# Design Plan: Automated Cross-DC Failover Testing Framework for Teracloud Streams

Version: 1.0

Date: October 26, 2023

Status: Design Final

## 1. Introduction

This document outlines the design for an automated cross-Data Center (DC) failover testing framework for Teracloud Streams. The framework aims to provide a robust, repeatable, and comprehensive solution for validating the resilience and recovery capabilities of Teracloud Streams applications deployed in a cross-DC configuration. This design is based on a thorough review of available Teracloud Streams documentation, including the REST API, Data Exchange features, and the Cross-DC Failover Toolkit, as announced for Teracloud Streams Version 7.2.0.1

The primary objective of this framework is to automate the entire test lifecycle: setup, execution, monitoring, validation, and reporting. It will leverage Python as the core automation language, utilizing standard libraries for interacting with Teracloud Streams interfaces and for simulating various failure conditions. This document details the architectural components, interaction patterns with Teracloud Streams, data strategies, failure simulation techniques, and deployment considerations.

## 2. Goals and Scope

### 2.1. Goals

The principal goals of the automated testing framework are:

* **Comprehensive Validation:** To rigorously test the end-to-end cross-DC failover mechanisms of Teracloud Streams applications.
* **Automation:** To achieve full automation of the test lifecycle, minimizing manual intervention for setup, execution, failure injection, monitoring, and results analysis.
* **Repeatability:** To ensure tests can be executed consistently, producing reliable results for regression testing and performance benchmarking.
* **Realism:** To simulate a variety of failure scenarios, including network issues, service outages, and complete DC failures, under realistic conditions.
* **Measurability:** To quantify key failover metrics such as Recovery Time Objective (RTO) and Recovery Point Objective (RPO).
* **Extensibility:** To design a framework that can be extended to accommodate new test scenarios and evolving Teracloud Streams features.

### 2.2. Scope

**In Scope:**

* Automation of test environment setup, including deployment of test Streams applications configured with the Cross-DC Failover Toolkit.
* Interaction with the Teracloud Streams REST Management API for application lifecycle management (deployment, monitoring, control).1
* Utilization of the Teracloud Streams Data Exchange REST API for injecting test data and retrieving output data for validation.1
* Monitoring and observation of the Teracloud Streams Cross-DC Failover Toolkit behavior through its configuration
* mechanisms, logs, and metrics.Simulation of various failure types:
  + API-driven failures (e.g., stopping jobs/instances).
  + Network failures (latency, packet loss, partitions) between DCs and between application components.
  + Process/node failures within a DC.
* Monitoring of Streams application health, job status, and relevant system metrics during tests, potentially via API and Prometheus/JMX integration.4
* Validation of data integrity, consistency, and application availability post-failover.
* Generation of detailed test reports in standard formats (e.g., JUnit XML 7, HTML).

**Out of Scope:**

* Performance testing of Teracloud Streams beyond what is necessary to validate failover behavior (e.g., maximum throughput testing is not a primary goal).
* Testing of underlying infrastructure resilience (e.g., storage replication, network hardware) except as it impacts Streams failover.
* Development of custom Teracloud Streams operators beyond what is needed for test applications and data exchange.
* Security vulnerability testing of Teracloud Streams.

## 3. Framework Architecture

The automated testing framework will consist of a central Orchestrator and optional, lightweight Agents deployed on specific nodes for granular failure injection.

### 3.1. High-Level Architecture

The framework will adopt a modular architecture comprising the following key components:

* **Test Orchestrator:** A Python application responsible for overall test execution management. It will:
  + Parse test configurations.
  + Manage the lifecycle of test Streams applications via the REST Management API.
  + Coordinate data injection and retrieval using the Data Exchange API.
  + Initiate failure simulations (directly via API or indirectly via Agents).
  + Monitor system and application state.
  + Collect and analyze test results.
  + Generate reports.
* **Test Configuration Module:** Defines test scenarios, application parameters, failure conditions, and validation criteria, typically in a structured format like YAML or JSON.
* **Teracloud Streams Interaction Module:** Python modules that encapsulate interactions with:
  + Teracloud Streams REST Management API (using a standard HTTP client like requests).
  + Teracloud Streams Data Exchange API (using requests).
  + Cross-DC Failover Toolkit (via API or configuration adjustments, if available).
* **Failure Injection Module:** Components responsible for inducing failures. This includes:
  + API-based injectors (interacting with the Streams Management API).
  + Network failure injectors (using tcconfig 9 and python-iptables 11 or direct iptables commands, potentially via Agents).
  + Process/Node failure injectors (using paramiko 12 to control Agents or execute commands remotely).
* **Monitoring & Validation Module:** Collects metrics from Teracloud Streams (via API, Prometheus 4, or JMX 6) and validates application state and data consistency post-failover.
* **Reporting Module:** Generates test summaries, detailed logs, and pass/fail status in formats like HTML and JUnit XML.7
* **(Optional) Test Agents:** Lightweight Python scripts deployed on specific Teracloud Streams nodes or network gateways. Controlled by the Orchestrator via paramiko 12, they execute low-level failure injection commands (e.g., iptables, tc, process kill).

The interaction between these components ensures a coordinated approach to testing. For instance, the Orchestrator, guided by the Test Configuration, would use the Teracloud Streams Interaction Module to deploy an application. It would then instruct the Failure Injection Module to simulate a network partition between data centers, while the Monitoring & Validation Module continuously checks application status and data flow. This separation of concerns promotes modularity and maintainability.

### 3.2. Component Interaction Diagram

Code snippet

graph TD  
 A --> B(Test Orchestrator - Python);  
 B -- Manages/Monitors --> C{Teracloud Streams REST Mgmt API};  
 B -- Injects/Retrieves Data --> D{Teracloud Streams Data Exchange API};  
 B -- Configures/Triggers? --> E{Cross-DC Failover Toolkit};  
 B -- Initiates Failures --> F(Failure Injection Module);  
 F -- API Calls --> C;  
 F -- Commands via Paramiko --> G((Optional) Test Agents);  
 G -- Modifies Network --> H[Network Infrastructure - tc, iptables];  
 G -- Kills Processes --> I;  
 B -- Collects Metrics/Status --> J(Monitoring & Validation Module);  
 J -- Queries --> C;  
 J -- Queries --> D;  
 J -- Queries --> K{Prometheus/JMX Endpoints};  
 B -- Generates --> L;  
  
 subgraph Teracloud Streams Environment - DC1  
 C1[...]  
 D1[...]  
 E1[...]  
 I1[...]  
 K1[...]  
 end  
 subgraph Teracloud Streams Environment - DC2  
 C2[...]  
 D2[...]  
 E2[...]  
 I2[...]  
 K2[...]  
 end  
 C --- C1 & C2  
 D --- D1 & D2  
 E --- E1 & E2  
 I --- I1 & I2  
 K --- K1 & K2  
 H --- H\_DC1\_GW & H\_DC2\_GW & H\_InterDC

### 3.3. Core Framework Components and Responsibilities

The following table details the primary software components of the testing framework:

|  |  |  |
| --- | --- | --- |
| **Component** | **Responsibility** | **Key Technologies/Libraries** |
| **Test Orchestrator** | Main control program; manages test lifecycle, coordinates other modules, makes decisions based on test configuration and runtime state. | Python |
| **Configuration Parser** | Reads and validates test scenario definitions (e.g., from YAML files). | Python, PyYAML (or similar) |
| **Streams API Client** | Handles all HTTP/S communication with Teracloud Streams REST Management and Data Exchange APIs. | Python, requests library 15 |
| **Failure Injection Engine** | Orchestrates various failure types: API-driven, network manipulation, process/node failures. | Python, requests, paramiko 12, tcconfig 9, python-iptables (or direct calls) 11 |
| **(Optional) Test Agent** | Executes low-level commands on target nodes for failure injection (e.g., iptables, tc, kill) under Orchestrator's SSH control. | Python, paramiko (for Orchestrator to Agent communication) |
| **Monitoring Client** | Gathers metrics and status from Streams APIs, Prometheus, and potentially JMX. | Python, requests, prometheus-api-client 4, jmxquery 6 |
| **Validation Engine** | Compares actual outcomes against expected results (e.g., RTO, RPO, data consistency, service availability). | Python |
| **Reporting Generator** | Produces human-readable (HTML) and machine-readable (JUnit XML) test reports. | Python, pytest-html (or custom templating), junit-xml 7 |
| **Test Data Generator/Manager** | Creates and manages deterministic, identifiable test data for injection and validation. | Python |

This component-based design allows for independent development and testing of each part of the framework. For example, the Streams API Client can be developed and unit-tested separately before being integrated into the Test Orchestrator. This modularity is essential for building a complex system and facilitates future enhancements, such as adding support for new types of failure injections or different monitoring systems.

## 4. Interaction with Teracloud Streams Interfaces

The framework will interact with Teracloud Streams primarily through its documented REST APIs and the Cross-DC Failover Toolkit. The Teracloud Streams 7.2.0 release notes indicate an "enhanced REST API" for managing domains, instances, and jobs, a "new Data Exchange feature" for tuple injection/retrieval, and a "new crossdcfailover toolkit".1

### 4.1. REST Management API

Interactions with the Teracloud Streams REST Management API will be performed using a standard Python HTTP client, specifically the requests library.15 The streamsx.rest library, primarily associated with IBM Streams or application-level interactions 17, is **not** considered applicable for interacting with the new Teracloud Streams 7.2.0 management plane APIs.

Key operations to be automated include:

* **Authentication:** Programmatic handling of API authentication (assumed to be token-based or API key-based).
* **Application Deployment:** Submitting Streams Application Bundles (SABs) to the Streams instance.
* **Job Submission & Management:** Starting, stopping, and canceling jobs.
* **Status Monitoring:** Querying the status of instances, jobs, and Processing Elements (PEs).
* **Metrics Collection:** Retrieving performance and health metrics, if exposed via this API.

**Example: Translating curl to Python requests for Job Submission (Conceptual)**

A manual curl command for job submission might look like:

curl -X POST -H "Authorization: Bearer <AUTH\_TOKEN>" -H "Content-Type: application/octet-stream" --data-binary @my\_app.sab "https://<streams\_api\_host>/streams/rest/v2/instances/<instance\_id>/jobs"

This translates to Python requests as:

Python

import requests  
  
def submit\_job(api\_host, instance\_id, auth\_token, sab\_file\_path):  
 """  
 Submits a Streams job using the REST Management API.  
 """  
 url = f"https://{api\_host}/streams/rest/v2/instances/{instance\_id}/jobs" # Endpoint is assumed  
 headers = {  
 "Authorization": f"Bearer {auth\_token}",  
 "Content-Type": "application/octet-stream" # Or appropriate content type for SAB  
 }  
 try:  
 with open(sab\_file\_path, "rb") as f:  
 response = requests.post(url, headers=headers, data=f, verify=False) # verify=False for self-signed certs, use True in prod  
 response.raise\_for\_status() # Raises an HTTPError for bad responses (4XX or 5XX)  
 job\_details = response.json()  
 print(f"Job submitted successfully. Job ID: {job\_details.get('id')}")  
 return job\_details  
 except requests.exceptions.RequestException as e:  
 print(f"Error submitting job: {e}")  
 if hasattr(e, 'response') and e.response is not None:  
 print(f"Response content: {e.response.text}")  
 return None  
 except FileNotFoundError:  
 print(f"Error: SAB file not found at {sab\_file\_path}")  
 return None  
  
# Conceptual usage:  
# submit\_job("your-streams-api.teracloud.com", "myInstance", "your\_token", "path/to/my\_app.sab")

This conceptual example illustrates how the framework will encapsulate API interactions, handling headers, data payloads (including file uploads for SABs), and basic error checking. The actual endpoints and payload structures will depend on the specific Teracloud Streams REST Management API documentation.

### 4.2. Data Exchange REST API

The Data Exchange feature is used for injecting test data into running Streams applications and retrieving output data for validation.1 This requires endpoint operators to be included in the Streams application design.3 The framework will use the requests library to interact with these Data Exchange REST endpoints.

Key operations:

* **Data Injection:** Sending tuples (likely in JSON format) to specific input ports of a test application.
* **Data Retrieval:** Fetching tuples from specific output ports for validation against expected data or patterns.

**Example: Python requests for Data Injection (Conceptual)**

Python

import requests  
import json  
  
def inject\_data\_tuple(api\_host, job\_id, port\_name, auth\_token, tuple\_data):  
 """  
 Injects a data tuple into a Streams job via the Data Exchange API.  
 """  
 # Endpoint structure is assumed based on typical REST patterns for data exchange  
 url = f"https://{api\_host}/streams/rest/dataexchange/v1/jobs/{job\_id}/inputports/{port\_name}/tuples"  
 headers = {  
 "Authorization": f"Bearer {auth\_token}",  
 "Content-Type": "application/json"  
 }  
 try:  
 response = requests.post(url, headers=headers, data=json.dumps(tuple\_data), verify=False)  
 response.raise\_for\_status()  
 print(f"Data tuple injected successfully to port {port\_name}.")  
 return response.json() # Assuming some response, e.g., ack  
 except requests.exceptions.RequestException as e:  
 print(f"Error injecting data: {e}")  
 if hasattr(e, 'response') and e.response is not None:  
 print(f"Response content: {e.response.text}")  
 return None  
  
# Conceptual usage:  
# sample\_tuple = {"id": 123, "value": "test\_data", "timestamp": "2023-10-26T10:00:00Z"}  
# inject\_data\_tuple("your-streams-dx-api.teracloud.com", "job123", "inputDataPort", "your\_token", sample\_tuple)

The ability to programmatically inject and retrieve data is crucial for validating RPO, ensuring that data processed before a failover is not lost and that data processed after failover is consistent.

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

The framework will interact with the com.teracloud.streams.crossdcfailover toolkit 1 to test its efficacy. The exact nature of this interaction depends on the toolkit's design:

* **Configuration:** Test applications will be built using this toolkit. The framework will configure the toolkit through the Streams Application Bundle (SAB) submission parameters when deploying the application.
* **Monitoring:** The framework will monitor the toolkit's behavior during failover events through:
  + Metrics exposed by the toolkit operators
  + Application logs containing toolkit activity information
  + Status streams produced by the toolkit (e.g., RemoteDataCenterStatus)
  + Overall application status reported by the Streams Management API
* **Observing Automatic Failover:** The toolkit manages failover automatically. The framework will:
  + Induce failures in the environment (e.g., network partitions, instance shutdowns)
  + Observe the toolkit's detection and response to these failures
  + Measure and validate the toolkit's performance during failover events.

The framework is designed to work with the toolkit's automatic failover capabilities rather than attempting to programmatically trigger failover directly.

**5. Technology Choices and Deployment Strategy**

### 5.1. Justification for Python

Python has been selected as the primary programming language for the automation framework due to several compelling reasons:

* **Rich Ecosystem and Libraries:** Python offers a vast collection of mature libraries that are directly applicable to the tasks required by this framework. This includes:
  + HTTP communication: requests for interacting with REST APIs.15
  + SSH communication: paramiko for remote command execution, essential for agent control and certain failure injection methods.12
  + Network manipulation: tcconfig as a Python wrapper for the Linux tc command to simulate WAN conditions 9, and python-iptables (or direct iptables calls via paramiko) for network partitioning.11
  + Testing frameworks: pytest for defining, organizing, and executing tests, along with plugins like pytest-html and junit-xml for reporting.7
  + API interaction: prometheus-api-client for querying metrics from Prometheus.4
  + JMX interaction (optional): jmxquery for interfacing with Java Management Extensions if needed.6
  + Chaos Engineering: Libraries like chaostoolkit are Python-based, offering potential for future integration and more sophisticated chaos experiments.24
* **Ease of Development and Readability:** Python's clear and expressive syntax promotes rapid development, easier debugging, and more maintainable code, which is crucial for a complex testing framework.
* **Cross-Platform Compatibility:** Python runs on a wide variety of operating systems, making it suitable for the orchestrator component and any potential agents deployed on different systems.
* **Strong Community Support:** A large and active community ensures ample resources, tutorials, and quick resolution for common issues.

**Alternatives Considered:**

* **Java:** While robust and widely used in enterprise systems, Java can be more verbose for scripting and integrating with command-line utilities compared to Python. Its strength in JMX interaction is noted, but Python also has viable options.
* **Go (Golang):** Go is excellent for concurrent and networked applications and has a growing ecosystem. However, for the breadth of specific testing and system interaction libraries required (especially wrappers for Linux utilities), Python currently offers a more extensive and mature selection.
* **Shell Scripting (e.g., Bash):** Suitable for very simple automation tasks, but lacks the robustness, error handling, modularity, and complex logic capabilities required for a comprehensive testing framework. Maintaining and scaling a large test suite in shell scripts would be significantly more challenging.

The extensive availability of specialized libraries in Python significantly reduces development time and effort, allowing the team to focus on the logic of the test framework rather than building foundational communication and interaction tools from scratch.

### 5.2. Deployment Implications

The deployment of the Python-based automation framework requires consideration of the runtime environment and dependencies:

* **Python Runtime and Dependencies:**
  + The Test Orchestrator host and any machines running Test Agents will require a compatible Python version (e.g., Python 3.9+ as indicated by dependencies like tcconfig 9).
  + All necessary Python libraries (as listed in Table 5.1.A) must be installed. Dependency management will be handled using a requirements.txt file and virtual environments are strongly recommended to isolate framework dependencies.
* **Agent Deployment (If Used):**
  + For fine-grained failure injection at the network or process level (e.g., manipulating iptables on a specific host, killing a specific PE process), lightweight Python agents may need to be deployed on target VMs or container nodes within the Teracloud Streams environment.
  + These agents will be controlled by the Test Orchestrator via SSH using the paramiko library. Agents would listen for commands or execute pre-defined scripts triggered by the orchestrator.
* **Network Access Requirements:**
  + The Test Orchestrator machine must have HTTPS network access to the Teracloud Streams REST Management API and Data Exchange API endpoints in both data centers.
  + If Test Agents are used, the Orchestrator requires SSH access (typically port 22) to the machines hosting these agents.
  + The Orchestrator and/or dedicated monitoring components within the framework need network access to Prometheus API endpoints and/or JMX ports if these are used for metrics collection.
  + Appropriate firewall rules must be configured in all relevant network segments (Orchestrator network, Teracloud Streams DC networks, inter-DC links) to permit these communication paths.
* **Operating System Privileges:**
  + Operations like network manipulation (tc, iptables) executed by Test Agents or directly by the Orchestrator (if running on a gateway machine) will require sufficient privileges (e.g., root or sudo access). Secure management of these privileges is essential.
* **Resource Allocation:** The Test Orchestrator machine should have adequate CPU, memory, and network bandwidth to manage multiple test executions and handle API communications and data processing.

Careful planning of the deployment topology, including the placement of the orchestrator and any agents, along with ensuring all network connectivity and permissions are in place, will be critical for the successful operation of the testing framework.

### Table 5.1.A: Core Python Libraries and Their Roles in the Framework

|  |  |  |
| --- | --- | --- |
| **Library** | **Purpose in Framework** | **Key Features Utilized** |
| requests | HTTP/S communication with Teracloud Streams REST Management API and Data Exchange API. | Session management, JSON/form data handling, file uploads (for SABs), error handling.15 |
| paramiko | SSH communication for remote command execution on Test Agents or directly on Streams nodes (e.g., process interruption, log retrieval, triggering agent scripts). | SSH client functionality, command execution, SFTP (if needed for log transfer).12 |
| tcconfig | WAN emulation: simulating network latency, jitter, packet loss, and bandwidth constraints between DCs or components. | Python wrapper for Linux tc command (netem and htb modules).9 |
| python-iptables (or direct iptables calls via paramiko) | Network partitioning: creating rules to block or reroute traffic between specific hosts or networks. | Programmatic manipulation of iptables rules, chains, and tables.11 |
| pytest | Test case definition, organization, execution, and fixture management. | Test discovery, assertion mechanisms, fixtures for setup/teardown. |
| pytest-html / junit-xml (pytest plugins) | Generation of test execution reports. | Creating JUnit XML for CI/CD integration and HTML reports for human review.7 |
| prometheus-api-client | Querying metrics from Prometheus servers monitoring Teracloud Streams. | Execution of PromQL queries, parsing of time-series metric data.4 |
| jmxquery (Optional) | Querying JMX MBeans exposed by Teracloud Streams Java components, if applicable and necessary. | JMX connection establishment, MBean attribute querying.6 |
| PyYAML (or similar) | Parsing structured test configuration files. | Loading YAML/JSON data into Python objects. |
| chaostoolkit-lib (Potential Future Integration) | Foundation for defining and running more complex chaos engineering experiments. | Experiment definition (JSON/YAML), probes (state checks), actions (faults).24 |

This table provides a clear overview of the primary software dependencies, underscoring the suitability of Python by highlighting the readily available tools that map directly to the framework's functional requirements.

## 6. Design Specifications and Confirmed Operational Parameters

This section outlines firm design decisions and operational parameters that have been confirmed based on available Teracloud Streams documentation, particularly the 7.2.0 release information.1 Items not definitively ascertainable are addressed in Section 8 (Key Assumptions) and Section 9 (Points Requiring Confirmation).

* **DS-001: REST API Interaction Protocol:** The framework WILL utilize HTTP/S for all REST API communications with Teracloud Streams services. The Python requests library is the designated HTTP client for these interactions.
* **DS-002: Management API Functional Scope:** The Teracloud Streams REST Management API, enhanced in version 7.2.0, is confirmed to provide capabilities for managing domains, instances, and jobs.1 This API will be the primary interface for test setup (application deployment, job submission) and control (job cancellation, instance status checks).
* **DS-003: Data Exchange API Functional Scope:** The Data Exchange feature, new in version 7.2.0, provides REST APIs enabling direct injection of tuples into and retrieval of tuples from running Streams applications.1 This functionality requires the use of specific endpoint operators within the Streams application's SPL code.3 This API is critical for test data management and validation.
* **DS-004: Cross-DC Failover Toolkit Identification:** The official toolkit for enabling application-level failover across two data centers is named com.teracloud.streams.crossdcfailover.1 Test applications will be designed to utilize this toolkit.
* **DS-005: Authentication Method (High-Level):** API interactions with both the Management and Data Exchange services will require authentication. The framework will be designed to handle a standard mechanism such as bearer tokens or API keys, to be passed in HTTP headers. Specific details are pending confirmation (see SME-001, SME-002).
* **DS-006: Test Data Interchange Format (Primary):** The primary data format for tuple injection and retrieval via the Data Exchange API will be JSON, due to its widespread support in REST APIs and Python. Support for other formats like CSV will be considered if the API explicitly supports them.
* **DS-007: Standardized Reporting Formats:** Test execution results WILL be generated in JUnit XML format to facilitate integration with CI/CD pipelines and common test management tools.7 A custom, human-readable HTML report will also be generated for detailed analysis.
* **DS-008: Network Emulation Capabilities:** The framework WILL support the simulation of common WAN impairments, including network latency, jitter, packet loss, and bandwidth constraints. This will be achieved using the tcconfig library (wrapping Linux tc and netem 26) on relevant network nodes or gateways.
* **DS-009: Supported Failure Injection Mechanisms:** The framework WILL support the injection of failures through multiple methods:
  + **API-driven:** Stopping/starting jobs or instances via the REST Management API.
  + **Network manipulation:** Inducing partitions, latency, or loss using iptables and tcconfig.
  + **Remote process control:** Terminating specific processes on target nodes using paramiko.
* **DS-010: Python Version Requirement:** The framework will target Python 3.9 or newer, based on the requirements of key dependencies like tcconfig.9

These specifications provide a foundational layer for the detailed design and subsequent implementation. They transform previously identified areas of investigation into concrete design choices based on the most current available information.

### Table 6.A: Key Design Specifications & Operational Parameters

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Spec ID** | **Category** | **Specification/Parameter** | **Value/Constraint** | **Source/Rationale** |
| DS-001 | API Interaction | HTTP Client | Python requests library | Industry standard, versatile, meets requirements for REST communication.15 |
| DS-002 | API Scope | Management API Functionality | Manages domains, instances, jobs (application deploy, start, stop, status) | Confirmed by Teracloud Streams 7.2.0 Release Notes.1 |
| DS-003 | API Scope | Data Exchange API Functionality | Tuple injection into and retrieval from running applications | Confirmed by Teracloud Streams 7.2.0 Release Notes.1 |
| DS-004 | Toolkit | Cross-DC Failover Toolkit Name | com.teracloud.streams.crossdcfailover | Confirmed by Teracloud Streams 7.2.0 Release Notes.1 |
| DS-005 | API Interaction | Authentication (General) | Token-based or API Key in HTTP Headers | Standard practice for REST APIs; specifics pending SME confirmation. |
| DS-006 | Data Handling | Data Exchange Format | Primarily JSON; CSV if supported by API | JSON is ubiquitous for REST APIs and well-supported in Python. |
| DS-007 | Reporting | Primary Machine-Readable Report Format | JUnit XML | Standard for CI/CD integration and test management tools.7 |
| DS-007b | Reporting | Primary Human-Readable Report Format | HTML | Provides accessible and detailed test results for analysis. |
| DS-008 | Failure Simulation | Network Emulation Tools | tcconfig (wrapping Linux tc/netem) | Powerful and flexible for simulating various WAN conditions.9 |
| DS-009 | Failure Simulation | Failure Injection Methods | API calls, network manipulation (iptables, tcconfig), remote process termination (paramiko) | Provides a comprehensive range of failure scenarios. |
| DS-010 | Development | Python Version | Python 3.9+ | Driven by dependency requirements (e.g., tcconfig 9). |

This table serves as a quick reference for the core design tenets and operational assumptions derived from the available documentation, providing a stable foundation for the more granular aspects of the framework's design.

## 7. Advanced Testing Considerations

To ensure thorough validation of Teracloud Streams' cross-DC failover capabilities, the framework will incorporate advanced testing strategies beyond simple pass/fail checks. These include detailed failure simulation, emulation of complex network behaviors, and a refined test data strategy.

### 7.1. Detailed Failure Simulation Strategy

The framework will support a range of failure injection techniques to simulate realistic outage scenarios.

#### 7.1.1. API-Driven Failure Injection

The Teracloud Streams REST Management API will be leveraged to simulate controlled administrative failures. This includes:

* Stopping specific Streams jobs in the primary DC.
* Terminating individual Processing Elements (PEs), if the API provides such granularity.
* Bringing down an entire Streams instance in the primary DC. These actions test the system's response to deliberate operational interventions and the Cross-DC Failover Toolkit's ability to manage state and transition services accordingly.

#### 7.1.2. Cross-DC Orchestration of Failures and Recovery

The Test Orchestrator will manage complex, multi-stage failure scenarios. For example:

1. Simulate a complete network isolation of the primary DC.
2. Monitor and validate the failover of applications to the secondary DC within the defined RTO.
3. Verify data consistency against RPO targets.
4. Once stable on the secondary DC, restore connectivity to the primary DC (or simulate its recovery).
5. Optionally, test failback procedures by inducing a failure in the (now active) secondary DC or by administrative action, if supported and configured. The Orchestrator will precisely time and sequence these events, ensuring that each step of the failover and recovery process is stressed and validated.

#### 7.1.3. Agent-Based and Remote Failure Injection

For failures that cannot be directly triggered via the Streams Management API, the framework will use paramiko 12 to execute commands remotely, either directly on Streams nodes or via lightweight Python agents:

* **Network Failures:**
  + **Partitioning:** Modifying iptables rules on gateway routers or critical nodes to simulate network partitions between the two DCs, or between application components and essential services (e.g., external databases, Zookeeper if applicable). The python-iptables library 11 or direct iptables commands 28 will be used. The architecture of projects like aalekhpatel07/partition-sim 30, which uses a supervisor to control iptables via an HTTP API on Dockerized nodes, provides a conceptual model for how agents could be managed if direct paramiko control becomes overly complex.
  + **Complete Connectivity Loss:** Blocking all traffic on specific network interfaces.
* **Resource Exhaustion:** Agents can run scripts to simulate resource contention by consuming high amounts of CPU, memory, or disk I/O on critical nodes, testing how the Streams application and failover mechanisms handle degraded node performance.
* **Process/Service Killing:** Terminating specific Streams PE processes, instance manager processes, or critical dependent services (e.g., distributed cache, messaging system components if external to Streams and part of the application architecture) to simulate abrupt software failures.

These granular failure injection methods allow for testing the resilience of the system to a wider array of real-world problems.

### 7.2. Simulating Advanced Cross-DC Network Behaviors

Real-world WAN links between data centers rarely experience clean failures; more often, they suffer from various forms of degradation. The framework will simulate these conditions.

#### 7.2.1. WAN Latency, Jitter, and Packet Loss Simulation

Using tcconfig 9, a Python wrapper for the Linux tc utility (specifically the netem module 26), the framework will introduce:

* **Latency:** Configurable delays (e.g., 50ms, 100ms, 200ms) on the network path between DCs.
* **Jitter:** Variations in packet delay.
* **Packet Loss:** Percentage-based random packet loss (e.g., 0.1%, 1%, 5%). These simulations will be applied on network gateways or dedicated network emulator VMs through which inter-DC traffic flows, controlled remotely by the Orchestrator via paramiko.

#### 7.2.2. Partial Network Degradation and Asymmetric Conditions

Beyond simple latency and loss, tcconfig allows for more nuanced simulations:

* **Bandwidth Throttling:** Limiting the available bandwidth between DCs to simulate congestion.
* **Packet Reordering:** Intentionally reordering packets to test the application's and toolkit's tolerance to out-of-order delivery.
* **Packet Duplication/Corruption:** Introducing duplicated or corrupted packets.
* **Asymmetric Conditions:** Applying different degradation parameters for traffic flowing in each direction between the DCs (e.g., high latency from DC1 to DC2, but low latency from DC2 to DC1).
* **Intermittent Connectivity ("Flapping" Links):** Scripting sequences of network up/down states or rapid changes in network quality. These advanced simulations are crucial for testing the robustness of the Cross-DC Failover Toolkit's heartbeat mechanisms, state synchronization protocols, and overall stability under adverse but not entirely failed network conditions.

### 7.3. Refined Test Data Strategy

Effective validation of failover, particularly RPO, relies heavily on the nature of the test data.

#### 7.3.1. Generation of Deterministic and Identifiable Data Patterns

The framework will include Python scripts to generate input data streams with the following characteristics:

* **Unique Identifiers:** Each tuple or message will contain a unique, sequential ID (e.g., transaction ID, event ID) allowing for precise tracking of individual data items.
* **Timestamps:** Accurate timestamps will be embedded in data to measure end-to-end latency and to help determine the exact point of data loss or recovery for RPO validation.
* **Deterministic Patterns:** Data payloads will include predictable patterns (e.g., specific sequences of values, counters, known checksums) that can be easily verified at the output to detect corruption, loss, or duplication.
* **Varying Load Profiles:** The data generator will be capable of producing data at different rates and volumes to simulate various load conditions. While general synthetic data generation tools exist 31, the framework will prioritize simpler, custom Python scripts tailored to generate data specifically for validating failover characteristics.

#### 7.3.2. Schema Design for Stressing State Transfer and Consistency

The test Streams applications deployed by the framework will utilize tuple schemas designed to rigorously test state transfer and consistency aspects of the failover mechanism. These schemas will include:

* **Diverse Data Types:** A mix of common data types (integers, floats, strings, booleans) and, if supported by the Data Exchange API and relevant to the application, more complex types like lists, maps, or nested structures.
* **Variable Tuple Sizes:** Schemas allowing for both small and large tuple sizes to test the performance and reliability of state transfer under different data volume conditions. Payloads might include padding to simulate larger records.
* **State-Critical Attributes:** Attributes that are essential for stateful processing within the Streams application (e.g., counters, aggregation windows, session information) will be explicitly included and validated post-failover.
* **Version/Sequence Markers:** Attributes that can indicate the version or sequence of state updates, helping to verify that the correct state is restored after failover. Data conforming to these schemas will be injected using the Data Exchange API 1 and retrieved from output ports for validation. The goal is to ensure that all critical state information is correctly and completely replicated or recovered by the Cross-DC Failover Toolkit.

### Table 7.1.A: Illustrative Failure Simulation Scenarios

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Scenario ID** | **Description** | **Injection Method(s)** | **Target Component(s)** | **Expected Framework Behavior** | **Key Validation Checks** |
| FS-NET-001 | Full Network Partition between DC1 and DC2 | iptables rules via paramiko on inter-DC network gateways/routers. | All network traffic between DC1 and DC2. | Detect primary DC unavailability, orchestrate monitoring of failover to secondary DC, validate application on secondary. | RTO, RPO, application availability on secondary DC, data consistency. |
| FS-API-001 | Primary Streams Instance Shutdown | REST Management API call to stop the Streams instance in the primary DC. | Entire Streams instance in Primary DC. | Detect instance shutdown, monitor failover, validate application on secondary DC. | RTO, RPO, job status on secondary DC. |
| FS-PROC-001 | Critical PE Failure in Primary DC Application | paramiko to execute kill command on a specific PE process on a primary DC node. | Target PE of a stateful operator. | Detect PE failure, monitor toolkit's response (e.g., PE restart on secondary, or job failover), validate data integrity. | RTO for PE recovery/failover, data consistency, no data duplication. |
| FS-NET-002 | High Latency (e.g., 500ms) and 5% Packet Loss between DCs | tcconfig via paramiko on inter-DC network path emulator. | Network path between DC1 and DC2. | Monitor application stability, potential graceful failover if thresholds exceeded, validate data flow. | Application responsiveness, error rates, RPO if failover occurs, state synchronization lag (if measurable). |
| FS-DX-001 | Data Exchange Service Unavailability in Primary DC (Simulated) | iptables to block access to Data Exchange API endpoints for primary DC. | Data Exchange API endpoints for primary DC applications. | Test application's resilience to temporary data I/O issues, behavior of failover toolkit if it relies on DX. | Application error handling, impact on state if DX is used for state persistence, failover behavior. |

This table provides examples of how abstract failure conditions are translated into concrete, actionable test scenarios, linking the type of failure to the method of injection and the key validation points.

### Table 7.3.A: Example Test Data Schemas for State Transfer Validation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Schema Name** | **Description** | **Key Attributes (with types)** | **Identifiable Pattern Example** | **Purpose** |
| SimpleCounter | Basic stream for testing sequence and completeness. | sequence\_id (int64), payload (string), event\_timestamp (timestamp) | sequence\_id: Monotonically increasing integer. payload: "Event-" + sequence\_id. | Verify RPO (no lost sequence numbers), basic data integrity, order preservation. |
| StatefulAggregator | Simulates an application performing windowed aggregations. | key (string), value (float64), window\_id (int32), update\_timestamp (timestamp) | key: Rotates through "A", "B", "C". window\_id: Increments per key. | Test replication/recovery of aggregated state, consistency of windowed calculations across failover. Stress state transfer size. |
| ComplexRecord | Represents more complex data structures, potentially with larger payloads. | record\_id (string), customer\_data (map<string,string>), sensor\_readings (list<float64>), is\_critical (boolean) | record\_id: UUID. customer\_data: Predefined set of key-value pairs. sensor\_readings: Fixed-size list of random floats. | Validate transfer of complex data types (maps, lists), handling of larger individual records, and boolean flag consistency. |
| TransactionStream | Simulates financial or critical transactions requiring high fidelity. | tx\_id (string\_uuid), source\_account (string), target\_account (string), amount (decimal64), tx\_time (timestamp) | tx\_id: Unique UUID per transaction. | Ensure exactly-once semantics (or at-least-once with deduplication) for critical data, precise RPO for high-value transactions. |

These example schemas guide the design of test applications and data generation scripts, ensuring that the tests effectively probe the capabilities of the Cross-DC Failover Toolkit in maintaining various types of application state.

## 8. Key Assumptions for Implementation

Due to the current inaccessibility of detailed technical documentation for certain Teracloud Streams 7.2.0 components (specifically, the ReDoc/Swagger for the REST Management API 33, Data Exchange API specifics 34, and Cross-DC Failover Toolkit internals 36), the following assumptions are made for this design. These assumptions must be validated with Teracloud Subject Matter Experts (SMEs) before or during the implementation phase.

* **A-001: REST Management API Functionality:** It is assumed that the "enhanced REST API" mentioned in the 7.2.0 release notes 1 provides comprehensive and granular programmatic control over:
  + Application deployment (e.g., submitting SAB files).
  + Job lifecycle management (e.g., start, stop, cancel jobs by ID).
  + Instance and PE status monitoring (e.g., health checks, current state).
  + Retrieval of relevant metrics for instances, jobs, and PEs.
  + Domain management functions relevant to setting up test environments.
* **A-002: REST API Authentication Mechanism:** It is assumed that both the REST Management API and the Data Exchange API utilize a standard, well-documented authentication mechanism (e.g., OAuth 2.0 Bearer Tokens, API Keys passed in headers) that can be readily implemented within the Python requests library. The specific details of token acquisition and refresh (if applicable) are assumed to be straightforward.
* **A-003: Data Exchange API Mechanics and Data Formats:** It is assumed that the Data Exchange API allows:
  + Targeted injection of data tuples to specific, named input ports of a running Streams application, identifiable by job ID and port alias/ID.
  + Targeted retrieval of data tuples from specific, named output ports.
  + Primary support for JSON as a data interchange format for tuples.
  + Clear mechanisms for handling data schemas and potential error responses (e.g., backpressure, invalid data format).
* **A-004: Cross-DC Failover Toolkit Control and Observability:** It is assumed that the com.teracloud.streams.crossdcfailover toolkit can be:
  + Configured either at application deployment time (e.g., via parameters in the SPL composite or job submission configurations) or potentially influenced at runtime via API calls.
  + Its operational status, current role (primary/secondary), and health can be monitored either through specific metrics exposed by the toolkit itself (via Prometheus/JMX or API) or inferred from the overall application/job status via the Management API.
* **A-005: Cross-DC Failover Toolkit Intrinsic Behavior:** It is assumed that the toolkit autonomously manages critical aspects of failover, including:
  + State replication or access to shared state across DCs.
  + Heartbeating or health checking between DC instances/components.
  + Leader election or role transition logic to designate an active instance in one DC.
* **A-006: Availability of Key Failover Metrics:** It is assumed that critical metrics necessary to validate failover success and quantify RTO/RPO are exposed and accessible. These include, but are not limited to:
  + Failover event detection time.
  + Time to restore service in the secondary DC (RTO component).
  + Data replication lag or last consistent data point (RPO component).
  + Current active DC or instance serving requests.
  + Error rates and throughput pre- and post-failover. These metrics are expected to be available via the REST Management API, Prometheus, or JMX.
* **A-007: Environment Access and Permissions:** It is assumed that the testing framework's orchestrator and any deployed agents will be granted the necessary network access (firewall rules, routing) and credentials/permissions to:
  + Communicate with all required Teracloud Streams API endpoints in both DCs.
  + Perform SSH operations on agent hosts (if applicable).
  + Execute privileged commands (e.g., tc, iptables) where necessary for failure injection.
* **A-008: Idempotency of Critical API Operations:** It is assumed that critical API operations (e.g., start job, stop job, deploy application) are either idempotent by design or provide mechanisms (e.g., transactional semantics, query-before-write) that allow the framework to safely retry operations in case of transient network issues without causing unintended side effects.
* **A-009: Consistency of API Behavior:** It is assumed that the behavior of the REST APIs is consistent and conforms to their (forthcoming) documentation, allowing for reliable automation.

These assumptions form the basis of the current design. Their validation is paramount for the successful implementation of the testing framework.

## 9. Points Requiring Confirmation from Teracloud SMEs

To address the information gaps resulting from currently inaccessible detailed documentation and to ensure the design aligns with Teracloud Streams 7.2.0 capabilities, the following points require explicit confirmation and clarification from Teracloud Subject Matter Experts (SMEs):

* **SME-001: REST Management API Documentation:**
  + Request for comprehensive documentation for the Teracloud Streams 7.2.0 REST Management API. This should ideally be an OpenAPI (Swagger) specification or ReDoc-generated documentation.
  + Details needed: All available endpoints, HTTP methods, request/response schemas (including data types and structures), error codes, and supported authentication mechanisms (e.g., token URL, grant types, required headers like X-API-Key if applicable 38).
  + Specific endpoints of interest: application deployment (e.g., equivalent to POST /jobs/create 39), job submission, job status query (e.g., equivalent to GET /jobs/get 39 or job status resources in 40), instance management 2, and metrics/log access.43
* **SME-002: Data Exchange Service API Documentation:**
  + Request for detailed documentation for the Data Exchange Service REST API.
  + Details needed: Specific endpoints for data injection (e.g., similar to concepts in 45) and retrieval (e.g., similar to concepts in 47), supported data formats (JSON, CSV, etc.), schema definition and validation, methods for identifying target jobs and stream ports, authentication details, and error handling.
  + Clarification on how endpoint operators within SPL applications expose these REST interfaces.3
* **SME-003: Cross-DC Failover Toolkit (com.teracloud.streams.crossdcfailover) Documentation:**
  + Request for in-depth technical documentation for the com.teracloud.streams.crossdcfailover toolkit.
  + Details needed: Architecture, configuration parameters (e.g., for heartbeat, replication, quorum), operational workflow during steady-state and failover, state replication mechanisms (synchronous/asynchronous, consistency model), any exposed APIs or JMX MBeans for monitoring or control, expected behavior under various failure types, and integration points with the broader Streams environment.
  + Guidance on configuring the toolkit for different application needs and failover policies.51
* **SME-004: API Rate Limits and Quotas:**
  + Confirmation of any rate limits, quotas, or throttling mechanisms imposed on the REST Management API and the Data Exchange API to ensure the testing framework operates within acceptable bounds.
* **SME-005: Failover Status Monitoring and Observation:**
  + What are the recommended methods to observe the current status/role (active/standby) of the Cross-DC Failover Toolkit or the application instances it manages? (e.g., specific metrics, log patterns, or status streams)
  + What are the expected behaviors and detection mechanisms of the toolkit when encountering different types of failures?
  + What is the recommended approach for testing the toolkit's automatic failover capabilities?
* **SME-006: Toolkit's Failure Detection Capabilities:**
  + What specific types of failures is the Cross-DC Failover Toolkit designed to automatically detect and react to (e.g., PE crash, host unavailability, network partition between PEs within a DC, complete loss of inter-DC connectivity, full DC outage)?
  + What are the typical detection times for these failures?
* **SME-007: State Replication Details and Guarantees:**
  + A more detailed explanation of how state is managed and replicated by the Cross-DC Failover Toolkit.
  + What are the consistency guarantees for application state upon failover (e.g., eventual consistency, strong consistency)?
  + How does the toolkit handle in-flight data or transactions during a failover event?
* **SME-008: Recommended Monitoring Metrics for Failover:**
  + What are the key metrics (exposed via REST API, Prometheus, or JMX) that Teracloud recommends monitoring to:
    - Assess the health and readiness of a cross-DC Streams deployment?
    - Track the progress and success of a failover event?
    - Measure RTO and RPO accurately?
* **SME-009: Security Context and Permissions for API Usage:**
  + What security principals, roles, or permissions are required for the API user/service account that the testing framework will use to perform actions such as deploying applications, managing jobs, accessing data via Data Exchange, and querying monitoring information?
* **SME-010: streamsx.rest Applicability for Management API:**
  + Confirmation that for the new Teracloud Streams 7.2.0 "enhanced REST API" for domain, instance, and job management, a standard Python HTTP client (like requests) is the appropriate interaction method, and that streamsx.rest (historically used with IBM Streams for application interactions 17) is not intended or suitable for this new management plane API.

Obtaining clear answers to these questions is crucial for refining the design, mitigating implementation risks, and ensuring the testing framework accurately and effectively validates the capabilities of Teracloud Streams 7.2.0.

## 10. Conclusion

This Design Plan Document details a comprehensive Python-based automated testing framework for Teracloud Streams cross-DC failover. By leveraging the newly announced Teracloud Streams 7.2.0 features, including the enhanced REST Management API, Data Exchange service, and the com.teracloud.streams.crossdcfailover toolkit 1, the framework aims to provide robust and repeatable validation of application resilience.

The architecture emphasizes modularity, allowing for clear separation of concerns between test orchestration, API interaction, failure injection, monitoring, and reporting. The choice of Python is justified by its extensive ecosystem of libraries well-suited for automation, API interaction, network control, and testing.7 The design incorporates advanced testing considerations, such as detailed failure simulation across multiple layers (API, network, process), emulation of complex cross-DC network behaviors, and a refined test data strategy focused on deterministic and identifiable patterns to accurately measure RPO and RTO.

A significant aspect of this design is the explicit identification of assumptions and points requiring confirmation from Teracloud SMEs. The current inaccessibility of detailed technical documentation for the new APIs and toolkit 33 necessitates these clarifications to ensure the framework's implementation aligns precisely with Teracloud Streams' capabilities.

Upon confirmation of the queried details, this design provides a solid foundation for the development of an effective automated testing solution that will significantly enhance confidence in the disaster recovery and business continuity posture of applications running on Teracloud Streams. The framework is designed to be adaptable, allowing for the incorporation of new test scenarios and evolving platform features in the future.

#### Works cited

1. Teracloud Streams Version 7.2.0 Release Notes, accessed May 14, 2025, <https://www.streams.teracloud.com/releases/7.2.0>
2. Teracloud Streams Version 7.2.0 Release Notes, accessed May 14, 2025, <https://streams.teracloud.com/releases/7.2.0>
3. Developing, accessed May 14, 2025, <https://doc.streams.teracloud.com/com.ibm.streams.dev.doc/doc/dev-container.html>
4. prometheus-api-client - PyPI, accessed May 14, 2025, <https://pypi.org/project/prometheus-api-client/>
5. Prometheus API: From Basics to Advanced Usage - Last9, accessed May 14, 2025, <https://last9.io/blog/prometheus-api/>
6. jmxquery · PyPI, accessed May 14, 2025, <https://pypi.org/project/jmxquery/>
7. 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>
8. Common JUnit XML Format & Examples - GitHub, accessed May 14, 2025, <https://github.com/testmoapp/junitxml>
9. tcconfig - PyPI, accessed May 14, 2025, <https://pypi.org/project/tcconfig/>
10. 1. - tcconfig's documentation!, accessed May 14, 2025, <https://tcconfig.readthedocs.io/en/stable/pages/introduction/index.html>
11. ldx/python-iptables - GitHub, accessed May 14, 2025, <https://github.com/ldx/python-iptables>
12. Client - Paramiko documentation, accessed May 14, 2025, <https://docs.paramiko.org/en/stable/api/client.html>
13. 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>
14. 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>
15. Python requests best practices for data engineers - Y42, accessed May 14, 2025, <https://www.y42.com/blog/python-requests-best-practices-for-data-engineers>
16. Requests Library: Python's Power for HTTP and Real Scenarios - Data Science Dojo, accessed May 14, 2025, <https://datasciencedojo.com/blog/requests-library/>
17. The Python REST API - IBM Streams, accessed May 14, 2025, <http://ibmstreams.github.io/streamsx.documentation/docs/python/1.6/python-appapi-devguide-6/>
18. Developing streams application with views, accessed May 14, 2025, <https://doc.streams.teracloud.com/com.ibm.streams.dev.doc/doc/views_developing.html>
19. Setting up Paramiko for SSH Communication - Python for Network Automation, accessed May 14, 2025, <https://python-automation-book.readthedocs.io/en/1.0/11_paramiko/01_intro.html>
20. Python, Paramiko and SSH Basic Overview - YouTube, accessed May 14, 2025, <https://www.youtube.com/watch?v=-wu4-5lTdsg>
21. tcconfig - PyPI, accessed May 14, 2025, <https://pypi.org/project/tcconfig/0.6.0/>
22. python-iptables - PyPI, accessed May 14, 2025, <https://pypi.org/project/python-iptables/0.1.1/>
23. Introduction — python-iptables 0.4.0-dev documentation, accessed May 14, 2025, <https://python-iptables.readthedocs.io/en/latest/intro.html>
24. chaostoolkit · PyPI, accessed May 14, 2025, <https://pypi.org/project/chaostoolkit/0.3.0/>
25. Chaos Engineering Toolkit & Orchestration for Developers - GitHub, accessed May 14, 2025, <https://github.com/chaostoolkit/chaostoolkit>
26. Basic WAN emulation - ALLOUTPUT.com, accessed May 14, 2025, <http://alloutput.com/technology/basic-wan-emulation/>
27. Linux Traffic Control WAN Emulation - YouTube, accessed May 14, 2025, <https://www.youtube.com/watch?v=iZHCdTLD4ck>
28. 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>
29. iptables - ArchWiki, accessed May 14, 2025, <https://wiki.archlinux.org/title/Iptables>
30. aalekhpatel07/partition-sim: Simulate a network partition between a distributed cluster of nodes. - GitHub, accessed May 14, 2025, <https://github.com/aalekhpatel07/partition-sim>
31. Introducing the Synthetic Data Generator - Build Datasets with Natural Language, accessed May 14, 2025, <https://huggingface.co/blog/synthetic-data-generator>
32. About Streaming Data - kdb products, accessed May 14, 2025, <https://code.kx.com/insights/1.12/microservices/stream-processor/index.html>
33. accessed December 31, 1969, <https://doc.streams.teracloud.com/external-reference/redoc-static.html>
34. 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)
35. accessed December 31, 1969, <https://doc.streams.teracloud.com/com.ibm.streams.dev.doc/doc/enabling-streams-data-exchange.html>
36. 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)
37. 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>
38. REST API Endpoints - Terra Docs, accessed May 14, 2025, <https://docs.tryterra.co/reference/streaming-api/api-endpoints>
39. Jobs API 2.0 | Databricks Documentation, accessed May 14, 2025, <https://docs.databricks.com/aws/en/reference/jobs-2.0-api>
40. Jobs - REST API Reference - Informatica Documentation, accessed May 14, 2025, <https://docs.informatica.com/integration-cloud/b2b-gateway/current-version/rest-api-reference/platform-rest-api-version-2-resources/jobs.html>
41. Job status - REST API Reference - Informatica Documentation, accessed May 14, 2025, <https://docs.informatica.com/integration-cloud/b2b-gateway/current-version/rest-api-reference/runajob-utility/runajob-utility-setup/job-status.html>
42. Teracloud Streams - Real-time Analytics, accessed May 14, 2025, <https://streams.teracloud.com/>
43. Monitoring your API | Cloud Endpoints with OpenAPI - Google Cloud, accessed May 14, 2025, <https://cloud.google.com/endpoints/docs/openapi/monitoring-your-api>
44. Azure Monitor overview - Learn Microsoft, accessed May 14, 2025, <https://learn.microsoft.com/en-us/azure/azure-monitor/fundamentals/overview>
45. Showcase | Teracloud Streams, accessed May 14, 2025, <https://streams.teracloud.com/showcase>
46. Using Ingestion API to Load Data Into Data Cloud | Data Cloud Decoded - YouTube, accessed May 14, 2025, <https://www.youtube.com/watch?v=3xWSVGcTORI>
47. REST API Retrieve Data from a Data Extension in Salesforce Marketing Cloud - YouTube, accessed May 14, 2025, <https://www.youtube.com/watch?v=ih_Bwp_nyac>
48. Requesting Data (REST API) - Terra Docs, accessed May 14, 2025, <https://docs.tryterra.co/docs/historical-data>
49. Axway Endpoint Solutions: Simplified Data Exchange, accessed May 14, 2025, <https://www.axway.com/en/products/axway-endpoint-solutions>
50. AWS Data Exchange quotas - AWS Documentation, accessed May 14, 2025, <https://docs.aws.amazon.com/data-exchange/latest/userguide/limits.html>
51. Configuring Teracloud Streams, accessed May 14, 2025, <https://doc.streams.teracloud.com/com.ibm.streams.cfg.doc/doc/ibminfospherestreams-containercfg.html>
52. Download 7.1.0 Installation Guide - Teracloud Streams, accessed May 14, 2025, <https://streams.teracloud.com/downloads/installation-guide>
53. pro.plus.txt - jsDelivr, accessed May 14, 2025, <https://cdn.jsdelivr.net/gh/hagezi/dns-blocklists@latest/rpz/pro.plus.txt>
54. Full text of "Processor Newspaper Volume 28 Number 15" - Internet Archive, accessed May 14, 2025, <https://archive.org/stream/processor-newspaper-v28i15/P___2815_djvu.txt>
55. High Availability | Teradata Data Mover - Automatic Failover, accessed May 14, 2025, <https://docs.teradata.com/r/Enterprise_IntelliFlex_Lake_VMware/Teradata-Data-Mover-User-Guide-20.03/High-Availability/Automatic-Failover>