DESIGN REPORT

SCRATCHPAD MANAGER

Group 9

Bram Stoffelsen	s2914786
David-Filip Ionas	s2960249
Elise Korsmit	s2999471
Mathijs Polman	s2976889
Matthijs Veldkamp	s2462478
Melle Bosboom	s2974630

University of Twente - Faculty of Electrical Engineering, Mathematics and Computer Science

TCS Design Project, 2025-11-14

Supervised by: ir. Dorus Abeln

Contents

1. Introduction

The CAES research group of the University of Twente maintains four powerful lab servers. These servers run a variety of software for simulation, synthesis, and other general-purpose scientific computing for use by students and staff. To ensure agile usage and availability of these four servers, persistent storage of data is not allowed on those. Instead, students and staff are given personal home directories on a separate networked file server, making their files accessible from any of the four lab servers.

Though convenient, the file I/O over the network is a bottleneck for the computationally intensive tasks as these generate and read many temporary files. As a solution, these lab servers are equipped with fast NVMe SSDs, known as *scratch disks*, or *scratchpads*. These disks are intended for temporary, high-speed storage, solving the bottleneck problem of networked file I/O.

However, if those temporary, user-generated files are not deleted after they served their purpose, they block others from using the limited space on the scratch disk. Currently, users are expected to delete their project folders once their projects or computations are complete. In practice, this manual process proves unreliable, as users forget to delete these directories, reducing the available storage space for others. Our project aims to design and implement a tool which addresses this accumulation of abandoned project files.

1.1. Project Description

The client provided beforehand a list of motivated core requirements our project had to adhere to, which he elaborated and expanded on during the first and second client meeting. A summary of the list is given here:

- The "scratchpad directory manager" must be a tool which automates the life-cycle of scratchpad directories by enforcing a policy of timed, automated cleanup. It has to have a command-line interface which is well-documented, with intuitive commands such as "*create*", "*remove*", "*extend*", "*help*" and include a "*man page*" (manual page).
- The manager has to keep track of the directories on the scratchpad reliably; so some sort of database is required with an entry for each directory on the scratchpad. The database would store, among other things, the directory's name and scheduled date of deletion.
- The tool also has to create scratchpad directories on a user's behalf. This means that users are required to create scratchpad directories through the system. Additionally, the system must provide users a method to extend the lifespan of their directories if they so desire, as the system will delete scratchpad directories and their contents after a predefined interval.
- To enforce these restrictions, the tool must be able to manipulate the scratchpad directory to which normal users don't have write permissions. This requirement also means that the tool must be secure against unauthorised manipulation or circumvention attempts.
- Lastly, the tool must notify its users when their directory on the scratchpad is nearing its deletion date. Therefore, it must send a notification email at some pre-determined moment(s) before deletion.

The client expressed a desire to be able to modify the tool in the future, stressing the importance of great documentation, maintainability and code quality. Also, to maintain the system, the client wishes to be able to change many variables from a config file such as; default retention time upon directory creation, maximum user-settable retention time and the default for when notification emails are sent.

1.2. Current Procedure

Currently, the file system is completely unmanaged, save what the administrator or the users do themselves. Users are expected to make a folder with the same name as their username and use it as they see fit.

After the user does not need their directory anymore, they are expected to delete their files manually, with the use of the Linux command-line interface.

Currently, everyone has read and write access to the scratchpad. As mentioned previously, that will change once our product is introduced.

Besides the forgetfulness, or laziness, of the users when it comes to deleting their directory after use, automating the process of creating and deleting directories will correct for user errors that can be made during the current manual process. Users might type their user name wrong accidentally, which results into vagueness as to who the directory belongs to (this is something we observed).

1.3. Stakeholder Analysis

Regarding this project, we determined three stakeholders with their respective interests:

Users:	Students and university staff who use the lab servers for their own projects and research.
Maintainer(s):	The individual(s) responsible for fair use of lab server resources, availability and responsible usage.
LISA:	The university ICT department responsible for the security of the lab servers.

1.4. Terminology

In order to make defining and reading the requirements easier, we constructed a list of concepts which reappear often in our requirements lists:

Scratchpad:	One of the high-speed storage disks as deployed on the lab servers, meant for temporary data storage.
Scratchpad Manager:	The product of interest. The system.
Scratch Directory:	A user-owned directory on the Scratchpad, by default at */local/scratch*, the directory managed by the Scratchpad Manager.
Expiry Date:	The date a Scratch Directory will be deleted, unless manually extended by the owning user.
MTTL:	Administrator-settable maximum allowed scratch directory time-to-live.

Additionally, we use the term administrator and maintainer interchangeably, because there is currently a single individual responsible for the tasks that correspond to both roles.

1.5. Project Phases

The initial phase of the project was gathering requirements and understanding the current environment. For this we had several meetings with our client, the administrator, to ask questions about his vision for the project and relevant specifications of the servers. Additionally, we were granted access to the servers, using our student credentials, to have a look around at how things currently work. During this phase we defined the goals and requirements for this project, some of which changed, disappeared or were added in later stages of the project when we gained new information or insights. We ordered the requirements according to their priority for the project and used this as a guide for our planning. We did not set any strict deadlines for functionalities, due to our preference for flexibility in our workflow. Nevertheless, we were continuously on track of where we wanted to be with our project (measured by looking how much was done vs. how much still needed doing).

The design phase overlapped largely with the analysis and requirements gathering phase and the implementation phase. We worked on some initial designs: we made some diagrams indicating our initial design, we thought about the database concept (which was relatively simple), and created some proof-of-concepts before starting to work on the actual commands. Most of our actualised design choices happened as we were also working on implementation: as we were discovering along the way what initial ideas worked and which didn't, how to adapt it to something that would work, and what it would mean for the next functionalities that relied on the previous ones. This flexibility worked well for us, because we spent most of our time working on the project during daily face-to-face meetings which allowed for quick and easy communication and collaborative troubleshooting.

During the first half of the implementation phase, we were making design choices based on changing requirements and implementation experiences. Nearing the end of this phase, we started writing the documentation and manual for the finished functionalities. Additionally, some developers started writing tests for the (nearly) finished functionalities before all of the implementation was finished, though by that time mostly small fixes and improvements remained.

Figure 1 shows the different parts of our project cycle and their interactions. Flexibility and reflection were very important to us throughout this project, which resulted in a lot of back-and-forth between and parallel working on the design phases.

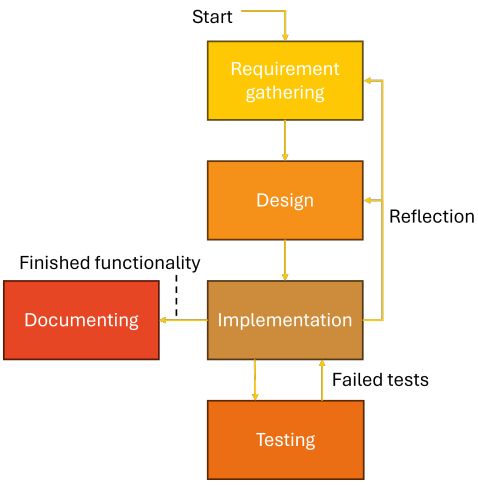

Figure 1: Design phases of our project.

2. Requirement Specification

This chapter will focus on the requirements, and the process of forming them. The requirement specification grows with the project and has been actively engaged with for the full duration of the design. The requirements need to be clearly defined for us to work with, this is why some techniques (e.g. client meetings, document analysis, observation and MoSCoW) have been selected in order to correctly identify and order the requirements.

Note that while we wrote our initial list in the first and second week, it has been continuously refined and received small updates when we made new discoveries for our project requirements. The requirements we present here are the final version for this project. Refer to Appendix A for initial design and final design diagrams.

2.1. Requirement Formulation and Prioritisation

The technical requirements are formulated using the MoSCoW method, which categorises them based on their importance and impact on key stakeholders. This approach ensures that the most critical functionalities are implemented first. It also gives a clear interpretation of requirements, which in turn makes defining tests easier. The prioritisation process relies on the urgency of the functionality and its importance to the stakeholders.

2.2. Requirement Types and Functionality Relations

For the formulation of our requirements, we distinguish between different types. Functional requirements capture the intent of the system. They are high-level and describe the specific functions our stakeholders should be able to perform using our program. There are also technical requirements. These do not directly follow from stakeholders' wishes, but instead support the implementation of the functional requirements. While functional requirements are closely tied to stakeholders, technical requirements are more relevant to the developers. Together, they form a bridge between stakeholder needs and the system's technical realisation.

2.3. Functional Requirements

User – basic scratch directory management

1. As a User, I want to be able to create a scratch directory on the Scratchpad.
2. As a User, I want to be able to create additional scratch directories on the Scratchpad. (Nice to have)
3. As a User, I want to be able to manually delete a scratch directory of mine when I deem it not necessary anymore.
4. As a User, I want to be able to list/view all existing scratch directories I have created.
5. As a User, I don't want to have to delete a scratch directory of mine when I don't need the contents of the directory anymore.
6. As a User, when I create a new scratch directory, I want to be the only user with read & write access to this new scratch directory by default.
7. As a User, I want to be able to rename a directory in the scratchpad. (Note: not possible out of system)

User – expiry date

8. As a User, I want to be able to see when one of my scratch directories expires.
9. As a User, I want to be able to extend the expiry date of a scratch directory of mine within the MTTL.
10. As a User, I want to receive a warning/multiple warnings when one of my scratch directories is soon to expire and be deleted. I want this, so in preparation I can copy over important files to my network drive or extend the expiry date, if needed.

User – manual and configuration

11. As a User, I want to view the man (manual) page of the Scratchpad Manager program.
12. As a User, I want to view my default expiry time, permission mask, and emailing configurations. (Nice to have)
13. As a User, I want to be able to configure my default expiry time, permission mask, and emailing configurations. (Nice to have)

Maintainer – basic scratch directory management

14. As a Maintainer, I want to access an overview/list of all User scratch directories on the Scratchpad with expiry dates.
15. As a Maintainer, I want to be able to individually change/extend the expiry date of User scratch directories on the Scratchpad. (Nice to have)
16. As a Maintainer, I want to be able to set the default and MTTL for User scratch directories.
17. As a Maintainer, I want unused and expired scratch directories to be automatically deleted.
18. As a Maintainer, I want Users to have to manually extend the expiry date of their scratch directories if they don't want their scratch directories to be deleted.

Maintainer – features to ease deployment & maintenance

19. As a Maintainer, I want to have documentation on how to set up and maintain the Scratchpad Manager.
20. As a Maintainer, I want to automatically have existing directories on the Scratchpad added as scratch directories to the Scratchpad Manager.
21. As a Maintainer, I want to be able to delete scratch directories off the Scratchpad, with the Scratchpad Manager handling the deletion gracefully.
22. As a Maintainer, I want to edit the system configuration while the system is running.
23. As a Maintainer, I want to be able to view any information about a configuration setting or value.

2.4. Technical Requirements

We have tagged the functional requirements to technical requirements and categorised them according to the MoSCoW method. Those that are not directly linked to any functional requirements, are either a desire of the client or part of the project description, indicated by the "PD" tag.

Must Haves

1. [FR 1, 2] The system must have a command to create directories on the Scratchpad.
2. [FR 3, 17] The system must be able to delete scratch directories on the Scratchpad.
3. [FR 4, 14] The system must keep track of existing directories and their expiration dates.
4. [FR 5, 17] The system must periodically check if a directory has expired.
5. [FR 5, 17] The system must delete a scratch directory when it has expired.
6. [FR 6] The system must create new scratch directories such that they have appropriate permissions, namely; only the creator can read & write its contents.
7. [FR 9] The system must have a command which allows users to extend the expiry dates of their scratch directories to the MTTL.
8. [FR 10] The system must send an email to notify the user when a scratch directory is about to expire.
9. [FR 11, 19] The system must have proper documentation.
10. [FR 18] The system must force users to manually extend their scratch directories' expiry date if they want to keep the directory on the Scratchpad.
11. [PD] The system must have a clear, well documented command-line interface (CLI).
12. [PD] The CLI must connect to the daemon in order to communicate with it.
13. [PD] The daemon must be able to identify the user using the CLI and allow users to only use the systems features on their own directories.
14. [PD] The system must secure sensitive data (if any) behind admin access permissions.
15. [PD] The system must not be able to provide admin only functions to normal users.

Should Haves

16. [FR 3] The system should have a command to manually delete scratch directories on the Scratchpad.
17. [FR 4, 8] The system should have a command which lets users view their scratch directories with their respective expiration dates.
18. [FR 14] The system should have a command which lets the maintainer view all directories with their expiration dates.
19. [FR 16, 22, 23] The system should have a configurations file containing settings for default MTTL, email and directory/file permissions.

Could Haves

20. [FR 2] The system could allow for the creation of multiple directories by the same user.
21. [FR 7] The system could have a command which lets users rename their existing directories.
22. [FR 9] The system could allow users to set their own custom expiry dates in their configuration file, that are under the limit set by the maintainer.
23. [FR 12, 13] The system could have a user config file which lets users configure their personal preferences for their email notifications about expiring directories.
24. [FR 12, 13] The system could have a user config file which lets users configure their personal preferences for the default expiry time.

25. [FR 12, 13] The system could have a user config file which lets users configure their personal preferences for the default permission mask for their directories.
26. [FR 15] The system could have a command which allows the maintainer to manually change the expiry date of users' scratch directories.
27. [FR 20] The system could scan the Scratchpad directory periodically to search for new unmanaged directories, and add them to the system as scratch directories.
28. [FR 20] The system could have a command to trigger a scan to search for new unmanaged directories, and add them to the system as scratch directories. (Later note: this is implemented within the `list` command instead of a separate command.)
29. [FR 21] The system could scan the Scratchpad directory periodically to check if any scratch directory does not exist anymore, and if a scratch directory is missing, remove it from the system.
30. [PD; FR 16, 22, 23] The administrator's configurations file could contain options for the default permissions and default group of scratchpad directories.

Will Not Haves

32. [OVER-ENGINEERING] The system will not have AI features.
33. [USING AND MAINTAINING SEPARATE] The system will not allow users to take the role of maintainer.
34. [OUT OF SYSTEM] The system will not have a make-directory-public (and make-private) command.
35. [OUT OF SYSTEM] The system will not be able to change scratch directories permissions, to set personal directories as read only or writeable for other users.
36. [OVER-ENGINEERING] The system will not send out emails warning for directory expiry that contain a (functional) extend button.

3. Design Choices

During the course of the project, we had to make several decision regarding the design of our system in order to make it safe, intuitive and convenient for both the users and maintainer. These choices were discussed in detail with our client, before being implemented, in order to make sure that our design matched his vision.

As described before, the client's desires resulted into certain choices already being made for us. To see the details, see Section 1.1.

Throughout our various meetings, the client stressed the edit-ability of the tool: both in configurable settings and possibility to alter the source code as admin.

The client left a lot of open room for expanding and implementing the required features. This led to several new features being added to the design of the system, and a lot of important decisions being made which severely impact usability and expendability.

3.1. Programming Language

We chose to use Python as our programming language for this project. The reason we chose this language is for it's simplicity and familiarity within the group. C++ was an alternative option but based on the scope of our project we deemed the trade-off in terms of speed and efficiency worth it to mitigate risk of errors and additional time needed to familiarise ourselves with the language. We chose for the version 3.11, because this is what is installed on the servers.

3.2. Framework, Libraries and Modules

Initial dependencies:

- Python-Daemon
- Lockfile
- Tomlkit
- Poetry

In our initial exploration of the requirements we established that we would require a daemon that would manage the back end. With help of our client we established we should instead use `systemd`, which the target systems already use. `systemd` will take care of the daemon requirements of our application. Switching to `systemd` allowed us to remove the dependencies `python-daemon` and `lockfile`.

`tomlkit` was used for some time to allow for reading and writing configuration files conform the TOML specification [2]. The client requested we didn't use `tomlkit`, as they found it unnecessarily complex, and instead advised us to use Python's standard `configparser`. However, we missed features such as exporting with comments; and type and input validation, so instead we implemented our own configuration reader, also based on the TOML language, which made the `tomlkit` dependency obsolete. `tomlkit` remains a dependency of Poetry and Pylint.

Docker was considered for creating a virtual environment, but later was deemed too complex and bloated. Instead, the developers using Windows, used Windows Subsystem for Linux (WSL) for development on windows computers, while the Linux users were able to run it directly. The reason we needed WSL is for the `python-ldap` dependency, which is not (easily) available on Windows, and Linux file permissions.

Crontab, a feature within Linux that allows for a program to run on a schedule, was among the early contenders for running our program. The reason we have not continued with crontab is due to it only running on an interval, this meant that interaction with the program from a front-end CLI was highly complicated.

Final dependencies:

- Python 3.11
- Poetry
- Argcomplete
- python-ldap

Poetry is used to manage dependencies, and manage them within the virtual environment of our project. python-ldap is used to facilitate the connection between our daemon and the university LDAP servers.

In consultation with our client, we chose Python 3.11 as the minimum Python version for our project. This is currently the Python version installed on all target systems.

Argcomplete creates argument completion code to be used by the target system's shell. This allows users to press `tab` to automatically complete words in the commands of the Scratchpad manager.

3.3. Command-line Commands

Besides the commands `create, extend, remove, help` that the client suggested, we decided to implement some more. We made a `rename` command, which allows users to rename their directories, as they otherwise do not have the permissions to do so. We created a `list` command which lists the directories that are owned by the user (or all directories for the maintainer) and their expiration dates. The final design for these commands can be found in Appendix B, which contains the man page. Note that there have been several iterations of sketches for designing things like the commands.

Initially it seemed like the `delete` command was already implemented out of system, through the regular Linux commands. However, once our system is integrated on the servers, the idea was to not give the users write permissions to scratchpad directory. As a result, they would be unable to delete their own directory except through expiry without the `delete` command. On the other hand, changing the read and write permissions of their personal directories is something that the users can do using Linux commands, therefore we did not implement this.

The `extend` command can both be used with a date and with a time period. For the latter we decided to approximate one month as 30 days, because this is the average duration of a month and that one day difference will not be of a lot of influence when the cleanup happens on the interval of once or twice a day.

We had some lengthy discussions about whether there should or should not be directories in on the scratchpad that are managed by our system (aka directories without expiry). In the end all directories on the scratchpad should be managed by our system, and should new directories appear that weren't made through our command-line tool, they should automatically be added with an expiry date when they are detected. To do this, our system checks whether the database is still up to date on a periodic interval that is configurable by the administrator, and also when `list` is ran. Originally, the idea was to make a separate command for checking the database against the local directory, but this may cause confusion

if the administrator adds a directory to the Scratchpad and then runs `list` before the system checks the database. In that case, the new directory would not show up until later.

Some commands behave different for the maintainer. For example, the `list` command will list the directories of all users when it is run by the administrator (root user), whereas for a regular user it would only list the directories they own with expiry dates. For most other commands, the administrator can run commands for other users, but does have more strict requirements for the command parameters. There is also an `ldap` command to set the username and password for using the LDAP server functionality (for looking up users' emails for the reminders) that only the administrator can run. There are some other small changes to the user commands and a handful of maintainer-only commands, which are explained in great detail in the man page of our system.

3.4. Database

Initially the system was designed to only support a JSON database, due to the relatively low amount of data that needed to be stored. However, we later decided that the option to switch between a JSON based database and a SQLite database would benefit the maintainer in the long run, since SQLite is faster and more reliable for larger amount of data, and this would allow the system to scale without the need for heavy modification. The switch occurs in the system's configuration, where the maintainer can also specify the path to the database file.

In the beginning the database only stored the directory name and the expiry date. We believed that this would be sufficient since we could access all other relevant information through the permissions of the directory. However during the implementation of the email system, we have discovered that without keeping track of the last email sent, multiple notifications could be sent on the same day with high frequency (depending on the administrator's configurations), which might be inconvenient to the user. To avoid this we have added a column in the database that keeps track of date of the last sent email.

Since we cannot stop the maintainer from modifying the database manually (assuming no malicious intent we can still expect human error) we are forced to handle any possible errors in the database manager, which we do.

There was a discussion during the implementation of the database about what the database manager should return after doing its periodical check on the database. We decided that it should return a list of directories that need to be deleted and a list of emails that need sending.

3.5. User Permissions

There is no distinction necessary between different kinds of users, as all users (student or employee) have the same rights and access on the servers and the scratchpads. Only the administrator needs special permissions, say access to the database, seeing everyone's directories and expiry dates, being able to alter other people's directory expiry times. We could have made a special kind of user for the administrator, but ultimately decided that we shouldn't for a couple of reasons. Instead the administrator functionality is given to the `root` user only. The administrator has root access, so they are perfectly capable of using it to get access to the administrator functionalities. This separation of user and maintainer functionality mitigates the risks of regular users getting elevated powers that they shouldn't have, or accidentally performing admin tasks while doing user things. This method allows us to use Linux' permission system instead of managing our own, which improves security.

3.6. Sockets

Users interact with the daemon by connecting through a Unix Domain Socket in order to create and delete files. These connections are created and managed by the socket manager which runs on a separate thread constantly listening for incoming connections.

With UD sockets, we can authenticate users by getting the so-called 'peercred' [6] of an established connection. This peercred is a struct of 3 integers; pid (process id), uid (user id) and gid (group id). This way, authentication is done using Linux permissions instead of a custom authentication flow, and the only way to become a more privileged user within the system, root, you'd need to gain root permissions in the whole system.

3.7. Threads

Threads are used minimally as they have only a small use case in our project. Threading's main application is for sockets, where the socket manager has its own thread to handle any incoming connections, and every connection has its own thread as well. Thread safety is mainly a concern for the JSON database, as it has been handled by the python built-in `sqlite3` library for the SQLite database,. The JSON database thus has it's own thread safety features implemented by us. It is done by having a read/write lock so only one process can read or write to the database at the time. This has to happen since one process writing to the file while another is reading, leads to corrupted input by the reading process resulting in a crash. Besides the r/w-lock, there is also an entry-lock, which is locked when doing non-constant operations, which change the state of the database. This lock ensures that any writes happen atomically.

3.8. Logging

The application runs 24/7, and acts on the file system. The maintainer might want to check what the application has done and when. To ensure integrity and accountability within the application, we want all changes to be logged properly.

We considered using `stdout`, as running the app as a `systemd` service will capture `stdout` in the `systemd` journal. For our client, this would have been sufficient. We decided this was not up to our desired standard for logging, as some systems are set up to remove logs every boot. Thus, we chose to add an option to log to a dedicated file.

3.9. Configurations

There were a couple times where we were deciding between two options that were both good, where we asked our client for input and he let us know we could choose ourselves or even better: do both and make it configurable. One example of this is the JSON and SQLite database.

While we were making things configurable anyways, we decided that there were a couple of things that would be nice for the user and administrator to set themselves.

As a user, it would be nice to determine when and how often you would get an email reminder to extend your directories lifetime. We decided that a user could set how many days beforehand they would receive a notification, with the maximum of 5 notifications. We initially also toyed with the idea to make it configurable to receive emails for other things than upcoming expiry, like a successful extension, but decided against this. The command line tool should provide enough feedback, and we do not want to spam the users with unnecessary emails.

For its implementation, we decided that the user configuration file needs to be read by the back-end, instead of being sent alongside the (relevant) commands. The back-end needs to be able to read the user configuration anyways, to get the email settings. This does mean that the place of the user configuration file is not configurable, unlike the daemon's configuration file.

The administrator can set the interval at which the database will be checked (for expired directories and emails). We recommend to check it at least every other hour, because if the email server is down during the check, the mails will get lost.

We already briefly explained the choices for implementing the configuration and reading and writing them using a configurations reader we made ourselves in Section 3.2. We contemplated creating functionality that allows users to edit the configurations file through the command line interface, but decided against this. Instead, we implemented a command that creates the configuration file for the user and a command that resets it to the default values. This also means we do not have the issue of having to reload the configuration through the command line.

3.10. Architecture of CLI and Daemon

In the design process, several approaches to the architecture were considered. Initially we considered a single program facilitating both the CLI and the periodic checks. This idea was later replaced by one where these two are separated and should interact through sockets. The reason the second approach has been selected is due it having a more clear separation of permissions, more readable code and it being easier to change and adapt it later (due to its increased modularity).

3.11. Constraint considerations

Within the program there are numerous situations where user input should have logical limits. We believe the main features that benefit from constraint are as follows:

- Expiration dates and extensions: Avoid never-expiring directories and past dates.
- Directory naming: Start with username, length should be handled by Linux.
- Quantity of user folders: Protect the server by preventing a possibly infinite quantity of folders.

The first and last constraints are configurable by the system maintainer, or can be disabled altogether. Their primary objective is to protect the server against damage from our program if it were to be used by a malicious user. The directory naming could by altered by altering the source code if desired.

3.12. Folder Naming Convention

The naming convention of folders was not part of the initial project. This was because there was only a single folder for every user. With the addition of multiple user folders there had to be a extension to the naming convention.

The initial convention was simply `username`, (a folder of John Doe would be DoeJ). To accommodate for the increase in user files an appending of names has been chosen. The format is `username-name`. This way folders are still clearly identifiable, simple and allow users to have more information about their projects in the folder title.

We are allowing folders with the name `username-username-name` to exists, but this leads to some perceived odd behaviour of the system (especially if `username-name` exists), and

managing that situation is up to the user. The manual should be of help, because it does follow the logic of the system's behaviour that was specifically tested for this situation.

3.13. Storage of Sensitive Data

The project does not handle much sensitive information, as user authentication is offloaded to Linux, and email address storage is offloaded to the LDAP server. Nevertheless, the LDAP server does require a login, so we do have to store one password.

We initially considered using the Linux kernel keystore for this, which can be controlled through the `keyctl` command, or programmatically through the `libkeyutils` C API [1, 3–5]. We used the persistent keyring to store the LDAP password. Only after implementing this functionality did we realise that the keyring would be garbage collected, thus "persistent" actually means "3 days", which means the key would be removed if it was not requested for 3 days. If the server ever went into maintenance for 3 days, the key would be removed, and email lookup would stop working.

When we realised this, we exchanged the use of `keyctl` with saving the password in plaintext file readable only by root in `/etc/`. This follows the key storage conventions of programs like `ssh`.

`scratchmgr` contains a command to set them LDAP password in this file, usable only by root, which ensures the file permissions are `600`: readable and writeable only by root. Furthermore, the program will check these permissions each time it reads the key file, try to fix and log a warning if the permissions are more permissive than expected.

4. Implementation

In this chapter we discuss the process of implementation, what choices we made whilst working and the approach we took to creating the different parts of the system.

4.1. Daemon

The main body of our system is the daemon. It's functionality encompassed everything that is required to handle the user input, and the general tasks of our system. The daemon has a main loop in which all tasks are handled in order (with exception of sockets, and socket handling). Within the code base all tasks are given their own directory, and files in which they operate (following the principles of modularity) The components we established are `command`, handling incoming commands. `config`, which is responsible for user and admin configuration. `database`, responsible for updating and checking the database. `email`, for finding when users require a email and sending it. `keys`, which is for handling LDAP credentials and `socket`, responsible for handling input coming from the CLI.

This clear separation of modules is both easier to read, and much simpler and clearer to implement.

4.2. Front-end / Command Design

The CLI design was initially directly based on the client requests. The list of commands as laid out in the project description was as follows:

- Create
- Remove
- Extend
- Help

These commands facilitate the core functionality of the project. In the implementation these commands were the first ones to be realised. In doing so, the needs for several additional commands arose, mainly to accommodate for options not covered by the current commands. The command `rename` allows users to rename their directory without removing and recreating a directory, which is both inefficient and risks losing files. `list` returns a list of all your directories. It followed from the possibility to create several directories as opposed to a single one. It also forces a scan for undiscovered folders (folders in the managed directory, but that are currently not managed by the system). `config` is a result of our system allowing for user configurations file, a feature not initially requested but expected to improve user experience.

4.3. Prototype

In order to ensure that the system design matches the expectation of the client, a prototype was created representing the minimum viable product. This version contains all the crucial features present in the original project description, other than clear documentation, since we planned on further expanding the system with additional features. In this prototype, only the `create` command had a formatter. The man page was still under construction. The `list` command was started, but not ready for showcasing yet. However, the `extend` and `delete` command were implemented, though there was no maximum on the extension period yet.

The prototype was presented to our client at the beginning of October, where we learned that our main functionalities were implemented as expected. The meeting also provided useful feedback on both user and maintainer configuration files, and led to a discussion on how logging was implemented. Here our client disagreed with the current implementation (of logging) and suggested some modifications. However since we believed that the current logger

was more suitable for the system, we have decided to meet the client half way and made it configurable. This way the clients needs were satisfied and we maintained the current logger.

The client was surprised that we already implemented the possibility to own multiple scratch directories, but this seemed to be a pleasant surprise. He liked the formatting of the directory names, which was nice, because we had some lengthy discussions about this as a group. We discussed the transfer of ownership of directories, and the options for the database and email settings. Additionally, we discussed the used libraries that were used, and some developer-environment related settings in our code that will be altered for the final product. We discussed the LDAP credentials, which we hadn't gotten to work before then. Lastly, we discussed the implementation of the user configuration at length: our client proposed a sample configuration file.

This was all very useful feedback to us, and we used it when we continued our implementation of the project.

4.4. Documentation and Manual

For documentation, we have a simple help menu, which is intended for general use and can be easily consulted. It contains syntax and short descriptions.

For detailed information about the system there is a man page available in several languages. These contain in-depth information about any system details that are meant to be interacted with (both by user and admin), such as configuration or setup.

We wrote the man pages directly in the Groff language used by the man command. We were aware that large man pages are frequently written in different languages, and then converted to Groff, but we decided to ignore this extra layer, as the size of the man pages is not very large.

5. Testing

Since the system deals with personal files, deleting them after a certain amount of time passes, it is important that every precaution possible is taken in order to avoid premature deletion and offer the user a chance to copy the files that they need. To achieve this the system needs to be thoroughly tested before deployment guaranteeing that it behaves exactly as specified by our client.

Ensuring the functionality of our system will also have a positive impact on the maintainer, since it reduces the risk of an unhandled error appearing or other unexpected behaviour. In the long run this leads to a trustworthy system that requires less maintenance, and thus it reduces the amount of work done by the maintainer.

In order to achieve this, a mix of automatic unit-tests and manual testing has been done, covering as much of our system as possible, taking into account every edge case we could think of.

5.1. Plan

For the system to perform to the current expectations we must asses the possible risks present, and develop a testing plan that will mitigate the current threats. Below the plan for assuring the integrity of system is listed.

5.1.1. Objective

The system aims to reduce the workload of the server maintainer, and thus the main objective is to deliver a program which can run twenty-four hours a day, seven days a week without any supervision. For this it is also important that the main functionalities of the system work correctly every time and there is no risk of important data being deleted prematurely, or without the user being notified in advance.

5.1.2. Risk Analysis

The nature of the system allows for a large number of risks, however, many of these are outside of the scope of our testing, due to the causes being outside of our control. As such we have identified the following risks (or scenarios that should not happen), that we must take into account when taking the necessary steps to test the system:

- Directories could get deleted before the expiry date provided.
- The user could not be notified of the upcoming expiry date.
- The system could encounter an unexpected error and shut down.
- Users could attempt to modify or delete other users directories.
- Data could be stored incorrectly and lead to the main functionality of the system not being functional.
- The system might be unable to create new directories and assign permissions.
- The system might try to access nonexistent directories.

5.1.3. Mitigation plan

In order to mitigate the identified risks, we implemented good programming practices and Gitlab practices. We wrote unit tests to test the commands in all the test scenarios we wrote out in a brainstorming document. This document contains about ten scenarios per command and their desired outcomes, see Appendix C. Besides this, we will do a “user” walk-through to see how things look in the CLI, as this is not tested in the unit tests. These should ensure

that there is a minimal chance of edge cases that have not been considered by the development team.

There is no user information that needs to be secure, so abuse of the system is the only risk we aim to mitigate. This might look like gaining elevated privileges, which we think we have sufficiently protected our application against through our use of root for administrator functionality.

5.2. Execution

Down below is a detailed explanation of the tests performed on each part of the system.

5.2.1. Database

The database is only accessible through the database manager by the daemon or manually by the maintainer. This allows us to assume that any input for the database manager is formatted correctly and valid, which means that only the functionality of the methods need to be tested.

Since the system could be using a JSON based database or an SQLite database, two unit-tests are created in order to assure that the methods of both return values in the same format. This will assure that the database manager will always have consistent and valid returns.

Each test has a set up which initialises it's database and sets up values that could later be used to verify results. At the end of each test the database is deleted in order to assure that previous test do not interfere with the current one.

Since the database is quite simple, the test manage to cover every function and check if they match the intended behaviour.

The email functionality has been integrated with the database after testing, but due to the simplicity of the function (one function with limited effects) we chose to exclusively user-test this in order to save time and energy.

5.2.2. Unit tests

The main form of testing during the project were unit tests. These tests are intended to interact with the daemon to determine if the responses it receives are correct. They focus on testing edge cases and potential security risks. Beyond that it simulates all system functionality individually for regular cases, both from admin and user perspectives. To see the full design for the tests, see Appendix C.

Most of the edge cases are brought on by the multiple directory naming scheme. Since we append the username to the beginning of each directory, we must verify the user input and handle it accordingly in order to avoid confusing names.

One example of this behaviour is the fact that the user might not understand that their username will be automatically appended to the beginning of the directory name, and as such will give it themselves. In order to avoid names such as `username-username` we have decided that, if the username is already given, we will remove it and instead append our own formatting in order to ensure consistency. Unfortunately this causes a waterfall effect on the rest of the commands, increasing their level of complexity as now we have to handle both cases when users give the full name of their directory and only the custom name they have chosen. As this behaviour could be quite confusing without reading the documentation, we had to handle it carefully, which is why many of the unit tests cover this type of behaviour.

The rest of the unit tests cover more common problems such as empty inputs, inputs outside the allowed range, and permission checks. These allow us to validate the security and integrity of our system. Here the maintainer commands should also be noted, certain commands have different behaviour if ran by an administrator, such as exceeding limits and overwriting permissions. As such we verify that this can only happen if ran with root permissions and no user can get access to elevated privileges without specific permission.

These test have proven to be of great importance, since they allowed us to properly handle some cases which were overlooked during the initial implementation, and they increased our confidence in the reliability and integrity of the system.

5.2.3. User tests

The scratch manager system is fairly restrictive. User usage can be clearly predicted as there is little freedom outside of the offered functionality. For this reason, user tests are largely automated. Using scripts we can simulate users issuing commands, requesting various things from the system both valid and invalid. Confirming to see if the interaction was successful is done manually by the development team.

The results from user testing revealed no new problems, most likely due to our extensive unit tests and relatively simple system.

It did reveal an edge case that could naturally occur, which we previously thought was impossible to reach. However, when `root` creates a directory for another user that already exists, it does throw an error. This is not the kind of critical error we initially thought it would be, when we considered regular users only.

6. Evaluation

6.1. Planning

This is also partially discussed in Section 1.5. We started with the requirements gathering and initial design, and we worked on this during the first two weeks of the project. During this time, we had two meetings with our client/supervisor, during which we were also informed that he would be absent during the last four weeks of the module. We planned to have all of our questions and concerns discussed while our supervisor was still available.

We valued flexibility over the stress of intermediate deadlines, so we set an order in which things had to be done and used the total picture vs. what was done to evaluate whether we were staying on track. We started out by setting up the foundations in Gitlab such that we could all branch off and start implementing pieces. During the implementation, we were always working on the design report, with a variable ratio. This way, we made sure that all of our requirements were up to date, and we would not have to rush the report at the end.

Things were immediately discussed with the team as they came up, which was one of the perks of our daily live meetings. Additionally, they also helped us motivate each other.

Overall, we stayed on track, besides getting into a small time crunch (as expected) at the end.

6.2. Final Result

We have managed to implement all of the functional and technical (with the exception of the 'Won't' category) requirements to our satisfaction. We would insert them again in a checklist format here, but we honestly do not see the point in taking up more space when you could look at Section 2.3 and Section 2.4 again and imagine them all checked off.

6.3. Contributions

Our team consists of six people with varying interests, strengths and expertises. We kept this in mind when dividing the responsibilities for this project, as this made the process both easier and more fun. The division of work is as listed below:

- **Bram:** Email, Poster, Report
- **Elise:** Requirements, Notes & Minutes, Report, Poster
- **Filip:** Database, Poster, Report, Slides
- **Mathijs:** Requirements, Testing
- **Matthijs:** Manual, Commands, Report
- **Melle:** Stub implementation, Email, Configuration, Commands, Deployment

Note that the designing, writing the report, preparing intermediate feedback sessions, and client communication were a group effort, meaning that we have all worked on this. It is worth noting that certain tasks that were performed by multiple people may vary in amount of contribution per person. Many things required group discussion and input, which has happened in plenty during our many face-to-face meetings and discord messages.

6.4. Collaboration and Communication

Collaboration within the group progressed smoothly throughout the entire project. We maintained several focus points of shared work. The programs we utilised are as follows:

Project environment

- Gitlab
- Typst
- Trello

Communication

- Discord
- Whatsapp
- Teams

Shared access

- OneDrive
- Google calendar

The working environment describes what programs we have primarily used for creating the software, report and the task division/planning. Gitlab has been used to allow for a shared working environment, with clear tools allowing smooth integration of the individual contributions. Typst has been used as a text processing tool rather than LaTeX, it offers a similar structure to LaTeX but has a simpler and more user-friendly syntax, whilst offering all required features that LaTeX does as well as online documentation. Within the group one person was more experienced with it as well, and everyone else had no preference, hence the choice. In hindsight, we were all very satisfied with this choice. We have also used Trello for getting a clear view of the tasks that need to be done. We mainly used it to keep track of To Do's. This might be a bit similar to how Trello is often used as a Sprint board, except we used one board throughout the entire duration of the project.

For communication between group members there are two main applications. Whatsapp has been used for simple and quick access messages, like notifications on delays for meetings or important changes in the schedule. Discord offers more functionality and has been used for logging our meetings, in-depth discussion about design and implementation, discussing feedback, and important links relevant to the project. Additionally, we had a Gitlab webhook, so we received notifications of changes to the Git through Discord.

Finally we have used OneDrive to share files. This is used as all group members have access to it through their university account. For quick meeting notes (especially at the start of this project) and design notes, Word worked fine. We were unfortunate enough to use online Powerpoint for our feedback presentations, as this was the easiest to quickly collaborate in. Additionally, we made a Google calendar to share our personal availabilities and our project meetings.

Using all of these tools allowed us to collaborate and communicate clearly and in an organised manner. Whilst some minor complications can always arise they could be solved quickly within the group.

Most notably, the main enabler of clear communication within our group, were our daily face-to-face meeting. Even when we were working individually or in smaller groups, being in the room together lowered the barrier to ask other team members questions about minor confusions. Additionally, it was very easy to make group decisions when we could immediately have a live discussion about it when the need arose.

6.5. Communication with Client and Supervisor

Since our project was provided from inside the university, the client and the supervisor were the same person. Contact was initiated on the day we received the project and we were able to meet that same week. During the first meeting we established how future communication will be handled, and more information on the project. After that all further digital communication was done through Microsoft Teams.

The next meeting was held the following week, where we were able to present a list of requirements and a couple of preliminary diagrams, through this we have gained a better understanding of the project, and we were able to clear some misunderstanding from our first meeting. During this we have also suggested some additional features that we could implement in order to deliver a better product.

During the following weeks we were working on developing a minimum viable product that we could present. During this time we were provided access to the server our program would run on, and some useful information on connecting our system to the LDAP in order to retrieve the email addresses of the users. At the beginning of October we were able to present our minimum viable product to the client, here we received confirmation that our current implementation was appropriate, and we had a discussion about certain choices we have made and possible changes we should implement. This meeting also provided more information on using LDAP and has led to the final breakthrough needed to make it work as intended.

In the following weeks the client/supervisor was unavailable, and as such we had no communication with him. However, we were confident that our design for the project met his expectations, and had a clear idea of how to proceed with our implementation. The final meeting took place on the 7th of November, where we showed our product and discussed the details of the final presentation, here we confirmed that our implementation meets the needs of our customer and we noticed some small issues that fortunately we were able to fix before deployment.

During our last meeting we did a final demo for our client/supervisor, and discussed the deployment of the system.

6.6. Follow Up

Something that remains to be done is the integration of our system into the servers. We would have liked to do the deployment and testing ourselves, but due to our client/supervisor being absent for the last four weeks of the project this was not possible.

The approach that we have taken is to facilitate a simple transition into the new system. There is a bundled zip file that you can download, which we recommend. A second option is the install script in the master branch, but Gitlab will probably fight you if you go this route. In the transition process we continue to assist in case this might be necessary.

Our system is designed to be a simple and elegant solution, as it only fulfils a minimal task within the overall server. Within the timeline we have not only finished the essentials, but also all additional wishes that the client has discussed with us.

This leaves us to conclude that only minimal future planning is required. Future plans should have a focus on software compatibility, such as ensuring that our system is compatible with more modern python versions.

6.7. Conclusion

To conclude this project, we would like to say that the final product has been delivered in a finished and well-polished state. During the entire project we actively designed and redesigned to achieve an optimal user and admin experience, whilst maintaining a clear and well-made system.

We believe this is thanks to our proactive stance towards design and collaboration. By having clear goals with a solid understanding of what works best in theory, we managed to also apply the theory in practice.

References

[1] Kan Li Kijewski. 2016. Make syscall in Python. Retrieved from https://stackoverflow.com/questions/37032203/make-syscall-in-python

[2] Tom Preston-Werner. TOML: Tom's Obvious Minimal Language. Retrieved from https://toml.io/en/

[3] The Linux man-pages project. 2014. keyctl(3) - Linux manual page. Retrieved from https://www.man7.org/linux/man-pages/man3/keyctl.3.html

[4] The Linux man-pages project. 2014. keyutils(7) - Linux manual page. Retrieved from https://www.man7.org/linux/man-pages/man7/keyutils.7.html

[5] The Linux man-pages project. 2025. keyctl(2) - Linux manual page. Retrieved from https://www.man7.org/linux/man-pages/man2/keyctl.2.html

[6] Troido. 2019. Different order for getsockopt SO_PEERCRED in Linux and OpenBSD. Retrieved from https://unix.stackexchange.com/questions/496577/different-order-for-getsockopt-so-peercred-in-linux-and-openbsd

A Use Case Diagrams

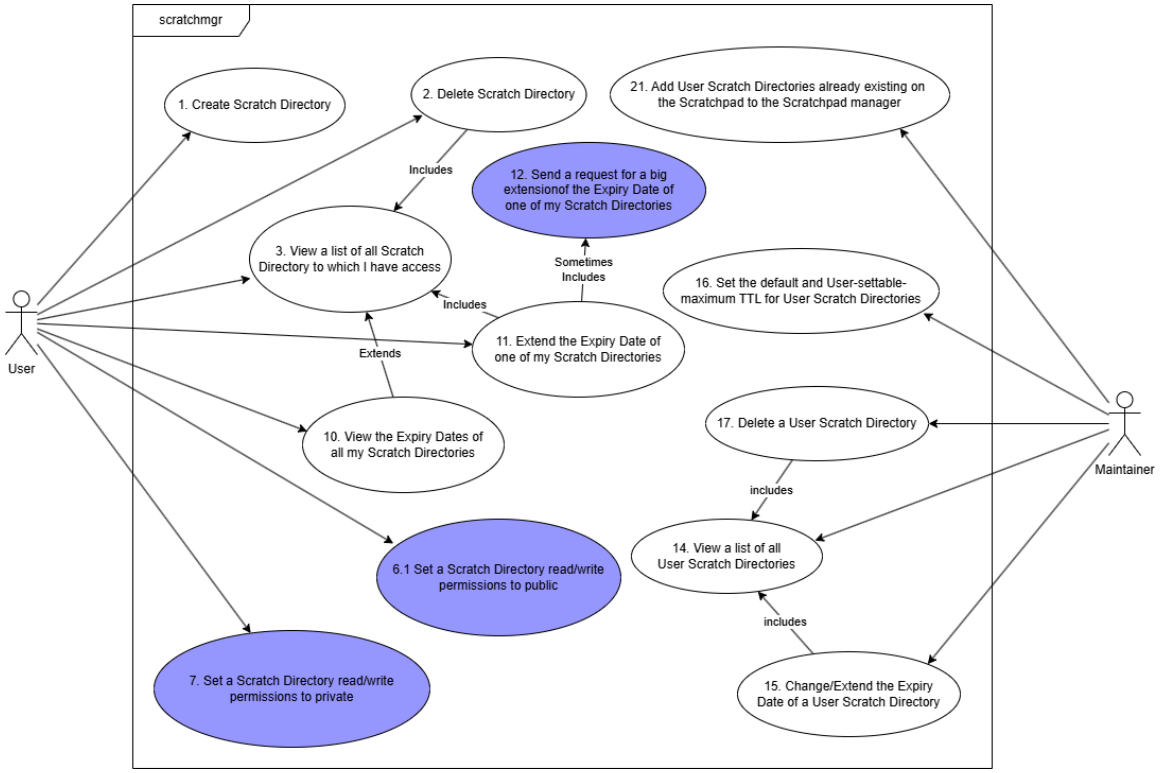

Figure 2: The use case diagram made for the initial design.

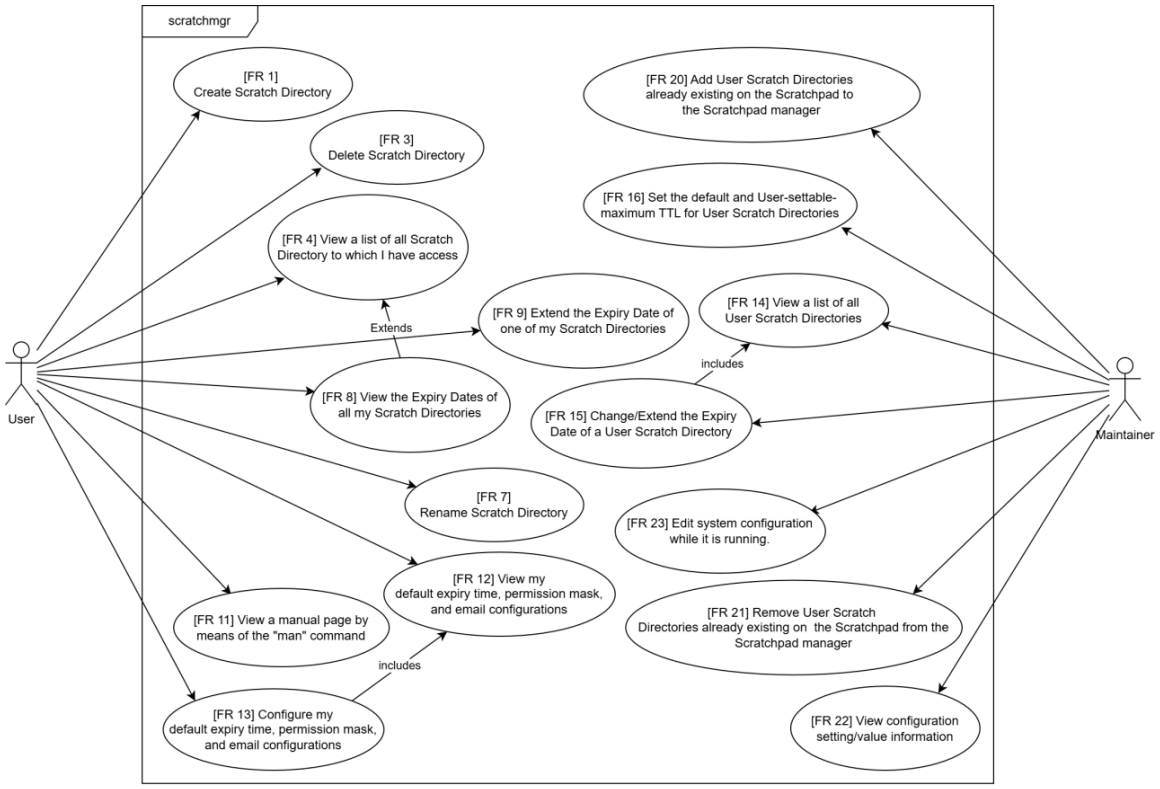

Figure 3: The use case diagram for the system that reflects all changes. The Maintainer can also do all User operations.

B Man page

Below is the text from the man page, as this reflects the final command functionality the best.

NAME

scratchmgr − Command-line management tool for Scratchpad directories

SYNOPSIS

scratchmgr [OPTIONS...] COMMAND [PARAMETERS...]

DESCRIPTION

scratchmgr is the command-line management utility for the Scratchpad. The Scratchpad is a temporary local storage space for anyone using this server. The space is intended as a faster alternative to the network drive which contains the /home directories. Once you are done, you are supposed to clean up your temporary files.

Because this was not always done by everyone, this tool has been developed. The **Scratchpad Manager** will automatically delete a Scratchpad directory after a certain amount of time. This tool, **scratchmgr**, is your interface with the Scratchpad Manager. You can create, remove, and extend the expiry date of your Scratchpad directories.

Throughout this man page, the following terminology is used:

The **Scratchpad** refers to the directory managed by the Scratchpad manager. It is usually located at /local/scratch. You can query the location of the Scratchpad using the **list** command, or when creating a directory.

The **Scratchpad Manager** refers to the daemon in control of the Scratchpad. Some settings can be set by the administrator, which change the default and maximum time-to-live of your directories, how many directories you can create, etc.

Your **personal directory** refers to the default directory created by the Scratchpad Manager. The name of this directory is your username with nothing else added. It is the shortest name you can give to a Scratchpad directory. Unless a directory name is provided, all commands will be performed on your personal directory.

The **command-line tool** is an alternative name for **scratchmgr**, the command about which you are currently reading the man page.

Commands have slightly different behaviour when run as the **root** user. **root**−only commands and different behaviour is explained in the *Administrator-only* section of this man page.

COMMANDS

create [*directory*]

Create a new directory on the Scratchpad.

If *directory* is not provided, your personal directory will be created.

If *directory* is provided, and assuming the directory does not already exist, a directory with that name will be created on the Scratchpad. It is not possible to create directories outside the Scratchpad with this command.

It is possible the Scratchpad manager has a directory limit. If you create more directories than allowed, the command will not create your directory and inform you that you've reached the limit.

Directory names must start with an alphanumeric character, and can additionally contain '−', '_' and

'.' except as the first character. No spaces or other special characters are permitted.

The directory name will be prepended with your username. If your username is 'user', the commands
$ scratchmgr create dir
and
$ scratchmgr create user-dir
will both create a directory with the name *user-dir*.

The directory's expiry time will be set automatically. If you wish to change it, this can be done with the **extend** command after the directory has been created.

extend [−−by *duration*] [−−until *timestamp*] [−y|−−no−confirm] [*directory*]

Extend the expiry date of one of your Scratchpad directories.

If *directory* is not provided, your personal directory will be used.

You can provide a directory name both with and without your username in front.

Directory expiry dates cannot be set in the past, or very soon in the future. Expiry dates can also not be set indefinitely in the future, but this depends on the manager configuration.

If you set the new expiry date earlier than the current expiry date, you will be asked for confirmation. If you wish to not be asked for confirmation, you can disable the prompt with the *--no-confirm* flag.

If neither *until* nor *by* are provided, the new expiration date will be set to the default time-to-live from the current time. The default time-to-live depends on the manager configuration.

until takes an absolute date, without time, in the formats YMD, DMY, or DM, where each part can be optionally separated by '−', '/', or '.'. Where D and M are the day and month, respectively, as a 2 digit integer. Y is a full year with 4 digits. The DM format will assume a date in the current year if that date has not yet passed, or in the next year if it has.

Examples:
2025−09−25, 25092025, 25.09, and 25/09/2025 are equal.

by takes a duration of days, weeks, or months in the form '<digit>{d|w|m}'. The new expiration date is calculated from the current expiration date.

Examples:
$ scratchmgr extend --by 2d - Extend by 2 days
$ scratchmgr extend --by 5w - Extend by 5 weeks

remove [−y|−−no−confirm] *directory*
Alias: **delete**

Remove a Scratchpad directory.

Contrary to most other commands, *directory* is a required argument.

You can provide a directory name both with and without your username in front.

You will be asked for confirmation before your directory is removed. To disable the confirmation prompt, you can use the *--no-confirm* flag.

rename [*current-name*] [*new-name*]

Rename one of your Scratchpad directories.

The first parameter must be the name of an existing Scratchpad directory. The second parameter must be the name of a non-existing Scratchpad directory.

With either parameter, you can choose to include your username in the directory name, but this is not required.

If only one parameter is provided, the rename operation will be performed your personal directory. If your personal directory exists, and the provided parameter does not, your personal directory will be renamed to the provided name. If the provided parameter is an existing directory, and your personal directory does not exist, the existing directory will be renamed to your personal directory.

Examples: (In the following examples, your username is *user*)
$ scratchmgr rename dir1 dir2 − Renames*user -dir1* to *user-dir2*
$ scratchmgr rename dir1 | If the directory*user* **does** e xist:
`−> Renames*user* to *user -dir1*
| If the directory *user* **does not** exist:
`−> Renames*user -dir1* to *user*

list

List your Scratchpad directories and their expiry dates. Will also print the location of the Scratchpad.

config *subcommand*

Subcommands:

reset
Alias: **default**

Reset any values in your personal configuration file to their defaults. This command does not add any missing values to your configuration file.

generate [−−*hard*/−*f*/−*H*]
Alias: **gen**

Generate a default user configuration file with all settable values present. If the user configuration file already existed, every correct value is kept. Every missing value is set to their default value. Every unrecognised key is removed.

If −−*hard* is given, present values will also be reset to their defaults. This makes
$ scratchmgr config generate --hard
equal to running
$ scratchmgr config generate
followed by
$ scratchmgr config reset

Administrator-only commands are listed at the bottom of this man page.

CONFIGURATION

Users can set a few options for **scratchmgr**. This is done by creating a user configuration file, and overriding any values you want to change. For any values missing from the configuration file, the default value will be used. The default value mentioned in this man page may differ from the actual value set in the manager configuration.

The user configuration file must located at*˜/.config/scr atchmgr.toml* where ˜ is your home directory.

The file syntax is TOML. You can generate a template file by running
$ scratchmgr config generate
If you run the command when you have an existing configuration file, all valid keys that were not yet present in your configuration will be added. Existing values will remain as-is, unless they are invalid. Unrecognised keys will be removed.

Note that values are typed. String must be in quotes, integers without quotes, arrays in square brackets, etc. For more information about TOML syntax, refer to https://toml.io/en/. Also note that file paths are treaded as strings.

The values you can override are:

email.frequency − (array[int]) Expiry warning email frequency
Default: [1, 3, 5]

Before your Scratchpad directories expire, you will receive an email warning you that, and when, your directory will be deleted. With this setting, you can set how many emails, and how many days in advance, you will receive emails. The list must contain no duplicate values, and contain no more than 5 elements.
Examples:
[] - Receive no email notifications
[3] - Receive 1 email, 3 days before your directory expires
[1, 4] - Receive 2 emails, one 4 days, and one 1 day before your directory expires

user.default_expiration_length − (string) The default time-to-live used by **create** and **extend** without arguments.
Default: 7d
This value cannot exceed the maximum allowed time-to-live of the system. The value must be in the format of the **extend** '−−by' parameter.

user.default_permission_mask − (octal integer) The permissions to use for newly created Scratchpad directories.
This only affects newly created directories. For existing directories, you can change the permissions with Linux' **chmod** command. You can optionally add the octal prefix, so*775* and *0o775* are equal. For more information about possible values, check 'man chmod'.

logging.log_file − (file path) Write scratchmgr logs to a file. If this setting is empty, logging to file will be disabled.

logging.log_level − (string) The logging level of scratchmgr
Valid options: DEBUG, INFO, WARNING, ERROR, CRITICAL This log level only applies to file logging. The console log level is set using commandline arguments.

ADMINISTRATOR-ONLY DIFFERENCES

When using the command with user **root**, most regular commands behave differently, and some additional commands become available.

Regular commands with root differences

create:
- *directory* is a required argument.
- Directory names must be typed in full.
- The maximum directory amount limit does not apply.
- Directories for any user can be created. If the directory name starts with a valid username, that user will get ownership of the newly created directory. If the user is not recognised, the directory will be owned by root.

extend:
- *directory* is a required argument.
- Directory names must be typed in full.
- Directories for any user can be extended.
- The maximum time-to-live limit does not apply.

remove:
- Directory names must be typed in full.
- Directories for any user can be deleted.

list:
- An extra column is added to the list table, showing the owner of the directory.

rename:
- Both directory names must be supplied.
- Directory names must be typed in full.
- This command does NOT automatically change the ownership of the directory.

Additional commands

config reload:
Reload the daemon configuration file from disk.

config save:
Write the current daemon configuration to disk.

config global-reset:
Reset the daemon configuration back to defaults.

ldap *username password*
Set the LDAP username and password used for fetching users' email addresses.

The LDAP credentials will be stored in a persistent keyring in the kernel, managed by **keyctl**.

C Unit tests commands

Create

User:

- create new directory without custom name
- create new directory with custom name
  - check formatting when username is included in custom name
  - check handling when other username is given in custom name
- create new directory with custom name === username
  - check formatting when custom name === username
  - check formatting when custom name == username + '-'
  - check handling when other username is given in custom name
- failure to create directory which already exists
- failure to create directory with custom name which already exists
- failure to create directory with invalid custom name
  - check what happens when the username contains invalid characters
- failure; reached maximum number of scratch directories
- failure; required parameter missing
- failure; too long a name (for Linux)

Maintainer specific:

- create new directory which exceeds the max directories of a user
  - check if ownership is of user
- create a directory without username
  - check if ownership is of root

Extend

User:

- valid extension; no target directory provided
- valid extension; target directory provided
  - check handling when username is given for target directory
  - check handling when other username is given for target directory
  - check handling when the target directory name contains more '-'
- valid extension; target directory === username provided
  - check handling when target directory == username + '-'
  - check handling when other username is given for target directory
- valid extension; duration
- valid extension; expiry date
  - check handling when expiry date is less than current expiry date
- invalid extension; provided extension duration is negative
- invalid extension; provided extension duration is too long
- invalid extension; provided expiry date is in the past
- invalid extension; provided expiry date is too late
- failure; target directory does not exist

Maintainer specific:

- valid extension; full target directory provided
  - check handling when the target directory name contains more '-'

- valid extension; directory === username provided
  - check handling when target directory == username + '-'
- valid extension; no extension limit
- failure; no target directory provided (otherwise we wouldn't know which directory to extend)
- failure; target directory not including username provided
- failure; target directory with non-existent username provided

General:
- failure; required parameters are missing

List

User:
- correctly lists all directories only owned by this user
- check if different users indeed get different lists of directories

Maintainer:
- correctly lists all directories in the database

Remove

Users:
- failure to remove directory without providing custom name
- remove directory with providing custom name
  - check handling when username is given for target directory
  - check handling when other username is given for target directory
  - check handling when the target directory name contains more '-'
- remove directory with providing custom name === username
  - check handling when target directory == username + '-' (should return a non-existence error)
  - check handling when other username is given for target directory
- failure to remove directory which does not exist
- failure; required parameters missing
- remove directory with confirmation

Maintainer specific:
- remove directory with providing custom name including username
  - check handling when the target directory name contains more '-'
- remove directory with providing custom name === username
  - check handling when target directory == username + '-'
- failure to remove directory without providing custom name (otherwise we wouldn't know which directory to remove)
- failure to remove directory without providing custom name including username
- failure to remove directory which does not exist

Rename

One Argument test (only the directory field is received)
- rename directory; target directory provided, default exists, target does not
- rename directory; target directory provided, target exists, default does not
- failure to rename; target directory provided, target & default exist
- failure to rename; target directory provided, target & default don't exist

- failure to rename; target directory provided is default directory
- failure to rename; target directory name is invalid
- failure to rename; maintainer must provide the new_name argument

Target directory tests (provide valid new name):

- rename directory; target directory provided
  - check handling when username is given for target directory
  - check handling when other username is given for target directory
  - check handling when the target directory name contains more '-'
- rename directory; target directory provided === username
  - check handling when target directory == username + '-' (should return a non-existence error)
  - check handling when other username is given for target directory

New name tests (provide valid target directory):

- rename directory; new name provided
  - check formatting when username is included in new name
  - check handling when other username is given in new name
  - check handling when the new name contains more than one '-'
- rename directory; new name provided === username
  - check formatting when new name === username
  - check formatting when new name === username + '-'
  - check handling when other username is given in new name

Failures:

- failure to rename a directory which does not exist
- failure to rename a directory to an already existing new name
- failure to rename a directory to itself
- failure to rename a directory with an invalid new name
- failure to rename; argument fields are not present or 'directory' is None or empty string
- failure to rename a directory to a name which is too long (for Linux)

Maintainer specific:

- rename a directory starting with a username which is not the username of the owning user
  - check that the owning user has not changed
- rename a directory to a name not including a username
  - check that the owning user has not changed
- failure to rename a directory to a name which already exists
- failure; new_name provided in the form of 'someName-'
- failure; new_name provided in the form of '-someName'
- failure to rename a directory which does not exist