

MD Git Integration

Integration of Git within MDCMS

Table of contents

1. MDCMS CI from Git	5
1.1 Setup & Configuration	5
1.2 Users & Access	5
1.3 Commits	5
1.4 Pull Requests	5
1.5 Local Workspace	6
1.6 Export & Integration	6
2. Setup	7
2.1 MDCMS Git Integration Overview	7
2.1.1 Introduction	7
2.1.2 Architecture and Components	7
2.1.3 Setup & Configuration Workflow	7
2.1.4 Attribute Mapping: The Foundation	8
2.1.5 Continuous Integration: Automated Code Deployment	9
2.1.6 Mapping & Assignment Configuration	10
2.1.7 Branch Management	11
2.1.8 Commits & Pull Requests	11
2.1.9 Local Workspace Integration	13
2.1.10 Recommended Git Flows	13
2.1.11 Integration Patterns	14
2.1.12 MDSRC2IFS: Source to IFS Conversion	14
2.1.13 Complete Setup Example	15
2.1.14 Testing the Integration	16
2.1.15 Best Practices	16
2.1.16 Security Considerations	17
2.2 Git Credentials	18
2.2.1 General Settings	18
2.2.2 SSH Authentication Settings	18
2.2.3 Generate New Key Pair	18
2.2.4 User/Password Authentication Settings	19
2.3 API Servers	20
2.3.1 General Settings	20
2.3.2 Azure DevOps Settings	21
2.3.3 Bitbucket Cloud Settings	21
2.3.4 Bitbucket Data Center (or Server) Settings	22

2.3.5	GitHub Settings	22
2.3.6	GitLab Settings	22
2.4	Git Repositories	23
2.4.1	Repository Settings	23
2.4.2	After Saving	24
2.5	Continuous Integration Definitions	25
2.5.1	Branch and Path	25
2.5.2	Target Application and Level	25
2.5.3	What to Request	26
2.5.4	Project and Task Assignment	27
2.5.5	Object Request Automation	28
2.5.6	Pipeline Integration	28
2.6	Task Type Mapping for Application and Level during CI	29
2.7	Repository Attribute Mapping	30
2.7.1	How MDCMS Resolves Attributes	30
2.7.2	Defining Attribute Mapping for an Individual Attribute	30
2.7.3	Bulk Mapping Generation using the Missing Paths Feature	31
2.8	Map Git Users to MDCMS Users	33
2.9	Git Branch Creation Rules	34
2.9.1	Settings	34
2.9.2	Reviewing and Manually Creating Branches	35
2.9.3	Check out a Created Branch in the Local Workspace	35
2.10	Webhook Configuration	36
2.10.1	Register Webhooks from MDOpen	36
2.10.2	Manual Webhook Registration	37
3.	Commits	39
3.1	Git Commits	39
3.1.1	Header Options	39
3.1.2	Row Options	39
3.2	Commit Templates	40
3.2.1	Git Commit Template Parameters	40
3.2.2	Attributes Assigned to Template	42
3.3	Configure GPG Signing for Git Commits	44
3.3.1	Install the Open Source Package gnupg2	44
3.3.2	Generate a GPG Key for the MD Service User	44
4.	Pull Requests	46
4.1	Git Pull Requests	46
4.1.1	Header Options	46

4.1.2	Row Options	46
4.2	Pull Request Levels	47
4.2.1	Pull Request Level Parameters	47
4.2.2	Automated Creation of Pull Requests	47
4.2.3	What to Request	48
4.2.4	Project and Task Assignment	49
4.2.5	Object Request Automation	50
4.2.6	Pipeline Integration	51
5.	Local Workspace	52
5.1	Local Workspace Mapping	52
5.1.1	Ways to Create a Local Workspace Mapping Entry	52
5.1.2	Usage of Mapping	52
6.	Export & Integration	53
6.1	MDSRC2IFS - Convert Source Members to IFS	53
6.1.1	Overview	53
6.1.2	Restrictions	53
6.1.3	Parameters	53
6.1.4	The MDSRC2IFS Confirmation Screen	54
6.2	Export IFS Content to Git	55
6.2.1	Before you Begin	55
6.2.2	Export to Git	55

1. MDCMS CI from Git

Welcome to the Git Integration guide for MDCMS. This documentation covers the integration of Git with Midrange Dynamics MDCMS for continuous integration and deployment workflows.

:fontawesome-regular-file-pdf: [Download PDF](#)

1.1 Setup & Configuration

Before implementing Git integration, review the following setup topics:

- [Overview & Prerequisites](#)
- [Git Credentials](#)
- [API Servers](#)
- [Git Repositories](#)
- [Continuous Integration Settings](#)
- [Task Type Mapping](#)
- [Attribute Mapping](#)
- [Branch Creation Rules](#)
- [Webhooks](#)

1.2 Users & Access

Learn about user and access management:

- [User Mapping](#)
- [Git Credentials](#)

1.3 Commits

Understand commit management:

- [Commits](#)
- [Commit Templates](#)
- [Configure GPG Signing](#)

1.4 Pull Requests

Explore pull request functionality:

- [Pull Requests](#)
- [Pull Request Levels](#)

1.5 Local Workspace

Work with local environments:

- [Local Workspace Mapping](#)

1.6 Export & Integration

Initial Loading of a Git Repository:

- [MDSRC2IFS](#)
- [Export IFS to Git](#)

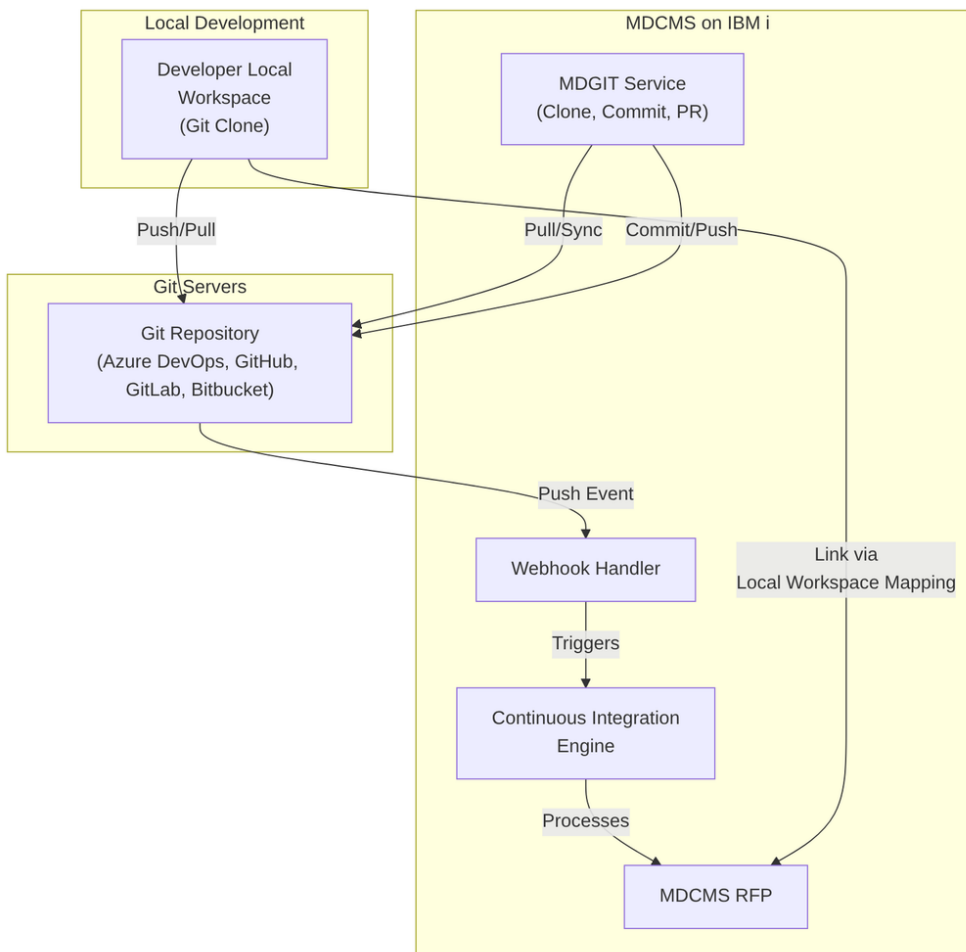
2. Setup

2.1 MDCMS Git Integration Overview

2.1.1 Introduction

MDCMS Git Integration enables seamless integration between Midrange Dynamics MDCMS and popular Git platforms (Azure DevOps, Bitbucket, GitHub, GitLab). This integration automates code deployment, branch management, commit tracking, and pull request workflows, bridging the gap between IBM i development and modern DevOps practices.

2.1.2 Architecture and Components



2.1.3 Setup & Configuration Workflow

Before using Git integration, configure the following components in order:

1. Git Credentials

Purpose: Authenticate with the Git server using SSH or username/password.

Key Options: - **SSH Authentication** — Recommended for production (ed25519 or RSA keys) - **User/Password Authentication** — Simpler setup, less secure

Configuration Path: Settings → DevOps Settings → Git Credentials → Add

Example:

```
Credential ID: GITHUB-MAIN
Description: GitHub primary credentials
Auth Method: SSH
SSH Key: github-main-key
Passphrase: [protected]
```

2. API Servers (Optional but Recommended)

Purpose: Enable advanced features like automated commit/PR creation and webhook registration.

Supported Platforms: - Azure DevOps - Bitbucket Cloud & Data Center - GitHub - GitLab

Configuration Path: Settings → DevOps Settings → API Servers → Add

Benefits: - Automatic commits after RFP deployment - Automatic pull request creation - Direct webhook registration from MDOpen - Access to pull request and commit history

Example Configuration:

```
Server ID: GITHUB-API
Description: GitHub API for repositories
Server Type: GitHub
API URL: https://api.github.com
User: your.email@company.com
Token: ghp_XXXXXXXXXXXXXXXXXXXX
```

3. Git Repositories

Purpose: Define the connection to each Git repository and configure MDGIT jobs.

Configuration Path: Settings → DevOps Settings → Git Repositories

Key Parameters: - **Repository ID** — Unique identifier for the repo - **URL** — HTTP(s) or SSH clone URL - **Git Type** — Azure DevOps, Bitbucket, GitHub, GitLab, or Other - **Credentials** — Link to configured credentials - **API Server** — Link to API server for advanced features - **MDGIT Jobs** — Primary/secondary/tertiary job numbers for high availability - **Main Branch** — Primary branch (main, master, develop)

Example:

```
Repository ID: MYAPP
URL: git@github.com:myorg/myapp.git
Git Type: GitHub
Credentials: GITHUB-MAIN
API Server: GITHUB-API
MDGIT Jobs: 1, 2 (for high availability)
Main Branch: main
```

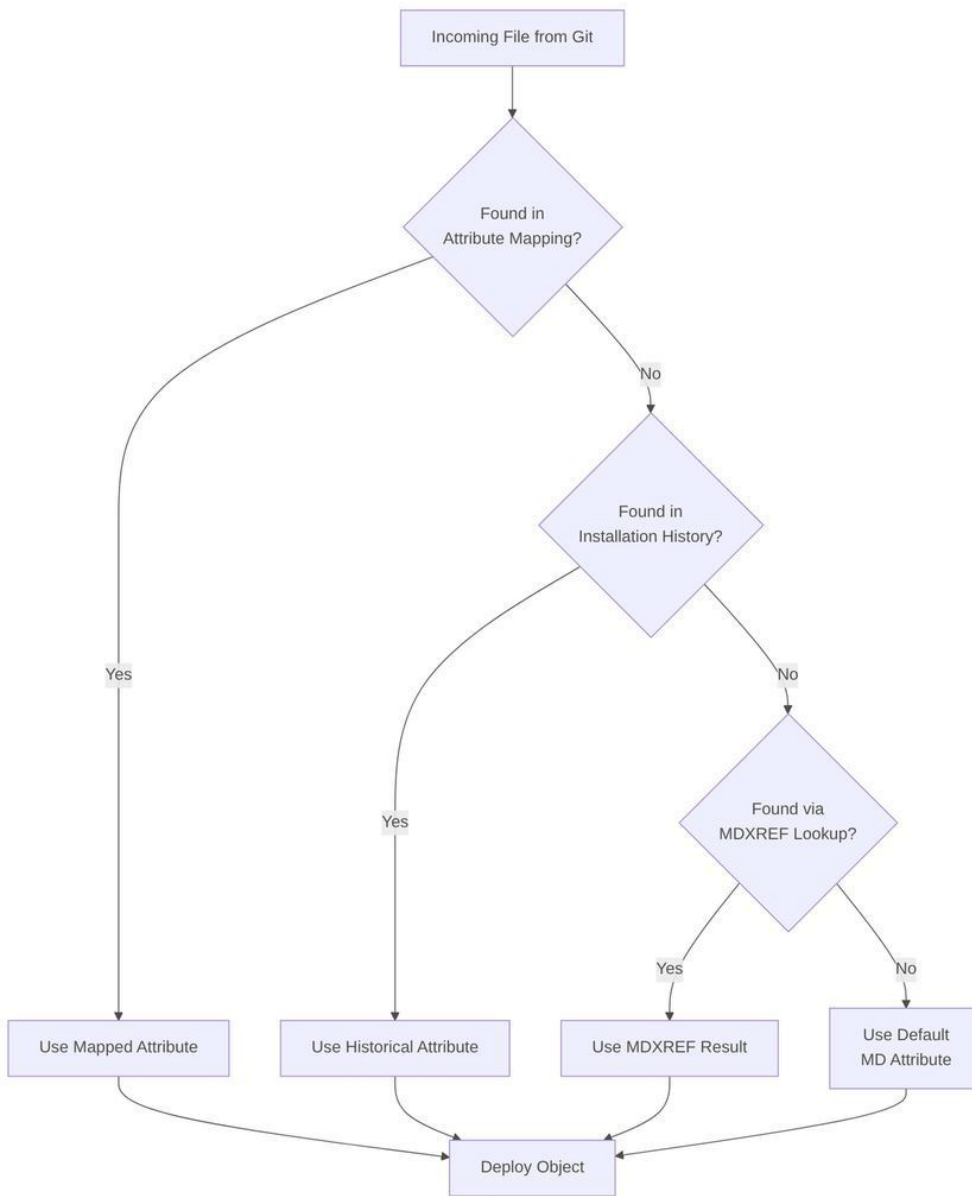
2.1.4 Attribute Mapping: The Foundation

Purpose: Map Git file paths and names to MDCMS attributes for proper deployment.

Configuration Path: Git Repositories → [repository] → Attribute Mapping

Attribute mapping is critical for determining: - How files are deployed (IBM i source vs. IFS vs. Remote) - Where files are deployed on the IBM i - Which MDCMS attribute to use

Mapping Resolution Order for Source Objects:



Example Attribute Mapping Table:

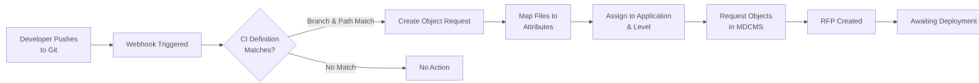
Application	Path	File Pattern	MD Attribute	Purpose
MYAPP	/src/rpg	*.rpgle	RPGLE	RPG ILE source
MYAPP	/src/dds	*.dspf	DSPF	Display file source
MYAPP	/src/sql/tables	*.sql	SQLTAB	SQL table source
MYAPP	/deployment	*	IFS	IFS deployment files
MYAPP	/test	*	*NONE	Exclude test files

2.1.5 Continuous Integration: Automated Code Deployment

Purpose: Define how Git pushes trigger automatic code deployment in MDCMS.

Configuration Path: Settings → DevOps Settings → Continuous Integrations

CI Definition Workflow



CI Definition Parameters:

Branch & Path Filtering: - **Main Branch Only** — CI on primary branch only - **Not Main Branch** — CI on feature branches only - **Branch Pattern** — `release-*`, `feat-*`, `hotfix-*` - **Specific Branch** — Exact branch name - **Path Filter** — `/src/rpg/*`, `/src/dds/*`, etc.

Target Deployment: - **Application** — Where objects deploy - **Level** — Which environment (DEV, TST, QA, PROD) - **Level is Target of Merge** — For PR-triggered deployments to non-checkout levels

What to Request: - **Checkout: Diffs** — Only changed files - **Checkout: Contents** — Complete folder contents - **Request IFS/Remote Objects** — Deploy to IFS/remote servers - **Request Source for IBM i Objects** — Deploy to IBM i source

Example CI Definition:

```

Repo ID: MYAPP
Branch Opt: Branch Naming Pattern
Branch Pattern: feat-*
Path: /src
File Pattern: *.{rpgle,sql,dds}

Application: MYAPP
Level: 1 (DEV)

Checkout: Diffs
Request IFS/Remote Objects: false
Request Source for IBM i Objects: true

Default MD Attribute: RPGLESRC
Default User: ADMIN
  
```

2.1.6 Mapping & Assignment Configuration

User Mapping

Purpose: Map Git user IDs to MDCMS user IDs.

Configuration Path: Git Repositories → [repository] → User Mapping

Use Cases: - Git username differs from MDCMS username - CI-generated requests need proper attribution - Different user repositories (GitHub vs. Azure DevOps)

Example:

```

Repository User: john.doe@github
MDCMS User: JDOE
Scope: *ANY (applies to all repositories)
  
```

Task Type Mapping

Purpose: Route deployments to different applications/levels based on task type.

Configuration Path: Git Repositories → [repository] → Task Type Mapping

Example:

```

Task Type: FEATURE
Target Application: MYAPP
Target Level: 110 (DEV branch 1)

Task Type: BUGFIX
Target Application: MYAPP
Target Level: 100 (DEV trunk)
  
```

```
Task Type: HOTFIX
Target Application: MYAPP
Target Level: 199 (DEV emergency)
```

2.1.7 Branch Management

Git Branch Creation Rules

Purpose: Automatically create Git branches when tasks reach specified statuses.

Configuration Path: DevOps Settings → Git Branch Creation Rules

Benefits: - Prevents naming inconsistencies - Eliminates manual branch creation errors - Enforces naming conventions

Example Branch Creation Rule:

```
Project: MYPROJ
Task Type: FEATURE
Repository ID: MYAPP
Branch Name: feat-++PRJTSK++
Create from: develop
Automatic Creation: true
Minimum Status: 3 (In Progress)
Generate for Subtasks: true
```

This creates branches like `feat-MYPROJ-123.1` for task MYPROJ-123, subtask 1.

Branch Naming Placeholders:

Placeholder	Example	Description
++PRJ++	MYPROJ	Project code
++TSK++	123	Task number
++PRJTSK++	MYPROJ-123	Project-Task
++PRJTSKSUB++	MYPROJ-123.1	Project-Task.Subtask

2.1.8 Commits & Pull Requests

Git Commit Templates

Purpose: Automatically commit IBM i code changes back to Git after RFP deployment.

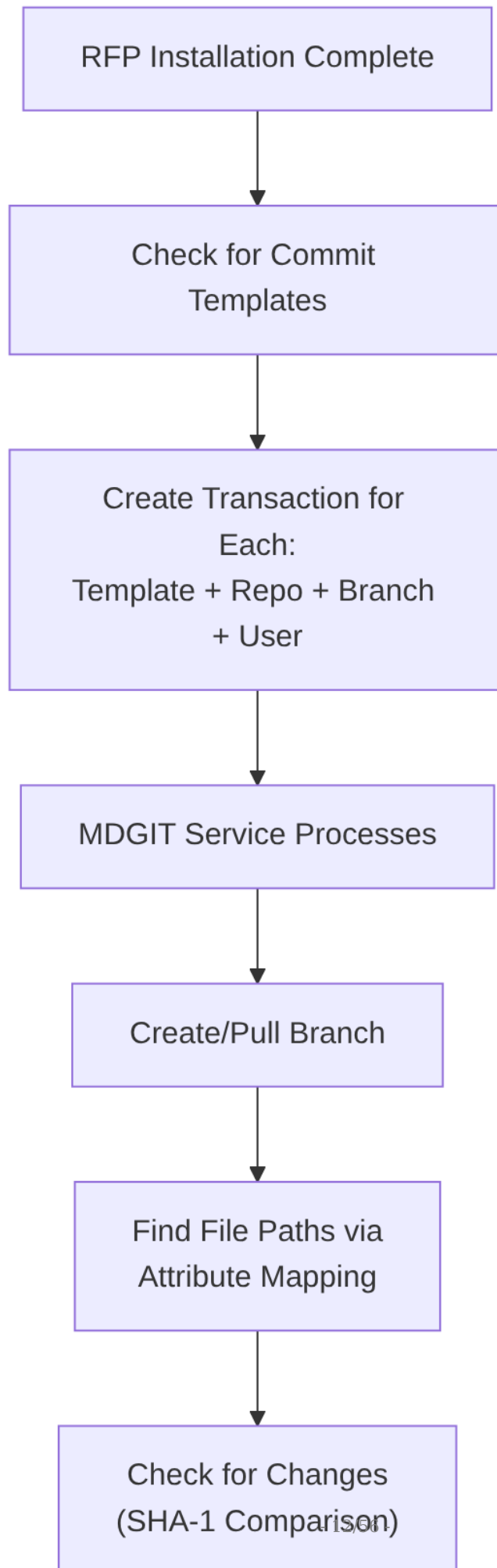
Configuration Path: Settings → Template Settings → Git Commit

Key Features: - Auto-commit on RFP installation - GPG signing support - Customizable commit messages - Branch-based or fixed-branch commits

Example Commit Template:

```
Template ID: AUTOCOMMIT
Repo ID: MYAPP
Commit to CI Branch: true
Branch Name: feat-++PRJTSK++
Commit Message: Deploy ++PRJTSK++ via RFP ++RFPNBR++
Committer Name: *USER
GPG Signing: true
End of Line: LF
Auto-Link to Local Workspace: true
```

Commit Process:



Pull Request Levels

Purpose: Define automated pull request creation at deployment levels.

Configuration Path: Settings → DevOps Settings → Pull Request Levels

Workflow: - When RFP deploys to a level with a PR Level defined, MDCMS automatically creates a PR - PR merges changes from feature branch to target branch - Target branch merge can trigger higher-level CI

Example PR Configuration:

```
Application: MYAPP
Level: 1 (DEV Feature Branch)
Repository: MYAPP
PR Target Branch: develop
PR Title: Feature ++PRJTSK++ deployed via RFP ++RFPNBR++

---

Application: MYAPP
Level: 2 (QA)
Repository: MYAPP
PR Target Branch: release
PR Title: Release ++PRJTSK++ ready for production
Create from Branch: develop
```

2.1.9 Local Workspace Integration

Purpose: Link developer's local Git clone to MDCMS for editing code in preferred IDEs.

Configuration Path: Settings → DevOps Settings → Local Workspace Mapping

Benefits: - Edit code in VS Code, IntelliJ, etc. - Leverage AI code assistance - Maintain MDCMS traceability - Easy sync between local workspace and IBM i

Example Mapping:

```
Repository ID: MYAPP
Local Workspace Path: C:\dev\myapp
```

Developer Workflow with Local Workspace:



2.1.10 Recommended Git Flows

1. Feature Branch Flow (Recommended for Most Teams)

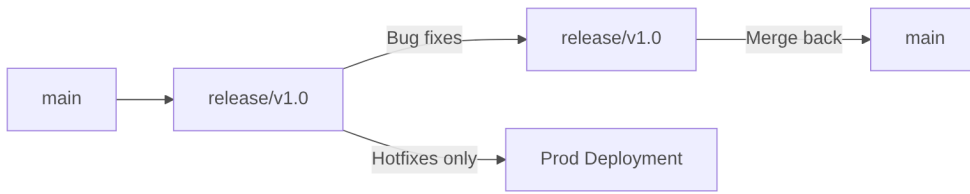


MDCMS Configuration: - **DEV Branch:** `feat-*` → Level 1 - **QA Branch:** `develop` → Level 2 - **PROD Branch:** `main` → Level 3 (merge target)

Branch Creation Rules:

```
feat-++PRJTSK++ created when Task status = "In Progress"
Auto-commit to same branch on RFP deployment
Auto-PR to develop on Level 1 deployment
Auto-PR to main on Level 2 deployment
```

2. Release Branch Flow



CI Configuration:

Branch Pattern: release-*
Auto-deploy to QA/PROD for specific release branches

3. Hotfix Flow

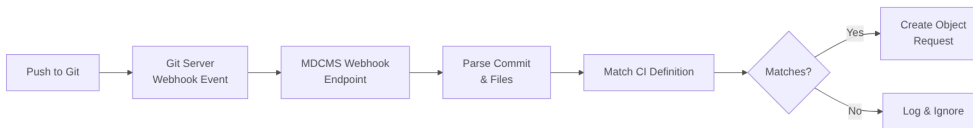


Branch Rules:

hotfix-++PRJTSK++ created from main
Deploy directly to QA/PROD

2.1.11 Integration Patterns

Pattern 1: Webhook-Triggered CI

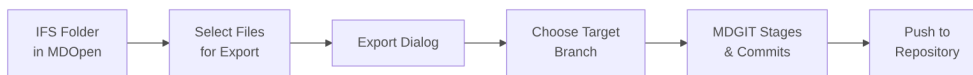


Webhook Registration: - Automatic: Via API Server in MDOpen - Manual: Configure on Git server (Azure DevOps, GitHub, etc.)

Pattern 2: Commit Back to Git



Pattern 3: Export IFS to Git



Use Cases: - Initial code migration from source files to Git - Bulk IFS file updates - Repository seeding

2.1.12 MDSRC2IFS: Source to IFS Conversion

Purpose: Convert IBM i source members to IFS files for Git-friendly format.

Command: MDSRC2IFS

Parameters: - **LIB** — Source library - **FILE** — Source file containing members - **PATH** — Target IFS parent path - **CCSID** — Character encoding (default: UTF-8) - **REPL** — Replace existing files (Y/N)

Example:

```
MDSRC2IFS LIB(APPLIB) FILE(QRPGLSRC) PATH(/myapp/src/rpg)
CCSID(1208) REPL(*YES)
```

Output Structure:

```
/myapp/src/rpg/
├── PROGRAM1.rpgle
├── PROGRAM2.rpgle
├── QRPGLSRC.subfolder/
│   ├── MODULE1.rpgle
│   └── MODULE2.rpgle
```

2.1.13 Complete Setup Example

Company: TechCorp**Scenario:** Migrating RPG application to Git with MDCMS CI/CD**Step 1: Create Credentials**

```
ID: TECHCORP-GH
Type: SSH (ed25519)
Key: techcorp-main
```

Step 2: Create API Server

```
ID: GITHUB-MAIN
URL: https://api.github.com
Token: ghp_xxxxx
User: devops@techcorp.com
```

Step 3: Create Repository

```
ID: CRMDEMO
URL: git@github.com:techcorp/crmdemo.git
Credentials: TECHCORP-GH
API Server: GITHUB-MAIN
MDGIT Jobs: 1, 2
Main Branch: main
```

Step 4: Attribute Mapping

```
/src/rpg/*.rpgle -- RPGLESRC (RPG ILE)
/src/rpg/*.bndrpg -- BNDRPG (Binding)
/src/dds/*.dds -- DDS (Display files)
/src/sql/*.sql -- SQLSRC (SQL)
/sql/procedures/*.sql -- SQLSPROC (Procedures)
/build/* -- *NONE (Exclude)
/docs/* -- *NONE (Exclude)
/deployment/* -- *IFS (IFS deployment)
```

Step 5: User Mapping

```
jsmith@github.com -- JSMITH
kdoe@github.com -- KDOE
```

Step 6: CI Definitions

```
DEV CI:
Branch: feat-*
Path: /src
Level: 1 (DEV)
Request: Source Objects
Checkout: Diffs

QA CI:
Branch: develop
Path: /src
Level: 2 (QA)
Request: Source Objects
Checkout: Diffs
```

Step 7: Branch Creation Rules

```
Rule 1:
Branch Name: feat-++PRJTSK++
Create from: develop
Auto-create: when Task status = "In Progress"

Rule 2:
```

```
Branch Name: release-v1.0
Create from: main
Auto-create: when Task status = "Release Planned"
```

Step 8: Pull Request Levels

```
Level 1 (DEV):
PR from feat-* to develop
Title: Feature ++PRJTSK++ via RFP ++RFPNBR++

Level 2 (QA):
PR from develop to main
Title: Release ++PRJTSK++ to production
```

Step 9: Commit Templates

```
Template: AUTOCOMMIT
Branch: feat-++PRJTSK++
Message: "RFP ++RFPNBR++ - ++PRJTSK++"
Committer: *USER
GPG Sign: true
```

Step 10: Local Workspace Mapping

```
Repo: CRMDEMO
Path: C:\dev\crmdemo
```

2.1.14 Testing the Integration

Verification Checklist:

- [] Repository connection test via "Test Connection" in MDOpen
- [] Webhook delivery confirmed in Git server logs
- [] Test commit push to feature branch triggers CI
- [] Object request created with correct attributes
- [] Branch created automatically for test task
- [] Commit template creates proper Git commit
- [] Pull request created for deployment level
- [] Local workspace mapping allows syncing

Troubleshooting:

Connection Issues: - Check MDGIT logs: MDOpen → Git Commits → MDGIT Logs - Verify credentials and permissions - Check firewall/proxy rules

CI Not Triggering: - Verify webhook delivery in Git server - Check CI definition branch/path filters - Review attribute mapping

Commits Not Created: - Verify commit template configured - Check MDGIT logs for staging errors - Validate attribute mapping for relative paths

2.1.15 Best Practices

1. **Branch Naming Conventions** — Use consistent, meaningful patterns
2. **Attribute Mapping** — Define mappings for every application/path combination
3. **High Availability** — Configure multiple MDGIT jobs (Clone 1, 2, 3)
4. **GPG Signing** — Enable for production commits
5. **PR Reviews** — Always review before merging to main/release
6. **Webhook Security** — Allowlist Git server IPs when possible
7. **Local Workspace** — Use for complex development with IDE features
8. **Test First** — Thoroughly test in DEV before QA deployments

9. **Documentation** — Keep mapping and rules documented for team reference
10. **Regular Sync** — Keep local clones updated with remote branches

2.1.16 Security Considerations

- Use SSH keys (ed25519) over username/password
- Enable GPG signing for commits
- Restrict webhook access via firewall rules
- Use separate credentials for each environment
- Rotate credentials regularly
- Audit commit history for changes
- Review API token permissions (minimal required)

2.2 Git Credentials

Most Git servers require authentication to communicate with a repository. If your repositories allow anonymous access, skip this section.

In MDOpen: **Settings** → **DevOps Settings** → **Git Credentials** → **Add**

2.2.1 General Settings

CREDENTIAL ID

A unique identifier of up to 10 characters for this authentication configuration.

DESCRIPTION

A human-readable description of the credentials.

GIT AUTHENTICATION

The authentication method to use when connecting to the Git server.

- **SSH** — A private/public key pair is used for authentication.
 - **User/Password** — A username and password are used for authentication.
-

2.2.2 SSH Authentication Settings

SSH KEY

The name of the private key registered in MDCMS. Use the content-assist icon to select from a list. To create a new key pair, click the **Generate New Key Pair** button (see below).

Note: MDCMS no longer supports SHA1 keys, which have been broadly deprecated by the SSH community.

SET NEW PASSPHRASE

If the private key is protected by a passphrase, enter it here.

2.2.3 Generate New Key Pair

KEY NAME

A distinct name for the private key. The public key will be given the same name with a `.pub` suffix.

KEY TYPE

The cryptographic algorithm to use for the key pair. **ed25519** is recommended. **RSA** is a legacy option; if selected, also set the RSA Size in Bits.

RSA SIZE IN BITS

Applies only when Key Type is RSA. Options are 1024, 2048, 3072 (recommended), or 4096 bits.

COMMENT

An optional comment appended to the end of the public key value.

PASSPHRASE

A secret value used to protect the private key.

After generating the key pair, click **View Public Key** and copy the entire string from the `.pub` editor. Add that value to the list of authorized keys on the origin Git server, then click **Save** to store the Credential definition.

2.2.4 User/Password Authentication Settings

REPO USER

The user ID of the account registered on the Git server.

SET NEW PASSWORD

The password for the Git server account.

Click **Save** once the values are entered.

2.3 API Servers

Defining an API Server is optional, but provides several significant advantages if your repositories are hosted on Azure DevOps, Bitbucket, GitHub, or GitLab and you have a MDWorkflow Pipeline license:

1. Commits of changes made to the code directly on the IBM i can be automatically performed by MDCMS on the developer's behalf, eliminating the need for developers to perform manual Git operations in their local workspace.
2. Pull Requests can be automatically generated once MDCMS completes deployment to a specified target level.
3. Webhook definitions can be generated directly from within MDOpen.

If none of these features applies to your situation, skip to [Git Repositories](#).

If you already have an API Server defined for Pipeline or Task features that points at the correct server for the correct project or organization, you can also use that API Server for Git features in MDOpen.

In MDOpen: **Settings** → **DevOps Settings** → **API Servers** → **Add**

2.3.1 General Settings

SERVER ID

A unique identifier of up to 10 characters for this server configuration.

DESCRIPTION

A human-readable description of the server.

SERVER TYPE

The type of Git server being connected to. One of: Azure DevOps, Bitbucket, GitHub, or GitLab.

API SERVER URL

The base URL of the Git server, for example `https://dev.azure.com`.

PROXY ADDRESS

The URL of the proxy server to use when connecting to the Git server, if applicable.

PROXY PORT

The port of the proxy server, if applicable.

PROXY TYPE

The type of the proxy server. One of: HTTP or SOCKS5

PROXY USER

The user to authenticate with the proxy server, if applicable.

SET NEW PROXY PASSWORD

The password to authenticate with the proxy server, if applicable.

AUTOMATICALLY FOLLOW REDIRECTS

- **True** — Automatically follow HTTP redirects when connecting to the Git server. This is typically recommended unless your server is known to have misconfigured redirects.
- **False** — Do not follow HTTP redirects when connecting to the Git server. This may be necessary if your server is known to have misconfigured redirects, but in general should be set to True.

FOLLOW AUTHORIZATION HEADER ON REDIRECTS

- **True** — Follow the Authorization header when following HTTP redirects.
- **False** — Do not follow the Authorization header when following HTTP redirects.

MAXIMUM NUMBER OF REDIRECTS

The maximum number of HTTP redirects to follow when connecting to the Git server.

ALLOW CERTIFICATE AUTHENTICATION ERRORS

- **True** — Allow connections to the Git server even if there are SSL certificate errors, such as an expired certificate or a hostname mismatch. This is not recommended for production environments, but may be necessary in some cases such as when connecting to a server with a self-signed certificate.
- **False** — Do not allow connections to the Git server if there are SSL certificate errors. This is typically recommended for production environments.

2.3.2 Azure DevOps Settings

USER

The user ID on the server, typically an email address.

SET NEW TOKEN

A valid Personal Access Token with rights to repositories, webhooks, and pull requests. In Azure DevOps, generate this via **User Settings → Personal Access Tokens**.

ORGANIZATION

The name of the Azure DevOps organization that the project containing the repositories belongs to.

PROJECT

The name of the Azure DevOps project containing the repositories.

QUEUE NUMBER

The MDAZUR job number that should be used for importing Work Items from Azure DevOps. This parameter is not relevant for Git features.

2.3.3 Bitbucket Cloud Settings

TOKEN TYPE

Specifies which kind of token will be used for authentication.

- **Access Token** — A token defined at the repository, project, or workspace level.
- **API Token** — A token defined for a specific user.

USER

The user the API Token belongs to. Leave blank if using an Access Token.

SET NEW TOKEN

A valid Access Token or API Token with rights to repositories, webhooks, and pull requests.

WORKSPACE

The ID of the Bitbucket workspace containing the repositories. This value is visible in the URL when viewing the Repositories list.

2.3.4 Bitbucket Data Center (or Server) Settings

USER

The user ID on the Bitbucket Data Center or Server instance.

SET NEW TOKEN

A valid Personal Access Token with rights to repositories, webhooks, and pull requests. In Bitbucket, generate this via **Manage User → Personal Access Tokens**.

PROJECT

The key of the Project containing the repositories.

2.3.5 GitHub Settings

USER

The user ID on the GitHub server.

SET NEW TOKEN

A valid Personal Access Token with rights to repositories, webhooks, and pull requests. In GitHub, generate this via **User Settings → Settings → Developer Settings → Personal Access Tokens**.

ORGANIZATION

The name of the GitHub organization that the repositories belong to, if applicable. Leave blank if the repositories are not under an organization.

2.3.6 GitLab Settings

USER

The user ID on the GitLab server.

SET NEW TOKEN

A valid Personal Access Token with rights to repositories, webhooks, and pull requests. In GitLab, generate this via **User Settings → Preferences → Access Tokens**. Set the token scope to `api`.

2.4 Git Repositories

A Git Repository is the definition of a Git repo on another server that MDCMS can connect to and clone into the IFS under `/MDCMS/EXTREF/<repo-id>/tree`.

In MDOpen: **Settings** → **DevOps Settings** → **Git Repositories**

2.4.1 Repository Settings

REPOSITORY ID

A unique identifier of up to 10 characters for this repository.

URL

The network address (including `http(s)://` or `git@`) for the repository. The correct URL can usually be copied from the Git server administration screen. If the URL is in HTTP format and contains `//user@host`, remove the `user@` portion before saving.

GIT TYPE

The type of Git server hosting the repository. MDCMS has a specific webhook defined for Azure DevOps, Bitbucket, GitHub, and GitLab, enabling direct CI triggering. Selecting **Other** requires a pipeline tool such as Jenkins to act as an intermediary, which then triggers CI via the MDCMS API endpoint `/git/checkout`. You may also choose **Other** for a supported server type if you prefer MDCMS not to be involved until later in the DevOps process and want a master pipeline tool to orchestrate CI instead.

DESCRIPTION

A human-readable description of the repository.

CREDENTIAL ID

The ID of the credentials used to access the repository. Required unless the repository allows anonymous access. Use content-assist to select from defined credentials.

API SERVER ID

The ID of an API Server tied to this repository. Required to enable simplified webhook registration and automated commit and Pull Request activities. Use content-assist to select from defined API servers.

MDGIT CLONE 1 JOB

The primary MDGIT service job number that should be used to perform Git activities for the repository. The number may not be higher than the maximum defined for the MDGIT service. It is useful to have multiple MDGIT jobs if MDCMS connects to multiple repositories.

MDGIT CLONE 2 JOB

The secondary MDGIT service job number that should be used to perform Git activities for the repository. This is used in conjunction with the Clone 1 Job to provide high availability for Git activities when multiple branches for the same repository are involved. The number may not be higher than the maximum defined for the MDGIT service.

MDGIT CLONE 3 JOB

The tertiary MDGIT service job number that should be used to perform Git activities for the repository. The number may not be higher than the maximum defined for the MDGIT service.

URL VALUE IN WEBHOOK

When the MDCMS webhook is triggered after a commit push, MDCMS maps the URL in the webhook request body to a repository ID. If the URL in the request body differs from the URL used by MDCMS to clone or sync with the origin, enter the

webhook URL value here. The correct value can be obtained from the Git Webhook Deliveries entry after the webhook has been triggered at least once.

MAIN BRANCH

The name of the primary branch for the repository. Typically `main` or `master`, though it may differ based on team conventions. This branch is used for cross-referencing and is the default branch for other operations.

2.4.2 After Saving

After saving the repository definition, the MDGit service on the IBM i will attempt to connect to the Git server using the specified credentials.

If the connection fails, click the **Connection Logs** icon for the Repository ID and open today's log for the correct MDGIT job number to review the exception details. If no log is available, the MDGIT service job did not auto-start — review the MDGIT service job settings such as job queue and auto-start configuration.

If the connection succeeds, MDCMS will automatically clone the main branch. A re-clone can be triggered at any time by selecting **Re-Clone Main Branch** for the Repository ID.

2.5 Continuous Integration Definitions

Once a repository is defined, the repository — or specific paths within it — can be configured for CI. Multiple CI definitions can be created for the same repository when more granular control is needed.

A CI definition has the following main purposes: - Define which commits trigger CI processing in MDCMS, based on branch and path filters. - Define how commits are processed, including what gets requested, where it goes, and how it is assigned. - Define Project/Task/Subtask assignment rules based on branch naming conventions or commit message parsing. This is required even when MDCMS requests are not triggered from the push of a commit, but rather from a manually initiated request in the local workspace.

In MDOpen: **Settings** → **DevOps Settings** → **Continuous Integrations**

2.5.1 Branch and Path

REPO ID

The ID of the repository this CI definition applies to. Use content-assist to select from defined Git repositories.

BRANCH OPT

Defines which branch within the repository triggers CI processing in MDCMS.

- **Main Branch** — CI is triggered when a commit is pushed to the branch designated as Main Branch in the repository definition.
- **Not Main Branch** — CI is triggered when a commit is pushed to any branch except the Main Branch.
- **Branch Naming Pattern** — CI is triggered when a commit is pushed to any branch whose name matches the specified wildcard pattern in the Branch parameter. For example, `release-*` matches all branches starting with `release-`.
- **Specific Branch** — CI is triggered only when a commit is pushed to the branch named in the Specific Branch parameter.

BRANCH

Required when Branch Opt is set to **Branch Naming Pattern** or **Specific Branch**. Enter the branch pattern, branch name or use content-assist to select from a list.

PATH

Restricts CI to files committed within a specific directory tree. If left blank, a file in any path committed to the repository will be considered for creating an object request in MDCMS (subject to attribute mapping exclusions). To monitor multiple distinct paths for the same repository, create a separate CI definition for each path. If the repository has already been cloned, content-assist can be used to browse and select the path.

FILE NAMING PATTERN

A file name filter applied within the configured path. If blank, all files in the path are considered (subject to attribute mapping exclusions). Use wildcard patterns to restrict by type, for example `*.jar` or by name, for example `MyFile*` or both.

2.5.2 Target Application and Level

APPLICATION

The target MDCMS application that pushed objects will be assigned to.

- ***APPL** — The application is determined by the Application field on the task that the object request is assigned to.
- ***TSKT** — The application and level are determined by the [Task Type mapping definition](#).
- **Specific application** — Enter the ID of a defined MDCMS application, or select using content-assist.

LEVEL

The target MDCMS application level that pushed objects will be deployed to. The level must allow checkouts unless **Level is Target of a Merge** is set to true.

If Application is set to `*APPL` and a specific level number is defined here, the Level field on the associated task is used instead. A value of 0 instructs MDCMS to use the task's level number when it is greater than 0; otherwise MDCMS retrieves the lowest level number that allows checkouts for the application. When **Level is Target of a Merge** is true, a specific level number must be provided.

LEVEL IS TARGET OF A MERGE

Set to true when the target level does not allow checkouts — such as a QA or production level. This situation typically occurs when a Pull Request has been completed, merging changes into a higher-environment branch.

USER FOR MERGE REQUESTS

Determines which user is recorded as the requester for object requests generated by a merge.

- **Requester on Source Branch** — The user who pushed commits on the source branch.
- **User that performed Merge** — The user who completed the merge into the target branch.

2.5.3 What to Request

CHECKOUT

Controls how MDCMS determines which files to retrieve from the repository.

- **Diffs** — Push only the differences (additions, changes, and deletions) by comparing the working tree before and after the commit.
- **Contents** — Push the complete contents of the configured path.

REQUEST IFS/REMOTE OBJECTS

Set to true when the path contains directories and files that will be deployed to the IFS or to a remote Linux or Windows server.

REQUEST SOURCE FOR IBMI OBJECTS

Set to true when the path contains source files for system objects on the IBM i. Source files may target source members or IFS source files on the partition. The same CI definition can handle both IFS/Remote Objects and Source for IBM i Objects, provided relative paths and file types map correctly.

To disable the automatic triggering of a CI definition after the push of a commit, uncheck both Request parameters.

DEFAULT MD ATTRIBUTE

The MDCMS attribute assigned to pushed objects when no match is found in the [Attribute Mapping Table](#). For IFS/Remote Objects, use an attribute of type `*IFS`, `*PIPE`, or `*REMOTE`. For source objects, this serves as a final fallback when mapping, history, and MDXREF all fail to identify an attribute.

DEFAULT USER

The default MDCMS user assigned to object requests when the committing Git user is not found in the [User Mapping Table](#).

RELATIVE PATH

Defines how the relative path is calculated for object requests of MDCMS object type *IFS or *REMOTE.

- **From Repo Root** — The relative path is the full path from the repository root.
- **From CI Definition Path** — The relative path is only the folders located below the path defined in the Path parameter.
- **From Mapped Attribute Path** — The relative path is the folders below the bottom of the specific repository path for the matched attribute.
- **No Relative Path** — All files are deployed directly to the target folder for the attribute, without any relative path beneath it.

2.5.4 Project and Task Assignment

PROJECT

The project to assign to the object requests. Leave blank if the project will be assigned manually before submitting the RFP. The following special values are also supported:

- ***BRANCH** — The repository branch name is parsed as the reference code. It may be an Azure Workitem, Jira Issue, ServiceNow Task, or an MDCMS Project/Task/Subtask. MDCMS parses the branch name rather than the commit message, and a trailing colon delimiter is not expected. MDCMS will automatically trim any description appearing after the Project-Task.Subtask value when the value is in `*MD` format.

If a constant is used at the beginning of the branch name, a positional value can be appended to `*BRANCH`. For example, if the branch names begin with `release-`, then specify `*BRANCH,9` as the value for the Project parameter.

If the branch naming convention doesn't consistently contain the reference code or `*MD` format, do the following: (1) Create a Custom Project/Task field of type String, up to 160 characters. (2) Add the field to the appropriate Task Types. (3) If using Azure DevOps, Jira, or ServiceNow, map the field to the MDCMS custom field. (4) Enter the branch name into the task field before committing.

- ***AZUR** — The project/task/subtask is derived from an Azure Workitem number placed at the very beginning of the commit message. The workitem must have already been exported into MDCMS.
- ***JIRA** — The project/task/subtask is derived from a Jira Issue Key placed at the very beginning of the commit message. The issue must have already been exported into MDCMS. For example, a commit message of `MDS-3891 - demo of CI` causes MDCMS to look up issue key `MDS-3891` and, if it maps to an MDCMS task or subtask, assign it to the object requests.
- ***KACE** — The project/task/subtask is derived from a KACE Service Desk number placed at the very beginning of the commit message. The workitem must have already been exported into MDCMS.
- ***SNOW** — The project/task/subtask is derived from a ServiceNow Task ID placed at the very beginning of the commit message. The task must have already been exported into MDCMS.
- ***LAST** — Uses the Project/Task/Subtask from the most recently installed object with the same name and attribute. Recommended when the CI definition targets a merge level (i.e., the completion of a Pull Request). Useful when higher MDCMS environments are triggered by pull requests merging into higher branches.
- ***MD** — The project/task/subtask is derived from MDCMS-formatted values placed at the very beginning of the commit message. The expected format is `PROJECT-TASK.SUBTASK:` where the colon is the end delimiter. MDCMS extracts the project ID as everything up to the colon or the final hyphen, the task number as the value between the final hyphen and the colon or final period, and the subtask number as the value between the final period and the colon. Task and subtask numbers are generally optional.

TASK

The task within the project to assign to the object requests. Leave blank when assigning directly to a project, or when the Project field is blank or uses a special value.

SUBTASK

The subtask to assign to the object requests. Leave blank when assigning directly to a project or task, or when the Project field is blank or uses a special value.

2.5.5 Object Request Automation

GENERATE RFP

Controls whether object requests are placed into a new RFP.

- **True** — Object requests are assigned to a new RFP, provided they are not already checked out. If an object request for the same object already exists from the same branch, it is automatically updated to the latest committed version.
- **False** — Object requests are not assigned to an RFP.

RFP DESCRIPTION

The description applied to the generated RFP, when Generate RFP is true.

- ***COMMITMSG** — Use the Git commit message as the RFP description.
- ***TASK** — Use the task description as the RFP description.

AUTO-REQUEST DEPENDENCIES

When requesting source for IBM i objects, enabling this option causes MDCMS to automatically request any dependencies that require recompilation — including those affected by file changes, module changes, or copybook changes. The source used for recompilation must exist in the source migration path, based-on path, or source search template on the target partition.

AUTO-CREATE IN DEV LIB

When requesting source for IBM i objects, enabling this option causes MDCMS to automatically attempt to compile the requested objects and their dependencies into the development library. If the source target is a source member, the stream file is copied to the source file of the same name in the developer's work library.

AUTO-SUBMIT RFP

When true, MDCMS auto-submits the RFP once all objects have been requested, even if the Auto-Submit parameter on the level is set to false.

2.5.6 Pipeline Integration

PIPELINE ATTRIBUTE

An attribute of type `*PIPE`. When specified, an object request is automatically generated for this attribute within the same RFP as the committed objects. This is typically used to trigger build, test, deploy, or approval jobs on pipeline servers. Use content-assist to select from defined pipeline attributes.

2.6 Task Type Mapping for Application and Level during CI

For the same [Continuous Integration](#) definition, the target MDCMS Application and Level can differ by Task Type.

When CI parameter **Application** is set to *TSKT, the application and level will be resolved from the Task Type mapping table.

The Task Type is obtained from the MDCMS Project Task that the Git Branch name or Commit Message refers to.

In MDOpen: **DevOps Settings** → **Git Repositories** → **[select repository]** → **Task Type Mapping**

TASK TYPE

The Task Type value as defined in MDCMS. Use content-assist to select from the list.

APPL

The application code for the target MDCMS application to which objects will be pushed when a branch or commit references a task of the specified Task Type. Use content-assist to select from the list, or enter a specific application ID.

LEVEL

The application level for the target MDCMS application to which objects will be pushed when a branch or commit references a task of the specified Task Type. Use content-assist to select from the list, or enter a specific level number.

2.7 Repository Attribute Mapping

When MDCMS receives committed files, it must determine which MDCMS attribute to assign to each file. Attribute mapping is the mechanism that makes this determination explicit and reliable.

Attribute Mapping is also used to determine the relative path for code when using Git Commit templates to sync changes from IBM i back to the origin repository. This is necessary to ensure that changes are committed to the correct location in the repository, and that the correct file extension is assigned to each file.

It is strongly recommended to define attribute mapping for all Git repositories. It is the fastest-performing mechanism, avoids potential path and file name resolution exceptions, and provides a clear overview of which attributes will be used for each path and file type.

In MDOpen: **Git Repositories** → **[select repository]** → **Attribute Mapping**

2.7.1 How MDCMS Resolves Attributes

For IFS / Remote Objects

1. Match on a path/file name pattern in the Attribute Mapping table for the repository.
2. Fall back to the Default MD Attribute defined on the CI definition.

For Source for IBM i Objects

1. Match on a path/file name pattern in the Attribute Mapping table.
2. If found in the Attribute Mapping Table, check MDCMS Object History for the most recent installation of the file into the same application for the same object type. If found, override the attribute from the Attribute Mapping table with the attribute used for that installation.

When not found in the Attribute Mapping table: 3. Check MDCMS Object History for the most recent installation into the same application and level. 4. Check MDCMS Object History for the most recent installation into the same application, regardless of level. 5. Check MDCMS Object History for any installation of the file, regardless of application or level. 6. In the MDCMS Attributes table, attempt to match on a combination of Folder and Default Source Type, or on a combination of Source Library, Source File, and Default Source Type to determine object type. If found, check for a reusable command for the object name and type; otherwise use the first matching MDCMS attribute. 7. Fall back to the Default MD Attribute defined on the CI definition.

2.7.2 Defining Attribute Mapping for an Individual Attribute

Click the **Add** link in the header to add a mapping entry from scratch, or click the **Copy** icon on an existing row to copy an existing mapping entry and adjust as needed.

Mapping Parameters

APPL

The target MDCMS application this mapping record applies to. A separate mapping record can be defined for each application targeted by CI definitions. Use content-assist to select from existing applications.

PATH

A specific path from the root of the repository, or a wildcard pattern for matching paths. The path value does not include the file name itself. Use content-assist to select from existing paths in the repository.

Defining the specific folder path from the root of the Git repository is strongly recommended for each attribute. This is necessary when using Git Commit templates to sync changes from IBM i back to the origin repository, and when relative paths for object requests depend on the mapped attribute path.

FILE NAME

A specific file name or a wildcard pattern for matching file names within the given path. For example, `*.sql` matches all file names with the extension of sql. Use content-assist to select from existing file names in the repository for the given path.

MD ATTRIBUTE

The MDCMS attribute to assign when a file matches the path and file name criteria. Use content-assist to select from existing attributes.

Use `*NONE` to explicitly exclude any matching files from CI processing entirely.

DESIGNATED GIT COMMIT TEMPLATE PATH FOR ATTRIBUTE

If the same attribute is used for multiple paths during CI processing, setting this flag to true will indicate that this is the path to use when committing from MDCMS back to the git repository.

DEFAULT ATTRIBUTE FOR PATH/FILE NAME

When multiple attributes are mapped for the same path/file name value, MDCMS will use the attribute by default where this flag is set to true. During CI processing, MDCMS Installation History will be checked to see if a different attribute should be used instead for the specific file.

For example, in path `/source/dds` for file name pattern `*.prt`, the default may be to use the attribute named `PRT`. However, certain printer files may need to be compiled with wider margins, and thus use the `PRTWIDE` attribute instead. By setting `PRT` as the default attribute for the path/file name pattern, `PRT` will be used for all files in that path with a `.prt` extension except for those that have previously been installed with the `PRTWIDE` attribute, which will continue to use that attribute.

2.7.3 Bulk Mapping Generation using the Missing Paths Feature

The fastest way to initially populate mappings is to click the **Missing Paths** link in the header of the Repository Attribute Mapping view.

Note: The Missing Paths list depends on current data in the MDXREF database. Ensure that `*GIT` and/or `*IFS` are registered for cross-referencing and are up to date before using this feature.

Appl/Level for Attribute Mapping Parameters

APPL

The target MDCMS application the mapping records will apply to. Use content-assist to select from existing applications.

LEVEL USING LIBRARY/FOLDER NAMES

If the folder structure of the repository is organized in a way that includes the folder/library names and/or source file names, enter the level number of the application where the attribute definitions match the names used in the repository. Use content-assist to select from existing levels.

REPO FOLDER STRUCTURE

Select the option that matches the folder structure of the repository. This will determine how MDCMS parses the path and file name to determine the attribute.

- **none:** the repository does not have a consistent folder structure that can be used to determine attribute mapping, so the full path and file name will be used as-is for matching in the Attribute Mapping table.
- **Source Libraries:** the repository is organized with a folder structure that includes the source library name as the first folder level after the relative path under the root. For example, /source/APP1/ would indicate that files in that folder are associated with source stored in the APP1 library.
- **Source Libraries/Source Files:** the repository is organized with a folder structure that includes both the library name and source file name as the first two folder levels after the relative path under the root. For example, /source/APP1/QRPGLESRC would indicate that files in that folder are associated with source stored in the APP1 within the QRPGLESRC source file, and both values would be used for attribute mapping purposes.
- **Source Files:** the repository is organized with a folder structure that includes the source file name as the first folder level after the relative path under the root. For example, /source/QRPGLESRC would indicate that files in that folder are associated with source stored in the QRPGLESRC source file.
- **IFS Source Folders:** the repository is organized with a folder structure that includes the IFS source folder name as the first folder level after the relative path under the root. For example, /source/application-1/ would indicate that files in that folder are associated with source stored in the application-1 IFS source folder.
- **IFS Object Folders:** the repository is organized with a folder structure that includes the IFS object folder name as the first folder level after the relative path under the root. For example, /dist/application-1/ would indicate that files in that folder are associated with *IFS or *REMOTE objects stored in the application-1 object folder.

REPO RELATIVE PATH TO LIBRARY/FOLDER NAMES

The relative path from the root of the repository to the folder level that contains the library and/or source file names used for attribute mapping. For example, if the repository uses a folder structure of /source/APP1/QRPGLESRC, and the relative path is defined as /source/, then MDCMS will parse the path to determine that APP1 is the library name and QRPGLESRC is the source file name for attribute mapping purposes.

Attribute Mapping - Unmapped Paths

After the **Next** button is clicked, the Unmapped Paths view will display each distinct path and file type in the repository that does not have a corresponding entry in the Attribute Mapping table based on the defined repository folder structure.

For each row where a matched attribute is found, the Select checkbox will be enabled.

If the attribute is not the correct attribute for the path/file type, or is set to "no match", the user can click the MD Attribute value to select a different attribute for the path/file type record.

Once one or more rows are selected, click the **Process Selections** link to process the selections. This will create new entries in the Attribute Mapping table for each selected row, using the matched or user-selected attribute. The new entries will be created with the path and file name pattern and will be assigned to the application defined in the previous step.

2.8 Map Git Users to MDCMS Users

If developers have different user IDs on the Git server than in MDCMS, their IDs can be mapped so that object requests created by the CI process are attributed to the correct MDCMS user.

In MDOpen: **DevOps Settings** → **Git Repositories** → **[select repository]** → **User Mapping**

REPOSITORY USER

The user ID of the account on the Git server. This value is case-sensitive.

MDCMS USER

The user ID of the corresponding user registered in MDCMS (via MDSEC) that requests should be attributed to. Use content-assist to select from the list.

REPO ID

The scope of the mapping.

- ***ANY** — Apply this mapping for any repository that contains the specified Repository User and a specific mapping entry doesn't exist.
- **Specific Repo ID** — Restrict the mapping to a single named repository. Use content-assist to select from the list.

2.9 Git Branch Creation Rules

MDCMS can automatically create and publish task-based branches for a Git repository once a task reaches a specified status. This ensures branches are not created prematurely and eliminates formatting or spelling errors that can occur when developers create their own branches manually.

The following MDOpen paths are available to view/configure branch creation rules: - DevOps Settings → Git Branch Creation Rules - DevOps Settings → Git Repositories → [select repository] → Git Branch Creation Rules - Tasks → [select task] → Git Branches → Git Branch Creation Rules

2.9.1 Settings

PROJECT

A project whose tasks are relevant to a Git repository.

TASK TYPE

The task type used by the project that should trigger branch creation.

REPOSITORY ID

The ID of the Git repository where the branch should be created.

BRANCH NAME

The name of the branch to be created. This can be defined using a combination of static text and placeholders representing task and project information. For example, the format `feat-++PRJTSK++` would create branches named `feat-MYPROJ-123.1` for tasks in the MYPROJ project with task number 123 and subtask number 1.

Press F7 to view available placeholders and to insert them at the current cursor position in the Branch Name field.

Content-Assist can be used to view the list of existing branch names and to select an existing branch name. Branch creation rules with existing branch names will attempt to create a branch with that name, and if a branch with that name already exists on the Git server, the creation will be considered successful and no new branch will be created.

CREATE BRANCH FROM

The existing branch that new task branches should be created from. If left blank or set to *MAIN, new branches are created from the branch designated as the Main Branch for the repository.

AUTOMATIC BRANCH CREATION

- **True** — MDCMS automatically creates and publishes the branch when a task meets the configured status criteria.
- **False** — A user must manually initiate branch creation for each task.

MINIMUM STATUS

The minimum status a task must have reached before its branch is created. If left blank, the branch is created as soon as the task is created.

MAXIMUM STATUS

The maximum status a task may be in for a branch to be created. If left blank, there is no upper status limit.

GENERATE FOR SUBTASKS

- **True** — Subtasks of a task that meets the status criteria will also have branches created for them, following the same Branch Name format but with the subtask number included in the placeholders (e.g., `MYPROJ-123.1` for the first subtask of task `MYPROJ-123`).
- **False** — Only the main task will have a branch created, and no branches will be created for subtasks.

2.9.2 Reviewing and Manually Creating Branches

Use option **Git Branches** on a Task to view the most recent branch creation attempt for each repository that has a rule matching the task's Project and Task Type combination.

Click the **Create and Publish Branch** icon to manually create a branch or to retry a failed attempt.

2.9.3 Check out a Created Branch in the Local Workspace

Once a branch is created and published by MDCMS, developers have the following options to checkout the branch in their local workspace: - perform a fetch within their local clone and then check out the existing branch from the Git server - click the **Checkout in Local Workspace** icon on the Branch for the Task in MDOpen - request an Object in MDOpen (such as from MDXREF), and then click the **Sync with Local Workspace** icon on the object request and then click the **Checkout in Local Workspace** icon

2.10 Webhook Configuration

MDCMS Webhooks can be configured for each integrated Git Repository on the Git Origin server whenever events of the following type occur:

Push - commits are pushed to a branch in the repository - this can trigger Continuous Integration within MDCMS and/or dynamically update external references for system objects.

Pull-Request Created - a pull request is created - this keeps MDCMS aware of Pull Requests that are created.

Pull-Request Updated - a pull request is updated - this keeps MDCMS aware of Pull Requests that are modified, merged or closed.

The http(s) target of the webhook is the same MDCMS http API server that is used for MDOpen. If the origin server resides outside of your network, you will need to allow inbound http traffic from the origin server to the MDCMS API Server on your development partition. The URL format for the webhook is:

```
https://<your-mdcms-host>/<mdcms-instance>/<server-type>/webhook
```

Examples:

- Azure DevOps: `https://devsys.mycompany.com/mdcms/azure/git/webhook`
- Bitbucket Cloud: `https://devsys.mycompany.com/mdcms/bitbucket-cloud/webhook`
- Bitbucket Data Center: `https://devsys.mycompany.com/mdcms/bitbucket/webhook`
- GitHub: `https://devsys.mycompany.com/mdcms/github/webhook`
- GitLab: `https://devsys.mycompany.com/mdcms/gitlab/webhook`

Network Recommendations: - proxy-forward the inbound traffic for the URL resource to the internal API Server URL if the API Server is not directly accessible from the origin server - allowlist the origin server IP addresses on the firewall of the API Server host, if possible, to restrict access to only the origin server(s)

2.10.1 Register Webhooks from MDOpen

If the repository has an API Server ID configured, webhook registration can be done directly from MDOpen for Azure DevOps, Bitbucket, GitHub, and GitLab:

1. In MDOpen, open the **Git Repositories** view.
2. Select option **Webhooks** for the relevant Repository definition.
3. A row will be displayed for each supported Event Type based on the Git Server Type of the repository. If the row already displays a URL, date, time and user, then MDOpen has previously registered the webhook for that event type. If the URL is blank, then the webhook has not been registered for that event type.
4. Click on an Event Type to create or delete/recreate the webhook for that event type. If the registration was successful, the URL, date, time and user will appear for the row. If not successful, click the **View Logs** button to view the REST API log for the API Server to troubleshoot the error. The most common errors are related to network connectivity or permissions of the user account on the Git server.

If routing through a proxy, the hostname and/or port of the webhook URL will likely need to be edited on the Git server after registration from MDOpen, to ensure the Git server can reach the webhook URL.

2.10.2 Manual Webhook Registration

Use the steps below if an API Server ID is not defined for the repository or if the user's credentials isn't scoped to include webhook management.

Azure DevOps

1. Log into Azure DevOps with a user who has administrative rights for the project.
2. Go to **Project Settings** for the project.
3. Click **Service hooks**.
4. Click **+** to add a subscription.
5. Select **Web Hooks** from the list and click **Next**.
6. Select the **Code pushed** trigger event.
7. Select the specific or **(Any)** repository, branch, and group member, then click **Next**.
8. Enter the MDCMS webhook URL (using the `/azure/git/webhook` path).
9. Leave optional settings at their default values.
10. Click **Finish**.
11. Repeat steps 4-10 for the **Pull request created** and **Pull request updated** trigger events.

Bitbucket

1. Log into Bitbucket with a user who has administrative rights.
2. Go to **Settings** for the repository.
3. Expand Workflow in the Repository Settings menu and click **Webhooks**.
4. Click **Add webhook**.
5. Enter a descriptive title (e.g., `MDCMS`).
6. Enter the MDCMS webhook URL (using the `/bitbucket/webhook` path).
7. Enable **Request history collection** to assist with troubleshooting during initial setup.
8. Ensure **Repository push, Pull request Created, Pull request Merged** and **Pull request Declined** is selected under Triggers.
9. Click **Save**.

GitHub

1. Log into GitHub with a user who has administrative rights.
2. Click on the **Settings** tab for the repository.
3. Click **Webhooks** under Code and automation in the Settings menu.
4. Click **Add webhook**.
5. Enter the MDCMS webhook URL (using the `/github/webhook` path) as the Payload URL.
6. Select the radio button **Let me select individual events**.
7. Select **Pushes** and **Pull requests**.
8. Click **Add webhook**.

GitLab

1. Log into GitLab with a user who has administrative rights.
2. For the given project, hover over **Settings** in the sidebar and click **Webhooks**.

3. Enter the MDCMS webhook URL (using the `/gitlab/webhook` path).
4. Ensure **Push events** and **Merge request events** is selected under Triggers.
5. Click **Add webhook**.

3. Commits

3.1 Git Commits

In MDOpen: **Settings** → **DevOps Settings** → **Git Commits**

This view displays the commits that have been made to the Git repository by MDCMS during the RFP deployment process.

3.1.1 Header Options

MDGIT LOGS

Open the MDOpen IFS content viewer for the logs of the MDGIT Service jobs.

3.1.2 Row Options

OBJECTS

The list of files that were considered for staging for the commit during the transaction, along with the option (modify or delete) and the status for the file. This is useful for troubleshooting and for understanding what changes were made to the Git repository as part of the RFP deployment process.

RETRY

If a commit failed, this option will attempt to retry the commit. This is useful for situations where a commit may have failed due to a transient issue, such as a network error or a temporary issue with the Git repository.

3.2 Commit Templates

In MDOpen: **Settings** → **Template Settings** → **Git Commit**

Commit Templates are Attribute Templates that are used to automate the commit of changes to source code or *IFS files after an MDCMS RFP installation is complete.

When an object request in the RFP uses an MDCMS Attribute that is assigned to a commit template, MDCMS will consider the code for staging and committing to the Git Repository defined for the template.

Git Commit Templates are useful to automatically replicate source changes to a Git repository, even when the development team otherwise doesn't use Git.

Git Commit Templates are also useful if further edits to the code occur directly on the IBM i prior to running the RFP, in order to ensure that those changes are also captured and committed back to the Git repository.

During the Post-Installation phase of the RFP, MDCMS will create an MDGIT transaction record for each distinct combination of: -
 Git Commit Template - Git Repository
 - Target Branch - Object Requester

For each transaction record, the MDGIT Service job named MDGITC will then perform the following steps: 1. Create the target branch if it does not exist.

1. Pull the branch from the origin Git repository.
2. Find the relative path and file naming/extension in the Repository based on the [Attribute Mapping definitions](#).
3. If the file already exists in the repository, generate an SHA-1 signature for the RFP code and an SHA-1 signature for the code in the repository and ignore the file if the signatures match. The signature generation ignores whitespace and line-ending differences, so that files will not be committed if there are no substantive code changes even if there are formatting differences.
4. Stage the files for commit to the branch.
5. Commit the staged changes to the local Git repository in the IFS with the following properties:
 6. a commit message based on the template
 7. the author of the commit set to the Object Requester, and the email set to the email of the Object Requester that is registered in MDSEC
 8. the committer name/email based on the template on the Git server
 9. GPG signing, if enabled for the template
10. Push the Commit to the Git Repository on the Origin Server

3.2.1 Git Commit Template Parameters

TEMPLATE ID

A unique identifier of up to 10 characters for this template.

DESCRIPTION

A human-readable description of the template.

REPO ID

The Git repository this template applies to. Use content-assist to select from existing repositories.

COMMIT TO CI BRANCH

- **True** — Commit changes to the same branch that the CI process or local workspace request used to generate the Object Request. If the object request did not originate from the CI process, or through a request from a clone in the local workspace, then the Branch Name parameter value will be used instead.
- **False** — Commit changes to the Branch Name parameter value, regardless of the branch used during request of the Object.

BRANCH NAME

The name of the branch to commit to. This can be defined using a combination of static text and placeholders representing task and project information. For example, the format `feat-++PRJTSK++` would create branches named `feat-MYPROJ-123.1` for tasks in the MYPROJ project with task number 123 and subtask number 1.

Press F7 to view available placeholders and to insert them at the current cursor position in the Branch Name field.

Content-Assist can be used to view the list of existing branch names and to select an existing branch name.

CRT FROM BRANCH

The existing branch that new branches should be created from if the branch defined in Branch Name does not already exist. If left blank or set to *MAIN, new branches are created from the branch designated as the Main Branch for the repository.

Press F7 to view available placeholders and to insert them at the current cursor position in the Crt from Branch field.

Content-Assist can be used to view the list of existing branch names and to select an existing branch name.

COMMIT MESSAGE

The commit message to use for commits made with this template. This can be defined using a combination of static text and placeholders representing RFP, task and project information. For example, the format

`Committing changes for ++PRJTSK++ using RFP ++RFPNBR++` would create commit messages like

`Committing changes for MYPROJ-123.1 using RFP 12345` for a commit related to a task in the MYPROJ project with task number 123 and subtask number 1, and an RFP number of 12345.

COMMITTER NAME

The name of the committer to use for commits made with this template.

***USER** — the committer name/email will be the same as the author (the Object Requester).

COMMITTER EMAIL

The email of the committer to use for commits made with this template. Ignored if Committer Name is set to *USER.

UPPER-CASE GIT FILE NAME FOR NEW SOURCE MEMBERS

- **True** — new source members created in the Git repository will have upper-case file names.
- **False** — new source members created in the Git repository will have lower-case file names.

UPPER-CASE GIT FILE TYPE FOR NEW SOURCE MEMBERS

- **True** — new source members created in the Git repository will have upper-case file types (extensions).
- **False** — new source members created in the Git repository will have lower-case file types (extensions).

END OF LINE CHARACTERS

The type of end of line characters to use for files committed to the Git repository that are converted from source members. -

CR — Carriage Return only (classic Mac-style) - **CRLF** — Carriage Return + Line Feed (Windows-style) - **LF** — Line Feed only (Unix-style) (recommended) - **LFCR** — Line Feed + Carriage Return (uncommon)

AUTO-LINK REQUESTS TO LOCAL WORKSPACE

- **True** — If the Object Request originated directly within MDCMS or MDOpen, such as from the MDXREF Objects view, check if a local clone of the Repository is in the workspace and link the request to that workspace so that the developer can edit the code locally to take advantage of AI and other IDE features. See [Local Workspace Mapping](#) for more information.

- **False** — Object Requests will not be automatically linked to the repository in the local workspace.

GPG SIGNING

- **True** — Commits made with this template will be GPG-signed using the MD Service User's GPG key. The Service User must have a GPG key configured on the IBM i for this to work properly. Instructions for configuring a GPG key for the Service User can be found in the [Configure GPG Signing](#) documentation.
- **False** — Commits made with this template will not be GPG-signed.

GPG KEY

The GPG key to use for signing commits if GPG Signing is set to True. The key is the 40-character hexadecimal key ID.

NEW GPG PASSPHRASE

The passphrase for the GPG key if GPG Signing is set to True. This is only needed if the GPG key requires a passphrase and the passphrase has not already been stored for the template. The passphrase is stored securely in the MDSEC database and is not retrievable after saving.

*NONE — No passphrase is used for the GPG key

3.2.2 Attributes Assigned to Template

Header Options

ADJUST FILTER

Click on the Filter icon to filter the list by Application, Level, Object Type, MDCMS Attribute, Library or assigned Template.

NOT ASSIGNED TO THIS

Limit the list of attributes to those that are not assigned to this template.

ASSIGNED TO ANY

Limit the list of attributes to those that are assigned to any template, including this one.

ONLY ASSIGNED TO THIS

Limit the list of attributes to those that are assigned to this template

ALL ATTRIBUTES

View all attributes regardless of assignment.

Row Options

ASSIGN TO TEMPLATE

Assign the attribute to the template. This option will only appear if the attribute is not already assigned to a template, and an [Attribute Mapping definition](#) is found for the attribute. The mapping definition must contain a specific path rather than a naming pattern for the path.

REMOVE FROM TEMPLATE

Remove the attribute assignment from a template. This option will only appear if the attribute is currently assigned to a template.

MAP ATTRIBUTE IN REPOSITORY

Create an Attribute Mapping definition that maps the attribute to a specific file path in the Git repository. This option will only appear if the attribute is not already assigned to a template, and no [Attribute Mapping definition](#) is found for the attribute.

SET AS DEFAULT PATH FOR NEW MAPPED ATTRIBUTES

Set the Mapped Path as the default path for subsequent Attribute Mapping creations. This can help save a lot of time when several attributes are missing Mapping definitions. First click on this option on a row that has a similar mapped path. Then, click the **Map Attribute in Repository** option on a row where the definition doesn't exist yet.

3.3 Configure GPG Signing for Git Commits

3.3.1 Install the Open Source Package gnupg2

The open source package gnupg2 is required for GPG signing of Git commits. This must be installed on the IBM i. The easiest method to install it is by using the Open Source Package Management tool in IBM i ACS. Full instructions for Open Source Package Management is found here: <https://www.ibm.com/support/pages/getting-started-open-source-package-management-ibm-i-acis>.

3.3.2 Generate a GPG Key for the MD Service User

Start a SSH Bash Terminal Session on the IBM i using the MD Service User Credentials

The MD Service User is the user profile that the MDGIT Service job runs under. This user is defined in the MDCMS System Settings parameter MD Service User Profile.

Before connecting, ensure that IFS directory /home/ exists on the IBM i, where is the name of the MD Service User. If it does not exist, create it using the 5250 command `CRTDIR DIR('/home/<service_user>')`.

On your PC, start a bash terminal or command window and use the `ssh <service_user>@<ibm_i_hostname>` command to connect to the IBM i using the MD Service User's credentials. Replace

`<service_user>` with the actual service user name and `<ibm_i_hostname>` with the hostname of the IBM i.

You will need to know the password for the service user to sign in, unless SSH keys are already configured and known for that user.

Set the Path Environment Variable

Use command `PATH=/QOpenSys/pkgs/bin:$PATH` followed by command `export PATH` to ensure that the gnupg2 executable can be found in the terminal session.

Generate the GPG Key

Use the command `gpg --full-generate-key` to generate a GPG key for the MD Service User. When prompted, select the following options: - Key type: RSA and RSA (option 1) - Key size: 4096 bits - Key expiration: No expiration (option 0), unless you have specific requirements for expiring keys
 - Real Name: Enter a user ID name that matches the Committer Name parameter for the Git Commit Template. - Email Address: Enter the email address associated with the Committer Name. This must match the Committer Email parameter for the Git Commit Template. - Comment: This is optional and can be left blank or used to provide additional information about the key. - Type O to confirm the information and continue to the next step or press one of the other options to edit the information you entered. - Passphrase: Enter a secure passphrase to protect the GPG key. You will need to store this passphrase in the New GPG Passphrase parameter for the Git Commit Template. - Repeat Passphrase: Enter the same passphrase again to confirm. - After completing these steps, the GPG key will be generated if enough movement is occurring in the terminal to generate entropy for the key generation process. If not, you may need to move your mouse or type random characters in the terminal until the key is generated.

Retrieve the GPG Key ID

After the GPG key is generated, use the command `gpg --list-secret-keys` to retrieve the key ID for the GPG key. The key ID is the 40-character hexadecimal string associated with the GPG key. You will need to enter this key ID in the GPG Key parameter for the Git Commit Template.

Import the GPG Public Key to Git Hosting Service

If you are using a Git hosting service such as GitHub, GitLab or Bitbucket, you will need to import the GPG public key to the hosting service in order for the GPG signatures to be verified and displayed on commits.

Use the command `gpg --armor --export <key_id>` to export the GPG public key in ASCII-armored format, where `<key_id>` is the 40-character hexadecimal key ID for the GPG key. Copy the entire output, including the `-----BEGIN PGP PUBLIC KEY BLOCK-----` and `-----END PGP PUBLIC KEY BLOCK-----` lines.

Then, follow the instructions for your specific Git hosting service to add a new GPG key to the account used in the Git Repository Credentials and paste the copied public key into the appropriate field.

4. Pull Requests

4.1 Git Pull Requests

In MDOpen: **Settings** → **DevOps Settings** → **Git Pull Requests**

This view displays the Pull Requests that have been created: - by MDCMS as part of the RFP deployment process - manually from MDOpen - from another process outside of MDCMS

4.1.1 Header Options

CREATE PULL REQUEST

Manually create a Pull Request for a given source branch/target branch combination. This is useful for testing and for situations where a Pull Request needs to be created outside of the RFP deployment process.

4.1.2 Row Options

REPEAT PULL REQUEST

MDGIT will attempt to recreate a Pull Request, using the same values for the source branch and target branch as the original attempt.

CONNECTION LOGS

Open the MDOpen IFS content viewer for the logs of the API Server linked to the Git Repository.

4.2 Pull Request Levels

MDCMS can automatically create a Pull Request (PR) in the Git repository as part of the RFP deployment process. A PR is typically used to merge changes from a development branch to a higher-level branch that triggers deployment to QA or production environments. The Pull Request Levels feature allows you to define the layers of Pull Requests necessary to eventually get a feature merged into the production branch.

The actual merge of a Pull Request is performed on the Git server by an authorized user or users after review is complete. The merge can then trigger higher-level CI processing defined in [Continuous Integrations](#).

In MDOpen: **Settings** → **DevOps Settings** → **Pull Request Levels**

4.2.1 Pull Request Level Parameters

APPL

The MDCMS application of an RFP.

LEVEL

The MDCMS level of an RFP.

REPOSITORY ID

The Repository ID of the Git repository that the Pull Request Level applies to. Use content-assist to select from defined repositories.

If multiple Repositories are used for a given application and level, a Pull Request Level definition must be created for each Repository.

PR TARGET BRANCH

The branch that the Pull Request is targeting. This is typically a higher-level branch than the source branch used for development work. For example, if developers are working in feature branches off of a `release` branch, the PR Target Branch might be `rel-project-task` for a lower-level Pull Request and `main` for the final Pull Request that triggers production deployment.

The target branch name can be a mix of static text and placeholders. Use content-assist to select a specific branch from the list of branches in the repository. For placeholders, position the cursor in the field where the placeholder value should be inserted and then press F7 to select the placeholder from the list.

CRT FROM BRANCH

If the target branch doesn't exist yet, MDCMS will automatically create the branch based on the branch name in this parameter.

The Create from branch name can be a mix of static text and placeholders. Use content-assist to select a specific branch from the list of branches in the repository. For placeholders, position the cursor in the field where the placeholder value should be inserted and then press F7 to select the placeholder from the list.

PR TITLE

The title of the Pull Request. This can be a mix of static text and placeholders. Position the cursor in the field where the placeholder value should be inserted and then press F7 to select the placeholder from the list.

4.2.2 Automated Creation of Pull Requests

When an RFP has completed installation, MDCMS checks if any object requests in the RFP are linked to branches in one or more Git repositories. If so, the Pull Request Levels table is checked to see if a matching definition exists for each distinct application,

level and Repository combination. If found, and a Pull Request isn't already open for the source branch/target branch combination, the MDCMS will attempt to create the pull request.

High-Level Logging of the creation attempt is included in the RFP Deployment log and in the is deployed to a level with a Pull Request Level defined, MDCMS automatically creates a Pull Request on the Git server based on the parameters defined for that level. The Pull Request is created after the RFP is submitted, and the title of the Pull Request includes the RFP number for easy reference.

APPLICATION

The target MDCMS application that pushed objects will be assigned to.

- ***APPL** — The application is determined by the Application field on the task that the object request is assigned to.
- ***TSKT** — The application and level are determined by the [Task Type mapping definition](#).
- **Specific application** — Enter the ID of a defined MDCMS application, or select using content-assist.

LEVEL

The target MDCMS application level that pushed objects will be deployed to. The level must allow checkouts unless **Level is Target of a Merge** is set to true.

If Application is set to `*APPL` and a specific level number is defined here, the Level field on the associated task is used instead. A value of 0 instructs MDCMS to use the task's level number when it is greater than 0; otherwise MDCMS retrieves the lowest level number that allows checkouts for the application. When **Level is Target of a Merge** is true, a specific level number must be provided.

LEVEL IS TARGET OF A MERGE

Set to true when the target level does not allow checkouts — such as a QA or production level. This situation typically occurs when a Pull Request has been completed, merging changes into a higher-environment branch.

USER FOR MERGE REQUESTS

Determines which user is recorded as the requester for object requests generated by a merge.

- **Requester on Source Branch** — The user who pushed commits on the source branch.
- **User that performed Merge** — The user who completed the merge into the target branch.

4.2.3 What to Request

CHECKOUT

Controls how MDCMS determines which files to retrieve from the repository.

- **Diffs** — Push only the differences (additions, changes, and deletions) by comparing the working tree before and after the commit.
- **Contents** — Push the complete contents of the configured path.

REQUEST IFS/REMOTE OBJECTS

Set to true when the path contains directories and files that will be deployed to the IFS or to a remote Linux or Windows server.

REQUEST SOURCE FOR IBMI OBJECTS

Set to true when the path contains source files for system objects on the IBM i. Source files may target source members or IFS source files on the partition. The same CI definition can handle both IFS/Remote Objects and Source for IBM i Objects, provided relative paths and file types map correctly.

To disable the automatic triggering of a CI definition after the push of a commit, uncheck both Request parameters.

DEFAULT MD ATTRIBUTE

The MDCMS attribute assigned to pushed objects when no match is found in the [Attribute Mapping Table](#). For IFS/Remote Objects, use an attribute of type `*IFS`, `*PIPE`, or `*REMOTE`. For source objects, this serves as a final fallback when mapping, history, and MDXREF all fail to identify an attribute.

DEFAULT USER

The default MDCMS user assigned to object requests when the committing Git user is not found in the [User Mapping Table](#).

RELATIVE PATH

Defines how the relative path is calculated for object requests of MDCMS object type `*IFS` or `*REMOTE`.

- **From Repo Root** — The relative path is the full path from the repository root.
- **From CI Definition Path** — The relative path is only the folders located below the path defined in the Path parameter.
- **From Mapped Attribute Path** — The relative path is the folders below the bottom of the specific repository path for the matched attribute.
- **No Relative Path** — All files are deployed directly to the target folder for the attribute, without any relative path beneath it.

4.2.4 Project and Task Assignment

PROJECT

The project to assign to the object requests. Leave blank if the project will be assigned manually before submitting the RFP. The following special values are also supported:

- ***BRANCH** — The repository branch name is parsed as the reference code. It may be an Azure Workitem, Jira Issue, ServiceNow Task, or an MDCMS Project/Task/Subtask. MDCMS parses the branch name rather than the commit message, and a trailing colon delimiter is not expected. MDCMS will automatically trim any description appearing after the Project-Task.Subtask value when the value is in `*MD` format.

If a constant is used at the beginning of the branch name, a positional value can be appended to `*BRANCH`. For example, if the branch names begin with `release-`, then specify `*BRANCH,9` as the value for the Project parameter.

If the branch naming convention doesn't consistently contain the reference code or `*MD` format, do the following: (1) Create a Custom Project/Task field of type String, up to 160 characters. (2) Add the field to the appropriate Task Types. (3) If using Azure

DevOps, Jira, or ServiceNow, map the field to the MDCMS custom field. (4) Enter the branch name into the task field before committing.

- ***AZUR** — The project/task/subtask is derived from an Azure Workitem number placed at the very beginning of the commit message. The workitem must have already been exported into MDCMS.
- ***JIRA** — The project/task/subtask is derived from a Jira Issue Key placed at the very beginning of the commit message. The issue must have already been exported into MDCMS. For example, a commit message of `MDS-3891 - demo of CI` causes MDCMS to look up issue key `MDS-3891` and, if it maps to an MDCMS task or subtask, assign it to the object requests.
- ***KACE** — The project/task/subtask is derived from a KACE Service Desk number placed at the very beginning of the commit message. The workitem must have already been exported into MDCMS.
- ***SNOW** — The project/task/subtask is derived from a ServiceNow Task ID placed at the very beginning of the commit message. The task must have already been exported into MDCMS.
- ***LAST** — Uses the Project/Task/Subtask from the most recently installed object with the same name and attribute. Recommended when the CI definition targets a merge level (i.e., the completion of a Pull Request). Useful when higher MDCMS environments are triggered by pull requests merging into higher branches.
- ***MD** — The project/task/subtask is derived from MDCMS-formatted values placed at the very beginning of the commit message. The expected format is `PROJECT-TASK.SUBTASK:` where the colon is the end delimiter. MDCMS extracts the project ID as everything up to the colon or the final hyphen, the task number as the value between the final hyphen and the colon or final period, and the subtask number as the value between the final period and the colon. Task and subtask numbers are generally optional.

TASK

The task within the project to assign to the object requests. Leave blank when assigning directly to a project, or when the Project field is blank or uses a special value.

SUBTASK

The subtask to assign to the object requests. Leave blank when assigning directly to a project or task, or when the Project field is blank or uses a special value.

4.2.5 Object Request Automation

GENERATE RFP

Controls whether object requests are placed into a new RFP.

- **True** — Object requests are assigned to a new RFP, provided they are not already checked out. If an object request for the same object already exists from the same branch, it is automatically updated to the latest committed version.
- **False** — Object requests are not assigned to an RFP.

RFP DESCRIPTION

The description applied to the generated RFP, when Generate RFP is true.

- ***COMMITMSG** — Use the Git commit message as the RFP description.
- ***TASK** — Use the task description as the RFP description.

AUTO-REQUEST DEPENDENCIES

When requesting source for IBM i objects, enabling this option causes MDCMS to automatically request any dependencies that require recompilation — including those affected by file changes, module changes, or copybook changes. The source used for recompilation must exist in the source migration path, based-on path, or source search template on the target partition.

AUTO-CREATE IN DEV LIB

When requesting source for IBM i objects, enabling this option causes MDCMS to automatically attempt to compile the requested objects and their dependencies into the development library. If the source target is a source member, the stream file is copied to the source file of the same name in the developer's work library.

AUTO-SUBMIT RFP

When true, MDCMS auto-submits the RFP once all objects have been requested, even if the Auto-Submit parameter on the level is set to false.

4.2.6 Pipeline Integration

PIPELINE ATTRIBUTE

An attribute of type `*PIPE`. When specified, an object request is automatically generated for this attribute within the same RFP as the committed objects. This is typically used to trigger build, test, deploy, or approval jobs on pipeline servers. Use content-assist to select from defined pipeline attributes.

5. Local Workspace

5.1 Local Workspace Mapping

Local Workspace Mapping is a feature that allows developers to link Object Requests in MDOpen to a local workspace on their machine that contains a clone of the Git repository. This enables developers to edit code locally using their preferred IDE and take advantage of features such as AI code assistance, while still maintaining the build, workflow and traceability benefits of using Object Requests in MDCMS.

The location of the local copy can vary by developer and machine, so Mapping entries are to be defined.

5.1.1 Ways to Create a Local Workspace Mapping Entry

1. In MDOpen: **Settings** → **DevOps Settings** → **Local Workspace Mapping**

This shows a list of all existing Mapping entries and allows users to create new entries or edit existing ones. When creating or editing an entry, the user must specify the Repository ID and the full path to the root folder of the local clone in the Local Workspace.

1. Within the Explorer extension Panel of the IDE: Right-click on the root folder of the local clone and select option **MDOpen** → **Set as Repository Root**. This will open the Add Local Workspace Mapping form with the Repo Root Path correctly pre-populated. Specify the Repository ID to map the path to and click Save.

5.1.2 Usage of Mapping

When an MDOpen connection to MDCMS is established, the mapping entries for the user are checked for existence in the workspace. Each found path is then registered in the MDOpen session.

note: If a new mapping entry is created, the user must then disconnect and reconnect to MDOpen to register the new path for the session.

Using Local Workspace Mapping enables the following advantages for the developer: - click the **Checkout in Local Workspace** icon for a branch on a Task - dynamic comparison of code in the workspace vs. the code on the IBM i - easy upload/download of changes between the workspace and the IBM i - request folders and files directly from the explorer extension panel - request files directly from the staged files list in the Source Control panel

6. Export & Integration

6.1 MDSRC2IFS - Convert Source Members to IFS

6.1.1 Overview

The Convert Source Members to IFS (MDSRC2IFS) command provides the ability to copy some or all members in a source file to various IFS folders depending on the source type.

6.1.2 Restrictions

- The command must be run interactively from a command shell within MDCMS
- The user performing the command must have authority to edit promotion levels (MDSEC code md/4)

6.1.3 Parameters

LIB - Source Library

Description: The Library containing the source file

Required: Yes

Values: - character-value - The name of an existing library in the current namespace

FILE - Source File

Description: The source file containing members to copy to the IFS

Required: Yes

Values: - character-value - The name of an existing source file in the library (LIB)

PATH - Parent Path

Description: The full path name from the IFS root for the parent folder that the members will either be directly copied to or that will hold subfolders for the source.

Required: Yes

Values: - character-value - The full path starting with / to the parent folder for the IFS source

CCSID - CCSID of IFS file

Description: Specifies the CCSID that should be applied when copying the source member to the IFS file.

Values: - *CCSID (default) - A CCSID in the Microsoft Windows encoding scheme - character-value - Any CCSID value that is accepted on the CPYTOSTMF STMFCCSID parameter

AUTH - IFS Authority Template

Description: Specifies the MDCMS Object Authority template of type *IFS that contains the authority definitions to apply to each generated IFS file and each created folder.

Values: - *DFTIFS (default) - The template of name *DFTIFS will be applied - character-value - An existing Object Authority Template of type *IFS

REPL - Replace Existing

Description: Specifies if an existing IFS file should be overridden.

Values: - *NO (default) - If the IFS file already exists, it won't be replaced by a new copy of the source member - *YES - The member will be copied to the IFS file, even if it already exists

6.1.4 The MDSRC2IFS Confirmation Screen

Once Enter is pressed on the command prompt, a confirmation screen will be displayed that lists each source type in the source file along with the number of members of that type.

Opt - Enter '1' to copy all source members of the given Member Type.

IFS File Type - The IFS file type that will be applied to the copied members of the given Member Type. The value is case-sensitive. The name of each IFS file will be the same as the source member name in upper-case.

Subfolder - an optional subfolder name to place the IFS file in under the parent path (PATH parameter). If left blank, the IFS file will be placed directly under the parent path. If provided, the subfolder will be created if it does not already exist. The subfolder name is case-sensitive. The subfolder can be multiple levels deep if required (e.g. /RPG/modules).

6.2 Export IFS Content to Git

If your IBM i source code is not yet in Git, MDOpen provides the option to export IFS folder structures and files to Git, including committing the export locally and publishing on the origin server.

6.2.1 Before you Begin

The [Git Credentials](#) and [Git Repository](#) must already be defined in MDOpen and MDGIT must be able to connect to the Repository.

If your source code is currently in source physical files on the IBM i, you must first use the [MDSRC2IFS](#) command to copy the code to the IFS.

6.2.2 Export to Git

In MDOpen: **MDXREF → IFS Contents**

1. Navigate to the IFS folder containing the content you wish to export
2. Multi-Select each folder or file that should be exported and then select option **Export to Git**. A selected folder will be part of the path in the Git repo, but any parent folders will not be automatically part of the repo path.

The Export Dialog

The option will open a dialog with the following parameters:

INCLUDE SUBFOLDERS

- **True** — Include all subfolders and their contents in the export.
- **False** - only files directly within the selected folder(s) will be included in the export. Note - empty folders cannot be exported to Git, as this isn't permitted by the Git commit process.

FILE NAME PATTERN

Optionally filter the export to only include files with names that match a specific pattern. For example, `*.sql` would only include files with the `.sql` extension. This parameter supports the standard wildcard character `*`. If left blank, all files will be included regardless of name.

GIT REPOSITORY

The Git Repository to export to. This must already be defined in MDOpen under Git Repositories. Use content-assist to select from a list.

BRANCH

The name of the branch to export to. Use content-assist to select from a list of existing branches on the origin server, or enter the name of a new branch to create. If the branch does not already exist on the origin server, it will be created during the export process.

It is recommended to export to a new branch and then validate the contents. If anything is not as intended, you can then simply delete the branch on the origin server (which triggers the deletion of the branch in the IFS) and then try again. If correct, then create a pull-request to merge the exported code into your main branch.

CREATE FROM BRANCH

The existing branch to create the new branch from. This is only relevant if you are exporting to a new branch that does not already exist on the origin server. Use content-assist to select from a list of existing branches on the origin server.

TARGET PATH IN REPOSITORY

The path within the Git repository to export to. For example, if you enter `src/sql` then the content will be exported to a folder named `sql` within a folder named `src` in the Git repository. If the specified path does not already exist in the repository, it will be created during the export process. If left blank, the content will be exported to the root of the repository.

COMMIT MESSAGE

The commit message to use for the Git commit that is created during the export process. This should be a meaningful message that describes the content being exported, such as "Initial export of source code from IFS".

COMMIT TEMPLATE FOR COMMITTER / GPG KEY

Optionally specify a [Commit Template](#) to determine the committer name/email and GPG key to use for signing the commit created during the export process. If left blank, the committer will be set to the current user's [mapped user in Git](#) and no GPG key will be used.

The Export Process

When you click the **Confirm** button to start the export, the MDGIT service jobs will perform the export. Once complete, if successful, the info message "Export to Git completed" will be displayed. If there are any errors, an actionable message will be displayed with ability view the MDGIT IFS logs. The MDGIT IFS logs can also be viewed during the process to follow the progress of the export.