DocReformed Technical Report

Ricky Atkinson, Connor Blaha, Dakota Deets, Austin Joseph, James Markijohn

Department of Computer Science

College of Arts and Sciences

Kent State University

November 16, 2025

CS 49999: Capstone Project

Dr. Augustine Samba

Table of Contents

Introduction

DocReformed is a print management software that provides users with a simple interface for inputting a PDF file, splitting the file into smaller pieces, and sending the resulting files to printers. This software's name is derived from the legacy print management software that it is replacing, Doc Transform. Doc Transform was used by this project's stakeholder, CPSstatements.com, which is a statement printing and mailing company based in Canton, Ohio. The stakeholders requested a modernized software with a redesigned yet familiar user interface, compatibility with Windows 11, and many of the features provided by Doc Transform, along with other requirements and constraints.

1.1. Problem Statement

CPSstatements.com has legacy print management software that cannot run on any version of the Windows operating system later than Windows 7. This software breaks large PDF documents (tens of thousands of pages, sometimes over one hundred thousand) into smaller pieces (one thousand or so pages), and can be used to send print jobs to printers on the local area network. CPSstatements.com requested new software that their in-house software development and information technology teams can maintain themselves, which needed to run on systems using Windows 11 as the operating system, along with future Windows versions. The legacy print management software that CPSstatements.com used, called Doc Transform, is essential to their daily operations, which involve mass printing of very large documents to mail out on behalf of their clientele. Since Windows 7 has not been supported by Microsoft for several years, a new software with the same capabilities that can be maintained into the future and run on a modern operating system was required to ensure the continuity of CPSstatements.com's daily business operations.

The customer of this project is a company called CPSstatements.com (hereon called CPS), a statement printing and mailing company based in Canton, Ohio. The primary liaison between the development team and CPS was Kyle Richmond, Vice President of National Sales & Client Care at CPSstatements.com. The customer requires an updated version of their legacy print management software (described above), which is critical to ensuring that their daily printing operations can continue. This is also necessary to fulfill the statement mailing requirements of their clients nationwide. Other stakeholders include the company's clients, who require statements to be printed and mailed promptly as part of their Service Level Agreement (SLA). Companies that offer similar services can be considered competitors to CPS. CPS's services have a direct impact on the revenue cycles of their clientele, hence why timely printing of statements is crucial to this company.

Employees working at CPS will be using this software. Two or three employees will be using this software per day. It will be consistently used throughout their entire shift(s). Users will be split into two categories: standard users and administrators (see section 1.3).

A major risk with the system involves the nature of the data that will be input into it. The documents that need to be printed contain financial, personally identifiable information (PII), and protected health information (PHI) related to the clients of CPS, as well as the customers of those clients. Handling of such information is protected under certain regulations, such as HIPAA for PHI. This system must be designed such that the confidentiality and integrity of the data are maintained.

1.2. Objectives

This project's main objective was to develop a replacement software for Doc Transform, which the authors have dubbed DocReformed. This software needed to fulfill several requirements to meet the project's primary and secondary objectives.

One primary objective of this project was to develop print management software that contained all of the core features of Doc Transform. These features were understood by the authors to be PDF splitting functionality, PDF archiving functionality, print job creation, and print job management (such as queuing jobs, moving jobs, and deleting jobs).

Another primary objective was to improve the user interface and, by extension, the user experience of DocReformed when compared to Doc Transform. Doc Transform featured many utilities that were not regularly used by CPS and were therefore deemed by the stakeholders and developers to be unnecessary in DocReformed. The user interface design of DocReformed aimed to be familiar enough so as not to confuse new users, but streamlined enough to eliminate unnecessary cognitive load from users. All changes were explicitly demonstrated to and approved, if not outright requested, by CPS; none were made unilaterally by the authors.

A secondary objective of this project was to add optional or low-priority features to DocReformed. An example of this is the ability to split PDFs into different sized pieces, not just the default of one thousand pages. Doc Transform is permanently set to one thousand pages, and while this is an acceptable value for CPS, they still expressed interest in being able to change this value.

1.3. Stakeholders and Users

The primary stakeholders of this project are CPS and their clients. Since this print management software is necessary for CPS to fulfill the terms of their Service Level Agreement (SLA) with their clients, the clients are considered secondary stakeholders in this project.

Employees working at CPS will be using DocReformed. On average, two or three employees will be using DocReformed per day. It will be consistently used throughout their entire shifts. Users will be split into two categories: standard users and administrators. Standard users of DocReformed will predominantly have backgrounds in business or finance, and will prefer familiar and simple UIs. These users will be responsible for managing how the statements are printed, at what time, in which order, etc. The administrators and users responsible for maintaining the software will be experienced with computer programming and information

technology. Such users will be able to perform any task that the standard user can, along with administrative/maintenance tasks such as adding and removing printers.

Another category of users will be the developers. CPS has an in-house software developer who will be maintaining DocReformed after it is officially released. The developer will be expected to have competence in the fundamentals of computer science and programming, and should be able to use, test, troubleshoot, and make changes to every aspect of the DocReformed application.

See section 4 for specific use case scenarios.

1.4. Constraints

Several constraints were provided by the stakeholders of this project. First and foremost, the software being developed needed to be capable of running on systems that meet the minimum system requirements for Windows 11. Furthermore, DocReformed needed to be compatible with the latest version of the Windows 11 operating system. This particular constraint was critical to the success of the project, as the replacement of the legacy print management software used by CPS was predominantly necessitated by its incompatibility with any version of the Windows operating system beyond Windows 7. The developer at CPS also requested that this application be developed using the C# programming language, specifically the .NET Framework version 4.8. Additional constraints were set by the functional requirements specified by the stakeholders (see the problem statement in section 1.1).

Aside from technical constraints, CPS also provided several constraints related to the user interface and user experience designs. It was explicitly stated that the new user interface needed to resemble that of Doc Transform, as it would be more convenient for users who had been using Doc Transform for several years. A more familiar user interface would maintain a positive user experience. Additionally, certain options and features offered by Doc Transform were deemed unnecessary by the stakeholders and developers, and subsequently not included in DocReformed.

2. Elicitation Plan, Assumptions, Risks, and Ethics

This section is dedicated to the techniques used during requirements elicitation, assumptions made by the development team, and risks associated with DocReformed. Much of this information pertains to the requirements of the project, which were discussed at length in section 1.

2.1. Elicitation Plan

The authors employed several techniques to elicit requirements from the stakeholders at CPS. This included regular weekly telephone calls, asynchronous email communication, and occasional text messages. During meetings, the authors always attempted to remain courteous to the stakeholders when proposing new or different ideas, and listened carefully when the stakeholders gave answers or feedback. Table 1 below shows some examples of questions and responses received by the authors during elicitation, which shaped how DocReformed was developed. A great deal of importance was placed on asking the right questions. All meetings were held with Kyle Richmond of CPS, though other members of the team would provide additional insights as needed.

These elicitations clarified several ambiguities regarding DocReformed's development and requirements, which ultimately led to a more refined final product.

Questions to Stakeholders and Stakeholder Answers	
How many users will be expected to use this software, and how often?	Two or three users will be using this software at maximum. This software will be used every business day at CPSstatements.com.
How exactly will this software be used in day-to-day operations?	The user will input the PDF file into a shared folder on the network. There are several such folders, each associated with a different printer. The print management software will poll these folders continuously and, upon finding a PDF file, will retrieve the file, split it into increments of no more than 1,000 pages, and queue print jobs for these increments. These jobs are sent to the appropriate printer. Finally, a copy of the original file will be sent to an archive folder on the network, which acts as proof of printing and serves compliance purposes.
What information needs to be stored in this archive?	Just a copy of all PDF files that are input into the print management software. An employee will categorize the documents by quarter based on when they were added to the archive. No other information related to the PDFs or print management software needs to be archived or logged.
Will any authentication be required to access this print management software?	Workstations running this software already require a login, so authentication is not required. However, the print management software could have an administrative console created to allow certain settings (such as page split size), which would require authentication.
To clarify: if the user inputs a PDF into a shared folder on the network, which has a name associated with a specific printer, the print management software will queue jobs for that respective printer?	Yes, although the user can move the queued jobs to a different printer as needed.

Table 1: Examples of questions provided to the stakeholders and their corresponding answers.

2.2. Assumptions

During the course of DocReformed's development, as many ambiguities as possible were removed from the project through meetings with stakeholders and elicitations. The greatest assumptions made by the developers were that the users would be able to use the final product and navigate the user interface comfortably, based on the frequent screenshots sent to them through development. It is also assumed that the provided user manual, which was delivered to the stakeholders alongside DocReformed, will be sufficient to explain the differences between the use and features of DocReformed when compared to Doc Transform.

It has also been assumed that CPS will rarely upload very large PDF files, with page counts exceeding one hundred thousand. While the stakeholders did say that this was possible, they did not describe this as a common event. See section 2.3 for discussion on the performance impacts such a large PDF may have.

The final assumption is that the users of DocReformed at CPS will not attempt to simultaneously queue jobs to the same printer. While this will most likely not cause an error on either the DocReformed system(s) or the destination printer(s), it will print documents in an unexpected manner. The user manual provided alongside DocReformed mentions this possibility and recommends avoiding this scenario.

2.3. Risks

The problem statement (section 1.1) describes some risks identified during DocReformed's development, specifically with regard to how PII and PHI are handled. Many of the development team's security concerns were addressed by the network design of the local area network within the CPS building, which contains several devices that are completely isolated from the internet. These devices will be used to host DocReformed.

Given that this software will be installed on workstations that are part of an isolated local area network with no wireless or internet access, many common cybersecurity threats are already mitigated. CPS informed the authors during development that each workstation requires authentication to access, meaning that, so long as standard best practices regarding password security are followed, DocReformed cannot be accessed by unauthorized users.

Any PDF documents entered into DocReformed will be deleted from temporary storage on the local system once the print jobs have been sent to the appropriate printers. This will prevent users from being able to access whole or split PDF documents within temporary storage. The documents entered into DocReformed will be archived for compliance purposes.

Other software-related risks include performance issues when very large PDF files (hundreds of thousands of pages) are uploaded to DocReformed. This scenario is not likely to cause any harm to either the system on which DocReformed is running or the destination printers, but there is a risk of the PDF splitting process taking a great deal of time to complete. See section 2.2 for more information on this scenario, and section 3.3 for a description of the PDF splitting process.

Several potential security issues were identified for DocReformed. These are discussed in section 6.4, but in summary, are primarily limited to network-based attacks or PDF-based malware. CPS did not highlight security as a major concern in the development of this software, and as a result, much of the task of securing DocReformed depends on those at CPS who maintain the network infrastructure and receive the PDFs that will be uploaded to DocReformed. These security concerns are mentioned in the user manual, along with recommendations for mitigating them.

2.4. Ethics

The developers adhered to several principles defined by the Association for Computing Machinery (ACM) Code of Ethics and Professional Conduct. The development of DocReformed was performed using responsible programming techniques, including those related to error recovery and security. DocReformed is not meant in any way to cause harm or create an undue risk on the computer system or network upon which it is installed. This satisfies point 1.2 of the ACM Code. Point 1.3 was satisfied through the frequent and transparent meetings between the development team and Kyle Richmond of CPS, where both parties were very honest about their expectations about DocReformed and what it should or could be capable of. At no time did the developers use DocReformed's development to speak on behalf of CPS. The development team met point 1.4 by never discriminating against their fellow developers or the stakeholders on the basis of age, color, disability, or any other inappropriate factor as listed in the ACM Code. The comprehensive user manual and simple user interface design of DocReformed was intended to make the software as accessible as possible for both new users and those familiar with Doc Transform. Point 1.5 was met by finding a license for DocReformed (MIT) that clearly defined ownership of DocReformed and the rights that both the developers and CPS had pertaining to modification and distribution of the software. Privacy and confidentiality, points 1.6 and 1.7 of the ACM Code, were also met to the best of the team's ability. No PDF files are being processed irresponsibly, sent to external sources, or modified in any way beyond what is expected by the end user. All of these aspects together satisfy point 1.1 of the ACM Code.

Points 2.1, 2.2, and 2.3 were satisfied by the development team in meetings with each other and CPS, in asynchronous correspondence, and in individual efforts. All parties involved maintained a courteous level of professionalism at all times. Those who were less competent in certain areas of the development process were either assigned tasks that suited their abilities better or made efforts to learn (this also satisfied point 2.6). Rules and regulations were considered and adhered to during development where necessary. Section 2.4 was met during lab meetings and asynchronous correspondence whenever the development team reviewed the works of others. This was done in the form of one member watching another write code, or through another member approving a merge request in the team's GitHub repository. Risk analysis and evaluations of the DocReformed application were performed throughout this project, the results of which can be seen in this report, and therefore point 2.5 is met. 2.7 and 2.8 were not relevant to this project, as the work was not being done for the sake of public awareness or educational

purposes. However, a comprehensive user manual was provided to CPS, which was meant to improve CPS's understanding of DocReformed and its functions. Point 2.9 refers to usability and security, which are described in more detail in section 6.4 of this report.

The development team cycled the role of team leader between each member, each serving for approximately three-week periods. The team was led by Austin Joseph from August 21, 2025 to September 6, 2025, by Dakota Deets from September 7 to September 27, by Connor Blaha from September 28 to October 18, by Ricky Atkinson from October 19 to November 8, and by James Markijohn beginning November 9; the latter will lead the project until November 29. Each team leader abided by the ACM Code's standards for project leads by creating a very open and supportive work environment that was always receptive to new ideas or questions (point 3.3). The leaders also made efforts to understand the ramifications of replacing Doc Transform with DocReformed (point 3.6), and took great care to integrate DocReformed into CPS's infrastructure as seamlessly as possible (point 3.7). Point 3.5 was satisfied by always providing learning opportunities, resources, or training to members of the development team who did not understand how a particular aspect of DocReformed's design or systems worked. Leaders applied the principles of this code even after or before their terms began, thus satisfying point 3.4. Again, since DocReformed was designed for a single company, points 3.1 and 3.2 were not relevant as pertains to the public good or to the cause of advancing the understanding of computing principles.

As a result of the adherence to the ACM Code, the authors are not aware of any significant ethical issues pertaining to the development, distribution, or operation of DocReformed, nor do they believe that CPS will encounter any such issues during their efforts to maintain and use DocReformed.

3. System Overview and User Interface

This section will describe an overview of the DocReformed software, along with descriptions of the functionality of the three major subsystems: the polling system, the job creator, and the print manager. Finally, the user interface will be described, along with examples. See section 4 for descriptions of use case scenarios, and section 5 for more details on the system architecture and implementation.

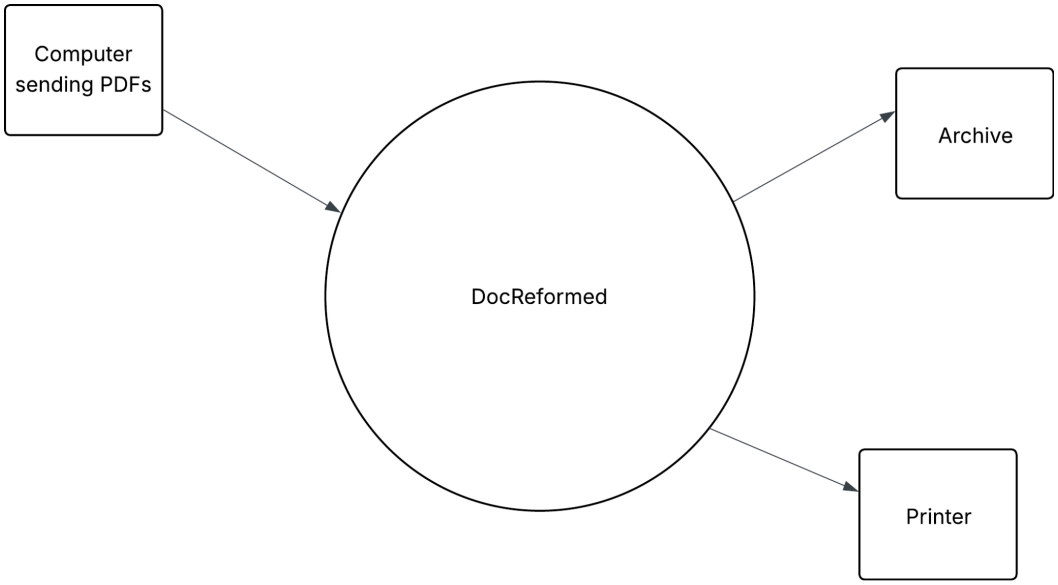

Figure 1: Context diagram showing a high-level overview of the DocReformed system.

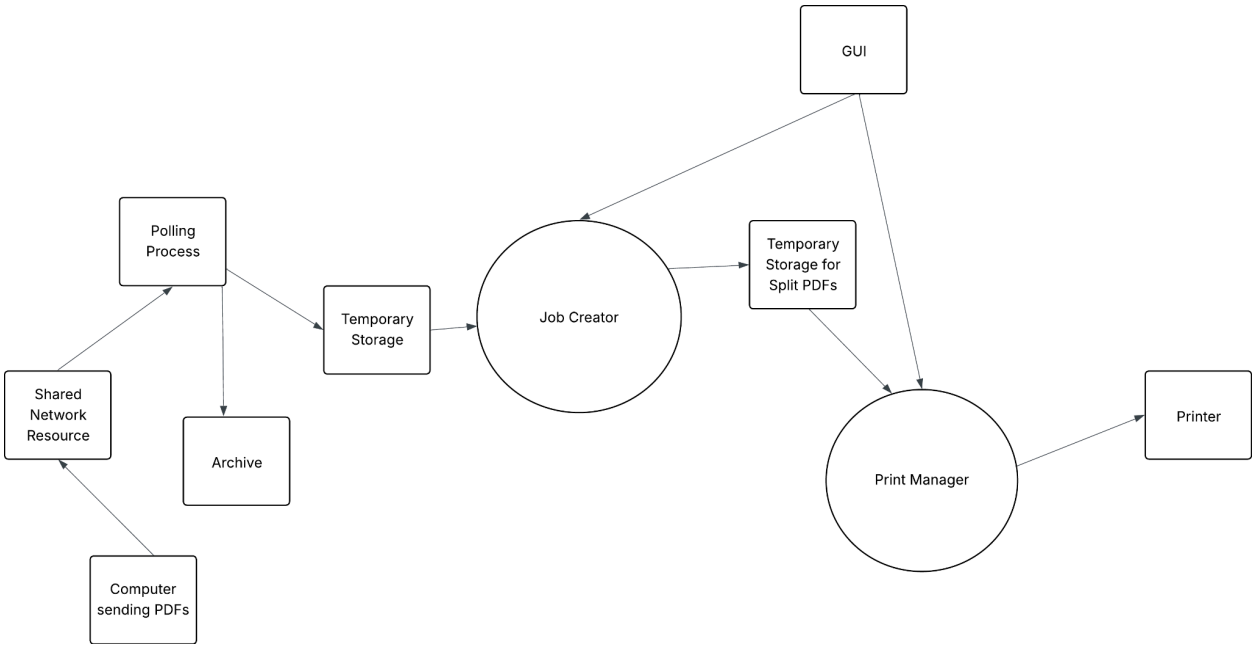

Figure 2: Context diagram showing an overview of all systems within DocReformed.

3.1. Summary & Overview

DocReformed's architecture (see section 5 for more details) makes use of three primary systems. A polling system is used to listen for upload events on a target directory, which can be specified by the user. When a PDF file is uploaded to this directory, it will trigger the polling system. A copy of the PDF is archived, then the original is sent to temporary storage to be accessed by the job creator. The job creator system is responsible for splitting the PDF into smaller files with a maximum page count specified by the user. It will also store jobs as persistent DTOs in order to ensure that closing DocReformed will not result in those jobs being lost. Finally, the print manager contains the bulk of the functions that a user will be expected to utilize during regular operation. It is capable of moving, deleting, viewing, and queuing jobs for printing.

3.2. Polling System

The polling system begins by getting the polling and archive directories specified by the user. This polling directory, which is expected to be a shared network resource, will be listened to by the polling system. Whenever a PDF file is uploaded to the polling directory, the system will first wait for the upload to complete, then create a copy of the PDF to be sent to a user-specified archive directory. Figure 3 shows the source code that performs the latter task. It is worth noting that the process of copying the PDF to the archive is performed in another thread, for the sake of performance. Then, as shown in figure 4, the file is moved to another directory. This output directory is used by the job creator (see section 3.3) as its source of data. Note that the file is moved at this point, not copied, meaning the polling system is no longer responsible for handling data after this point.

```
// Archive original PDF if it doesn't yet exist
string archivePath = Path.Combine(_archiveDirectory, Path.GetFileName(fullPath));
if (!File.Exists(archivePath))
{
    // Copy (using another thread) the original PDF to the archive directory
    await Task.Run(() => File.Copy(fullPath, archivePath, overwrite: false), token).ConfigureAwait(false);
    Debug.WriteLine($"Successfully archived PDF: {fullPath} -> {archivePath}");
}
else
{
    Debug.WriteLine($"Archive already contains: {archivePath}");
}

token.ThrowIfCancellationRequested();
```

Figure 3: The source code for the polling system's archiving feature.

```
// Move PDF to output directory
string outputPath = Path.Combine(_outputDirectory, Path.GetFileName(fullPath));
if (!File.Exists(outputPath))
{
    await Task.Run(() => File.Move(fullPath, outputPath), token).ConfigureAwait(false);
    Debug.WriteLine($"Moved PDF: {fullPath} -> {outputPath}");
}
else
{
    Debug.WriteLine($"Output already contains: {outputPath}");
}
```

Figure 4: The source code that moves the original PDF out of the polling system.

3.3. Job Creator

The job creator is the second major system within DocReformed. It is responsible for splitting a PDF and creating jobs with associated information. The jobs themselves are what will be passed to the print manager, and are what the user will perform operations on. The most important components of the job creator are its PDF splitter (a separate class, only used here) and its MakeJobsAsync function. The MakeJobAsync function, seen in figure 5, is responsible for creating the task for job creation. This creates an absolute path for the soon-to-be split PDF within a temporary storage location (known as the job well) and passes arguments to the CreateSplitTaskAsync. This function, shown in figure 6, is responsible for calling the PDF splitter (see figure 7) and splitting the original PDF into smaller pieces, none of which may be larger than the user-defined maximum page size. These splits are performed on a threadpool thread in order to improve performance. The list of split PDF filenames is returned and used by CreateSplitTaskAsync to create jobs for each split PDF. These are stored in a dictionary, mapped by a key which is the filename of the split PDF.

```
public Task<Job> MakeJobAsync(string printerName, string pdfName, bool simplex)
{
    if (string.IsNullOrWhiteSpace(pdfName)) throw new ArgumentException(nameof(pdfName));

    // normalize to absolute path inside JobWell — this is the key used to dedupe concurrent requests
    var inputPdfPath = Path.Combine(AppSettings.JobWell, pdfName);
    var key = Path.GetFullPath(inputPdfPath);

    // If a split for the same PDF is already running, return the existing task so callers wait on the same work.
    return _inProgressSplits.GetOrAdd(key, _ => CreateSplitTaskAsync(printerName, pdfName, simplex, key));
}
```

Figure 5: Function for creating a job.

```
private Task<Job> CreateSplitTaskAsync(string printerName, string pdfName, bool simplex, string key)
{
    // Run the work on a background thread.
    var task = Task.Run(async () =>
    {
        try
        {
            var inputPdfPath = Path.Combine(AppSettings.JobWell, pdfName);
            if (!File.Exists(inputPdfPath))
            {
                throw new FileNotFoundException("Pdf not found in JobWell", inputPdfPath);
            }

            if (!Directory.Exists(AppSettings.JobDir))
            {
                Directory.CreateDirectory(AppSettings.JobDir);
            }

            // perform split on a threadpool thread
            var splitFiles = await Task.Run(() => m_pdfSplitter.SplitPdf(inputPdfPath, AppSettings.JobDir, AppSettings.MaxPages)).ConfigureAwait(false);

            var job = new Job(printerName, splitFiles, simplex, pdfName);
            return job;
        }
        finally
        {
            // Ensure the in-progress entry is removed when the task completes (success or failure).
            // Use discard for the out parameter to avoid declaring an unused variable.
            await Task.Run(() =>
            {
                _inProgressSplits.TryRemove(key, out _);
            });
        }
    });

    return task;
}
```

Figure 6: Creating the task to split the PDF.

```
//splits a pdf into smaller pdfs but slicing every maxPages amount of pages and returns of a list of all file names to the pdfs it created
public List<string> SplitPdf(string inputFilePath, string outputDirectory, int maxPages)
{
    //check if inputpath exists
    if (!File.Exists(inputFilePath))
    {
        //throw error
        throw new FileNotFoundException("Input PDF not found.", inputFilePath);
    }
    //check if output directory exists
    if (!Directory.Exists(outputDirectory))
    {
        throw new DirectoryNotFoundException($"Output directory not found: {outputDirectory}");
    }
    //check if valid maxPagesAmount
    if (maxPages < 1)
    {
        throw new ArgumentOutOfRangeException(nameof(maxPages), "Max pages must be greater than 0");
    }
    //output file name list
    List<string> outputNameList = new List<string>();
    //using statement ensures proper clean up of pdfdocument object because it uses idisposable interface
    using (PdfDocument input = PdfReader.Open(inputFilePath, PdfDocumentOpenMode.Import))
    {
        //open pdf and read from original file
        int inputPageAmt = input.PageCount;
        for (int i = 0; i < inputPageAmt; i += maxPages)
        {
            //create new pdf
            //determine where the end page should be
            int endPage = Math.Min(i + maxPages, inputPageAmt);
            using (PdfDocument output = new PdfDocument())
            {
                //copy pages from input pdf to output
                for (int j = i; j < endPage; j++)
                {
                    output.AddPage(input.Pages[j]);
                }
                //get original filename
                string fileName = Path.GetFileNameWithoutExtension(inputFilePath) + $"_part_{i+1}-{endPage}.pdf";
                //add file name to the
                outputNameList.Add(fileName);
                //create output path for file
                string outputPath = Path.Combine(outputDirectory, fileName);
                //save output pdf
                output.Save(outputPath);
            }
        }
    }
    return outputNameList;
}
```

Figure 7: PDF splitter function source code.

3.4. Print Manager

The job manager performs a great deal of the functions that a standard user will require from DocReformed, including queuing, moving, and deleting jobs. It also handles the administrative tasks of adding and removing printers manually from DocReformed.

The source code for the queuing function, seen in figure 8, demonstrates how the jobs are sent to printers. A separate class exists within DocReformed exclusively for printers, with a method that allows jobs to be added to them. Once a job is added to a printer, it is sent to the printer itself for printing. The queuing function also has provisions in place for deduplicating jobs and creating an object of the printer class if one does not yet exist for the specified printer.

A job can be deleted via the print manager class's functions. The source code for deleting a job can be seen in figure 9, which shows that doing so will release the job from the list of jobs associated with the printer. The ability to delete a job is one of the fundamental actions of DocReformed, and will be commonly used. Another very prominent feature is the ability to move jobs to another printer, which is shown in the code in figure 12. This code will remove the job from the queue it was originally located in, then throw an exception if the job that the user is trying to move is not found in the list. As with the queuing function, the moving function will also create a new object of the printer class if the target printer does not exist yet.

The code in figures 10 and 11 show how printers are added and removed. These functions are called when users use the GUI to manually add or remove printers from DocReformed. The function that adds a printer will first check if the provided name is valid, then create the printer if it does not already exist. If the name is invalid, an exception is thrown. If a user chooses to remove a printer, the function will verify that the name is valid (and again throw an exception if not) and remove its jobs and print directory. This will completely remove all data pertaining to the printer from DocReformed. It will need to be manually re-added after this.

The act of printing is handled by Adobe Reader, software which is present on the machines that CPS will be installing DocReformed onto. The function responsible for handling job printing (see figure 13) will validate input and throw exceptions if invalid arguments were passed, find the Adobe Reader executable, and use command line arguments to send a print command to Adobe Reader. Another function (figure 14) will monitor the print job and determine whether or not it successfully completed prior to a timeout value. A job is considered completed if it leaves the print spooler or is explicitly reported to have been completed. If the timeout is met and the Adobe Reader process is still running without an updated status for the print job, it kills the process.

The persistent lists of jobs are stored as persistent DTOs and mapped to each printer. Figure 15 shows the function that is called to convert the stored data into instances of the printer class. This ensures that closing and reopening DocReformed will not result in job information being lost, as it will be saved in DTOs. More information about this feature can be found in section 7.2. Several functions in the print manager will save to these DTOs after performing their operations.

```
// queues a job
public void QueueJob(Job job)
{
    if (job == null) throw new ArgumentNullException(nameof(job));

    lock (_lock)
    {
        var targetName = string.IsNullOrWhiteSpace(job.printerName) ? UnassignedPrinterName : job.printerName;
        var target = _printers.FirstOrDefault(p => string.Equals(p.Name, targetName, StringComparison.OrdinalIgnoreCase));

        if (target == null)
        {
            // create printer entry if not present
            target = new PrinterClass { Name = targetName, Status = "Unknown" };
            _printers.Add(target);
        }

        // avoid duplicate job ids
        if (!target.Jobs.Any(j => j.jobId == job.jobId))
        {
            job.printerName = target.Name; // keep job consistent
            target.Jobs.Add(job);
            SavePrinterStore();
        }
    }
}
```

Figure 8: Source code for queuing a job.

```
// release job
public void ReleaseJob(Guid jobId)
{
    lock (_lock)
    {
        bool changed = false;
        foreach (var p in _printers)
        {
            var existed = p.Jobs.RemoveAll(j => j.jobId == jobId) > 0;
            if (existed) changed = true;
        }

        if (changed) SavePrinterStore();
    }
}
```

Figure 9: Source code for releasing (deleting) a job.

```
// adds printer to printer set (and persists the Printer including its Jobs list)
public bool AddPrinter(string PrinterName)
{
    if (string.IsNullOrWhiteSpace(PrinterName)) throw new ArgumentException(nameof(PrinterName));

    // ensure directory exists
    if (!Directory.Exists(AppSettings.PrinterDir))
    {
        Directory.CreateDirectory(AppSettings.PrinterDir);
    }

    // create printer directory if missing
    string printerPath = Path.Combine(AppSettings.PrinterDir, PrinterName);
    if (!Directory.Exists(printerPath))
    {
        Directory.CreateDirectory(printerPath);
    }

    lock (_lock)
    {
        if (!_printers.Any(p => string.Equals(p.Name, PrinterName, StringComparison.OrdinalIgnoreCase)))
        {
            var newPrinter = new PrinterClass { Name = PrinterName, Status = "Unknown" };
            _printers.Add(newPrinter);
            SavePrinterStore();
            return true;
        }
    }

    return false;
}
```

Figure 10: The source code for adding a printer.

```
// removes printer and moves its jobs to the unassigned printer
public PrinterClass RemovePrinter(string printerName)
{
    if (string.IsNullOrWhiteSpace(printerName)) throw new ArgumentException(nameof(printerName));

    lock (_lock)
    {
        var entry = _printers.FirstOrDefault(p => string.Equals(p.Name, printerName, StringComparison.OrdinalIgnoreCase));
        if (entry == null) return null;

        var unassigned = _printers.FirstOrDefault(p => string.Equals(p.Name, UnassignedPrinterName, StringComparison.OrdinalIgnoreCase));
        if (unassigned == null)
        {
            unassigned = new PrinterClass { Name = UnassignedPrinterName, Status = "Unassigned" };
            _printers.Add(unassigned);
        }

        // move jobs
        foreach (var j in entry.Jobs)
        {
            j.printerName = unassigned.Name;
            if (!unassigned.Jobs.Any(x => x.jobId == j.jobId))
            {
                unassigned.Jobs.Add(j);
            }
        }

        _printers.Remove(entry);
        SavePrinterStore();

        // remove directory if exists
        try
        {
            string printerPath = Path.Combine(AppSettings.PrinterDir, printerName);
            if (Directory.Exists(printerPath)) Directory.Delete(printerPath, true);
        }
        catch
        {
            // ignore IO errors
        }

        return entry;
    }
}
```

Figure 11: The source code for removing a printer.

```
// sets a job to a printer (moves job object between Printer.Jobs lists)
public void SetJobPrinter(Guid jobId, string printerName)
{
    if (printerName == null) throw new ArgumentNullException(nameof(printerName));

    lock (_lock)
    {
            // find and remove job from current printer (if any)
        Job job = null;
        foreach (var p in _printers)
        {
            var existing = p.Jobs.FirstOrDefault(j => j.jobId == jobId);
            if (existing != null)
            {
                job = existing;
                p.Jobs.Remove(existing);
                break;
            }
        }

        // if job was not found, we cannot move it — caller should QueueJob first
        if (job == null) throw new KeyNotFoundException("Job not found; queue the job before assigning");

        // determine target printer
        var targetName = string.IsNullOrWhiteSpace(printerName) ? UnassignedPrinterName : printerName;
        var target = _printers.FirstOrDefault(p => string.Equals(p.Name, targetName, StringComparison.OrdinalIgnoreCase));
        if (target == null)
        {
            target = new PrinterClass { Name = targetName, Status = "Unknown" };
            _printers.Add(target);
        }

        job.printerName = target.Name;
        target.Jobs.Add(job);

        SavePrinterStore();
    }
}
```

Figure 12: Source code for moving a job.

```
// command line arguments for adobe reader
string arguments = $"/t \"{inputPdfPath}\" \"{printerName}\"";

var psi = new System.Diagnostics.ProcessStartInfo
{
    FileName = AdobeReaderPath,
    Arguments = arguments,
    UseShellExecute = false,
    CreateNoWindow = true,
    WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden
};

Debug.WriteLine("starting process for print");
System.Diagnostics.Process proc = null;
try
{
    proc = System.Diagnostics.Process.Start(psi);
    if (proc == null)
        throw new InvalidOperationException("Failed to start Adobe Reader process.");

    TimeSpan timeoutMs = TimeSpan.FromSeconds(120); // 120 seconds timeout for printing

    bool completed;
    try
    {
        completed = WaitForPrintJobCompletion(printerName, pdfName, timeoutMs);
    }
    catch
    {
        // propagate any exceptions from the spooler waiting logic (e.g. job error states)
        try { if (proc != null && !proc.HasExited) proc.Kill(); Debug.WriteLine("process timed out"); } catch { }
        throw;
    }

    // timed out waiting for spooler -> do not throw, return false per requirement
    if (!completed)
    {
        try { if (proc != null && !proc.HasExited) proc.Kill(); } catch { }
        return false;
    }
```

Figure 13: Code snippet of PrintJob function, showing the creation of a print process and command line arguments passed to Adobe Reader, along with examples of error handling.

```
private bool WaitForPrintJobCompletion(string printerName, string documentName, TimeSpan timeout)
{
    var server = new LocalPrintServer();
    PrintQueue queue;
    try
    {
        queue = server.GetPrintQueue(printerName);
    }
    catch
    {
        return false; // printer not available
    }

    var sw = Stopwatch.StartNew();
    int? observedJobId = null;

    while (sw.Elapsed < timeout)
    {
        try
        {
            queue.Refresh();
            var jobs = queue.GetPrintJobInfoCollection().Cast<PrintSystemJobInfo>().ToList();

            // try to find the job by name first, then fallback to submitter + time heuristics
            var job = jobs.FirstOrDefault(j => string.Equals(j.Name, documentName, StringComparison.OrdinalIgnoreCase))
                      ?? jobs.FirstOrDefault(j => j.Submitter == Environment.UserName && (observedJobId == null || j.JobIdentifier == observedJobId));

            if (job != null)
            {
                observedJobId = job.JobIdentifier;

                // job finished successfully (removed from spooler) or reported completed
                if (job.IsCompleted || job.IsDeleted) return true;

                // Some environments set JobStatus flags; check for error states
                if (job.JobStatus.HasFlag(PrintJobStatus.Error) || job.JobStatus.HasFlag(PrintJobStatus.Offline))
                    throw new InvalidOperationException($"Print job in error state: {job.JobStatus}");

                // number of pages may be available as an indicator
                if (job.NumberOfPages > 0 && job.IsCompleted) return true;
            }
            else
            {
                // if we previously saw a job but now it is gone, assume completion
                if (observedJobId != null) return true;
            }
        }
        catch
        {
            // transient spooler errors — retry until timeout
        }

        Thread.Sleep(500);
    }

    return false; // timed out
}
```

Figure 14: The WaitForPrintJobCompletion function, demonstrating how jobs are monitored, how Adobe Reader error statuses are used, and more error handling.

```
private void LoadPrinterStore()
{
    lock (_lock)
    {
        try
        {
            if (!Directory.Exists(AppSettings.PrinterDir)) Directory.CreateDirectory(AppSettings.PrinterDir);

            var path = AppSettings.PrinterStoreFile;
            if (!File.Exists(path))
            {
                _printers = new List<PrinterClass>();
                return;
            }

            using (var fs = File.OpenRead(path))
            {
                var ser = new DataContractJsonSerializer(typeof(List<PersistedPrinterDto>));
                var dtoList = ser.ReadObject(fs) as List<PersistedPrinterDto> ?? new List<PersistedPrinterDto>();

                // convert DTOs to PrinterClass instances
                _printers = dtoList.Select(d =>
                {
                    var p = new PrinterClass
                    {
                        Name = d.Name,
                        Status = d.Status ?? "Unknown",
                        Jobs = d.Jobs ?? new List<Job>()
                    };
                    return p;
                }).ToList();
            }
        }
        catch
        {
            _printers = new List<PrinterClass>();
        }
    }
}
```

Figure 15: Loading the persisted printer information.

3.5. User Interface

The next several figures will demonstrate the graphical user interface of DocReformed, and explain the rationale behind its design. This user interface will be the component of DocReformed that the average user interacts with the most. It is designed to be very simple to use and understand. The settings page (figure 16) allows administrators of DocReformed to change the directories that PDFs are polled, where jobs are created, where jobs are stored, and where PDFs are archived. Users can also add printers that are detected by DocReformed, or remove them after they have been added. A pop-up message will tell users when a printer has been successfully added or removed.

The only other page of DocReformed is the print job management window, or the primary interface of DocReformed. This is where most users will spend the majority of their time. Users have the ability to manage queued jobs here, as well as browse their system's directories manually and, using the "Send Job" button, manually trigger a PDF to be processed by the job creator. The user can select whichever printer they prefer from the "Select Printer" dropdown menu, where the manually-triggered PDFs will have their jobs sent. This page can be seen in figure 17.

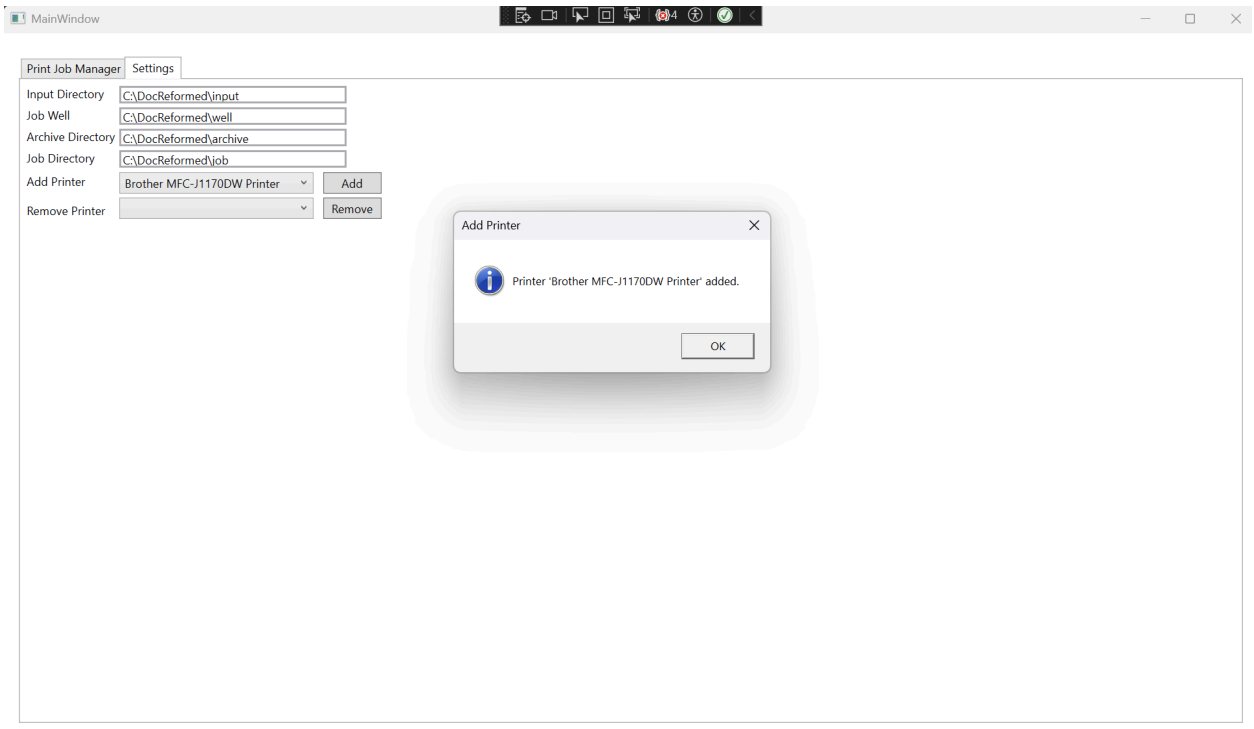

Figure 16: The Settings page of DocReformed, along with a pop-up.

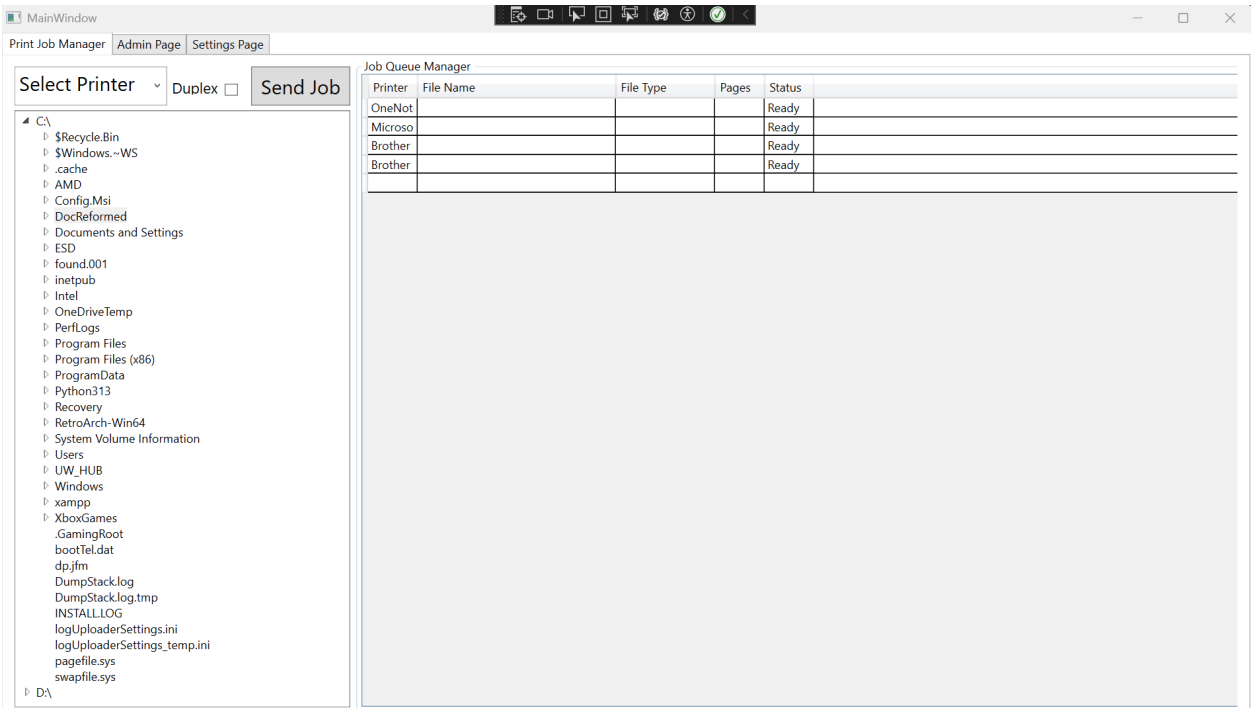

Figure 17: Primary interface of DocReformed, featuring a list of printers and directory browser.

4. Use Case Scenarios

This section will describe use case scenarios, which are categorized into two groups. Standard use case scenarios refer to regular, normal operations that DocReformed will be expected to perform on a daily basis. Such use cases will feature standard employees as actors who are not experienced with computer science or information technology. Administrative use case scenarios will feature more experienced users as actors, who will be expected to have a degree of proficiency in computer science or information technology. The latter will be used much less often than standard use cases.

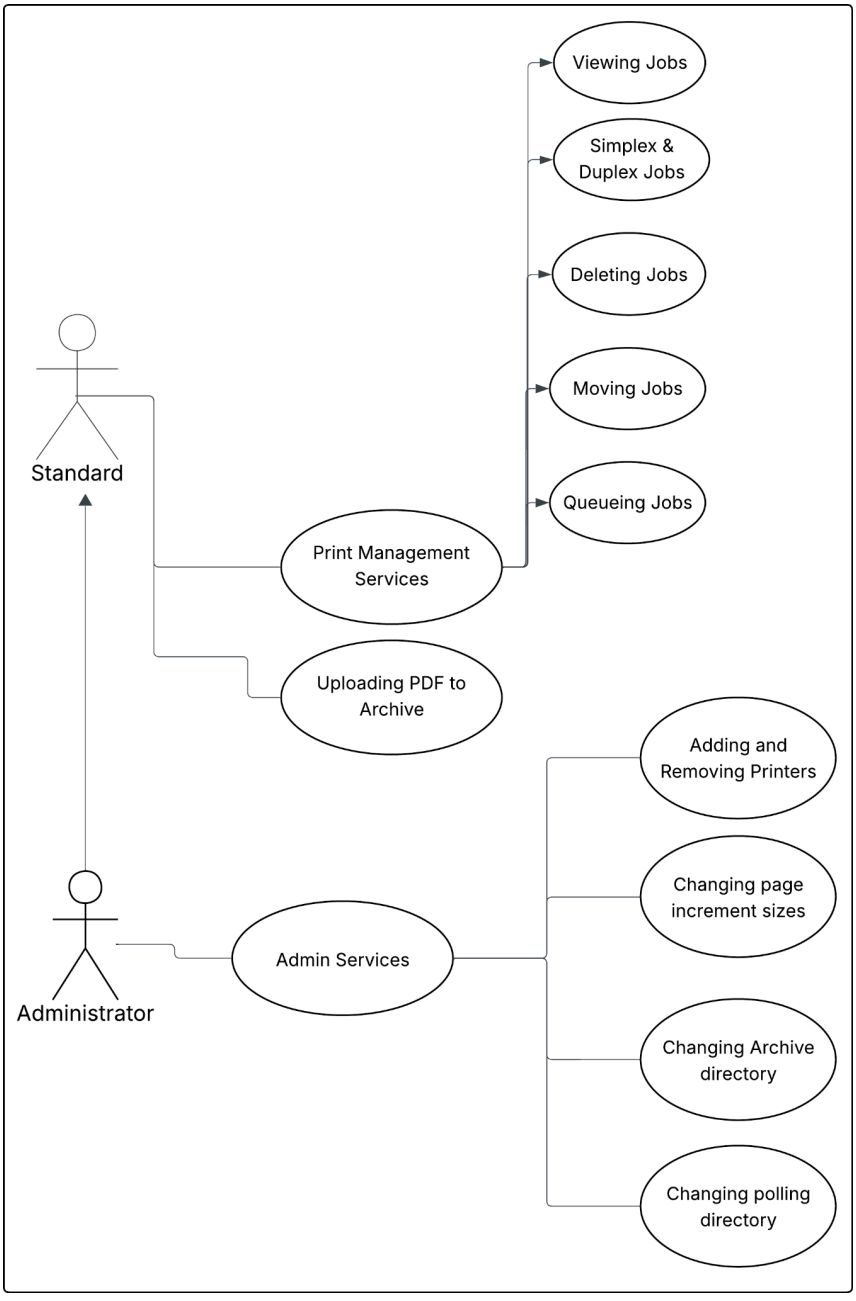

Figure 18: Use case diagram for DocReformed, described in sections 4.1 and 4.2.

4.1. Standard

Several use case scenarios can be defined for DocReformed, which are visualized in figure 18. These can be split into two general categories: standard and administrative. Within the standard category, two subcategories can be defined. The first subcategory includes the scenario of uploading a PDF to an archive directory. Within DocReformed, administrators can specify an archive directory (see section 4.2). Every time a user uploads a PDF file to DocReformed or the polling directory, a copy of that PDF file is sent to the archive directory. This process is handled by the polling system (see section 3.2) and does not require user action or intervention. The second subcategory includes all print management services, which are described below.

The first - and perhaps most important - print management service is the ability to queue jobs for printing. After a PDF file has been uploaded and split by DocReform (see section 3.3), each split PDF will be considered a print job by the system. Once the job is created, the user can manually queue the job for printing in its respective printer.

Another print management service allows users to move the jobs. This includes both moving the job's position within a queue, or moving the job to another printer's queue entirely. In either case, the job's contents will not be affected by the move operation; the split segment of the PDF file will not change. Users can move any job to any queue or position until the job is sent to a printer.

Users can also use the delete print management service to delete queued jobs. Deleting a job will completely remove it from DocReformed. This means the job will no longer be queued for any printer, the split segment of the PDF file will be deleted from the system's storage, and DocReformed will no longer store the job's information in persistent storage, meaning closing and re-opening the application will not cause the job to reappear. Users will need to manually re-upload a copy of the original PDF in order to create a new, equivalent job.

Another option is the ability to view a job. This will open the PDF file associated with that particular job in the system's default PDF application (i.e., Adobe Acrobat or a web browser). The options that users can access within the default application will vary depending on which application is being used.

4.2. Administrative

Administrators will be able to utilize features organized in the administrative services category, in addition to every feature accessible to standard users. Administrative services primarily manage the application's settings rather than individual jobs, and will not be accessed nearly as often as the standard print management services or archiving categories described in section 4.1.

The first administrative service is the ability to add printers. This will allow administrators to manually add a printer that is connected to the same local area network as the

system running DocReformed. DocReformed will automatically attempt to detect printer systems on the local area network, but this feature will allow administrators to add printers themselves if automatic detection fails.

Administrators can also change two critical directories required for DocReformed's operation: the polling directory and the archive directory. The purpose of both directories is described in section 3.2. These directories should be changed by an administrator with caution; this is reiterated in the user manual.

Finally, administrators can change the page increment size. This is the value that defines the size of the split PDF file segments, and is set to one thousand pages by default. Again, this setting should be changed with caution, as sending print jobs to printers that do not have enough memory to accommodate a large number of pages can lead to print errors or failures. Administrators will need to refer to the printer's documentation to determine the appropriate page increment size.

5. Systems Architecture and Design

This section will complement sections 3 and 4 by describing the architecture of the systems and functions that were listed in those sections. This includes context, data flow, and message sequence diagrams to visualize the relationships between all system components.

5.1. Architecture Overview

DocReformed's architecture has been designed to be as streamlined as possible. As is visible in the context diagram shown in section 3's figure 1, DocReformed will be interacting with very few external resources. The first is the computer system responsible for sending PDFs to DocReformed, either via local upload or polling of a shared network resource. The second is an archive directory, which is simply a shared network resource where an uploaded PDF is copied to. Finally, DocReformed will send print jobs to a destination printer. It is worth noting that DocReformed is not able to communicate with network resources beyond CPS's LAN. In figure 2's context diagram (again, see section 3 for this figure), DocReformed contains all of the subsystems discussed at length in section 3. Due to the nature of how data flows through and is processed by this application, DocReformed was designed around the pipe-and-filter architecture. The rationale behind the architecture will be explained in section 5.2 below.

5.2. System Design

The expanded context diagram, seen in figure 2, demonstrates the relationships between the systems. The polling process system receives data from a shared network resource (or the local system, depending on the use case) and will then output data to the archive directory and temporary storage on DocReformed's local system. This storage is where the job creator system receives its input data. It splits the PDF file that it receives and creates jobs, sending these split PDFs and corresponding job details to another temporary storage location. This second temporary storage location is where the print manager receives its input. Once the print manager performs the user-specified operations, it queues the jobs to be sent to the printer. Above both the job creator and print manager, a GUI allows the user to control certain aspects of these systems. This includes changing settings used by the job creator and initiating operations within the print manager.

Figure 19 below shows another visualization of DocReformed's systems, this time as a data flow diagram. The arrows represent the flow of data (primarily PDFs and corresponding job information) between components of the architecture. The network share folders, archive directory, and printer are not internal components of DocReformed, but rather external directories or devices on the LAN that the application is communicating with.

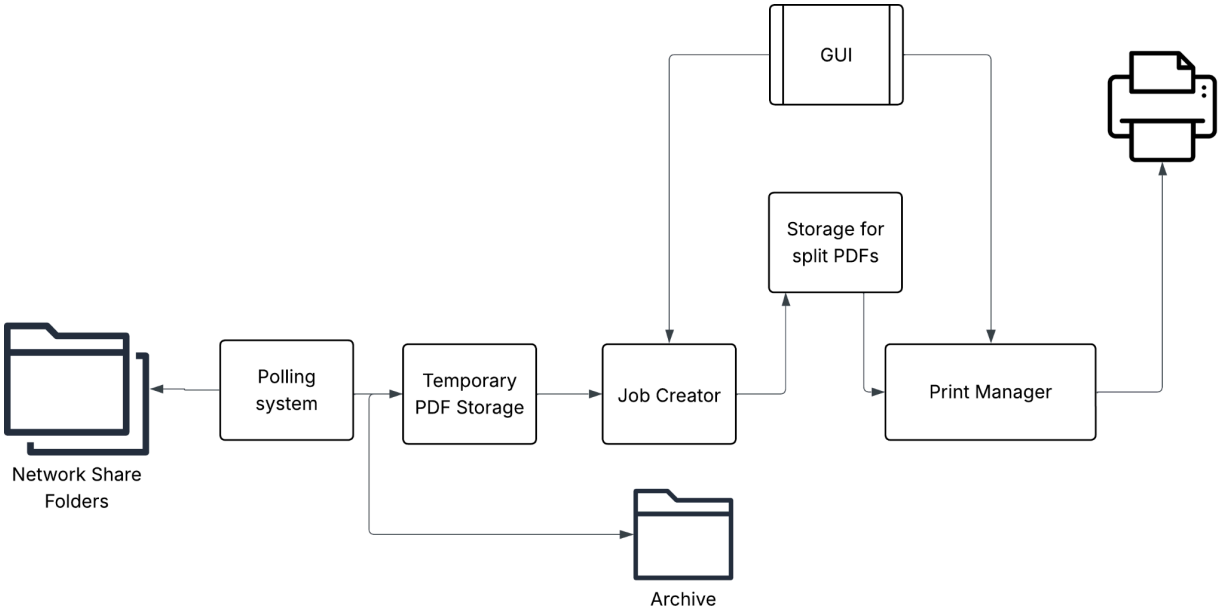

Figure 19: Data flow diagram for DocReformed.

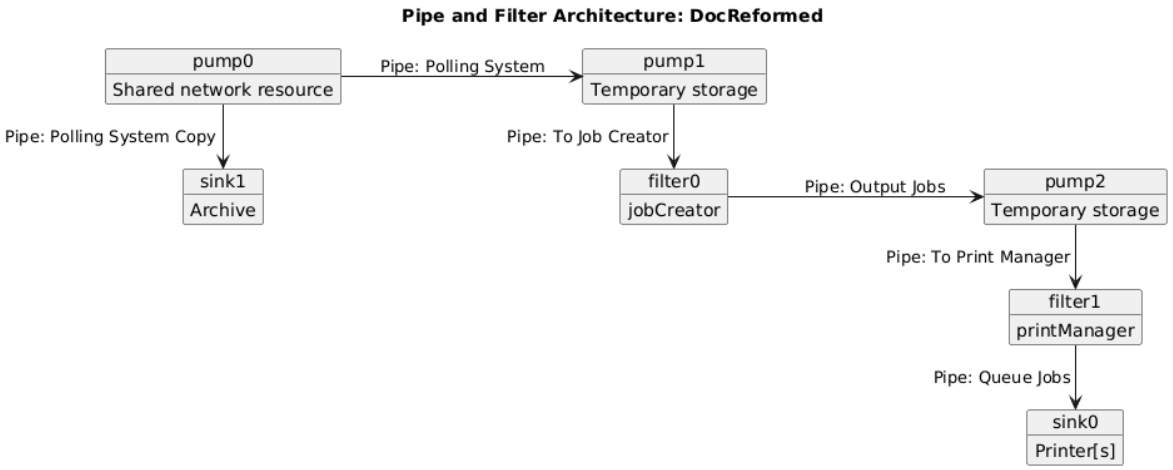

Figure 20: Pipe and filter architecture of DocReformed.

Based on the requirements for this system, the developers elected to use the pipe and filter architecture for DocReformed. This was due to the rather straightforward way that data flows through and is processed by the system. Figure 20 shows how each of the aforementioned major system components can be expressed within this architecture as pipes, pumps, sinks, or filters.

The first pump in the system, pump0, is the shared network resource from which the original PDF file originates. This is detected by the polling system, which will copy it to the archive (sink1) and as such act as a pipe. The polling system acts as another pipe when it sends the polled PDF file to pump1, temporary storage. From here, the jobCreator object will get its input from pump1, which acts as another pipe. The jobCreator object is called filter0 since it is

the first system to process the data, by splitting the PDF and adding corresponding job details. Once finished, filter0 will act as another pipe when it outputs jobs to pump2, another temporary storage location. This is where printManager uses another pipe to get files in that storage location as its input, thus acting as filter1. Users can perform their desired operations via the GUI, and when finished, filter1 queues the jobs via the final pipe to sink0, which is the final destination of the jobs: the printer(s).

While not pictured due to its secondary nature, DocReformed also incorporates elements of a layered architecture. This is because the GUI will be considered its own layer (View Layer), with a middle layer (ViewModel Layer) to handle business logic from inputs from the GUI, and a layer for models and data that the ViewModel layer will interact with. The polling system is also event-driven in nature, but since this is a single component of a larger architecture, the developers do not consider DocReformed to follow the event-driven architecture paradigm.

6. Quality Attributes and Operations

This section will discuss several additional aspects of DocReformed that did not fit into previous sections. These include performance considerations and analyses, scalability concerns, availability of DocReformed's systems, and application security. It will conclude with a brief description of the OA&M of DocReformed.

6.1. Performance

Performance was an important facet of DocReformed's development since its inception. A PDF splitting process that was too lengthy would result in a poor user experience, and potentially cause instability in the application. There was also the risk of DocReformed consuming too many system resources during its splitting process. During development, several tests were created to test DocReformed's ability to handle PDFs of various sizes. These tests, and their results, can be seen in figure 21 below. In each test, a PDF of ten thousand, twenty thousand, forty thousand, one hundred thousand, and two hundred fifty thousand pages are split into increments of one thousand pages. The most common cases (ten, twenty, and forty thousand) take 11.7 seconds at most. The largest PDF size that CPS estimated they would be splitting was approximately one hundred thousand pages, although this was said to be uncommon. This will take approximately 51.1 seconds. Finally, a PDF much larger than anything CPS estimated that they would need to split, two hundred fifty thousand pages, takes over five minutes to split. See section 8.1 for deeper analyses of these tests.

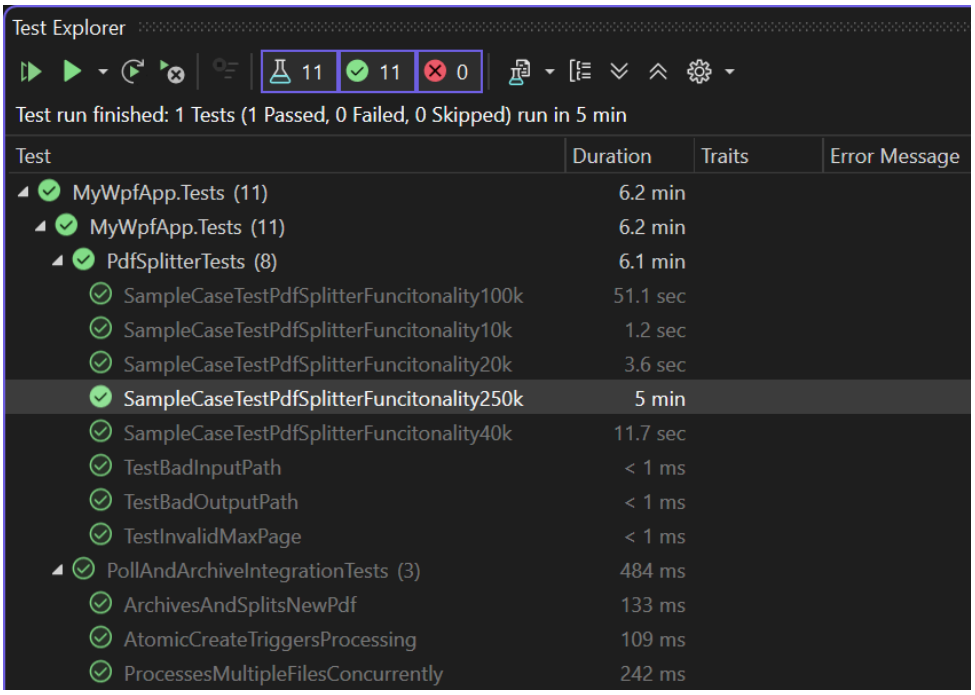

Figure 21: Tests showing the time it takes to split PDFs into 1,000-page pieces.

6.2. Scalability

CPS did not describe scalability as a major requirement for their Doc Transform replacement. As such, DocReformed is not designed to be deployed on a large scale within a single LAN. While it is easily installable on any system running Windows 11 through an executable file, simultaneous use of DocReformed by many users on the same LAN is not recommended. This is because of the inherent resource bottleneck presented by the limited number of printers on the LAN. Those using DocReformed need to be cognizant of accidentally queuing a large number of jobs to be sent to only a few printers, instead of distributing the jobs to multiple printers. Such a scenario is mentioned in the user guide, along with the complications that it brings. Overall, DocReformed was always intended to be used on a small scale.

6.3. Availability

In a pipeline where data flows from a shared network resource to DocReformed, to a printer, DocReformed is the least likely component to become unavailable. The application itself is designed to be stable (see section 7) and use as few system resources as possible. If an issue on the network, or even DocReformed's local system, causes network connectivity issues between DocReformed and the printers or shared resources, DocReformed itself will still be operational; it simply will not be able to communicate with other devices on the network. DocReformed does not attempt to change any network settings on the local system, nor does it do so to the printers, and therefore would not be responsible for causing availability issues on the LAN.

The only scenario where DocReformed may not be available for use while open is when a PDF is being split. See section 8 for PDF splitting times in test environments. This will be a relatively short period of time, and while the user may not be able to upload new PDFs or manage jobs in that time, DocReformed is still performing a background task to split the PDF. Even in this state, DocReformed could still be considered available.

Ultimately, the ability for DocReformed's services to remain available relies primarily on the stability of CPS's LAN, rather than DocReformed itself. As long as a connection between the system running DocReformed, the shared network resources, and the printers remains intact, DocReformed will remain available.

6.4. Security

Per the NIST CSF 2.0, security threats that may be present within DocReformed need to be identified, detected, protected against, responded to, and recovered from, with a governance layer over each of these. As mentioned in section 2.3, due to the nature of the LAN in which DocReformed will be utilized, many threats are already mitigated. In the context of CPS's network, the company falls under tier 3 of NIST CSF 2.0's tiers, indicating that leadership and employees are aware of network-based security threats and possess the knowledge to perform daily tasks within such a network.

There is also the possibility for PDF-based malware to be uploaded to DocReformed. As there will be a feature that allows users to view split PDFs within the software, this effectively amounts to allowing users to execute any malicious payload in a given PDF. It is worth noting that the PDFs sent to the stakeholders (and thus to DocReformed) will be generated and submitted by the stakeholders' clients, who have an interest in avoiding inserting malware into these PDFs to avoid damage to their own reputations or violating regulations by causing a data breach. Furthermore, the clients will be uploading these PDFs through a secure online portal, which is accessible only to these clients. The sensitive data within these PDFs prevents them from being uploaded to a cloud-based anti-malware service to scan them for malware. CPS informed the developers that they were generally not concerned about this particular risk, and therefore, no mitigation measures were put in place within DocReformed to address this threat.

Security governance recommendations for DocReformed, including descriptions of the potential threats outlined above, are present in the software's accompanying user manual.

6.5. OA&M

In addition to this technical report, a comprehensive user guide has been provided to CPS. This user guide will include information pertinent to regular use of DocReformed, its administration, and maintenance of its systems. Future maintenance and administration of DocReformed is the responsibility of CPS, though the provided resources in the user guide are sufficient to answer any OA&M-related questions that they may have.

7. Error Handling and Recovery

Proper error handling and recovery were of particular importance to this project. It would not be desirable for, as an example, an error to occur whilst DocReformed is splitting a fifty-thousand-page document, or when sending several dozen jobs to printers. This could cause unintentional data loss, failure to print necessary files, and an overall negative user experience. Errors were handled and recovered from primarily through using try, throw, and catch statements, as well as persistent storage of job information.

Additional programming techniques that were used to avoid errors or unexpected behavior in DocReformed can be seen throughout section 3's description of the source code of key systems.

7.1. Error Handling

```
private async Task ProcessFileAsync(string fullPath, CancellationToken token)
{
    try
    {
        const int maxAttempts = 10;
        const int delay = 1000;

        var ready = false;
        for (int i = 0; i < maxAttempts; i++)
        {
            token.ThrowIfCancellationRequested();

            try
            {
                // Tryy opening file to see if it's been fully written to directory
                using (var fs = File.Open(fullPath, FileMode.Open, FileAccess.Read, FileShare.None))
                {
                    // Double checking that file has content
                    if (fs.Length > 0)
                    {
                        ready = true;
                        break;
                    }
                }
            }
            catch (IOException)
            {
                // File is not written yet
                Debug.WriteLine($"File not yet written: {fullPath}");
            }
            catch (UnauthorizedAccessException)
            {
                // Permissions error
                Debug.WriteLine($"Insufficient privileges for: {fullPath}");
            }

            // Delay task for 1000ms (1s)
            await Task.Delay(delay, token).ConfigureAwait(false);
        }
```

Figure 22: An example of how try, throw, and catch statements were used.

```
// Poll input directory for new PDF file, archive it, pass it to PdfSplitter, log actions/errors
public PollAndArchive(string inputDirectory, string archiveDirectory, string outputDirectory)
{
    // Checking if input and archive directories exist
    if (!Directory.Exists(inputDirectory))
        throw new DirectoryNotFoundException($"Input directory not found: {inputDirectory}");
    if (!Directory.Exists(archiveDirectory))
        throw new DirectoryNotFoundException($"Archive directory not found: {archiveDirectory}");
```

Figure 23: Another example of how throw statements were used for error recovery.

Error handling was chiefly accomplished by using try, throw, and catch statements in the code. In figure 22, these statements are used to handle scenarios that may cause unexpected behavior from DocReformed. This particular code was taken from the PollAndArchive class, which is the implementation of the polling system (see section 3.2). It is designed to catch exceptions caused when a file is not fully written or if the system has insufficient permissions to access a file. DocReformed will gracefully handle the error by continuously waiting and then checking the file again, waiting for it to either be fully written or given proper permissions. Figure 23, taken from the same class, demonstrates how throw statements were used to prevent unpredictable behavior at the very beginning of an object's construction. It ensures that the necessary directories are able to be found before the polling process begins. These statements help prevent unexpected behavior or crashes.

The throw statements seen in figure 24 are another example of how errors are handled within DocReformed. These statements will throw errors if the PDF input to be split is not found, if the output directory (the job well) does not exist, or if the user inputs an invalid value as the maximum page count for the PDF splitting process.

```
public class PdfSplitter
{
    //splits a pdf into smaller pdfs but slicing every maxPages amount of pages and returns of a list of all file names to the pdfs it created
    public List<string> SplitPdf(string inputFilePath, string outputDirectory, int maxPages)
    {
        //check if inputpath exists
        if (!File.Exists(inputFilePath))
        {
            //throw error
            throw new FileNotFoundException("Input PDF not found.", inputFilePath);
        }
        //check if output directory exists
        if (!Directory.Exists(outputDirectory))
        {
            throw new DirectoryNotFoundException($"Output directory not found: {outputDirectory}");
        }
        //check if valid maxPagesAmount
        if (maxPages < 1)
        {
            throw new ArgumentOutOfRangeException(nameof(maxPages), "Max pages must be greater than 0");
        }
```

Figure 24: Throw statements as seen in the PdfSplitter class.

```
// --- Public API ---

// queues a job
public void QueueJob(Job job)
{
    if (job == null) throw new ArgumentNullException(nameof(job));

    lock (_lock)
    {
        var targetName = string.IsNullOrWhiteSpace(job.printerName) ? UnassignedPrinterName : job.printerName;
        var target = _printers.FirstOrDefault(p => string.Equals(p.Name, targetName, StringComparison.OrdinalIgnoreCase));

        if (target == null)
        {
            // create printer entry if not present
            target = new PrinterClass { Name = targetName, Status = "Unknown" };
            _printers.Add(target);
        }

        // avoid duplicate job ids
        if (!target.Jobs.Any(j => j.jobId == job.jobId))
        {
            job.printerName = target.Name; // keep job consistent
            target.Jobs.Add(job);
            SavePrinterStore();
        }
    }
}
```

Figure 25: The QueueJob function's use of basic conditional statements to prevent errors.

In addition to try, throw, and catch statements, more fundamental programming techniques were used to ensure the stability of DocReformed. Figure 25 shows an example of this, where basic conditional if statements are used to ensure that a target printer exists whenever a job is queued to a printer. It also uses this technique to avoid any duplicates of jobs, based on the ID associated with every job. This prevents the queuing and printing of multiple copies of the exact same job, which would otherwise result in loss of time, effort, and printing resources for CPS. Furthermore, see section 8.1 for a brief discussion on how the PDF splitting function is capable of detecting invalid input and preventing unexpected behavior.

7.2. Error Recovery

DocReformed's error handling measures are designed to avoid crashes in as many places as possible, which inherently allows the software to recover from errors. However, in the event of a crash, the developers have taken measures to ensure that as much data as possible is preserved. If DocReformed crashes or is otherwise closed unexpectedly, all data associated with a print job will be retained. When DocReformed is reopened, the jobs will still be recognized by the application. Figure 26 shows the implementation of this error recovery measure. When the PDF is split, the job is created and stored as a DTO in order to store the job persistently.

```
private Task<Job> CreateSplitTaskAsync(string printerName, string pdfName, bool simplex, string key)
{
    // Run the work on a background thread.
    var task = Task.Run(async () =>
    {
        try
        {
            var inputPdfPath = Path.Combine(AppSettings.JobWell, pdfName);
            if (!File.Exists(inputPdfPath))
            {
                throw new FileNotFoundException("Pdf not found in JobWell", inputPdfPath);
            }

            if (!Directory.Exists(AppSettings.JobDir))
            {
                Directory.CreateDirectory(AppSettings.JobDir);
            }

            // perform split on a threadpool thread
            var splitFiles = await Task.Run(() => m_pdfSplitter.SplitPdf(inputPdfPath, AppSettings.JobDir, AppSettings.MaxPages)).ConfigureAwait(false);

            var job = new Job(printerName, splitFiles, simplex, pdfName);
            return job;
        }
        finally
        {
            // Ensure the in-progress entry is removed when the task completes (success or failure).
            // Use discard for the out parameter to avoid declaring an unused variable.
            await Task.Run(() =>
            {
                _inProgressSplits.TryRemove(key, out _);
            });
        }
    });

    return task;
}
```

Figure 26: Implementation of job creation.

8. Results and Discussion

This section will describe the results of tests performed on DocReformed by the developers, as well as an analysis of these results. Then, it will transition to a discussion of feedback provided by CPS statements once the finished product is delivered. This feedback will be analyzed and explained as needed.

8.1. Test Results

Test results were discussed briefly in section 6.1. Deeper analyses will be conducted here. Figure 27 below shows the results of each test conducted on DocReformed. These tests are primarily designed to test the archive, polling, and splitting features, as job creation and print management were easily visually tested through using the GUI or examining the behavior of DocReformed when it was closed and reopened.

The first batch of tests were conducted on PDFs, each PDF being of a different size. The smallest, at ten thousand pages, took just over a second to split into 10 one thousand page increments. The largest, at two hundred fifty thousand pages (over double the highest number of pages that CPS asked the developers to accommodate), took over five minutes to split into 250 one thousand page increments. This final test was primarily performed to demonstrate that DocReformed was capable of handling a PDF of this size without crashing or becoming unstable, even if the performance is noticeably worse than with smaller PDFs. Other PDF sizes took far more reasonable times to split, with twenty thousand pages taking 3.6 seconds, forty thousand pages taking 11.7 seconds, and one hundred thousand pages taking 51.1 seconds. The stakeholders requested that splitting a PDF of approximately one thousand pages take approximately one minute to complete. These tests show that this requirement has been met.

Other tests were performed to test invalid inputs for the PDF file that was passed to the PDF splitting function, the output directory passed to the same, and the maximum page count specified by the user. These were easily caught by DocReformed in less than one millisecond. This ensures that, if invalid arguments are passed to the PDF splitting function, it will not behave unexpectedly or cause DocReformed to crash.

Additional tests were conducted on the polling system. The first test (source code seen in figure 28) shows that the polling system is capable of archiving PDFs that were passed to them, then passing the PDF to the splitter. The second test (figure 29) demonstrates that, when a PDF is written to a directory watched by the polling system, it will process the PDF. The third test (figure 30) shows that multiple files can be concurrently processed without causing unexpected behavior.

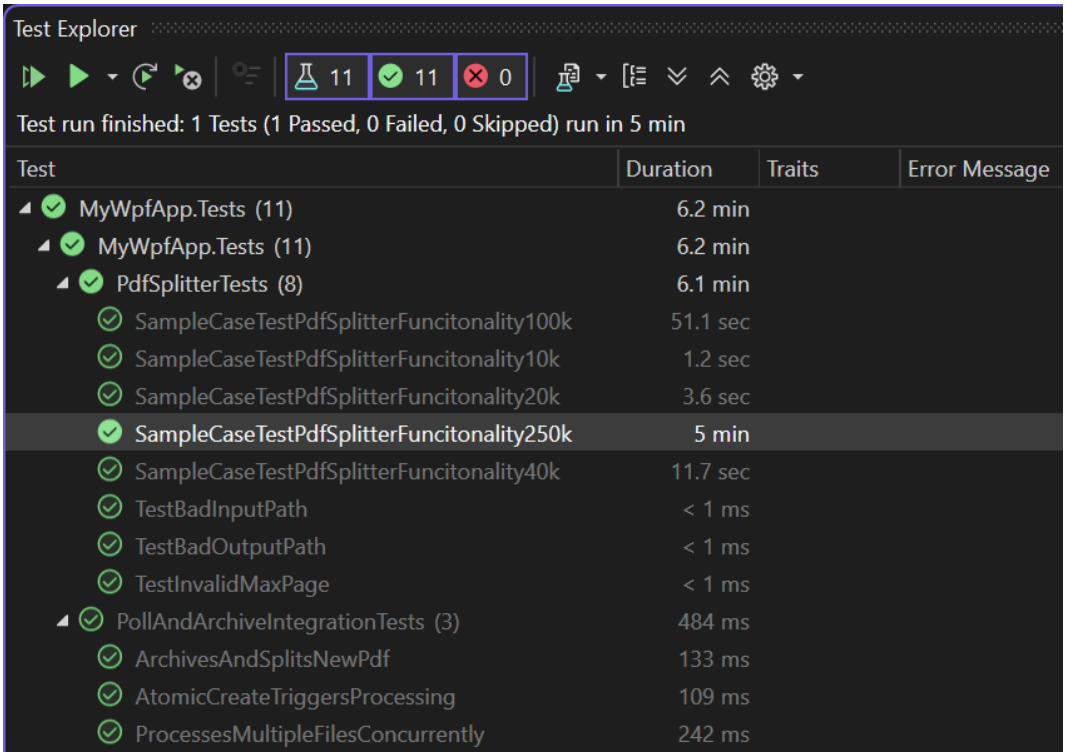

Figure 27: Results of all tests performed on DocReformed.

```
[Fact]
| 0 references
public async Task ArchivesAndSplitsNewPdf()
{
    var poller = new PollAndArchive(_inputDir, _archiveDir, _outputDir);
    try
    {
        poller.StartWatching();

        // Write to temp outside watched dir, then move in
        var temp = Path.Combine(Path.GetTempPath(), Guid.NewGuid().ToString() + ".pdf.part");
        CreateValidPdf(temp);

        var target = Path.Combine(_inputDir, "test.pdf");
        File.Move(temp, target);

        // Check for archived copy inside
        var archived = Path.Combine(_archiveDir, "test.pdf");
        var archivedAppeared = await WaitForFileAsync(archived, timeoutMs: 10000);
        Assert.True(archivedAppeared, $"Expected archived file to appear: {archived}");

        // Check for split PDF (at least 1) in output dir
        var splits = await WaitForFilesAsync(_outputDir, "*.pdf", timeoutMs: 10000);
        var outputFile = Path.Combine(_outputDir, "test.pdf");
        Assert.True(File.Exists(outputFile), "Expected the PDF to be moved to the output directory.");
    }
    finally
    {
        poller.Dispose();
    }
}
```

Figure 28: Source code for test demonstrating archive system.

```
[Fact]
| 0 references
public async Task AtomicCreateTriggersProcessing()
{
    var poller = new PollAndArchive(_inputDir, _archiveDir, _outputDir);
    try
    {
        poller.StartWatching();

        // Write a PDF to a temp subdir and then move into watched dir
        var tmpDir = Path.Combine(Path.GetTempPath(), "pa_test_tmp_" + Guid.NewGuid());
        Directory.CreateDirectory(tmpDir);
        var tmpFile = Path.Combine(tmpDir, "testpollandarchive.pdf");
        CreateValidPdf(tmpFile);

        var target = Path.Combine(_inputDir, "testpollandarchive.pdf");
        File.Move(tmpFile, target);

        // Assert archive exists
        var archived = Path.Combine(_archiveDir, "testpollandarchive.pdf");
        Assert.True(await WaitForFileAsync(archived, 10000), "Archived PDF did not appear within timeout.");

        var outputFile = Path.Combine(_outputDir, "testpollandarchive.pdf");
        Assert.True(File.Exists(outputFile), "PDF was not moved to the output directory.");
    }
    finally
    {
        poller.Dispose();
    }
}
```

Figure 29: Source code for a test showing that PDF creation will trigger processing.

```
[Fact]
| 0 references
public async Task ProcessesMultipleFilesConcurrently()
{
    // Arrange
    var poller = new PollAndArchive(_inputDir, _archiveDir, _outputDir);
    try
    {
        poller.StartWatching();

        const int fileCount = 3;
        var fileNames = Enumerable.Range(1, fileCount).Select(i => $"multi_{i}.pdf").ToArray();

        // Checking if multiple files can be handled at once
        foreach (var name in fileNames)
        {
            var tmp = Path.Combine(Path.GetTempPath(), Guid.NewGuid().ToString() + ".pdf.part");
            CreateValidPdf(tmp);
            var dest = Path.Combine(_inputDir, name);
            File.Move(tmp, dest);
        }

        // Wait longer for multiple files
        var sw = System.Diagnostics.Stopwatch.StartNew();
        var succeededAll = false;
        while (sw.ElapsedMilliseconds < 15000)
        {
            var allArchived = fileNames.All(n => File.Exists(Path.Combine(_archiveDir, n)));
            if (allArchived)
            {
                succeededAll = true;
                break;
            }
            await Task.Delay(200).ConfigureAwait(false);
        }
        Assert.True(succeededAll, "Not all files were archived in time.");

        // Verify that for each file at least one split PDF exists in output dir
        var outputs = Directory.Exists(_outputDir) ? Directory.GetFiles(_outputDir, "*.pdf") : Array.Empty<string>();
        Assert.Equal(fileCount, outputs.Length);
    }
    finally
    {
        poller.Dispose();
    }
}
```

Figure 30: Source code demonstrating successful concurrent processing of PDFs.

8.2. Stakeholder Feedback and Analysis

Stakeholder feedback on DocReformed has been very positive throughout its development. Demonstrations of the progress made, the features added, and the new ideas put forth by the development team have almost always been well-received. Those that were not were never immediately cast aside by the stakeholder, but rather led to discussions about more efficient solutions. CPS was very helpful to the development team by holding numerous internal discussions about the features that would be most useful to them, along with suggestions about how the development team should implement them. CPS made efforts to distinguish their requirements between what was necessary, what would be useful to have, and what was optional. The necessary requirements, and some of the more useful requirements, were included in DocReformed.

Efficient elicitation of stakeholder requirements (see section 2.1) helped make development much easier. Whenever the development team had a question regarding requirements or a particular feature, CPS was consulted before a decision was made. Frequent meetings with the stakeholders were also beneficial to the development process.

9. Conclusion and Future Work

Section 8 gives an overview of the results of DocReformed, both in terms of testing and in stakeholder satisfaction. Overall, the system performs all major functional requirements that were requested by CPS. It is capable of accepting large PDFs, splitting them efficiently, allowing users to manipulate the jobs, and queuing the jobs to printers on the LAN. It is also capable of running on modern systems and Windows 11, using the requested C# framework and version, all while using a streamlined yet familiar user interface. As such, the primary objective of the project has been satisfied, and a solution applicable to the problem statement has been provided by DocReformed.

In the future, those continuing to develop and maintain DocReformed may wish to improve the efficiency of the PDF splitting process even further. In addition, a robust logging system may be desirable for more efficient tracking of user actions, as well as for troubleshooting purposes. The level of logging is entirely at the discretion of CPS based on their requirements.

It may also be the case that niche features that were rarely used with Doc Transform may be needed, yet missing, from DocReformed. Such features would not have been provided to the authors as functional requirements during the planning and elicitation stages of this project. The developers may need to add such features, although the authors are confident that this will be a rarity due to all major functional requirements already being satisfied by the software.

10. Glossary

Table 2 below is a glossary of technical acronyms, abbreviations, and terms that have been used in this technical report, sorted alphabetically. When possible, examples are provided within the definitions for added clarity.

Glossary of Terms and Definitions	
ACM	Association for Computing Machinery
CPS	Shortened form of CPSstatements.com, the project stakeholder.
CSF	Cyber Security Framework, as provided by NIST.
Doc Transform	The legacy print management software used by CPS, replaced by DocReformed.
DocReformed	The new print management software developed by the authors, replaced Doc Transform.
DTO	Data Transfer Object
GUI	Graphical User Interface
GUI	Graphical User Interface
GitHub	The development and version control platform used by the developers to share and update DocReformed's code.
HIPAA	Health Insurance Portability and Accountability Act of 1996, a federal law that protects PHI within the United States of America.
LAN	Local Area Network
MIT	Massachusetts Institute of Technology, the institution responsible for creating the MIT license that was applied to DocReformed.
NIST	National Institute of Standards and Technology, which provided the CSF.

OA&M	Operations, Administration & Maintenance
PDF	Portable Document Format, a standard for representing documents as a digital file.
PHI	Protected Health Information, protected by HIPAA.
PII	Personally Identifiable Information, such as a name or home address.
Pipe and filter	A software architecture consisting of pumps (data sources), filters (systems that process data), sinks (data destinations), and pipes (methods of transferring data).
Print management software	Any software used to create, manage, and queue print jobs (such as Doc Transform and DocReformed).
SLA	Service Level Agreement, such as CPS's agreement to print and mail statements on their clients' behalf promptly.

Table 2: Glossary of technical acronyms, abbreviations, and terms.

Acknowledgements

The authors wish to extend their gratitude to Kyle Richmond, Vice President of National Sales & Client Care at CPSstatements.com, for providing them with the requirements and objectives for this project, as well as for his willingness to keep in regular contact with this team throughout the development cycle.

Additionally, the authors would like to thank Kim Trenner, Compliance Officer at CPSstatements.com, for her insights regarding the compliance requirements of DocReformed and the regular operation of Doc Transform.

Special thanks to the rest of the staff members at CPSstatements.com for their internal discussions and feedback during DocReformed's development, without which DocReformed would not have been able to satisfy many of the more nuanced requirements expected by the CPSstatements.com team.

Finally, the authors wish to thank Dr. Augustine Samba, Professor of Computer Science at Kent State University, and Hasan Shamim Shaon, Graduate Assistant in the Department of Computer Science at Kent State University, for their support and guidance throughout the CS 49999: Capstone Project course. Their lectures, resources, and feedback have been valuable assets to this team.

References

CPSstatements.com. (2025). *CPS statements - statement processing and document creation*. CPS Statements - Statement Processing and Document Creation. https://www.cpsstatements.com/

Gotterbarn, D., et al. (2018, June 22). *ACM Code of Ethics and Professional Conduct*. Association for Computing Machinery. https://www.acm.org/code-of-ethics

Microsoft Corporation. (2025, August 19). *Exception-handling statements - throw and try, catch, finally*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements

Open Source Initiative. (n.d.). *The MIT License*. Open Source Initiative. https://opensource.org/license/mit

U.S. Department of Commerce. (2024, February 26). *The NIST Cybersecurity Framework (CSF) 2.0*. National Institute of Standards and Technology. https://csrc.nist.gov/pubs/cswp/29/the-nist-cybersecurity-framework-csf-20/final