

How To Break a Monolith

An Example of a Monolith Migration to Microservices


Introduction

This document relates to my contract work at CitiBank. I worked in ICG (Institutional Client Group), essentially the capital markets division of CitiBank. I worked in the research group:

- 800-900 analysts spread around the globe
- publishing anywhere from 200-400 papers a day
- from one-page notes to industry level quarterly analysis of 150-200 pages
- Over 300K professional licensed users around the world

The publishing system was called Rendition, it was an in-house built black box for all intents and purposes though the code was available.

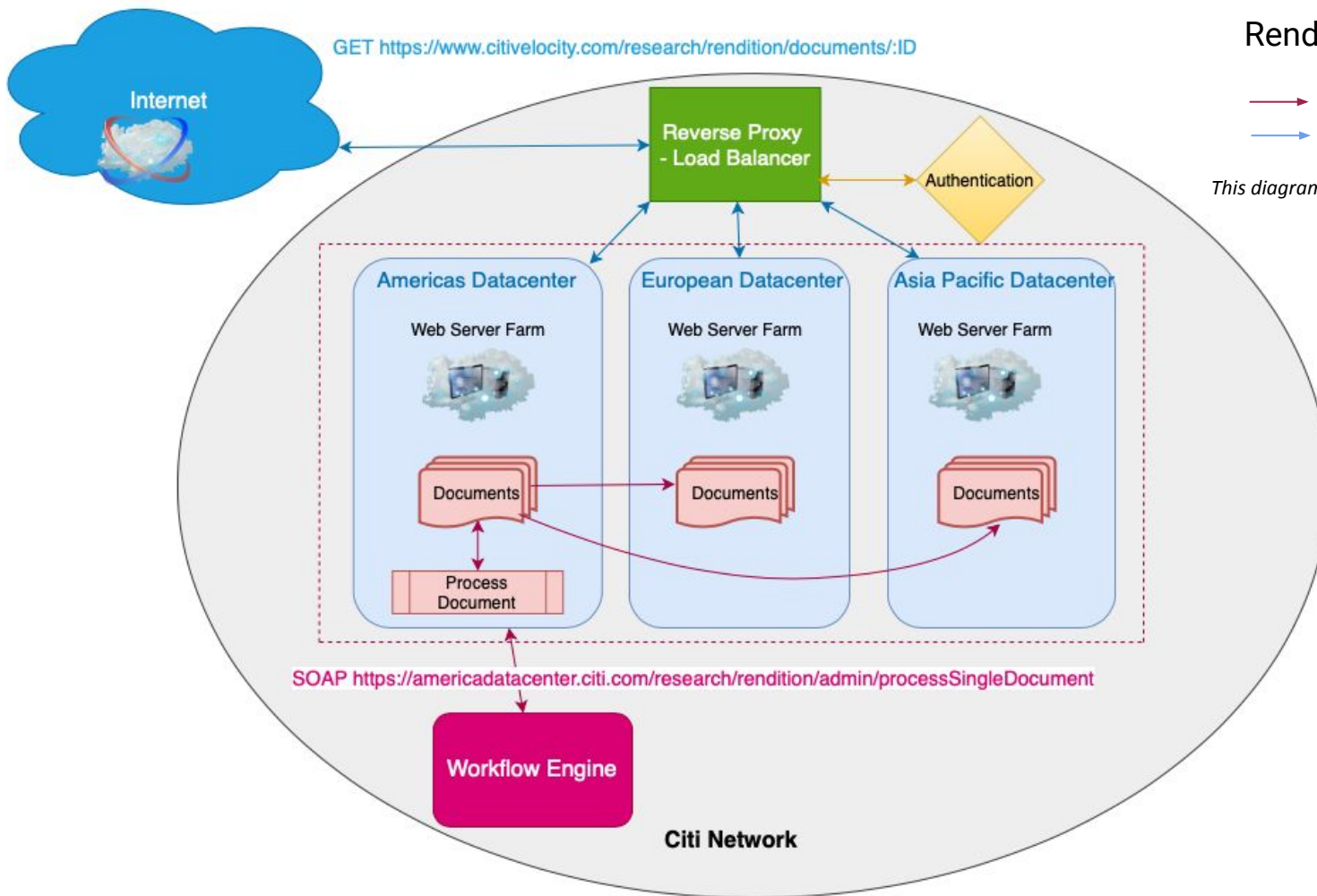
The rest of the presentation illustrates the progression of the architecture:

- from its initial state (base architecture) as a monolith system
 - through its split into back end and UI projects and then
 - to a full conversion of the back end into microservices architecture.
 - The last version stage was a POC AWS migration
- 

Rendition Functionality

- Publish a document
- View a document

This diagram shows a basic functionality of the system



The Original Architecture - a Monolith

Well, strictly speaking there was very little architecture

- It was just functionality thrown in mostly ad-hoc.
- The reason I can tell that is that there was no design document of any kind, there was not even a basic manual. That's one simple give-away clue.
- But beyond that, looking at the code, I could see that there was no design, no greater understanding beyond the code being written. As the requirements trickled in the code was written wherever it looked like it would fit best at that moment.

The Devil is in the detail

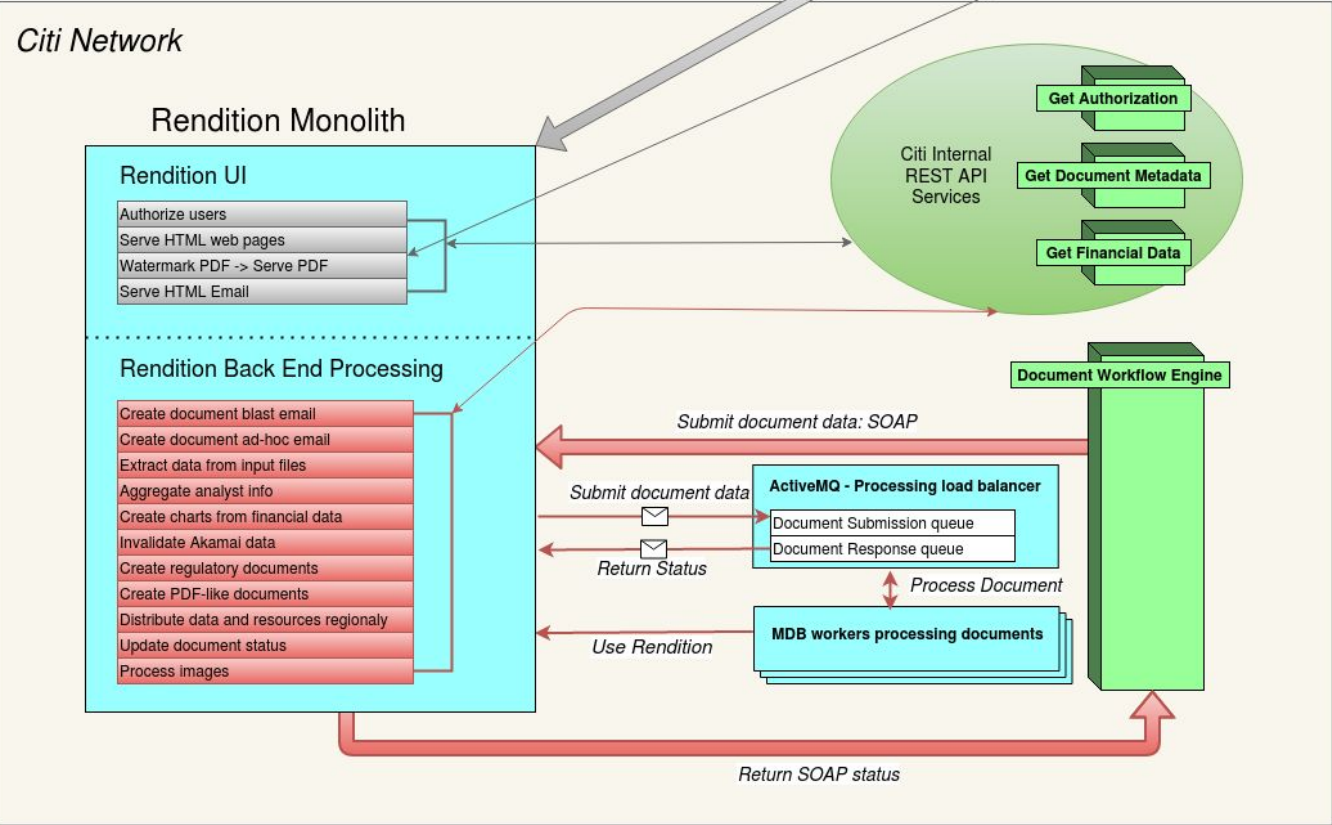
- The two top-level use cases seem simple enough (Publish and View)
- The functionality listed in below diagrams is just the very top level of the business requirements, these are supported by hundreds of various lower level services
- There was a top level folder in the code called 'services' that contained about 90 service classes which made use of hundreds of other smaller services below them
- It took me months looking into logs and code to put together the diagram below (no documentation, no SME).



Rendition System Architecture v1

Rendition System with Dependent Subsystems

External Services Used by Rendition



Breaking up the Monolith - Phase 1

Why the Breakup?

- For the very reason microservices architecture came about, namely, to make the monolith more manageable, e.g. make it easier to understand, to change, to add new functionality, to fix bugs and so on
- On the other hand, the speed to market was not relevant to us as Rendition had a quarterly release schedule planned before the start of each year and was dependent on a number of other system's release dates

Slow and Steady - Phased Approach


- In phase I (Rendition v2) we started on the journey to microservices. However instead of jumping to full microservices architecture immediately we took a phased approach.
- First we took out all the back end processing from the web server and split it into two main processes: document processing and document post-processing.
- Then we dropped the SOAP interface since there was no discernible purpose for it other than for the developers to be able to put on their resume that they used SOAP technology.
- We also needed to add major new functionality which was to create a pseudo-PDF version of the document that could be served to users with minimal entitlements. This PDF-like version was actually a web page that looked like a pdf document but lacked printing and saving (the document) functionality. Since we were starting the system migration to microservices, we decided to implement this functionality as a new and first real microservice running on Spring Boot to extract and serve the PDF-like web pages.

Summary of the Changes in Phase 1

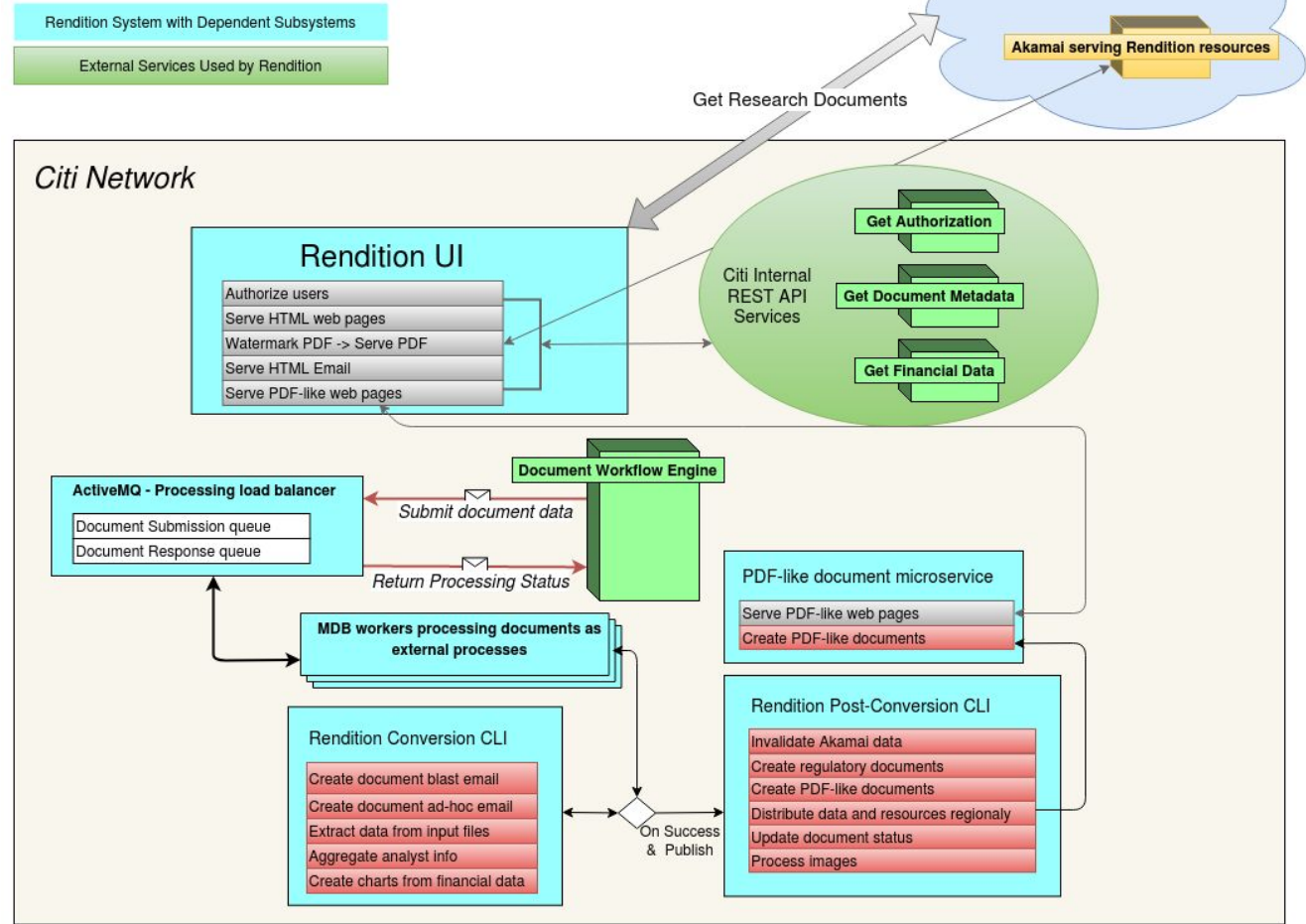
Summary of the major changes:

- Separate back end processing from the UI. Take back end processing out of the web server to improve memory management. -> SINGLE RESPONSIBILITY & MODULARITY PRINCIPLES
- Split the back end processing into 2 major CLI processes to be run as stateless processes by MDB workers -> IN ANTICIPATION OF FUTURE MICROSERVICE REDESIGN EXTRACT MAJOR PROCESSES FROM THE MONOLITH AND MAKE THEM STATELESS PROCESSES (12Factor App Principle)
- Add PDF-like document functionality into a separate microservice. PDF-like documents are html web pages that look like PDF documents but don't have printing and saving functionality - > ADD NEW FUNCTIONALITY TO A MONOLITH AS A MICROSERVICE ('Strangler' design pattern)
- Drop SOAP interface for engine workflow client and ask the client to use ActiveMQ queues as the interface to Rendition -> DROP UNNECESSARY DEPENDENCIES & SIMPLIFY INTERFACES

Results:

- Clear separation of concerns basic layers: a) UI, b) Back-end - document processing
 - Clear separation of two main business functionalities on the back end: a) Document Conversion (conversion of Word document to a web page) b) Document post-conversion tasks
 - Improved stability of Rendition, the document processing success rate went from 95% to almost 99.9%
 - Improved stability to ActiveMQ, used to crash occasionally with no errors after the conversion to CLI processes never crashed again
 - Simplified interface to the workflow engine
- 

Rendition System Architecture v2



Full-on Microservices - Phase 2

What's Next?

- While the Phase 1 version of Rendition was very successful in that it solved most if not all technical issues (primarily instability and processing errors) in the back end processing
- it did make the overall manageability of Rendition only partially easier. Both major processes were still difficult to understand and change.

Here we stopped and considered our options.

- The point of microservices is to split a business domain embodied by the system into its subdomains which we achieved with these 2 main services “conversion” and “post-conversion”.
- Any further division would not be strictly speaking business-driven but rather driven by technical services.
- These services were easily discernible and would be quite easily extracted.
- Additionally there was already a single self-contained service dangling on its own (PDF-like service)

And so we continued with further breakup of the 2 main business services based on technical service footprints

Finally we ended with 15 or so microservices including infrastructure services



Full-on Microservices - Phase 2

Summary of the major changes:

- Split 2 base service into microservices - FULL MICROSERVICES ARCHITECTURE
- Add Rendition Orchestrator microservice as the processing driver - SAGA ORCHESTRATOR PATTERN TO COORDINATE SERVICES
- Add Infrastructure microservices such as registration service, circuit-breaker, Kafka for aggregating UI logs etc (none shown)

Results

- **Advantages:** improved manageability, scalability, change management in short all the benefits of microservices.
- **Disadvantages:** very complex infrastructure which is more difficult to manage. In other words the business complexity was replaced by infrastructure complexity and that's always the tradeoff.
 - The reason this tradeoff is usually worth it is because a lot of the infrastructure functionality can be automated, sometimes all of it
 - Most of the development cannot be automated and by simplifying the domain one presumably increases the speed of development



