MCRBuddy requirements

1. Context

1.1. Background

1.1.1. Miles and the Miles Product Team

Miles is a Contract Management System for Financial Services. It covers most of FS Business Processes such as Contract Preparation, Document Review, Contract Booking, ... Miles is developed by an external partner called Sofico. Sofico delivers versions of the Miles application to the Miles Product Team, who will eventually make sure this is implemented in the production environment.

The Miles Product Team is a BizDevOps team. The team has:

- Configurators
- Operational resources

The Configuration experts will configure the specific business needs in the Miles Application. They are responsible for the DTAP process of this configuration.

1.1.2. Current processes

Currently, The configuration process is handled by up to 10 individual steps, just to import & export configuration. This presents a lot of possibility for human mistakes/errors. There is no automatic backup-solution for the configuration, so rollbacks are only possible if a manual backup has been made. More detail on the process can be found here (4).

At present, configuration is first configured on the country's CFG environment.

Then the configuration is transferred to the TEST and INT environments using the simple export and simple import functionalities in MMC. This happens configuration object by configuration object.
A solution note is written referring to the changed/added configuration and the relevant JIRA issue and analysis.

When the configuration is tested by business and approved, the configuration XMLs are moved to a directory. There they are picked up by IBM and imported into production when they do a deploy.

1.2. Scope

This document describes the requirements for the development of the 'MCRBuddy'-application, which will be an application the Miles Product Team can use to **effectively en efficiently manage Application Configuration**, for different countries on different environments. Also, rollback of those Configuration commits should be covered by the application. All this:

- Integrated with Miles rest Web Services to manage Configuration Items over all environments
- Integrated with the Agile Toolchain (Bitbucket for repository, Confluence for documentation)
- Integrated in the Miles Application landscape, for ease of access to all necessary API's, NAS,...
- Integrated with an Oracle database (for metadata etc.)

The application needs a User Interface self explanatory enough for experienced IT-users to use (in the document you find some non-binding mockups to clarify the requirements).

The application needs to:

- minimize manual interventions to get config from one environment to others
- be able to cope with a multi country and multi environment landscape (CFG, TST, INT, PROD)
- be able to create & manage a common layer of config objects over all countries
- provide an in-team reviewing mechanism

Config items in Common layer must be alterable if needed in a country specific layer.

The approach will be agile and that of delivery of a Minimal Viable Product in a first phase. Following items are on the backlog, but right now not in scope of the MVP:

- Possibility to open Miles CFG via MCRBuddy, and capture all changes made to Config Objects, to automatically 'load' in to Stage 1 Commit.

- Version control of application code
- Integration with Jira Xray for test Management
- Integration with Confluence for automated Documentation

1.3. Glossary

term	explanation
ATC	Agile Toolchain, the BMW hosted environment of the Atlassian Suite.
MVP	Minimal Viable Product.
Config object	In Miles, there are a lot of possibilities to alter the specific behaviour of certain functionalities, as well as how calculations are done. This is done through objects in Miles like properties, formulas, workflows, lookup tables,... as well as objects containing meta data (e.g. the 'Panel' object in Miles, which can be configured when you have the correct role, contains information about how a certain panel in Miles RIA needs to look, including which fields are present,...). These objects act as the configuration, and can be transferred (individually) from one environment to another (e.g. from the INT environment to the PROD environment once it is all tested). Whenever we mention **config objects** in this document, we are talking about these types of objects (the total list of object types is over 50, so we will not list them all here).
DTAP	Development, Testing, Acceptance & Production
Miles RIA	RIA stands for Rich Internet Application. Miles RIA is the main application for internal users at BMW FS and contains all functionalities internal users will need. Is a web application based on the GWT (Google Web Toolkit) technology. Also contains a lot of screens for config objects like formulas, configuration properties, lookup tables, worksflows,...
Miles Web	Miles Web is a web application for both external (dealers etc.) and internal (department sales) users, in which quotes can be made and calculated.
MMC	Stands for Miles Management Console. Using the same technology as Miles RIA, it allows configurators to manage the configuration not found in Miles RIA. These are usually the more technical config objects, such as operations, policies, metadata on object types itself, GUI settings,... It also offers an export and import functionality to export config objects into XML files and import those same files on a different environment (essentially copying the configuration from one environment to the next).

MCR	The MCR will stand for the Miles Config Repository, the collection of Git repositories that contain the exported config object XML files. For more information, we refer to the "Miles Config Repository" document that goes into deeper detail of the whole MCR concept.

2. Features and functionalities

In the following sections, we'll give a high-level overview of features we need developed to implement our Miles Config Repository concept. For each of these features we'll give a full description and indicate where/how/by who this feature can be developed. For some there will be multiple possibilities, and for each possibility arguments will be given why this possibility is the best idea.

2.1. Managing Git repositories

2.1.1. General

Context

As discussed in the Miles Config Repository concept document, the Miles Config Repository, or MCR, contains out of 3 separate Git repositories, each repository belonging to a certain stage in our development.

What

In order to manage the MCR repositories, the MCRBuddy application will need to:

- Have these repositories checked out in a to the application itself accessible location
  => Git will need to be installed on this location (in order to check out the repositories and still do emergency changes)
- Have the correct libraries to manage the git repository
  => Another possibility might be just manipulating the git repositories through command line execution of git commands, but then the application would of course might also need additional libraries
- Have the correct log in data to manage the Git repositories. This needs to be kept securily, and the admin users of the MCRBuddy application need to be able to change this data if needed in an easy way.

When talking about managing the repositories, we should note that this primarily will entail following functionalities:

- Checking out a certain commit
- Pulling
- Committing & pushing
- Creating/deleting branches

Note that there is no mention of merging and cherry picking, as these are functionalities that will not be needed.

How

Details on how the git repositories will be managed are to be decided by the development team.

There is no preference in technology, but we do have some remarks apart from the above mentioned requirements:

- Everything needs to happen in a secure way.
- The repositories would primarily be managed by MCRBuddy, but in rare cases a technical individual of our team might alter the repositories "manually" using the git command line.
- The repositories will be stored on the BMW BitBucket service.

2.1.2. Multirepository setup

Context

As mentioned in the concept, we are working with multiple stages of development (stage 1, 2 and 3). We also mentioned that each stage will get it's own repository.

However, we need to expand on this a little bit. Sofico, vendor of Miles, works with major releases. 2 times a year, a major version is released, and unlike patches (which come every 2 weeks), these may include changes to configuration objects (e.g. new configuration objects or additional fields). Either way, these major releases would generate XMLs that are incompatible with other versions (and the other way around).

At BMW, as with most Sofico customers, upgrading to a new major version (usually once a year), is usually a small project. There are a few phases:

1. First a copy of the CFG environment is taken and upgraded to the new major version.
   - Configurators will process a whole bunch of release notes and change configuration where needed.
   - New changes have to be configured on both CFG environments, but this shouldn't happen anymore (as there is a feature freeze, so only bugfixes should happen during this period).
2. After processing the release notes on our new CFG environment, the TST environment is upgraded with the config of our new CFG environment.
   - Tests are done on the TST environment
3. After our tests on TST, the INT environment is upgraded with the config of our new CFG environment.
   - Tests by business are done on the INT environment

4. Our PROD and MIR environments are upgraded with the config of our new CFG environment.
    - After a while, the old CFG environment is deactivated.

As you can see, the process is quite intense. If we would translate this to the MCR, this would mean that at some point, we'd need 2 different stage 1 repositories, as there are different CFG environments.

What

MCRBuddy needs to support a dynamic way of working with repositories. MCRBuddy needs to:

- Be able to support the creation of new repositories. For these repositories, we'll categorize them into stage 1, stage 2 and stage 3 environments, as the committing process is different (see 2.6)
    - For stage 1 repositories, MCRBuddy will have to build up the repository using the assigned CFG environments (e.g. we select only 1 country CFG environment instead of all).
    - For stage 2 repositories, the user will have to select a stage 1 repository as source, of which the contents are copied as a starting point for the repository.
    - For stage 3 repositories, the user will have to select a stage 2 repository as source, of which the contents are copied as a starting point for the repository.
- Be able to support the specific assignments of environments to repositories. This way, when committing on a repository, the contents of the commit will be deployed to the assigned environments (taken into account the layers, so BE environments will only have common layer and BE layer files deployed, NL environments only common layer and NL layer files).
- For the stage 1 environments, we need to assign one CFG environment per layer. In this environment MCRBuddy will fetch the XML files (see 2.6.1). The other environments will be seen as TST environments.

The standard setup, all environments on release 2020.1, would look like this:

Repository	Environments	Comments
Stage 1 (2020.1)	BMW_CFG_MASTER, BMW_CFG_BE, BMW_CFG_NL, BMW_TST_BE, BMW_TST_NL	BMW_CFG_MASTER is for the common layer, BMW_CFG_BE for the BE layer and BMW_CFG_NL for the NL layer
Stage 2 (2020.1)	BMW_INT_BE, BMW_INT_NL	
Stage 3 (2020.1)	BMW_PROD_BE, BMW_PROD_NL, BMW_MIR_BE, BMW_MIR_NL	

For example, when we are doing a release upgrade to for example release 2021.1, the following steps above can be translated to the following actions:

1. An admin user creates new environments (BMW_CFG_MASTER_2, BMW_CFG_BE_2, BMW_CFG_NL_2), creates a new stage 1 repository "Stage 1 (2021.1)" and assigns the new environments to this new repository (each to their layer).
    - Creating a new repository will trigger a nightly run that will export all CO XMLs from these CFG environments into the newly created repository, and do one first commit.

      > **Note**
      >
      > Details on the most efficient way to do this full export are still being looked into, more details will follow.

    - After the repository is created and has had it's first commit, users are to be able to do stage 1 commits on both "Stage 1 (2020.1)" and "Stage 1 (2021.1)".
2. The admin user will reassign the BMW_TST_BE and BMW_TST_NL environments to the new "Stage 1 (2021.1)" repository.
    - This will trigger a nightly run that will go through the whole database on these environments, and check the timestamps for each CO. Each CO with a timestamp BEFORE the last commit on this new repository that contains this CO, will be updated.
    - After this nightly run the TST environments are usable for testing.
3. A new "Stage 2 (2021.1)" repository is created and BMW_INT_BE and BMW_INT_NL are assigned to this.
    - We select "Stage 1 (2021.1)" as source, so the contents of this new repository are copied from the "Stage 1 (2021.1)" repository.
    - A nightly run like on the TST environments in point 2 will occur.
4. A new "Stage 3 (2021.1)" repository is created and BMW_PROD_BE, BMW_PROD_NL, BMW_MIR_BE and BMW_MIR_NL are assigned to it.
    - We select "Stage 2 (2021.1) as source, so the contents of this new repository are copied from the "Stage 2 (2021.1)" repository.
    - A nightly run like on the TST environments in point 2 will occur.

> **Note**
>
> There can be more than one stage 1 repositories that have environments linked to them.
>
> There can however only be one stage 2 repository and one stage 3 repository with environments linked to them.
>
> Repositories that don't have any environments linked to them are there as history. They can't be used to commit new work on.
>
> This is why it's very important that when a new stage 2 or stage 3 is created, the moment when the environments are transferred, there can be no outstanding commits (in the form of stage 2 requests or outstanding stage 2 work) on the old repositories. This shouldn't block it, but a warning should be given that work will be lost.

How

Details on how the git repositories will be managed are to be decided by the development team.

In order to keep everything dynamic, we'll save a lot of meta data on repositories, environments and even layers in the MCRBuddy database, allowing us to add or change repositories/environments (see 2.2.5).

A few admin panels will need to be created in the MCRBuddy application, where an admin user can create/change repositories, environments and layers, as well as assign environments to specific repositories.

2.2. Meta-data

In order to function as effeciently as possible, MCRBuddy will need to keep and maintain some meta-data. Following subsections will describe what we will store and where.

2.2.1. Layers, environments and repositories information

Context

In order to support our dynamic multirepository setup (see 2.1.2), we need to store some information on layers, environments and repositories.

What

We'll have a database table with layer meta-data, with following columns:

- Layer ID
- Layer name

We'll have a database table with environment meta-data, with following columns:

- Environment ID
- Environment name
- Is config: Indicates whether this environment is a configuration environment (and thus potential source)

We'll have a database table that links environments to layers, with following columns:

- Environment ID
- Layer ID

We'll have a database table with repository meta-data, with following columns:

- Repository ID
- Repository name
- Stage
- Source: Link to stage 1 or 2 repository in case of a stage 2 or 3 repository.

We'll have a database table that links repositories to environments, with following columns:

- Repository ID
- Environment ID

How

TBD by development team.

2.2.2. Issue information

Context

For each issue we work on in MCRBuddy, we'll need to store some information.

What

We'll have a database table with issue meta-data, with following columns:

- Issue reference: Can be used as primary key
- Stage: The current stage of the issue

How

TBD by development team.

2.2.3. Config version and commit information

Context

While we can perfectly use Git to browse commits and their contents, this is a cumbersome process. We also want to link other information, like a config version identifier, as well as an issue reference, to a commit. We could store this in the commit message (which we will, in case we browse our repositories in bitbucket) and then find this info there, but this is a cumbersome way of working. In order for the application to be performant, we need quick access to this information.

What

A database table with commit meta-data, with following columns:

- ID: Unique incrementing ID
- Repository ID
- Config version identifier
- Timestamp
- Commit hash
- User ID
- Roll-back to: Filled with commit ID of roll-back point in case of a roll-back commit, empty if it's not (see 2.6.4).

How

TBD by development team.

2.2.4. Config object information

Context

One of the things we want to avoid as configurators, is working on the same config objects at the same time. Now usually we don't have situations where we are working on 2 tasks that are covering the same config objects, and if we do, we just discuss this among the team and make sure no issues can occur. In a small team this is feasible of course, but as the team will grow and we'll get multiple markets live, we'll have to avoid situations like this.

What

We'll have a database table with config object meta-data, with following columns:

- ID: Unique ID as primary key
- Layer: Layer of where the config object exists in (common, BE, NL)
- Config Object Type: Type of the config object (e.g. Formula, Process Type,...) (David Vanpachtenbeke (FG-6-C2-BE) Store the SRO ID ?)
- Config Object ID: Id of the config object
- Deleted: Indication for config objects that no longer exist

How

TBD by development team.

2.2.5. Config Objects Lifecycle (COL)

Context

One of the things we want to avoid as configurators, is working on the same config objects at the same time. Now usually we don't have situations where we are working on 2 tasks that are covering the same config objects, and if we do, we just discuss this among the team and make sure no issues can occur. In a small team this is feasible of course, but as the team will grow and we'll get multiple markets live, we'll have to avoid situations like this.

What

With the introduction of the MCRBuddy application, we have a chance to help us avoiding situations like this. We'll do this with the **Config Objects Lifecycle (COL) functionality**. The COL will keep a lifecycle on each config object that is being worked on.

How

The COL will be kept in the MCRBuddy database. Each individual entry would look like this:

- Config object ID: Id of the config object (foreign key to our config object table, see 2.2.3)
- Commit id (see 2.2.3)
- Addition/Change/Deletion

Each time a commit is done, an entry is created for each config object part of that commit.

When a config object is included in stage 1 commit, a check is done:

- If the last entry was one with Stage set to "Stage 3", then the commit can happen.
- If the last entry was one with Stage set to "Stage 1", then the commit can ONLY happen if the issue reference is the same as the one on the commit information.
- If the last entry was one with Stage set to "Stage 2", then the commit can ONLY happen if the issue reference is the same as the one on the commit information. Even if the issue reference is the same, a pop up is shown informing the user and asking confirmation.

The config objects lifecycle will not only help us avoid mistakes, but will also give us a full overview of the development history of each config object.

2.2.6. Solution notes

Context

Right now, every issue that needs configuration gets a solution note on confluence. These are fine for rereading about certain developments, but don't offer us an easy way to, for example, find out for which issue(s) a certain config object has been changed or added.

By storing these in the MCRBuddy database, it will be much easier to find this information.

What

We'll have a database table for general solution note information, with following columns:

- ID
- Issue reference
- Title
- Type
- **Markets**
- Analysis link
- Change Description
- Solution

For each config object that was changed, we'll create an entry in a seperate database table with following columns:

- Solution note ID
- Config object ID
- Addition/Change/Deletion
- Description

There will be exactly one solution note for each issue. More information on the creation, see 2.6.2.

How

TBD by development team.

2.6.7. Requests

Context

MCRBuddy allows us to enforce the 4-eyes principle that's part of our processes. For this we need to store requests in our database.

What

We'll have a database table for request information, with following columns:

- Request ID
- Type: This to make a distinction between a stage 2 commit request or a stage 3 roll-back commit request (see 2.6.2 and 2.6.4)
- Timestamp
- Requester
- Number of impacted config objects
- Issue reference

How

TBD by development team.

2.3. Config Object Touch Tree (COTT)

Context

Each config object is linked to other config objects. This can happen in a direct way or indirect way. A couple of examples of direct links:

- Formulas can use other formulas as subformula, meaning the subformula will be executed and the result of this will be used in the main formula's logic.
- A process (workflow) can use other processes as subprocesses, but also uses formulas to program it's logic.
- A lookup table is linked to its lookup table definition.

And here are a couple of examples of indirect links.

- A formula can use a value of a certain attribute in it's calculation. This attribute is given a value by another formula.

- A property contains a list of lookup tables that need to be looked at by a certain policy. There is no direct link on a database level (the property value is saved as text).

As you can imagine, it is important for configurators to keep these links in mind. Configurator A can not change a lookup table at the same time that configurator B is changing its format by altering the lookup table definition. And while configurator C is changing a formula and believe his or her work is working fine, if a day later configurator D changes something to another formula that determines an attribute used in configurator C's formula, this might affect C's work completely, causing errors for C's development when D's work is transferred to stage 2.

It is thus vital that configurators are aware of what other config objects might influence or be influenced by their work, and make sure they're not interfering or being interfered with. Even at this moment, with only one live market, this is a risk. If we imagine working with multiple live markets, a common layer and having an ongoing implementation at the same time, with a bigger team in different locations, the risk will quickly go from "theoretical" to "practical issue".

What

We'll introduce the next functionality, the: **Config Object Touch Tree (COTT) functionality:**

- The COTT will provide us with all COs that are either used by or are using the config object (in 2 separate lists).
- The COTT will contain config objects both directly and indirectly linked. There will be an order in the lists, based on impactfullness (which is decided just by which type of link, e.g. a subformula is more impactfull than a formula that determines an attribute used by the given formula).
- The COTT is not a static thing. Each time it is requested for a certain config object, it is generated again.
- The COTT's generation will be different for each type of config object.

The development of the COTT functionality will allow us to do an additional check when adding a config object to a stage 1 commit. The COTT for the config object is generated, and for each of the other config objects on the list, MCRBuddy checks if any of them are currently in a state of development or testing (using the last lifecycle entry found for that config object) and that aren't linked to the same issue reference. If there are any, a warning is shown, detailing the config object, the issue reference, and the configurator who last changed this. The configurator can continue, but only after confirming. We want to trigger the configurator in this way to do the additional checks with his or her colleagues.

How

When developing the COTT functionality, regardless of who develops it, we'll have to work with an MVP approach. First we'll only develop this for the most common config objects, the one where the risks are the greatests. Also, not all links will be included in a first version, as covering everything from the start will be difficult. Some brainstorming sessions to make a selection of which objects and which links for an MVP will be needed. If developed as part of MCRBuddy, an idea would be to generate the COTT based on queries found in XML documents, and these XML documents would be expandable by the MPT itself. This way, with a framework in place, expansion of the MVP would be possible by the team without additional development.

Sofico involvement

Having the COTT functionality developed as part of Miles is a good idea. It has multiple benefits:

- It can be plugged into Miles RIA, so we'd also be able to see our COTT on the config object page itself
- It will be supported by Sofico, so any changes to the data model will be reflected in the COTT functionality as well.
- The developers from Sofico are much closer to the source, their is less risk of missing links.
- Not only BMW configurators, but configurators from other companies and those of Sofico itself would benefit from this development.

If Sofico decides to develop this functionality inside of Miles RIA, we will have to request a webservice that can be contacted by MCRBuddy to fetch the COTT for a certain config object.

2.4. Config Trace integration

Context

At this moment, the process of downloading the XML files for all config objects of a certain development is taking not only a lot of time, but also contains a high amount of risk on human errors.

The introduction of the MCR and more specifically MCRBuddy will not only minimize that risk, but also save us a lot of time.

However, if we look at the process for a stage 1 commit (see section 3.1.1.), we still need to select the individual config objects we have altered or added. While there is no more risk of downloading an XML but then uploading an older version, we still have the risk of forgetting an object we altered. We also have to select the individual objects each stage 1 commit. For bigger tasks, the amount of stage 1 commits can run up quite high (which is something we want, the more versions, the more points to go back to and the more the configurator is testing intermediate versions). We can minimize this:

- We can show users a list of recent objects they worked with
- We can allow users to first give a issue reference. All config objects linked to that issue are then selected, and filtered out if they were not changed since their last life cycle entry (see 1.4.2.). This way only the newly added or altered config objects need to be selected.
- We can run queries on the database, selecting all config objects that were altered or since the last commit of this user. However, some remarks must be made:
    - This will also show config objects altered by other users that have since then been committed by those users.
    - Running the queries every single time might be bad for performance.

Implementing above measures will minimize the work even more, but there will still be risk and might be bad for performance. Having these functionalities developed and maintained will also take some time.

What

Let's take a look at a functionality from Miles that will help us solve these concerns.

The config trace functionality in Miles, once activated, keeps a trace of all changes to configuration, and links each change to a given task and user.

A set of config changes can be exported in a zip file, called a change set, and moved to another environment and imported there.

The exported zip file contains an XML file for each config change. The XML file is the same as if we would export a single config object, the one that was changed/altered. Each XML is named after the ID of the config trace entry in the database.

We have recently decided to start using this functionality from the beginning of may, regardless if and when MCR will be developed. The benefits of using this when we implement MCR are clear:

- Miles tracks all our changes. If we export all changes for a given task, we have no risk of missing out a change. There is still a risk of human mistakes ofcourse (people could fill in the wrong issue reference in Miles), but Miles also allows changes to the issue reference for each tracked change, so mistakes can always be set right.
- Users will only need to fill in the issue reference for stage 1 commits, saving a lot of time in giving a complete list of added or altered config objects.
- For a stage 1 request, MCRBuddy will only need to do one request towards Miles: Requesting 1 config set zip file instead of a request per config object, each one containing an individual XML file.

How

There are two distinct ways we can use this config trace, each having its pros and cons:

- **Sofico adds 2 webservices:** Unlike the standard single import/export XMLs, change set zip files cannot be exported or imported through a web service. Sofico could develop these for us. For the export, we preferably have a webservice with following arguments:
    - Task name (required): Exported changes need to be linked to the given issue reference.
    - Timestamp (optional): Only changes after a given timestamp are exported.

  Basically, using the web service, MCRBuddy would fetch all changes for a certain issue. If there is one, a timestamp is sent as well of the date and time of the last commit for this issue, so only recent changes are requested and received.
  Two things are happening:
    - The individual files in the change set are parsed to get to know the type of object and the ID. The checks are done to see whether there are any dependencies (using the COL and COTT).
    - If there are no blocking dependencies, the files are renamed and put on the correct location in the stage 1 repository.
- **We use the config trace database table**: Using our own query on the config trace table in the Miles database, we find all config objects changed for a specific task and use the already existing single export webservice to fetch the XMLs.

The benefit of having Sofico create the webservices is ofcourse that we only need one request, and the functionality to bundle these XMLs already exists. The webservice would also be supported by Sofico, so any changes to the change set functionality are also included in our webservice.

The downside is that Sofico has to develop new webservices (at a cost). Another disadvantage is that MCRBuddy first needs to download a zip, read the contents and only then can tell the user whether his or her commit is not violating any dependencies.

The ideal solution is of course a combination of both:

- MCRBuddy checks the config trace table in the Miles database, getting all changed or added config objects since the last commit, for a given issue reference.
- An overview is shown to the user which config objects would be committed. Conflicts (using the COL) are marked in red and block the user from going further. Dependencies (using the COTT) are marked in yellow and user needs to explicitely approve these.
- Once user approves the commit, a webservice is used to fetch the config object XMLs (whether as a set using our new webservice or individually, this is to be decided).

Another benefit we have with using config trace will be described in section 1.4.5.

2.5. Nightly synchronization

Context

One of the risks in the current setup is potential desynchronisation of configuration accross environments. For example, if a configurator changes something on the CFG environment, but never transfers it to other environments, and never rolls the change back, this specific change is present on the CFG environment, but not on the other environments. Because no business data is present on the CFG environment and no tests never occur there, we don't know whether this change might cause exceptions or faulty behavior. Months later, someone else might alter the config object for another issue, and take with him or her the existing change, not present on other environments. This way, without knowing it, the configurator might introduce an error on the TST environment. If the faulty behavior is not visible in the scope of his or her issue, it might even get transferred to PROD without finding out. We want to avoid this at all costs.

What

Every night, MCRBuddy will do a synchronization check.

How

Every night at midnight, MCRBuddy will consult the config trace database table in Miles, to get a list of all config objects changed or added in the last 4 days, together with an exact timestamp of their last change.

For each change MCRBuddy will check the COL table to check whether a commit including this config object was done after that last change. For any check that has failed, one of two things can happen:

- For uncommitted changes less than 3 working days old, the config object is added to an email sent to all configurators, giving all changes that are yet uncommitted (including the issue for which the change has happened according to the config trace database table).
- For uncommitted changes more than 3 working days old, the config object is reverted, meaning that the latest version of the config object XML is taken from the repository and deployed on the environment. Rolled back changes are mentioned in the same email as those that are uncommitted.

This way, configurators have 3 working days to commit their changes. The email is sent to the team, so if a colleague goes on holliday, others can still commit for him or her.

2.6. Commit functionalities

In following subsection we'll discuss the different type of commit functionalities MCRBuddy needs to offer.

2.6.1. Stage 1 commits

Context

When a configurator is working on a specific issue that requires configuration, he or she will want to occasionally test their work on the TST environment.

He or she is in the first stage, and in order to get his or her configured work to the TST environments, as well as save a snapshot of their work, they will have to create new stage 1 commits.

What

The introduction of the MCR shows us that we have the stage 1 repository as a representation to the current situation on the CFG and TST environments.

MCRBuddy will include the possibility to create stage 1 commits.

How

1. First a check is done if no other user is currently creating a stage 1 commit:
    - If there isn't, the stage 1 commit is locked by storing the user and an expiration timestamp (current time + 10 minutes) in memory.
    - If there is, MCRBuddy looks at the expiration timestamp.
        - If this is passed (meaning more than 5 minutes of inactivity), MCRBuddy will block the other user and move the lock to the new user.
        - If this isn't not passed yet, MCRBuddy will inform the user that another user is creating a stage 1 commit (including who that other user is).

    In case of no locks, the first screen the user sees is a list of issue references:
    - These are fetched from the config trace database tables in the Miles CFG environments (those from the last 4 days, see 2.5. for the reason why it's 4 days)
    - If more than one stage 1 repository exists, the user can switch between them, each showing a different list (see 2.1.2 and 2.2.6)
2. On the second screen a list of all config objects are shown, fetched by looking at the config trace database table and selecting all records for the selected task with a timestamp higher than the last commit for this issue (if there is any, otherwise just select all). Following information is shown:
    - Config object type
    - Config object ID
    - Indication whether it's an addition, change or delete
    - Time and date of addition, change or deletion (using the config trace database table)
    - If applicable, time and date of last commit that included this config object (using the COL table)
    - If applicable, issue reference of last commit that included this config object (using the COL table)
    - If applicable, stage of last commit that included this config object (using the COL table)

    In this list there is also an indication whether a config object has any conflicts.
    - If the config object has a direct conflict, which we find out using the COL (see 2.2.3), then the row is marked red, from "blocking conflict".
    - If the config object has a indirect conflict, which we find out using the COTT (see 2.3), then the row is marked yellow, from "warning conflict".

    On this screen the user needs to confirm the changes.
    - In case of one or more blocking conflicts, the user cannot confirm and the user is notified that for those config objects with a blocking conflict, the changes are rolled back on the CFG environments (see 2.6.4)
    - In case of no blocking conflicts but one or more warning conflicts, the user can confirm, but upon pressing the confirm button, a pop up is shown, asking for an additional confirmation that the user has indeed investigated the warnings.
    - In case of no conflicts, the user can just confirm without additional pop ups.

    Before confirmation, the lock is checked:
    - If the lock is no longer for this user (e.g there is no lock or another user is locking), the user gets a message that his or her session has expired and that he or she has to try again
    - If the lock is still for the user, the expiration timestamp is updated (current time + 10 minutes)
3. After confirming, the actual commit happens, of which the progress is shown on the third screen.
    - Using the Miles webservices, the relevant config object XMLs are fetched on the CFG environments and placed on the correct location in our stage 1 repository
    - A new config version identifier is generated (see 2.2.3).
        - The last config version identifier for the selected repository is taken and the minor version is incremented by 1.
    - The commit message is generated using the issue reference, config version identifier and config objects.

Commit message example

```
Config version 1.14.3 for issue MILES4ALL-1012

Config objects:
- Formula 300736 (Addition, common layer)
- Formula 600271 (Change, BE layer)
- Process Type 600281 (Addition, BE layer)

Issue reference: MILES4ALL-1012
Config version: 1.14.3
```

- The actual commit on Git happens and is pushed
- The commit is deployed, by importing the CO XMLs on the relevant environments (see 2.2.4)
- Our meta data is updated:
    - Meta-data on the issue is saved or updated (see 2.2.2).
    - Meta-data on the new commit is saved (see 2.2.3).
    - For new config objects, new entries in the config object table are made (see 2.2.4).
    - The COL table is updated with new entries.
- The current lock is removed.

In case of errors during any of these steps, the following needs to happen:

- We retry the step that failed.
- If still errors occurred after the third try, everything is rolled back and a clear exception is shown towards the user. The current lock is also removed.

2.6.2. Stage 2 commits

Context

When a configurator is one developing and testing for an issue, the work is ready to be transferred to the second stage and deployed on INT environments for testing by business.

Before the actual transfer and deployment, we want the work first documented and reviewed by a colleague.

Where now this happens on confluence and by all loyally following the procesesses, with MCRBuddy we'll be able to enforce this as part of the stage 2 commits.

What

The introduction of the MCR shows us that we have the stage 2 repository as a representation to the current situation on the INT environments.

MCRBuddy will include the possibility to create stage 2 commits. This will work in 2 steps:

- Request: The author (or one of the authors) of the work for this issue requests a stage 2 commit. As part of the request he or she fills in the solution note.
- Review: Another configurator (not an author) reviews the request. The solution note is used and the reviewer is expected to do some small tests on the TST environments. The reviewer can either approve or dissapprove (with feedback).

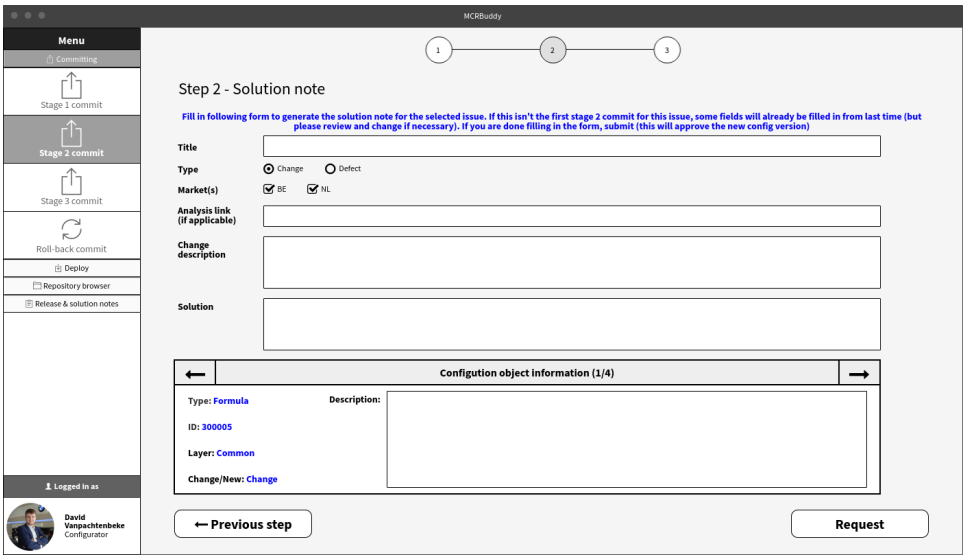
How

The first screen contains 2 tabs. One for requests, one for reviews. From the second screen on, we'll have a slightly different flow.

We'll first see the request flow:

1. On the first screen in the request tab we'll have a list of all issues for issues currently in stage 1 exist and for which the current user is one of the authors. Following fields are shown:
    - Issue reference
    - Number of impacted config objects
    - Authors

   We can easily use the issue database table to fetch our issues with in stage 1 (2.2.2), the config version table for the authors (2.2.3) and the COL table for the number of unique impacted config objects.
   The user can select 1 of these issues (if there are any) and press the "Create request" button to go to the next screen.
2. On the second screen the user needs to create the solution note. Here is a design of how this screen could look like.



Note how the config object information section has most fields already filled in, using the meta data tables.
Only when all fields are filled in (including the descriptions of all config object changes) can the user press the request button.
When pressing the request button, the solution note is saved in the solution note tables (see 2.2.6).

> **Note**
>
> When generating the solution note, a check is done. Any config objects that have been added AND deleted in the same issue are NOT included in the solution note.

> **Note**
>
> If this is not the first time that a request is submitted for this issue (either because there were new requirements after testing or because the first request was denied), most of the solution note data is already filled in.
>
> Only new config objects will have their description field empty. The user however needs to be able to change any existing fields.
>
> When updating the solution note in the database, we need to take into account that since the last solution note a previously added config object might have been deleted again (and thus it needs to be removed from the solution note).

3. The third screen gives a confirmation to the user the request has been submitted.
   Apart from this, an entry is created in the requests table (2.2.7) and the issue table (2.2.2) is updated (the stage is set to stage 2).

Now we'll take a look at the review flow.
Because we'll do the actual commit at the end of an approval, we'll need the same lock mechanism as for stage 1 requests (of course, a seperate lock than the stage 1 lock, as there is no problem simultanously committing on stage 1 and 2).

1. On the first screen in the review tab we'll have a list of all stage 2 requests. Following fields are shown:
   - Issue reference
   - Requester
   - Date and time of request
   - Number of impacted config objects
   - Authors

   The user can select 1 of these requests (if there are any) and press the "Start review" button to go to the next screen.
   - This button is only enabled when there is no lock, otherwise a message is shown.
   - Pressing the button will start a new lock (same mechanism as stage 1, see 2.6.1).

   A user can see all requests, but only select the ones where he or she is not an author.
2. On the second screen the user sees the solution note. This is the same as in our request flow, only now everything is filled in and non-editable.
   There are now 2 buttons:
   - Approve: Pressing this button will approve the request and MCRBuddy will go to the next screen.
   - Deny: Pressing this button will deny the request and MCRBuddy will show a pop-up in which the user needs to fill in a reason. Following things happen:
     - An email is sent to the requester, notifying of the decision and the reason why.
     - The request entry in the requests table (2.2.7) is removed.
     - The issue table (2.2.2) is updated (the stage is set to stage 1)
     - The flow is broken off and the user is brought back to the home screen.
3. After approval, the actual commit happens, of which the progress is shown on the third screen.
   - All impacted config object XMLs are copied from the stage 1 repository and pasted on the same locations in the stage 2 repository.
   - A new config version identifier is generated (see 2.2.3).
     - The last config version identifier for the selected repository is taken and the minor version is incremented by 1.
   - The commit message is generated using the issue reference, config version identifier and config objects.

Commit message example

```
Config version 1.14.3 for issue MILES4ALL-1012

Config objects:
- Formula 300736 (Addition, common layer)
- Formula 600271 (Change, BE layer)
- Process Type 600281 (Addition, BE layer)

Issue reference: MILES4ALL-1012
Config version: 1.14.3
```

- The actual commit on Git happens and is pushed
- The commit is deployed, by importing the CO XMLs on the relevant environments (see 2.2.4)
- Our meta data is updated:
    - Meta-data on the issue is saved or updated (see 2.2.2).
    - Meta-data on the new commit is saved (see 2.2.3).
    - For new config objects, new entries in the config object table are made (see 2.2.4).
    - The COL table is updated with new entries.
- The current lock is removed.

2.6.3. Stage 3 commits

Context

After business testing and their approval, the development is ready to moved to production. Right now, one of the product owners gives his final go, causing the configuration XMLs to be picked up in the next production deploy.

There is no official way of giving the go signal however. There is a risk of miscommunication or and accidentily taking issues to production while not tested fully (or forgetting work that has been greenlighted and promised to business).

What

The introduction of the MCR shows us that we have the stage 3 repository as a representation to the current situation on the CFG and TST environments.

MCRBuddy will include the possibility to create stage 3 commits, which unlike stage 1 and stage 2 commits, will not be deployed right away (for this, see 2.7.2).

How

1. The first screen will show a list of all issues that are currently in test (use issue database table, see 2.2.2). The user can select one of them and press the "Review" button.
2. The second screen will show the solution note, filled in, just like in in our stage 2 review. The user can however only confirm.
3. After confirmation, the stage 3 commit will happen, of which the progress is shown on the third screen.
    - All impacted config object XMLs are copied from the stage 1 repository and pasted on the same locations in the stage 2 repository.
    - A new config version identifier is generated (see 2.2.3).
        - The last config version identifier for the selected repository is taken and the minor version is incremented by 1.
    - The commit message is generated using the issue reference, config version identifier and config objects.

Commit message example

```
Config version 1.14.3 for issue MILES4ALL-1012

Config objects:
- Formula 300736 (Addition, common layer)
- Formula 600271 (Change, BE layer)
- Process Type 600281 (Addition, BE layer)

Issue reference: MILES4ALL-1012
Config version: 1.14.3
```

- The actual commit on Git happens and is pushed
- Our meta data is updated:
    - Meta-data on the issue is saved or updated (see 2.2.2).
    - Meta-data on the new commit is saved (see 2.2.3).
    - For new config objects, new entries in the config object table are made (see 2.2.4).
    - The COL table is updated with new entries.

As you can see, this is mostly the same than the stage 2 review, apart from the fact that no deployment is done.

This functionality will only be available for product owners, for more information, see 2.8.1).

2.6.4. Roll-back commits

Context

One of the main benefits of working with the MCR is that every single commit is a snapshot of how the situation was at a certain moment.

This allows us to roll-back some of the work we have done.

What

We'll have a functionality to roll-back our work using MCRBuddy. While the possibilities are endless, we need to keep things on the practical and safe side. This is why we'll only allow following roll-backs:

- Stage 1 roll-back: For an issue in stage 1, we allow to roll-back work to a certain snapshot.
- Stage 2 roll-back: For an issue in stage 2, we allow the roll-back of a commit.
- Stage 3 roll-back: For an issue in stage 3, we allow the roll-back of a commit after a double review, but only if all impacted config objects are also in stage 3 and last changed because of this issue.

How

The first screen will contain 3 tabs, one for each stage.

We'll first take a look at the stage 1 flow. Because we'll do an actual commit on the stage 1 repository, we'll use the same lock as from stage 1 commits (see 2.6.1).

1. On the first screen in the request tab we'll have a list of all issues currently in stage 1 and for which the current user is one of the authors, as well as all issues that don't have any commits yet (use config trace table).
   A second list, updated each time the user selects an issues, shows all commits done for this issue, as well as a "Initial situation" entry. This last one will allow the user to roll the entire issue back and will be the only option for issues that don't have any commits yet.
   Upon selecting a roll-back point, an "Overview" button becomes clickable (if there is no lock ofcourse, if there is, a message is shown that someone is currently working on stage 1 commits).
2. After selecting an issue and a roll-back point, the second screen will show an overview of impacted config objects. Following fields are shown:
   - Type
   - ID
   - Timestamp of last change before or at the roll-back point.
   - Issue for which last change happened before or at the roll-back point
   - Number of changes that will be rolled back

   This information can easily be distilled from the COL table (see 2.2.4).
   Below this list is a "Roll-back" button.
3. Upon pressing the roll-back button, the roll-back commit will happen, of which the progress is shown on the third screen.
   - The stage 1 repository is checked out to the roll-back point.
     - In case "Initial situation" was selected, this is the commit before the first commit for this issue.
     - If are no commits yet, this is the HEAD.

     From this checked out repository, all impacted config objects are copied and pasted into a temporary directory.
   - The stage 1 repository is checked out to the HEAD, and all config objects in our temporary directory are pasted on the correct places.
   - A new config version identifier is generated (see 2.2.3).
     - The last config version identifier for the selected repository is taken and the minor version is incremented by 1 and an "r" is added at the end to indicate this is a roll-back commit.
   - The commit message is generated using the issue reference, config version identifier and config objects.

     Commit message example

     ```
     Config version 1.16.23r for issue MILES4ALL-1012

     Config objects:
     - Formula 300736 (Deletion, common layer)
     - Formula 600271 (Change, BE layer)
     - Process Type 600281 (Deletion, BE layer)

     Issue reference: MILES4ALL-1012
     Config version: 1.16.23r
     ```

   - The actual commit on Git happens and is pushed.
   - Our meta data is updated:
     - Meta-data on the issue is saved or updated (see 2.2.2).
       - In case we have gone to the initial version, we change our stage to 0.
       - In case we didn't have an entry yet, one is created, again with stage set to 0.
     - Meta-data on the new commit is saved (see 2.2.3).
     - The COL table is updated with new entries.

Our stage 2 flow will run almost the same way, except that a roll-back of a issue on stage 2 rolls-back all work. We'll also work with a lock, this time the stage 2 lock (see 2.6.2).

1. On the first screen in the request tab we'll have a list of all issues currently in stage 2.
   Upon selecting a roll-back point, an "Overview" button becomes clickable (if there is no lock ofcourse, if there is, a message is shown that someone is currently working on stage 2 commits).
2. After selecting an issue, the second screen will show an overview of impacted config objects. Following fields are shown:
   - Type
   - ID
   - Timestamp of last change before or at the roll-back point.
   - Issue for which last change happened before or at the roll-back point

   This information can easily be distilled from the COL table (see 2.2.4).
   Below this list is a "Roll-back" button.
3. Upon pressing the roll-back button, the roll-back commit will happen, of which the progress is shown on the third screen.
   - The stage 2 repository is checked out to the the commit before the first commit for this issue.
   - From this checked out repository, all impacted config objects are copied and pasted into a temporary directory.
   - The stage 2 repository is checked out to the HEAD, and all config objects in our temporary directory are pasted on the correct places.
   - A new config version identifier is generated (see 2.2.3).
     - The last config version identifier for the selected repository is taken and the minor version is incremented by 1 and an "r" is added at the end to indicate this is a roll-back commit.
   - The commit message is generated using the issue reference, config version identifier and config objects.

     Commit message example

     ```
     Config version 1.16.23r for issue MILES4ALL-1012

     Config objects:
     - Formula 300736 (Deletion, common layer)
     - Formula 600271 (Change, BE layer)
     - Process Type 600281 (Deletion, BE layer)

     Issue reference: MILES4ALL-1012
     Config version: 1.16.23r
     ```

   - The actual commit on Git happens and is pushed.
   - Our meta data is updated:
     - Meta-data on the issue is saved or updated (see 2.2.2).
       - The stage is set to 1 again.
     - Meta-data on the new commit is saved (see 2.2.3).
     - The COL table is updated with new entries.

The stage 3 flow will be the same as the stage 2 flow, except for the fact that we'll work with a request and double review system (use same principles as 2.6.2) and there will be no lock.

2.7. Deploying

When talking about deploying, we need to talk about both the general deployment procedures as well as the stage 3 deployment screens.

2.7.1. General deployment

Context

With deployment on an environment, we want to import the configuration encapsulated in a certain commit to be imported on the relevant environments.

Where this is now a tedious manual process, we want to automate this in MCRBuddy.

What

Automatic deployment of config object XMLs on an environment.

How

Details for now are unclear, as we are still in talks with Sofico about potential webservices.

At this moment, the possibility exists to use a webservice to upload one single XML file at a time. Problem is that one has to do the correct order of files, because otherwise dependency errors might occur.

Sofico does however have algorithms to import files in bulk, yet no webservice exists for this yet.

We would want a webservice like this to be developed. MCRBuddy would send a zipfile with XMLs, which would be imported as a whole.

> **Note**
>
> We would ask for a test mode to exist on this webservice. This would allow MCRBuddy to first send the XMLs and see if import is possible, if it is, only then the commit will happen.
>
> If this test mode will be delivered, all commit algorithms will be extended, first doing this check.

2.7.2. Stage 3 deployment

Context

Unlike stage 1 and stage 2 commits, stage 3 commits are not immediately deployed.

This is because we want to work, just like in our current setup, with releases. Because some configuration is linked to a certain patch of Miles (installed on all environments but not yet on PROD), we want the installation of the configuration to happen simultanously.

What

A seperate functionality for stage 3 deployments will be added.

The deployment will consist out of 2 phases:

- The planning: An operational user plans in a new deployment, a new major config version is created, release notes are generated.
- The execution: On the planned time the newly created major config version is deployed on the relevant environments.

How

We'll first look at the planning. Because we'll be deploying on all repositories, we'll work with our locks (so all of them need to be free before the user can start, and he or she will lock all repositories, but only for a very short time).

1. The first screen an operational user needs to select a date and a time for the deployment.
2. On the second screen a generated release note is shown and can be edited. Upon review/editing the user can press the "Plan" button.
    - The release note template will be given on a later date, but will contain information on all delivered issues.
3. The new major config version is created, of which the progress is shown on the third screen.
    - The generated release note is written to a file and moved to all repositories.
    - A new config version identifier is generated (see 2.2.3) for each repository.
        - The last config version identifier for each repository is taken and the major version is incremented by 1.
    - A commit message is generated.

      Commit message example

      ```
      A new major version is created.

      Config version: 1.14.0
      ```

    - The actual commits on Git happen and are pushed.

The execution will happen on the planned time, deploying the last stage 3 major version.

- All individual minor versions between the last major version are taken, and all impacted config object XMLs are taken from the stage 3 repository, moved to zips (one for each environment).
- Each zip is sent to the deployment webservice (in test mode).
    - If all are successful, the actual deployment is done. An email is sent to all users with this information.
    - If not, the deployment is finished. An email is sent to all users, with error messages included.

2.8. Security

2.8.1. User rights

Context

Just like any system, we need good security on our application to make sure only the right people have access.

What

MCRBuddy will provide security based on user rights and roles.

- Each functionality in MCRBuddy should have a right to it.

- We want to work with roles that mimic reality, like "Feature team member - Configuration", "Feature team member - Operational" and "Product owner".
- There should be an admin role, allowing users with this role to change other roles and assign roles to users.

Apart from this it should be possible to have IWA-login integration, so MCRBuddy users are linked to the BMW accounts.

How

TBD by the development team.

2.8.2. Secure connections

Context

Apart from security on users rights, we also need secure connections between MCRBuddy and Miles.

What

Secure connections need to be established between MCRBuddy and other servers (see architecture, section 3).

How

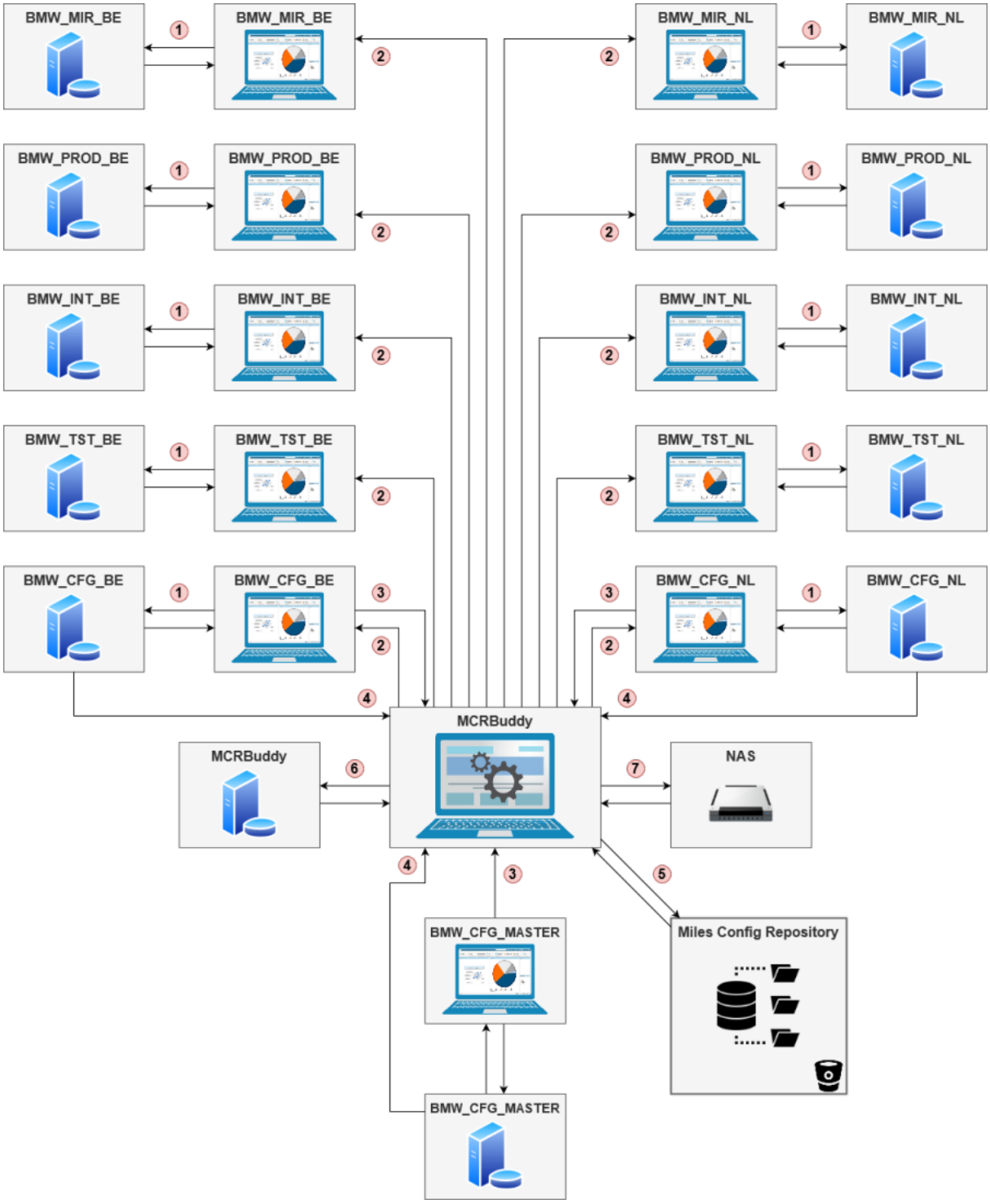
TBD by the development team.

2.9. Other functionalities

Here is list of other functionalities that can be provided by the MPT (still to be written out):

- Repository browser: Browse all config objects and all changes ever done to them.
- Issue browser: Browse all issue currently in development and those from the past.

3. Architecture



	Explanation
BMW_MIR_[countrycode]	Countryspecific mirror environment.
BMW_PROD_[countrycode]	Countryspecific production environment.
BMW_INT_[countrycode]	Countryspecific integration environment.
BMW_TST_[countrycode]	Countryspecific test environment.
BMW_CFG_[countrycode]	Countryspecific configuration environment.
MCRBuddy	

BMW_CFG_Master	Non-coutryspecific configuration environment.
Miles Config Repository	Git repository in the Agile Tool Chain (atc.bmwgroup.net)
NAS	Fileserver to be used by MCRBuddy

3.1. Technologies

The MCRBuddy application will have to be able to interface with following technologies:

- Interfacing with a Git repository (5)
- Interfacing with Oracle SQL databases (4)
- Interfacing with 2 REST web services (2, 3)
- Interfacing with a remote NAS fileserver (7)
- Interfacing with JIRA (optional)

4. Infrastructure

The application should be hosted in the BMW intranet.

5. Non Functional requirements

5.1 Performance

Performancy wise we have a couple of requirements:

- The application should run "smooth" when going from one screen to the next (e.g. no long loading times).
- When committing and deploying, we accept that this can take some time, however since we are blocking other configurators from creating commits, we don't want this to take to long (depending on the number of config versions, a commit should take maximum 1 to 5 minutes).

A lot will ofcourse be decided by the performancy of the repositories itself and the webservices of Miles. Because of this focussing on stable connections between all servers (MCRBuddy server, Miles, NAS, Bitbucket,...) will be important.

5.2 IT Governance

The application should be developed & designed according to IT Workinstructions of the Agile Working Model: IT Work Instructions and Guiding Principles. Any deviation must be documented and accepted by the Miles product Team prior to development.

5.3 Operations

An Operations Manual according to the BMW standard in the Agile Working Model should be delivered.

=> TBD: Who will take on operational management? I would opt for the MPT itself.

5.3.1 Backups

Every day at midnight a backup of the MCR needs to be taken on a separate NAS server. This is not in the scope of the MCRBuddy application, as we can set this up inside the existing BMW framework.

=> Jens Vanpachtenbeke (FG-6-C2-BE) Klopt dit wat ik zeg?