.2.0 (and later):** The target platform for which failover testing is performed. This version is crucial due to the introduction of key enabling features 9:
  + **Enhanced REST Management API:** Provides programmatic control over Teracloud Streams domains, instances, and jobs, essential for application lifecycle management (deployment, start, stop, status) during tests.
  + **Data Exchange Feature:** Enables the injection of test data into and retrieval of state from running Streams applications via REST APIs, critical for pre-failover setup and post-failover validation.
  + **Cross-DC Failover Toolkit (com.teracloud.streams.crossdcfailover):** A toolkit designed to facilitate application-level failover across data centers, enabling disaster recovery and business continuity. The framework will test applications utilizing this toolkit's capabilities for maintaining state consistency and managing automatic failover between primary and secondary data centers.
* **Python requests Library:** The standard Python HTTP client library used for all interactions with the Teracloud Streams REST APIs.13
* **Python paramiko Library:** Used for SSH communication to execute remote commands on Linux nodes for fault injection purposes.15
* **Network Fault Injection Tools:**
  + **tcconfig:** A Python wrapper for the Linux tc (traffic control) utility, used to simulate WAN conditions like latency, packet loss, and bandwidth restrictions.11
  + **python-iptables or direct iptables commands:** Used to create network partitions by manipulating firewall rules on Linux systems.15
* **Configuration Management (PyYAML):** For loading and parsing framework and test scenario configurations from YAML files.20
* **Reporting (junit-xml or pytest):** For generating test reports in JUnit XML format, facilitating integration with CI/CD systems.22
* **Monitoring Clients:**
  + **Prometheus Python Client (prometheus-api-client):** If Streams 7.2 exposes metrics in Prometheus format, this client will be used for collection.15
  + **jmxquery:** If Streams 7.2 relies on JMX for detailed metrics, this Python library will be used to query JMX MBeans.15
* **Logging (logging module):** Python's built-in logging module for comprehensive event tracking within the framework.26

The tight coupling with Teracloud Streams 7.2.0 implies that the framework, as designed, may not be compatible with older versions of Streams if the aforementioned key features are absent or significantly different. This focus ensures the framework leverages the latest capabilities for robust failover testing.

### 1.5. Document Conventions

To ensure clarity and consistency throughout this document, the following conventions are used:

* Monospace: Used for code examples, file paths, commands, API endpoints, configuration parameters, and JSON/YAML keys.
* **Bold**: Used for emphasis, key terms, and UI elements where applicable.
* *[Placeholder]*: Italicized text within angle brackets indicates a placeholder that should be replaced with a specific value.
* Snippet IDs (e.g., 9) are used to reference supporting information or data sources compiled during the research phase. These are for traceability and are not part of the framework's operational output.

## 2. Framework Architecture and Design

This section details the architecture of the Automated Cross-DC Failover Testing Framework, its core components, design principles, and interaction model with Teracloud Streams.

### 2.1. High-Level Architecture Diagram

*(A textual description of the diagram is provided here. In a final document, this would be a graphical diagram.)*

The architecture comprises the **Python Automation Framework** as the central orchestrator, typically running on a dedicated host or a CI/CD agent. This framework interacts with two **Teracloud Streams Data Centers (DC1 - Primary, DC2 - Secondary)**.

**Key Interactions:**

1. **Control Plane (Framework to Streams):**
   * The framework communicates with the **Teracloud Streams REST Management API** in both DC1 and DC2. This is used for deploying Streams Application Bundles (SABs), starting/stopping jobs, querying job/instance status, and retrieving logs/metrics.
   * The framework interacts with the **Teracloud Streams Data Exchange Service API** in both DCs to inject test data into running applications and retrieve state for validation.
2. **Fault Injection (Framework to Network/Nodes):**
   * The framework's **Fault Injection Module** uses SSH (via paramiko) to connect to designated **Network Manipulation Nodes** (which could be routers, dedicated VMs, or the Streams hosts themselves if permissible and necessary).
   * On these nodes, it executes commands using tc (via tcconfig) or iptables (via python-iptables or direct commands) to simulate network degradation (latency, loss, bandwidth limits) or partitions between DC1 and DC2, or between clients and a DC.
3. **Monitoring (Streams to Framework/Monitoring System):**
   * The framework's **Monitoring and Metrics Collector** queries the Streams REST API for metrics.
   * If Streams exposes Prometheus endpoints, the collector scrapes these.
   * If Streams uses JMX, the collector uses jmxquery (potentially via an intermediary agent if direct access is restricted).
4. **Application Behavior (Within Streams):**
   * The **Streams Application under Test** (running as a job) is deployed in DC1 (initially active) and is designed using the **Cross-DC Failover Toolkit**.
   * This toolkit enables the application to be DC-aware, potentially replicate state, and manage its failover to DC2 when DC1 becomes unavailable or a switchover is triggered.

**Flow Example (Simplified Failover Test):**

1. Framework deploys SAB to DC1 via REST API.
2. Framework injects test data into the DC1 application via Data Exchange API.
3. Framework initiates fault injection (e.g., network partition between DC1 and DC2).
4. Framework monitors the application and Streams environment. The Cross-DC Failover Toolkit within the application detects the issue and initiates failover to DC2 (or the framework triggers it if an API exists).
5. Framework verifies the application is running in DC2.
6. Framework retrieves data/state from the DC2 application via Data Exchange API and validates consistency.
7. Framework removes fault injection.
8. Framework collects final metrics/logs and generates a report.

This architecture allows for comprehensive testing by controlling the application lifecycle, manipulating network conditions, and observing the system's response, particularly the efficacy of the Cross-DC Failover Toolkit.

### 2.2. Core Components of the Python Automation Framework

The framework is composed of several modular Python components, each with specific responsibilities. This modularity facilitates development, maintenance, and future extensions.

**Table 2.2.1: Core Framework Components and Responsibilities**

|  |  |  |
| --- | --- | --- |
| **Component Name** | **Primary Responsibility** | **Key Technologies/Libraries Used** |
| **Orchestration Engine** | Manages the overall test lifecycle: setup, execution of test steps as per scenario, teardown, and invoking reporting. Coordinates other components. | Python (core logic) |
| **Teracloud Streams API Client** | Handles all HTTP/S communication with the Teracloud Streams REST Management API and Data Exchange Service. Encapsulates API call logic. | requests |
| **Test Scenario Executor** | Interprets test scenario definitions (from YAML). Executes individual test steps like application deployment, data injection, fault induction, etc. | Python, PyYAML |
| **Fault Injection Module** | Simulates network faults (partitions, latency, loss) and potentially service disruptions. | paramiko (for SSH), tcconfig (for tc), python-iptables or direct iptables calls |
| **Monitoring and Metrics Collector** | Gathers performance metrics, health status, and failover indicators from Teracloud Streams instances and applications. This component will implement a flexible approach to detect failover events based on multiple observation methods: - Direct observation of application and instance metrics via Streams REST APIs - Pattern recognition in application and framework logs - Tracking data flow changes between primary and secondary DCs - Monitoring for specific events or state changes reported by the Cross-DC Failover Toolkit (once we better understand its observability interfaces) This adaptive design allows the framework to evolve as we gain more insight into the toolkit's behavior patterns and logging conventions. | requests (for API-based metrics), prometheus-api-client (if Prometheus used), jmxquery (if JMX used) |
| **Data Generation & Validation Module** | Creates deterministic, identifiable test data for injection. Validates data consistency and integrity after failover. | Python (custom data generation logic) |
| **Configuration Manager** | Loads, validates, and provides access to framework-level and test-scenario-level configurations. | PyYAML |
| **Reporting Module** | Generates summary test reports, typically in JUnit XML format for CI/CD integration. Aggregates results from test execution. | junit-xml or pytest's reporting capabilities |
| **Logging Module** | Provides comprehensive and structured logging for all framework operations. | Python's built-in logging module |

The successful operation of these components, particularly the API Client, Test Scenario Executor, and Data Generation & Validation Module, is intrinsically linked to the actual capabilities and interfaces exposed by Teracloud Streams 7.2. Any discrepancies between assumed and actual API behavior or feature limitations will necessitate adjustments in these components. For instance, if the REST API does not provide a direct call to stop a specific processing element (PE) within a job, the framework cannot automate that precise action and may need to rely on broader job cancellation, impacting test granularity. Similarly, the throughput characteristics of the Data Exchange service will dictate its suitability for testing applications with substantial state.

### 2.3. Key Design Principles and Decisions

The design of the framework is guided by several core software engineering principles to ensure robustness, maintainability, and usability:

* **Modularity:** Each component listed in Table 2.2.1 is designed to be as self-contained as possible, with well-defined responsibilities and interfaces. This loose coupling allows for independent development, testing, and modification of components without adversely affecting others. For example, changes to the Teracloud Streams API version could be largely isolated within the Teracloud Streams API Client module.
* **Extensibility:** The framework is designed to be adaptable to new requirements. This includes adding new types of test scenarios, different fault injection techniques, or more sophisticated validation checks. The use of configuration files for test definitions and a modular structure for actions and probes (inspired by systems like Chaos Toolkit 12) supports this.
* **Idempotency:** Where feasible, operations performed by the framework should be idempotent. This means that if an operation is executed multiple times with the same parameters, the outcome should be the same as if it were executed only once. For instance, deploying a specific version of a Streams application should result in that application being in the desired state, regardless of whether it was already deployed or not (the API might handle this by rejecting a duplicate deployment or by updating an existing one). This simplifies error recovery and makes the framework more resilient to transient issues.
* **Configuration-driven:** All aspects of the framework are externalized into configuration files (primarily YAML). This includes:
  + Target environment details (DC endpoints, Streams instance identifiers)
  + Test scenario definitions and parameters
  + Fault injection specifications
  + Validation criteria and thresholds
  + Framework operational parameters

This configuration-driven approach provides several advantages:

* + Allows non-developers to create and modify test scenarios
  + Facilitates testing different configurations of the Cross-DC Failover Toolkit
  + Enables quick adaptation as we learn more about the toolkit's specific capabilities
  + Makes the framework more maintainable by separating test logic from test scenarios
  + Supports integration with CI/CD pipelines through parameterized test execution
* **Statelessness (of the framework):** The automation framework itself aims to be stateless. It does not maintain persistent information about past test runs or environment states between executions. All necessary state is either derived from the Teracloud Streams environment at runtime (via API calls) or defined in the input configuration files for a given test run. This simplifies deployment and execution, as there's no complex framework state to manage or restore.
* **Leverage Official APIs:** The primary method for interacting with Teracloud Streams is through its official Version 7.2 REST APIs and the com.teracloud.streams.crossdcfailover toolkit.9 This approach ensures compatibility and leverages supported interfaces, reducing the risk associated with unofficial or internal access methods. The move by Teracloud Streams towards comprehensive REST APIs for management and data interaction 9 is a positive trend that greatly simplifies the development of such automation tools compared to reliance on CLI parsing or older, potentially proprietary interfaces. Many streamtool commands are noted as removed in version 7.2 9, reinforcing the API-first approach.

The choice of Python, along with standard, well-maintained libraries like requests for HTTP communication 13 and paramiko for SSH 15, contributes to development efficiency and broader accessibility due to a large talent pool. However, it's important to note that while the Python framework itself is largely cross-platform, the fault injection capabilities relying on tc (traffic control) and iptables are specific to Linux environments.11 This implies that the nodes upon which network faults are simulated must be Linux-based. Given that Teracloud Streams 7.2 targets Red Hat Enterprise Linux (RHEL) 8 and 9 9, this is consistent for simulating faults directly affecting the Streams hosts or the network paths between them.

### 2.4. Interaction Model with Teracloud Streams

The framework's interaction with Teracloud Streams is multifaceted, primarily relying on the new and enhanced capabilities introduced in version 7.2.0.

* 2.4.1. Leveraging the REST Management API  
  The framework will employ a Python client, built using the requests library 13, to communicate with the Teracloud Streams REST Management API. This API, enhanced in Streams 7.2 9, is the cornerstone for programmatic control over the Streams environment. Interactions are anticipated to cover the full lifecycle of test applications:
  + **Application Deployment:** Submitting Streams Application Bundle (SAB) files to a specified Streams instance.
  + **Job Management:** Starting, stopping, and canceling deployed Streams jobs.
  + **Status Monitoring:** Querying the status and health of Streams instances, individual jobs, and potentially specific Processing Elements (PEs) within jobs.
  + **Metrics Retrieval:** Fetching performance metrics and operational statistics from jobs and instances.
  + **Log Collection:** Accessing and retrieving application logs generated by Streams jobs. The older IBM Streams documentation sometimes referred to a "Web management service (SWS)" in conjunction with REST APIs 29, and messages from the "Application manager service (SAM)" hinted at API-driven job submission and cancellation.30 It is assumed that the enhanced REST API in Teracloud Streams 7.2 consolidates and expands upon these capabilities.
* 2.4.2. Utilizing the Data Exchange Feature  
  The Data Exchange feature, also new in Streams 7.2 9, provides a RESTful interface for directly injecting data into and retrieving data from running Streams applications. This is pivotal for failover testing:
  + **Pre-failover State Seeding:** The framework will use this service to inject known, uniquely identifiable test data into the Streams application running in the primary DC. This establishes a baseline state.
  + **Post-failover State Verification:** After a failover event (simulated or actual), the framework will use the Data Exchange service to retrieve data from the application now running in the active DC (which could be the secondary DC). This retrieved data is then compared against the initially injected data (or expected transformed data) to verify data integrity, consistency, and the correctness of state recovery. Teracloud Streams documentation indicates that "endpoint operators" must be included in the SPL application to enable this REST-based data interaction.32 The specifics of these operators are crucial for both the Streams application developer (to include them) and the framework developer (to interact with them).
* 2.4.3. Integrating with the Cross-DC Failover Toolkit  
  The framework is designed to test Streams applications that utilize the com.teracloud.streams.crossdcfailover toolkit. This toolkit provides "application-level failover across two data centers enabling Disaster Recovery (DR) and Business Continuity (BC)." The toolkit's approach means that the Streams application code, using the toolkit's operators and functions, becomes an active participant in the failover process. This differs from purely infrastructure-level failover where the application remains passive during transitions between data centers.  
  The framework's interaction with this toolkit may involve:
  + **Configuration:** Setting toolkit-specific parameters during application submission, if these are exposed via the REST Management API.
  + **Monitoring:** Observing the status of the toolkit or the application's failover state, potentially through specific metrics or logs exposed by the toolkit or the application.
  + **Triggering (Conditional):** If the toolkit allows for external (e.g., API-driven) triggering of failover or failback for testing purposes, the framework would leverage this. This capability is currently an assumption. Testing "application-level failover" implies that the framework must validate not only that the application is running in the secondary DC but also that its internal state, data processing logic, and connections to external resources are correctly re-established and functioning as per the toolkit's design. This requires a deeper level of observation than simply checking process availability.

### 2.5. Assumptions Made During Design and Implementation

Given the current limitations in accessing detailed Teracloud Streams 7.2 documentation, the design and subsequent implementation of this framework are based on the following key assumptions:

1. **Target Environment:** Teracloud Streams 7.2.0 or a later compatible version is the exclusive target environment. The framework relies on features explicitly mentioned as new or enhanced in this release.9
2. **REST Management API Completeness:** The enhanced REST Management API provides all necessary endpoints and functionalities for comprehensive application lifecycle management (deploy, start, stop, status), instance and domain monitoring, and retrieval of logs and metrics required by the framework. (This is a major area requiring confirmation due to inaccessible documentation 1).
3. **Data Exchange Service Functionality:** The Data Exchange feature supports RESTful injection and retrieval of data using a common format like JSON. It is assumed to offer sufficient performance (throughput and latency) for the volume of test data required and allows for interaction with named "endpoint operators" within the SPL application. (Requires confirmation 3).
4. **Cross-DC Failover Toolkit Behavior:** The Cross-DC Failover Toolkit provides mechanisms for application-level failover that can be reliably observed by the automation framework through application logs, metrics, and potentially specific toolkit-provided indicators. Based on the toolkit's purpose of enabling DR and BC capabilities, we assume it exposes sufficient information about its operational state to allow our framework to detect failover events, validate successful transitions, and verify state consistency. While the exact interfaces for observing and potentially triggering failover events will need direct confirmation from Teracloud documentation or SMEs, our framework will be designed to accommodate various observation methods.
5. **API Authentication:** Authentication to all Teracloud Streams REST APIs (Management and Data Exchange) is achievable using a standard mechanism compatible with the Python requests library, such as an API key passed in an HTTP header (e.g., X-API-Key or Authorization: Bearer <token>). The specific header and token/key generation method are pending clarification.34
6. **Network and Permissions:** The framework's execution environment has the necessary network connectivity to the Teracloud Streams management interfaces in all relevant data centers. Appropriate permissions and credentials will be available for API access and for SSH access to nodes where fault injection tools are executed.
7. **SSH Access for Fault Injection:** SSH access (using paramiko) is available and permitted to the Linux-based nodes where network fault injection tools (tc, iptables) will be run. While Teracloud Streams 7.2 release notes mention that "Streams will no longer use SSH to communicate with other resources" for *internal domain management* 9, this assumption pertains to the *framework's external need* to manipulate the OS-level network environment of test machines or VMs, not for managing Streams itself via SSH.
8. **Python Client SDK:** It is assumed that no official, feature-complete Teracloud Streams Python client SDK for version 7.2 exists or is mandated for use, thus justifying the development of a custom API client wrapper using requests. If such an SDK exists and is suitable, its use would be prioritized. The IBM Streams Python SDK streamsx.rest 15 is for a predecessor product and may not be compatible or cover the new 7.2 features.

These assumptions will be revisited and updated as more information becomes available from Teracloud SMEs. Section 7 explicitly lists these as points requiring clarification.

## 3. Teracloud Streams Interface Deep Dive (Based on Research & Assumptions)

This section delves into the specifics of the Teracloud Streams interfaces that the automation framework will interact with. Due to the previously mentioned inaccessibility of detailed API documentation 1, much of this section is based on information from release notes 9, general REST API principles, and logical inferences. **All API endpoint details, authentication mechanisms, and specific parameter names are hypothetical and require confirmation from Teracloud SMEs.**

### 3.1. REST Management API (Teracloud Streams 7.2+)

* 3.1.1. Overview and Capabilities  
  The Teracloud Streams 7.2.0 release notes announce an "enhanced REST API" for managing domains, instances, and jobs.9 This API is expected to be the primary programmatic interface for the automation framework to orchestrate test scenarios. This includes deploying Streams applications, controlling their lifecycle (start, stop), monitoring their status, and gathering telemetry (logs and metrics). The emphasis on "enhanced" suggests a significant improvement or expansion over any prior APIs, making version 7.2 a critical baseline. Older IBM Streams documentation mentioned a "Web management service (SWS)" and an "Application manager service (SAM)" related to API-driven operations.29 It is plausible that the new Teracloud Streams REST API consolidates and modernizes these functions. While the streamsx.rest Python library existed for IBM Streams 15, the Teracloud Streams 7.2 API is a distinct evolution.
* 3.1.2. Authentication Mechanisms  
  Specific authentication details for the Teracloud Streams 7.2 REST API are currently unavailable from the accessible documentation.  
  Assumption: Authentication will likely employ a standard token-based mechanism. Common approaches include:
  + **API Key:** An API key string provided by the client in a custom HTTP header (e.g., X-API-Key: <your\_api\_key>) or as part of the Authorization header. This pattern is seen in other streaming or cloud services.34
  + **Bearer Token (OAuth 2.0):** The client obtains a temporary bearer token from an authorization server and includes it in the Authorization: Bearer <token> header. The streamsx.rest library for older IBM Streams supported username/password or service instance configurations for connection 15, but it is unknown if these are retained or superseded in Teracloud Streams 7.2. This is a critical point for clarification. The framework's API client module will need to be designed flexibly to accommodate the confirmed method.
* 3.1.3. Key Endpoints for Framework Interaction   
  While we await confirmation on the exact API endpoints and parameters, the framework's design should accommodate the following expected REST API capabilities: 1. \*\*Application Lifecycle Management\*\* - Deploying Streams Application Bundles (SABs) - Starting and stopping jobs - Checking job status and health 2. \*\*Monitoring and Metrics\*\* - Retrieving instance and job metrics - Accessing application and system logs - Obtaining PE (Processing Element) status information 3. \*\*Health Checks\*\* - Verifying instance availability in both primary and secondary DCs - Checking domain health status Rather than hardcoding assumed endpoint paths, the framework will implement a configuration-driven approach where API endpoints can be easily updated as official documentation becomes available. The API client module will provide a logical abstraction layer that maps these capabilities to whatever specific endpoints are confirmed by the Teracloud documentation. This approach ensures that the framework remains adaptable while still providing immediate utility based on our current understanding of the Teracloud Streams REST API capabilities mentioned in the release notes.  
  **Table 3.1.3.1: (Assumed) REST Management API Endpoints Summary**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **HTTP Method** | **Endpoint Path (Assumed)** | **Description** | **Key Request Parameters (Assumed)** | **Expected Response (Success)** | **Purpose in Framework** |
| POST | /streams/v7/instances/{instance\_id}/jobs | Submit/Deploy a Streams Application Bundle (SAB) | SAB file (multipart/form-data), submission parameters (JSON) | 201 Created (Job ID) | Deploying test application. |
| GET | /streams/v7/instances/{instance\_id}/jobs | List all jobs in an instance | Optional filters (e.g., status, name) | 200 OK (List of jobs) | Verifying deployment, getting job IDs for further actions. |
| GET | /streams/v7/instances/{instance\_id}/jobs/{job\_id} | Get status and details of a specific job | - | 200 OK (Job details) | Monitoring job state (running, failed, stopping), getting PE info. |
| DELETE | /streams/v7/instances/{instance\_id}/jobs/{job\_id} | Cancel/stop a specific job | Optional force flag | 204 No Content or 200 OK | Terminating applications post-test or as part of a fault scenario. |
| POST | /streams/v7/instances/{instance\_id}/jobs/{job\_id}/stop | Stop a specific job (alternative to DELETE) | Optional parameters | 202 Accepted or 200 OK | Graceful shutdown of a job. |
| POST | /streams/v7/instances/{instance\_id}/jobs/{job\_id}/start | Start a previously submitted/stopped job | Optional parameters | 202 Accepted or 200 OK | Restarting a job (less common for fresh deployments). |
| GET | /streams/v7/instances/{instance\_id}/health | Get health status of a Streams instance | - | 200 OK (Instance health) | Verifying instance availability before/after tests. |
| GET | /streams/v7/instances/{instance\_id}/jobs/{job\_id}/metrics | Get metrics for a specific job | Optional metric selectors, time range | 200 OK (Metrics data) | Collecting performance data during tests. |
| GET | /streams/v7/instances/{instance\_id}/jobs/{job\_id}/logs | Get logs for a specific job/PEs | Optional PE ID, log level, time range, number of lines | 200 OK (Log data) | Collecting diagnostic information. |
| GET | /streams/v7/domains/{domain\_id}/status | Get status of a Streams domain | - | 200 OK (Domain status) | Verifying overall domain health (less frequently used by framework). |
| GET | /streams/v7/instances | List all instances in a domain | Optional filters | 200 OK (List of instances) | Discovering available instances. |

\*\*Note:\*\* The above table is highly speculative and serves as a placeholder for information to be obtained from Teracloud SMEs. Actual endpoint paths, parameters, and response codes may differ significantly.

* **3.1.4. Request/Response Formats** **Assumption:** JSON (JavaScript Object Notation) is the data interchange format for both request and response bodies. This is a common standard for modern REST APIs due to its human-readability and ease of parsing by various programming languages, including Python. Confirmation of this and the specific JSON schemas for different requests/responses are needed.

### 3.2. Data Exchange Service (Teracloud Streams 7.2+)

* 3.2.1. Purpose and Use Cases in Failover Testing  
  The Data Exchange service, introduced in Teracloud Streams 7.2, allows for the direct injection and retrieval of tuples (data records) from a running Streams application using a REST API.9 This capability is invaluable for failover testing, enabling:
  + **State Seeding:** Before inducing a fault or triggering a failover, the framework can inject a known, deterministic set of uniquely identifiable data tuples into the application running on the primary DC. This establishes a verifiable application state.
  + **State Validation:** After the failover has occurred and the application is running on the secondary DC, the framework can retrieve data from it. This retrieved data can then be compared against the initially injected data (or its expected transformed output) to verify:
    - Data integrity (no corruption).
    - Data consistency (correctness of state).
    - Data loss or duplication.
    - Correctness of state recovery and processing logic post-failover.
  + **Stress Testing State Transfer:** By injecting varying volumes and velocities of data, the framework can test the performance and robustness of the application's state management and replication mechanisms (potentially handled by the Cross-DC Failover Toolkit) during failover. The general concept of data exchange platforms emphasizes automation, improved data accuracy, and streamlined data flow between disparate systems or components 36, all of which align with the requirements of a comprehensive testing framework.
* 3.2.2. API for Data Injection and Retrieval  
  Specific API endpoints, data formats, and authentication methods for the Data Exchange service are not detailed in the currently accessible Teracloud documentation.3  
  Assumption: The service will expose REST endpoints, likely structured relative to the instance and job, and a specific "endpoint name" defined within the SPL application. Hypothetical endpoints include:
  + POST /streams/v7/instances/{instance\_id}/jobs/{job\_id}/exchange/{endpoint\_name}/inject
    - Used to send a batch of data (e.g., JSON array of tuples) to a specific input endpoint operator within the running Streams job.
    - The request body would contain the data payload.
  + GET /streams/v7/instances/{instance\_id}/jobs/{job\_id}/exchange/{endpoint\_name}/retrieve
    - Used to fetch data from a specific output endpoint operator within the running Streams job.
    - May support parameters for controlling the amount of data or polling frequency. Authentication is assumed to be consistent with the main REST Management API (see Section 3.1.2).

**Table 3.2.2.1: (Assumed) Data Exchange API Interaction Summary**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **HTTP Method** | **Endpoint Path (Assumed)** | **Description** | **Key Request Parameters (Assumed)** | **Expected Response (Success)** | **Purpose in Framework** |
| POST | /streams/v7/instances/{instance\_id}/jobs/{job\_id}/exchange/{endpoint\_name}/inject | Inject data into a Streams application endpoint. | Data payload (e.g., JSON array of tuples matching SPL schema). | 202 Accepted or 200 OK | Seeding application with test data before failover. |
| GET | /streams/v7/instances/{instance\_id}/jobs/{job\_id}/exchange/{endpoint\_name}/retrieve | Retrieve data from a Streams application endpoint. | Optional: max items, timeout, cursor for subsequent retrievals. | 200 OK (Data payload, e.g., JSON) | Validating application state and data integrity after failover. |

\*\*Note:\*\* This table is speculative and requires official documentation for validation.

* 3.2.3. Role of Endpoint Operators  
  Teracloud Streams documentation indicates that Streams applications must incorporate specific "endpoint operators" to expose data ingestion (source) and emission (sink) points that the Data Exchange REST API can interact with.32 These operators act as bridges between the internal SPL data streams and the external REST interface.  
  The SPL application developer is responsible for including and configuring these operators within their application's graph. The automation framework will then target these named endpoints for data injection and retrieval. Crucial details, such as the actual SPL names of these operators, the toolkit they belong to, their configuration parameters (e.g., data format, port binding, security settings), and how they are identified in REST API calls, are currently unknown and need clarification from Teracloud SMEs.
* 3.2.4. Data Schemas for Test Data  
  The framework must generate or handle test data whose schema (tuple structure and types) precisely matches what the target Streams application's endpoint operators expect for input and what they produce as output.  
  For effective failover testing, this data should be:
  + **Deterministic:** Generated in a repeatable and predictable manner for consistent test runs.
  + **Uniquely Identifiable:** Each data tuple, or at least each batch of tuples, should contain unique identifiers (e.g., sequence numbers, UUIDs) and potentially timestamps. This allows the validation module to accurately track data flow, detect loss or duplication, and verify order if relevant, across the failover event. While some research snippets discuss synthetic data generation for ML 39, the core principle of controlled, identifiable input is directly applicable here. JSON is the assumed data format for interchange over the REST API due to its widespread use and ease of handling in Python. The framework will need to serialize Python objects representing test data into JSON for injection and parse JSON responses from the retrieval endpoint.

### 3.3. Cross-DC Failover Toolkit (Teracloud Streams 7.2+)

* 3.3.1. Overview and Functionality  
  The com.teracloud.streams.crossdcfailover toolkit provides "application-level failover across two data centers enabling Disaster Recovery (DR) and Business Continuity (BC)." The toolkit appears to consist of SPL operators, functions, and other artifacts that developers integrate into their Streams applications. These components enable applications to:
* Monitor the health and connectivity of streams processing elements across data centers
* Detect failures or unavailability in the primary DC - Coordinate failover to components in a secondary DC when necessary
* Manage application state consistency across data centers, likely through replication or synchronization mechanisms
* Handle reconnection to data sources and sinks in the newly active DC

The "application-level" nature indicates that Streams applications play an active role in the failover process, rather than relying solely on infrastructure-level mechanisms. This has important implications for testing, as our framework must validate both the toolkit's ability to trigger appropriate failovers and the application's ability to maintain processing integrity and state consistency throughout the failover event.

While specific implementation details of the toolkit require confirmation, this understanding provides a foundation for designing tests that can properly evaluate failover capabilities.

* 3.3.2. Configuration Parameters  
  The specific configuration parameters for this toolkit are unknown due to inaccessible documentation.4  
  Assumption: Based on general patterns for distributed, fault-tolerant systems (e.g., concepts from Teradata DSA failover 41 or Cassandra DC failover 42, though these are different systems), parameters might include:
  + Identifiers/addresses for primary and secondary DC resources (e.g., connection endpoints for inter-application communication or shared state stores).
  + Failover mode (e.g., active-passive, active-active if supported, though active-passive is more common for DR).
  + Heartbeat intervals or timeout values for failure detection.
  + Credentials or security tokens for accessing resources or services in the peer DC.
  + Paths to configuration files specific to the toolkit's operation (some generic failover systems use configFileName parameters, but Streams-specifics are needed).
  + State replication parameters (e.g., synchronous/asynchronous, buffer sizes). It is critical to understand how these parameters are specified: are they SPL operator parameters set at compile time, submission-time parameters passed when deploying the SAB via the REST API, or loaded from external configuration files by the toolkit at runtime? This impacts how the framework can configure different test scenarios.

**Table 3.3.2.1: (Anticipated) Cross-DC Failover Toolkit Configuration Parameters**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Parameter Name (Anticipated)** | **Description** | **Assumed Data Type** | **Typical Values/Example (Speculative)** | **How Set (Assumed: SPL, Submission Time, File)** |
| primaryDcEndpoints | Connection info for peer components in the primary DC. | list<rstring> | ["host1:port", "host2:port"] | Submission Time / File |
| secondaryDcEndpoints | Connection info for peer components in the secondary DC. | list<rstring> | `` | Submission Time / File |
| activeDcRole | Initial role of this application instance (e.g., PRIMARY, SECONDARY). | enum | PRIMARY | Submission Time |
| heartbeatIntervalSecs | Interval for health checks between DC-aware components. | int32 | 5 | SPL Parameter / File |
| detectionTimeoutSecs | Timeout before declaring a peer DC or component as failed. | int32 | 30 | SPL Parameter / File |
| stateReplicationMode | Strategy for state sync (e.g., ASYNC, SYNC). | enum | ASYNC | SPL Parameter |
| sharedStateStoreConfig | Configuration for accessing an external shared state store (if applicable). | rstring | path/to/zookeeper\_config.properties | File / Submission Time |

\*\*Note:\*\* This table is highly speculative and serves as a placeholder for critical information needed from Teracloud.

* 3.3.3. Application-Level Failover Mechanisms  
  The toolkit facilitates failover within the application. This implies that the Streams job, using the toolkit's components, actively manages its transition from one DC to another. This could involve:
  + Operators that monitor connectivity to peers or critical services.
  + Logic to elect a new active instance or set of PEs in the secondary DC.
  + Mechanisms to transfer or reconstruct application state. For example, operators might checkpoint state to a replicated store, or a "hot standby" might receive continuous state updates.
  + Operators that re-initialize connections to external data sources or sinks, now targeting those in the active DC. The testing framework needs to understand this internal application behavior to properly validate a successful failover. This is more intricate than simply verifying if a process is running in the secondary DC; it involves checking if the application has correctly resumed its processing with consistent state.
* 3.3.4. Interaction with the Automation Framework  
  The automation framework will interact with applications built using this toolkit in several ways:
  1. **Deployment:** The framework will deploy SABs that include and are configured to use the Cross-DC Failover Toolkit.
  2. **Monitoring:** During simulated fault conditions, the framework will monitor the application's behavior to determine if the toolkit correctly detects the issue and initiates/completes the failover as designed. This involves observing application logs, metrics (e.g., change in active PEs, data flow resuming in the secondary DC), and potentially any status information exposed by the toolkit itself via REST API or Data Exchange.
  3. **Triggering (if possible):** If the toolkit or Streams API provides a mechanism to externally trigger a failover (e.g., for a controlled switchover test), the framework would use this. This is a key point for clarification, as it would allow for testing the failover process without necessarily inducing a destructive fault.
  4. **Validation:** Post-failover, the framework will use the Data Exchange service and metrics to validate that the application in the new active DC is functioning correctly, has consistent state, and meets performance expectations.

The critical dependency on unavailable documentation for all three interfaces (REST Management API, Data Exchange Service, and Cross-DC Failover Toolkit) is the most significant factor affecting the precision of this guide and the initial development of the framework. The assumptions made are based on industry best practices and available high-level information but require urgent validation. The emphasis on "enhanced" and "new" for these Streams 7.2 features 9 strongly indicates their importance and suggests that prior versions may have lacked the robust programmatic interfaces necessary for this type of advanced automation. Furthermore, the concept of "application-level failover" 9 implies a level of complexity in testing that requires deep visibility into the application's state and behavior, making the Data Exchange service particularly vital.

## 4. Implementation Guide (For Senior Engineers & LLM)

This section provides detailed guidance for senior software engineers and serves as a specification for LLM-based code generation tools tasked with building the Automated Cross-DC Failover Testing Framework. It covers development environment setup, core Python modules, test lifecycle automation, code structure, and reporting.

### 4.1. Development Environment Setup

A consistent and correctly configured development environment is crucial for building the framework.

* **Python Version:** Python 3.9 or newer is recommended. This aligns with the requirements of potential dependencies such as tcconfig (requires Python >=3.9 11) and chaostoolkit (Python 3.8+ 12).
* **Operating System:** While the core Python framework can be developed on Linux, macOS, or Windows, the fault injection components (tc, iptables) are Linux-specific. Therefore, the machine executing these fault injections, or the target nodes for fault injection, must be Linux-based. The framework host itself can be any OS supporting Python.
* **OS-Level Packages for Fault Injection:** On Linux machines designated for fault injection, the following packages are required:
  + iproute2: Provides the tc utility (essential for tcconfig).
  + iptables: Provides the iptables utility. Installation commands vary by distribution (e.g., sudo apt-get install iproute2 iptables on Debian/Ubuntu; sudo yum install iproute2 iptables on RHEL/CentOS).
* **Python Virtual Environment:** It is strongly recommended to use a Python virtual environment (e.g., using venv or conda) to isolate framework dependencies and avoid conflicts with system-wide Python packages.  
  Bash  
  python3 -m venv venv  
  source venv/bin/activate # On Linux/macOS  
  # venv\Scripts\activate # On Windows
* **Core Python Dependencies:** A requirements.txt file should list all necessary Python libraries. Key libraries include:
  + requests: For HTTP communication with Teracloud Streams APIs.
  + PyYAML: For parsing YAML configuration files.
  + paramiko: For SSH communication to execute remote commands.
  + junit-xml: For generating JUnit XML test reports.
  + tcconfig: (Optional, if used directly as a library rather than CLI via paramiko) For tc command wrapping.
  + python-iptables: (Optional, if used directly as a library) For iptables wrapping.
  + prometheus-api-client: (If Streams exposes Prometheus metrics) For querying Prometheus.
  + jmxquery: (If Streams uses JMX and direct query is needed) For JMX interaction. Install dependencies using: pip install -r requirements.txt.
* **Teracloud Streams Environment Access:** -
  + **Version**: Teracloud Streams version 7.2.0 or a subsequent compatible version must be installed and operational.
  + **Data Centers:** At least two data centers (Primary and Secondary) must be configured for cross-DC failover testing. These can be:
    - Two physically separate data centers
    - Two logically separated environments in the same data center for testing purposes - Simulated DC separation using network segregation techniques
  + **Streams Instances:** Teracloud Streams instances must be deployed and running in each participating data center.
  + **Cross-DC Failover Toolkit:** The com.teracloud.streams.crossdcfailover toolkit must be installed and available. As we gain more information about the toolkit's specific requirements and configurations, this section will be updated with more detailed prerequisites.
  + **Test Applications:** Applications utilizing the Cross-DC Failover Toolkit should be designed with observability in mind, exposing useful metrics and logs that help monitor failover behavior.
  + **User Accounts and Permissions:** A service account or user credentials with the necessary permissions for:
    - Accessing the Teracloud Streams REST Management API
    - Deploying and managing applications
    - Accessing metrics and logs
    - Using the Data Exchange Service
* **Credentials:**
  + Credentials for accessing the Teracloud Streams REST APIs (method and actual credentials TBD, see Section 7).
  + SSH credentials (username/password or private key) for accessing Linux nodes where fault injection commands will be executed.

### 4.2. Core Python Modules and Libraries

The framework will be built upon several core Python modules, leveraging well-established libraries for common tasks.

* 4.2.1. HTTP Communication (requests)  
  A dedicated Python class (e.g., TeracloudStreamsApiClient) will encapsulate all interactions with the Teracloud Streams REST APIs. This client will utilize the requests library.14  
  Best Practices:
  + **Session Objects:** Use requests.Session() objects to persist parameters like base URLs and authentication headers across multiple requests, and for connection pooling, which can improve performance.13
  + **Error Handling:** Implement robust error handling. Check HTTP status codes for every response. Use response.raise\_for\_status() to automatically raise an HTTPError for 4xx or 5xx responses. Catch specific exceptions like requests.exceptions.Timeout, requests.exceptions.ConnectionError, and requests.exceptions.HTTPError.14
  + **Timeouts:** Configure explicit connect and read timeouts for all requests to prevent indefinite blocking (e.g., requests.get(url, timeout=(5, 30))).14
  + **Data Handling:** Manage JSON request payloads (serialization) and response bodies (deserialization).
  + **Streaming Responses:** For potentially large responses (e.g., fetching extensive logs or large data sets via Data Exchange), use response.iter\_content(chunk\_size=...) to process data in chunks and avoid high memory consumption.13
  + **Pagination:** If the Teracloud API uses pagination for endpoints that return lists of resources, the client must implement logic to handle it (e.g., following next links or using page number parameters).13
  + **Authentication:** The client should be configurable with the appropriate authentication mechanism (e.g., injecting an API key into headers) once clarified by Teracloud SMEs.
* 4.2.2. Remote Operations (paramiko)  
  The paramiko library 16 will be used to establish SSH connections and execute commands on remote Linux nodes for fault injection. A utility module (e.g., ssh\_client.py) should provide functions for:
  + Connecting to a remote host using username/password or key-based authentication.
  + Securely handling host keys. For development, client.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy()) can be used, but for production or secure environments, host keys should be verified against a known list.45
  + Executing commands (e.g., tc, iptables commands).
  + Retrieving stdin, stdout, and stderr streams, as well as the exit status of commands.16
  + Properly closing SSH connections.
  + Error handling for connection failures, authentication errors, and command execution failures.
* 4.2.3. Network Impairment (tcconfig, python-iptables)  
  These tools/libraries will be used by the Fault Injection Module, typically invoked via paramiko.
  + **tcconfig:** This Python library acts as a wrapper around the Linux tc utility, simplifying the configuration of network traffic control parameters.11
    - **Capabilities:** Simulate WAN conditions including network latency (e.g., tcset eth0 --delay 100ms), packet loss (e.g., tcset eth0 --loss 0.1%), bandwidth limitation (e.g., tcset eth0 --rate 1Mbps), packet corruption, duplication, and reordering.
    - **Targeting:** Rules can be applied to specific network interfaces (e.g., eth0), IP addresses/networks, and ports. It can also target Docker container network interfaces (veth) if Streams PEs are containerized and such manipulation is desired.11
    - **Dependencies:** Requires the sch\_netem Linux kernel module to be loaded on the target system.11
    - **Usage:** The framework will construct tcset commands (or use tcconfig library functions if preferred) to apply desired network impairments and tcdel commands to remove them cleanly after the test phase.
  + **python-iptables or direct iptables commands:** The python-iptables library 15 provides a Pythonic interface to iptables. Alternatively, iptables commands can be constructed as strings and executed directly via paramiko.
    - **Capabilities:** Primarily used to simulate network partitions by adding rules to DROP packets between specified source/destination IP addresses and/or ports.19
    - **Mechanism:** Involves manipulating rules within iptables chains (e.g., INPUT, FORWARD, OUTPUT) and setting targets (e.g., DROP, ACCEPT). For a partition, DROP rules would be inserted.
    - **Usage:** The framework will need functions to dynamically add iptables rules to create a partition and, critically, to remove these specific rules to restore connectivity post-test, without inadvertently flushing other essential firewall rules.
* 4.2.4. Test Orchestration and Control Flow  
  The Orchestration Engine will contain the primary logic for executing a test scenario. This involves:
  + Parsing the test case definition (from YAML).
  + Sequentially executing test phases:
    1. **Setup:** Prepare environment, deploy Streams application.
    2. **Pre-failover:** Inject baseline data, collect initial metrics.
    3. **Fault Induction:** Apply the defined fault using the Fault Injection Module.
    4. **Failover Window:** Monitor for failover events through observable indicators such as application logs, metrics patterns, and job status changes. The framework will observe how the Cross-DC Failover Toolkit responds to the induced fault condition and track the progression of the failover process.
    5. **Post-failover Validation:** Verify application availability in secondary DC, retrieve and validate data, collect metrics.
    6. **Teardown:** Remove fault conditions, stop/undeploy application, clean up resources.
  + This will be implemented using Python classes and functions to manage state transitions and interactions between framework components.
* 4.2.5. Monitoring Hooks (Prometheus client, JMX query)  
  The Monitoring and Metrics Collector will interface with Teracloud Streams' monitoring capabilities.
  + **Prometheus:** If Streams 7.2 exposes metrics via Prometheus endpoints (a common practice for modern cloud-native applications), the prometheus-api-client Python library 15 can be used.
    - Connect to the Prometheus HTTP API.
    - Execute PromQL queries to fetch specific metrics (e.g., sum(rate(node\_cpu\_seconds\_total{mode!="idle"}[1m])) by (instance) 53).
    - Collect KPIs like throughput, latency, error rates, resource utilization (CPU, memory) of Streams jobs/PEs.
  + **JMX (Java Management Extensions):** If Streams, being Java-based, exposes detailed metrics via JMX:
    - The jmxquery Python library 15 can be used. This library typically bundles a small Java agent (.jar) and requires a Java Runtime Environment (JRE) on the machine executing jmxquery.
    - It connects to the JMX RMI interface of the target Java application (Streams PEs or management services).
    - Queries specific MBean (Managed Bean) names and attributes (e.g., kafka.cluster:type=\*,name=\*,topic=\*,partition=\* for Kafka metrics 25).
    - The TIBCO Spotfire service documentation provides an example of monitoring a Python service *itself* via JMX, which involves setting JMX properties like jmx.rmi.port.54 Similar configuration might be needed for Streams if JMX is remote. The specific metrics to collect will depend heavily on what Teracloud Streams exposes and what is deemed critical for assessing failover impact.
* 4.2.6. Configuration Management (PyYAML)  
  The framework will be heavily configuration-driven. PyYAML 20 will be used to load and parse YAML files.
  + **config.yaml (Framework Configuration):** Contains global settings like API base URLs, default timeouts, logging configurations, paths to external tools.
  + **Test Scenario Files (e.g., my\_test.yaml):** Define specific test cases, including the Streams application to deploy, fault conditions, data injection parameters, and validation criteria.
  + **Security:** Use yaml.safe\_load() instead of yaml.load() to prevent arbitrary code execution from untrusted YAML files.21
  + The Configuration Manager component will be responsible for loading, validating (e.g., against a schema), and providing access to these configurations.
* 4.2.7. Logging (logging module)  
  Comprehensive logging is essential for debugging the framework and understanding test execution. Python's built-in logging module will be used.26
  + **Structured Logging:** Log messages should be structured and informative.
  + **Log Levels:** Utilize standard log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL). The level should be configurable.
  + **Handlers:** Configure handlers to output logs to both the console (StreamHandler) and files (FileHandler or RotatingFileHandler).26
  + **Formatters:** Define a log format that includes timestamps, log level, module/function name, and the message (e.g., %(asctime)s - %(name)s - %(levelname)s - %(message)s).26
  + **Log Rotation:** For file logs, implement rotation (e.g., RotatingFileHandler) to manage file sizes and prevent excessive disk usage.
  + **Configuration:** Logging can be configured via logging.basicConfig() for simple setups, or more advanced configurations using dictionaries (dictConfig) or configuration files (fileConfig).27

### 4.3. Implementing Test Lifecycle Automation

The core of the framework lies in its ability to automate the entire lifecycle of a failover test.

* 4.3.1. Test Case Definition (YAML/JSON structure)  
  Test cases will be defined in external YAML files, adhering to a predefined schema. This allows users to create new tests without modifying Python code.  
  **Table 4.3.1.1: Test Case Definition Schema**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Field Name** | **Data Type** | **Description** | **Example Value** | **Required** |
| test\_id | string | Unique identifier for the test case. | tc\_network\_partition\_full\_dc1\_dc2 | Yes |
| description | string | Human-readable description of the test scenario and objectives. | Simulate full network partition between DC1 and DC2, verify app failover. | Yes |
| streams\_application\_sab | string | Path to the Streams Application Bundle (SAB) file to be deployed. | applications/my\_failover\_app.sab | Yes |
| submission\_params | dict | (Optional) Key-value pairs for submission-time parameters for the SAB. | {"inputTopic": "topicA", "checkpointInterval": 60} | No |
| pre\_failover\_data | dict | Configuration for data to be injected before failover. | See sub-fields below. | No |
| pre\_failover\_data.generator\_script | string | (Optional) Path to a custom Python script for generating data. | data\_generators/order\_stream.py | No |
| pre\_failover\_data.data\_file | string | (Optional) Path to a file containing data to inject (e.g., JSON). | test\_data/initial\_orders.json | No |
| pre\_failover\_data.target\_endpoint | string | Name of the Data Exchange endpoint in the Streams app for injection. | OrderIngestEndpoint | If data injection used |
| fault\_scenario | dict | Defines the fault to be injected. | See sub-fields below. | Yes |
| fault\_scenario.type | enum | Type of fault: NETWORK\_PARTITION, LATENCY, PACKET\_LOSS, BANDWIDTH\_LIMIT, SERVICE\_STOP (future). | NETWORK\_PARTITION | Yes |
| fault\_scenario.parameters | dict | Specifics for the fault type. E.g., for NETWORK\_PARTITION: {"source\_dc\_nodes": ["ip1", "ip2"], "target\_dc\_nodes":} | {"target\_interface": "eth1", "latency\_ms": 200, "loss\_percent": 5} | Yes |
| fault\_scenario.duration\_seconds | integer | Duration for which the fault should be active. | 120 | No (implies manual removal or until test end) |
| failover\_trigger\_method | enum | How failover is expected/initiated: failover\_condition: Monitored condition for failover (automatic being the primary mode). | AUTO\_TOOLKIT\_DETECT | Yes |
| post\_failover\_validation\_checks | list[dict] | List of checks to perform after failover. | `` | Yes |
| post\_failover\_validation.target\_endpoint | string | Name of the Data Exchange endpoint for retrieving data for validation. | ValidatedOutputEndpoint | If data validation used |
| expected\_recovery\_time\_seconds | integer | Target Recovery Time Objective (RTO) in seconds. | 300 | No |
| expected\_data\_loss\_percentage | float | Target Recovery Point Objective (RPO) as max acceptable data loss. | 0.0 | No |

* 4.3.2. Deploying Teracloud Streams Applications (SAB files) via API  
  The Orchestration Engine will invoke the TeracloudStreamsApiClient to deploy the specified SAB file.
  1. Read streams\_application\_sab and submission\_params from the test case definition.
  2. Construct an API request to the (assumed) POST /streams/v7/instances/{instance\_id}/jobs endpoint. The SAB file will likely be sent as multipart/form-data, and submission parameters as a JSON part.
  3. Send the request and handle the response:
     + On success (e.g., HTTP 201 Created), extract the job\_id from the response. Store this ID for subsequent operations.
     + Log success or failure, including any error messages from the API.
  4. Poll the job status using (assumed) GET /streams/v7/instances/{instance\_id}/jobs/{job\_id} until it reaches a stable running state or a timeout occurs.
* 4.3.3. Initiating and Monitoring Failover Scenarios  
  This is the core of the test execution.
  1. **Pre-computation/Pre-analysis (State Seeding):**
     + If pre\_failover\_data is defined, use the Data Generation module to prepare test data.
     + Inject this data into the deployed Streams application via the Data Exchange service, targeting the specified pre\_failover\_data.target\_endpoint. Record timestamps and unique identifiers of injected data for later validation.
  2. **Inducing Faults:**
     + Based on fault\_scenario.type and fault\_scenario.parameters, invoke the Fault Injection Module.
     + Example: For NETWORK\_PARTITION, use paramiko to execute iptables commands on relevant nodes to block traffic between DC1 and DC2.
     + Example: For LATENCY, use paramiko to execute tc commands (via tcconfig) on inter-DC network path interfaces.
     + Log the exact commands executed and their outcomes.
  3. **Triggering/Observing Failover:**
     + The framework will primarily focus on observing failover events that occur in response to injected faults. This includes:
       - Monitoring application logs for messages indicating failover activities
       - Tracking metrics that show processing transitions between DCs (e.g., traffic shifts from DC1 PEs to DC2 PEs)
       - Querying job status via the Streams API to detect changes in active processing elements
       - Observing data flow patterns that indicate a successful transition to the secondary DC
       - If the toolkit provides APIs or mechanisms for controlled failover triggering, the framework will incorporate these once they're confirmed. However, the core testing approach will focus on realistic fault scenarios that allow the toolkit to demonstrate its autonomous failover capabilities.
       - The framework will record precise timestamps when failover events are detected to enable accurate RTO/RPO measurements.
  4. **Monitoring During Failover Window:** Continuously collect logs and metrics from both DCs throughout the fault duration and failover process.
* 4.3.4. Collecting Metrics and Logs  
  The Monitoring and Metrics Collector will be active throughout the test.
  + Periodically query the Streams REST API for job-level and instance-level metrics.
  + If Prometheus integration is used, scrape Prometheus endpoints at configured intervals.
  + Fetch application logs from Streams instances via the REST API, potentially filtering by timestamp or severity.
  + Store collected metrics and logs with timestamps, associating them with the current test phase (pre-fault, fault-active, post-failover). This correlation is vital for analysis.
* 4.3.5. Generating Deterministic Test Data and Validation  
  The Data Generation and Validation Module plays a key role in evaluating failover success by providing a controlled way to verify data consistency across a failover event. - **Generation:**
  + Python functions will create data streams with the following characteristics: - Unique sequence IDs for tracking individual records - Timestamps indicating when each record was created - Deterministic payloads that can be verified after processing - Optional checksums for integrity verification This approach provides complete control over test data, making it easier to identify any issues during failover.
  + **Injection:**
    - * Use the Data Exchange API to send the generated data to the application's input endpoint
      * Carefully control the timing of data injection relative to fault introduction - Maintain detailed records of all injected data for later comparison
      * Tag data with metadata that allows tracking its path through the system
  + **Retrieval and Validation:** After failover completes and the application stabilizes in the secondary DC, the framework will:

1. Query the application's output endpoint(s) to retrieve processed data

2. Compare against the original injected data (accounting for expected transformations)

3. Calculate key metrics:

* + - * + **Completeness:** What percentage of injected records was successfully processed?
        + **Correctness:** Were all records processed according to the application's business logic?
        + **Ordering:** If sequence matters, was the correct order maintained? –
        + **Duplicates:**\*\* Were any records processed more than once?
        + **Recovery Point:** What was the effective data loss (if any) measured in time or record count?
  + These measurements provide quantitative evidence of how well the Cross-DC Failover Toolkit maintained data consistency during the failover process.
* 4.3.6. State Management and Verification  
  This is closely tied to data validation but focuses more on the internal consistency of the application as managed by the Cross-DC Failover Toolkit.
  + Streams applications under test may need to be designed to expose critical state information via the Data Exchange service or specific metrics for this purpose.
  + The framework would query this state information before inducing the fault and after the failover is complete.
  + Validation involves comparing these state snapshots, accounting for any legitimate state changes that should have occurred due to ongoing processing or the failover event itself. For example, if the application maintains counters, these should reflect continuous processing without unexpected resets or gaps. The complexity here is that "application-level failover" implies the application, with the toolkit's help, manages its own state recovery. The framework needs to assess how well it does this.

### 4.4. Code Structure and Modularity

A well-organized code structure is essential for maintainability and scalability. A suggested Python package structure:

teracloud\_failover\_tester/  
├── main.py # Main entry point for the CLI  
├── orchestrator/  
│ ├── \_\_init\_\_.py  
│ └── test\_executor.py # Core test execution logic  
├── streams\_client/  
│ ├── \_\_init\_\_.py  
│ └── api\_client.py # Wrapper for Teracloud Streams REST Management API  
│ └── data\_exchange\_handler.py # Wrapper for Data Exchange Service API  
├── fault\_injection/  
│ ├── \_\_init\_\_.py  
│ └── network\_faults.py # Implements tc, iptables based faults  
│ └── service\_faults.py # (Future) For stopping/starting services  
├── monitoring/  
│ ├── \_\_init\_\_.py  
│ └── metrics\_collector.py # Collects metrics (Prometheus, JMX, API)  
│ └── log\_collector.py # Collects logs via API  
├── data\_handler/  
│ ├── \_\_init\_\_.py  
│ └── data\_generator.py # Generates test data  
│ └── data\_validator.py # Validates data consistency post-failover  
├── reporting/  
│ ├── \_\_init\_\_.py  
│ └── junit\_reporter.py # Generates JUnit XML reports  
├── config/  
│ ├── \_\_init\_\_.py  
│ └── loader.py # Loads and validates YAML configurations  
│ └── schema.py # (Optional) JSON schema for config validation  
├── utils/  
│ ├── \_\_init\_\_.py  
│ └── ssh\_manager.py # Paramiko SSH client wrapper  
│ └── logger\_setup.py # Centralized logging configuration  
└── tests/ # Unit and integration tests for the framework itself  
 ├── \_\_init\_\_.py  
 ├── test\_orchestrator.py  
 └──...  
├── requirements.txt # Python dependencies  
├── config.yaml.example # Example global framework configuration  
└── scenarios/ # Directory for test scenario YAML files  
 └── example\_scenario.yaml

Each module should encapsulate a specific domain of functionality. Classes should be used to represent key entities (e.g., TestExecutor, StreamsJob, NetworkFault). Clear interfaces between modules will promote loose coupling. This detailed structure, including class and method responsibilities (implicitly defined by their purpose and interaction with external APIs/tools), is designed to be directly useful for LLM-based code generation.

### 4.5. Generating Test Reports (JUnit XML)

For integration with CI/CD systems and standardized reporting, test results should be generated in JUnit XML format.

* **Library Choice:**
  + The junit-xml library 23 can be used to programmatically create JUnit XML files.
  + Alternatively, if pytest 56 is used as the test runner for the framework's scenarios (treating each scenario file as a test), pytest can automatically generate JUnit XML reports using the --junitxml=report.xml command-line option.22
  + **Report Content:** The JUnit XML report should map test outcomes to standard elements, including specific information about Cross-DC failover performance:
  + <testsuites>: Root element for all test scenarios.
  + <testsuite name="FailoverScenarioSuite" tests="N" failures="M" errors="E" time="T">: Represents a collection of test scenarios run.
  + <testcase classname="ScenarioFile" name="test\_id\_from\_yaml" time="t\_case">: Represents a single test case execution.
    - Include custom properties for key failover metrics:

xml

<properties>

<property name="recovery\_time\_seconds" value="45.2"/>

<property name="data\_consistency\_percentage" value="99.8"/>

<property name="failover\_type" value="auto\_detected"/>

</properties>

* + - For validation failures:

xml

<failure message="Recovery Time Objective exceeded: 120s vs 60s target" type="RTOViolation">

Detailed timeline of failover events:

- 10:15:00.000 - Network partition introduced

- 10:15:22.342 - Failover detection by toolkit observed

- 10:17:00.125 - Processing resumed in secondary DC

</failure>

* + - For unexpected errors:

xml

<error message="Failed to connect to Streams API" type="ConnectionError">

Stack trace or detailed error information.

</error>

* + <system-out>: Include key observations about the failover process, such as detected toolkit behavior patterns.
  + <system-err>: Include error messages or warnings observed during the test.

This reporting approach will provide clear visibility into the Cross-DC Failover Toolkit's performance and any potential issues, even as we continue to learn more about its specific behavior patterns.

**Table 4.5.1: JUnit XML Report Structure Mapping**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Outcome/Data Point** | **JUnit XML Element/Attribute** | **Framework Source** | **Example** |
| Overall Test Run | <testsuites> | Orchestration Engine | <testsuites name="TeracloudStreamsFailoverTests"> |
| Group of Scenarios (e.g., by type) | <testsuite name="..." tests="..." failures="..."> | Orchestration Engine (grouping by scenario file or category) | <testsuite name="NetworkPartitionScenarios"...> |
| Individual Test Scenario | <testcase classname="..." name="..." time="..."> | Test Scenario Executor (classname=scenario\_file, name=test\_id) | <testcase classname="scenarios/dc\_isolation.yaml" name="TC001\_FullIsolate" time="350.5"> |
| Successful Validation Check | (No specific element, contributes to testcase success) | Data Validation Module | (Implicitly part of a passed testcase) |
| Failed Validation Check | <failure message="..." type="...">...</failure> | Data Validation Module, Orchestration Engine | <failure message="RPO violated: 5% data loss detected" type="ValidationFailure">Details...</failure> |
| Framework Error (e.g., API unavailable) | <error message="..." type="...">...</error> | Any framework component encountering an exception | <error message="Failed to connect to Streams API" type="ConnectionError">Traceback...</error> |
| Key Log Output for a Scenario | <system-out>...</system-out> | Orchestration Engine, Log Collector | <system-out>Failover initiated at 2023-10-26T10:05:00Z</system-out> |
| Attachments/Links (if supported by consumer) | <property name="attachment" value="URL\_or\_path"/> in <properties> | Reporting Module | <property name="detailed\_logs" value="http://logserver/run123.log"/> |

Adherence to this structure ensures that the reports are consumable by common CI/CD platforms like Jenkins, GitLab CI, etc., for displaying test results, tracking trends, and potentially gating deployments based on failover test outcomes. The robustness of the chosen Python libraries for core functionalities reduces development overhead and leverages community-vetted solutions, which is a sound engineering practice.

## 5. User Guide (For Deployment and Operations)

This section provides practical instructions for operational users, such as SREs or QA teams, to deploy, configure, and run the Automated Cross-DC Failover Testing Framework, and to interpret its results.

### 5.1. Prerequisites and System Requirements

Before deploying and using the framework, ensure the following prerequisites are met:

* **5.1.1. Framework Host Machine:**
  + **Operating System:** A Linux distribution is recommended, especially if fault injection tools (tc, iptables) are to be run locally or if consistency with target node OS is desired. Core Python components are cross-platform (Linux, macOS, Windows).
  + **Python:** Python version 3.9 or later (as specified in Section 4.1). Verify with python3 --version.
  + **Disk Space:** Sufficient free disk space for the framework's codebase, Python virtual environment, generated logs, and any temporary files or reports (estimate at least 1-2 GB, more if logs are verbose or retained long-term).
  + **Network Access:**
    - Unrestricted outbound TCP/IP connectivity to the Teracloud Streams management API endpoints in all relevant data centers.
    - SSH (typically TCP port 22) access from the framework host to any Linux nodes designated as targets for fault injection.
  + **Required Tools (if not using paramiko for everything):** ssh client if manual SSH is ever needed.
* **5.1.2. Teracloud Streams Environment:**
  + **Version:** Teracloud Streams version 7.2.0 or a subsequent compatible version must be installed and operational.9
  + **Data Centers:** At least two data centers (Primary and Secondary) must be configured and operational for cross-DC failover testing.
  + **Streams Instances:** Teracloud Streams instances must be deployed and running in each participating data center.
  + **Cross-DC Failover Toolkit:** The com.teracloud.streams.crossdcfailover toolkit must be installed and available within the Streams environment for use by applications under test.9
  + **User Accounts and Permissions:** A dedicated service account or user credentials with the necessary permissions to:
    - Access and utilize the Teracloud Streams REST Management API (for deploying applications, managing jobs, querying status, metrics, logs).
    - Access and utilize the Teracloud Streams Data Exchange Service API (for injecting and retrieving data from applications). The specific roles and permissions required will depend on Teracloud Streams' security model and need to be confirmed (see Section 7).
* 5.1.3. Target Nodes for Fault Injection (if applicable):  
  These are the machines whose network behavior will be manipulated (e.g., inter-DC routers, specific application servers if testing client-side failover, or the Streams hosts themselves if network faults are to be simulated at that level).
  + **Operating System:** Linux (as tc and iptables are Linux utilities).
  + **Required Packages:**
    - iproute2 (provides tc).
    - iptables.
  + **SSH Server:** An SSH server (e.g., OpenSSH Server) must be installed, enabled, and configured to allow access from the framework host machine using the provided SSH credentials (key-based authentication is highly recommended over passwords).

### 5.2. Deployment of the Failover Testing Framework

Follow these steps to deploy the framework:

* **5.2.1. Installation Steps:**
  1. **Obtain Framework Code:**
     + If distributed as a Git repository:  
       Bash  
       git clone <repository\_url>  
       cd teracloud\_failover\_tester
     + If distributed as an archive (e.g., .zip or .tar.gz):  
       Bash  
       tar -xzvf teracloud\_failover\_tester.tar.gz  
       cd teracloud\_failover\_tester
  2. **Set up Python Virtual Environment:**  
     Bash  
     python3 -m venv venv  
     source venv/bin/activate # On Linux/macOS  
     # venv\Scripts\activate # On Windows (if framework host is Windows)
  3. **Install Dependencies:**  
     Bash  
     pip install -r requirements.txt  
     This command will install all Python libraries listed in the requirements.txt file into the active virtual environment.
* 5.2.2. Initial Configuration:  
  The framework requires a global configuration file, typically named config.yaml, located in the root directory of the framework or a specified path. A template or example file (config.yaml.example) should be provided with the framework.
  1. Copy the example configuration: cp config.yaml.example config.yaml
  2. Edit config.yaml with a text editor and populate the necessary parameters.

**Table 5.2.2.1: Framework Configuration Parameters (config.yaml)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Parameter Name** | **Description** | **Data Type** | **Example Value** | **Default Value (if any)** |
| streams\_api\_base\_url\_dc1 | Base URL for Teracloud Streams REST Management API in Primary DC. (Needs confirmation) | string | https://streams-api.dc1.example.com/streams/v7 | None |
| streams\_api\_base\_url\_dc2 | Base URL for Teracloud Streams REST Management API in Secondary DC. (Needs confirmation) | string | https://streams-api.dc2.example.com/streams/v7 | None |
| streams\_api\_auth\_type | Authentication type for Streams API (e.g., API\_KEY, OAUTH2). (Needs confirmation) | string | API\_KEY | None |
| streams\_api\_key\_header | HTTP Header name for API Key (if auth\_type is API\_KEY). (Needs confirmation) | string | X-API-Key | None |
| streams\_api\_key\_value\_dc1 | API Key for Primary DC. (Store securely, e.g., env variable, and reference here) | string | your\_dc1\_api\_key\_here | None |
| streams\_api\_key\_value\_dc2 | API Key for Secondary DC. (Store securely) | string | your\_dc2\_api\_key\_here | None |
| default\_ssh\_user | Default username for SSH connections to fault injection nodes. | string | tester\_user | None |
| default\_ssh\_key\_path | Default path to SSH private key for fault injection nodes. | string | ~/.ssh/id\_rsa\_failover\_tester | None |
| log\_file\_path | Path to the framework's main log file. | string | logs/failover\_tester.log | failover\_tester.log |
| log\_level | Logging level for the framework (DEBUG, INFO, WARNING, ERROR, CRITICAL). | string | INFO | INFO |
| reports\_dir | Directory to store generated JUnit XML reports. | string | reports/ | reports/ |
| default\_api\_timeout\_seconds | Default timeout for API calls (connect, read). | integer | 60 | 60 |

\*\*Security Note:\*\* API keys and SSH credentials should be handled securely. Avoid hardcoding them directly in `config.yaml` if possible. Consider using environment variables, a secrets management system, or prompting the user if running interactively, and then referencing these from the configuration. The example values above are illustrative.

### 5.3. Configuring Test Scenarios

Test scenarios are defined in separate YAML files (e.g., stored in a scenarios/ directory). Each file describes one or more tests to be performed. Refer to Table 4.3.1.1 for the detailed schema of a test case definition.

* 5.3.1. Defining Target Environments within a Scenario:  
  While global API endpoints are in config.yaml, a scenario might need to specify which Streams instance ID to target within DC1 and DC2 if multiple are available.
  + Example snippet within a scenario YAML:  
    YAML  
    target\_primary\_instance\_id: "streams\_instance\_prod\_dc1"  
    target\_secondary\_instance\_id: "streams\_instance\_dr\_dc2"  
    fault\_injection\_nodes\_dc1: ["10.1.1.5", "10.1.1.6"] # IPs for tc/iptables  
    fault\_injection\_nodes\_dc2: ["10.2.2.5", "10.2.2.6"] # IPs for tc/iptables
* 5.3.2. Specifying Fault Conditions and Monitoring Failover Response:  
  The fault\_scenario section in the test case YAML defines the specific fault conditions to be injected. While we continue to gather information about the Cross-DC Failover Toolkit's exact behaviors, our framework will focus on creating realistic fault conditions and observing how applications respond.
  + **Network Partition Example:**  
    YAML  
    fault\_scenario:  
     type: NETWORK\_PARTITION  
     parameters:  
     # Block all TCP traffic between data centers  
     source\_dc\_subnets: ["192.168.1.0/24"] # Subnet of DC1 application nodes  
     target\_dc\_subnets: ["192.168.2.0/24"] # Subnet of DC2 application nodes  
     protocol: "tcp" # "all", "tcp", "udp", "icmp"  
     direction: "BOTH" # "FORWARD\_ONLY" (source to target), "REVERSE\_ONLY", "BOTH"  
     duration\_seconds: 180  
    # Define metrics and indicators to monitor during failover

failover\_monitoring:

# List of metrics or log patterns that indicate failover progression

indicators:

- type: "LOG\_PATTERN" pattern: "Starting failover procedure to secondary DC"

- type: "METRIC\_CHANGE"

metric\_name: "active\_processing\_elements\_dc2"

change: "INCREASE"

* + **Latency Injection Example:**  
    YAML  
    fault\_scenario:  
     type: LATENCY  
     parameters:  
     target\_interface: "eth1" # Interface on fault injection node handling inter-DC traffic  
     latency\_ms: 150  
     jitter\_ms: 20 # Optional latency variation  
     duration\_seconds: 300

*# Focus on observing how the application and toolkit respond*

failover\_observation:

max\_detection\_time\_seconds: 60 *# Expected maximum time for failover detection*

max\_completion\_time\_seconds: 300 *# Expected maximum time for complete failover*

* + **Packet Loss Example:**  
    YAML  
    fault\_scenario:  
     type: PACKET\_LOSS  
     parameters:  
     target\_interface: "eth1"  
     loss\_percent: 5  
     duration\_seconds: 120

*# Define data consistency checks to perform after failover*

post\_failover\_validation:

data\_integrity\_checks: true

expected\_max\_data\_loss\_percent: 0.1

The framework will monitor application behavior during and after the fault injection, capturing key metrics and events that indicate how the Cross-DC Failover Toolkit is responding.

* 5.3.3. Customizing Test Data:  
  The pre\_failover\_data section in the scenario YAML controls data injection.
  + If using a custom Python script for complex data generation:  
    YAML  
    pre\_failover\_data:  
     generator\_script: "scenarios/data\_generators/complex\_event\_stream.py"  
     target\_endpoint: "ComplexEventIngest"  
     # Parameters for the script can be passed if the script supports argv  
     # generator\_params: ["--rate", "1000", "--duration", "60s"]
  + If injecting data from a static JSON file:  
    YAML  
    pre\_failover\_data:  
     data\_file: "scenarios/test\_data/baseline\_transactions.json"  
     target\_endpoint: "TransactionIngest"

Ensure that the data schema produced by the generator or contained in the file matches the SPL tuple schema expected by the target\_endpoint in the Streams application.

### 5.4. Running Failover Tests

* **5.4.1. Step-by-Step Execution Guide:**
  1. Ensure the Python virtual environment is activated: source venv/bin/activate (or equivalent).
  2. Navigate to the root directory of the framework.
  3. Execute the framework using the main script, specifying the scenario file:  
     Bash  
     python main.py --scenario scenarios/my\_test\_scenario.yaml
  4. **Command-Line Arguments (Illustrative):**
     + --scenario <path\_to\_scenario.yaml>: (Required) Path to the test scenario definition file.
     + --config <path\_to\_config.yaml>: (Optional) Path to the global framework configuration file (defaults to config.yaml).
     + --log-level <LEVEL>: (Optional) Override the log level from config.yaml (e.g., DEBUG).
     + --dry-run: (Optional) Print actions that would be taken without actually executing them (useful for debugging scenario configurations).
     + --tags <tag1,tag2>: (Optional, if tagging is implemented) Run only scenarios or tests matching specified tags.
* **5.4.2. Monitoring Test Progress:**
  + **Console Output:** The framework will print status messages to the console indicating the current phase of the test (e.g., "Deploying application...", "Injecting fault: Network Partition...", "Monitoring for failover...", "Validating data..."). Error messages will also appear here.
  + **Log Files:** For more detailed information, tail the framework's log file specified in config.yaml (e.g., tail -f logs/failover\_tester.log). This provides verbose output including API calls, command executions, and detailed error stack traces.

### 5.5. Interpreting Test Results

After a test run completes, analyze the outputs to determine the success or failure of the failover.

* **5.5.1. Understanding Output Logs:**
  + **Framework Logs:** Located at log\_file\_path (from config.yaml). These are the primary source for understanding the framework's actions and any issues it encountered. Look for:
    - Timestamps for each major action.
    - Successful API call confirmations.
    - Errors from API calls (HTTP status codes, error messages).
    - Commands executed for fault injection and their output.
    - Data validation summaries (number of records lost, corrupted, duplicated).
    - Overall test pass/fail status.
  + **Streams Application Logs:** If the framework is configured to fetch them (via API), these logs will provide insight into the application's behavior during the failover. They might be stored in a subdirectory within logs/ or referenced in the main framework log.
  + **Fault Injection Tool Logs:** tcconfig and iptables commands typically output to stdout/stderr, which should be captured by paramiko and logged by the framework.
* **5.5.2. Analyzing JUnit XML Reports:**
  + Reports are stored in the reports\_dir (from config.yaml), typically named like TEST-<scenario\_id>.xml.
  + These can be opened in any XML viewer or, more commonly, are automatically parsed and displayed by CI/CD systems (Jenkins, GitLab CI, etc.).
  + The report will show:
    - Number of test cases (scenarios) run, passed, failed, or errored.
    - Execution time for each test case.
    - For failures/errors: a message and often a snippet of the error output or assertion failure.
  + Correlate any failures in the JUnit report with the detailed information in the framework logs to understand the root cause.
* 5.5.3. Key Metrics for Failover Success:  
  A comprehensive evaluation of failover success requires measuring multiple aspects of the application's behavior. Our framework will focus on the following key metrics:  
  **Table 5.5.3.1: Test Result Interpretation Guide**

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric/Observation** | **How to Check** | **Expected Outcome (Success Example)** | **Potential Issues if Deviates** |
| **Recovery Time Objective (RTO)** | Measured from fault detection/injection to full service restoration in the secondary DC | Framework will automatically calculate this based on timestamps in logs and observed application behavior | Deviations from expected recovery time may indicate configuration issues or resource constraints |
| **Data Consistency (RPO Performance)** | Measured by comparing data injected before failover with data retrieved after failover | Quantifies any data loss during the failover process | Critical for validating the Cross-DC Failover Toolkit's state management capabilities |
| **Processing Performance** | Measures throughput and latency after failover compared to baseline performance | Identifies any degradation that might impact service level agreements | May reveal resource allocation issues in the secondary DC |
| **Failover Process Completeness** | Evaluates whether all application components properly failed over | Checks for any "split-brain" scenarios or components left in an inconsistent state | Verifies correct shutdown of components in the primary DC |

For each test scenario, you should define acceptable thresholds for these metrics based on your specific application requirements and business continuity objectives.

### 5.6. Troubleshooting Common Issues

This section provides guidance on diagnosing and resolving common problems encountered when using the framework.

* **Framework Connectivity Problems:**
  + **Symptom:** Errors like "Connection refused," "Host unreachable," "Timeout" when the framework tries to contact Teracloud Streams APIs or SSH into fault injection nodes.
  + **Checks:**
    - Verify network connectivity from the framework host to the target API endpoints/SSH hosts (use ping, telnet, curl).
    - Check firewall rules on the framework host, intermediate network devices, and target hosts.
    - Ensure correct API base URLs and SSH hostnames/IPs in config.yaml and scenario files.
    - Confirm that the Teracloud Streams services (e.g., SWS, SAM) are running and listening on the expected ports.
* **Authentication Failures:**
  + **Symptom:** HTTP 401 Unauthorized or 403 Forbidden errors from Streams APIs. SSH authentication failures.
  + **Checks:**
    - Verify correctness of API keys, tokens, SSH usernames, passwords, or SSH key paths in config.yaml or environment variables.
    - Ensure the account used has the necessary permissions in Teracloud Streams.
    - For SSH key-based auth, confirm the public key is in the authorized\_keys file on the target node and file permissions are correct (~/.ssh as 700, ~/.ssh/authorized\_keys as 600).
    - Check for clock skew issues if time-sensitive tokens are used.
* **Fault Injection Script Errors:**
  + **Symptom:** Errors reported by paramiko during remote command execution, or faults not being applied as expected.
  + **Checks:**
    - Examine framework logs for the exact tc or iptables commands executed and any error messages returned.
    - Manually run the same commands on the target fault injection node to see if they work.
    - Ensure iproute2 and iptables are installed on target nodes.
    - Ensure the sch\_netem kernel module is loaded for tc network emulation (lsmod | grep sch\_netem). If not, try sudo modprobe sch\_netem.
    - Verify the user executing commands via SSH has sufficient privileges (e.g., sudo access if required for tc/iptables, and sudoers configured appropriately for passwordless sudo if used).
* **Streams Application Deployment Failures:**
  + **Symptom:** API errors during SAB submission, or job failing to start.
  + **Checks:**
    - Validate the SAB file is correct and compatible with the target Streams version.
    - Check Streams instance logs for detailed error messages related to the job submission or startup.
    - Ensure the Streams instance has sufficient resources (CPU, memory, PEs) to deploy the application.
    - Verify submission-time parameters are correct.
* **Test Scenarios Not Behaving as Expected:**
  + **Symptom:** Failover doesn't occur, data validation fails unexpectedly, RTO/RPO targets missed.
  + **Checks:**
    - Review the scenario YAML for logical errors or misconfigurations.
    - Increase framework log verbosity to DEBUG to get more detailed operational flow.
    - Verify that the Streams application is properly integrating the Cross-DC Failover Toolkit components.
    - Examine application logs for any toolkit-specific messages that might indicate issues with failover logic.
    - Ensure the fault injected is correctly targeting communication paths between data centers.
    - Verify that monitoring indicators are properly configured to detect failover events.
    - Review Streams instance metrics to identify any resource constraints that might impact failover performance.
    - Check network configurations to ensure that only the intended fault is being introduced.
    - Confirm that the application's configuration parameters for the Cross-DC Failover Toolkit match the expected environment.
* **Interpreting Common Error Messages:**
  + **Framework:** Refer to framework logs for Python exceptions and custom error messages.
  + **Teracloud Streams:** Consult Teracloud Streams documentation (once available) for the meaning of specific API error codes or application log messages (e.g., CDISUxxxx messages mentioned in older IBM Streams docs 30).

A robust troubleshooting guide, ideally expanded over time with experience from actual usage, will significantly enhance the framework's usability and help users resolve issues independently.

## 6. Advanced Topics & Extensibility

This section explores advanced usage patterns and ways to extend the capabilities of the Automated Cross-DC Failover Testing Framework.

### 6.1. Integrating with CI/CD Pipelines

Automating failover tests within a Continuous Integration/Continuous Deployment (CI/CD) pipeline (e.g., Jenkins, GitLab CI, GitHub Actions, Azure DevOps) is a primary goal for ensuring ongoing system resilience.

* **Execution:** The framework's main script (main.py) can be invoked as a build step in the CI/CD pipeline.  
  Bash  
  # Example CI/CD pipeline step  
  python main.py --scenario scenarios/nightly\_failover\_suite.yaml --log-level INFO
* **JUnit XML Reports:** The generated JUnit XML reports (see Section 4.5) are crucial for CI/CD integration. Most CI/CD systems can parse these reports to:
  + Display test results (number of passed, failed, skipped tests).
  + Track test trends over time.
  + Mark a build/pipeline as failed if critical failover tests do not pass.
  + Potentially gate deployments to production based on the success of failover tests in a staging environment.
* **Credential Management:** API keys, SSH credentials, and other secrets required by the framework must be managed securely within the CI/CD environment. Use the CI/CD system's built-in secrets management features (e.g., Jenkins Credentials, GitLab CI variables, GitHub Secrets) rather than hardcoding them in pipeline scripts or framework configuration files checked into version control. The framework can be designed to read these secrets from environment variables.
* **Notifications:** CI/CD pipelines can be configured to send notifications (email, Slack, etc.) upon completion of failover tests, especially if failures occur.
* **Scheduling:** Failover tests can be scheduled to run regularly (e.g., nightly, weekly) to continuously monitor the resilience of Teracloud Streams applications.

Integrating these tests into a CI/CD workflow transforms them from periodic manual checks into an automated, continuous verification process, significantly improving the ability to detect regressions in failover capabilities early.

### 6.2. Customizing Probes and Actions (Inspired by Chaos Toolkit)

The Chaos Toolkit project 12 provides a useful paradigm for structuring reliability experiments using "probes" (to measure the steady-state of a system or verify conditions) and "actions" (to introduce changes or faults into the system). The Teracloud Streams failover testing framework can be designed with similar extensibility in mind.

* **Custom Validation Checks (Probes):**
  + The framework needs flexibility to accommodate application-specific validation requirements.
  + Users should be able to define custom validation logic that can:
    - Analyze application-specific state indicators
    - Validate domain-specific data consistency rules
    - Evaluate whether business functions operate correctly post-failover
    - Measure application-specific performance metrics during recovery
  + **Implementation Approach:**
    - Allow probe configuration via YAML that specifies a Python module and function:

post\_failover\_validation\_checks:

- type: CUSTOM\_PROBE

module: "custom\_probes.my\_app\_probes"

function: "check\_order\_processing\_integrity"

args: {"expected\_completeness": 100, "allowed\_duplicates": 0}

* + - The framework will dynamically import and execute the custom probe function, providing:
      * Access to application metrics and state
      * Toolkit-generated logs (once we understand their format)
      * Data retrieved via the Data Exchange Service
      * Timing information about the failover sequence
    - Custom probes should return standardized results that include:
      * Pass/fail status
      * Detailed metrics or measurements
      * Human-readable descriptions of any detected issues
* **Custom Fault Injection (Actions):**
  + While the framework provides common network faults, users might want to simulate other types of failures (e.g., specific service crashes on Streams nodes, resource exhaustion, DNS failures).
  + The Fault Injection Module could be designed with a pluggable architecture where new fault types can be added by implementing a specific interface or base class.
  + **Implementation Sketch:**
    - Define a FaultInjector base class with methods like apply(params) and remove().
    - New fault types (e.g., StopServiceFaultInjector) would inherit from this base class.
    - Scenario YAML would reference these custom fault types:  
      YAML  
      fault\_scenario:  
       type: CUSTOM\_SERVICE\_STOP  
       module: "custom\_faults.service\_manipulation" # Module containing the custom injector  
       class: "StopCriticalCacheService"  
       parameters: {"service\_name": "streams-cache-svc", "target\_node\_group": "dc1\_app\_tier"}

This approach allows the framework to evolve and adapt to more diverse and complex testing requirements without requiring modifications to its core engine. The long-term value of the framework is significantly enhanced by such extensibility, as failover testing needs often become more sophisticated over time.

### 6.3. Extending the Framework for New Test Cases

Beyond programmatic extensions, users will primarily extend the framework's utility by creating new test scenario definitions in YAML.

* **Guidance for New Scenarios:**
  + Clearly define the objective of the new test case. What specific aspect of failover is being tested?
  + Identify the target Streams application (SAB file).
  + Choose an appropriate fault scenario (type and parameters) that will realistically test the objective.
  + Define clear pre-failover conditions and data injection strategies if state validation is required.
  + Specify precise post-failover validation checks. What constitutes a successful recovery for this scenario?
  + Set realistic RTO and RPO targets if applicable.
* **Designing Testable Streams Applications:** For effective testing with this framework, Teracloud Streams applications should be designed with testability in mind:
  + **Utilize Data Exchange:** Expose appropriate input and output streams via endpoint operators to facilitate test data injection and state validation.
  + **Logs:** Implement logging that clearly indicates:
    - Application state transitions
    - Cross-DC Failover Toolkit activities
    - Detection of failover conditions
    - State synchronization events
    - Completion of failover steps
  + **Instrumentation:** Add custom metrics that provide visibility into:
    - Data processing rates in each DC
    - Pending operation queues
    - Replication lag metrics (if applicable)
    - Resource utilization during normal operation and failover
  + **Fault Tolerance:** Design application components with appropriate error handling and recovery logic to complement the Cross-DC Failover Toolkit's capabilities.
  + **State Management:** Consider how application state is maintained and synchronized across DCs:
    - Use idempotent operations where possible
    - Implement appropriate transaction boundaries
    - Consider eventual consistency requirements
    - Ensure state can be validated via observable outputs
  + **Configurable Parameters:** Allow key toolkit parameters to be configurable at submission time to facilitate testing different failover scenarios.

By following these guidelines, you can create applications that not only leverage the Cross-DC Failover Toolkit effectively but also provide the necessary observability for comprehensive testing.

## 7. Points Requiring Clarification/Confirmation from Teracloud SMEs

This section consolidates critical questions that need to be answered to optimize the implementation of the Automated Cross-DC Failover Testing Framework. While the framework is designed to be adaptable based on progressive discovery of Teracloud Streams features, obtaining clear and definitive answers to these questions will significantly enhance its effectiveness and accuracy.

The questions are organized by component to facilitate focused discussions with Teracloud Subject Matter Experts. As answers are obtained, this document and the framework implementation should be updated accordingly.

### 7.1. REST Management API (Teracloud Streams 7.2+)

The "enhanced REST API" for managing domains, instances, and jobs is central to the framework.9 However, detailed documentation (like OpenAPI specs or ReDoc) is currently inaccessible.1

* **Authentication:**
  1. What are the precise authentication methods supported by the Teracloud Streams 7.2 REST Management API (e.g., API Key, OAuth 2.0, Basic Authentication)?
  2. If API Key based:
     + What is the exact HTTP header name to be used (e.g., X-API-Key, Authorization)?
     + How are API keys generated, managed, and revoked?
  3. If OAuth 2.0 based:
     + What grant types are supported (e.g., Client Credentials, Authorization Code)?
     + What is the URL for the token endpoint?
     + What scopes are required for the framework's operations?
  4. Are there different authentication requirements for domain, instance, and job-level operations?
* Endpoints & Functionality:  
  5. Is there an official OpenAPI/Swagger specification or ReDoc documentation available for the v7.2 REST API? The URL https://doc.streams.teracloud.com/external-reference/redoc-static.html mentioned in documentation searches is inaccessible.1  
  6. Please provide or confirm the exact endpoint paths, HTTP methods, request parameters (query and body), and response structures for the following operations:  
  \* Submitting/deploying a Streams Application Bundle (SAB) file.  
  \* Starting a deployed job.  
  \* Stopping/canceling a running job (and options for graceful vs. forced stop).  
  \* Retrieving a list of all jobs in an instance (with filtering options, e.g., by status, name).  
  \* Getting detailed status and information for a specific job (including PE status, host allocation).  
  \* Retrieving a list of all instances in a domain.  
  \* Getting health status and details for a specific instance.  
  \* Getting health status and details for a domain.  
  \* Retrieving application/job metrics (and what types of metrics are available through this API).  
  \* Retrieving application/job logs (including options for filtering by PE, time range, severity, and pagination/streaming for large logs).  
  7. How are submission-time parameters passed when deploying a SAB file via the API?
* Request/Response Formats:  
  8. Is JSON the standard data format for all request and response bodies? Are there specific JSON schemas available?  
  9. How are file uploads (e.g., SAB files) handled (e.g., multipart/form-data)?
* Rate Limiting & Error Handling:  
  10. Are there any API rate limits imposed on clients? If so, what are the limits, and how are they communicated (e.g., HTTP headers like X-RateLimit-Limit, X-RateLimit-Remaining)?  
  11. Could a comprehensive list of API error codes, their meanings, and recommended client actions be provided?
* API Versioning:  
  12. What is the base URL structure for the API? How is API versioning handled in the URL or headers (e.g., is /streams/v7/ the correct pattern based on Table 3.1.3.1 assumptions)?

### 7.2. Data Exchange Service (Teracloud Streams 7.2+)

The new Data Exchange feature is critical for test data injection and state validation.9 Detailed documentation is currently inaccessible.3

* API Details:  
  13. What are the specific REST API endpoints for injecting data into a Streams application's endpoint operator?  
  14. What are the specific REST API endpoints for retrieving data from a Streams application's endpoint operator?  
  15. Does the Data Exchange API use the same authentication mechanism as the REST Management API?  
  16. Are there specific request headers or parameters required for Data Exchange API calls?
* Endpoint Operators:  
  17. What are the exact SPL names of the "endpoint operators" used to enable Data Exchange within a Streams application?  
  18. Which SPL toolkit contains these endpoint operators?  
  19. What are the mandatory and optional SPL parameters for these endpoint operators (e.g., for defining data schema, port, security, buffering)?  
  20. How is an endpoint operator instance named or identified in the REST API calls (e.g., is it the SPL operator's alias)?
* Data Formats & Schemas:  
  21. What data formats are supported for injection and retrieval (e.g., JSON, CSV, binary)? If JSON, what is the expected structure (e.g., an array of objects representing tuples)?  
  22. How are SPL tuple schemas (attribute names and types) mapped to the chosen data format (e.g., JSON types)?  
  23. Are there any limitations on the size of data payloads per request or overall throughput for the Data Exchange service?  
  24. How does the service handle schema mismatches between the provided data and the SPL endpoint operator's expected schema?
* Transactional Guarantees & Performance:  
  25. What are the message delivery guarantees for data injected via the Data Exchange service (e.g., at-least-once, at-most-once into the Streams application)?  
  26. What are the expected performance characteristics (latency, throughput) of the Data Exchange service under various loads?  
  27. Does the retrieval endpoint support blocking calls, polling, or streaming of data?

### 7.3. Cross-DC Failover Toolkit (com.teracloud.streams.crossdcfailover)

The com.teracloud.streams.crossdcfailover toolkit is fundamental to the applications being tested as it enables the cross-DC failover capabilities we aim to validate. While we know the toolkit provides "application-level failover across two data centers enabling Disaster Recovery (DR) and Business Continuity (BC)," detailed operational specifics are needed to optimize our testing approach. The following questions seek clarification on how applications interact with and utilize this toolkit.

* Documentation Access:  
  28. Can detailed documentation (SPLDOC, developer guides, examples) for the com.teracloud.streams.crossdcfailover toolkit be provided?
* Configuration Parameters:  
  29. What are all the configuration parameters available for the operators and functions within this toolkit?  
  30. How are these parameters specified by the application developer (e.g., as SPL operator parameters at compile time, as submission-time parameters when deploying the SAB via REST API, or loaded from external configuration files referenced by the toolkit at runtime)?  
  31. What are the data types, valid value ranges, and default values for each parameter?  
  32. Are there example configurations for common active-passive DR scenarios?
* Operational Behavior & Workflow:  
  33. What is the detailed mechanism used by the toolkit to detect primary DC failure or application unhealthiness? (e.g., heartbeating, loss of connection to critical service, external trigger).  
  34. What is the step-by-step operational workflow the toolkit (and application using it) follows during a failover event?  
  35. What is the expected behavior of applications using this toolkit under various fault conditions (e.g., full network partition between DCs, partial network issues, primary Streams instance failure, underlying host failures)?  
  36. How does the toolkit handle split-brain scenarios, if applicable?
* State Replication/Synchronization:  
  37. How does the toolkit manage or assist in the replication or synchronization of application state between the primary and secondary DCs?  
  38. What consistency models are supported or guaranteed for application state during and after failover (e.g., eventual consistency, strong consistency for certain data)?  
  39. Does the toolkit integrate with or require specific external state stores (e.g., replicated databases, distributed caches)?
* Monitoring & Observability:

40. What are the recommended methods to observe the current status, health, or role (primary/secondary) of an application using this toolkit?

41. What specific log patterns, metrics, or status streams does the toolkit produce that can be used to detect and monitor failover events?

42. Are there best practices for monitoring the toolkit's automatic failover behavior for both testing and production environments?

* Dependencies & Limitations:  
  43. Does the toolkit have any dependencies on specific infrastructure components (e.g., load balancers with particular capabilities, replicated storage, specific network configurations)?  
  44. Are there any known limitations or unsupported scenarios for the toolkit?

### 7.4. Python Client/SDK

1. Is there an official Teracloud Streams Python SDK or client library specifically for version 7.2 that is recommended for interacting with the REST Management API and Data Exchange service? The IBM Streams Python SDK (streamsx.rest) 15 is for a predecessor product. Teracloud Streams 7.2 introduces Python operator support 9 but an official client SDK for management APIs is not explicitly mentioned in accessible materials. Generic cloud provider client library pages 59 are not specific to Streams.

### 7.5. Monitoring and Metrics

1. What specific metrics (available via API, Prometheus, or JMX) are recommended for monitoring:
   * The health and status of a Streams application during normal operation?
   * The progress and success of an application failover event?
   * The performance of an application post-failover in the secondary DC?
2. Does Teracloud Streams 7.2 (or applications built with its toolkits) expose metrics natively in Prometheus format? If so, what is the typical scrape endpoint and configuration?
3. If JMX is the primary mechanism for detailed performance monitoring, could a list of key MBeans and attributes relevant to job performance, PE status, and toolkit operation be provided?

Obtaining answers to these questions is paramount. The current lack of detailed, accessible primary documentation for these new and enhanced Teracloud Streams 7.2 features represents the most significant risk to the timely and accurate development of the proposed testing framework. The categorization of these questions by component should assist Teracloud SMEs in efficiently addressing these critical information gaps. This document will require substantial updates in Sections 3, 4, and 5 once these clarifications are received.

## 8. Appendix

### 8.1. Example Configuration Files

* **Sample config.yaml for the Framework (Illustrative - many values depend on SME feedback):**  
  *# Global Framework Configuration (config.yaml.example)*

*# Teracloud Streams API Endpoints (Primary DC)*

streams\_api\_base\_url\_dc1: "https://<streams\_api\_host\_dc1>:<port>/streams/v7" streams\_api\_auth\_type\_dc1: "API\_KEY" *# Update based on actual authentication method* streams\_api\_key\_header\_dc1: "X-API-Key" *# Update based on actual header name* streams\_api\_key\_value\_dc1: ${DC1\_STREAMS\_API\_KEY} *# Use environment variable for security*

*# Teracloud Streams API Endpoints (Secondary DC)*

streams\_api\_base\_url\_dc2: "https://<streams\_api\_host\_dc2>:<port>/streams/v7" streams\_api\_auth\_type\_dc2: "API\_KEY" streams\_api\_key\_header\_dc2: "X-API-Key" streams\_api\_key\_value\_dc2: ${DC2\_STREAMS\_API\_KEY}

*# SSH Configuration for Fault Injection Nodes*

default\_ssh\_user: "tester"

default\_ssh\_key\_path: "~/.ssh/id\_rsa\_streams\_tester" *# Path to private key*

*# Logging Configuration*

log\_file\_path: "logs/failover\_tester.log"

log\_level: "INFO" *# Options: DEBUG, INFO, WARNING, ERROR, CRITICAL* log\_rotation\_max\_bytes: 10485760 *# 10MB*

log\_rotation\_backup\_count: 5

*# Reporting* reports\_dir: "reports"

junit\_report\_format: true

*# Default Timeouts*

default\_api\_timeout\_seconds: 60 *# For individual API calls* default\_job\_deploy\_timeout\_seconds: 300 *# For a job to become 'running'* default\_failover\_detection\_timeout\_seconds: 600 *# Max time to wait for failover completion*

*# Monitoring Configuration*

monitoring\_interval\_seconds: 5 *# How frequently to check for failover indicators* metrics\_collection\_interval\_seconds: 15 *# How frequently to collect performance metrics*

*# Cross-DC Failover Toolkit Configuration* *# This section will be populated as we learn more about toolkit configuration parameters*

toolkit\_observability: *# List of known log patterns that indicate toolkit actions* log\_patterns: - pattern: "Initiating failover sequence" event: "FAILOVER\_START" - pattern: "Failover completed successfully" event: "FAILOVER\_COMPLETE" *# To be expanded as we discover more about toolkit behavior*

* **Sample Test Scenario YAML File (scenarios/example\_network\_latency.yaml):**  
  YAML  
  *# Test Scenario: Full Network Partition Between DCs*

test\_id: "TS001\_FULL\_DC\_PARTITION"

description: "Simulate complete network partition between primary and secondary DCs and verify automatic failover"

*# Target Streams Instances*

target\_primary\_instance\_id: "<your\_primary\_streams\_instance\_id>" target\_secondary\_instance\_id: "<your\_secondary\_streams\_instance\_id>"

*# Application Under Test*

streams\_application\_sab: "apps/FailoverTestApp.sab"

submission\_params:

replication\_mode: "async" *# Example parameter - update based on actual toolkit parameters*

failover\_detection\_timeout: 30 *# Example parameter - update based on actual toolkit parameters*

*# Test Data for Validation*

pre\_failover\_data:

generator\_script: "data\_generators/transaction\_stream.py"

count: 1000

rate: 10 *# records per second*

target\_endpoint: "TransactionIngest" *# Endpoint name in application*

*# Fault Injection*

fault\_scenario:

type: "NETWORK\_PARTITION"

parameters:

source\_dc\_subnets: ["192.168.1.0/24"] *# Subnet of DC1 application nodes* target\_dc\_subnets: ["192.168.2.0/24"] *# Subnet of DC2 application nodes* protocol: "all"

direction: "BOTH"

duration\_seconds: 300 *# Keep fault active for 5 minutes*

*# Failover Monitoring*

failover\_monitoring:

expected\_detection\_time\_seconds: 60 *# Expected time for toolkit to detect the partition*

expected\_recovery\_time\_seconds: 180 *# Expected time for complete failover* indicators:

- type: "METRIC"

name: "active\_pes\_in\_secondary\_dc"

condition: "INCREASING"

- type: "LOG\_PATTERN"

pattern: "Failover process initiated"

*# Validation Checks*

post\_failover\_validation:

data\_integrity:

retrieval\_endpoint: "TransactionEgress" *# Endpoint for retrieving processed data*

expected\_data\_loss\_percentage: 0.0 *# Expected zero data loss* allow\_duplicates: false *# Duplicates should not be present*

performance: max\_throughput\_reduction\_percent: 10 *# Max acceptable performance degradation*

### 8.2. Sample Teracloud Streams Application (Conceptual)

The following SPL example is conceptual and illustrates how an application might integrate with the Cross-DC Failover Toolkit based on our current understanding. This example will need refinement once detailed toolkit documentation is obtained, but it provides a starting point for visualizing how a test application might be structured.

```spl

// Conceptual SPL Application: FailoverTestApp.spl

// This application demonstrates integration with the Cross-DC Failover Toolkit

// It processes a stream of transactions with failover capabilities

namespace application;

// Import the Cross-DC Failover Toolkit

use com.teracloud.streams.crossdcfailover::\*;

// Import the Data Exchange Service toolkit for test data injection and retrieval

use com.teracloud.streams.dataexchange::\*;

/\*\*

\* Main composite for the failover-capable transaction processing application

\*/

composite FailoverTestApp {

param

// Example parameters that might be used to configure failover behavior

expression<rstring> $activeDcRole: getSubmissionTimeValue("activeDcRole", "PRIMARY");

expression<int32> $heartbeatIntervalSec: getSubmissionTimeValue("heartbeatInterval", 5);

expression<int32> $failoverThresholdSec: getSubmissionTimeValue("failoverThreshold", 30);

type

// Data schema for test transactions

Transaction = rstring id, timestamp createTime, rstring payload, int32 value;

// Schema with processing results

ProcessedTransaction = Transaction, timestamp processTime, rstring processingDc;

graph

// Data injection endpoint for test data

stream<Transaction> RawTransactions = DataExchangeSource() {

param

endpointName: "TransactionIngest";

// Additional parameters would depend on the actual Data Exchange implementation

}

// Apply failover-aware processing using the Cross-DC Failover Toolkit

// This is conceptual and will need to be updated based on actual toolkit operators

@failover(role: $activeDcRole, heartbeatInterval: $heartbeatIntervalSec)

stream<ProcessedTransaction> ProcessedTransactions = FailoverAwareProcessor(RawTransactions) {

param

// Parameters for the Cross-DC Failover Toolkit would be specified here

stateReplicationMode: getSubmissionTimeValue("replicationMode", "async");

// Example state handling logic

state: {

mutable map<rstring, Transaction> pendingTransactions = {};

mutable int64 processedCount = 0;

}

onTuple RawTransactions: {

// Store in state map to demonstrate state that needs to be preserved during failover

pendingTransactions[id] = RawTransactions;

// Process the transaction

ProcessedTransaction result = (ProcessedTransaction){};

result.id = id;

result.createTime = createTime;

result.payload = payload;

result.value = value;

result.processTime = getTimestamp();

result.processingDc = $activeDcRole;

// Remove from pending and increment counter

removeM(pendingTransactions, id);

processedCount++;

// Submit result

submit(result, ProcessedTransactions);

}

}

// Expose processed transactions via Data Exchange for validation

() as ResultsExport = DataExchangeSink(ProcessedTransactions) {

param

endpointName: "TransactionEgress";

// Additional parameters would depend on the actual Data Exchange implementation

}

// Example of exposing metrics for monitoring toolkit behavior

() as FailoverMetrics = MetricsExporter() {

logic

onProcess: {

// Export metrics about the failover state and processing

// These details would depend on what metrics the toolkit exposes

publishMetric("transactions.processed.count", processedCount);

publishMetric("transactions.pending.count", size(pendingTransactions));

publishMetric("failover.active\_dc", $activeDcRole);

// Additional metrics would be exposed based on toolkit capabilities

}

}

}

/\*\*

\* This custom operator would encapsulate the Cross-DC Failover Toolkit integration

\* The actual implementation would use the specific operators and functions provided by the toolkit

\*/

custom operator FailoverAwareProcessor(input In; output Out) {

// This is a placeholder for the actual implementation that would use

// the Cross-DC Failover Toolkit's specific constructs

// Once we have detailed toolkit documentation, this would be replaced with

// the appropriate operators and logic

}

This example illustrates key concepts that would likely be part of a failover-capable application:

1. Integration with the Cross-DC Failover Toolkit
2. State management considerations
3. Data exchange endpoints for testing
4. Metrics for observability

The actual implementation details would need to be adjusted based on the specific operators, functions, and parameters provided by the Cross-DC Failover Toolkit.

#### Works cited

1. accessed December 31, 1969, <https://doc.streams.teracloud.com/external-reference/redoc-static.html>
2. accessed December 31, 1969, [https://doc.streams.teracloud.com/com.ibm.streams.admin.doc/doc/managemonitorstreamsapi.html?hl=rest%2Cmanagement%2Capi%2Cservice](https://doc.streams.teracloud.com/com.ibm.streams.admin.doc/doc/managemonitorstreamsapi.html?hl=rest,management,api,service)
3. accessed December 31, 1969, [https://doc.streams.teracloud.com/com.ibm.streams.dev.doc/doc/enabling-streams-data-exchange.html?hl=data%2Cexchange](https://doc.streams.teracloud.com/com.ibm.streams.dev.doc/doc/enabling-streams-data-exchange.html?hl=data,exchange)
4. accessed December 31, 1969, [https://doc.streams.teracloud.com/com.ibm.streams.toolkits.doc/spldoc/dita/tk$com.teracloud.streams.crossdcfailover/tk$com.teracloud.streams.crossdcfailover.html?hl=cross%2Cdc](https://doc.streams.teracloud.com/com.ibm.streams.toolkits.doc/spldoc/dita/tk$com.teracloud.streams.crossdcfailover/tk$com.teracloud.streams.crossdcfailover.html?hl=cross,dc)
5. accessed December 31, 1969, <https://doc.streams.teracloud.com/com.ibm.streams.ref.doc/doc/api-container.html>
6. accessed December 31, 1969, <https://doc.streams.teracloud.com/com.ibm.streams.dev.doc/doc/enabling-streams-data-exchange.html>
7. accessed December 31, 1969, <https://doc.streams.teracloud.com/com.ibm.streams.ref.doc/doc/spltoolkits_intro.html>
8. accessed December 31, 1969, <https://doc.streams.teracloud.com/com.ibm.streams.toolkits.doc/spldoc/dita/tk$com.teracloud.streams.crossdcfailover/tk$com.teracloud.streams.crossdcfailover.html>
9. Teracloud Streams Version 7.2.0 Release Notes, accessed May 14, 2025, <https://www.streams.teracloud.com/releases/7.2.0>
10. Teracloud Streams Version 7.2.0 Release Notes, accessed May 14, 2025, <https://streams.teracloud.com/releases/7.2.0>
11. tcconfig - PyPI, accessed May 14, 2025, <https://pypi.org/project/tcconfig/>
12. Chaos Engineering Toolkit & Orchestration for Developers - GitHub, accessed May 14, 2025, <https://github.com/chaostoolkit/chaostoolkit>
13. Python requests best practices for data engineers - Y42, accessed May 14, 2025, <https://www.y42.com/blog/python-requests-best-practices-for-data-engineers>
14. Python's Requests Library (Guide), accessed May 14, 2025, <https://realpython.com/python-requests/>
15. The Python REST API - IBM Streams, accessed May 14, 2025, <http://ibmstreams.github.io/streamsx.documentation/docs/python/1.6/python-appapi-devguide-6/>
16. Client - Paramiko documentation, accessed May 14, 2025, <https://docs.paramiko.org/en/stable/api/client.html>
17. 1. - tcconfig's documentation!, accessed May 14, 2025, <https://tcconfig.readthedocs.io/en/stable/pages/introduction/index.html>
18. ldx/python-iptables - GitHub, accessed May 14, 2025, <https://github.com/ldx/python-iptables>
19. aalekhpatel07/partition-sim: Simulate a network partition between a distributed cluster of nodes. - GitHub, accessed May 14, 2025, <https://github.com/aalekhpatel07/partition-sim>
20. Python YAML Configuration Guide - GitHub, accessed May 14, 2025, <https://github.com/cobanov/python-yaml-guide>
21. Parse a YAML file in Python - GeeksforGeeks, accessed May 14, 2025, <https://www.geeksforgeeks.org/parse-a-yaml-file-in-python/>
22. JUnit-XML based report upload using PyTest | BrowserStack Docs, accessed May 14, 2025, <https://www.browserstack.com/docs/test-management/upload-reports-cli/frameworks/pytest>
23. Common JUnit XML Format & Examples - GitHub, accessed May 14, 2025, <https://github.com/testmoapp/junitxml>
24. prometheus-api-client - PyPI, accessed May 14, 2025, <https://pypi.org/project/prometheus-api-client/>
25. jmxquery · PyPI, accessed May 14, 2025, <https://pypi.org/project/jmxquery/>
26. Python Logging Tutorial: How-To, Basic Examples & Best Practices - Sematext, accessed May 14, 2025, <https://sematext.com/blog/python-logging/>
27. Logging HOWTO — Python 3.13.3 documentation, accessed May 14, 2025, <https://docs.python.org/3/howto/logging.html>
28. chaostoolkit · PyPI, accessed May 14, 2025, <https://pypi.org/project/chaostoolkit/0.3.0/>
29. Reference, accessed May 14, 2025, <https://doc.streams.teracloud.com/com.ibm.streams.ref.doc/doc/reference-container.html>
30. Application manager service, accessed May 14, 2025, <https://doc.streams.teracloud.com/com.ibm.streams.msg.doc/cdisucontainer.html>
31. Teracloud Streams messages, accessed May 14, 2025, <https://doc.streams.teracloud.com/com.ibm.streams.msg.doc/parentcontainer.html>
32. Developing, accessed May 14, 2025, <https://doc.streams.teracloud.com/com.ibm.streams.dev.doc/doc/dev-container.html>
33. Developing streams application with views, accessed May 14, 2025, <https://doc.streams.teracloud.com/com.ibm.streams.dev.doc/doc/views_developing.html>
34. User Guide to Cloud Stream API - EEX, accessed May 14, 2025, <https://www.eex.com/fileadmin/EEX/Downloads/Market_Data/EEX_Group_DataSource/API/20250505_CloudStream_API_User_Guide.pdf>
35. REST API Endpoints - Terra Docs, accessed May 14, 2025, <https://docs.tryterra.co/reference/streaming-api/api-endpoints>
36. Data Exchange Service - Developer Documentation - developer.siemens.com, accessed May 14, 2025, <https://developer.siemens.com/insights-hub/docs/apis/analytics-dataexchange/api-dataexchange-overview.html>
37. What Is Data Exchange? | Teradata, accessed May 14, 2025, <https://www.teradata.com/insights/data-platform/what-is-data-exchange>
38. Data Exchange API - Tricentis, accessed May 14, 2025, <https://documentation.tricentis.com/neoload/2024.2/en/content/apis/data_exchange_api.htm>
39. Introducing the Synthetic Data Generator - Build Datasets with Natural Language, accessed May 14, 2025, <https://huggingface.co/blog/synthetic-data-generator>
40. About Streaming Data - kdb products, accessed May 14, 2025, <https://code.kx.com/insights/1.12/microservices/stream-processor/index.html>
41. DSA Failover Configurations - Teradata Data Mover, accessed May 14, 2025, <https://docs.teradata.com/r/Enterprise_IntelliFlex_Lake_VMware/Teradata-Data-Mover-Installation-Configuration-and-Upgrade-Guide-for-Customers-20.01/Installing-and-Configuring-Software/Configuring-the-Environment/High-Availability-Overview/DSA-Failover-Configurations>
42. How do I enable cross-datacenter failover programatically in the Cassandra Java driver v4?, accessed May 14, 2025, <https://stackoverflow.com/questions/72710237/how-do-i-enable-cross-datacenter-failover-programatically-in-the-cassandra-java>
43. Python Requests Tutorial - Tutorialspoint, accessed May 14, 2025, <https://www.tutorialspoint.com/requests/index.htm>
44. How to Run Commands on Remote Windows Machine Using Python Paramiko Module, accessed May 14, 2025, <https://www.calsoftinc.com/blogs/run-commands-remote-windows-machine-using-python-paramiko-module.html>
45. Paramiko SFTP: A Guide with Examples - SFTPCloud, accessed May 14, 2025, <https://sftpcloud.io/learn/python/paramiko-sftp-examples>
46. Python SSH Client - Paramiko. SSH with Python. - YouTube, accessed May 14, 2025, <https://www.youtube.com/watch?v=AkcxJWbHV0w>
47. tcconfig - PyPI, accessed May 14, 2025, <https://pypi.org/project/tcconfig/0.6.0/>
48. Basic WAN emulation - ALLOUTPUT.com, accessed May 14, 2025, <http://alloutput.com/technology/basic-wan-emulation/>
49. 25 Most Frequently Used Linux IPTables Rules Examples - Knowledgebase - tzulo, inc., accessed May 14, 2025, <https://www.tzulo.com/crm/knowledgebase/9/25-Most-Frequently-Used-Linux-IPTables-Rules-Examples-.html>
50. iptables - ArchWiki, accessed May 14, 2025, <https://wiki.archlinux.org/title/Iptables>
51. Practical Guide to Iptables | A Beginner's Tutorial to the Linux Firewall - Hostman, accessed May 14, 2025, <https://hostman.com/tutorials/iptables-overview-and-practical-use/>
52. iptables command in Linux with Examples - GeeksforGeeks, accessed May 14, 2025, <https://www.geeksforgeeks.org/iptables-command-in-linux-with-examples/>
53. Prometheus API: From Basics to Advanced Usage - Last9, accessed May 14, 2025, <https://last9.io/blog/prometheus-api/>
54. Monitoring the Service using JMX - TIBCO Product Documentation, accessed May 14, 2025, <https://docs.tibco.com/pub/sf-pysrv/latest/doc/html/TIB_sf-pysrv_install/_shared/install/topics/monitoring_the_service_using_jmx.html>
55. Debian -- Details of package python3-junit.xml in buster, accessed May 14, 2025, <https://packages.debian.org/buster/python/python3-junit.xml>
56. Getting Started with Pytest - GeeksforGeeks, accessed May 14, 2025, <https://www.geeksforgeeks.org/getting-started-with-pytest/>
57. Pytest Tutorial - Tutorialspoint, accessed May 14, 2025, <https://www.tutorialspoint.com/pytest/index.htm>
58. Teracloud Streams - Real-time Analytics, accessed May 14, 2025, <https://streams.teracloud.com/>
59. Python Cloud Client Libraries, accessed May 14, 2025, <https://cloud.google.com/python/docs/reference>
60. Cloud Computing Managed Services | Teracloud.io | AWS Partner, accessed May 14, 2025, <https://www.teracloud.io/>