**GitHub Migration Testing Environment Set in SKY-UK**

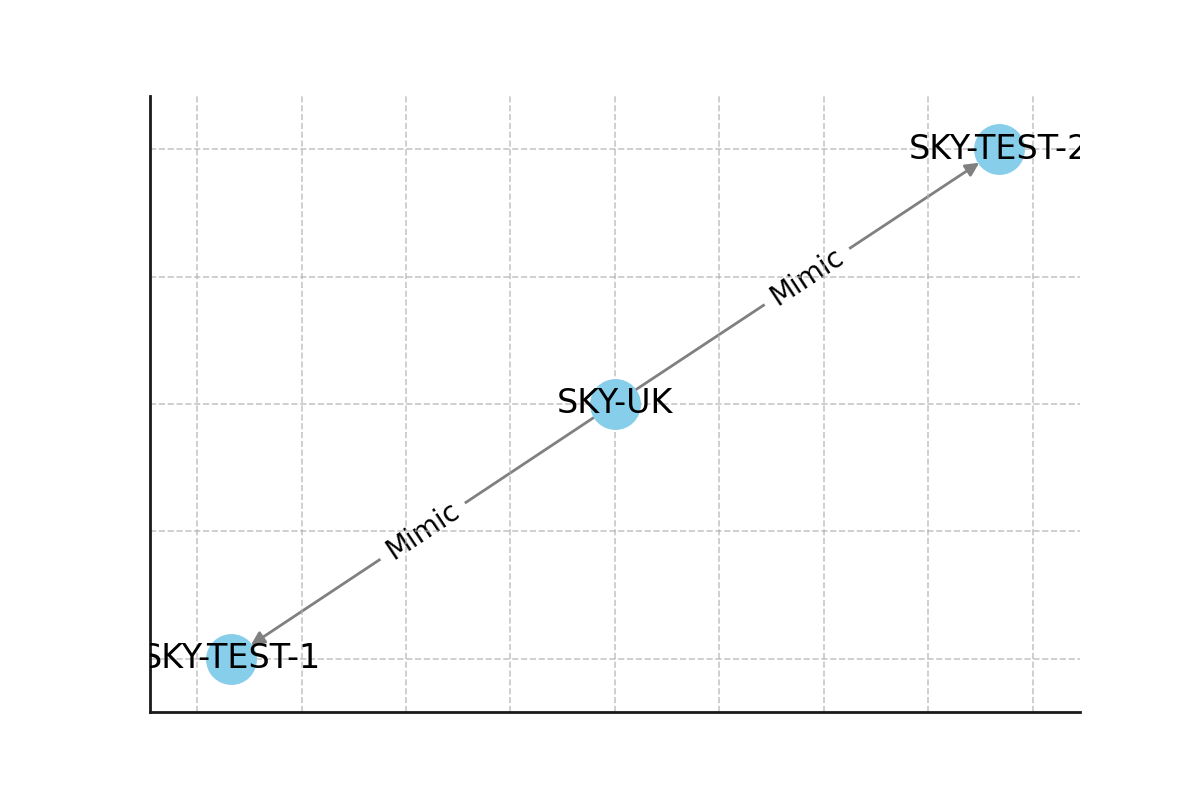
This document amalgamates a detailed plan and comprehensive overview for testing the migration from SKY-UK's GitHub Enterprise Cloud (GEC) to a new target environment (organization) which is based on the same Plan (GEC). It includes a step-by-step guide, visual diagrams, potential challenges, and strategies for error handling, rollback, and preserving forking history, ensuring a well-rounded and effective migration process.

# Setting Up Testing Environment

The testing environment will simulate the current GitHub setup within SKY-UK and involves creating two mock organizations, SKY-TEST-1 and SKY-TEST-2. These organizations will mimic real-world scenarios with users, teams, and projects.

1. Permissions: Assign 'Owner' permissions to the project lead in both organizations to ensure complete control over migration activities and settings.
2. Creating Mock Data: Generate fake user accounts, repositories, teams, and project data using scripts. This data should closely resemble the actual data in SKY-UK for a realistic testing scenario.
3. Simulating Activities: Implement scripts to create dynamic repository activities like commits, pull requests, and issue interactions, reflecting typical user behavior.

Figure 1: Schematic Diagram of Testing Environment Setup



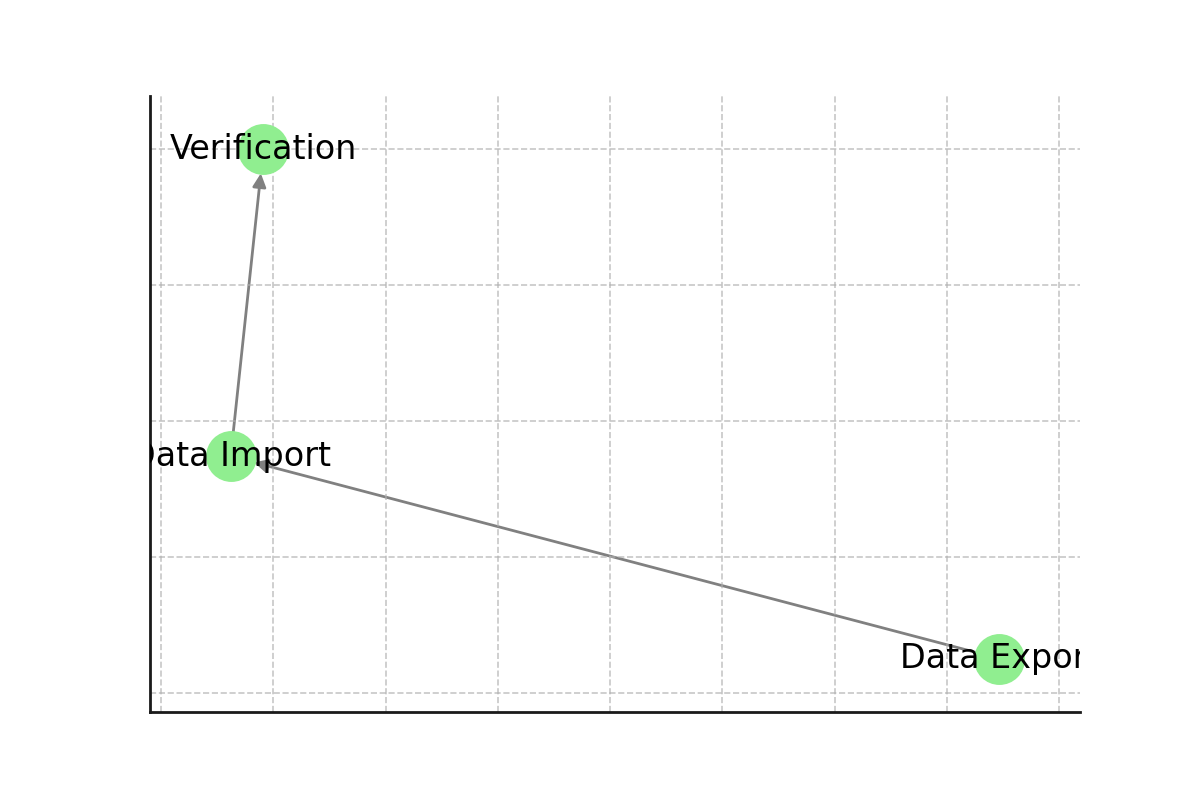
# Detailed Migration Steps and Verification

This section outlines specific steps for migrating data from SKY-TEST-1 to SKY-TEST-2, serving as a representative of the actual migration, including post-migration verification.

1. Data Export: Export data from SKY-TEST-1 using GitHub’s API, including all repositories, user accounts, and project data.
2. Data Import: Import the exported data into SKY-TEST-2, ensuring accurate mapping of users, repositories, and teams to the new environment.
3. Verification Process: Conduct thorough verification to ensure all data, including commit history and team permissions, is accurately and completely transferred.

Caveat: Be mindful of potential data inconsistencies, especially in user permissions and repository settings, which might require manual adjustments post-migration.

Figure 2: Flowchart of the Migration Process



# Error Handling and Rollback Strategies

Incorporate robust error-handling mechanisms and establish rollback strategies to mitigate any issues encountered during the migration process.

1. Script Error Handling: Implement comprehensive error handling in migration scripts to manage potential API failures or data inconsistencies.
2. GitHub Actions for Rollback: Utilize GitHub Actions to automate rollback procedures in case of critical failures or significant discrepancies during migration.
3. Continuous Monitoring: Establish a continuous monitoring system to track the migration process, enabling quick identification and resolution of issues.

Caveat: Rollback actions must be tested thoroughly to ensure they can revert changes without causing data loss or corruption.

# Handling Forking History

Since standard migration processes do not retain forking history, alternative strategies are required.

1. Documenting Forking Relationships: Record the forking history of important repositories before the migration to maintain a reference.
2. Post-Migration Fork Recreations: Manually recreate the forks in the new environment based on the documented history to preserve repository relationships.
3. Embedding Forking Information: Embed forking history within repositories or create a dedicated reference repository in the new environment for historical context.

Caveat: Manual processes for handling forking history can be time-consuming and may require additional documentation and training for users.

# Utilizing GitHub Cloud Logs for Additional Metadata

GitHub Cloud Logs provide a rich source of additional metadata, which is crucial for a comprehensive migration process.

1. Extracting User Activities: Use API calls to extract detailed logs of user activities, including commits, pull requests, and issue interactions.
2. Repository Interaction Logs: Retrieve logs related to repository interactions for a thorough understanding of user and repository activities.
3. Analyzing and Comparing Logs: Analyze the extracted logs for completeness and compare them against the target environment to ensure a successful migration.

Caveat: Extracting and analyzing logs can be complex and time consuming, often leading to an over specified output.

# Organization Migration via GitHub API

Using the GitHub API for organization migration is a critical process for transferring repositories, teams, and user data between GitHub Enterprise Cloud environments. The 'Start an organization migration' REST API endpoint is used for this purpose.

The process involves a POST request to '/orgs/{org}/migrations' with required parameters like organization name and repositories list. Repositories can be locked during migration to ensure data consistency.

Example JavaScript Code for API Call:

// Octokit.js  
// https://github.com/octokit/core.js#readme  
const octokit = new Octokit({ auth: 'YOUR-TOKEN' })  
  
await octokit.request('POST /orgs/{org}/migrations', {  
 org: 'ORG',  
 repositories: ['github/Hello-World'],  
 lock\_repositories: true,  
 headers: {'X-GitHub-Api-Version': '2022-11-28'}  
})

# Special Considerations for Migration

During the migration process, special attention needs to be paid to locked repositories, tagging, and handling of repositories marked for archiving.

1. Locked Repositories: Locking repositories during migration prevents changes and maintains data integrity. It's essential to verify post-migration that the repositories are unlocked and accessible.
2. Tagging: Tags, typically used for marking releases, should be preserved during migration. Verification steps should include checking the transfer of all repository tags.
3. Archiving: For repositories marked for archiving, consider whether it's necessary to migrate them 'as is' or use alternative storage solutions like AWS S3. This decision depends on the repository's relevance and size.

# User Migration via API

User migration is a critical component of transitioning to a new GitHub environment. The GitHub API provides endpoints for migrating user data, as outlined in the 'Start a user migration' section.

This process involves creating a POST request to the '/user/migrations' endpoint. The request should include specific parameters such as the repositories and gists to be migrated. It's essential to handle user data with care, ensuring compliance with data protection regulations.

Example API Call for User Migration:

// Example using Octokit.js  
const octokit = new Octokit({ auth: 'YOUR-TOKEN' })  
  
await octokit.request('POST /user/migrations', {  
 repositories: ['username/repo'],  
 lock\_repositories: true,  
 exclude\_attachments: false  
})

# Source Imports (Applications)

Source imports involve transferring external source code, like open-source projects or third-party applications, into GitHub repositories. The GitHub API’s 'source imports' feature facilitates this process.

This feature supports importing from various version control systems and requires setting up a POST request to the '/repos/{owner}/{repo}/import' endpoint. The request includes details about the source, such as the URL, version control system type, and additional configuration settings.

Example API Call for Source Import:

// Example using Octokit.js  
const octokit = new Octokit({ auth: 'YOUR-TOKEN' })  
  
await octokit.request('POST /repos/{owner}/{repo}/import', {  
 vcs\_url: 'url-of-the-source',  
 vcs: 'subversion', // or 'git', 'mercurial', etc.  
 vcs\_username: 'username',  
 vcs\_password: 'password'  
})

# The choice of language programming to interact with GitHub’s API

As most of the examples given by the official GitHub’s API reference call book come either built on JavaScript or cURL; but otherwise, Python is also well designed to handle API calls because of the integrations and versatility. Without pretending to be exhaustive, the choice of one way or another depends on a number of factors and the context in which migration is performed:

1. **Client-Side Operations:**
   * JavaScript is traditionally used for client-side operations in web applications. If your GitHub API interactions need to be integrated into a web interface or a browser-based application, JavaScript is a natural choice.
2. **Node.js for Server-Side:**
   * With the advent of Node.js, JavaScript can also be used for server-side scripting. This can be particularly useful if your existing infrastructure or project already heavily relies on JavaScript or Node.js.
3. **Octokit SDK:**
   * GitHub provides an official client library for JavaScript called Octokit. It simplifies working with the GitHub API by providing convenient methods and abstractions. If your team is already familiar with JavaScript, using Octokit can streamline the development process.
4. **Real-Time Interaction:**
   * For applications that require real-time updates from GitHub (like a CI/CD pipeline or a dashboard reflecting current repository status), JavaScript, particularly with Node.js, can be a practical choice due to its event-driven, non-blocking nature.
5. **Full-Stack Development:**
   * In a full-stack JavaScript environment (using frameworks like React, Angular, or Vue.js on the front end, and Node.js on the back end), it’s more efficient to use JavaScript for all layers, including API interactions.

However, this doesn't mean Python cannot be used. Python is equally capable of handling API requests and is often preferred for:

1. **Backend Development:**
   * Python is widely used in backend development and scripting, thanks to its simplicity and the vast array of libraries available for various tasks, including API interactions.
2. **Data Manipulation and Scripting:**
   * Python excels in data manipulation, making it a great choice for handling and processing data retrieved from API calls.
3. **Wide Range of Applications:**
   * Python’s versatility makes it suitable for a wide range of applications, from web development (using frameworks like Django or Flask) to scientific computing and machine learning.

In conclusion, the choice between JavaScript and Python for interacting with the GitHub API should be based on the specific needs of your project, the existing technology stack, and the expertise of your team. Both languages are capable, but each has its strengths in different scenarios.

# Personal Access Token (PAT) Migrations

Migrating access tokens as part of a GitHub migration requires careful consideration due to security implications and GitHub's token management policies. Here's an overview of how to address token migration:

**Understanding Access Tokens**

1. **Purpose**: Access tokens are used to authenticate against GitHub for various operations, often in automated scripts or CI/CD pipelines.
2. **Security**: Tokens are sensitive data and should be handled with the highest security standards.

**Key Points in Token Migration**

1. **Token Migration is Generally Manual**:
   * GitHub does not provide a direct method to migrate tokens via their API or migration tools due to security reasons.
     + Each token should be considered individually to determine if it needs to be migrated, regenerated, or decommissioned.
2. **Regenerate Tokens in the New Environment**:
   * The recommended approach is to create new tokens in the target environment.
     + This ensures tokens are fresh, secure, and have the correct permissions.
   * Update any systems or scripts that use these tokens with the new ones.
3. **Audit and Revoke Old Tokens**:
   * Review the usage of existing tokens in the current environment.
   * Revoke tokens that are no longer needed or new ones in the target environment will replace that.
     + This step is crucial to prevent unauthorized access and maintain security.
4. **Update CI/CD Pipelines and Scripts**:
   * Ensure that all automation scripts, CI/CD pipelines, and applications that use these tokens are updated to use the new tokens.
     + Test these integrations to confirm that they work correctly with the new tokens.
5. **Documentation and Communication**:
   * Document the process of token regeneration and distribution.
     + Communicate with the team members or system administrators involved, ensuring they are aware of the token changes and updates.
6. **Security Best Practices**:
   * Treat tokens as sensitive data. Always use secure methods to distribute and store them.
     + Regularly audit and rotate tokens even post-migration to enhance security.

**In conclusion:**

Token migration is primarily a manual process focusing on generating new tokens in the target environment and ensuring that all systems using these tokens are updated accordingly. Due to the sensitive nature of tokens, this process must be handled with attention to detail and adherence to security best practices.

# GitHub Enterprise Cloud Migration: API Endpoints

Just a brief resumè of which REST API’s Endpoints are used to monitor, set up, and track the migration between GEC instances at different levels.

|  |  |
| --- | --- |
| Category | API Endpoints / Actions |
| Organization Migrations | 1. Start Organization Migration (`POST /orgs/{org}/migrations`) 2. Get Status of Migration (`GET /orgs/{org}/migrations/{migration\_id}`) 3. Download an Organization Migration Archive (`GET /orgs/{org}/migrations/{migration\_id}/archive`) 4. Unlock an Organization Repository (`DELETE /orgs/{org}/migrations/{migration\_id}/repos/{repo\_name}/lock`) 5. List Repositories in an Organization Migration (`GET /orgs/{org}/migrations/{migration\_id}/repositories`) |
| Source Imports (Apps, Repos) | 1. Start a Source Import (`PUT /repos/{owner}/{repo}/import`) 2. Get Import Progress (`GET /repos/{owner}/{repo}/import`) 3. Update Import (`PATCH /repos/{owner}/{repo}/import`) 4. Cancel Import (`DELETE /repos/{owner}/{repo}/import`) |
| User Migrations | 1. Start a User Migration (`POST /user/migrations`) 2. Get the Status of a User Migration (`GET /user/migrations/{migration\_id}`) 3. Download a User Migration Archive (`GET /user/migrations/{migration\_id}/archive`) 4. Unlock a User Repository (`DELETE /user/migrations/{migration\_id}/repos/{repo\_name}/lock`) 5. List Repositories for a User Migration (`GET /user/migrations/{migration\_id}/repositories`) |

# Using GitHub Actions for Migration Testing and Rollback process

GitHub Actions can be an invaluable tool for automating aspects of the migration testing process. By setting up specific workflows, you can automate tasks like code checks, data validation, and notifying stakeholders of the migration's progress or issues.

Here's an example of a GitHub Actions workflow that could be used, I drafted a mop-up, but the structure think is the right one:

name: Migration Test Workflow

on:

push:

branches:

- main

jobs:

migration-test:

runs-on: ubuntu-latest

steps:

- name: Checkout repository

uses: actions/checkout@v2

- name: Run migration tests

run: python migration\_test\_script.py

- name: Notify stakeholders

if: failure()

run: python notify\_admins.py

* The migration test is triggered on a push to the main branch.
* It checks out the repository, runs migration tests, and notifies stakeholders if a failure occurs.

**Rollback Strategy**

In case of migration issues, a rollback strategy is essential. You can implement a workflow that reverts changes if specific conditions are met. For example, if migration tests fail, the workflow could trigger scripts to rollback changes or restore data from a backup.

**Communication of Migration Events**

To communicate migration events, especially to admin users or regular users, you can utilize GitHub's webhook feature. Webhooks can be configured to send real-time notifications to a specified endpoint (like a server or an application) whenever certain events occur in a repository. webhook to notify a team's communication platform (such as Slack or Microsoft Teams) whenever there's a push to a specific branch or when a migration-related issue is opened or closed.