

Migrate from Server to Cloud

Required:

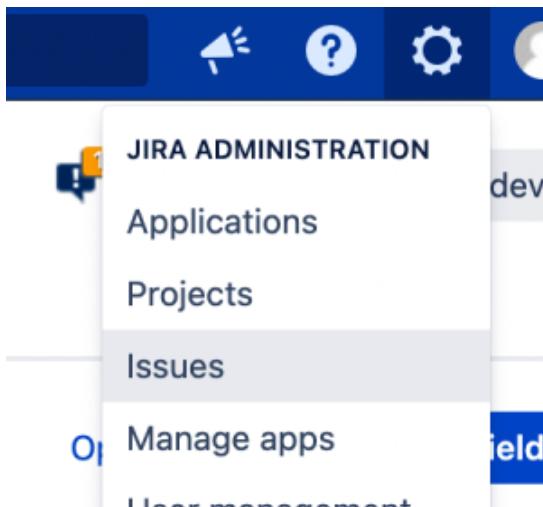
- MLCS Server 6.1.0 plugin or newer installed on the Jira Server / Datacenter instance
- Jira server / datacenter 8.20.0 or newer
- Jira Cloud instance

Migrate Tree options

You can download json file of tree options from MLCS server / data center and import it on MLCS Cloud.

Export json tree from MLCS Server / DC

Go to **Settings Issues**



Go to **Custom fields** on left side and edit MLCS Custom field type, click on 3 dots ... **configure**

A screenshot of the Jira Administration 'Custom fields' page. The left sidebar shows categories like Issue types, Screens, and Fields. Under 'FIELDS', 'Custom fields' is selected and highlighted with a red box. A red arrow points from this box to the 'MLCS' custom field in the main list. The 'MLCS' field is also highlighted with a red box. A context menu is open over the 'MLCS' row, with options 'Configure', 'Edit', 'Translate', and 'Screens'. The main table lists various custom fields with columns for Name, Type, Available contexts, Screens, Last value update, and Actions. The 'MLCS' field is listed as 'Multi-Level Cascading Select' with 3 projects, 17 screens, and a date of 27/Aug/21.

Go to **edit options** of your custom field type

Default Configuration Scheme for MLCS

Default configuration scheme generated by Jira

Applicable contexts for scheme: [Edit Configuration](#)

Issue type(s):
Global (all issues)
Project(s):
A-Test
SD
test

Default value: (Read Only - License is invalid) [Edit Default value](#)

- Options:
- 1
 - 1.1
 - 1.1.1
 - 1.1.2
 - 1.2
 - 1.2.1
 - 2
 - 2.1
 - 3.1
 - 3
 - 3.1
 - 4
 - 4.1
 - 5.1
 - 5
 - 5.1

[Edit Options](#)

Export json tree file on right side (required plugin version 6.21.0 or newer)

The screenshot shows the Jira settings interface for a project named 'ptkdev'. At the top, there is a search bar and several icons. Below the header, there are buttons for 'Add option', 'Import options', 'Sort options alphabetically', and 'Export CustomFields'. A red box and arrow highlight the 'Export CustomFields' button. The main area displays a list of custom fields with 'Configure', 'Edit', 'Disable', and 'Delete' actions. The entire interface is framed by a red border.

Import json tree from MLCS Cloud

Go to **Settings App**

The screenshot shows the Jira Settings page. At the top right, there is a gear icon. Below it, under the 'JIRA SETTINGS' section, there is a 'Aops' section with a red box around it. The 'Aops' section contains the following text: 'Aops' and 'Add and manage Jira Marketplace apps.'

Go to **MLCS Cloud** on left side and open tab "Import / Export"

The screenshot shows the 'MLCS Cloud' interface with the 'Import / Export' tab selected. On the left, there is a sidebar with sections like 'ATLASSIAN MARKETPLACE', 'APPS', and a list of installed apps including 'MLCS Cloud' (which is highlighted with a red box). In the main area, there are three import sections: 'Import CustomFields Tree (BETA)', 'Import Data (BETA)', and 'Import CSV (BETA)'. The 'Import CustomFields Tree (BETA)' section is highlighted with a red box. It contains a file input field 'Scegli file' with the placeholder 'Nessun file selezionato', a checkbox 'Merge backup with existing data', and a blue 'Import' button.

Choose json file from MLCS Server / DC and import it.

Migrate Custom Fields Type from all issue (multi selects values)

There are two ways to migrate MLCS field values from server to cloud:

1. Using xml data importer

Go to server instance, Issues Search for issues

The screenshot shows the Jira navigation bar with 'Issues', 'Boards', 'Plans', and 'Insight' tabs. Below the navigation bar is a search bar labeled 'Search for issues'. A red arrow points from the text 'Current search' to the search bar. To the right of the search bar is a section titled 'Archived issues'.

RECENT ISSUES

On right side, **Export XML**

The screenshot shows the 'Export' dropdown menu in Jira. The menu includes options like 'Printable', 'Full Content', 'RSS (Issues)', 'RSS (Comments)', 'CSV (All fields)', 'CSV (Current fields)', 'HTML (All fields)', 'HTML (Current fields)', 'XML', 'Word', and 'Dashboard charts'. A red arrow points from the text 'XML' to the 'XML' option in the menu.

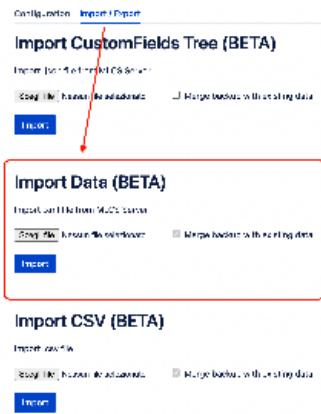
hing this issue

Create the field if it does not exist.

Open the XML file using your preferred editor, and replace all instances of the server's MLCs custom field ID (ex: customfield_13243) with the cloud MLCs custom field id

The screenshot shows a search and replace dialog. The search term 'customfield_10083' is entered in the search field. The replace field contains the placeholder 'Replace' and has a 'Replace' button. There are also 'AB' and '...' buttons.

Now you can go to **Cloud** version of plugin and **Import Data (XML)**:



2. Using ScriptRunner Console

Configuration Steps

1. Set Up Jira Cloud API Access:

- Determine your Jira Cloud REST API Base URL. It typically follows the format `https://[YourInstance].atlassian.net`.
- Create an API token from your Atlassian account. Follow Atlassian's guide for creating an API token.
- Combine your email and API token in the format `email:token`. This will be used for basic authentication.

2. Identify the Custom Field:

- Find the name of the MLCS custom field in your Jira Server instance that you wish to migrate.
- Obtain the ID of the corresponding custom field in your Jira Cloud instance.

3. Update the Script:

- Replace `jiraCloudApiBaseUrl` with your Jira Cloud instance's base URL.
- Update `encodedAuthString` with your email and API token.
- Update `customFieldName` with the name of your MLCS custom field in the Jira Server instance.
- Replace `cloudCustomFieldId` with the ID of the corresponding custom field in your Jira Cloud instance.

```
import com.atlassian.jira.issue.customfields.option.LazyLoadedOption
import com.atlassian.jira.component.ComponentAccessor
import com.atlassian.jira.issue.fields.CustomField
import com.atlassian.jira.issue.Issue
import com.atlassian.jira.issue.CustomFieldManager
import com.atlassian.jira.security.JiraAuthenticationContext
import com.atlassian.jira.bc.issue.search.SearchService
import com.atlassian.jira.issue.search.SearchException
import com.atlassian.jira.web.bean.PagerFilter
import groovy.json.JsonOutput
import groovyx.net.http.RESTClient
import groovyx.net.http.ContentType

// Jira Cloud REST API base URL
final String jiraCloudApiBaseUrl = "https://[YourInstance].atlassian.net"

// Basic Authentication Encoded String
def encodedAuthString = "your email:token".bytes.encodeBase64().toString()

// Get necessary components
def issueManager = ComponentAccessor.getIssueManager()
def customFieldManager = ComponentAccessor.getCustomFieldManager()
def authenticationContext = ComponentAccessor.getJiraAuthenticationContext()
def searchService = ComponentAccessor.getComponent(SearchService.class)

// Define and find the custom field
final String customFieldName = "mlcs employee"
final String cloudCustomFieldId = "customfield_10101"
def customField = customFieldManager.getCustomFieldObjectsByName(customFieldName)?.first()
```

```

if (customField == null) {
    log.error "Custom field not found: $customFieldName"
    return
}

// Fetch issues
def user = authenticationContext.getLoggedInUser()
def query = "\"${customFieldName}\" is not EMPTY"

// Parse the JQL query
def parseResult = searchService.parseQuery(user, query)

def transformElement(LazyLoadedOption element, int index) {
    String label = element.getValue()
    return [label: label, value: index + 1]
}

if (!parseResult.isValid()) {
    log.error "Invalid JQL Query: ${query}"
    return
}

try {
    def results = searchService.search(user, parseResult.getQuery(), PagerFilter.getUnlimitedFilter())

    results.getResults().each { Issue issue ->
        List serverValue = issue.getCustomFieldValue(customField)
        if (serverValue) {

            def transformedArray = [:]

            // Populate the map
            serverValue.eachWithIndex { element, index ->
                transformedArray["lv$index"] = transformElement(element as LazyLoadedOption, index)
            }

            def payload = [
                fields: [
                    (cloudCustomFieldId): JsonOutput.toJson(transformedArray)
                ]
            ]

            def client = new RESTClient(jiraCloudApiBaseUrl)

            client.setHeaders([
                'Content-Type' : ContentType.JSON,
                'Authorization': "Basic $encodedAuthString"
            ])

            try {
                def response = client.put(
                    path: "/rest/api/3/issue/${issue.key}",
                    contentType: ContentType.JSON,
                    body: payload
                )

                if (response.status != 204) {
                    log.error("Failed to update issue ${issue.key}: ${response.data}")
                } else {
                    log.info("Successfully updated issue ${issue.key}")
                }
            } catch(Exception e) {
                log.error("Error updating issue ${issue.key}: ${e.message}")
            }
        }
    }
} catch (SearchException e) {
    log.error("Error executing search: ${e.message}")
}

```

}

Execution

1. Run the Script:

- Open the ScriptRunner Console in your Jira Server instance.
- Paste the updated script into the console.
- Execute the script.

The screenshot shows the Jira ScriptRunner Console. The top navigation bar includes links for Dashboards, Projects, Issues, Boards, Plans, Create, Search, and ScriptRunner. The left sidebar has sections for Administration, Applications, Projects, Issues, Manage apps, User management, Latest upgrade report, System, and ScriptRunner. Under ScriptRunner, there are links for Browse, Built-in Scripts, Jobs, Listeners, Fields, Behaviours, Workflows, Fragments, JQL Functions, REST Endpoints, Resources, Mail Handler, and Script Editor. The main content area is titled 'Script Runner' and contains a 'Script Console' section with instructions to run one-off ad-hoc scripts or experiment with the Jira API. It features a 'Script' text input field and a 'File' tab. Below this is a 'Documentation & Tips' section and a 'Show' button. The bottom half of the screen is a code editor window titled 'Script' with the following Groovy code:

```
1 import com.atlassian.jira.issue.fields.option.LazyLoadedOption
2 import com.atlassian.jira.component.ComponentAccessor
3 import com.atlassian.jira.issue.fields.CustomField
4 import com.atlassian.jira.issue.Issue
5 import com.atlassian.jira.user.ApplicationUser
6 import com.atlassian.jira.security.JiraAuthenticationContext
7 import com.atlassian.jira.bc.issue.search.SearchService
8 import com.atlassian.jira.issue.search.SearchException
9 import com.atlassian.jira.web.filters.webbb.filter.PageFilter
10 import groovy.json.JsonOutput
11 import groovy.json.http.RESTClient
12 import groovy.json.http.ContentType
13
14 // Jira Cloud REST API base URL
15 final String jiraCloudBaseUrl = "https://[YourInstance].atlassian.net"
16
17 // Basic Authentication Encoded String
18 def encodedAuthString = "your_email:[token].bytes.encodeBase64().toString()"
19
20 // Get necessary components
21 def issueManager = ComponentAccessor.getIssueManager()
```

Troubleshooting

If issues fail to update, check the log for error messages. Ensure that all instance URLs, field names, and IDs are correct. Verify that your API token has appropriate permissions. Also, be aware of the following HTTP status codes and their implications:

1. 400 Bad Request

- This error occurs if the user lacks the necessary permissions to edit the issue or to view it, or if the mlcs field is not found in the cloud or not associated with the issue's edit screen.

2. 401 Unauthorized

- This status is returned if the token or the email is not valid.

3. 403 Forbidden

- This can happen if the user does not have the necessary permissions to edit the issue or to view it.

4. 404 Not Found

- Returned if the issue is not found or the user does not have permission to view it.