REPUBLIC OF ESTONIA
GCIO OFFICE

NEXT GENERATION DIGITAL GOVERNMENT ARCHITECTURE

version 0.9

Kristo Vaher

Government Chief Technology Officer

FEBRUARY 2020

Contents

Abstract

The following is a *vision paper*, a tentative document on proposals contained herein for debate, discussion and further research and development. This paper is intended for digital government leaders, IT development teams, managers, architects and engineers therein and other interested public servants as well as private sector partners and academia.

The core objective of this paper is to establish a common understanding of concepts and principles with the goal of these principles to become the foundational layer for next generation government technology architecture. While this paper focuses on the digital government stack of The Republic of Estonia, the issues within this paper likely apply to any modern digital government stack in part or whole.

While this paper focuses primarily on *software and solutions architecture*[1] layers, it also addresses data and business architecture dependencies.

The main topics of this paper focus on hypothesis of national scale implementation of domain driven design[2] and business process modelling[3], event driven microservice architecture[4], intelligent virtual assistant[5] and the concept of *#bürokratt* contained within *Estonia's national artificial intelligence strategy 2019-2021*[6].

Paper is structured in a way that first lays out an example of the vision, then the proposed technology big picture supporting the realization of that vision and then diving deep into four key areas that make up the proposed whole. Background information is provided for each layer.

Multiple existing concepts that this paper is relying on are only explained at a high level - more thorough materials are available for these topics for further research and recommended material is linked herein.

[1] https://dzone.com/articles/solution-architecture-vs-software-architecture and TOGAF https://www.opengroup.org/togaf and EIRA https://joinup.ec.europa.eu/solution/eira
[2] https://en.wikipedia.org/wiki/Domain-driven_design
[3] https://en.wikipedia.org/wiki/Business_process_modeling
[4] https://en.wikipedia.org/wiki/Microservices
[5] https://en.wikipedia.org/wiki/Virtual_assistant
[6] https://www.kratid.ee/in-english

This paper uses the following terms that benefit from explanation:

Administration sector is a semi-autonomous government organization, primarily encapsulated by responsibilities, structure and domain expertise of a single ministry - such as health or environment.

IT development team is a semi-autonomous IT and digital services and development providing team or organization that supports an administration sector, building any kind of digital services. In Estonia, less IT-dependent administration sectors share IT development teams - usually consolidated under a separate organization - with other areas.

Business stakeholder and **technical stakeholder** are abstract terms encapsulating multiple roles. Business stakeholder may mean *business/service/product owner/manager/leader*. Technical stakeholder may mean *technical architect/engineer/analyst/leader*.

The majority of concepts in this paper are not for a software engineer to implement alone. It is important that business stakeholders understand technical details at a high level and technical stakeholders understand the business processes and requirements in cooperation. Until we are struggling with language and common understanding of said concepts, we are unable to make the desired impact.

Multiple hypotheses in this paper require further research and study for feasibility of implementation on national and international scale. Academia is encouraged to pick up various listed topics. Proposed topics and research areas are listed at the end of each section.

As Estonia is one of the global pioneers for open solutions of government technology and international sharing of digital government lessons and ideas, this paper is first-hand published in English for accessibility reasons to enable international discussion, feedback, debate and cooperation with existing and new partners.

Author

At the time of publication of this paper, Kristo Vaher[7] is the Government Chief Technology Officer for the Republic of Estonia - working at Government CIO Office.

He has worked as a software engineer since 2000 in various fields including interactive media and computer games, advertising technology solutions, financial technology and banking solutions and government IT and architecture. He has worked as a lead engineer, team lead and enterprise architect as well as an expert consultant in the related fields. He has a computer science degree with a thesis related to a government military simulation project.

Contributors

Multiple contributors have helped in the creation of this paper and have given feedback. Most notable contributors are listed below.

Siim Sikkut is the core enabler and chief investor in the ideas of this paper and his feedback from early drafts is implemented thoroughly across this document.

Marten Kaevats is the originator of the *#bürokratt* idea and many of the concepts laid out in this paper are the result of technical/business brainstorming meetings and homework with him.

Petteri Kivimäki, **Uuno Vallner, Ilja Livenson, Kuldar Aas, Liivi Karpištšenko** and **Märt Aro** contributed with further ideas and polish, some of which are directly implemented into this paper.

[7] https://www.linkedin.com/in/kristovaher/

1. Problem statement

In 2017 Wired Magazine called Estonia *"the most advanced digital society in the world"*[8]. With the exponentially increasing adoption of technology in all areas - not just the government - and the emergence of new tools and IT-enabled and supported services and devices that barely existed or did not exist even 10 years ago, being the most advanced digital society today is not good enough anymore. It is important to be the most advanced digital society in the world - *tomorrow*. Without smart reformation in how government designs, builds and deploys their services, Estonia will fall behind.

Success of digital Estonia and digital government since regained independence in 1991 is three-fold: wide scale government investment particularly into education through Tiger's Leap[9] program, regulation wild-west after the fall of the Soviet Union - where new government was free to adopt previous laws and regulations or even start over in some areas - and last but not least, the advancements in internet based technology and personal computing across the world without any existing legacy to deal with. These three particular ingredients, paired with government strategy enabled for growth for digital government unlike seen anywhere else before.

Estonia set e-governance as a strategic goal in late 1990s and as a result, Estonia enabled tax declaration over the internet in 2000. X-Road[10] - a fundamental solution in registry based government technology data exchange - was launched in 2001. Nationwide digital identity, first based on a chip-technology enabled ID card[11] emerged in 2002. Estonians were able to vote over the internet in 2005 and by the year 2019 almost 50% votes are cast digitally in democratic elections. Nationwide digital identity was also enabled over mobile devices through SIM-cards as Mobile-ID[12] in 2007 and Estonia started using what later became known as blockchain technology to assure data integrity in government in 2008[13].

[8] https://www.wired.co.uk/article/estonia-e-resident
[9] https://en.wikipedia.org/wiki/Tiigrih%C3%BCpe
[10] https://en.wikipedia.org/wiki/X-Road
[11] https://en.wikipedia.org/wiki/Estonian_identity_card
[12] https://en.wikipedia.org/wiki/Mobile_identity_management#Estonia
[13] https://e-estonia.com/solutions/security-and-safety/ksi-blockchain/

While there have been multiple success stories since 2008 - most notably the innovative programme of e-residency - reality has emerged that the solutions built in the last decade are becoming an impediment in the road ahead.

The elements that made digital Estonia successful have changed and the environment today is different. While Estonian investments and opportunities for education are continuously strong[14] and evolution and adoption of technology is today faster than ever. This is also enhanced by the healthy growth of Estonian start-up sector.

But the benefits of regulation wild-west has vanished, partly internally, but also partly due to being a well-established and active member state of the European Union. Differently from the 1990s, there is no more *tabula rasa* - a blank slate - in designing, building and deploying government services. Business logic complexity of existing services is becoming overwhelming as more services are being designed and built and integrated between one another. Technologies implemented in the last twenty years are becoming outdated in part or whole not just in infrastructure, but also in software architecture.

Benefits of European Union are great for the citizens of Estonia and their quality of life as freedom of education, work, shopping and travel are inspiring, enabling this freedom is difficult. There are more laws, more regulations and more corner-cases than ever that need to be supported in automated digitized services. In recent years, impact of *General Data Protection Regulation*[15] and cyber security threats on international scale are also especially relevant that were not foreseen when first building blocks for Estonian government technology were put in place. It has been stated by leaders of Estonian public sector that *IT developments are unable to keep up with new laws and regulations* and it is becoming a critical problem[16].

Existing IT infrastructure and technology solutions are becoming outdated and difficult to manage to support the sustainable growth for the next decades in this environment and it is a disappointing reality that administration sectors are still prioritizing new solutions and new systems over maintenance and quality proofing of existing solutions, enhancing the problem further. Existing critical, highly-dependent and dependable information systems need immediate attention. It is important to get government technology stack to a state where only *bad legacy* is

[14] http://www.oecd.org/pisa/publications/pisa-2018-resultshtm.htm
[15] https://en.wikipedia.org/wiki/General_Data_Protection_Regulation
[16] https://tehnika.postimees.ee/6142719/politsei-palub-riigikogult-rahu

actually considered bad. Estonia does not have the manpower, nor the resources to rebuild its technology stack again every five years - and the same could be said about most countries, if not all.

The latter statement is also more thoroughly supported by the recent report of the National Audit Office of Estonia to *Riigikogu*, our parliament.[17] Its main message can be summarised as: *digital government does not just mean new services, but also sustainability and upkeep of existing systems.* Relying on such audits it is clear that the next generation digital Estonia needs to make a shift in how government procures, designs, builds and manages software and how Estonia implements core principles of e-government, such as once-only principle[18].

At the same time it is important for the government technology stack to be healthy in a way where bad legacy would not impede progress and does not set up barriers. Governments that have not yet digitized the majority of their services have the freedom to make decisions based on the state of technology as it is today - cloud, microservices and artificial intelligence - without being held back by legacy solutions of past decades that have to be continuously maintained. It is thus a challenge to still enable and encourage the growth of the ecosystem as if you are starting over from a blank slate.

Increased demands for data analysis and data-driven business decisions have also become a point of focus across all administration sectors in Estonia. Multiple areas are setting up dashboards for situation overview and automated reporting for management. A nation wide project REGREL is ongoing - a large scale project for automated registry based census of the population. Large scale and integrated use of data is raising challenges unlike anything encountered in the past decades, not just from technology standpoint, but also information and data management difficulties by business stakeholders.

Also, citizens of 2020 have different needs and expectations from the citizens of 2000. While the dot-com boom[19] and the resulting explosion of multiple information websites and online services - including government websites - as well as early wave of social media was a way of life for two decades, the next generation will have different expectations based on everyday use of

[17] https://www.itl.ee/uudised/riigikontrolli-aastaaruanne-eesti-e-riigi-arenguvoimalused-ja-riskid/
[18] https://en.wikipedia.org/wiki/Once-only_principle
[19] https://en.wikipedia.org/wiki/Dot-com_bubble

touch-based, voice enabled handheld mobile devices, the emergence of Internet of Things[20] and the continuous disappearance of personal stationary computers cannot be avoided.

The citizens of today also neither have the need nor any desire to be aware of complex administrational layers of the government - while at the same time said complexity and especially the use of private citizen data should still become more transparent, if need does arise. Citizens should be able to use seamless services irregardless of their everyday environment. While the government can be a complex web of processes and often unavoidable bureaucracy, citizen experience within should not be and this needs to become a number one focus to enable the best environment to live in - digital or otherwise. If a citizen thrives, so does the government.

To address those issues, to learn from the past and to avoid repeating some of the mistakes in the future, a new way of thinking is required in how government designs, builds, integrates and deploys services and uses data.

While technology is only an aspect of the big picture, it is a critical one and the focus of this paper. Government technology stack needs to support citizen experience of the next generation by being more flexible and loosely coupled to support the ever-changing landscape and business requirements. It needs to exist more naturally in the environment that citizens live in. And everything that we build today needs to last longer than everything that was built yesterday due to increased demand and digitization of services.

This is the first challenge of Estonian ICT for the current and next generation: how to extend the lifespan of the next generation systems in order to stay ahead of the curve and not get weighed down with expensive costs of maintenance and complexity of existing solutions to provide the best citizen experience possible.

Second important challenge is long-term sustainability and how to achieve environmentally sustainable software through a sustainable software development process. This topic has gained more attention in recent years and it is expected to be even more popular in the near future.

[20] https://en.wikipedia.org/wiki/Internet_of_things

1.1. The Story

How do you expect your citizen experience to be like?

Imagine that you - as a citizen of Estonia - are visiting Finland. You and your partner are expecting a baby soon, but it is a while before the due date, so you can still walk around the early autumn Helsinki, enjoying the carefree tourist lifestyle, taking selfies and walking around with cups of hot cocoa.

But something goes wrong. Baby is coming sooner than expected.

Being in a different country is complicated. You are not aware of where the hospitals are, you don't know what the phone number of the taxi company is or what ride sharing services are available. You are not even sure if your health insurance can support you, or what you have to do next as you'd be barely prepared at home, but abroad it is even more difficult.

So you pick up your phone and say: *"Help us, my partner is about to have a baby"*. Rotating processing wheel starts spinning on the phone screen, until a kind automated voice replies that everything is going to be alright and that it will get back to you soon.

Barely half a minute has passed as the kind voice continues. Your virtual assistant shows you where you are and directs you to a corner of the street barely fifty metres away. *"Everything is going to be alright! I have booked a transport for you: a car with the number ABC-123 is going to take you to a hospital one kilometre from here. Hospital has been notified that you are coming. Do not worry, you are about to be a parent soon!"*

Phone will pop up a notification, asking for a consent whether you agree to forward medical data from Estonia to Finnish government for this medical emergency, which your partner quickly accepts.

A car picks you up and drives you to the hospital. While your phone knows your payment details, you will instead get a notification from Estonian government, saying that your trip is subsidized so that you don't have to worry about anything other than your child.

Everything goes in the hospital as expected and soon after you have become a parent! Your phone congratulates you on becoming a parent and optionally recommends you multiple

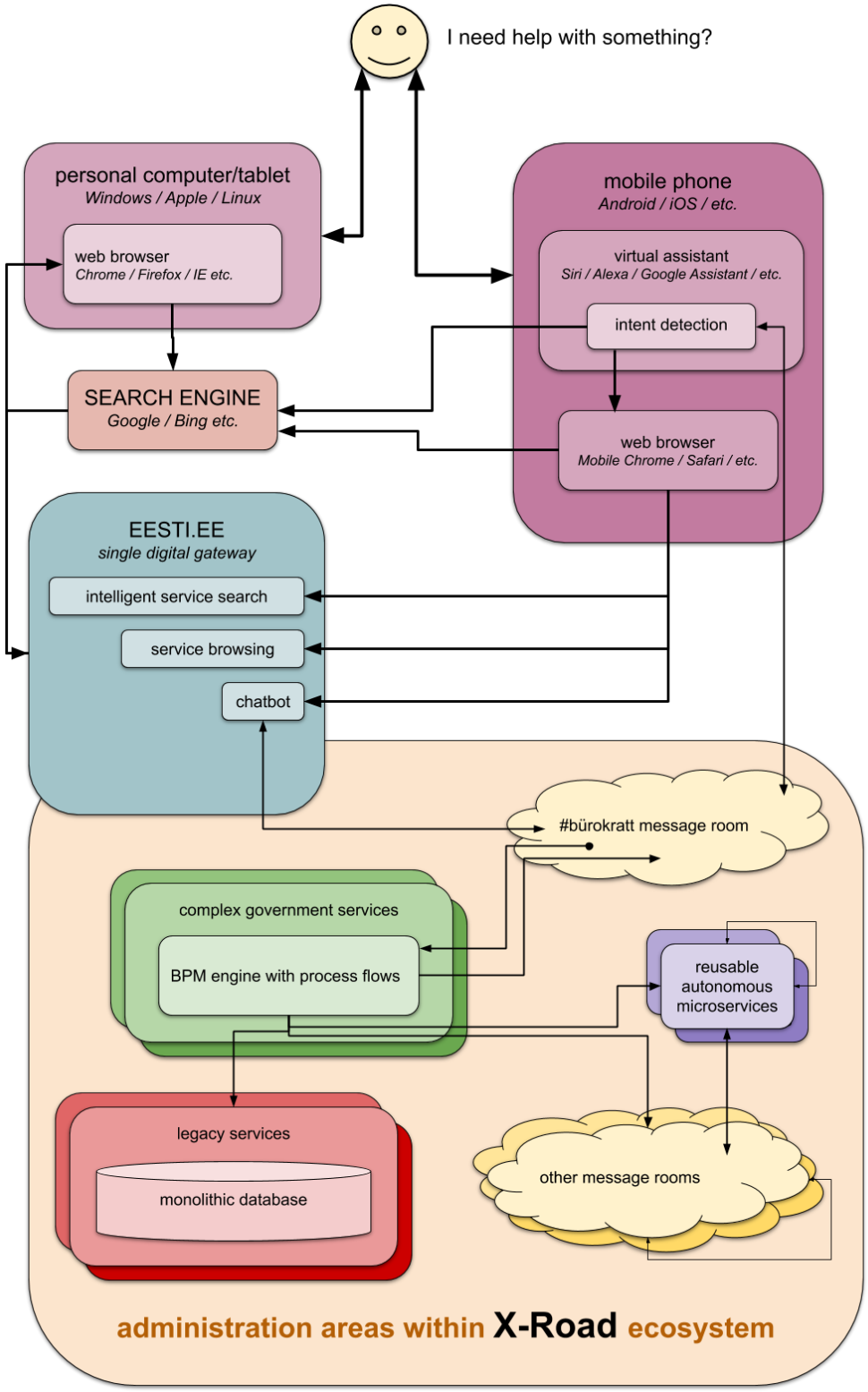
beautiful baby names, remarking that the recommended names will likely be unique among your child's classmates in the future.

“I am also preparing government support programs and services for you at home and will contact you if we need information from you. Let me know if you need any further help!” - says the phone as it goes silent, giving you and your partner time to get to know little Eha.

Remember this story.

The concepts in this paper are laid out to support the realization of this vision.

1.2. The Big Picture



Hypothesis of this vision paper is that the next generation digital government architecture could be achieved by focusing on three key areas:

- Government services need to become reusable proactive invisible services without requiring complexity awareness and multiple form filling within multiple administration sectors from the citizen. This paper proposes achieving this through **domain driven design** and **business process modeling** and related flow tools.
- Citizen communication layer with the government needs to transform from website-based services to seamless services in whatever environment the citizen finds themselves in. This paper proposes achieving this through using **virtual assistants** and related automated **message rooms** that can also be used for cross-border data sharing.
- Government needs to tackle existing monolithic legacy and build more re-usable technology stack for the next generation. This paper proposes multiple avenues for achieving this, primarily the concept of nation-wide scale **event driven microservice architecture** achieved through concept of distributed and **X-Road enabled message rooms**.

Last, but not least, it is also important to make a shift in how the government manages modern software projects in the public sector. This paper covers various aspects of this, from lessons learned about agile development and **becoming post-agile** as well as encouraging mindshift change for **vertical project delivery planning to horizontal** and the importance of impact in any system designed.

2. From silos to proactive services

In private sector services are generally paid for by the customer directly or indirectly *(by selling their data or showing them advertisements)*. In private sector user retention and active use of the service is critical for the success of the business.

Same is not generally true in the public sector. In Estonia, funding for projects is assigned through a complex government budgeting process where taxpayer finances are assigned to various administration sectors and projects. Same is true with European Union budget funding where the taxpayer does not have a direct say in funding for services.

Estonia does not have central core registries and databases. Digital government IT development and technical architecture is distributed between multiple areas of administration with said administration sectors having freedom in development and management of services.

Estonia has multiple ministries and agencies that are supported either by their own internal IT or partners from the private sector or - most commonly - by one of the larger IT development centres. Estonia has 6 larger IT development centres. Some of the IT development teams serve multiple administration sectors and many IT centres have their own data centres.

IT development centres are horizontally supported by cross-area IT and cybersecurity solutions from Estonian Information System Authority as the central IT agency providing key digital government platforms - from digital identity to shared tools.

This approach has given a lot of flexibility and freedom for all administration sectors to develop solutions based on their own needs and in fact can be considered one of the reasons for digital government success so far. This is because every administration sector has been able to solve their problems from their own perspective, taking into account only their own processes and requirements without having to know the details of other administration sectors other than what dependencies they have. But as a result, many of those services have become complicated, confusing and fragmented for the end user - citizens and residents.

While GCIO Office in Estonia is encouraging more and more involvement of citizens, analysis of user behavior and end-user involvement in the development of services, in majority of cases it happens either too late *(often once the service is already deployed live)* or based on feedback

not within statistical significance. Business decisions are still overwhelmingly dominated by business stakeholders opinions within administration sectors.

As a result, Estonian government has multiple administration sectors today with different records management systems and procedures and proceedings information systems, developed with a focus on the government official needs. The numbers are so plenty, that it often seems that everybody has their own system for every kind of proceeding imaginable. Due to unique flows and business processes and complexity of existing systems it is difficult, if not impossible, to transfer built solutions over to another administration sector.

This has enabled the possibility of administration sector silos, of sorts, across the digital government. One stack of an administration sector looks increasingly different not just in functionalities, but also in the technology stack supporting said functionalities. Thus the problem is two-fold: not only are business flows incompatible with another administration sector, so is the technology stack.

Low re-use is not for the lack of desire, but mostly due to complexity as different areas have implemented different tools from their own business perspective. This is also impacted by the passing of time, as even within a single administration sector services become old and outdated. Programming languages or database technologies or versions of each or either are used that have fallen out of popularity.

This has increased costs of maintenance and increased the desire of engineers from within the public sector as well as from the private sector to build a new solution from scratch. Ironically it has also been easier to get funding for new systems rather than upgrading and refactoring existing systems. This has also fractured the government technology infrastructure, as different software needs are built on different infrastructure solutions and they are not cross-compatible between one another: low number of services are in the cloud, the majority are virtualized and some are still running on dedicated hardware.

While IT development teams of different administration sectors have raised the desire to know more about what other administration sectors are using in terms of tools and services, in reality the ecosystem does not encourage nor enable this.

There’s an issue of transparency of government technology stack, tools, components and databases. There is no simple way for one administration sector to learn what another

administration sector is using and how, without going in depth into the administration sector. In some administration sectors the same problem is prevalent even within the same area. Estonia has *RIHA*[21], which is a high level registry of government information systems and data registries, but the information stored therein is outdated. Concept for the next version, that offers transparency and automated updates is in development, but has not yet been realized.

Yet even when there is a desire to re-use services from another administration sector to save time and costs, re-use is universally low between administration sectors due to the complexity of adoption of said services. Majority of e-services are monolithic, which means that they have been built as a single software system that is intended to function as a whole, delivering very specific functionality to that original administration sector.

Adding the complexity of user and permissions management and domain specific expansive functionality means that the system becomes very specific and very unique so that it is hard to take apart services and reuse data or components in an entirely different domain. In words from the movie *Fight Club*, every information system has become a *unique and beautiful snowflake*.

To tackle some of the aforementioned problems, Estonian government approved a plan for proactive services on 07.12.2018. While the details of this are more complex for the scope of this paper, the general principle is that *"public services are going to be made user-friendly proactive, seamless and automatic life event services"*.

The core idea behind proactive background services is that such services are the next evolutionary step following Estonian once-only principle[22] where it is important not to continuously ask the same data from the citizens over and over again by different administration sectors to solve a single problem or handle a single life event of the citizen, such as a birth.

Once-only principle is adopted in Estonia, but the reality is that continuous fragmentation between autonomous government silos means that despite once-only principle, the government still asks data from the citizen often in a repetitive manner. For example, during application for passport the Police and Border Guard Board relies on old data from previous applications instead of more accurate data from population registry.

[21] https://www.riha.ee/Avaleht
[22] https://en.wikipedia.org/wiki/Once-only_principle#Estonia

Part of the problem is a complex web of legalities and regulations also enhanced by the GDPR. It took Estonian Ministry of Economic Affairs and Communications years to work out and get government approval for proactive background services that do not require citizen consent in every step of the way and would allow different administration sectors to proactively share citizen data between themselves for the benefit of a more seamless user experience for the customer.

But while the government has given a go-ahead and some of the services have already been integrated between one another as proactive services, the reality is that there are no standards and no set principles on how to both design such services nor how to actually properly build them.

2.1. Conway's Law

In order to tackle the issue of silo-based fragmented services with complex monolithic processes, it is important to see what enables such services in the first place.

One of the best examples of digitization are hospitals. Health sector is carefully handling what is the most precious: *human lives* - and there are multiple lessons to learn from them. In 2019 multiple hospitals in Estonia had power outages and power outages can mean a tragedy if you are not prepared. Representative from one of Estonian hospitals has said that they expect the bare minimum for a larger hospital to be able to handle power outages up to three days[23]. Often with large scale power outages networking is also impacted that will mean the hospitals are unable to transfer data over the internet or request medical data related to the patient - even if they have their internal power generators up and running.

An interesting example of this is Rapla hospital that had a power outage due to cable malfunction in 2019[24]. Hospital said that despite outages they were able to continue their work using paper albeit in a limited manner and once computer systems were restored they were able to digitize the important material.

These examples, while sounding simple at first, will become very important examples for information system design.

A famous computer scientist Melvin Conway[25] defined in 1967 a *law* that says the following:

"organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."

What this means is that you can take any organization in the world and ask them to design a computer system that helps them work in the most ideal way possible. What they will end up designing will actually map closely that organizations manual routines and communication patterns. *As an example, if your everyday work involves taking a piece of paper to your colleague for signing, then the ideal computer system for you would automate this so that you can do it digitally.*

[23] https://sakala.postimees.ee/6813103/viljandi-haigla-loodab-voolukatkestuste-korral-generaatoritele
[24] https://www.ohtuleht.ee/973532/rapla-haigla-jai-osaliselt-elektrita
[25] https://en.wikipedia.org/wiki/Melvin_Conway

Implications of this can be difficult to see at first, but can be critical for not just the public sector, but any organization that has a need to develop services or systems to enhance and optimize their everyday work routines. This is especially true for an organization in the size of a government.

There are two key takeaways from Conway's Law: *organizations change and have to be able to change* and *bad processes are not improved by good technology.*

Organizations change

Bad legacy software and complex business processes do not happen by itself, neither do they happen just because enough time has passed. Software systems become a *bad legacy* most frequently simply because the organizations become unhappy using them - *because organizations change.*

According to Conway's Law, the ideal information system for an organization maps the organization's communication routines, but if those people change and new people have different expectations, then those new people and their routines and processes do not easily map into existing software.

And the more this happens, the more such a computer system becomes a *bad* legacy and the more upset the users become. This ends up with organization having to find a new solution, often developing a new system from scratch. This has happened frequently in government administration sectors as well.

What can be done to handle the risk of Conway's Law in case your organization and business processes are frequent to change:

- You want your services and information systems to be designed and built in a way that allows you to be almost as flexible as you can be when changing organization and its processes itself. This is addressed in this paper below.
- Do not design and build monolithic software, if you intend the system to be used for any extended length of time. Monolithic software is inflexible and increasingly difficult to iterate over multiple development cycles and changing business needs.

Bad process isn’t improved by good technology

You cannot cheat Conway’s Law or try to avoid it in any way. Conway’s Law implies that your maximum potential for an information system is the maximum potential of your organization. This means that if your organization or process itself is problematic, then so will be your information system and resulting automated service. This also means that you cannot jump over your own shadow and if your organization is inherently problematic, it might be better to use simpler tools than trying to fix it with a new complex system. In other words: bad input can end up with nothing better than bad output.

Keep the following in mind:

- Do not expect technology to be the silver bullet to fix everything. Start with the organization and the processes you are responsible for. Only once these processes work well, technology can help optimize and automate the routines.
- Do not forget information and data management. Just like it is important to have firm control over your services, data is critical, from both understanding of data as well as classifying it within its own domain or when mapped in comparison to other domains.
- Make sure you understand the roles and domains in your process and organization (see the next section).

2.2. Domain Driven Design

Good software starts from a good design and clear understanding of business processes. It is very common that when an organization realizes that it needs to develop a new information system and invites business stakeholders to such a brainstorming meeting, quite often the results are very *function* focused: program needs to do X and Y. It needs to integrate with Z. It needs to work on W. It needs to comply with A, B and C.

Computer scientist Eric Evans, author of Domain Driven Design concepts, has said that such an approach is misguided and often leads to problems - this is because such an approach attempts to cheat Conway's Law.

To illustrate what Domain Driven Design is, let's look at the following example:

Imagine that you have inherited 50 tons of experimental next generation electric car batteries from your late uncle. He left you a note, saying that this will make you rich. So you try the batteries out and indeed they are more effective than anything you have used before.

You decide to start up a business.

Owning 50 tons of batteries is not exactly an easy problem. Batteries need to be held in rooms fitting a specific condition, you need to have control over how many batteries there are and in what state they are in. Thus, you decide to hire an ***inventory manager*** *who is responsible for the warehouse and state of your batteries.*

But this is not enough. While you can now be sure that your inventory is good and nothing happens to it, no one still knows that you have those batteries. So you decide to hire a ***marketing guru*** *who ends up wearing battery costumes and running around petrol stations to advertise that your company has the best electric car batteries in the world.*

But this is still not enough. People now do know that you have those batteries and wish to buy them, but they cannot. To handle this problem you hire a ***sales manager*** *and make sure that the marketing guru can share their contact information. Suddenly sales start happening. With each sale, the sales manager asks the inventory manager if they have enough batteries left, because demand is incredibly high.*

*Despite sales happening, customers are still not getting their batteries because they are in your warehouse, handled by inventory manager. So you decide not to hire a transportation guy as transportation is an area that may not be something your company is best at - so instead you make a deal with **DHL** and use their services to deliver batteries to the customers.*

Suddenly the entire business flow works. Inventory manager handles the state of warehouse, marketing manager makes sure your voice is heard, sales manager handles sales and money and DHL delivers batteries to customers.

*There are some situations that still cause a headache though. Some customers are calling and saying that their addresses have changed since they placed the order and they are worried that their batteries are delivered to the wrong address. Thus you hire a **customer representative** that handles your customer data and makes sure that when customer address changes that sales representative is aware of it with all outstanding orders - address does not need to be changed in archived orders after all.*

All of those employees are working essentially in an open office environment.

This is a good example of Domain Driven Design that will be further explained below as well as Conway's Law. As a result the whole company could be automated almost entirely:

- Inventory manager - due to dangerous physical goods, inventory could be automated through inventory registry information system and warehouse robotics;
- Marketing guru - could be nothing other than a marketing website with good search engine optimization;
- Sales manager - is nothing more than a sales service with an API;
- DHL - while logistics is still needed, requests for transportation could be automated with logistics service integrated with DHL's own APIs and RPA[26] could be used, if logistics company has no API, but still has a website;
- Customer representative - could be no more than a chatbot with its own customer database registry;

[26] https://en.wikipedia.org/wiki/Robotic_process_automation

- And open office environment could simply be event-driven architecture communication rooms where the different services can share data.

This example is an important one. It doesn't say that such a company would not need any actual manual labor, but it does mean that the most common routines could be automated so that the rest of the company could deal with external problems and exceptions that might arise.

And what is difficult with humans, is not so difficult with computer systems. If such a company is automated, it is possible to introduce a new service into that same communication room and see if it is able to do its job better than the existing service. It would be possible to integrate UPS service alongside DHL service and then, through comparing how the services work, either have them share load or prefer one over the other.

And once the batteries run out, you can close down the inventory manager service and warehouse robotics services. Your website can become an electric battery information portal that starts taking consultation requests - which can be a modified sales service API with a whole new set of products.

This kind of flexibility is only possible if it is started from Domain Driven Design way of planning. Doing it the other way around would have likely ended up with a large monolithic e-shop with customer management, inventory, payment and logistics functions, increasing complexity and removing flexibility you need of your business changes, personnel changes or laws and regulations change.

Template

Domain Driven Design is a term originally authored by Evans in his 2004 book[27] of the same name and what it effectively means is that system design should be driven by domains of the process and organization. Eric Evans says that while in an ideal world the whole of that world would be mapped to a single unified model, reality is different and similarly to Conway's Law, this reality cannot be avoided.

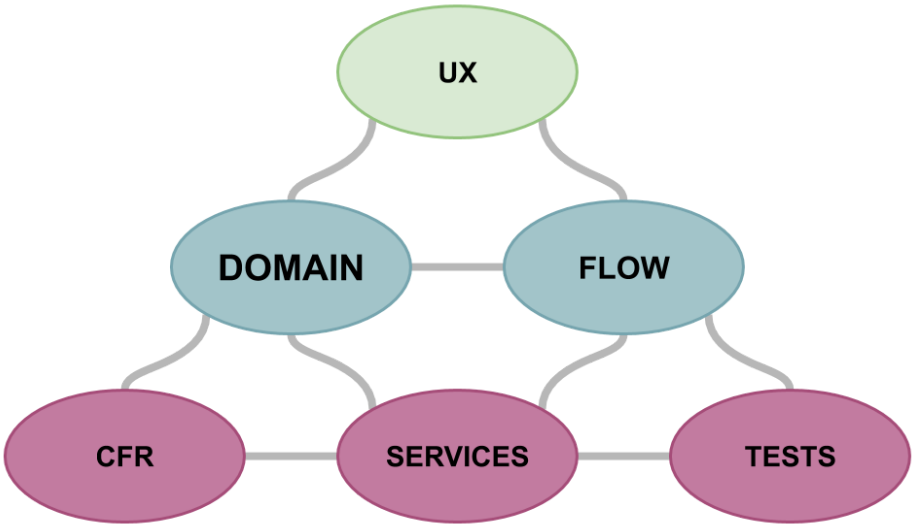
[27] http://dddcommunity.org/book/evans_2003/

In order to get to a more tangible understanding of domain driven design as a concept, it is important to explain what a *domain* actually is: a *domain* is a constrained sphere of *knowledge*, *influence*, *function* or *activity*.

To give an example: every employee in an organization is responsible for at least one domain. Many employees share the same domain and many employees are working in multiple domains. *For example, a domain may be "inventory management", which means that it's a body of work and its functions related to inventory management in organization.*

While the topic of Domain Driven Design is one about which whole books have been written, in order to make a change for business stakeholders in the public sector I will lay out basic groundwork and suggestions from my own experience.

The following is a basic template that you should follow from the moment you have realized that *yes*, there is a service or system you want to create and there is an actual business case for such a service:



Using this template, process of laying out scope for the project from domain-driven perspective is as follows:

1. **DOMAIN** - Write down the list of domains that are supposed to use the information system or service. Every single domain is important! What are the business roles that deliver value in the organization? *Remember that a domain is a constrained sphere of*

knowledge, influence, function or activity - such as a specific role in a company with the specific set of duties.

2. Once you have your domains in place, map out what are the required **FLOW**'s of those domains. A flow is a specific activity that should be possible in the service or system with a clear beginning, middle and an end. Remember that there should be no flows that have no domains and a single flow may span multiple domains.

3. Once you know what domains and flows are in your planned system, you can start involving your technical stakeholders and engineers. Your technical stakeholders are responsible for the three bottom pillars of the above graph and your user interface and user experience designers are responsible for the very top.

4. **UX** means user experience, but in this context it can mean anything related to user interface and graphic design. It can even mean a technical designer in case the planned system is only ever meant as an API[28] (Application Programming Interface). Note that UX does not actually require the bottom technical layer to exist. The only thing UX requires is the awareness of what domains there are and what flows need to be supported for those flows. User experience designers will benefit greatly from knowing that they are not asked for 'another website'. Their focus can go to design an experience for a certain kind of employee that needs to do certain kinds of things in the organization.

5. The bottom three spheres on the graph belong to technical stakeholders of the project. Similarly to other spheres, make a note of the connections between spheres. The **SERVICES** means all technical components and endpoints. Services are your actual organisation routine automation back-end components. Ideally - if your goal is a flexible system - you will intend to have multiple services, usually one service per unique domain.

6. The **TESTS** sphere is not interested in domains themselves, tests are meant to assure that functionality works either during development or routinely in the background. Ideally you need to plan out tests for every flow in your system and those tests are making sure that services are delivering those flows as expected. While this is subjective personal

[28] https://en.wikipedia.org/wiki/Application_programming_interface

opinion, the only critical tests are the ones that actually test the business functionality of a system.

7. The **CFR** sphere means Cross-Functional Requirements. The term “non-functional requirements” is more popular, but many software architects in recent years have agreed that you have no purpose for requirements that have absolutely no connection to business domains. While the intended meaning is similar, it is more accurate to name requirements cross-functional - meaning principles, standards and requirements that are inherited from laws and regulations as well as good engineering practices that are adopted to the system for business need.

Following these points it is possible to set out a scope for a system with business stakeholders with domains and flows and then designers on UX and technical stakeholders on the bottom layer and start development.

There are other key benefits as well:

- It supports a more agile way of software development: the listed flows are very much like *user stories* of Agile methodology. Implementing Domain Driven Design can also help organizations get more familiar with more agile software development practices.
- It helps to find clarity in organization business and information architecture and to reflect back on inconsistencies in organization as a whole. If the above exercise ends up like a complex mess, it is likely because organization processes are a complex mess that should be addressed before anyone starts writing code.
- Domain Driven Design is also helpful in getting organizations plan projects in a more horizontal manner as delivery roadmap of a project can be domain-by-domain and flow-by-flow.
- Domain Driven Design brings to the forefront the importance of role automation in whatever process and removes boundaries between business and engineering as both need to understand the process at a very similar level.
- Domain Driven Design is a foundational enabler of maintainable and decentralized autonomous service architecture that will be covered further in later sections.

- In the public sector, the public servant and the citizen are two different domains. Third possible domain is entrepreneurs and businesses. As a result, they should possibly be handled differently within an information system as well, including separation of back-end services and user interfaces and tests.

What is good about the previous visual template is that you can design your whole business functional information system in this manner without writing a single line of code. The whole design can be played out between people and using pieces of paper.

Returning to the previous example of Rapla hospital, they were able to continue working on paper while the power was out. This is because their processes were not designed with computers in mind as a primary focus - computers simply allowed to automate and optimize processes that already had to exist in the organization anyway. Thus the hospital was both subject to Conway's Law, as well as domain driven design internally. This is how it was then possible to transfer manual processes back to digital once power was restored.

When implementing Domain Driven Design, few other key criteria has to be kept in mind:

- If it seems that your domain responsibilities are incredibly complex and full of dependencies and edge cases and that you don't think you can map your domains and flows out in just as easy a way as the practical example given in the beginning, you are likely not looking at the problem close enough.
- If the above still does not help, a popular methodology - strangler pattern[29] - which is used in software development to break apart complex monoliths, could be applied to complex business processes as well.

[29] https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler

2.3. Business Process Modeling

In software architecture there are two abstract extremes how information systems are designed:

- traditional orchestrated one - *a system similar to a conductor waving hands before orchestra to do their bidding*;
- choreographed one - *a system similar to a dance routine where all dancers react to one another according to predefined core principles*.

Looking at challenges faced by the public sector and distributed digital government as a whole, neither extreme is a healthy choice for long term. There are too many predefined regulations and laws on how certain processes need to work, making choreography as a de facto standard nearly impossible. This is supported by various use cases, even companies that have pioneered choreographed event-driven architecture, like Netflix, have learned their lesson and realized that you still need orchestration in places[30].

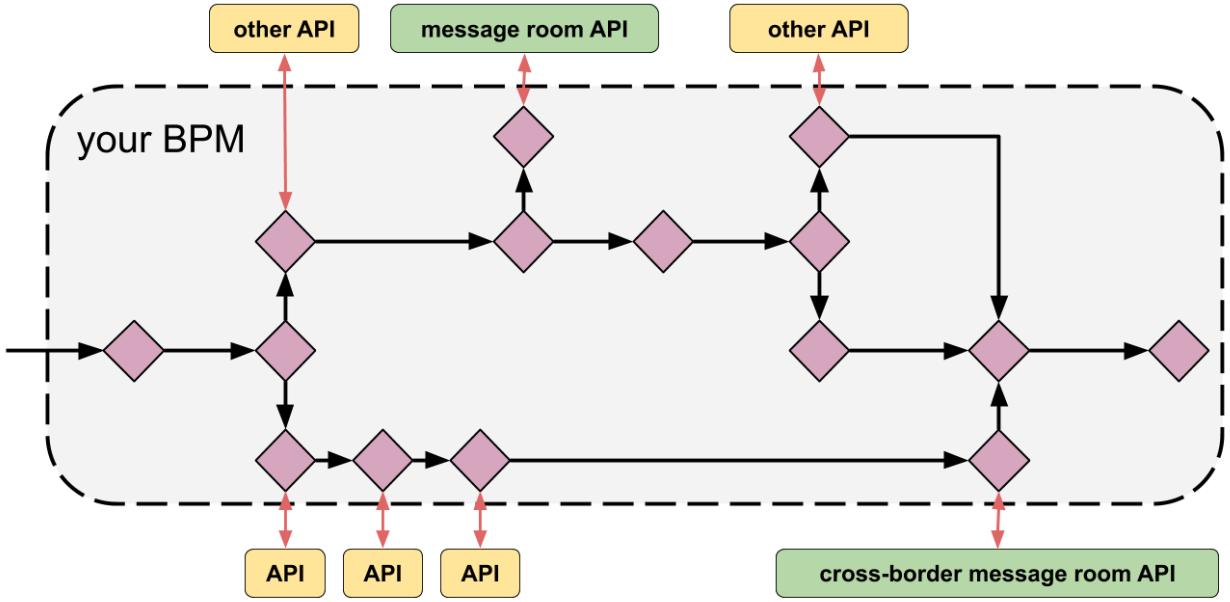
But if choreography is too dynamic for laws and regulations and orchestration is too complex in integrations, something in between could be an option. If digital government uses orchestration at a high level - for large scale business processes - and uses choreography at a low level - for functional services and functional tasks - then perhaps digital government decoupling and foundations for cross-area background services would be possible.

One of the ways how orchestration can be achieved - without sacrificing re-use and flexibility - is the use of Business Process Modeling and related workflow tools. The core idea of this is that business processes are handled in a separate piece of software from all the other functionalities. *For example, sending out an email is not an integrated part of software and instead is behind a separate API.*

As described earlier, the complexity of public sector monolithic information systems comes often as a result of complexity of business processes. Engineering difficulty is often with specific functionalities and not as much in business flows themselves. It is important to decouple the two, as it would be possible to re-use domains and functions without having to re-use business functionality themselves.

[30] https://medium.com/netflix-techblog/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40

Abstract view of BPM's role is the following:



Business Process Modeling workflow software can be engineered from scratch - *and often is engineered from scratch as part of if/else condition labyrinth in a monolithic software* - but it is recommended not to. Most popular tools used for this task are Camunda[31] and Flowable[32].

This approach allows system to separate business flows from functional APIs. While workflow tools such as Camunda and Flowable still require engineers to manage and maintain the complex aspects of workflows, then these tools are also a visual aid for business stakeholders. This means that business stakeholders would have a visual overview of their business process exactly as it is rather than an interpretation of code. It allows us to show both complexity as well as bottlenecks.

It also opens up an opportunity to share and re-use APIs since these functionalities - *domains and flows in Domain Driven Design* - are not coupled with business process flows directly. This can lead to more distributed government technology architecture, especially when paired with event driven microservice concepts described later in the paper.

[31] https://en.wikipedia.org/wiki/Camunda
[32] https://en.wikipedia.org/wiki/Flowable

BPM essentially acts as the domain of specific business processes, such as the role of a manager that has to make sure the team - tools and technical services - have their tasks at the right time and can get to the result that organization needs in the end. At the same time, while BPM workflows can in itself be coupled, the functions used can be reused by across the organization by other flows.

BPM's are also near ideal tools for orchestrating proactive background services as you can model the entire business flows across administration sectors with such BPM's. Workflow engines of different administration sectors could also communicate with each other, setting off new processes and flows as a result.

Important takeaways regarding Business Process Modeling and workflow engines:

- Administration sectors with complex business processes likely need multiple workflow engines for those complex business processes. It is not recommended to create a new single point of failure by using a single workflow engine for everything that is automated within an administration sector.
- Start experimenting with BPM and workflow engines with a smaller project first in order to get familiar with it.
- Workflow engines can create real time views for business stakeholder dashboards as well and it perfectly maps with Lean development principles. It is possible to see both bottlenecks and manual load in your BPM charts.
- Implementation of Business Process Modeling tools and Domain Driven Design is absolute key in getting modern software architecture principles more widely adopted in digital government.
- Business Process Modeling allows to have an up to date and exact overview of business processes. Most business stakeholders in the public sector have to rely on analysts or engineers interpretations of the business flow, but BPM and workflow tools give an exact state of business processes as-is.
- Most monolithic software in digital government are the result of business process complexity being tightly coupled within functional codebase. BPM helps to keep the two

logically separate and encourage re-use. Other administration sectors do not wish to use your complex flows they don't understand, but they'd gladly use your most valuable automated tools for picture analysis, message sending or tax calculation.

Data management

Designing business processes in a seamless manner is important, but it is perhaps even more important to have a firm grasp on the data of the organization, especially if that data is shared between services.

Many information systems and registries are built with just a supporting back-end relational database for the services, but the increased growth of data means that unless making it a business priority, data can become complicated and even impossible to maintain long term.

Data management is a huge topic in its own right, but the following is a list of what should be kept in mind:

- Make sure your organization has clear roles in place whose responsibility is information and data management.
- Use data governance tools that help keep track of organizations data quality.
- You need to handle the growth of data in a proactive manner. Needs of your system are likely to grow in an exponential manner in the era of artificial intelligence and smart management of this growth is crucial to sustainable data architecture.
- Make sure you have classification of data in place either internally within organization or also for external dependencies. *But at the same time do not create central classification dependencies and related services - which would break autonomy of services.*
- Data quality is not just the quality of the database structure and data stored within, but also the API's and services themselves that expose data.

2.4. Key takeaways

Transforming silo-based government architecture to cross-administration-area architecture with focus on citizen experience can be difficult, especially if the starting point is as complex as it is in governments like Estonia today. It is important to make sure regulation aligns with this concept. In the example of Estonia this has taken years to enable support of seamless background services.

Even without regulation in place, it is smart to involve multiple participants across different administration sectors and start working on first integrated background services. Focus should be on services where citizens experience today is divided into multiple contact points or filling of forms, especially when those forms are provided by two different government websites or other contact points - even ones that are non-digital.

Domain Driven Design should be used to map out how the process should actually work. What are the domains and roles in this business process and what are the flows that need to work end-to-end for this service.

Business Process Modeling tools should be tried out to start building this cross-area service, Camunda or Flowable being the main ones to recommend. While seemingly complex at first and still requiring an engineer to make the best use of it, BPM tools have a high potential for the right kind of orchestration within government.

Data management needs to be in place so that organizations have a clear overview of what data they store and how it relates to their services.

What’s most important is to pilot and test and get used to new ways of building services sooner, rather than later. And even if your administration sector has no cross-area integrations and dependencies, similar concepts are invaluable even within a single organization.

3. From websites to intelligent virtual assistant #bürokratt

European Union expects every member state to have a single digital gateway[33] by the end of 2023.

"The single digital gateway will guide citizens and companies to information on national and EU rules, rights and procedures and the websites where they can carry out these procedures online. And users looking for assistance will be guided towards problem-solving services."

Estonian domestic equivalent to single digital gateway has been *eesti.ee*, a *"government web portal*" and single contact point website for citizens and businesses alike. This web portal was launched in 2003[34] and over time has evolved to become a key part of Estonian citizen's digital experience.

Eesti.ee has become more than just an information portal and also has enabled administration sector services to be integrated into the web portal, such as checking your address data from population registry or seeing your diplomas. For the citizen it provides an overview of how and when their data has been accessed as well as direct or linked access to other government services.

But with the increased complexity in developing e-services in administration sectors and changing citizen expectations, what is essentially a complex website is becoming outdated:

- Administration sectors have had notable complexity in developing their service endpoints inside government portal software stack. Such integrations have to become loosely coupled[35]. Difficulty in this area has resulted in some administration sectors to set up their own citizen web portals, complicating citizen experience.
- Citizens are using web search engines to query about everyday problems, such as what to do when identity document is lost and expect to find a single source of truth through search engines. This can often lead the citizen to less accurate results.

[33] https://ec.europa.eu/growth/single-market/single-digital-gateway_en
[34] https://www.eesti.ee/est/teemad/kodanik/riigiportaali_abi/riigiportaali_ajalugu
[35] https://en.wikipedia.org/wiki/Loose_coupling

- The citizen does not visit and has no interest in visiting government web portal anywhere near as frequently as they visit Instagram or Facebook. This means that the rate of notable changes in government web portal can negatively impact citizen experience since it may seem to the citizen that the website is changing every time they actually visit.

- Fragmented user experience is also a threat with multiple administration sectors developing their own citizen communication portals on the side of eesti.ee due to aforementioned complexities in developing services within eesti.ee. This can lead to issues similar to the United States ESTA Visa program where - unless you actually know the website you are looking for - the search engine can lead you to middle-men websites that can ask premium expenses for free or notably cheaper government services.

Digital government single digital gateway *eesti.ee* is not going to be eliminated or replaced, but it is important to look into how it can evolve and support new services. Government web portal also needs to exist as a backup in case other expected solutions are not delivering. In late 2018 a new and updated beta version of eesti.ee was launched, but further changes are required.

The hypothesis laid out by this paper is that artificial intelligence enabled virtual assistants should become the main way the government provides services. As such, *Estonia's national artificial strategy* was published on 28.05.2019. Among multiple areas that Estonia is going to focus on, one of the focuses is the following:

"Developing the #BürokrattAI concept for interoperability of public sector AI solutions as well as shared AI interface for citizens for use of public services"

Kratt is a mythological being[36] in Estonia, an artificial man-like creature meant for manual labor. Due to its inherent similarity, Estonia adopted the use of the word *kratt* as an equivalent of artificial intelligence and by the year 2020, this word is being used to describe even other forms of automation in the public sector that may technically not even be an AI. All administration sectors now speak of said *kratt's* and multiple administration sectors have involved actual machine learning solutions in various projects[37] as a result.

[36] https://en.wikipedia.org/wiki/Kratt
[37] https://www.kratid.ee/kasutuslood

Name of the concept for virtual assistant #bürokratt comes from the words "*bürokraatia*" *(bureaucracy)* and the aforementioned kratt. Within the AI strategy there are two points of focus for #bürokratt:

- Creation of digital government assistant for a more seamless citizen experience;
- Trialing and assuring the interoperability of multiple artificial intelligence solutions.

Relying on the Story laid out in the beginning, with the successful implementation of #bürokratt all of the following would be possible:

- You can use your phone or tablet or intelligent TV set or other home digital assistant to get access to government services.
- You do not have to learn the complexities of government bureaucracy, communication with the government would be seamless and natural.
- Government can notify you of important changes to your status, benefits or otherwise.
- You do not have to know which administration sectors to contact and which websites to visit.
- All the other benefits would still remain or would be enhanced further, such as that you would have better transparency over your data use and consent related to data.

This paper lays out a concept of how such a virtual assistant could be realized and what are the key ingredients to make it happen on top of the government technology stack.

3.1. Next generation seamless citizen experience

Estonia is providing online digital services to customers through websites maintained by administration sectors as well as eesti.ee as the single digital gateway. A lot of services are integrated into gateway, but not all of them, and many services are linked from single digital gateway to website that provides the actual service. Core services are available through eesti.ee directly, such as to change your living address, or see what kind of data has been requested about you from the government, and more.

While digitization of government services is a great success story in Estonia, the majority of digitization and IT developments in Estonia are focused on the heavy end of governments own services and needs of the public servant rather than the citizen, as mentioned earlier. This is the complexity that citizens should never have to encounter, which means that a different interface will become important to offer those services for the citizen.

Classic user interfaces that we are used to today are becoming outdated. Websites and even phone digital user interfaces are inconvenient unless you use them daily and the public sector will encounter issues and dissatisfaction related to this issue much earlier than the private sector will. Citizens do not wish to visit government websites frequently, if at all, thus any kind of unfamiliar interface for them is inconvenient. This also means that it is more difficult to make changes to public web interfaces, since even small changes can negatively impact a visitor that rarely visits - as it may look different every time.

This is an opportunity for the public sector to tackle this problem as pioneers. Three areas have to be investigated to make this happen:

- Creation of a virtual assistant[38] that streamlines communication between the citizen and the government;
- Domestic language support;
- Enabling the concept of digital twin[39].

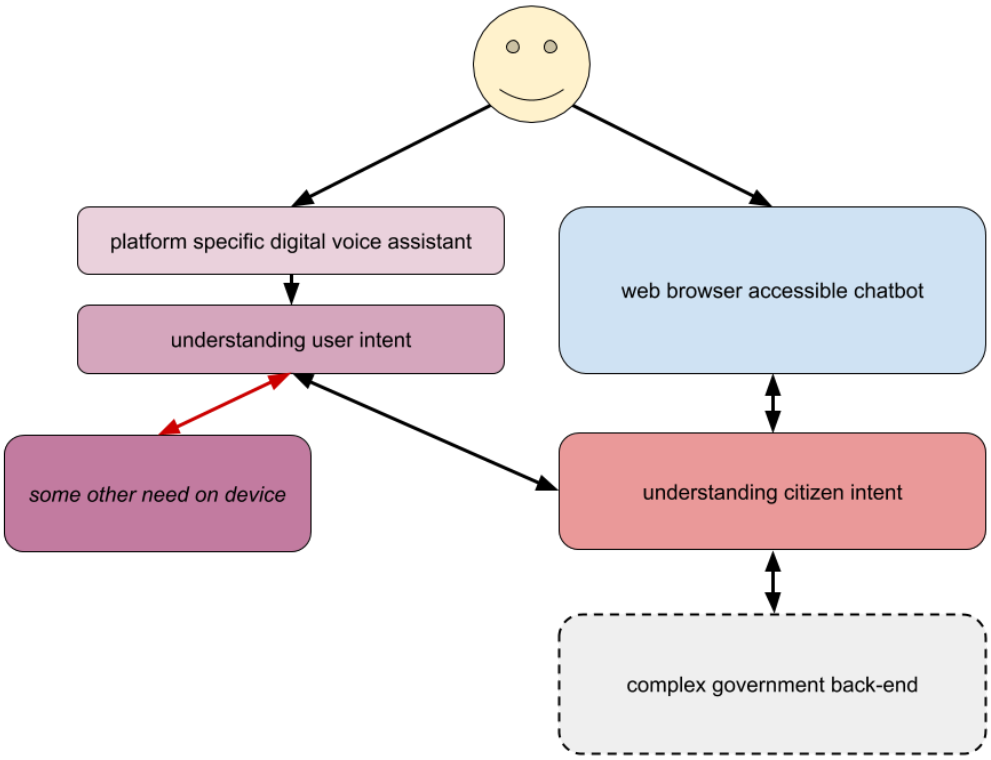
[38] https://en.wikipedia.org/wiki/Virtual_assistant
[39] https://en.wikipedia.org/wiki/Digital_twin

Digital government virtual assistant

Governments should not, if at all possible, build phone apps for citizens that everybody has to install in order to use government services. This does not mean that all apps should be avoided, but an app causes an increased digital divide and complexity for its users which should be carefully considered. It is important that the citizen can use most important government services in whatever environment they are in - from physical to digital.

When it comes to virtual assistants this means that the virtual assistant needs to be accessible over a variety of devices from any provider without requiring citizens to install anything extra, other than what is provided by the device itself. It is highly likely that with a well designed ecosystem such virtual assistants could be provided entirely by the private sector.

Due to requirements to be as much vendor and device-agnostic as possible, this means that a virtual assistant needs to work on two separate layers: *web browsers* and *mobile devices with internal digital assistants*:



Here is an example flow of how this might work:

1. Citizen loses a passport;
2. Citizen either uses their phone or logs onto their computer and visits government single digital gateway;
    a. Citizen tells their phone *"I have lost my passport";*
        i. Phone *(internally, if possible)* processes user intent and detects that this is government related;
        ii. Phone sends this message to government endpoint for processing;
        iii. Government endpoint responds that message is received and notifies the phone internally with a transaction/session ID related to this event;
    b. Citizen writes to government single digital gateway website chat bot window *"I have lost my passport"*
        i. Chatbot sends this message to government endpoint for processing
        ii. Government endpoint tells chatbot that message is received.
3. In either case, citizen is told that the government has received the message and is working on it.
4. *A complex web (tackled in other sections of this paper) of back-end communication happens on the government side, which may involve multiple administration sectors.*
5. A government realizes that it needs to know the identity of the user, so depending on the environment citizen is at, the following happens:
    a. Phone asks the customer to authenticate themselves either through web endpoint and browser or using Mobile ID, Smart ID or other similar government-accepted solutions. This request carries the same transaction/session ID thus responses are directed to the same government back-end (phone should not have to detect further intents).
        i. Customer identifies themselves.
        ii. Process continues in the background.
    b. Single digital gateway website asks citizen to authenticate themselves, for example using their identity card or Mobile ID.
        i. Customer identifies themselves.
        ii. Process continues in the background.

6. In either case,the citizen is notified that the loss of their document is reported and related certificates have been flagged. The citizen is told that the government will contact them, if anything else is required.

Domestic language support

There is one important cornerstone to making virtual assistants happen in such a scope: virtual assistant needs to understand the language of the citizen. In Estonia this means definitely understanding Estonian, but possibly also Russian and English and requests in these languages should be understood correctly by digital government services. By making sure the understanding of language is separate from services themselves it would be possible to offer government services for any resident regardless of their native language.

For countries where Google or Apple have already integrated domestic language support and digital assistants already speak the local language, getting to such virtual assistants is not as complicated. In Estonia it is critical to get those everyday devices that citizens use to actually speak their language.

Multiple things have to happen to try out the virtual assistant #bürokratt concept:

- As mentioned, without language support #bürokratt can only be achieved as a chatbot implemented to single digital gateway. While it is possible to test multiple concepts of #bürokratt as a result, for wider adoption within citizens' living environment chatbots are not good enough.
- Mobile devices need to be able to understand if a request is government related or not internally and then direct the request to the government communication room/service for processing. Without such internal intent processing and redirection to the government it is almost impossible to implement the concept.
- Hypothetical worst case scenario is that the virtual assistant has to be implemented as a phone app that can handle government related requests.
- Success of #bürokratt does not rely only upon having an native-language-understanding virtual assistant on the phone. For #bürokratt to happen it is important for government technology architecture to enable AI-driven communication and access to data. This is

covered in further topics in this paper related to event driven microservice architecture and message rooms.

My data and digital twin

While virtual assistants are a complicated challenge in their own right, in order to make virtual assistants effective it is also important to handle the issue of citizen data and identity. This is especially important in light of General Data Protection Regulation[40] in Europe that gives more defined control over data to the citizen.

#bürokratt acts as a virtual assistant, but in many ways it can also act as a digital twin[41] and a vault for MyData[42], having more direct control over what citizen preferences are in terms of government communication, data exchange, getting notifications when private data is being accessed and more.

And while this is a challenge for technology, it is also a legal challenge. Multiple issues need to be investigated:

- Government needs to assure that the citizen is unable to make an unintended mistake, such as deletion of diplomas. If data is stored in a vault locked by private key, management of this key and security of it becomes critical. This could be encrypted by digital identity.
- Cybersecurity and also quantum computing is a risk, especially if control over data is closer to the citizen and thus further from government controlled databases. It is important to make sure that the tools used by the citizen do not compromise citizens status and freedoms is any way.
- How does the concept of consent services apply to digital twins and MyData? Estonia is in the process of implementing a government-wide consent service tool paired with its digital identity today, but this primarily focuses on the government side of the problem.
- If at all possible, such solutions should not have to be developed by the public sector itself. Virtual assistants and digital twin solutions could ideally be provided by the private

[40] https://en.wikipedia.org/wiki/General_Data_Protection_Regulation
[41] https://en.wikipedia.org/wiki/Digital_twin
[42] https://mydata.org/

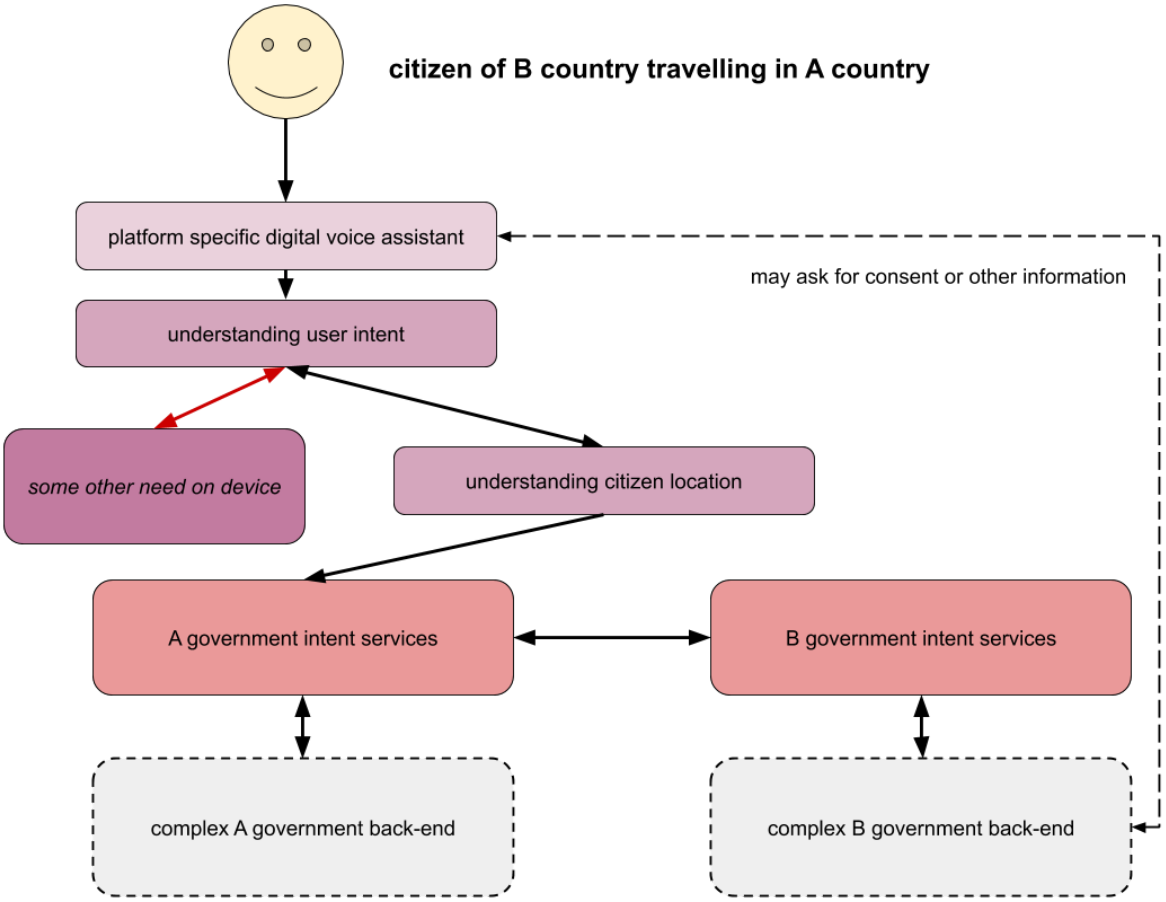
sector and the role of the government is to simply assure that communication back-end is available.

3.2. Cross-border citizen experience

Many private sector companies are offering their services independently from the country you are visiting. For example, Uber and Bolt exist in multiple countries and users can use those services without having to do anything differently from what they experienced in their own home - even though the services are functioning slightly differently in other countries.

If governments were building services that are more standardized to citizens everyday environments, then it will also be possible to start offering a citizen experience independently from the country they are staying at, since the majority of services provided by the government are universal to all governments.

Building on top of the flow of a #bürokratt in previous example, the following flow would be possible between digital governments of multiple countries if citizen of one country is physically present in another country:



This chart is simplified as reality is more complex and this can involve automated cooperation between multiple services within either governments back-end, but processing within single government is entirely in that governments control. If a government does not wish to offer certain services cross-border, then they still have control over this just as they would without virtual assistants.

While integrating services between multiple countries can be a difficult problem, virtual assistants can potentially make it more natural due to standardization of how user input is interpreted - in other words, governments can remain as complex as possible, but if we are able to standardize the way we understand citizen intent, then it will make it much easier to offer cross-border citizen experience.

Certain things need to happen to make it work:

- Governments that wish to start offering virtual assistant based cross-border services need to agree upon a messaging standard to achieve semantic interoperability and establish technically a digital room or rooms where domain-specific messages are shared between governments. This room can be running on either government infrastructure.
- It is critical to separate the communication layer from intent detection and handling of said intent. While for a citizen losing a passport is very similar regardless of country they are visiting, governments themselves process this very differently in background.
- Multiple rooms can exist for this type of communication, including between different sets of governments and organizations as well as private sector.
- Governments retain complete control over their own services, the only point of connection are the rooms where citizen intents are shared as messages.

It is important to point out that in any case, this should not complicate an already existing complexity of digital government services. If standardization of citizen communication does not carry the expected benefits of actually making government services better, then another alternative is required long term.

Last but not least, a government could still - in theory - replace their whole digital government technology stack as long as they still understand messages published in such rooms.

3.3. Fallback routine

It is important that digital government does not create a new kind of tight-coupling with the use of a virtual assistant. It is paramount that should virtual assistant fail, there must always be a fallback routine in place. This is important to make sure that the digital divide is not further increased and that government services do not become a new single point of failure.

Concept of a Single Digital Gateway *(eesti.ee in Estonia)*, is not going to be replaced anytime soon as there needs to always remain a single trusted and verifiable source of truth for the citizen, regardless of how many different virtual assistants and other solutions they use. If all else fails, the digital government needs to have a backup.

The proposed fallback for citizen experience is as follows:

- Querying virtual assistant on your phone/tablet to solve the problem, if it fails:
- Using everyday search engine that directs you to single digital gateway to solve a problem, if it fails:
- Visiting single digital gateway and using its chatbot to solve the problem, if it fails:
- Visiting single digital gateway and using its internal search functionality, if it fails:
- Visiting single digital gateway and browsing its categories and hierarchy to find a solution, if it fails:
- Calling the government or walking to the government office.

Similar approach should exist for bilateral communication between citizen and the government, including when government needs information from the citizen:

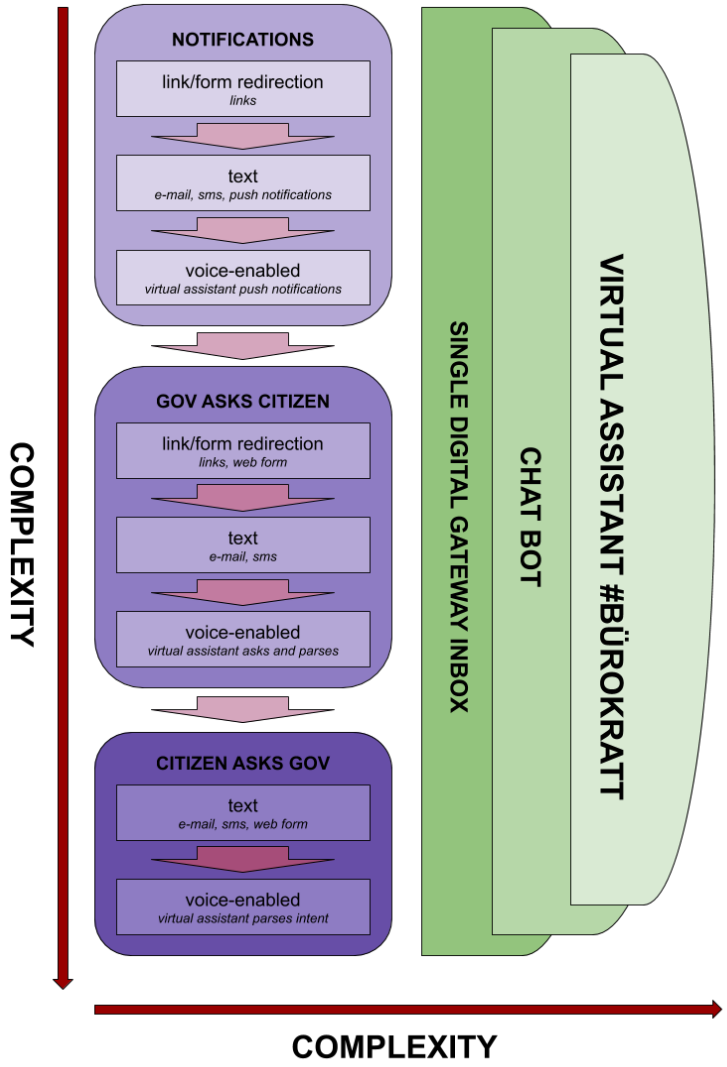
- Government attempts to contact the citizen over their phones virtual assistant, if it fails:
- Government sends a message in citizen-preferred protocol:
    - E-mail
    - SMS
    - Online messengers (that government can support)
    - If all of the above fails, then:
- Government sends message to internal inbox in single digital gateway, if it fails:
- Government calls the citizen or visits their physical address, if required.

Having a fallback options is also important for two reasons. For one, security: both when one of the communication channels is under risk or when there is a need to validate data from two different sources. Another reason is the digital divide or inability to access digital services - there needs to be an alternative in its most basic minimal form for most critical government services even without a computer device.

3.4. Key takeaways

Virtual assistants are the future, but their capabilities today are barely minimal - able to set alarms or calendar notes or remind you to buy milk. A lot of things need to happen to realize the original story proposed at the beginning of this paper. But just because the road is complicated does not mean that progress towards the goal cannot be made.

It is also a mammoth topic to implement in full scope from the beginning, as such it is important to do it step by step and learning from mistakes on the way:



It is important for governments to cooperate with private sector and vendors of mobile phones as well as voice-enabled IoT devices and related software to support not only domestic languages of your user, but to also integrate hooks with government services. What is really important is that the voice-enabled device understands the language and then understands the intent of the user. If this intent is government related, then it should be forwarded to government services that are able to ask further questions from the user.

At the same time, laying groundwork for virtual assistants does not require mobile phones and other devices to understand domestic language. Chatbots can already be used and piloted today in government services and those chatbots themselves should have hooks to back-end services that can be reacted to by other government services.

The most important thing is to start moving. Integrating artificial-intelligence enabled communication bots is a first step on a road to provide next generation citizen experience.

4. From monoliths to event driven microservice architecture

Complexities in describing Estonian digital government architecture in comprehensible scale is a challenge, but despite fragmentation of technology between administration sectors, at an architectural level there are many similarities.

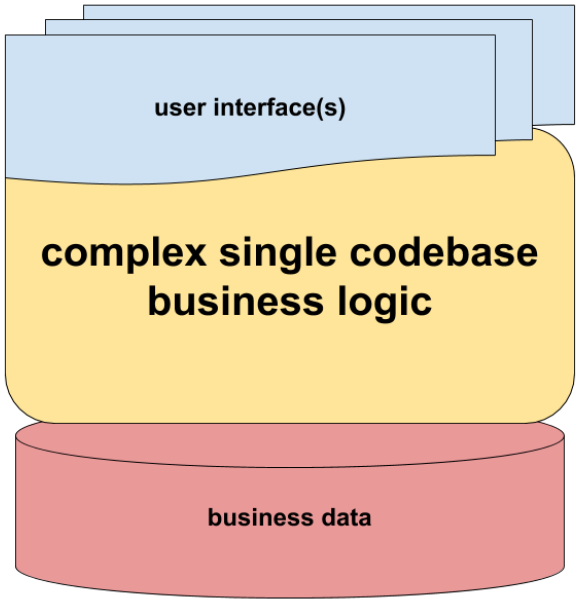
This section is split based on evolutionary steps of software architecture and their relation to digital government: monolithic architecture, service oriented architecture and X-Road that are present in all administration sectors today.

Despite numerous efforts across administration sectors, digital government suffers from multiple risks of single point of failure both from infrastructure, as well as service side. Multiple services either require the existence of other services such as digital government population registry - and fail without - or end up replicating entire registries to handle such a risk. Tight and dependent coupling of systems is a problem everywhere.

Multiple concepts are proposed to solve and address the aforementioned issues, from microservices and event driven architecture to further developments of X-Road as well as fact registries as a potential to simplify and streamline both archiving and backup solutions.

4.1. Monoliths

Majority of digital government services that are deployed and running today in are monolithic[43] stacks of software. Information systems and software has been traditionally built as monoliths, which means that the vast complexity of business logic, data and its use as well as user interface ends up being a single whole:



Many software developers attempt to mitigate the risks of monolithic software by building it in a modular[44] manner. Due to monolithic nature it is also difficult to reuse parts of the software, even if the software is developed as modules. Modular monolith allows the IT development team to develop large scale software keeping business functionalities apart from one another in separate modules, which is a healthy idea, but re-use of those modules is nearly impossible unless the core software framework is the same. And even then this is difficult, as in complex information systems business domain specifics are often leaking into each module, making module re-use difficult even when underlying software framework is the same.

But monoliths are an attractive software architecture for an e-service even in the year 2020 and it is shortsighted for a software architect to automatically consider every monolith bad. Monoliths

[43] https://en.wikipedia.org/wiki/Monolithic_application
[44] https://en.wikipedia.org/wiki/Modular_programming

are much quicker to set up and develop and easier to maintain and operate than the alternatives. They are easier to get up and running and to test business hypothesis and many argue[45] that if a business case is small, it is actually better to develop it as a monolith.

In public digital government monoliths carry with it multiple risks that are difficult to avoid and directly impact how quickly a software stack can be defined as *bad* legacy. This is because majority government services are larger than a small business case and require integrations with other systems across digital government:

- Monolithic systems are tightly coupled, which means that in order to change part of a monolithic system you need to have relative understanding of the whole. This is perfectly fine during the initial development cycle when the team is fully aware of the majority of the whole stack. But as time passes, this awareness will definitely get lost, especially if personnel changes and even more so when development partners change entirely.

- When part of monolithic software breaks, it impacts all service functionalities within the monolith. This means that changes in database structure can break functionalities that you may not even be aware of due to the vast scope of the system. This also increases the load in software testing as all tests have to encapsulate monolith as a whole.

- Monolithic software is usually written in a single programming language *(or often two, if including front-end user interface)* and using single database back-end. If certain programming language becomes less popular or database licenses become more expensive, this notably increases the costs in managing that system. This also means that the whole software is susceptible to security problems of selected technologies.

- Monolithic systems are very difficult to scale for performance. If a functional part of the monolith is under a heavy load then the whole system needs to be scaled for those maximum peaks, even during downtime when the performance requirements are exponentially lower. This means increased costs for infrastructure. Cloud technologies cannot assist here either as monoliths are not designed to run with multiple instances.

- Most monolithic systems can - at most - go through one or two additional development cycles before further development becomes inconvenient and engineering teams begin

[45] https://adevait.com/software/why-most-startups-dont-need-microservices-yet

thinking about rewriting everything. This is because additional developments frequently don't follow original development patterns and architecture, making new developments more like patches and injections to existing software stack. This increases the complexity of the system and makes it notably more difficult for another development team to understand the monolith.

- There's a good principle that if a system becomes more complex that is able to fit in an engineers head, the system is more complex than it should be. Not having comprehensive understanding of information system architecture is a risk for further developments.

- Monoliths can lead to vendor locking in both technology as well as partners. It is difficult to start implementing new technologies or features in another programming language or ordering developments from another partner who is not familiar with the system.

- It is very hard to replace parts of a monolith without having to refactor the whole system in entirety, even if monolith is developed using modular manner.

- Handling every single risk mentioned above becomes exponentially more difficult over time leading to an inevitable situation where technical stakeholders conclude that it is better to simply start over from scratch.

- Digital government also faces an issue of vendor locking, as multiple software solutions rely on both Java and Oracle stacks as well as VMWare on the infrastructure side. While this is addressed in newer developments with focus to use open source software stacks, many existing systems are still deeply rooted in expensive vendor systems and due to those systems being monolithic, are incredibly expensive to re-use not only due to the aforementioned reasons, but also due to license costs involved.

Despite all that, monolithic software remains popular. In fact, a notable software architect and consultant Martin Fowler has mentioned that despite the evolution of software architecture, monoliths - and monolith-first - strategy is continuously popular[46]. And if you are building a system for singular short-term purpose - such as marketing websites or a disposable blog - can be the right call.

[46] https://martinfowler.com/bliki/MonolithFirst.html

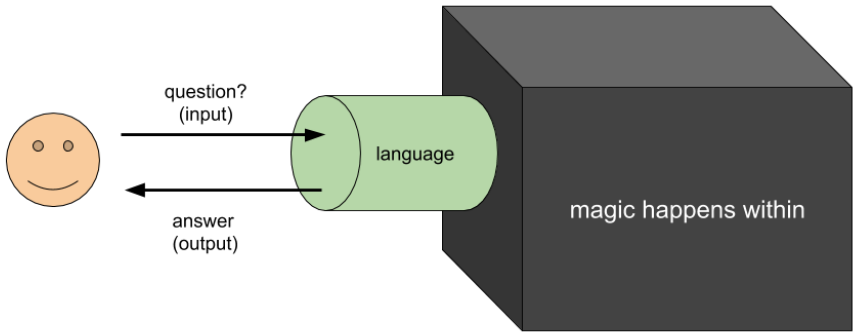
But as Jeff Bezos has said, the first decision when building a system is to decide whether your decision is a one-way or a two-way door[47]. Often once you start building a monolith, be sure if you are making a two-way door decision as if you are not managing it properly, it is difficult to break it up later down the line.

[47] https://www.entrepreneur.com/article/328284

4.2. Service Oriented Architecture

But while the majority of digital government services in Estonia are built as monolithic stacks of software, digital government stack as a whole is not monolithic and instead follows Service Oriented Architecture[48] pattern in an abstract sense.

Evolutionary idea behind Service Oriented Architecture is not originating from engineering directly, but instead from philosophy, most notably Computational Theory of Mind[49] and theory of Black Box[50]. The core idea is that something of value *(such as a service)* can be observed as a black box, where you provide it with specific *input data* and it computes and processes - without you being able to observe what it does - and it returns you a set of *output data*.



In Service Oriented Architecture this language and interface of communication is called an API - Application Programming Interface[51].

While the history of APIs go way back to the 1970's, it wasn't until 2000 that APIs and their use exploded through the use of internet as web APIs became a core part of Service Oriented Architecture. Web-based systems started integrating web-based APIs, creating a network of distributed functionality. Essentially the black box became a network of black boxes communicating with one another across the internet, reusing services without having to develop every single business functionality from the beginning. This itself is based on the concept of

[48] https://en.wikipedia.org/wiki/Service-oriented_architecture
[49] https://en.wikipedia.org/wiki/Computational_theory_of_mind
[50] https://en.wikipedia.org/wiki/Black_box
[51] https://en.wikipedia.org/wiki/Application_programming_interface

Distributed Cognition[52], which essentially says that "*smarts*" of anything can consist of multiple autonomous parts that are working together.

Service Oriented Architecture[53] pattern has many success stories starting from the year 2002. After the Dot-com bubble[54] burst many surviving companies realized that reinventing the wheel and trying to build everything by themselves in a monolithic manner into your own company is an impossibly expensive problem. Most notable example that came after Dot-com bubble is the controversial API Mandate[55] written and internally published by Amazon founder Jeff Bezos - a computer scientist by background. This mandate clarified multiple principles, including *(the following is paraphrased)*:

- Every team starts to offer their services strictly over APIs;
- Teams are only allowed to integrate and use services over those APIs;
- Every other form of integration is disallowed: direct linking, direct database connections and calls, shared memory and filesystems and backdoors of every kind;
- Every service has to be developed following the principle that an internal user is as safe or as unsafe as an external user. Without exceptions;
- Anyone that doesn't do this, will be fired.

While harsh at the time of publishing, it led to major transformation of two-three years of Amazon technology stack and services ending up as the business behemoth they are today. When visiting Amazon online stores today it may not be obvious that the majority of services that Amazon provides - including items that they are selling - are not actually on Amazon's own warehouses and are not actually updated by Amazon's employees. Majority of what we see on Amazon today is a vast array of API integrations to other stores and platforms.

With the popularization of Service Oriented Architecture another new feature emerged: a requirement of a central middleware, gateway, service bus or a road - of sorts - so that those

[52] https://en.wikipedia.org/wiki/Distributed_cognition
[53] https://en.wikipedia.org/wiki/Service-oriented_architecture
[54] https://en.wikipedia.org/wiki/Dot-com_bubble
[55] https://api-university.com/blog/the-api-mandate/

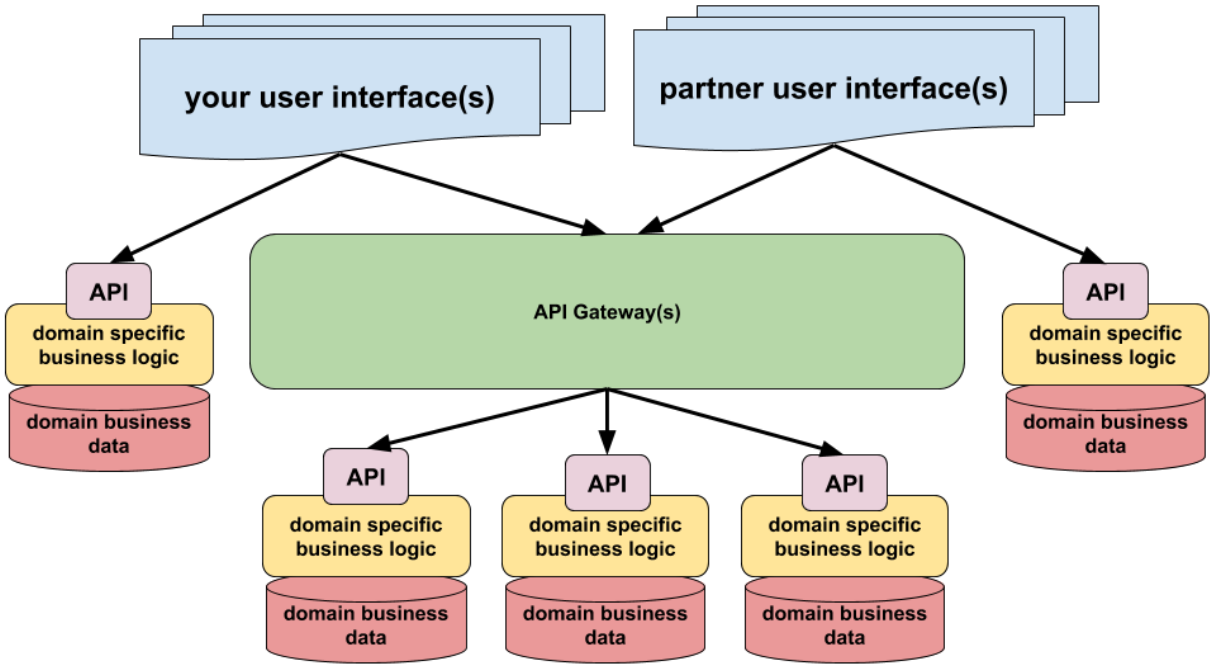
web APIs could communicate with one another. Over time it became difficult to tell if service is making requests to a middle man or the service API directly, but that is to be expected.

Two popular web API standards/styles emerged for making API requests: SOAP[56] *(Simple Object Access Protocol)* and REST[57] *(Representational State Transfer)*. The key benefits of emergence of these standards was that the language between systems become independent from the programming languages and databases just like with the example of the aforementioned black box. As long as you understood the language, you were able to use any API - regardless of what is actually running inside the black box. Of the two protocols, REST has emerged as more popular due to ease of adoption with web browsers and mobile clients.

The concept of API management[58] and API gateways are also widely popular when implementing Service Oriented Architecture principles. An API gateway essentially builds a wall around a set of systems and APIs can only be accessed over the API gateway.

[56] https://en.wikipedia.org/wiki/SOAP
[57] https://en.wikipedia.org/wiki/Representational_state_transfer
[58] https://en.wikipedia.org/wiki/API_management

A high level abstract view on Service Oriented Architecture is as follows:



Service Oriented Architecture carries with it multiple benefits over monoliths:

- You can replace anything behind an API. You can replace database or programming languages and as long as your system still understands and is able to respond to those API calls, nothing breaks for your API consumer.
- You can publish an API long before the system itself is ready and developed. This means that you can mock[59] parts of your software functionality and your consumers can already start testing their own integrations with it even while actual functionality is still being developed.
- You can scale up a performance of a single API stack without having to scale up the whole architecture. This means that infrastructure management will be far more cost effective.
- You can set your APIs behind an API gateway and make requests to the API gateway. This gives immense freedom to replace APIs themselves in the background or provide

[59] https://en.wikipedia.org/wiki/Mock_object

multiple versions of APIs. API gateway can also take care of logging and user authentication as well as request limits without having to have this logic in every single API.

That being said, there are still problems that are difficult to avoid:

- Service Oriented Architectures that use API gateway's need to be careful, because critical business logic may leak into gateways and can, if not well governed, become a monolith in their own right.
- Central API gateway is a complexity in its own right, as it requires permission and privilege management, authentication, logging, redirections and load balancing and internal networking and routing.
- API gateways can become performance bottlenecks and require careful load-balancing.
- API gateways can also create a false sense of security for services that are running behind the gateway. What this means is that if the gateway becomes compromised, so does every service that the gateway is intended to protect.

Amazon was not the only company to make such a change towards Service Oriented Architecture, SalesForce also adopted SOA to great success in 2007[60] and by today, Service Oriented Architecture is widely adopted among large scale organizations as a good compromise to manage some coupling within the organization.

Many of Service Oriented Architecture principles also ended up as part of Estonian digital government stack. Most notably in public sector developments the term "API-first" was evangelized by digital government architects. API-first meant that in any kind of information system design it is important that the back-end and front-end are separated and decoupled and communicate first-hand over API. While implementation of this principle was low, it has increased over time.

[60] https://www.infoworld.com/article/2641331/salesforce-com-announces-salesforce-soa.html

4.3. X-Road

Most known example of Service Oriented Architecture in Estonian digital government stack is technically X-Road. The name of X-Road originates from the idea of network of roads and crossroads that connect different information systems between one another.

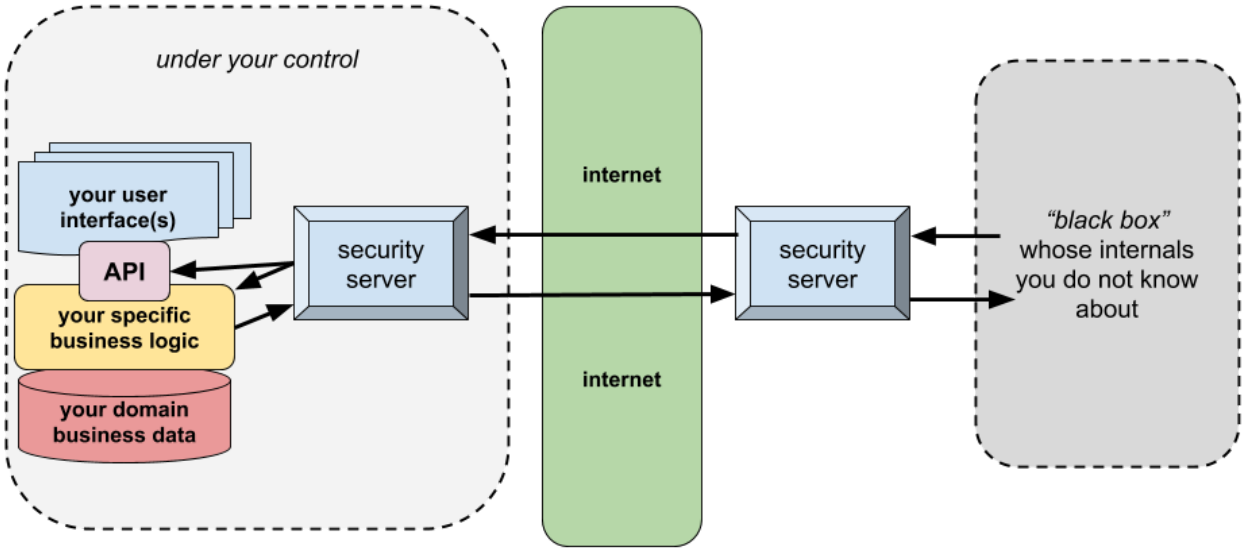
Originally launched in late 2001, X-Road has been fundamental to Estonian digital government success, but by today X-Road is not an Estonia-only solution. It is used in various scale by Finland, Iceland and the Faroe Islands and is co-developed together with Finland under Nordic Institute for Interoperability Solutions[61]. Iceland will also become a NIIS member from 2020.

X-Road is a single solution to connect information systems of vastly different technology stacks with one another between multiple administration sectors - somewhat similarly to what an API Gateway would do in Service Oriented Architecture - however, X-Road itself doesn't have a central API gateway[62].

Most implementations of X-Road integrate SOAP APIs due to SOAP being popularized in early 2000's. REST is supported by X-Road from 2019 and wider implementation of REST is planned from 2020 onwards by X-Road users.

That being said, the architecture of X-Road is slightly more advanced from classic Service Oriented Architecture. Every information system that accesses X-Road has a required component of a *security server*, which in many ways acts like a local API gateway that is only intended for X-Road communication. Security server of X-Road is essentially an application-level gateway[63]. While in classic Service Oriented Architecture the requests are made through a single gateway, in X-Road the communication happens between two separate secure *gateways*. This gives both more control and security to both sides of the transaction, without requiring a central single-point gateway that becomes a risk dependency for all.

[61] https://www.niis.org/
[62] https://www.niis.org/blog/2020/1/20/interoperability-puzzle
[63] https://en.wikipedia.org/wiki/Application-level_gateway



As can be seen, many principles are similar to Service Oriented Architecture. You could have *(and in Estonia's case there are)* multiple *black boxes* behind X-Road security servers: validating requests, logging requests and sharing data in a secure manner.

X-Road is one of the main reasons why Estonia has been able to be this fast in growing their digital e-services across the whole nation without requiring monolithic central databases for every government service. This has been accomplished by giving administration sectors complete freedom in building their information systems, but as long as they are connected to X-Road, then they can use services themselves or provide services to other administration sectors over X-Road with no impediment, if another administration sector uses different technologies. In other words, if you speak English, others that can speak English can understand what you are saying, can ask information from you and you can do the same in return - in a secure manner.

X-Road only sets specific minimal demands on the technical components of the services that want to communicate over X-Road or want to provide services over X-Road. Primary requirement for information system is that it is able to communicate with the security server and understand requests coming from the server.

But while X-Road has been immensely successful helping digital government to evolve into where it is today, a few things require addressing to make sure X-Road does not become outdated:

- Information systems that are connected over X-Road are internally complex and often monolithic, resulting in difficulty in implementing new business rules driven by laws and regulations and X-Road by itself encourages monolithic approach due to the complexity of setting up and running security servers.
- Getting X-Road up and running - even for trialing reasons - is very complex and often considered an impediment. Expectations of trialing software stacks today are to simply download, install and configure, but this is only partly possible with X-Road today with the majority of difficulty related to set up configuration.
- While the ideal vision for services available on X-Road has been for each service to have their own autonomous service endpoints to connect to, due to complexity multiple large scale information systems are designed so that they share the same endpoint - even if internally the services have little in common with one another.
- While X-Road allows for complex requests from multiple different sources, these requests are synchronous[64]. Due to this, implementing massive data analysis is becoming a problem with X-Road, which was not originally intended for massive data requests. However, as business demand for such requests is there in the era of data analysis, emerging alternative solutions might fragment data exchange in digital government.
- With the emergence of APIs in private sector both domestically and at an international level, integration to APIs have become easier over time. Integration of X-Road in comparison is difficult and is often brought out as a negative. This needs addressing, if X-Road is to become a solution for not only domestic, but also cross-border communication.

While Service Oriented Architecture benefits also apply for X-Road, due to nation-wide digital government scale of these services and their integrations has ended up creating tight coupling

[64] https://en.wikipedia.org/wiki/Request%E2%80%93response

problems on an unprecedented scale. This means that while it is indeed possible to replace a service and its technical components behind an X-Road security server just like you could replace an API behind a gateway, in reality it happens very infrequently due to the size of those services. While no fault of X-Road, due to synchronous and tightly coupled integrations the digital government stack has become a distributed monolith in its own right and steps have to be made to mitigate this in the future.

This also impacts secure sustainability of critical government services. Due to architectural complexity and tight coupling, services have impediments to function without their dependencies. This is especially evident in the concept of data embassies[65]. Estonia is pioneering in setting up data embassies outside their own territories, meaning that critical data would be stored outside physical territory of the Republic of Estonia in order to attain digital independence of its citizens, but due to architectural tight coupling these services in data embassies are merely data backups and cannot function independently.

With the current architecture of government technology of Estonia it is difficult, if not impossible, to assure that government technology stack isn't susceptible to cascading failures due to such dependencies.

[65] https://www.valitsus.ee/en/news/estonia-establish-worlds-first-data-embassy-luxembourg

4.4. Microservices

Before microservices can be better explained, it is a good idea to understand the concept of Ship of Theseus[66], which is a philosophical puzzle. There is a ship, called Theseus, that sets sail around the world and visits all the ports of the world. Every now and then parts of the ship break down or sails need repair or replacement. By the time it arrives back home years later, every single piece of the ship has been replaced with a new component, some pieces many times over. Can you then still claim that the ship is Theseus by the time it arrives home?

The whole concept of Conway's Law and Domain Driven Design in earlier sections are about making the responsible business stakeholders as well as the engineer accept the reality that systems need to be as flexible as humans are - for better or worse - and designing this flexibility into core of the systems is an inherent responsibility of both stakeholders: technical and business alike.

And while Ship of Theseus is in many ways an ideal, information systems designed in a way that keeps these concepts in mind will lead to more natural migration of technology and better standardization and evolution of standards.

Martin Fowler, renowned expert in the fields of software architecture has said that if you are unsure how to do any better, building a monolith is not a bad decision. It will be cheaper and quicker to implement.

But taking into account the requirements of the public sector and the aforementioned topics of proactive background services, design, virtual assistants and needs for a flexible architecture, it is expensive for digital government to build services any differently than in a flexible manner. Anything temporary that is important enough tends to become permanent, especially in governments.

While Service Oriented Architecture was an important evolutionary step in software architecture and took a large step closer to enabling more agile development as well as better system design through Domain Driven Design, it still encountered multiple issues. These issues were tackled in 2011 in Venice where a software engineering workshop was held and the term "microservices" was actually first mentioned in the context that it is known today and the use of

[66] https://en.wikipedia.org/wiki/Ship_of_Theseus

the new technology buzzword has exploded since 2012. Microservices are a way to build information systems that have features similar to the Ship of Theseus.

Microservices is not a revolutionary concept, rather it is an evolutionary step from monoliths to service oriented architecture to microservices. In fact, microservices are a way to build a more autonomous and scalable Service Oriented Architecture, thus most of the benefits of Service Oriented Architecture still apply to concepts of microservices as well.

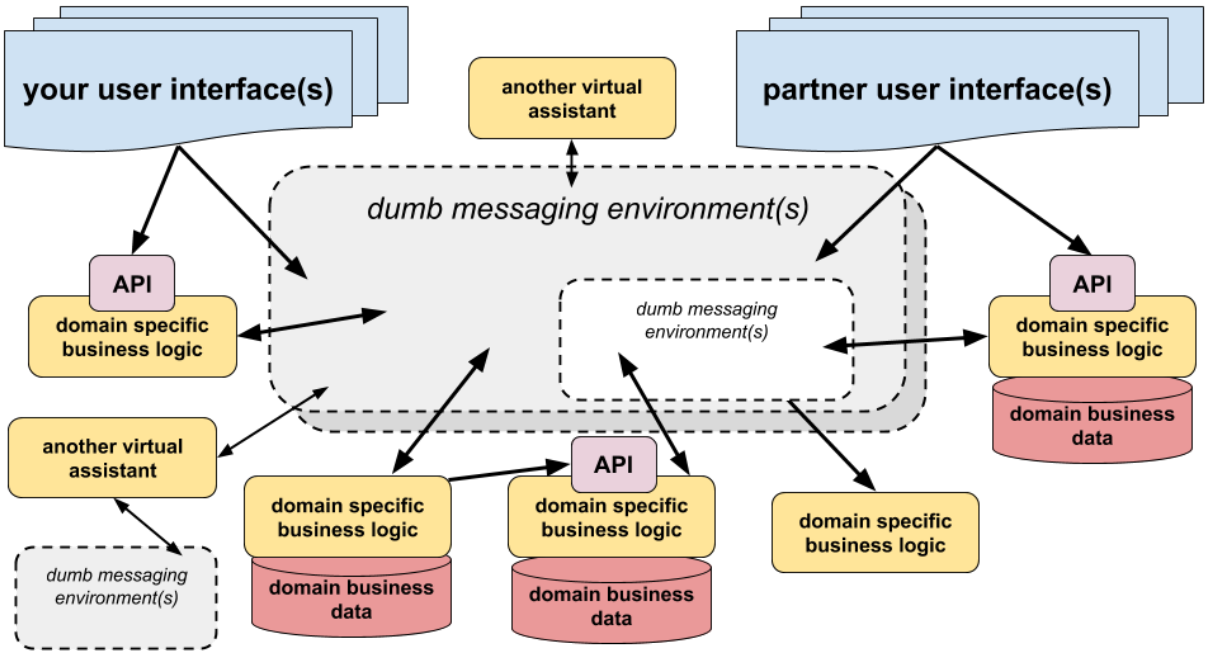
The truly revolutionary part of this evolution is that concepts of Service Oriented Architecture are merging of microservice architecture concepts with the concepts of Event Driven Architecture[67]. The latter is a concept as old as Service Oriented Architecture, but with the emergence of microservice patterns the two concepts are being combined for the eventual benefits of both.

In Service Oriented Architecture it was expected that all of the technical services have an API that can be used by other services and user interfaces. Event Driven Architecture dispels this expectation: your services may have an API, but they are not required to do so. Instead what is expected is that your services themselves are connecting to *"dumb messaging environments"* in a concept called *"smart endpoints dumb pipes"*[68]. A good real-life example is that your employees are smart, but the physical spaces where they work in are dumb. Thus technical components are smart, but the environments they communicate with each other are dumb and often agnostic to business smarts.

At a high level this looks something like this:

[67] https://en.wikipedia.org/wiki/Event-driven_architecture

[68] https://medium.com/@nathankpeck/microservice-principles-smart-endpoints-and-dumb-pipes-5691d410700f



Multiple differences are already evident compared to a more traditional Service Oriented Architecture: some of the services have APIs, some don't. API gateway has technically vanished, but has actually been replaced by an expansive dumb messaging environment (or environments, as you could have many) that every service can connect to. Security duties of API gateways are carried by services themselves as they become more autonomous. And multiples of these messaging environments are possible while the service can be connected to various different messaging environments.

Features of a good microservice

As mentioned previously, microservice is an evolutionary step from a technical component with an API within Service Oriented Architecture. Golden rule of microservices is to be able to change a service and get it to production without having to change anything else. Simple?

Reality is that ever since microservice became a buzzword, it has been forgotten that microservices don't exist in their own right. Similarly to Agile development, it is also not actually a problem for engineers to solve alone.

Leaving microservices for engineers to solve results in microservices being built from the technical perspective instead of business perspective. As Conway's Law and Domain Driven Design has shown, this should not be the case. An engineer will rely upon their technical background and compartmentalization of technical components and logic: engineers grow up with principles to use framework, build modular systems and not to duplicate data. But a microservice is not an alternative way to encapsulate and standardize communication between technical modules and database components.

To get to actual microservices it is important to start with Domain Driven Design. Doing anything differently means gambling and hoping to avoid Conway's Law.

Thus to expand on what are the identifying features of a good microservice, it is necessary to expand on what are the identifying features of a good API within Service Oriented Architecture:

- A good API is stateless HTTPS/REST service. Service being stateless means that every request to the API happens in complete isolation and output of a stateless API is generally always the same as long as the input is the same.
- API should traditionally not act as an interface for Remote Procedure Call[69] or RPC. Remote Procedure Calls are not stateless as they depend upon their environment.
- A good API service is loosely coupled and autonomous. This means that even if other services run into conflicts or become unavailable, service is still up and running - even if with limited features.
- A good service is versioned, meaning that if new features are added to the service then existing functionality does not break down. It both allows for backwards compatibility within reason, as well as more opportunity to evolve the service without creating fear of further development due to the amount of consumers of the API. This assumes well planned change management[70].
- A good service implements caching protocols and standards[71].

[69] https://en.wikipedia.org/wiki/Remote_procedure_call
[70] https://en.wikipedia.org/wiki/Change_management_(engineering)
[71] https://en.wikipedia.org/wiki/Web_cache

- A good service is mockable, meaning that it is possible to see and understand how an API works even if actual requests to that specific API are not made.
- A good service is self-documenting and publishes an internally accurate documentation. A good tool for this is Swagger[72]. Swagger is a tool that produces API descriptions in OpenAPI description format. Earlier the specification was known as Swagger Specification, but it was renamed to OpenAPI Specification in 2015.
- A good service is monitored and logged by the environment and includes traceable correlation ID that can be used to trace a wide array of business requests that are dependent upon one another. *Note that there is no existing recommended standard for correlation ID's and in the case of Estonia, a standard needs to be agreed upon by the engineering community. Once defined, this is expected to be implemented in the next versions of X-Road.*
- A good service is covered with acceptance and integration tests that assure integrity and quality of functionality.
- A good service is idempotent, which means that not only is it stateless, but it also handles the concept of eventual consistency. A good example is that deletion of an object over an API should be possible from multiple clients at the same time without one of the consumers getting an error - as long as both consumers had the permission to delete the object.
- A good service uses a non-central method to authenticate user requests. Standardized solutions such as JSON Web Token (JWT) are recommended for this, as it allows to authenticate requests without requiring tight coupling with user authentication services. This also makes it possible to enable concept of Single-Sign On[73]

All of the above features of a good service are benefits of Service Oriented Architecture evolution over monolithic software. In terms of microservices, those features are not replaced and instead a few new features and expectations are added:

[72] https://en.wikipedia.org/wiki/Swagger_(software)
[73] https://en.wikipedia.org/wiki/Single_sign-on

- A good microservice is built for cloud and supports as many concepts from Twelve-Factor App methodology[74] as much as possible. In case of Estonia, due to low cloud readiness the architecture council agreed upon four simplified requirements for services:
    - Service setup and configuration is automated through scripts and service is possible to start up and be recovered with the use of those scripts.
    - Service must be able to consist of multiple independent instances.
    - Service must be scalable potentially between at least two different physical locations.
    - It must be possible to back up data of the service as well as restore data of the service in case of corruption (with automated scripts).
- A good microservice is primarily choreographed, meaning that it responds to its environment and acts as a consumer and/or publisher in dumb messaging environments.
- A good microservice continues to be stateless and does not store data within its container irrelevant to the number of instances/copies that are being run at the same time.
- A good microservice is re-usable driven from its design. Following Domain Driven Design principles, a good microservice is responsible for a single domain in whole or autonomous flows of said domain in parts.

It is also important to mention that microservice does not mean a small codebase, it means a small, autonomous set of business functionalities that have a potential to be infinitely scaled or re-used in serving a wide variety of business cases.

Paraphrasing the words of Sam Newman from his book Building Microservices, a good principle for microservices is to *combine services and functionalities that all change for the same reason and separate those functionalities and services that change for different reasons*. Internally this means the same thing as the concepts explained in Domain Driven Design and ideally your

[74] https://en.wikipedia.org/wiki/Twelve-Factor_App_methodology

microservice architecture - as a result - maps to your actual organization and business process in the end.

Note that microservices are not a silver bullet. When implementing microservices certain things still need to be kept in mind:

- While autonomy is an ideal, it can be an incredibly expensive solution and often compromises need to be made. While autonomy of microservices implies that services are fully autonomous in their own function, library sharing for common functionalities (such as for JWT validation) should not be built from scratch for every single service - thus library re-use is a good option. But libraries and frameworks inherently create their own threat of technology locking and coupling, so decisions where to use libraries must be carefully considered.

- Concept of microservices are becoming even more evident in Edge computing[75] and Internet of Things[76] as soon it is difficult to distinguish autonomous IoT devices from digital microservice - both, if designed well, carry almost indistinguishable similarities.

- It is difficult to do microservices properly without also implementing cloud technologies well. While it is possible, the true benefits of microservices come to fruition once microservice architecture is built for and deployed to cloud - but this also means that your IT development teams need to be well versed in both microservices and cloud technologies.

For further research, most notable examples of microservice success stories come from Spotify, Netflix, Amazon as well as BestBuy. The upcoming *2nd Edition of Building Microservices* book by Sam Newman is also expected to bring out further detailed use cases of success stories - and failures.

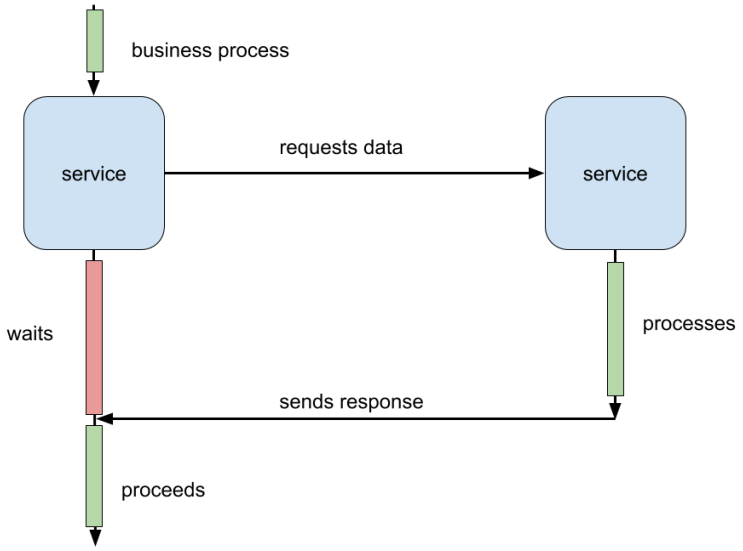
Synchronous vs asynchronous communication

While microservices have multiple important features that could be expanded upon further, as this paper also focuses on message based event driven architecture it is important to cover the topic of synchronous vs asynchronous communication in order to achieve autonomous

[75] https://en.wikipedia.org/wiki/Edge_computing
[76] https://en.wikipedia.org/wiki/Internet_of_things

microservices and manage the risk of cascading failures. This concept is not just important for technology, but also for Domain Driven Design and business processes themselves.

Synchronous communication means that if you request something - for example if you wish to grant permissions to your colleague to access a certain information system and in order to do so, you need security department to make required changes - you make the request and will be waiting for confirmation of security department response. Until you get that response, your own business process is locked up. Imagine that you would be unable to continue your work with any other task until that response comes?



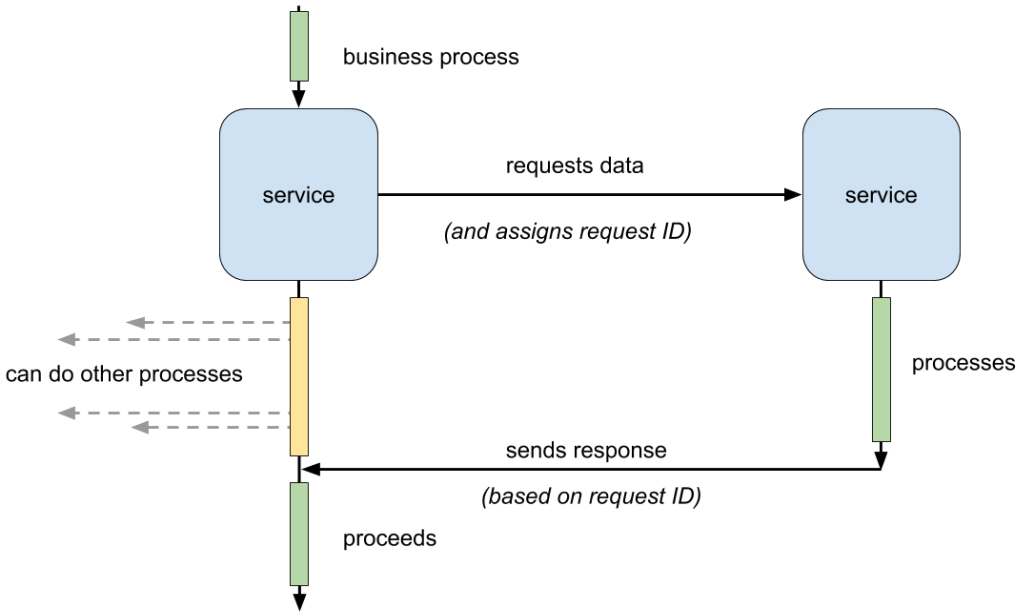
In day-to-day life this situation seems unfathomable, but the reality is that the majority of information systems communicate with one another over synchronous requests today both within government and without. These requests may time out, causing complications and possible cascading failures across the business process.

The reason why synchronous requests rarely become a problem is that information systems and processing is usually fast and responses are far quicker than having to wait for a reply from another department.

But synchronous requests are a problem that needs handling in architecture. If your business complexity involves the use of multiple services and multiple API requests, then all of those requests are adding up in response time. If the service you are using makes their own requests

to further services, then processing time of all of those services adds up to your own waiting time. This situation is made worse if some services are more popular and have to serve hundreds, if not thousands of requests at the same time. This locks up every single synchronous request while those requests are being served.

Asynchronous communication is the alternative that avoids this problem. While messaging environments explained in later sections inherently don't demand asynchronicity, it is a more natural form of communication that is also more similar to how organizations work internally. A service is able to start multiple processes and sub-processes and handle them at the same time as multiple threads.



This is a far healthier model to handle communication between technical services just like it is healthier for organizations in whole. Technology also allows to scale services, especially autonomous microservices so the scale can be balanced across thousands or interactions within a second, if required.

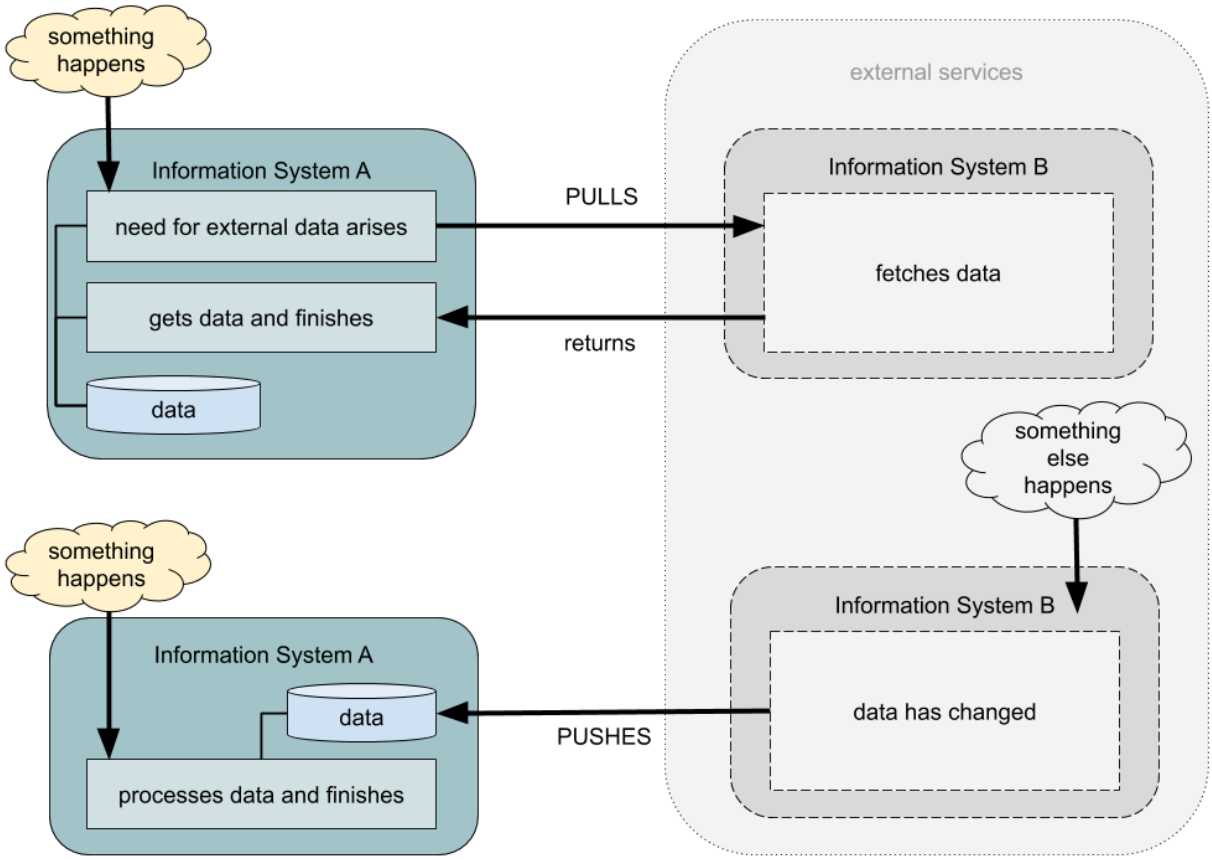
But this can be more complex for an IT development team to implement and requires experience in making multithreaded requests. Benefits of such implementation mean a more autonomous services in architecture and better potential for scalability.

Event driven messaging environment

One of the key drivers behind microservices is the challenge to have even more decoupled software architecture than possible with Service Oriented Architecture. Coupling is bad as it means that dependencies are impeding the growth and evolution of software architecture.

If you build a service that is used by only one consumer and if you wish to change that service, you then have to manage the change with that one consumer. This is often a phone call or an email degree of separation problem and not difficult to handle. But if your service has dozens, hundreds or thousands of consumers that are dependent on your service, making changes to this service is a far more complex problem and in the example of digital government can mean an incredibly slow to non-existent changes in critical technology stacks. This is because those integrations are tightly coupled between two information systems.

Traditional integrations between software systems follow a *push* or *pull* model. This is in many ways the aforementioned choice of synchronous requests versus asynchronous requests on a large scale. In pull model you connect to services that hold the data that interests you, in push model the data that interests you is pushed to you as it happens and you may also push data from your systems to other systems, that are interested in your data.



While the pull model is not bad and can be beneficial in some circumstances, it is essentially a direct dependency that needs to be managed. This means that if the Information System B on the previous chart is unavailable, then Information System A may also fail to function. If this system is not fault tolerant, then it can lead to further cascading failures across the architecture. In comparison, pull model makes sure that Information System A can still function while Information System B is down because it depends on data that is provided at the time Information B is up and running.

Service Oriented Architecture led architects to implementing concepts of API gateways as a solution for loose coupling, but while it works in some instances, the API gateway is frequently too smart, creating coupling in itself that needs to be handled and managed separately. In order to tackle the issues of coupling within complex architecture, Event Driven Architecture has emerged with possibly the best and most natural solution. The hypothetical *loosest coupling* is achieved with your system being subscribed as a listener to *dumb message rooms* where data that interests you is pushed to and your services reacts to those events. This creates a more

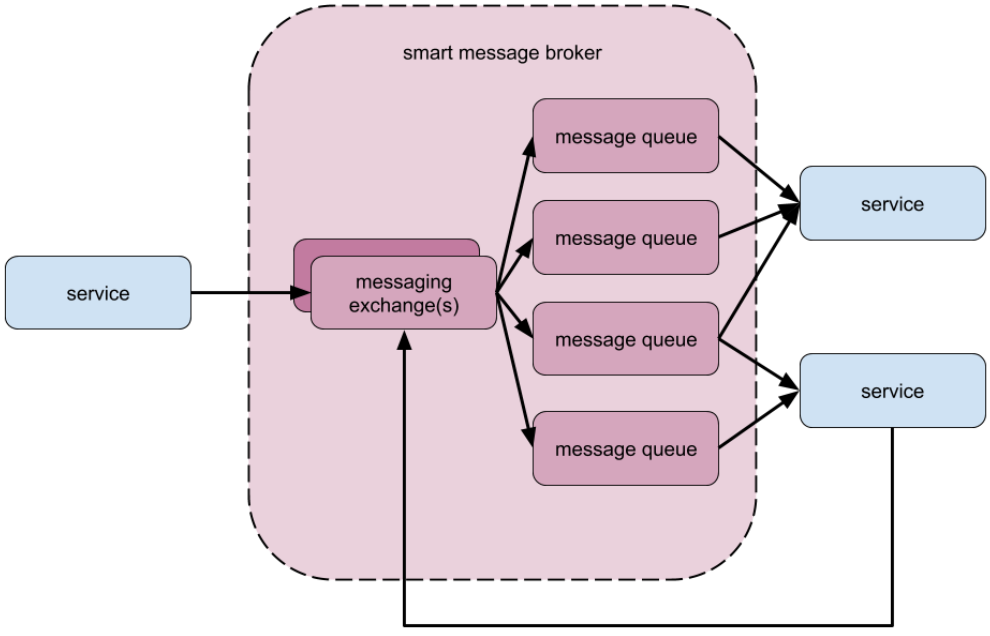
dependency-loose environment for the services to thrive on while it increases complexity as data sharing now needs three separate endpoints.

The idea behind dumb messaging space and messaging rooms is - similarly to every good concept - rooted in how humans themselves work and cooperate. As humans we cannot be sure if the way we cooperate in real life is the absolute ideal of communication possibilities, but as we are restricted by Conway's Law, the best we can do is have technology automate the best routines in our everyday life and this is exactly what dumb messaging rooms are meant to represent.

The core concept is that messaging rooms are like working spaces and meeting rooms in your organization and technical services that are communication participants in those rooms are like employees in the organization. This concept was illustrated earlier by the practical example of Domain Driven Design.

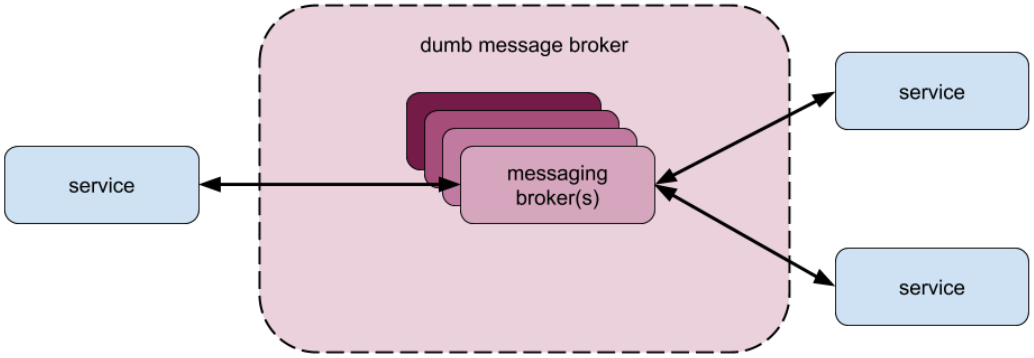
There are two popular methods for setting up decoupled messaging environments:

- Smart broker / dumb consumer - solution where services are being served messages intended only for them. Good examples of such technology stack to try out without having to build it from scratch yourself is RabbitMQ[77].



[77] https://www.rabbitmq.com/

- Dumb broker / smart consumer - solution where the services themselves have to be aware of which messages are important to them and how to use them. Good example of such technology stack to trial without having to build it from scratch yourself is Apache Kafka[78].



One option is not necessarily better than the other and whether a certain kind of messaging solution serves your business requirements better depends on what those business requirements are.

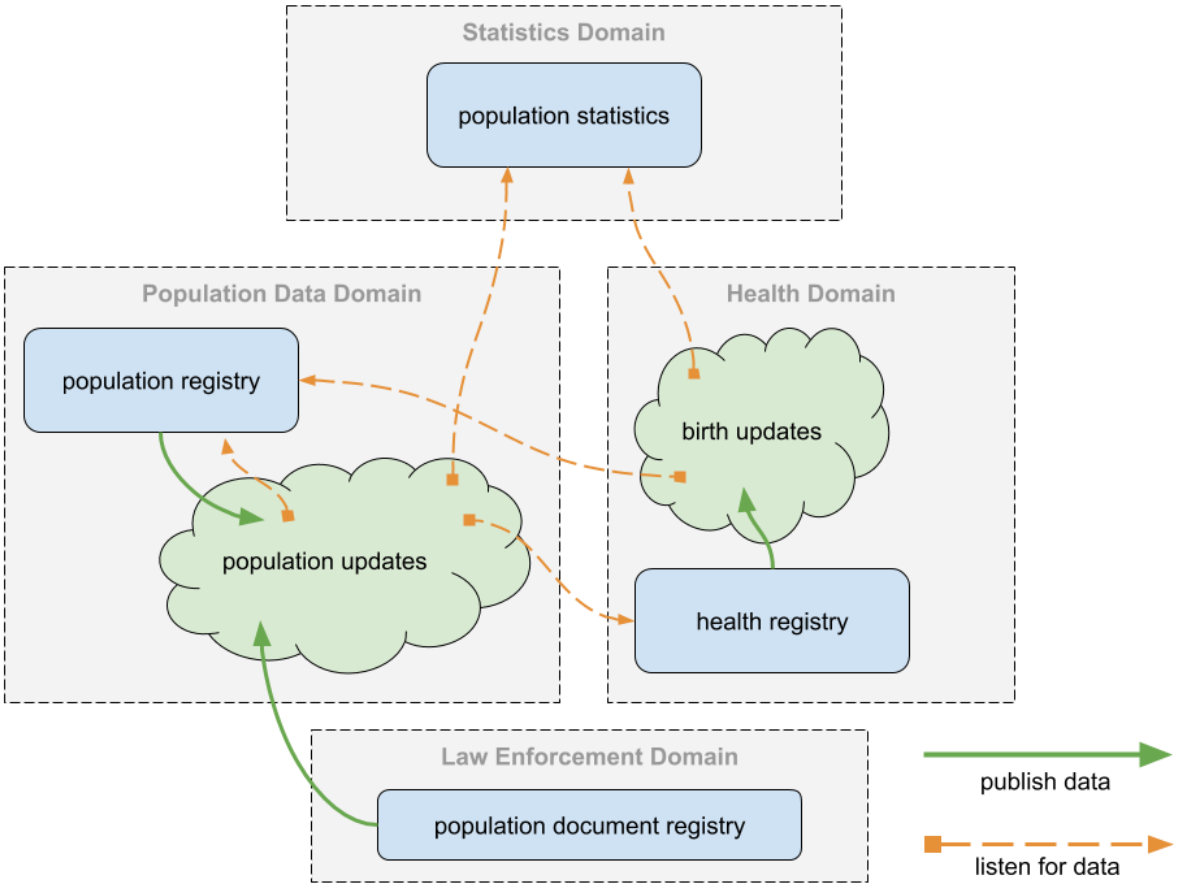
It is important to note though that if you are dealing with large scale organization and especially a set of dependent organizations - such as administration sectors in public sector - then the concept of dumb messaging broker and smart consumer is a better natural fit. This allows for message rooms to be shared between administration sectors without enforcing business requirements onto the message broker itself. Same applies to cross-border data sharing, as governments expect their services to be smart and in control.

The way dumb message brokers work is that they rely upon topics and *publish/subscribe*[79] model of integration. Smart consumers (*such as your technical components*) are listeners of message rooms as subscribers to certain topics that impact their workflow. If something happens of interest to them, they are able to react to it.

[78] https://kafka.apache.org/
[79] https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

Reacting to such messages is the core concept of event driven architecture as it implies that your architecture is driven by events that happen in your business logic, such as actions of a citizen on a website, entry of forms, signing of documents and so on.

In the scale of a whole country or even a group of countries, every administration sector can be an owner of message rooms that are part of the services they are responsible for. Services that are using these message rooms can by themselves internally also use multiple message rooms. For example, a complex business process that is mapped using Business Process Modeling can connect to different message rooms within their flow - to support cross-domain proactive services.



Such messaging environments can become a healthy evolution for government technology stack as a whole, as it allows to decouple services from one another in ways not reasonably

possible before. It is only in recent years that server infrastructure and cloud has matured well enough that such principles are scalable for organizations in the size of a nation.

CAP Theorem

In order to round up issues of tight coupling from monoliths to more loosely coupled microservices, it is also important for business and technical stakeholders to understand the concept of CAP theorem.

Known computer scientist Eric Brewer is the author of CAP Theorem[80] which states that it is impossible for a service to provide more than *two* of the following three features:

- **Consistency**: Every read receives the most recent write or an error.
- **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

While this may sound a little technical, what it means in reality from service running perspective is that your service can only be one of three types:

1. Consistent and partition tolerant
2. Available and partition tolerant
3. Consistent and available

In architecture where services are distributed and integrated over network, partition tolerance is a *required* feature that cannot be avoided. This means that the third option is not actually an option in distributed service oriented architecture and as a result you can only pick between the following two options below.

Consistent and partition tolerant service

A service that is consistent means that the data of the service is consistent irregardless where you request the data from. If your consumer requests data from your service they always get the most up to date state of said data.

Partition tolerant service means that the service will be able to function even if there are network communication errors between multiple instances of the service.

[80] https://en.wikipedia.org/wiki/CAP_theorem

However, consistent and partition tolerant services sacrifice availability. This means that at times the service is unavailable either due to having to synchronize data in order to assure consistency or dealing with partition tolerance.

Available and partition tolerant service

A service that is available means that the service can be connected to and requested data from at all times.

Partition tolerant service means that the service will be able to function even if there are network communication errors between multiple instances of the service.

However, available and partition tolerant services sacrifice consistency. While in good architecture the data becomes eventually consistent, at any point in time data is not consistent and 100% up to date - but it is always available.

The concept of eventual consistency

The decision whether to build partition tolerant services that are either consistent or available can depend upon what the business requirements are. As such, it is critically important to make sure that all parties involved understand that services are inherently incapable of being both 100% available and 100% consistent over distributed network at the same time. Any expectations for such are misguided.

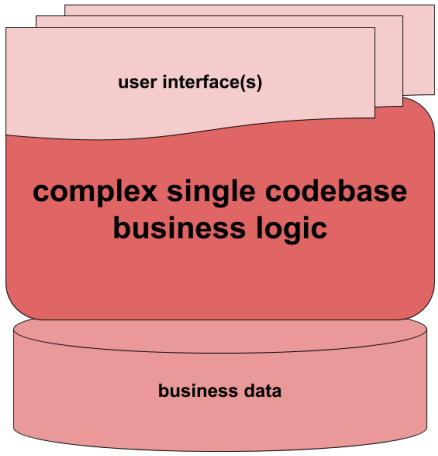
The concept of eventual consistency is possibly the only option for large scale information architecture that attempts to be as loosely coupled and flexible as possible. The idea of eventual consistency is that you accept that your entire system architecture - up to the level of whole digital government stack - is never consistent and 100% up to date with the most accurate information at all times.

However, eventual consistency means that eventually the information will be up to date in a service or database that presently has outdated information.

Handling and planning for this eventual consistency is something that technical stakeholders need to take into account early in planning software architecture.

Cloud-native services

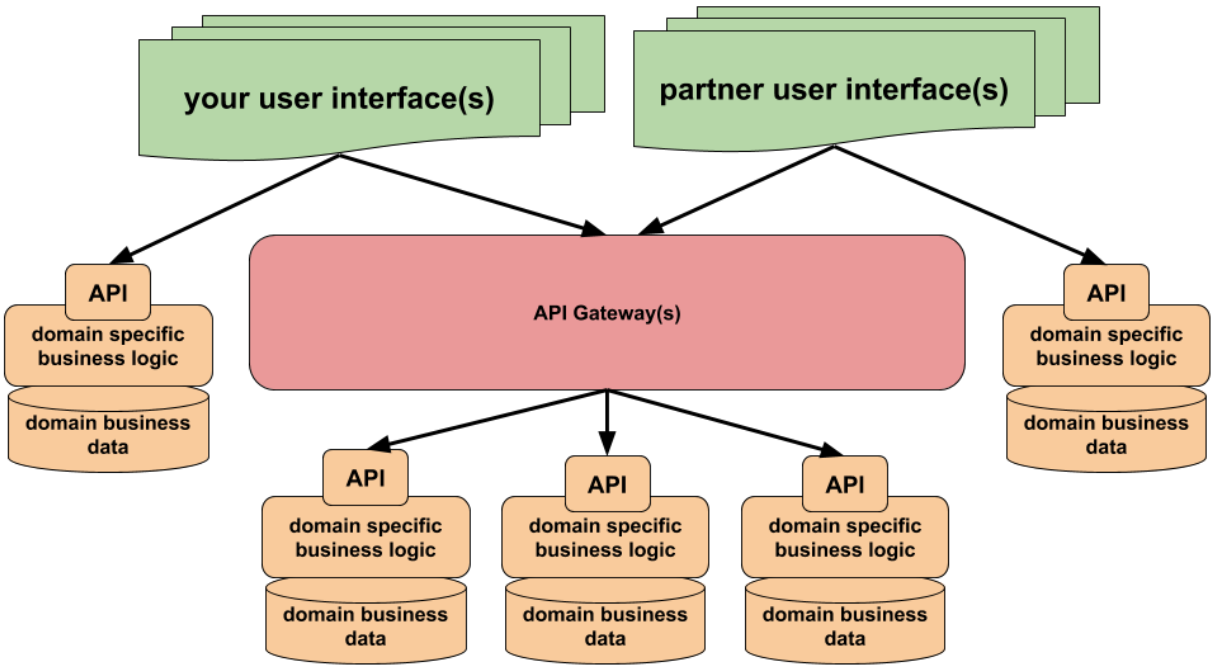
Microservices are inherently cloud ready, but not everything that is cloud ready is a microservice. And while there have been thorough books written about the topic, it is important that business and technical stakeholders share a common understanding regarding what it means for services to be cloud-native.



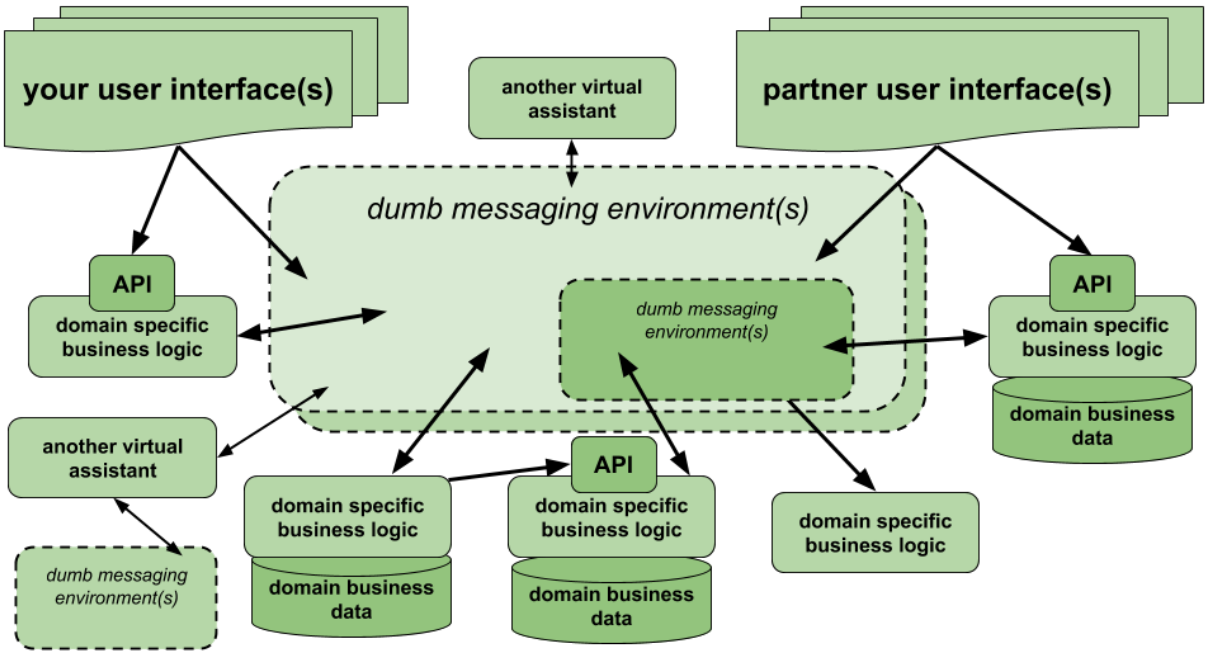
Monolithic information systems are not cloud-native. If demand for your monolithic service increases, your only option is to either acquire more hardware for server that runs this service, or more scale for your virtual machines. If peak demand for the service decreases, you are stuck with increased capacity that is not used.

Amazon originally faced this problem as they built huge data centers with increased capacity because of Black Friday and Christmas. The amount of sales volume during those hours was multitudes higher than during any other season, but this also meant that during every other season Amazon had to maintain notably larger than demand requires server stack and capacity.

In order to solve this problem, Amazon ended up offering cloud as a service for companies and private individuals alike. The extra capacity was monetized and diverted into extra revenue, ending up with one of the most successful cloud services at the time of writing of this paper.



In Service Oriented Architecture cloud-nativeness is more widely adapted. User interfaces are decoupled from backend logic over APIs and can be scaled independently. API gateway scale is still a problem and the majority of services are likely to be virtual machine stacks that are running and supporting their API functionality. This is a slightly easier problem to scale, but peak hours can still impact increased costs and requirements.



In microservice architecture every component can be scaled individually if deployed to cloud, such as AWS. While this is a very simplified model, if all services and service components are in cloud, then they can be scaled per single component demand. If good principles for microservice architecture are kept in mind, then cloud platform is able to scale components in a dynamic manner[81] by creating multiple instances temporarily to deal with the load.

If cloud infrastructure is shared between multiple administration sectors then benefits of increased performance can also be shared. Instead of requiring high performance once a week and having to pay the difference during downtime, the available capacity could be used by other administration sectors.

What this effectively means that with well implemented cloud platform and architecture you would be actually only paying for what you are actually using and this could have a positive impact on service costs, opening up opportunities to perhaps develop a service that beforehand you were unable to due to infrastructure costs.

[81] https://patterns.arcitura.com/cloud-computing-patterns/design_patterns/dynamic_scalability

This could be considered even further towards serverless[82] architecture – how microservices could be deployed without the need for maintaining underlying infrastructure resources.

Chaos engineering

A true test of cloud based microservice architecture comes from concept of chaos engineering[83] and tools such as Chaos Monkey. The core idea of Chaos Monkey is that parts of your information system - perhaps networking, perhaps database, perhaps static file serving server, perhaps session storage - gets removed - randomly. This approach was pioneered by Netflix who uses chaos engineering to this day to assure integrity and quality of their architecture.

Applying chaos engineering to public sector information systems will, in many cases, lead to cascading failures across the information system, requiring restarting of services and extra validation steps.

But the litmus test of well engineered and well architected information systems is to survive tests of chaos engineering and not only remain up (albeit with limited functionality) during downtime of certain services, but also recover.

This is possible when designing with CAP theorem and microservice autonomy and replicated cloud infrastructure in mind.

Topics requiring further research:

- Could chaos engineering be reasonably implemented in public sector services?
- Is designing information systems without chaos engineering and CAP theorem in mind a risk for public sector services?

Risks of microservices

This paper focuses on the evolutionary road of large scale architecture from Monolithic Architecture to Service Oriented Architecture to Microservices and Event Driven Architecture. It is incredibly easy to look in the rear-view mirror and see the multiple aspects where monoliths

[82] https://en.wikipedia.org/wiki/Serverless_computing
[83] https://en.wikipedia.org/wiki/Chaos_engineering

were failing and SOA was lacking. This same rear-view mirror does not exist yet for microservices.

Microservices have multiple key benefits that seemingly make sense compared to abstractions of other architecture patterns, but in many ways microservices are still an abstraction and are still flawed.

It is important for IT development teams to take into account all of the following:

- Do not build microservices for the sake of building microservices. Unless you have a well laid out system design with your business stakeholders *(such as through Domain Driven Design as described earlier in the paper)*, you will likely make irreversible and expensive mistakes.
- While microservices are intended to be autonomous, be wary of introducing dependencies to microservices. If all microservices are built upon the same software framework or use the same software library and this framework or library changes, your autonomy may vanish quicker than you can deploy the services.
- At the same time you do not want all microservices to be *completely* autonomous either, otherwise you have to reinvent the wheel in writing the same functional code that does the same function over and over and over in all microservices again.
- Be aware that while autonomous microservices as a principle is a somewhat matured and well established concept for knowledgeable engineers by now, cloud is not. Containerized cloud technologies and Docker[84] are in rapid development and continuous change since 2013. This fluctuating environment needs to be addressed as a risk.
- Do not build microservice architecture where microservices are all dependent upon a single database.

[84] https://en.wikipedia.org/wiki/Docker_(software)

4.5. X-Rooms

In previous sections a lot of focus was given to building scalable autonomous microservices, how to decouple said services and how to plan for their communication better through message brokers. A lot of these topics are new for digital government technology stacks that have been around for years. Attempting to make a shift from synchronous tightly coupled communication to asynchronous loosely coupled communication across digital government stack, expecting all administration sectors to rethink and rebuild how their services traditionally integrate can be a high ask.

Estonia has benefited greatly from technology solutions that are overarching across digital government stack: namely Estonian digital identity and X-Road. Estonia uses X-Road for fast interoperability and secure data exchange between large scale information systems and data registries in different administration sectors of the country. While X-Road can be complicated to set up, once it is set up it works and is by far the most trustworthy way how autonomous administration sectors can exchange data and request data from other registries in other administration sectors.

But if we look at concepts tackled in this paper, it is likely evident that the idea of Domain Driven Design, microservices and asynchronous communication can be an impediment when X-Road is involved.

At an abstract level X-Road communication between services works as follows:

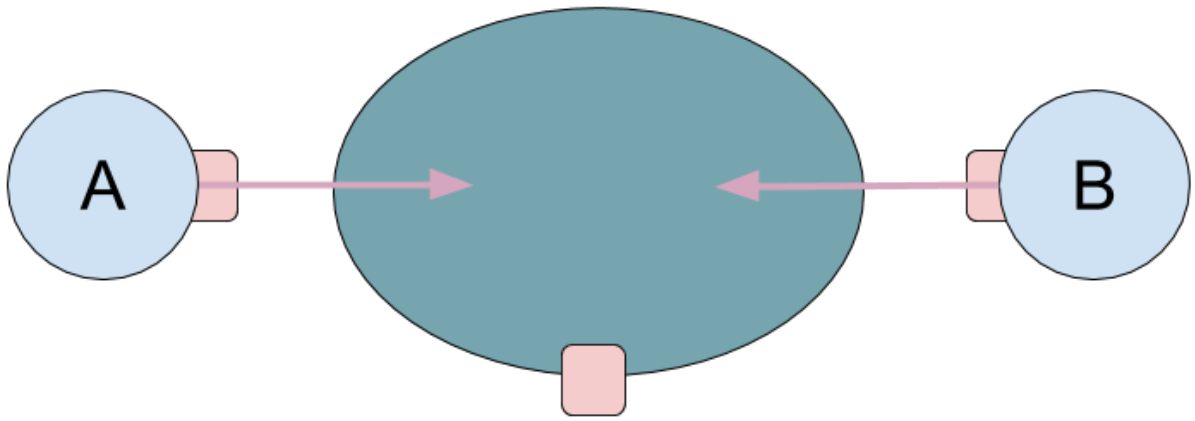
Service A wishes to request data from service B over X-Road. This request is a synchronous request, meaning that service A will be waiting for a response from service B.

As previously described, there are few problems with this method of communication:

- Service A needs to wait until service B responds and can only then proceed. There is a complex alternative in service A multi-threading the request, but this increases internal Service A complexity.
- Service B needs to exist or Service B itself needs to act as a gateway to other dependent services behind it. This, similarly to the previous point, would require a custom solution on service B side.
- Service A and B are tightly coupled, communication between these services is dependent on either side not changing - otherwise integration breaks down.

Each of those three issues can be handled with custom solutions on the side of Service A and Service B, but these are fundamental issues shared by every single consumer and provider on X-Road. As such, it is recommended that the solution itself is provided by X-Road rather than having to build complex solutions on Service A and B side that increase fragmentation of technological architecture.

What is proposed is that X-Road - which is technically a network road infrastructure between different administration sector service endpoints - would also start providing messaging rooms within that infrastructure, called X-Rooms.



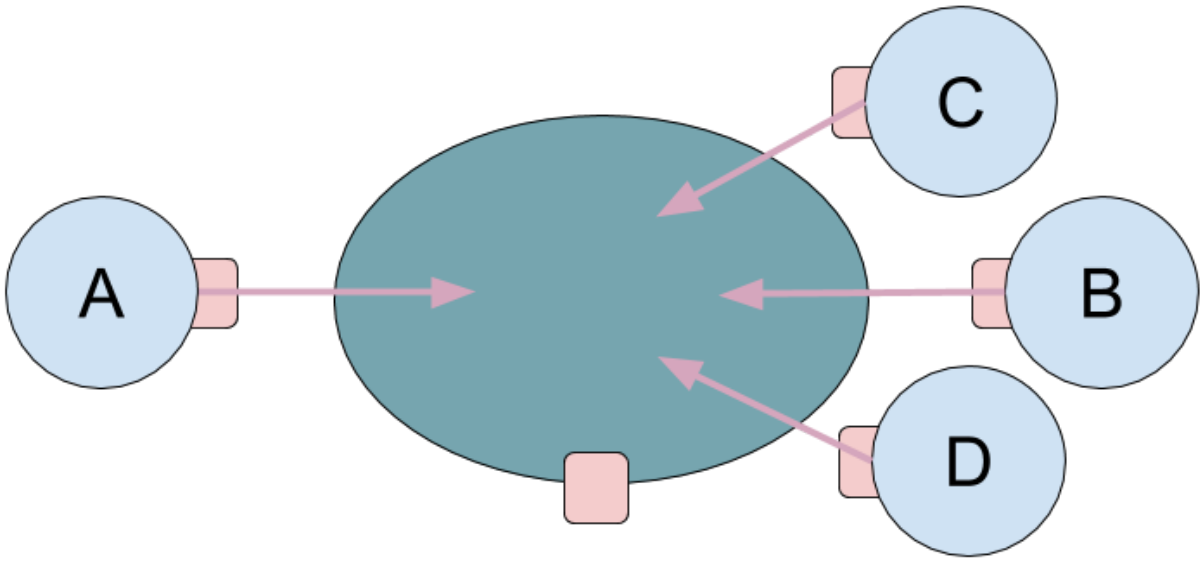
These networked X-Rooms would be built following publish/subscribe messaging model and the earlier described concept of dumb brokers and smart consumers. This means that the services A and B are still responsible components delivering business function just as they always have

been - in X-Road the endpoints have always been where business smart of digital government is implemented.

X-rooms would be messaging rooms that possibly would have their own security server that makes sure that participants in the X-room have the right to be in that messaging room. All other existing features of X-Road would still apply just as they would with direct service communication. There are multiple key benefits over having to implement messaging rooms outside X-Road infrastructure:

- Administration sectors would not have to reinvent the wheel. If you are already using X-Road, starting to use X-Road provided messaging rooms is not more difficult than making requests over X-Road. Note that handling asynchronous requests is still important.
- X-Road is already the most trustworthy data access logging service available, this functionality would be beneficial to message rooms themselves as otherwise custom made message rooms outside X-Road would hide data that is important.
- Most notably citizen data ownership and transparency in the use of data would enhance implementation of GDPR as well.
- Virtual assistants and #bürokratt could also be a user of various message rooms.
- It is a possibility that services that implement X-Road messaging rooms would not have to deal with *andmejälgija* (*data observer*, a concept and a set of tools for administration sectors to enhance transparency in use of citizen data) on the service side at all, as messaging rooms would be able to provide this functionality internally.
- Adoption rate of X-Road is slow and it is difficult to get other countries on board to use X-Road due to multiple complexities. This means that it is important to have a good set of reasons why X-Road should be considered in this day and age. One of the best arguments possible is that X-Road would support complex decoupled and flexible digital government architecture for the next generation. By providing both secure data exchange as well as secure asynchronous message rooms X-Road would be my own personal choice for any large scale system interoperability - even if not in the public sector.

While messaging rooms over X-Road already carry benefits of a more decoupled architecture, other benefits of messaging rooms will also be possible - most notably the multitenancy prospect. This means that it would be possible for multiple services to participate in the same room, reacting to messages and publishing their own messages.



This would mean that service A that requests data from X-Room may not have to care from whom the response comes from, service B, C or D. Service A also does not need to rebuild itself just because participants in X-Rooms change. It is only when their own business flow monitoring shows that processes in service A are not working anymore can engineers start handling the problem.

Potential of standardized X-Rooms goes even beyond the public sector. X-Rooms can be set up for cooperation with the private sector and even by the private sector itself. For example there could be an X-Room that is intended for ride-sharing mini-procurements. If a government service requires transportation from point A to point B, they publish such a request to an X-Room dedicated for ridesharing services from the private sector. Private sector participants are subscribers to that X-Room and once detecting a request they can start their own internal processes and then publish an offer to that same X-Room. Original requesting service can then make an automated decision or ask the user to pick the most convenient option provided by the private sector.

And this service would work independently of how many ridesharing services are integrated with the X-Room.

Here are the next steps and key takeaways for implementing X-Rooms:

- Messaging rooms should become a feature provided by X-Road to both their consumers and publishers.
- Publish/subscribe messaging rooms (dumb messaging rooms/brokers) are recommended in order to keep business logic itself as much away from X-Road solution as possible.
- Having transparent understanding of what services are provided through message rooms and who are participants in the message rooms are important. Thus documentation and transparency are critical to encourage growth of the message rooms.
- Correlation ID should become standardized over X-Road for data logging and tracking purposes. If a request is made over X-Road, it should get assigned a Correlation-ID that will be handled, tracked and potentially forwarded between services that are handling the requests. Correlation ID is important for GDPR as well as gaining visibility over complex processes over distributed architecture. *X-Road already automatically assigns an unique ID to each request/response which is delivered in a specific HTTP header when the REST interface is used. The SOAP interface does not currently forward the ID to consumer nor provider information system.*
- Decoupled messaging rooms would also enhance X-Road viability for mass data analysis and real time reporting. With subscribers to events it would be possible to follow events as they happen, without having to request huge amounts of data at once every day or month.
- Next version of X-Road looks to expand its cloud capabilities of security servers and messaging rooms are inherently best scalable over cloud.

If X-Road will not provide messaging rooms, then these messaging rooms have to be implemented within administration sectors by themselves, leading to technological fragmentation. This also sets heightened expectations to security, as fragmentation and custom

solutions of such message rooms will have to be at least as secure as data exchange is over X-Road.

It would also mean that an important part of data exchange and data interoperability is not part of X-Road, which may lower the adoption rate and benefits gained from using X-Road.

Messages or events

One key criteria that needs to be agreed upon or to be transparent is what type of payload is posted into message rooms: messages or events.

An *event* provides information that a specific event (e.g. a child was born) has happened and the event contains a link/reference to another endpoint/service that provides the full event data. Subscribers that are interested in the full data will send a request to the second endpoint/service – they also need to be authorized to access that endpoint/service. From a security perspective this is a good solution, because the message room will not store any sensitive data, and data locality in a public cloud is not an issue either. For a subscriber this alternative is more complicated since accessing the data requires an additional request to be sent.

Differently, a *message* contains the full event data. All subscribers receive the full data and additional requests are not required. In case this approach is used and sensitive data is published, the access rights of the message room must be managed strictly. In addition, also data locality might become an issue especially in public cloud environments.

From X-Road's point of view how the concept is technically implemented there's not much difference between the alternatives since X-Road is fully payload agnostic. However, different alternatives may have different requirements regarding access rights management, authorization and where + how the data sent to message rooms is stored.

Both options are possible simultaneously, but needs to be carefully considered in system design.

Cross-border potential

With standardization and shared tools that allow governments to share data without having to decouple their architecture directly with that of another country or countries, a new opportunity

emerges. For example, the government of Finland has also adopted X-Road within their digital government stack.

This potentially gives a unique opportunity in discovering together a new way for cross-border data exchange. Due to the standardized nature of message rooms and concept of X-Rooms within X-Road stack, it would be possible for government data exchange to happen over X-Rooms. It is already possible to integrate multiple X-Road networks between one another and using X-Rooms is a natural next step.

Here are the core reasons why cross-border connectivity over X-Rooms would surpass in effectiveness the various alternatives:

- In the same way that organizations would benefit from decoupling within their internal architecture and especially between different organization architectures, cross-border interoperability is an exponentially more difficult problem. No country wants to couple their government systems with cross-border systems any more than they have to.
- Concepts of Domain Driven Design would also apply for cross-border data exchange. Business processes that are required to happen within a country when requesting data from another country can be mapped following similar concepts. Instead of having to send an email and waiting for manual processing by a government official, this could be automated through X-Rooms and freely integrated by governments own back-end services - whatever they may be.

4.6. Fact registries

The concept of fact registries is by far most raw for next generation digital government architecture and should be treated as such, but potential of fact registries can be huge. Fact registries are inspired by the existing solution Integrated Data Infrastructure in New Zealand[85].

IDI is essentially a large research database that holds microdata about people and households. The data is about life events, like education, income, benefits, migration, justice, and health. It comes from government agencies, Stats NZ surveys, and non-government organisations (NGOs). The data is linked together, or integrated, to form the IDI.

Multiple government services and databases report facts about certain events into IDI, which allows to conduct wide scale statistics and research in a convenient way.

What if a similar approach could be used to handle three important issues that are prevalent in public sector digital government architecture?

1. Replacing services technology stack is incredibly expensive both to refactor as well as to start from scratch. The cost of migration of data from one functional database to another requires both the understanding of not only source database, but also functionalities of the source system. This leads to issues of data quality as well as bloated costs of analysis and careful migration planning.

2. Many governments, Estonia included, are relying on functional information systems understanding of truth and the state of the world. This means that citizen address is assumed to be what it is in the functional database of population registry. If something goes wrong in that registry or an error is made, it is difficult to detect and while expensive methods are implemented for most critical databases.

3. Archiving important business data is difficult, as is backing up the most crucial data. It is hard to separate which data from which database is relevant to such purpose in the long term and it can be expensive to archive and back up everything.

Today, the Government of Estonia is backing up data of its most critical information systems into cross-border Data Embassy as a solution to assure digital independence for the citizens. Should

[85] https://www.stats.govt.nz/integrated-data/integrated-data-infrastructure/

something go wrong and original databases become inaccessible, then data embassy is the source of truth for assuring who a specific citizen is.

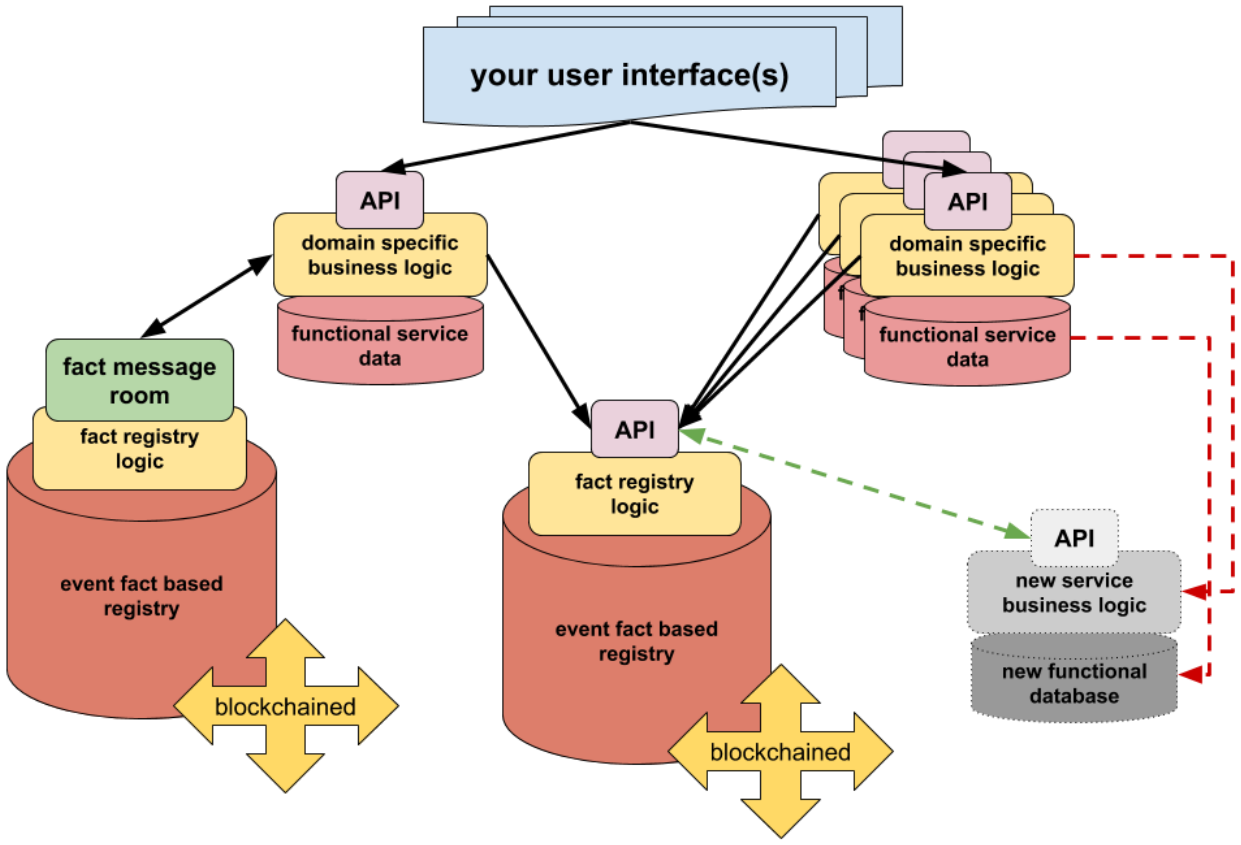
There are multiple issues related to functional database backups into data embassies and the most critical issue is that these backups are difficult to understand without the functional logic of the information system and its integrations around that functional data. While the state's digital continuity is definitely assured to an extent, maybe something better is possible for next generation digital government architecture.

Same is true with long term digital archives. The National Archives of Estonia is responsible for deciding which data is sufficiently important to be kept for future generations, gathering it into the national Digital Archive, detecting and archiving important factual data for future generations. Doing so today involves a lot of manual work, from data dumps to classification and continuous manual changes whenever data sources change.

Today the process of archiving data from a monolith in a long-term understandable way is highly problematic. A *normal* database implements a highly complex data model which is only understandable to IT experts, thus a database dump from this database cannot be expected to be reasonably reusable in 50 years from now. Further, most of the data in the database is in fact not worth preserving for the future. For example, while generic factual information about buildings (like construction plans and ownership) is certainly valuable for future generations, then data about the many small steps in the workflow of issuing a building or renovation permit is only needed for a relatively short time period in handful of years.

Nowadays the process of classifying which data is actually relevant for archiving, exporting it, enriching with contextual metadata and transferring to the digital archive is largely manual and can take months to carry out.

But what if we do not store single source of truth within a functional database of data registry information system anymore?



The concept of fact registry, if implemented well, would mean two things:

- If the government wants to start developing a new service, they do not have to worry about data migration from the old service anymore. They do not even need an understanding of how exactly the previous information system worked.
- New information systems will be built independently of whatever was there before and multiple information systems (even within the same domain) could be in development at the same time. Data migration becomes an non-issue - in a way. This means that the fear of breaking old systems and old integrations is much less of a problem and you also do not need Big Bang type of service releases, as you can actually run multiple domain registries at the same time (such as two population registries).

To make this happen, we need fact registries. Similarly to the core concept of IDI, a fact registry stores government events. You can have multiple fact registries. For example, population fact registry would store births and new personal codes that have been assigned to citizens, facts

about name changes, address changes and more. All of those facts would be stored as events in the fact registry that can be monitored and requested by authorized parties.

Services own databases will only act as functional databases for the service itself and have no need to store knowledge or awareness beyond the functional scope of the system. Everything that factually matters for the government would be stored and signed in the fact registry.

For example:

1. A birth is registered by the hospital.
2. Population registry detects the birth either by being subscribed to a message room that handles this information, or an authenticated API call was made from the hospital.
3. Population registry starts the internal process in registering the birth. This can involve notifying different message rooms, querying data as well as registering the personal identity code:
    a. …
    b. A digital fact document is made by population registry and signed by private key of population registry information system. This document is then submitted to the birth related fact registry.
    c. Fact registry authenticates the signature of the new fact to assure it is from the right source.
    d. Fact registry stores the new fact and adds it to internal blockchain.
    e. Fact registry publishes new event to related message rooms and other systems can react to this new fact (possibly integrating some of its data within their own internal databases).
    f. …
4. Population registry stores whatever it deems necessary within its own functional database.

Another flow that is important to go over is related to the situation where the government decides it is time to start developing a new population registry. With the concept of fact

registries, classic data migration from previous information system to a new one is not necessary anymore:

1. Business stakeholders agree about the feature scope of new population registry service. Preferably using Domain Driven Design and involving technical stakeholders as necessary.
2. Engineering team has access to metadata and/or anonymized data from population data related fact registry.
3. Engineering team develops a new information system. This includes functional logic that understands facts that are stored in fact registry as well as ability to authenticate signatures of fact registry.
4. Entire functionality of the new population registry can be finished without any data migration required from existing population registry.
5. Live testing is possible with new population registry as population registry can listen to facts of population fact registry.
6. New population registry can stream over all relevant historical facts of population fact registry and build its own functional database as required by the new information system.
7. Two population registries are able to work side by side, including a new population registry submitting new facts to the population registry.
8. Once everything seems to be working as expected, old population registry can be removed in entirety.

The core benefit of this approach is decoupling of software development requirement of having to understand previous information systems and its process flow. As long as business stakeholders have understanding (and, if complex enough, related documentation) of business flows, then it is possible to develop new government services that replace old services without care of complexity of the old service itself. You only care about understanding standardized (and versioned) fact documents in fact registries.

Last but not least, fact registries are the only thing that is actually important to be backed up in data embassies and archived in national digital archives. Information systems that are able to understand and communicate with fact registries can be entirely open sourced and available in any public code repository. If a problem happens and original services are not available anymore, it would be possible to request fact registry data access from data embassy and build new instance of the service based on open source code and thus rebuild service functionality independently from physical territory.

There are other issues regarding fact registries that require further research:

- If fact registry data integrity is assured by blockchain, what happens if the government has a lawful requirement to erase certain set of data? Would in that case fact registry store data directly in blockchain and the whole chain is re-generated - which may defeat the purpose - or are only fingerprints stored in blockchain and fact database itself has to remain without blockchain? What are the risks regarding either?
- There is an opportunity to also implement linked data[86] concepts to connect various data sets between one another across fact registries.

[86] https://en.wikipedia.org/wiki/Linked_data

4.7. Key takeaways

Transformation from monolithic architecture to event driven microservice architecture can be a mammoth task for even the most experienced IT development team. It is not the goal of this paper to lay out in black and white that monoliths are evil and the only true way for building services is to do so with microservices. But it is important to make the right choice at the right time.

But it is also important to understand that monolithic architecture in scale of a government is a huge risk. Estonia is still struggling with large monolithic databases and tightly coupled business services that were built up to twenty years ago.

X-Road has enabled a far more flexible distributed government service architecture in Estonia, even as it connects monoliths between one another. This too is a risk that has to be kept in mind and managed well.

But in order to get to microservice architecture, it is important to establish a well functioning cooperation between business stakeholders and IT development team. It is in designing and developing microservices especially where problems of this cooperation can become a serious impediment. Domain Driven Design is critical and engineers should not write a single line of code before business design is clearly laid out.

It is also important for the IT development team to experiment with microservices on a smaller scale before tackling a larger project. A key recommendation is to do so at the same time as investigating options of cloud platforms, if cloud competences in the IT development team are also lacking. Microservices and cloud make for ideal partners.

In terms of government, Apache Kafka is recommended to be tried out as a messaging platform. While complex to set up and get running properly, its set of features matches well with expectations of government technologies, especially if message rooms are shared between multiple administration sectors.

It is still important to keep in mind who is the master owner of core data. While distributed architecture allows for replication of data for functional purpose, it is important for government to still know where the single source of truth is, if required.

Last, but not least, it is a strong recommendation for X-Road development roadmap to consider the options of X-Road enabled messaging rooms as a feature for government technology stack.

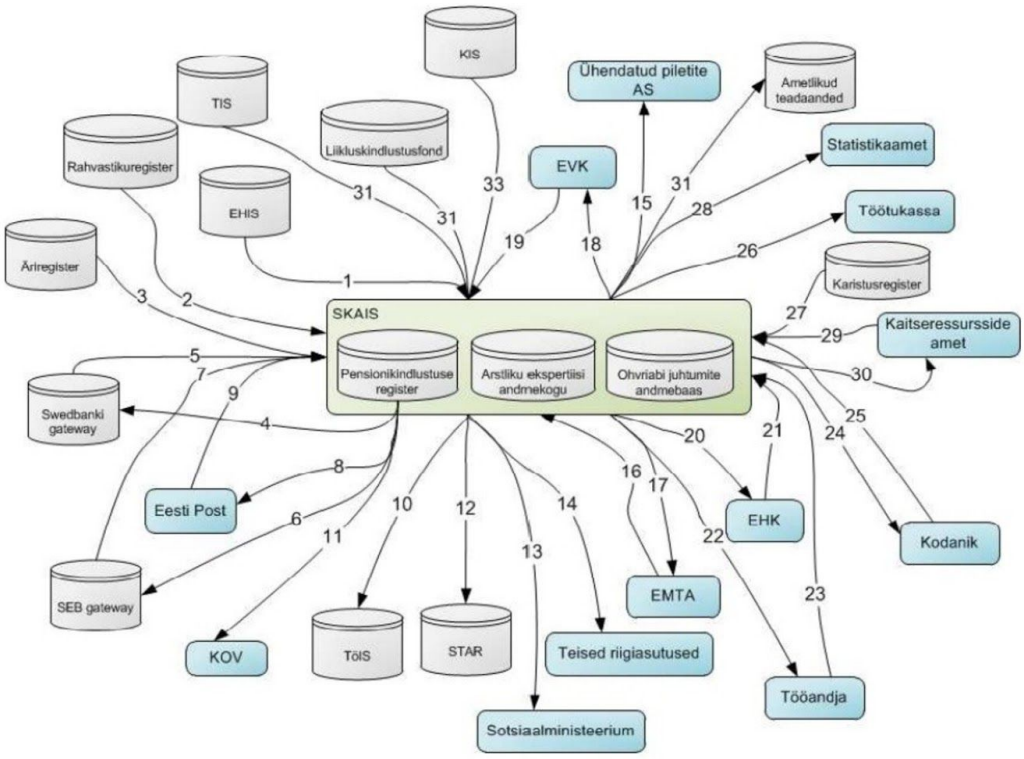
5. Becoming a post-agile business stakeholder

While this paper primarily focuses on government technology architecture and establishing a common language and understanding between business and technical stakeholders in aforementioned topics, there are multiple issues that are more in control of business stakeholders that also need addressing.

Software development in digital government is primarily conducted using waterfall[87] model. This means that from idea and business requirement to analysis and actual development and also live implementation is a step by step process, where previous step is largely finished before any proceeding steps. This front-loads an immense amount of expectations and understanding about the solution before a single line of code is written and it often carries with it a huge amount of risks.

One of the better known examples of this is *SKAIS 2* project, which handles social welfare related payments and related registries in Estonia. In 2012 the high level architecture snapshot of the existing system was the following:

[87] https://en.wikipedia.org/wiki/Waterfall_model



It is clear already from such a high-level picture that building this system using waterfall model of development can be very complicated and these complications became evident quickly. Development of the project had a 5m EUR budget used for before the project was temporarily halted due to complications. Today, 7 years later, the total cost is estimated to be 23m EUR and the project is still ongoing with further complications likely.

Public sector software procurement and development audit[88] have revealed that the main problems of SKAIS 2 developments were the following:

- Development was slower than changing business requirements, which had to be implemented retroactively to previous developments;
- Project in whole and in parts had unclear ownership and stakeholders;
- Communication between the administration sector and partnering development team was hectic and often inconsistent and conflicting.

88 https://www.riigikontroll.ee/DesktopModules/DigiDetail/FileDownloader.aspx?AuditId=2488&FileId=14400

CHAOS Report 2015[89] published results that only up to 6% of large scale projects actually succeed and stay within budget. It is common for public sector projects to continuously gamble with such odds. Reasons for this are due to big projects being more newsworthy and seemingly more impactful in both scale and scope. Large projects also seem more safe in their lifespan at first, as delivery of such projects are not expected to be immediate and thus are less stressful - at least until delivery becomes late. In other words, projects that are planned to take years seem like they will be delivered in high quality simply *because* they take years - which is a fallacy. Only once the actual details and needs and complexity emerges during the execution of the large project it becomes clear that the project will not deliver how it was expected.

It has also not helped that the majority of software launches attempt to go live with a Big Bang, meaning that all at once. While some may argue that there are situations where this is unavoidable, this sets up immense and often impossible expectations to analysts and project management before even a single line of code is written.

This is not a problem unique to Estonian digital government and in the last decade, multiple approaches to software development have been implemented to tackle this. For one: agile development. In Estonia, agile development is becoming more widely adopted in the public sector through tight cooperation with private sector enthusiasts and consultants: there are Scrum teams, sprints, demos and retros.

But over the years the world *'agile'* has become tainted for agility has been enforced from the ground up, starting from engineering teams instead of starting from business process and management. This has meant that the Agile process - despite being called Agile - has not worked and has caused more frustrations. It can be misguided to hope that just because your IT development team has adopted Agile development principles that the service development is actually agile.

There are also other issues that need addressing to avoid similar mistakes in the future:

- Information systems are complex to develop and refactor due to the changing nature of the technology landscape. Popularity of programming languages and database technologies change in popularity, which impacts the amount of engineers and their

[89] https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf

interest to develop those technologies. TIOBE index[90] should be used as an indicator when selecting core programming languages.

- Government e-services technology stack today is largely not cloud-ready. Many services are relying on bare-metal solutions, and a strong majority are running as virtual machines. Cloud adoption rate is slow both due to lack of competence and training, but also due to the architecture of services themselves.
- In a majority of situations it is easier to develop the system from ground zero instead of refactoring or continuing development of existing systems.
- Communication between IT development centres and administration sectors is often flawed. Responsible business stakeholder from administration sector is often not the actual owner and the domain expert of the project and instead is a person responsible for communication. This often means that if the development team requires clarity regarding business features, they cannot get this directly from the business domain expert.
- Many business stakeholders, including domain experts in administration sectors are actually managing multiple projects and developments at the same time.
- Public sector projects - even large ones - are often lacking a technical stakeholder, such as a solutions architect. This means that whether code quality is up to the standard, and whether functional and non-functional requirements are taken into account is left to the development partner to self-audit.

Financial impact is often inadequately measured for the development of new systems. Many projects are funded that end up more expensive for the taxpayer: investment and maintenance financial cost over five years costs more than continuing to do the same job manually.

Many of the concepts in this paper are unable to be implemented in digital government unless the business stakeholders are aware of the terms and expectations. Without business stakeholder involvement and drive and expertise, many of these concepts will never happen.

[90] https://www.tiobe.com/tiobe-index/

5.1. Being post-agile

There's a saying that if you tell engineers to do Agile[91] development, they will do Agile and if you tell engineers to do Waterfall[92] development, they will do Waterfall - but if you look at it closely, they will actually do the same thing - this is because it is not an engineering problem to solve. Implementing Agile development in not just digital government, but the whole software development sector has been difficult due to that reason.

Differently from common criticism directed towards it, especially by Agile development enthusiasts, Waterfall is *not* a bad model for software development. But it is important to be careful when using the Waterfall development model. At a high level there are three working models to software development:

- Waterfall this works, when you have a thorough understanding of the problem and the solution, which is rarely the case ;
- Agile model works, when you have a thorough understanding of the problem, but do not yet know the solution;
- Lean[93] model works, when you do not yet understand the problem.

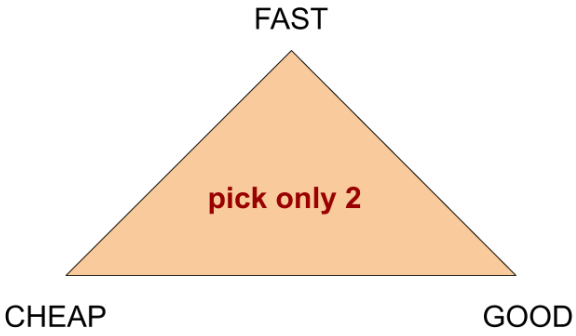
If you are responsible for refactoring an old service or starting to build a new one, the first question to ask yourself is whether you understand the business problem that the service is intended to solve? And if you understand the problem, do you also know the solution?

The next question to ask is: *are you sure?* Have you asked the right people? Are you taking the right data into account in making this decision?

It is a common pattern in software development where a project is "solved" from a single perspective - often business requirements - and then this solution is then extrapolated across all stacks throughout: from procurements to scheduling to budget planning to technology.

[91] https://en.wikipedia.org/wiki/Agile_software_development
[92] https://en.wikipedia.org/wiki/Waterfall_model
[93] https://en.wikipedia.org/wiki/Lean_software_development

Reality is that the assumed solution is never actually the *true* solution and what the service actually ends up being. Thus all aspects of scheduling to budget to technology planning are flawed or incomplete. If you choose the Waterfall method of development in this situation then you should get ready to cross red lines of budgets and deadlines or have to make compromises in quality. While this is a popular model shown and explained thoroughly elsewhere, without accepting this you will continue to always struggle with your expectations:



If you are a business stakeholder, you are responsible for clarifying *what* is needed. It is the responsibility of engineering teams *how* that is achieved and the compromise (both financial and in quality of technology) is in between.

Here are some important concepts of post-agile mindset to follow:

- If you are planning a new development for a service, you have to have a technical stakeholder involved. This does not mean that you need an engineer at every meeting, but the moment your service is about the development of new features to your system, new integrations within the government or cross-border, you need the opinion and feedback of the technical stakeholder. This needs to happen before scheduling, budgeting and planning.
- During development you have to be involved with the engineering team. It is not good enough for the end result if you only casually take a look at how things are progressing. As a business stakeholder you have certain expertise and domain knowledge that the team greatly benefits from and desires even if they are unable to communicate it well. This is important for the success of the end result.

- Do not assign communication people between business stakeholders and development teams and if such exist, find ways to reorganize. Teams should be able to ask the responsible business owner and domain expert directly - and hopefully without having to schedule meetings.

- Do not underestimate risks raised by the technical stakeholders and remember that software engineers are often optimists under pressure and partners may be financially biased.

- As mentioned earlier, business stakeholder is responsible for *what*. Technical stakeholder is responsible for *how*. The scope of *what* and quality of *how* depends on the *fast/cheap/good* triangle. Make sure both sides understand the compromises that are being made and on what scale. Same applies to your partners.

- If you are using a partner company in software development, make sure that you have your own technical stakeholder involved in the project, especially for larger projects. While expertise of partners *(especially from the highly competitive private sector)* is often unmatched in the public sector, it is important that a technical four-eye[94] principle from two different parties is used. Otherwise it is difficult to validate that the agreed principles and expectations are met.

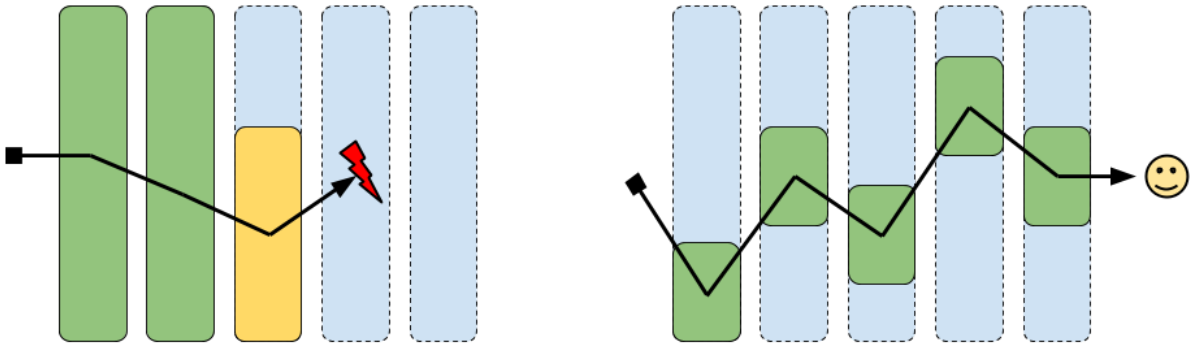
- Minimize the risks of having to handle multiple projects at the same time as a business stakeholder. Successful development teams are always singularly focused on one project at the same time, the same *(or at least close)* should be expected from business stakeholders.

[94] https://whatis.techtarget.com/definition/four-eyes-principle

5.2. From vertical to horizontal

One of the main ways how large scale projects get into trouble is that they are planned and executed in a vertical manner. This is especially evident in various roadmaps for projects which all end with a single *“live”* stage at the very end. If you are planning a project roadmap and it involves today even a single live stage, this can be a symptom of an unintended Waterfall development and it is important to start considering alternatives. Single-live roadmaps in planning is a little shortsighted as it assumes that you know your service will work as expected once it is finished in whole and none of the business expectations will change in what is often years of development before life.

It is important to make a transition in roadmap planning like the following:



In the picture on the left, large stacks of features are being planned and developed before live actually happens. While it is possible to test business flow up to a certain point, you cannot test it throughout as every previous step needs to be more or less complete before next steps can be developed. If your single well-thought-out stack cannot provide business value within its own scope, then you need to go from vertical to horizontal as quickly as possible.

There are multiple projects in the public sector developed in this manner. They are often titled with “phases” and “steps” and “parts” of development and frequently even funded as such - while actual business value will not happen once all stacks have been developed.

On the picture to the right, none of the pillars are complete, yet the entire flow goes through all pillars and can be tested by an actual end-user.

To get to the result of horizontal development, these things must happen:

- Project roadmap should involve either multiple “live” steps, or actually none at all. The latter is an ideal as you could plan it in a way where every single step actually ends with their own “live”.
- Feature development should, from very early on, be full-stack end-to-end. Meaning that back-end logic, front-end logic, permissions, privileges and everything else involved should work from end-to-end - even though you know that the project will need hundreds of more features over time.
- This way of thinking will also enable better Agile and Lean software development as feature mapping and testing becomes more natural and your project roadmap will match better with software development roadmap as a result.
- Not every step needs to be perfect in the horizontal feature flow. As long as the flow itself works, you can analyze it, detect corner cases that you might not have encountered and you are able to test it with actual users to see if it matches their needs.

5.3. Build only what you know best

… and use as a service everything that others know better.

This is an important takeaway from the Dot-com bubble bursting mentioned earlier, as the companies that came out of the crisis successfully learned how to use the services of other companies instead of attempting to reinvent the wheel.

As was described in the background section earlier, re-use of services in the public sector is low and quite frequently reinvention of wheel happens almost naturally. This happens for two reasons:

- It is difficult to know what other administration sectors have
- It is difficult to reuse what is in use in other administration sectors

In itself this is a problem that feeds itself. In order to break the cycle, software development in the public sector needs to become more open and more transparent and needs to comply with commonly agreed with principles and practices so that systems that work in autonomous silos become compatible between one another.

In itself this is a larger issue to tackle in the scope of a single paper, but this is what needs to happen:

- Software and code and solutions that have been developed and are paid for by tax-payer funding needs to be open. There is no excuse to restrict such code from being open source. This can lead to complex services to become new businesses and result in the government not having to develop and maintain a service. This can also increase standardization and re-use, especially if codebase becomes healthy and more widely adopted. *In case of Estonia, share code and re-use code at koodivaramu.eesti.ee. Estonian public sector code repository is still in development and is likely to change without notice.*
- Software development also frequently involves non-code components. Binaries and containers should also be shared or made open, if at all possible, such as docker

containers and artifacts. *In the case of Estonia, such an open artifactory does not exist yet, but it is planned that it will.*

- Closed code gives a false sense of security, called *security through obscurity*[95]. Reinventing the wheel by building existing solutions from scratch is also a security risk.
- It is required, when starting a new development, to make sure that another administration sector already doesn't have something you could already use instead. It is important to ask them to be sure.
- It is also recommended to find another partner in another administration sector who are interested in the same solution - as this approach inherently already enables more reuse potential in the service.
- It is important to make existing services or newly developed services more transparent to other administration sectors. Thus it is important to maintain an up to date service registry of not just technology, but also the business services and their relations. *In the case of Estonia, make sure that RIHA is widely used and gives enough transparency about the services and its components so that other administration sectors may benefit.*
- Make sure your service documentation - both business and technical - are up to date. Best you can do is to automate such documentation, especially technical documentation.
- Make sure you use commonly agreed standards of software development. Standardization has a major impact in re-use, but it also needs to be handled carefully as over-standardization can impede innovation. *In the case of Estonia, use the Architecture Council's agreed principles and cross-functional requirements in government.*

[95] https://en.wikipedia.org/wiki/Security_through_obscurity

5.4. Responsible investment planning

The most important role of technology is to automate the routines in our everyday lives so that we can save time for what is really important: friends, family and outside the box problem solving that computers cannot automate.

IT developments in the public sector are vast not just in Estonia, as in the world digitization has become a trillion-dollar-challenge for everybody. In Estonia, the public sector is far more uniformly digital than the private sector is, where there is a notable digital divide. This can be a healthy indicator, but also an unhealthy one: it can mean that the public sector is building digital systems just because we expect (and are expected) to.

Estonian government has in the past two decades made sure that IT is a strategic focus. But public sector IT is not just for the sake of IT. The reason why technology is especially important for Estonia, is because of our low and aging population. If technology cannot assist us in saving work hours of our population, then Estonia will be lagging behind in economy and wellbeing sooner than we might fear.

As a responsible business stakeholder it is your duty to make sure that when building new services you are not actually increasing costs for the taxpayer. If it is cheaper to do something manually, than developing a system for it, continue doing so manually and until you have a testable hypothesis for a cheaper solution (technological or otherwise), those investments could be more impactful elsewhere.

It can be difficult to fight peer pressure or political pressure that says *“get this project done”*, but if the project does not actually save money or save lives - directly or indirectly - why does it deserve your or anyone else's attention?

Here are the items that should be kept in mind:

- Make sure you understand the true cost of the service - in whatever situation it is today - as it is. Measure it, if possible: the amount of materials, man hours, speed, direct and indirect costs. Worst case scenario is that you should order analysis on the existing state so you have something tangible to compare against.

- Plan the new service keeping in mind how it should optimize and enhance the existing service. If it is a new service, that too needs to streamline or simplify something that exists.
- Plan out a whole lifecycle of the service. The assumed development, it's estimated user growth over time, further development cycles and planned *end of life* of the service. It is important to never consider any service as indefinite.
- Set a hypothesis that can be believed in: what is the impact of the new service? How many work hours does it save over a year? What is the financial impact of the saved time? Does it enhance citizen experience directly or indirectly? Can you measure it?
- Does something like this already exist in the market that could be used - even if it isn't as perfect as you would like? Often it can be cheaper to use a less complete solution than trying to develop an ideal solution by yourself.
- What is the estimated cost of developing the service functionality and the time and effort needed in getting it deployed live.
- What is the estimated cost over a certain amount of years *(for example, 5 years)* of maintaining and supporting this service?

And keeping all of the above in mind, is then the total cost cheaper or more expensive than continuing as-is? If the answer is *no*, is it justified that digital government will - as a result - cost more for the tax-payer than it would without?

5.5. Monitor everything

Last but not least, it is important to make sure that you monitor and automate monitoring of service use thoroughly. In 2020 it is not good enough anymore to pay for analysis or send out surveys to ask what your users think. It is important to build services from the ground up in a way that they can be monitored, so you can make the right decisions in how to develop the service further.

While choice of software architecture can vary per service, basic monitoring is possible regardless of technology chosen. As long as you make sure that your development teams implement business activity logging, you are able to lay groundwork for basic monitoring for your service.

In the most minimal way this means to automate when a service process starts and when it ends. This allows you to make already the basic business decisions, for example seeing if the changes made to the service have made the service more optimized and quicker for the end user or not.

This is especially important for digital governments as the length of a service process delivery has a direct relation to throughput of government. If you focus your decisions on how quickly something can be completed in your service, you also have a direct impact on how many service processes you can run daily or monthly.

While A/B testing[96] is not a highly developed practice today in terms of virtual assistants, it is still a great methodology for introducing change in front-end environments that end-user has to interact with, especially government officials using information systems and different proceedings systems.

It is also possible to understand if your end-user is happy with the service without actually asking them, simply by reviewing the complexity of their behavior when using the service.

[96] https://en.wikipedia.org/wiki/A/B_testing

5.6. Key takeaways

While business stakeholder decisions and responsibilities are not directly related to issues of software architecture, it would be irresponsible to not handle at least the basic shifts in mindset that have to be made for the benefit of next long term sustainability of digital government technology stack.

It is very important to not consider Waterfall, Agile and Lean in a directly contrasting manner. The right methodology should be chosen from the perspective of the problem that needs to be solved. There is no silver bullet.

It is also important to start thinking about horizontal and service oriented delivery of government services. The key goal is to make sure that your service - even if it is brand new - works from end to end starting from the first development or test deploy.

Reuse of services that are already out there is critical to optimizing costs and not reinventing the wheel. Building something from scratch is incredibly expensive and if you make this decision, you better also make sure that you have a good hypothesis that you are able to solve it better than others.

It is also incredibly important for government business stakeholders to understand that services should not make the digital government more expensive for the tax-payer. Automated services should be optimized for the goal of making processes quicker and cheaper and of better quality.

And it is important to monitor everything in an automated manner from the very beginning and use this data to make better decisions for further development of services.

6. Conclusions

Remember the story.

As was written before, the most important role of technology is to automate the routines of our everyday lives so that we can focus on what is really important. Technology does not exist for the sake of technology. It has been a goal of this paper to focus on how we could assure that it is not technology nor ever will be technology that becomes an impediment for the success of the country and welfare of our citizens - in Estonia as well as anywhere.

Digital government is huge. It consists of thousands of technical components and even more dependencies that have been built in the last few decades. It is increasingly more important to maintain and handle what already exists in digital government rather than what else can be built on top of it. Everything that we add to the digital government technology stack quickly becomes something that engineers and partners have to maintain. Digital government is constantly growing, offering more services and more components and there is not a single administration sector where anything other than that is true.

Agile development and cooperation between business stakeholders and technical stakeholders has been lacking and needs addressing to achieve better cooperation. Realization of the impact of Conway's Law and utilizing Domain Driven Design will bring engineers and process owners closer together and end up with solutions that are understood by both parties the same way. Better planning will allow the government to test services sooner and also fail fast in a safer way at times. Failing should not be feared as the impact of failing fast is much smaller than failing *big* - when it is too late.

Business Process Modelling and workflow tools will allow for more decoupling of functional services from process services and also open up new opportunities for re-use across administration sectors and between.

Estonian government web portal and digital services from administration sectors have been an immense success, but the expectations of citizens are changing. The future involves virtual assistants that help navigate complex bureaucracy of government are unavoidable - it's only a

matter of time. Next generation citizen experience will rely on them, but it is also important to make sure that there are fallback options.

X-Road has been an immense success for the digital government of Estonia, but it is facing challenges of the expectations of a more decoupled and more data-analysis driven world. If X-Road becomes easier to trial and test, easier to use regardless of the amount of services you have and more open for asynchronous and message-room based solutions, such as X-Rooms, then X-Road will not only continue to be a foundational layer of digital government of Estonia, but possibly bring it to the next level.

It also has to be understood that there is no silver bullet that solves everything for the next generation. As Thomas Edison has said, vision without execution is hallucination - thus concepts need to be piloted, hypothesis tested and heated discussions held to take us further. It is too expensive to do anything different.

It is also important to understand that the high level proposed solutions in this paper may have a difficult time working in parts, without the whole concept in mind. It will be difficult to adopt new architecture patterns without the involvement of business stakeholders just as it will be difficult to adopt new laws and regulations in technical services quickly without the involvement of engineers. While it is difficult to expect harmony between these two layers, there are numerous examples from both large and small scale organisations where tight cooperation and understanding between those two layers is fundamental to success.

This means that business owners, analysts, engineers and software enthusiasts are encouraged to pick up some of those proposals herein and try them out on a small scale as proof-of-concepts and thus build a new kind of awareness that will be invaluable in solidifying many of those proposed solutions in the future.

If I would round up all of the proposed solutions in this paper, then it is difficult to avoid any other conclusion that our duty - as business stakeholders and technical stakeholders - is to do everything that we can to extend the lifespan of our future digital government services even a few years more compared to services today.

While it may seem conservative at first, it is important to realize that the goal is to reduce the impact of cascading maintenance across all of the services that almost act as a compound

interest[97] on technical debt. Estonia will not have the engineers, nor even partners, to maintain digital government if we do any differently and continue to only focus on short-term goals.

When working in the public sector, tackling huge projects with vast budgets and battling tight schedules, it is often difficult to keep in mind that the long term benefit is often a bigger value for the citizen. Short term value - delivering project quickly, just getting it out there, cutting corners - can be misguided. While it is true that the majority of public servants and especially engineers working for the public sector frequently switch jobs long before problematic software development rears its ugly head in digital government, we must not compromise. We need to understand that it is something for *us* that we are building and something for *us* that we want to be healthy.

As future owners of companies, future parents, future receivers of health benefits, future pensioners wishing to travel the world, it is us who will be using those services that we are building today. We are paying for their development and maintenance with our taxes. As civil servants, it is our duty to make sure our technology stack can live longer - which can save millions of euros of investments. This can provide opportunities to build better services or perhaps invest the saved money into even better education systems where engineers of tomorrow are coming from.

It is our responsibility to assure that these engineers are going to find a healthier stack of digital government to evolve and take us further.

[97] https://en.wikipedia.org/wiki/Compound_interest

Possible Research Topics

- Readiness of modern day citizens for virtual-assistants-enabled government services. What does a citizen expect? What does e-resident[98] expect?
- Decoupling three loosely-coupled-required layers: #bürokratt understanding of language, detection of user intent and communication with government background. Is it possible?
- Readiness of Apple, Google, Amazon, Microsoft and other virtual assistant providers to support concepts of #bürokratt virtual assistant.
- Feasability of the concept of X-Rooms to be supported by X-Road technology stack.
- Feasability of business process modelling tools, such as Camunda and Flowable, as a solution to orchestrate large scale government services across multiple administration sectors.
- Feasability of concept of fact registries in digital government as a way for loosely coupling critical data.
- Feasability of use of blockchain in fact registries in digital government in the era of GDRP.

[98] https://en.wikipedia.org/wiki/E-Residency_of_Estonia