Increasing the robustness of a service in a complex information flow
Abstract

In complex information flows where a lot of varied data is transmitted through many companies and divisions, incidents will occur. When Visma Spcs had an incident were invoices sent from Visma to Visma's customers were duplicated and the service meant to receive the transactions did not handle the duplicates properly. They decided that the receiver service was to be upgraded to prevent this incident from happening again, as well as fixing some other issues the service had had. Incidents like this one must be investigated and a solution must be implemented to decrease the likelihood that similar incidents will happen again. In this report, the reader will see examples on how this can be handled and the benefits of tackling technical debt, along with how much more complicated the solutions might get if the service is not allowed to be taken offline.

**Keywords:** Technical debt, incident handling, developer morale, cloud service, Zero Downtime Deployment.
Preface

I would like to thank my external Supervisor Alexander Tunroth for all the time he put in inspecting my work so that my work could be put in the live production environment.
1 Introduction

Big software companies like Visma Spcs handles and transfers a lot of varied data between many machines. This data variety makes the systems complex and a single service deployment that introduces a bug can paralyze big parts of the network it belongs too. To prevent this, the services must, among other things, not trust incoming data and keep running even when malformed and duplicated data reaches the services. Making changes to these systems can be hard since they are only allowed to be taken offline if it is not absolutely necessary as this disrupts the network and affects other services that rely on the information that the offline service provides.

In complex systems containing many services with a large customer base, things will go wrong. And when an error occurs that makes a service unavailable, a quick fix is often implemented to get the service up and running and to minimize downtime for the customers. Also, large services need maintenance with time as they have a lot of infrastructures that need to be kept up to date.

In January 2018, Visma Spcs had an incident. A Service at Visma Spcs came under heavy load with incoming traffic from its partner’s service. The partner was sending duplicate transactions to the service at a high rate. When it was noticed at Visma Spcs the incident had already been resolved in the partner service. Since the team was currently on a deadline to finish another project, the fact that the partner had resolved the incident was enough at the time.

However, when a time came where the team had a relatively low amount of deadlines and a new developer resource had joined the team. It was a great opportunity to implement a solution that would protect the service from similar incidents, as well as solving some other issues.
1.1 Background

1.1.1 Simple overview of the system.
To better understand the implementation, it might be useful to know how the information flowed through the service.

Visma has a program where our customers can send invoices to their customers via our partner InExchange. Our customers can also receive from our customer's customers.

![Diagram](image)

Figure 1.1.1.1
If our customers don’t want paper invoices sent home to them and then having to type them in manually, they can give their customer an address that goes to our other Partner CogiDocs. CogiDocs is a scanning central that scans papers. This way our customers can get all their invoices straight into their Visma program.

Figure 1.1.1.2

Figure 1.1.1.1 and Figure 1.1.1.2 show the part of the system that handles the customer's invoices. Here all invoices have to reach its intended target since losing a customers invoice would be very bad for their trust in the system and Visma's reputation.
Continuously while our customers are sending and receiving invoices, InExchange keeps track of everything that is sent with their system and sends this information to Visma so that they can bill their customers.

The service that receives this billing data is called Autoinvoice and is the service that this report will be all about.

After AutoInvoice has received the billing data (also known as the transactions or invoices that our customers will pay for.), they are sent to the central hub called Visma Online.

Every month, Visma Online takes all the transactions that have been received in the past month and put them all on a single invoice per customer. Then the customer gets the invoice and has to pay for the things that have been sent. This is demonstrated in figure 1.1.1.3.

![Diagram](image)

Figure 1.1.1.3
There is also another way the service AutoInvoice receives billing data. When our customer's customers send things to the CogiDocs address to get them scanned, sometimes the items received cannot be scanned. For example, they have received bricks and bicycle helmets in the past these are not ordinary scanning articles in the primary invoice source system from InExhange.

These items are sent to our customers and CogiDocs keeps track of what is sent and the cost. Every month they put up a file with all the items that our customers will have to pay for. This file is downloaded by our service and the transactions in it are sent to Visma Online for billing as shown in fig. 1.1.1.4
1.1.2 The Incident

In January 2018, our partner InExchange had an incident where they bombarded our service with duplicates transactions. These duplicates slipped through our service.

Figure 1.1.2.1 The information flow on how InExchange sent the duplicate transactions.

1. InExchange Contacts AutoInvoice and tries to transfer the billing data.
2. AutoInvoice tries to store the data, but since it is under heavy load by multiple calls from InExchange it took a long time to save to the database.
3. While the database saves is underway, the call from InExchange times out after a certain amount of time and notes down that the billing data it tried to send could not be received. It will now wait some time before it tries again.
4. The AutoInvoice service saves to the database finally finishes and returns an OK to the AutoInvoice service.
5. Since the call from InExchange has timed out, there is no way for the AutoInvoice service to tell if the transactions were saved or not.
6. The AutoInvoice service transfers the transactions to Visma Online and after the transfer is completed it deletes the transactions from the AutoInvoice database.
7. Some time later, InExchange decides to send the transactions again.
8. However, since the transactions have already been deleted, AutoInvoice once again saves these transactions and this time the service might answer the call before the timeout.

1.1.3 Other problems with the service.

The service already had problems with the file from CogiDocs. Since CogiDocs could not scan the items from the secondary invoice source they had to put in the information by hand into the system. Since there are about 2000 items every month, human error will occur.

This made Autoinvoice unable to parse the file and add it to the database. So each month a developer had to check with Visma Online if the bills were even for our customers and check with CogiDocs what had happened. The service also had a bug where it would truncate the file if it was too large, once again making the AutoInvoice service unable to parse the last line and of course missing all transactions in the last part of the file that was lost.
1.1.5 Detailed overview of the billing system

Note that this figure does not show how the system sent and received our customers invoices, but how we made sure that our customers paid for the invoices they sent and received.

CodiDocs
(Start Secondary Invoice Source)

Visma AutoInvoice fetches a file from CodiDocs via FTP containing invoices for invoices that could not be scanned, but the customer still had to pay for. (For example if one of our customer had sent a bicycle helmet to CodiDocs scanning central, and CodiDocs had then forwarded that helmet to our customer. Then the customer would have gotten an invoice from us where they would have had to pay for the shipment.

Visma AutoInvoice
(The service this report is about and all changes are made)

Batches with invoices that our customers had to pay for were sent to Visma AutoInvoice from InExchange via AutoInvoices API.

InExchange
(Start Primary Invoice Source)

All Invoices were forwarded to the Visma data hub

Visma Online
(Invoice Endpoint)

All Invoices for a single customer were put together into a single invoice. On it the customer could see all invoices that the customer had sent and received with our service under the last month.

Customer

Figure 1.1.5.1 An overview of how the invoices flow to our customers.
Figure 1.1.5.2 describes how the invoices flows are received by the system when the service receives invoices from the primary invoicing source:

![Diagram showing the process of receiving invoices]

Figure 1.1.5.2. A more detailed view of how the service received the invoices from the primary invoicing source before the new solution was implemented. Internally the invoices that the service receives were called transactions. So when the reader of this report reads transactions, it is the invoices that we had received and was to be sent to Visma Online to be clumped together and be billed to our customers, while invoices are the ones that our customer receives and sends to our customer's customer. Since this report only covers the transactions part of the service, whenever the reader reads invoices or transactions, they are interchangeable.
Figure 1.1.5.3 describes how the invoices were received by the system when the service receives invoices from the secondary invoicing source:

Figure 1.1.5.3 A more detailed view on how the service received the invoices from the secondary invoicing source before the new solution was implemented.
Figure 1.1.5.4 describes how the invoices were transmitted to Visma Online, this is the same procedure for both the previous tasks 1 and 2:

Figure 1.1.5.4. Here we can see how all transactions are sent to Visma Online.

When the service transmitted the transactions to Visma Online. It was unknown if the transaction were saved or not because, for example, the call to Visma Online might have timed out before Visma Online's call to their own database timed out. This means that after the transaction was saved in the database, Visma Online could not respond with an “OK” since there was no connection to respond to, or they actually did fail to save the transaction but could not respond with an “ERROR” either.
There are two fields in the database table relevant to this upgrade: NextTransfer and TransferSuccessful.

NextTransfer told the system the next time it was supposed to try and transfer the transaction after it had failed. In the beginning, this was used to re-transfer transactions, that had failed, to Visma Online. Although, over time, with changes to the service and other services connected to this service, there was no longer a certain way to know if a transaction marked with NextTransfer had reached Visma Online or not. This changed the NextTransfer field from a date where it should retry, into a boolean-like field that was set to a high date like the year 9999 just to prevent the system from retrying. This was probably a way to get a change out quick without changing any of the code. Making it a safe and very quick way of disabling the feature.

TransferStarted was set to the time that the transfer to Visma online was started.

1.1.6 Words and concepts used in the report

- Software deployment is all of the activities that make a software system available for use [1].

- Service Owner is the person “responsible” for the service. She/He gets directions from people who in one way or the other have a stake in the service and plans the future of the service with the stakeholders amongst other things [2].

- Service Architect / System architect is a person in a team that is responsible for how solutions for the service are architected [3].

- A heartbeat is a way of telling if a service is available or not [4]. For example, let's say you have an API towards your customers. You would want a way to see if the service can be reached. This can, in turn, be used with for example a service like Pingdom. Pingdom is an availability and performance monitoring system. It can be used to ping a heartbeat to see if a service is available. If the service returns an error or is unavailable then an alert can be sent to the team members with for example mail from Pingdom.
• A Configuration Transform is a way of having multiple setting files at the same time [5].
So, for example, a service hosted in many environments can have the need for many different configurations. In Visual Studio the user can then choose which configuration to use when running the service so that the user can target for example a local database instead of the production environment database.

• A database index increases the speed the database can access data at the cost of taking more space and having to perform more writes to maintain the index. [6]

• High availability is when a service is available a lot more than it is not. For example, if a service is required for a customer to use a company's product then the service needs to be available practically all the time. To make sure that the service is available during deployments something called Zero-downtime deployment is used [7].

• Zero-downtime deployment is when you upgrade the code of a project without the users noticing. This is done, for example, by starting a new instance of the code while the old code is still running. When the new instance is fully operational and has been thoroughly tested, the workload balancer will stop pointing new requests to the old instance that runs the old code and instead point at the new instance so that the new code handles all future requests to the service. When the old instance finishes all requests in its queue it is terminated [7].
This way of deploying can make things difficult. For example, since both the old and new code can run against the same database at the same time it is important that the database is compatible with both. In cases like this, it can, for example, be possible to add new rows and tables to a database, but not delete old rows and tables. In cases like this, a secondary deploy is needed to remove the old fields.
1.2 Related work

No related work was found to such a specific problem as the one that was implemented in this report. An article on the negative effects technical debt can have on developer moral was found [8]. There were also articles about how technical debt can be used to reach a deadline and to deliver features to customers quicker [9, 10].

Related work regarding upgrading a database while making it backward compatible was found [11, 12].

No articles were found regarding the use of technical debt to get a service running again after an incident and paying off the debt in the near future.

1.3 Problem formulation

Visma Spcs had an incident where one of their services received duplicate invoices from one of their partners. The service had recently received bad data through their secondary invoicing source from another partner that required developer time each month to correct. With these two incidents, Visma Spcs decided to upgrade the service so that it would run more smoothly without the interaction of a developer and prevent the duplicate transactions from entering the system.

Two goals were set:

1. Make sure that the services primary invoice source can handle duplicate invoices and still continue execution without missing any incoming invoices.
2. Make sure that the services secondary invoice source can handle errors in the data without requiring the need for human intervention.
Since we receive invoices from different sources in different formats and the data we receive from the secondary source often has unparsable data we can’t expect a perfect result. Attempts will be made to estimate how many invoices are lost. Since these are invoices that our customers would have paid Visma, it is not worth the time of having a developer fix a small amount of unparsable invoices. Other than that the two invoicing sources should be free from regular maintenance and instead of having to be acted upon based on monitored metrics.

Hopefully, the service will be updated without having to turn it off as a release where the service has to go offline will have to be done overnight and force the developer team to be onsite during the deployment. There is a problem where the data from the secondary invoicing source is being truncated and this problem will be fixed.

1.4 Motivation
When dealing with Accounting services, things need to be stable. If this incident had not been noticed, customers might have received invoices were they would have had to pay double for the invoices that they sent and received with Visma's service. Customers need to trust Visma's service if they are to use it to handle their accounting. Even though this incident did not affect the invoices from customer to customer or any other feature of the program, an error like getting an invoice from Visma with an incorrect billing amount might have made the customer lose trust in the program and the company.

In order to have customers allow a company's service to handle their own companies accounting, a lot of reliability is required. If the service is too often unavailable they might also lose trust for the system.

Other than customer value and trust, wanting to upgrade a service that takes a lot of manual maintenance is also natural. Doing manual maintenance work regularly can distract developers from issues that require their expertise. The time lost is probably the main reason many companies choose to upgrade a service as a time investment now has been assessed to pay off in the long run. However, something that might get overlooked is that performing very repetitive work is boring. In an economy where there are more developer jobs
than developers, a company might lose bored developers to other companies. Since Visma Spcs believes in fingertip decision making where the developers working on the project also knows what needs to be done/maintained, it was only natural for Visma Spcs to follow the developer's suggestion to upgrade the service. Hopefully, the result can be used as a helpful guide on how a service in an information flow can be constructed to increase robustness in other similar setups.

1.5 Objectives

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>O1</strong></td>
<td>Investigate how the primary invoice source works and suggest at least one solution on how to handle duplicate invoices so that the service can continue to provide its function.</td>
</tr>
<tr>
<td><strong>O2</strong></td>
<td>Implement the appropriate solution suggested in O1.</td>
</tr>
<tr>
<td><strong>O3</strong></td>
<td>Investigate how the secondary invoice source works, investigate if the solution for the primary invoice source can be reused and suggest at least one solution on how to handle duplicate invoices so that the service can continue to provide its function.</td>
</tr>
<tr>
<td><strong>O4</strong></td>
<td>Implement the appropriate solution suggested in O3.</td>
</tr>
<tr>
<td><strong>O5</strong></td>
<td>Investigate why the data from the secondary invoice source is getting truncated when it is read.</td>
</tr>
<tr>
<td><strong>O6</strong></td>
<td>Implement a solution that prevents the secondary invoice source from being truncated.</td>
</tr>
</tbody>
</table>

When the new solutions are implemented. I expect duplicates will be prevented from entering the system and that the truncation problem will be fixed. As well as that all manual transaction handling regarding this service will have been removed.
1.6 Scope/Limitation

Since this was a large service with carefully selected extensions that must fit the needs of all developers in the project, there were as little infrastructure changes as possible. For example, even if a change from the current database integration framework to another framework that handles removing database items in bulk better had been implemented, it would have been too big for this project. Also, the service might have gotten a framework that would have worked worse than the then current installed framework in other parts of the code.
1.7 Target group

This report can be read by all types of developers to see examples of how someone can make a Zero Downtime deployment with database changes in a cloud service environment like Microsoft Azure. It can also be read by team managers or Service Owners as the report also shows the benefits of taking care of technical debt and the benefits it can have on morale.
1.8 Outline

Chapter 1 is the Method chapter. Here the method used to investigate and solve the problems will be presented. The chapter also contains the validity and reliability of my results and how they were reached, along with chapter 2 discussing ethical considerations.

The different solutions paths and the final solution will be presented in chapter 3, the implementation chapter. It contains 5 different sub-chapters where the implementations along with why the implementations were chosen are presented.

Chapter 4 is the Result chapter where the results are presented. This is then followed by chapter 5, the Analysis chapter, where the results are given more meaning, closely followed by chapter 6, the Discussion chapter, where self-reflection on the work will be presented.

Last of all is chapter 7, the Conclusion chapter, presenting the conclusions reached as well as future work to be done on this topic.
2 Method

2.1 Verification and Validation

The two requirements mentioned earlier with the second one split into two:

R1. Make sure that the services invoice sources can handle duplicate invoices and still continue execution without missing any incoming invoices.
R2. Make sure that the services secondary invoice source can handle errors in the data without requiring the need for human intervention.
R2.1. Make sure that the FTP file clipping problem is fixed.

2.1.1 Verification

| T1: R1: | The test was executed from a test client pretending to be the primary invoicing source. After the invoices had been transferred a manual login to the test database will be performed to check that invoice duplicates received within a year of the original invoice are not added again to the database. |
| T2: R1: | Make sure that the primary invoice source does not stop the invoice transfer even if it receives a duplicate invoice. This test will be the same as T1, although this time there will just be a check to see that no matter if we send a duplicate transaction to the service or not, the service will just continue receiving. |
| T3: R2: | Test that the secondary invoice source does not interrupt the invoice registration process when missing or bad data is present. This is tested by manually writing a file with test data and letting the service process it. |
| T4: R2: | Make sure that the secondary sources invoice registration process alerts the developers if the estimated error rate is more than 3%. This is tested by manually writing a file with test data and letting the service process it. |
| T5: R2.1: | Test that the FTP scanner can read double the amount of data than it currently can. This together with a proof that the code theoretically should be able to read an infinite amount of data will be enough to prove that the implementation works. |
2.1.2 Validation
The three developers in the team should not have done any manual work in the 6 months after the service was updated.

2.2 Reliability and Validity
Since this is a relatively specific service, everything would need to be the same to reach the same result. (The same service, the same surrounding services, the same system resources, etc). This project describes solutions that worked in this specific service in this specific environment with theses specific partner interactions. The solutions chosen in this project might not be the appropriate solutions for other similar projects.

Even if someone else had followed these verification steps on this very service, they would almost certainly not have reached the same implementation. However, they would probably have reached the same result with their own implementations since the Verification tests specify well what needs to be done.

2.3 Ethical Considerations
Since the transactions are now saved instead of instantly deleted after the transfer, there is a possibility that a database leak would reveal what kind of transactions and how many transactions a customer has sent the last year. The transactions are only saved for as long as it is necessary and are then deleted. The information is already stored in Visma Online so no new data is collected, just the same data in a new place.
3  Implementation

3.1  Primary invoicing sources duplication prevention

The first thing implemented was the duplication prevention designed to prevent further incidents like the one that had already happened and was the main reason for this project.

Two simple statements were decided for how this should be implemented:

- Add a unique constraint on column TransactionId and do not delete transactions after they've been transferred.
- Add a value TransferCompleted to the database table.

Two solution suggestions for how this was supposed to be implemented was presented to the external supervisor and the team's system architect:

Solution suggestion 1:
Bundle the two fields TransferStarted and NextTransfer together with TransferCompleted and make the field a statusField. NextTransfer should, in this case, be renamed TransferFailed since it would be a more accurate name. A TransferPending enum should be added in the code instead of comparing with null as was currently the case since this is more descriptive.

This would result in a field with the Statuses:
- TransferPending
- TransferStarted
- TransferCompleted
- TransferFailed

This solution would make it easy to add new statuses in the statusField if needed in the future, but removing the fields TransferStarted and NextTransfer would have been a change in the database and would also have forced us to change the way we handled the transactions when sending to Visma Online.
Solution suggestion 2:
Adding TransferCompleted as a separate field.
The way AutoInvoice Transfers to VON would be completely unchanged and we knew it would continue to work exactly as it was. However, adding a separate Field for this in the database would have left us with three fields that could have been a single field and a more difficult code to read and might have been bad practice.

After much discussion, the first solution suggested was chosen.

The fields transferStarted and nextTransfer was changed into an enum named TransferStatus with the values:

- Pending
- Started
- Completed
- Failed
Changing the database of a high availability service that is always running might be problematic. When a new version of the service was deployed, it was done in many steps with a deployment tool called Octopus Deploy. The two most important things to know about the deploy was that the database was upgraded approximately 20 minutes before the code was. This left some special scenarios where the old code ran against the new database as described in figure 3.1.1.

![Diagram](image)

Figure 3.1.1 The Octopus deploy deployment procedure.

Since the code and database are updated like the scenario in Figure 3.1.1, fields could be added in the first deploy but not be removed since the old code had to be able to run against the new database. A lot of testing had to be done to make sure that for example, the database could handle missing data in
the new fields while the updates were being made. When the old code was running on the new database, the code was still deleting all transactions that had been successfully transferred to Visma Online from the new database. This means that every transaction that did not have a value for the statusField had been received and created by the old code and when the new code was swapped in the new code did not send/handle a transaction that did not have the transferStatus field set in the database. Therefore a second release with a conversion script had to be made. The two fields that were removed with the conversion script were TransferStarted and NextTransfer. Since they both acted like booleans there were 4 different states that a mid-deploy created invoice could have.

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Next Transfer</th>
<th>Transfer Started</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>not set</td>
<td>not set</td>
<td>There was no attempt to transfer this transaction, probably because there were other transactions in the queue before this one. A transaction with this combination was converted to Pending.</td>
</tr>
</tbody>
</table>

| Scenario 2 | set           | not set          | The transfer failed and the transaction was set to the codes way of saying “Failed”. A transaction with this combination was converted to Failed. |

| Scenario 3 | not set       | set             | The transfer was started, but there was no way of knowing if the transfer was interrupted or if the code did not have time to delete the transaction before the code was swapped and shut down. The code should have been allowed to end all jobs before it was shut off, but if this kind of transaction was present then the code did not gracefully stop executing. If this type did occur in the |
database then at least there should only have been one as the code only handled one transaction at a time. It was better to be safe than sorry and set it to error if it somehow did exist. A transaction with this combination was converted to Failed.

<table>
<thead>
<tr>
<th>Scenario 4</th>
<th>set</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since the code always did set the TransferStarted status to Null before the NextTransfer was set, this should neither have been present in the database. If it somehow did exist transactions like this, then they were all set to Failed.

Figure 3.1.2 The four states a transaction could have in the old system.
When a way to store old transactions had been implemented, the next step was to implement how to handle an incoming duplicate transaction. Once again, two main possible solutions on how the service should handle receiving duplicate transactions were presented to the external supervisor and the team's system architect.

**Solution suggestion 1:**
Save the transactions one-by-one to the database. This would have taken a longer time to execute since AutoInvoice would then have had to make a database connection for every transaction, but the solutions would have been fast to implement.

**Solution suggestion 2:**
Save the transaction in batches. Only one database connection would be needed for every batch. However, if a transaction already exists then it would be hard to read out from the error which transaction that is a duplicate. If there are multiple duplicates then the transaction batch would have been sent many times to the database since the database would have returned an error as fast as it found a duplicate in the batch without checking the remaining transactions.

The solutions were discussed among the three of us.

We concluded that if we were to send in the transactions in batches to the database, the database would have aborted the whole batch save if it were to find a single duplicated transaction. This would have been fine under normal circumstances since we received very few to no duplicate transactions in the everyday operation of the service, but we had learned from previous experience that the service had received most of the duplicates when the database had been overloaded.

When the service received a call from the primary invoice source the call would timeout since the database was overloaded. However since only the API call to us from the primary source had timed out, the database was still in the process of inserting the transactions into the database. So when the invoicing source retried to send the same batch again it was accepted once again and also started to insert the batch into the database, since the first
transaction still might not have been added to the database. Also since the service deleted all transactions that had already been received. The service could receive the same transaction batch the next time our partner tried to transmit since they had noted that the last call had timed out. The transactions would already have been deleted so the transactions could be inserted again and transferred to Visma Online again.

The service also had a bad practice/technical debt way of checking if the transaction already existed in the database. Before an invoice was added to the database, the code requested the same invoice from the database. If the database could not find the transaction, the code added it to the database. Since a database does jobs concurrently, the database SELECT was a quick operation that returned that these transactions were not in the database, when in reality they just had not been inserted yet and the database was still trying to insert them.

It is unclear why this approach was chosen. Perhaps they wanted all logic to be handled in the code or perhaps more likely this was a thing done to get a quick fix out, or perhaps to meet a deadline.

After discussing the solutions, solution 1 was chosen since the database can handle the load and we believed that saving them one-by-one would make errors less likely since we are most worried about the database when it is under high load.
Figure 3.1.3. A figure demonstrating how the service received invoices from the primary invoicing source with the new statuses and the new way of handling exceptions from the database.

The check was switched out for a unique key constraint together with an index on the ID of the transactions. Also, a new code was implemented that handled the SQL exception that was thrown from the database when a duplicate transaction was attempted to be saved. This was tested by sending in batches with duplicate transactions at a high frequency. The same scenario that occurred during the incident was not properly simulated as it was too hard to reenact.
3.2 File truncation problem

A thorough investigation of the code and on one of the truncated files was done. The file was investigated first since the file transmitting process could at the time not be run and tested on a local debug environment. It was discovered that the file was cut at approximately 940 lines, and each line was roughly 110 characters. Because of this new finding, it was suspected that a full buffer was not flushed before it was closed.

When the reasonable problem cause was found, it was deemed that a test would need to be done on a local computer to make sure this was the problem. This since testing in the Azure test environment would have required help from Visma Operations who were the only ones allowed to handle the Azure environments.

The local environment was set up by editing all the services settings and creating a configuration transform for local debugging. This local transform targeted a locally FTP installed on the laptop that the tests were supposed to run on. The error was found by debugging the code. The memory stream created by the file reader was transformed into an array by the command .ToArray().

Since streams have a buffer so that they do not read the whole file into memory, the stream was full when the ToArray() command was called. This was why the file got truncated, the streams buffer was defaulted to 100 KB and emptied into an array. The buffer was then no longer empty and started reading in data from the file again. However, the stream was closed and the remainder of the file was not read.

The flow through the parser and database saver was redesigned to handle the data with a stream all the way, instead of converting the stream to an array.
3.3 Secondary invoicing sources duplication prevention

The same solution suggested for the Primary invoicing source was also implemented for the secondary invoicing source, but in this case, three other parts also had to be changed in the code regarding the handling of the FTP transmitted file from the secondary invoicing source.

Figure 3.3.1 An overview of the service after the solution was implemented.

First came the parsing step. Here the service tried to parse whatever was in
the received file. The total number of lines was noted along with the total number of successfully parsed lines.

Then the service tried to save the lines. Since the solution for the primary invoicing source also was deemed to work here it was implemented. Now the number of lines that were able to be saved to the database was noted. Out of this came a calculated error percentage.

The Service Owner was presented with information on how much money was involved in the missing transaction versus how much time and the approach the developers would have taken to fix the errors in the file. The Service Owner then presented the appropriate error rate where cost versus work had been weighed. The error rate decided by the Service owner was 3% and it was agreed upon that this was a suitable number.

If the file contained more than the allowed error percentage, the developers were alerted by saving a periodic task log in the database PeriodicTaskLog table. When the uptime monitoring application then pinged the service heartbeat regarding the success of the file reading, to see that the latest file had been transferred successfully, the code returned a down alert. The uptime monitoring application then sent out a DOWN alert via mail to the developers.

T3&T4:R2: The second invoicing source was tested by making an API test that constructed fake files with specified error rates. To test that the service alerted the developers when the error rate was estimated to be above 3%. Files containing 2.5% error rates and 3.5% error rates were fed to the code. T5:R2.1: Also the file was roughly 10 times larger than what the service previously could handle.
3.4 New implementation for forwarding transactions

Figure 3.4.1. A figure showing the new way the invoices were transferred from the AutoInvoice service to Visma Online after the changes to the service.
3.5 Implementing the CleanupManager for both invoice sources

When the duplication prevention had been implemented in both invoicing sources, the cleanup manager was implemented.

There was already a cleanup task for removing old login tokens. This cleanup was generalized into a cleanup Manager that could be used by any task that wanted to clean up old entries older than a certain amount of days chosen by the separate tasks. This cleanup manager bypassed the Framework installed at that time and performed the batch cleanup job. This because the framework currently installed was made to handle one transaction at a time and not bundles of transactions. A change of database framework would have made the cleanup easier. However, the new framework would probably not have been suited in other places of the code. It was too big of a job to try and change the current framework so no more time was laid on thinking about how this could be done and instead it was decided that this was outside the scope of this project.

A test to see if the cleanup manager could “catch up” if, for example, the cleanup had not been able to clean up for a while because of, for example a bug, was also tested. The cleanup manager was simulated to have not been able to run for three months and then had to delete all transactions in those months.

The cleanup task is already time measured by the service. So all tests conducted on the cleanup manager was measured with the time the tasks reported. This means that the test does not only measure how long the deletion took but how long the whole task took. In the result, only the primary invoice source was measured as the implementation is very similar to the secondary and the primary contains a lot more entries so it was only worth to measure how long the service took to clean up a great number of transactions as this would be a more high load test.

The test was run on a laptop while the user was using it. This was an environment much slower than the one that was to be used in Azure. When the database was filled with 1 year and three months worth of transactions
randomly spread out over all dates, the time to clean up was measured.

The CleanupManager normal case was also tested. The database had been filled with one year worth of test transactions similar to the ones from the primary invoice source with dates spread out across the whole year. The operation to remove older transactions was run every hour by the cloned test service against a local database and the time it took was measured.
4 Results

The three upgraded tasks, as well as the new implementation of cleaning up old transactions, is the result of this report.

Figure 4.1. The finished implementation of the task that receives transactions from the primary invoice source.
Figure 4.2 The finished implementation of the task that receives transactions from the secondary invoice source.

R1 & R2: Tasks 1 and 2 had been implemented for 8 months in the production environment as this report was written and had until then required no manual maintenance from the two invoice sources.

T1:R1 & T3:R2: No duplicate transactions younger than 1 year could enter the database since the new way of checking if a transaction existed was implemented and since the index for unique Transaction ID was added.
T2:R1: The high load duplicate transactions test was passed by the primary invoice source implementation. The new implementation passed the test.

T3&T4&T5:R2: In the second invoicing API test, the file with fewer errors was allowed, transferred and logged as usual while the other file was rejected and the service's heartbeat was set to down along with that the Pingdom check went down and sent alert emails.

When the database was filled with 1 year and three months worth of transactions randomly spread out over all dates, the time to clean up was measured and took 1 hour 39 minutes and 49 seconds.

The normal case cleanup manager Task that is performed every hour took 28 seconds. Note again that this test was conducted on a device with less resources available than the one the service was hosted on.

The tests and results showed so far from the cleanup manager are tests outside the ones defined in the verification phase that could not be predicted before the project was started in the project plan. Below are results that are within the scope or certain requirements and/or test.
# Analysis

Comments on all test designed for the Verification phase:

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: R1</td>
<td>The test was executed from a test client pretending to be the primary invoicing source. After the invoices had been transferred a manual login to the test database will be performed to check that invoice duplicates received within a year of the original invoice are not added again to the database. Result comment: Same as the result comment for T2.</td>
</tr>
<tr>
<td>T2: R1</td>
<td>Make sure that the primary invoice source does not stop the invoice transfer even if it receives a duplicate invoice. This test will be the same as T1, although this time there will just be a check that no matter if we send a duplicate transaction to the service or not, the service will just continue receiving. Result comment: Although the test for prevention duplicate transactions was not exactly like the real incident scenario were the partner sent transactions, duplicates, and non-duplicates, very inconsistently. The newly implemented way of saving to the database was proof enough for my external supervisor that the new service would not let transactions slip past. It does not matter in which order, how far apart and how many batches are duplicates since the index in place in the database will handle all duplicate transactions and throw an error.</td>
</tr>
<tr>
<td>T3: R2</td>
<td>Test that the secondary invoice source does not interrupt the invoice registration process when missing or bad data is present. This is tested by manually writing a file with test data and letting the service process it. Result comment: None.</td>
</tr>
<tr>
<td>T4: R2</td>
<td>Make sure that the secondary sources invoice registration process alerts the developers if the estimated error rate is more than 3%. This is tested by manually writing a file with test data and letting the service process it.</td>
</tr>
</tbody>
</table>
Result comment: None.

T5:R2.1: Test that the FTP scanner can read double the amount of data than it currently can. This together with a proof that the code theoretically should be able to read an infinite amount of data will be enough to prove that the implementation works.

Result comment: The file tested was, in fact, many times larger, but as long as the file read is larger than the buffer and the transfer is handled with streams all the way instead of being put into the arrays as it previously was. This enough proof for my external supervisor that the service can handle files of relevant size (at least 10 times the original size of the file was truncated at).

The fact that no manual work regarding the invoice sources has been done the last 8 months answers the Validation question from the Method chapter.
6 Discussion

The service could have prevented transactions from reentering the service forever instead of just for a year. If for example we had only kept the transaction ID and cleared out all information from the transactions. However, since I was a newly hired junior developer, they did not trust me to make relatively big changes to the service.

We were assured that this would not happen for transactions older than 1 year. Also the time I would have invested in making this solution might not have been worth the time, as other projects could make more use of my time.

The cleanup manager has still not been used in production. There could still be unforeseen consequences. The cleanup manager was tested on a large database with a lot of transactions. However, there could potentially be a difference where the service is running with load (e.g. receiving invoices and such) while the cleanup is made.

Making time for technical debt and bad practice methods can be hard in a high availability service with constant changes, high pace, and deadlines. This is a prime example of that fixing such problems can give developers peace of mind when they don't have to worry about time-consuming manual work.

The cleanup manager cleans with its SQL statement in a string to more quickly finish the batch jobs as the currently installed database framework can’t handle it in a quick and efficient way. This could be considered bad practice/technical debt.
7 Conclusion

O1: Investigate how the primary invoice source works and suggest at least one solution on how to handle duplicate invoices so that the service can continue to provide its function.

The investigation of the invoicing source went as planned and a figure describing how the primary invoicing source worked was presented. Two solutions were presented on how to handle the status fields and two solutions were presented on how to save the transactions to the database. The new cleanup task was presented and deemed necessary.

O2: Implement the appropriate solution suggested in O1.

The service now prevents transactions younger than 1 year from entering again. No matter how overloaded the service is, it would still be able to prevent duplicate transactions from entering. The service also cleans up all transactions that have been in the system for more than a year.

O3: Investigate how the secondary invoice source works, investigate if the solution for the primary invoice source can be reused and suggest at least one solution on how to handle duplicate invoices so that the service can continue to provide its function.

Once again a figure describing how the secondary invoice source worked was presented. The investigation concluded that the same solution that was implemented for the primary invoice source could be implemented here as well, with some additional logging, a heartbeat, and a Pingdom check.

O4: Implement the appropriate solution suggested in O3.

All mentioned in the O1&O2 conclusion was implemented along with the additions mentioned in O3. The service no longer requires manual handling from developers in its everyday operation unless the 3% error limit is crossed.

O5: Investigate why the data from the secondary invoice source is getting truncated when it is read.

The reason for the truncation was found by studying the services code and then a test was made to find out if this was the real reason the truncation occurred.
O6: Implement a solution that prevents the secondary invoice source from being truncated.
A solution was implemented to remove the truncation. It was tested and worked. As mentioned in the result the service has now run for 8 months and has yet to require any manual handling by the developers.

The problem was solved. The goal of this project was to prevent duplicate transactions and end unnecessary manual work for the developers. Since no manual work has been done in the last 8 months (other than support cases for our customers), it is safe to say that for the moment the service has been properly automated to remove manual work. Although all manual handling is gone, things might still go wrong with the service. As mentioned in the discussion, the cleanup manager has still not been used in production and could still have problems.

As discussed before, since the database interaction framework is not suited for bulk saves and bulk cleanup a change to another framework might have allowed me to choose a better solution. However since the current framework works great in other areas of the code, a change would overall have been an unimprovement to the service.
7.1 Future work

As the Cleanup Manager looks now, it uses a string with the SQL script that executes the batch cleanup job. This is because the service is using a database framework that is made to handle database changes that only affect one transaction at a time. If a cleanup job was to be run by this framework, it would have made one SQL statement for every transaction that is supposed to be removed from the database. I would like to see that another framework is joined with the one already installed that could handle the batch jobs better, or that a better framework that could handle both (if this exists) was implemented instead.

I would also like to see the service somehow prevent transactions older than a year. This could be done by for example having the transaction duplicate check in two steps. Say that the first step is only a database table with the Transaction ID and the second check is the one implemented right now. This way the transactions could still be deleted from the database after one year. The database logic would remain fast since the database would just consist of a year's worth of transactions. While the first duplication prevention database compares TransactionIDs with its own table that just contains TransactionIDs and makes sure that it is not the same transaction as one that has already been received.
References


