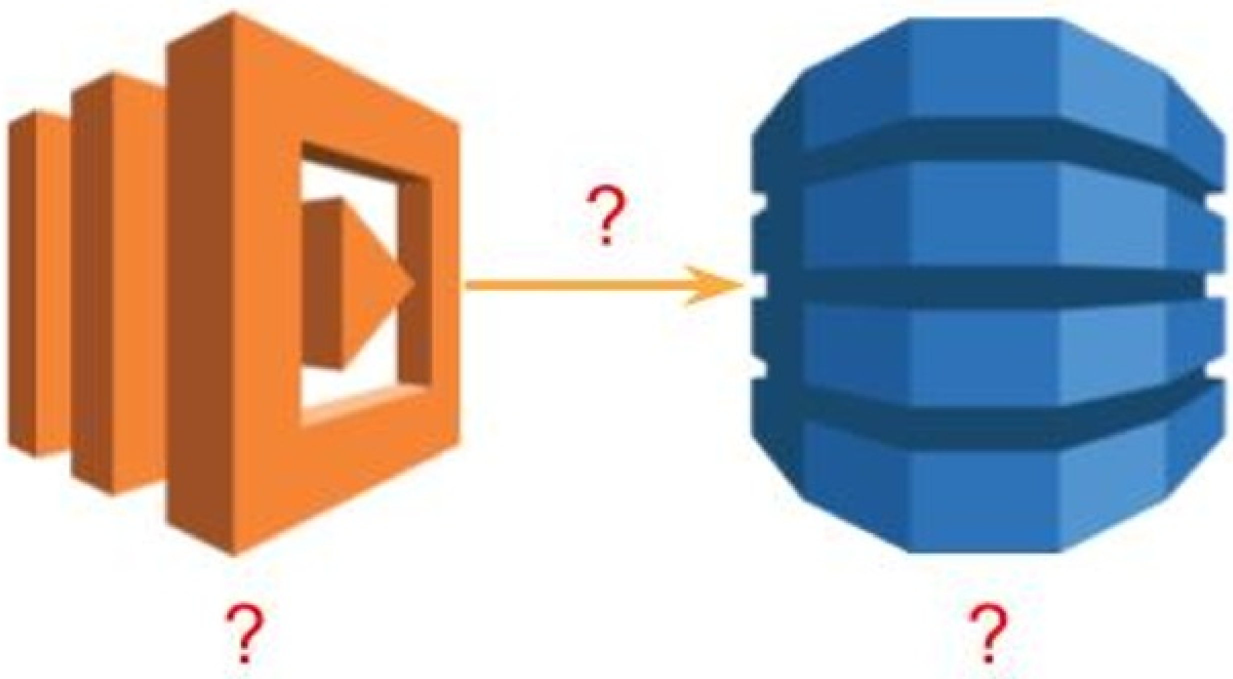
Serverless Architecture Language

hackernoon.com/documenting-serverless-architectures-69bf63fc67b0

What's the Problem?

With Serverless computing platforms we, software developers, can at last put aside all irrelevant technicalities and start delivering what we are payed for — business features. It is, indeed, a potentially tectonic shift in the software industry (S. Wardley suggests some important strategic insights on the subject).

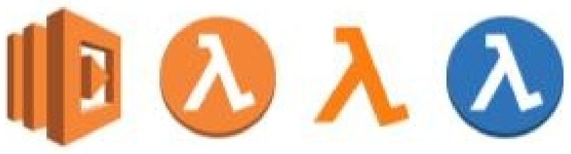
Why the fuss about serverless?
*To explain this, I'm going to have to recap on some old work with a particular focus on co-evolution.*hackernoon.com



What does this picture mean and why we call it "Architecture"?

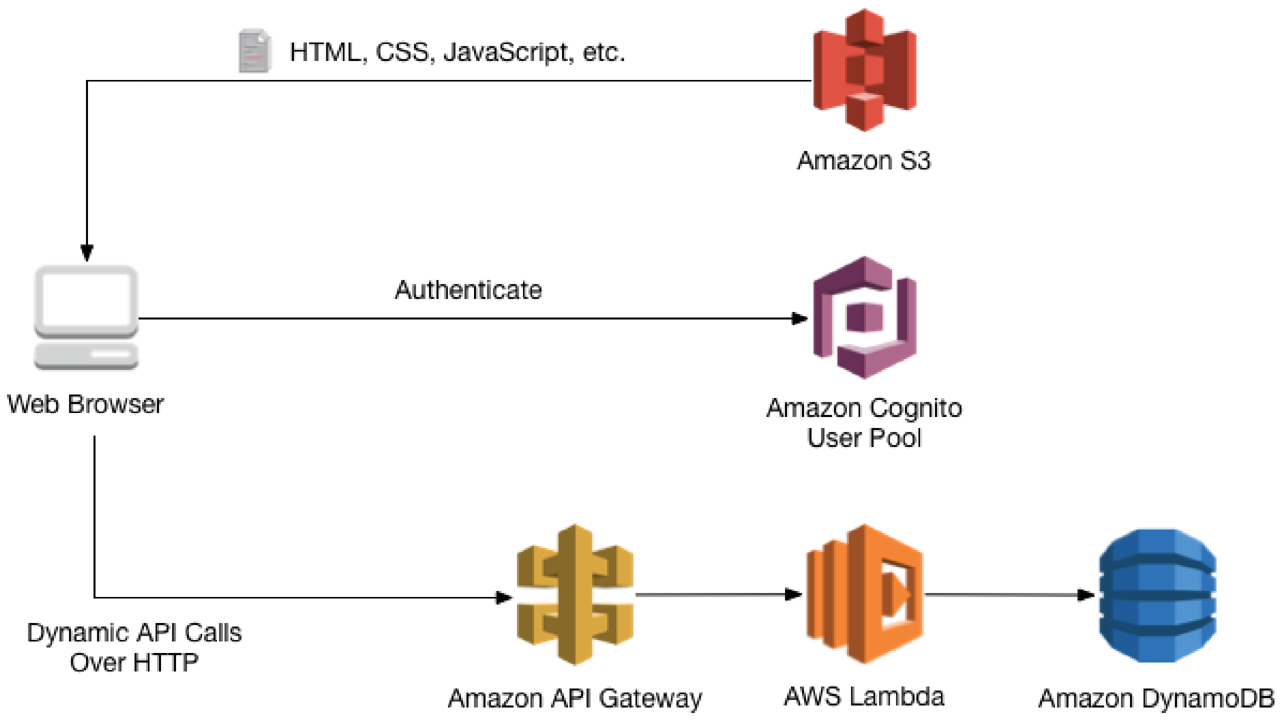
But look at the picture above. This is how we typically document our Serverless architectures. It is impossible to assign precise and unambiguous meaning to neither the individual elements, nor the diagram as a whole. This diagram is too ambiguous.

Which one to use and when?



Typically, it is hard to decide which icon variation to choose in what circumstances. Do they all mean the same and it's just a matter of taste? Or they all are different and the choice depends on context. How to pick up the right context then?

Here is a diagram from the official AWS Serverless Web Application Workshop:



AWS Serverless Web Application Workshop Architecture

This diagram seems to tell us that there are five AWS services, which in principle could talk to each other and that Web Browser could talk to some of them. It conveys very little, however, about how the application is built and what it does.

This diagram is not very much useful for conducting architectural analysis of latency, throughput, security, availability, cost, or productivity. There is no way to separate core business functionality from generic or auxiliary components. There is no way to identify duplication scattered across multiple applications. There is no way to compare multiple options and to justify choosing one over another.

Furthermore, this diagramming style does not scale beyond the scope of very simple applications. When the application size grows, intellectual control over its structure is usually lost. At Serverless Meetups, it's not uncommon to hear worrisome stories about startups, which produced ~500 Lambda Functions within 6 months of development; and now, nobody has a clue what's going on and how to maintain them. Sometimes it makes an impression that Serverless computing is just a new fancy way to mess ourselves up beyond all recognition.

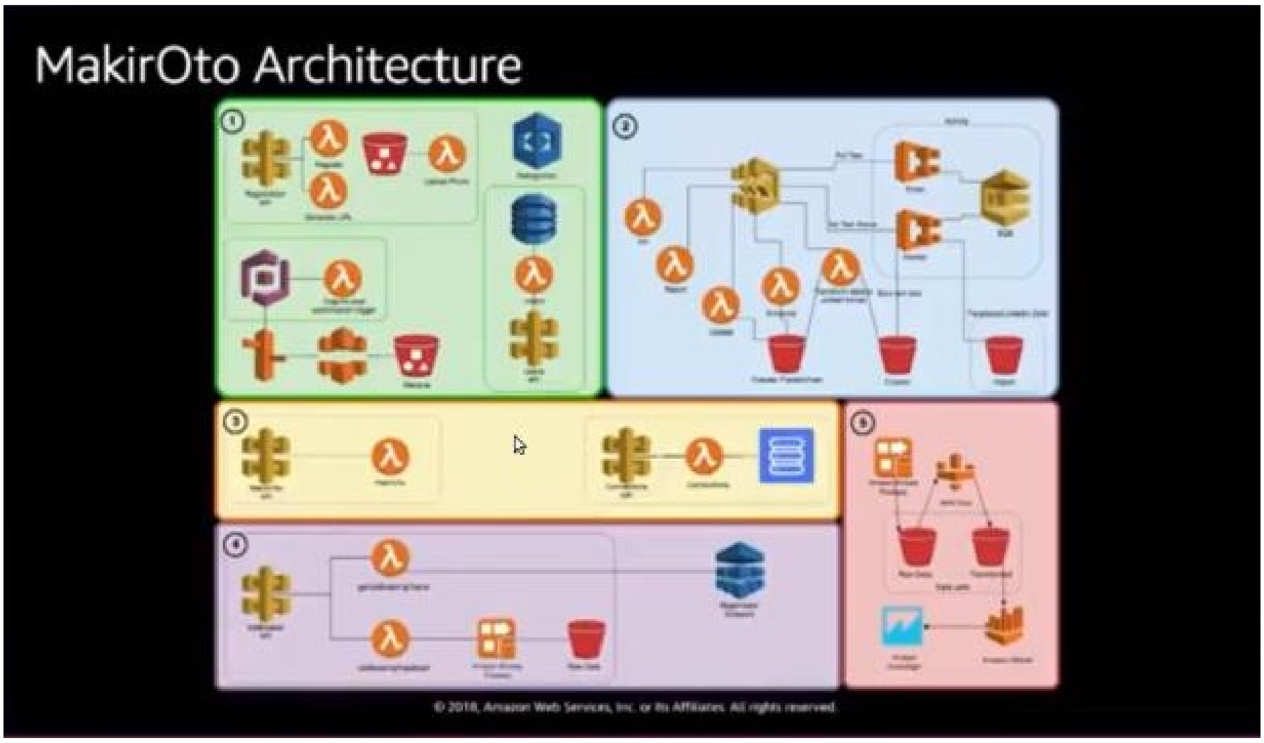
There are some more advanced topics here. Software architecture blueprints have to constitute a solid basis for system observability. The same blueprints have to help with strategic planning: which parts of our application are mostly likely to be commoditized by the could vendor (using the ILC model) and how our open source strategy should look like (to get it right you will probably need to read the whole book). How all this could be possibly done by using pictures that do not have precise semantics?

wardleymaps
*The use of topographical intelligence in business strategy*medium.com
Today, we do not have an adequate, rich, yet consistent language to address all these concerns; and we desperately need it.

What Could be Done About It?

As a running example, I will use a sample application presented by AWS solution architects at the "Rapid Development using Serverless Infrastructure" track of the recent AWS Summit 2018 in Tel Aviv.

The application is called MakirOto ("knows him" in Hebrew). It's a cute application demonstrating many AWS Serverless capabilities. Here is how the MakirOto overall architecture was presented on the stage:


MakirOto Architecture
© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

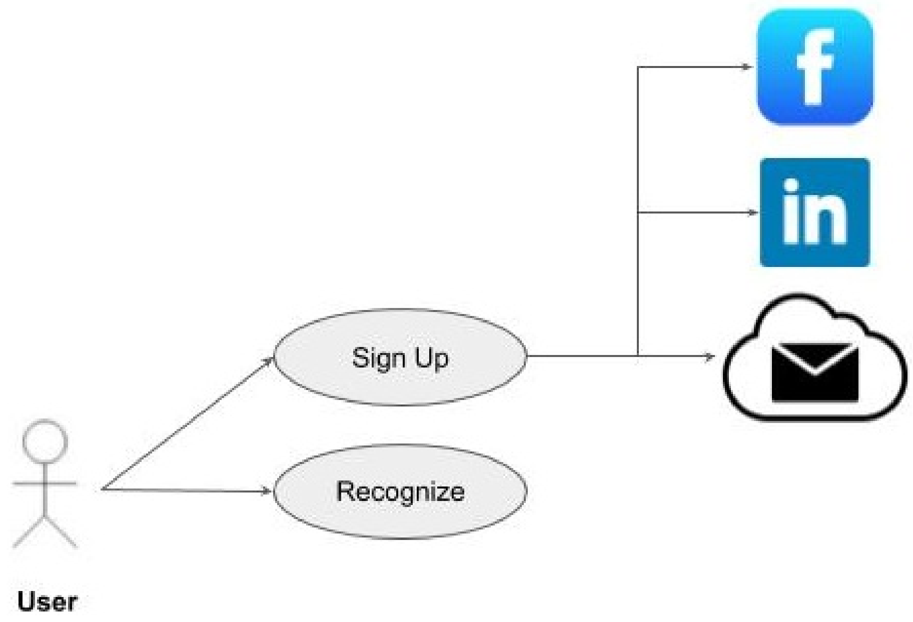
MakirOto Architecture

On this diagram, there is mixture of icons designating AWS services, such a Rekognition and API Gateway, and individual resources, such as Lambda Functions and S3 Buckets. It's unclear whether the icons representing Lambda Function mean instances, Lambda Function specifications (aka SAM templates) or individual deployments within particular AWS region.

Before we start making improvements we first need to understand and document precisely what the system does. More specifically, how the system interacts with its external actors (Use Case Model), and how these Use Cases are realized by the system internal components (Logical Model).

Use Case Model

Here is a Use Case model for MakirOto as I managed to understand it from the presentations (available in Hebrew on YouTube):



MakirOto UseCase Model

Proper modeling of architecturally essential use cases is typically bypassed, causing any subsequent technical debate to lack a reasonable starting point, defined by user needs. Nowadays, the very term “Use Case” is too often applied incorrectly to anything which might have a slightest smell of *use*.

To keep this paper self-contained here is a semi-formal definition of the term “Use Case” I normally apply: “ Use Case is a closed interaction between external actors and the system, normally initiated by some actor to accomplish a particular goal”.

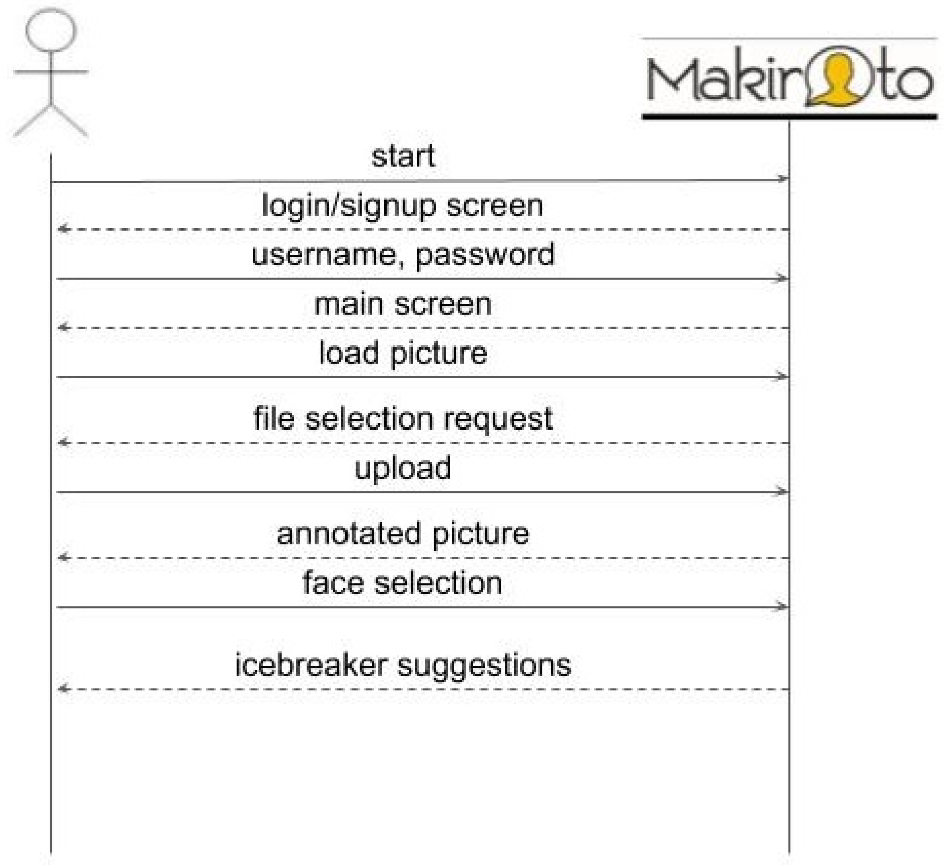
The Use Case model above does not say too much, but it does provide us with the right context for discussing architectural decisions. This is so important, that I want to stress it one more time:

until we have a clear and agreed upon picture of how the system interacts with its external actors to deliver some value, we should not start any technical discussion of how the system internals are organized.

The first Use Case, “Sign Up”, is typical for many interactive systems. In MakirOto signing up new user assumes sending confirmation code via email , uploading the user picture, and extracting additional personal information from social networks.

The seconds Use Case, “Recognize”, reflects the main value proposition of the system — it is THE major architecturally significant Use Case that justifies the system to be developed in the first place. The rest, including the “Sign Up” Use Case, will be needed only to support the “Recognize” Use Case. Supporting Use Cases might be technically very challenging (or interesting), but they do not reflect the true nature of the system and, therefore, should be analyzed only after we get a clear grasp of the system main Use Cases.

So, what happens in the course of the “Recognize” Use Case? A typical event flow could be illustrated with the following Sequence Diagram:



MakirOto “Recognize” Use Case Typical Event Sequence

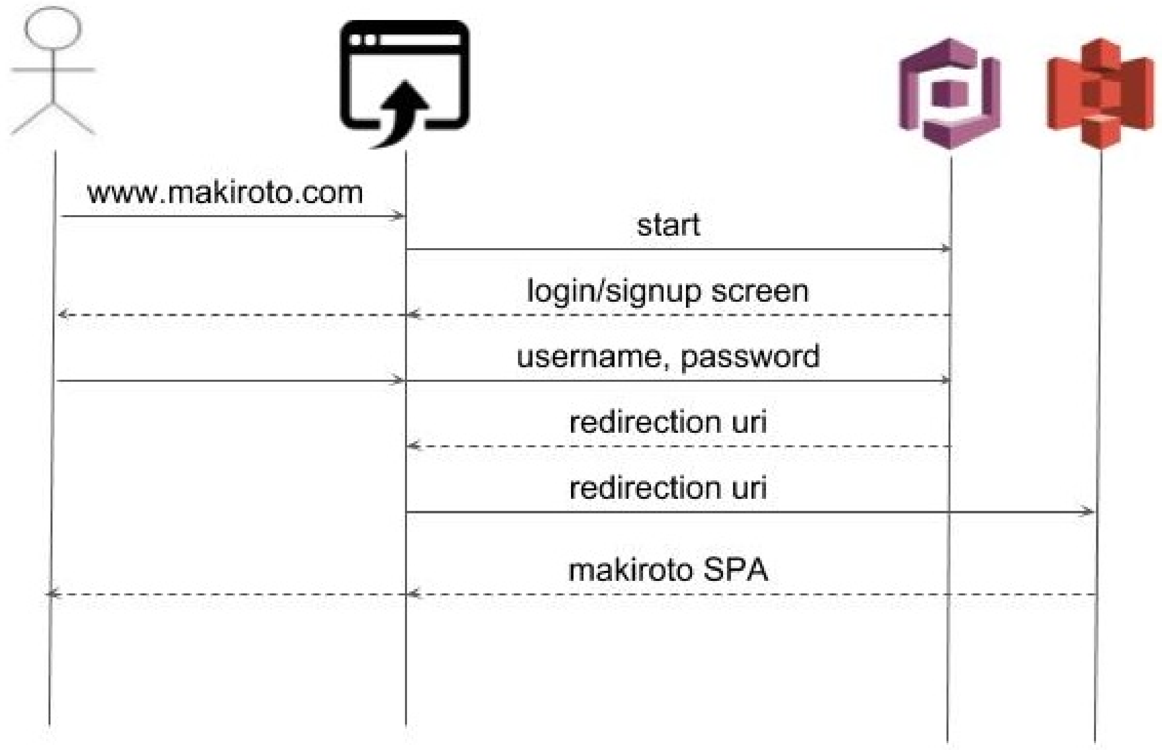
Sequence Diagram is an extremely powerful and too often underutilized tool for architectural analysis. I will cover some important subtleties of sequence modeling in a separate post, but for now it would suffice to say that all performance (latency, throughput) and availability requirements analysis will start from here.

In the case of MakirOto there is another scenario when recognition request is submitted for live camera, but since this a minor variation of the first scenario we will skip it here.

The first couple of steps, up to the “Load Picture” command, are common for any user identification process and by themselves are not very much interesting. After successful login the User uploads a picture for facial recognition of people presented on it. After that the user may ask for additional information about recognized individuals: where and when (s)he might have encountered them in the past, what are their interests, and what discussion topics would be better to use for starting a conversation. This is the heart of the system and this where we should keep our focus at.

Logical Model

So far we treated the system as a black box — we want to know what happens externally before we dive into details. The next step would be to see what the system does internally in order to support these interactions. For that purpose we need to zoom into the corresponding Sequence Diagram:



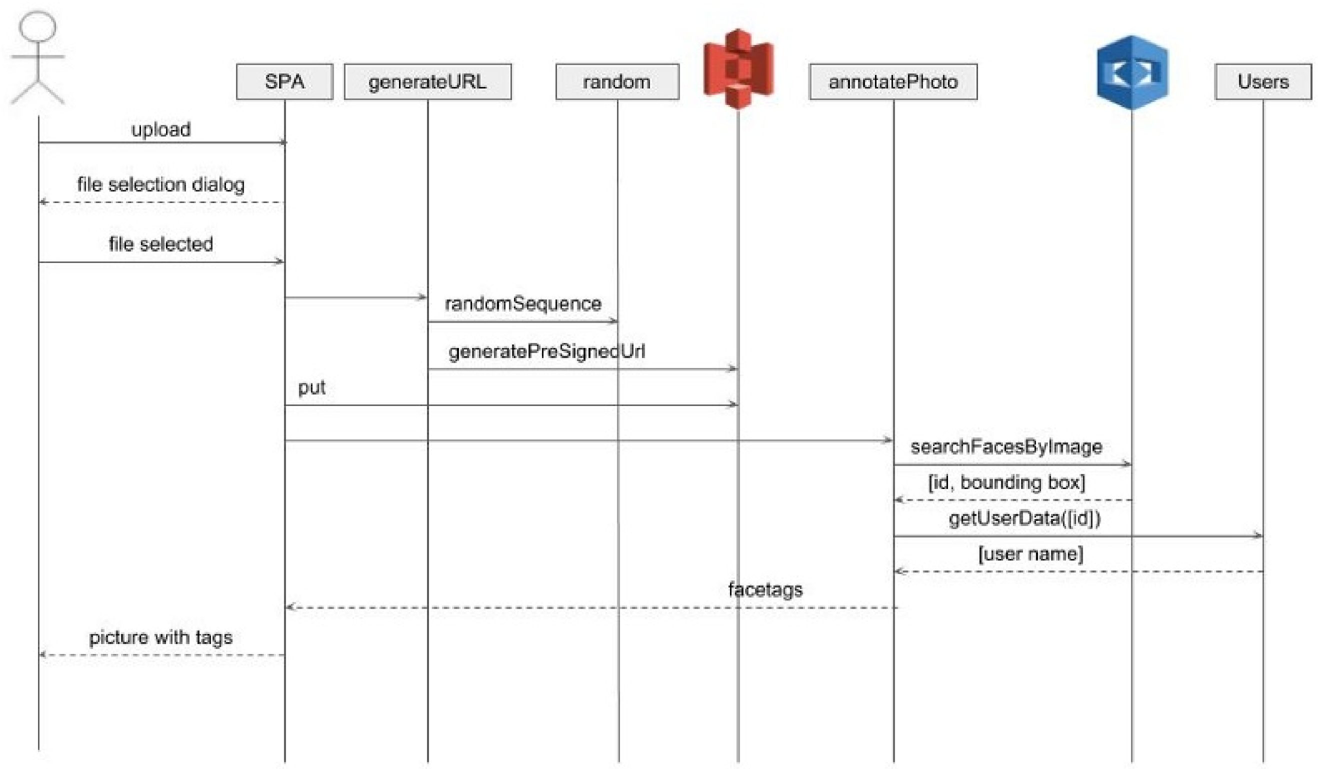
MakirOto “Recognize” Use Case, Login Sequence

Within the context of Serverless architecture, this type of model is intended to answer the following question: “which parts of the system logic need to be custom developed and which may be delegated to managed services?”

Having this very specific question in mind, we could now specify with absolute precision semantics of every AWS icon at this diagram — it will designate an SDK of some AWS fully managed service we will delegate some activities to.

What the sequence diagram above tells us is that the whole user identification process is delegated to Amazon Cognito, which at the end of successful authentication will redirect the Browser to a Single Page Application (SPA) hosted at AWS S3. This is a completely commodity service, and we could anticipate that in the future this functionality will be packaged in some reusable component hosted at the AWS Serverless Application Repository (see also Simon Wardley’s forecast about how the future Serverless development process might look like).

Now let’s take a look at what happens when the User clicks the “Load Picture” button (the Browser icon is omitted to save some space):



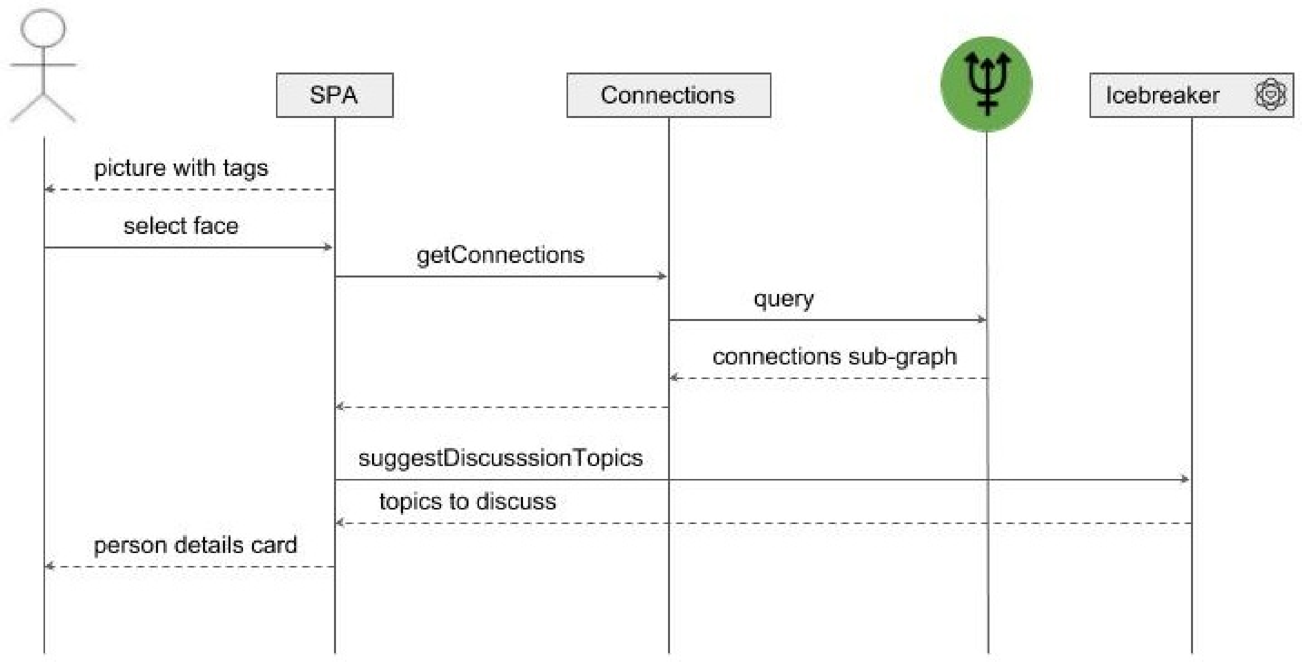
MakirOto “Reconginze” Use Case, Upload Picture Sequence

Apparently, this is about tagging all faces, recognized on the picture, with names. Here we make an implicit assumption that the system will recognize faces only of the registered users. The actual steps are pretty straightforward:

- Generate random character sequence
- Ask AWS S3 to create an pre-signed URI from this random sequence plus some fixed prefix
- Upload the picture to S3 using this pre-signed URI
- Ask Amazon Rekognition service to identify faces on the picture
- Retrieve additional information about each recognized face (specifically name)
- Display annotated picture

This is also a very common task—there are many applications, which might need a face tagging service. Therefore it’s yet another good candidate to be placed on the AWS Serverless Application Repository. This is still not the heart of MakirOto system.

To get there we need another Sequence Diagram:



MakirOto “Recognize” Use Case, Select Face Sequence

When user selects a tagged face on the picture, the SPA will retrieve a sub-graph connecting the current user with the selected person. These connections could be some places they both attended (such as school, army service, university, workplace or vacation trips), interests and potentially many other things. The next step is to ask the custom AI service (called Icebreaker) to suggest which topics would be the most reasonable to use as a conversation-starter.

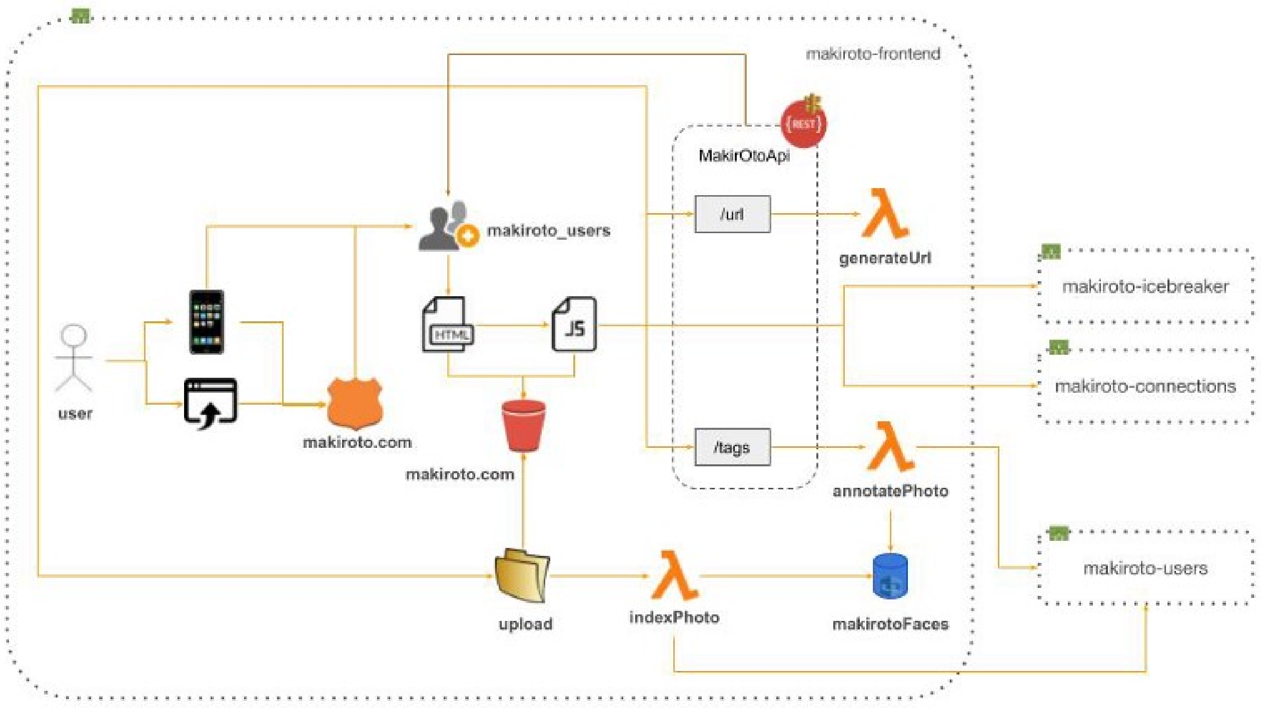
This is the system’s core domain and core value proposition — to suggest discussion topics, based on personal information available, the rest is secondary and intended to support the core.

Here, we will need a close cooperation between sociologists, psychologists, data scientists, and software engineers. Here, we will need to spend some quality time with Product Manager and Business Development to see whether the whole structure leads to an attractive ROI and is resilient to potential competition. At this point we might think about additional Use Cases for the same core technology (for example, meeting preparation).

In the diagram above some concrete architectural decisions were reflected, namely, that AWS Neptune will be used to store and retrieve connection graph, and that the whole process will be orchestrated by the SPA. This diagram reflects my understanding of what the MakirOto team did based on the presentation materials available. This understanding might be inaccurate or completely wrong, but I would argue that it is reflected in a very precise form, which is a big improvement over what we have today.

Process Structure

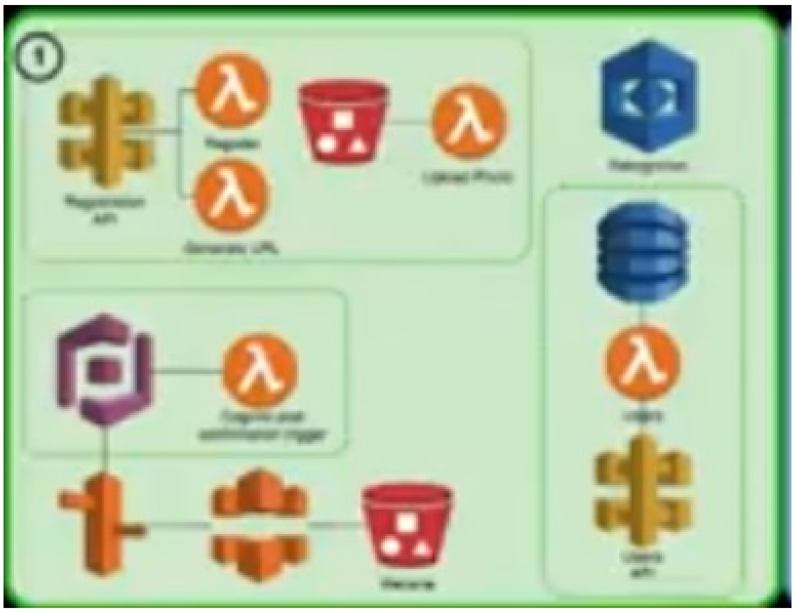
We have spent some substantial time and energy on analyzing the system interaction and core value proposition. It was an absolutely essential step, which cannot be bypassed—without it, all subsequent steps would be void of meaning. Now, we could start thinking about which boxes we will need in order to implement the system logic:



MakirOto Process Structure, v1

Here, every icon denotes either a computation process instance (e.g. Lambda Function) or a resource from an AWS managed service: S3 bucket, DynamoDB Table, Rekognition Image Collection, API, Route 53 Hosted Zone. All connection lines have standard UML semantics, namely pointer or aggregation. Computations and resources belonging to the same subsystem (aka microservice) are wrapped together in a CloudFormation stack.

In terms of precision this is an improvement over diagram presented from the stage, where resource icons were mixed with service icons, and many important details were missing:

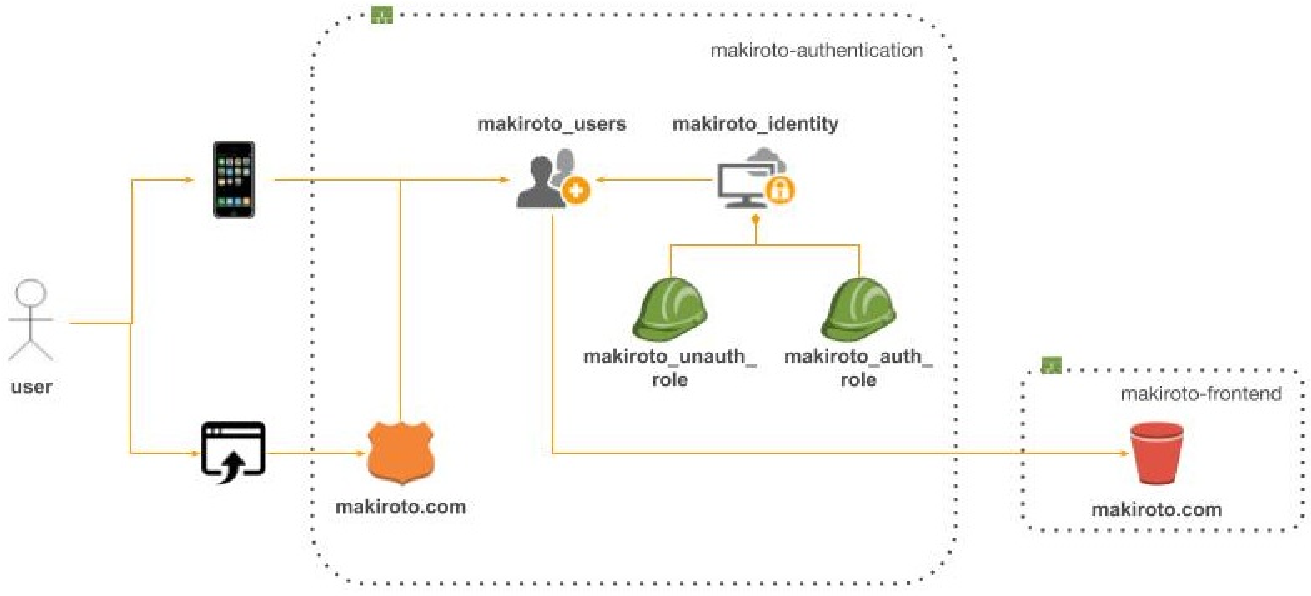


Now, we could annotate, if we want to, every icon with some important capacity details (e.g. amount of RAM and maximal concurrency for Lambda Functions) and every arrow with minimal required access permissions.

This diagram is still imperfect — hard to comprehend and, therefore, hard to use for an architectural analysis of throughput, cost or security. The problem is not only with the number of boxes and connections squeezed together in one small place, but also with putting together too many unrelated concepts. In other words, the proposed architecture lacks cohesion. Let's see if we could improve it by some gentle refactoring.

From initial analysis of the system logic we've got some idea which parts constitute its core and which play a merely supporting role. We can use these clues to break down the system into smaller and more manageable chunks.

First, let's extract out all elements related to User Authentication into a separate microservice:



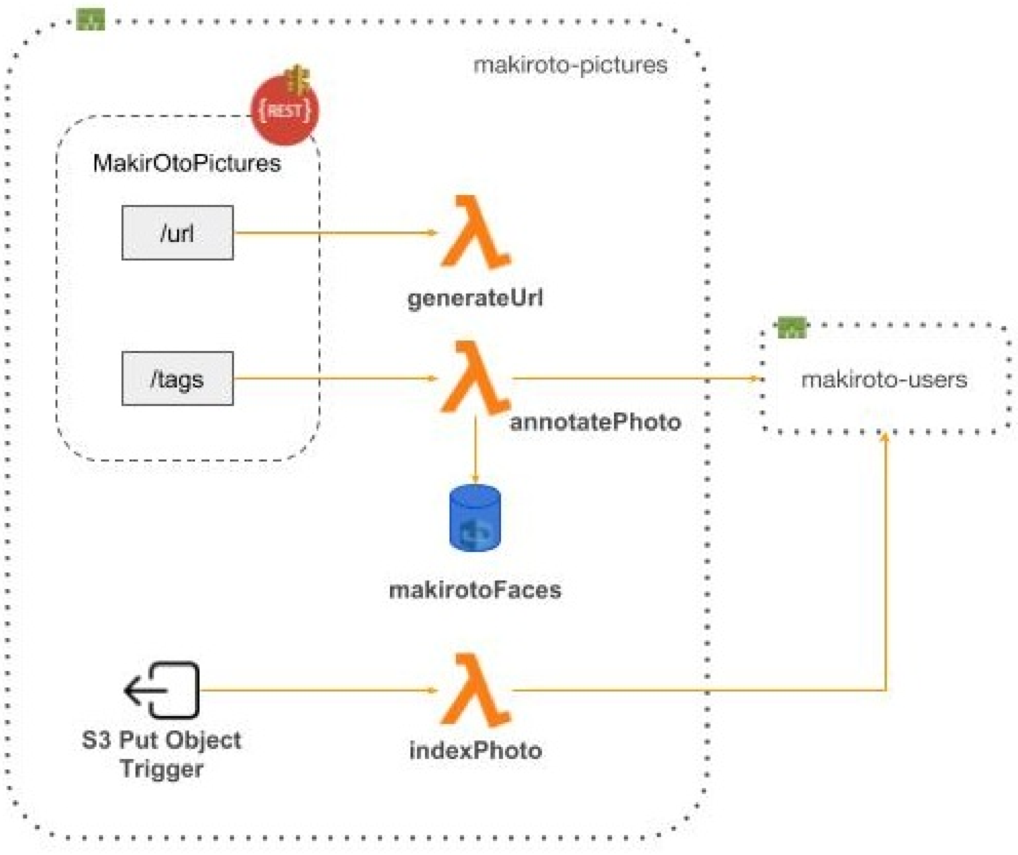
MakirOto Authentication Microservice

Now, we have not only precision, but also more clarity. This diagram of MakirOto Authentication microservice clearly tells its story. There are AWS Cognito User and Identity pools linked together (normal Cognito practice). I've added Identity Pool to support authentication federated with popular social networks, such as Facebook and Twitter, thus making the direct registration process optional.

Mobile application will authenticate or register users directly against the MakirOto User Pool, while Web Application users will access it using the makiroto.com domain name registered in a Amazon Route 53 hosted zone. The Identity pool will define two IAM roles: one for non-authenticated and one for authenticated users. To please our CISO we might need to elaborate on specific policies attached to each role, but this is a topic for another post.

The User Pool will be configured to redirect Web Application users to a static website hosted at AWS S3 (again, normal AWS Cognito practice).

Now let's extract the next piece of supporting functionality, namely MakirOto Pictures microservice:



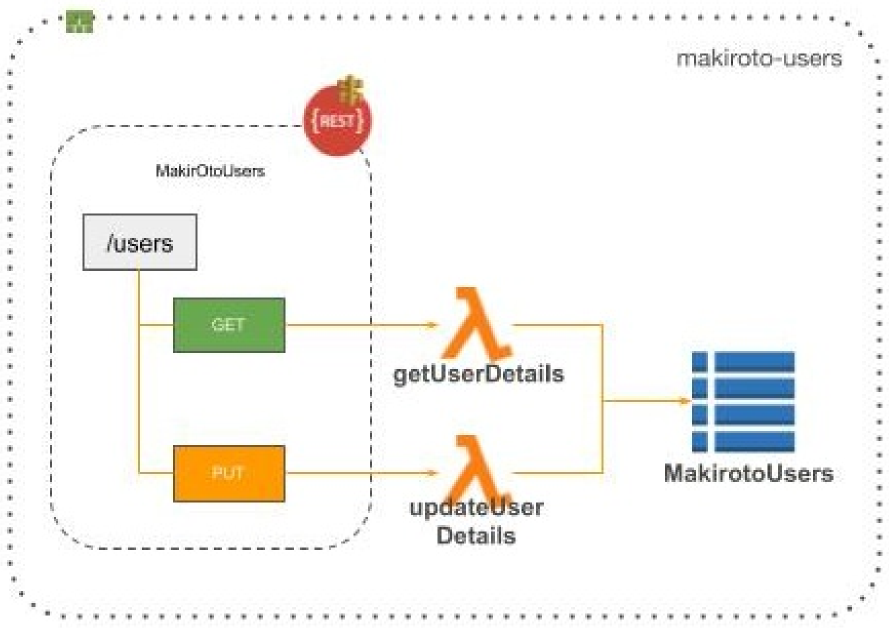
MakirOto Pictures Microservice

I consider high cohesion and loose coupling as the primary architecture guidelines. Therefore, I decided to put together all Lambda functions dealing with picture upload and processing (URI generation, photo indexing, annotation), and a AWS Rekognition Collection used to backup the process. Not bad, but there are a couple of issues to consider.

First, two Lambda functions are wrapped with a REST API using AWS API Gateway, while the third (indexPhoto) is supposed to be invoked by an S3 PutObject trigger. Is this lack of consistency really justified? I'm not sure, but in order to form my own opinion I need to understand the rationale behind the decision the MakirOto team made, so let's leave it as is for the moment.

The next question is whether the S3 bucket should be included into this microservice making it slightly more self-contained and more secure by keeping the upload area completely isolated from the rest of the system? That sounds like a reasonable idea.

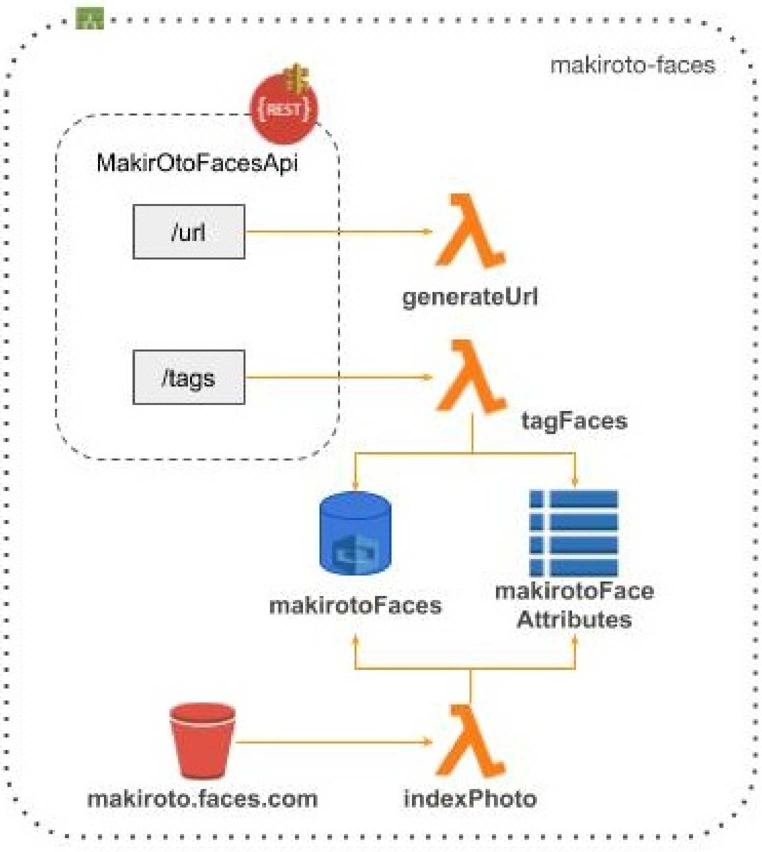
The last problem is a dependency of the Pictures microservice on the Users microservice. This is a bigger problem, since direct dependencies between microservices are considered to be an anti-pattern eventually leading to so-called death star structure. Do we really need it? In order to understand it, let’s look inside the Users microservice:



MakirOto Users Microservice

This is a so-called CRUD microservice encapsulating basic create, retrieve, update, delete operations over MakirOto user records. It looks like the whole purpose of the MakirotoUsers DynamoDB table is to keep additional information about the user. First, this is the user’s full name, which AWS Rekognition system does not keep. Second, these are properties, such as age group and gender, extracted from the face by Rekognition, cached to speed up future access.

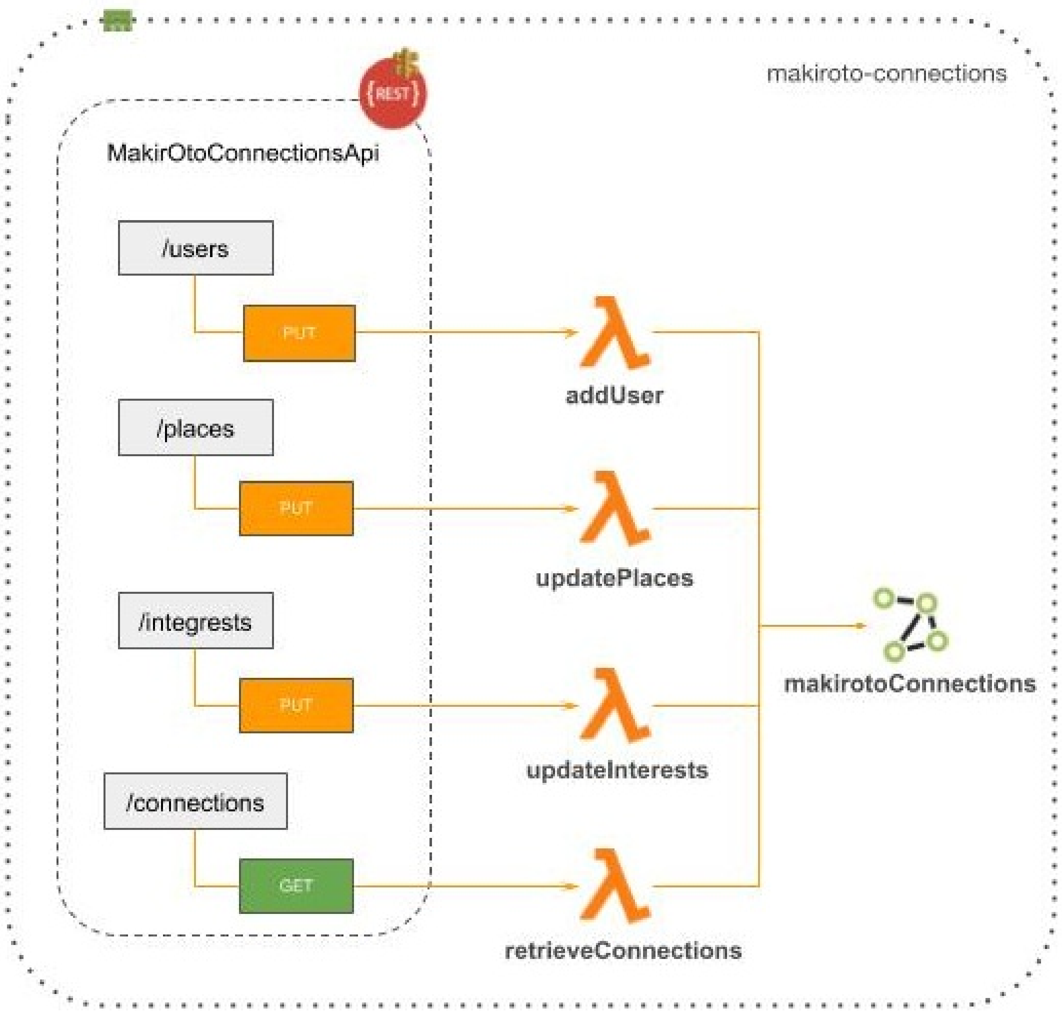
Since the annotatePhoto function should return real names, rather than user IDs, it might make sense to just include this table inside the Faces (renamed from Pictures to better convey its main purpose) microservice:



MakirOto Faces Mciroservice

Now, we have more cohesive and self-contained microservice, which is also a good candidate for further unification.

Before we put everything back together let’s take a brief look at the two remaining microservices.

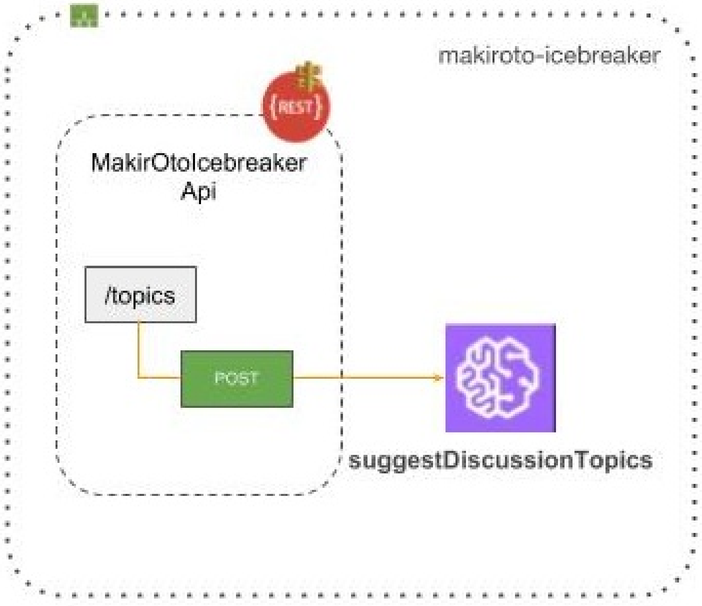


MakirOto Connections Microservice

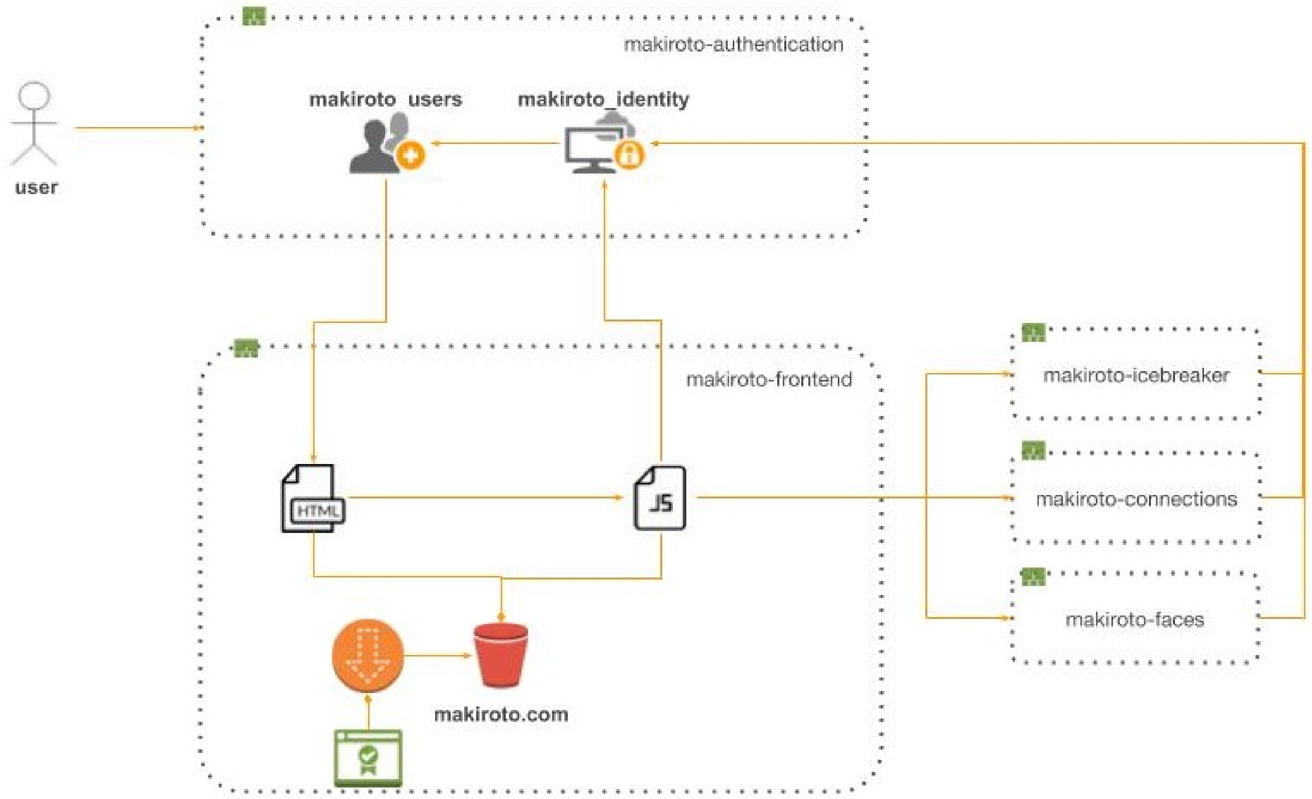
Here, I decided to wrap each update and retrieve operation into a separate Lambda Function in order to allow a fine grained access control. That will not necessarily be the most optimal solution, and additional Use Case analysis might be required. Lambda Function granularity is an important and challenging architecture topic. It would better to defer it to a separate blog post.

MakirOto Icebreaker Microservice

Some technical details are still missing: social networks crawling for Connections, AWS SageMaker model training for Icebreaker, and maintaining MakirOto Data Lake, but these are advanced topics better to be deferred to separate posts.



Now we could bring everything back together:



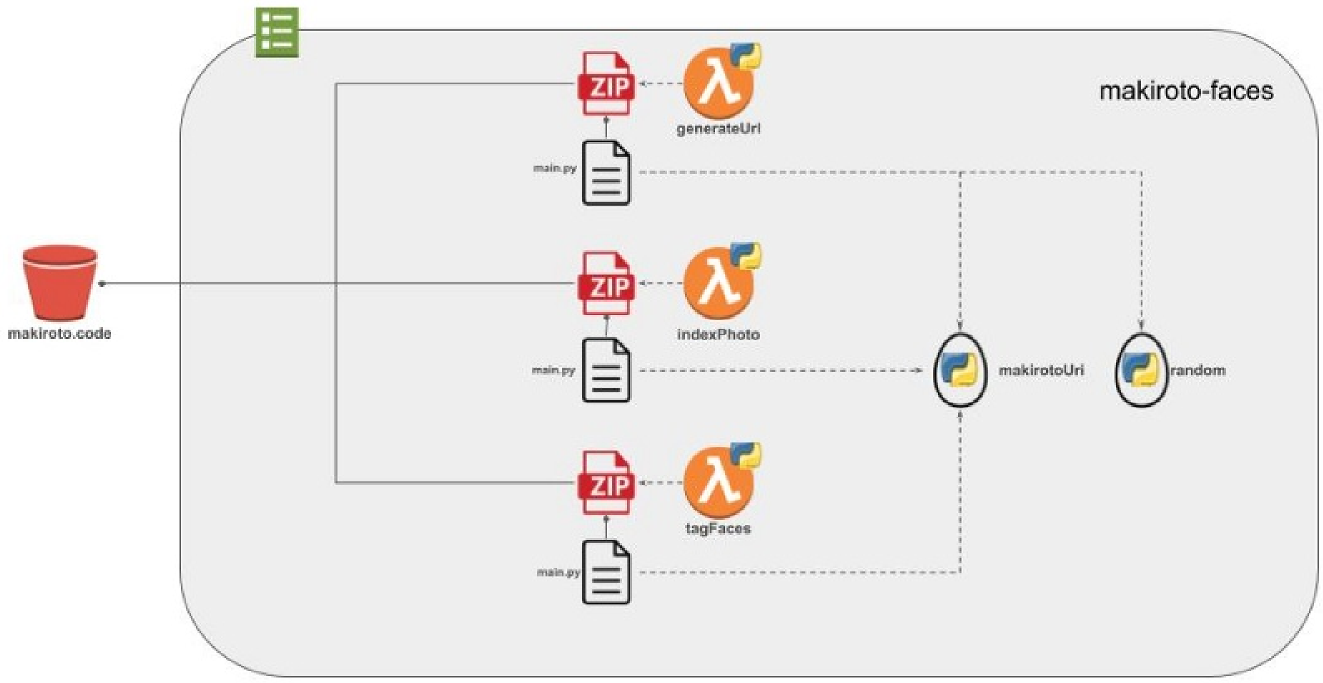
MakirOto Frontend Microservice

Hopefully things are more intellectually manageable now. Just keep in mind that within this context an arrow means a pointer with particular access rights and nothing more. It does not represent, for example, possible flow of commands or data between two elements. This distinction will be important when we talk about Serverless applications observability (panned for another post).

We are probably not completely done yet. Currently each back-end microservice has its own API exposed via AWS API Gateway. Is it really justified? Why not to keep back-end microservices at Lambda level and wrap them all together in one API Facade? What are alternatives to the AWS API Gateway, and what considerations would lead to choosing one over another? Good questions to be addressed in a separate post.

Implementation Model

Now, we might want to specify how some of microservices are built internally. Most likely not each microservice will deserve such attention, only those which we find to be non-trivial. In our case that would be the Faces microservice:



MakirOto Faces Microservice Implementation

Here, boxes mean sources files and build artifacts: shared modules (e.g Python Eggs), Zip Files, Lambda Functions, etc. while arrows means build dependencies: “in order to build this we will need that.”

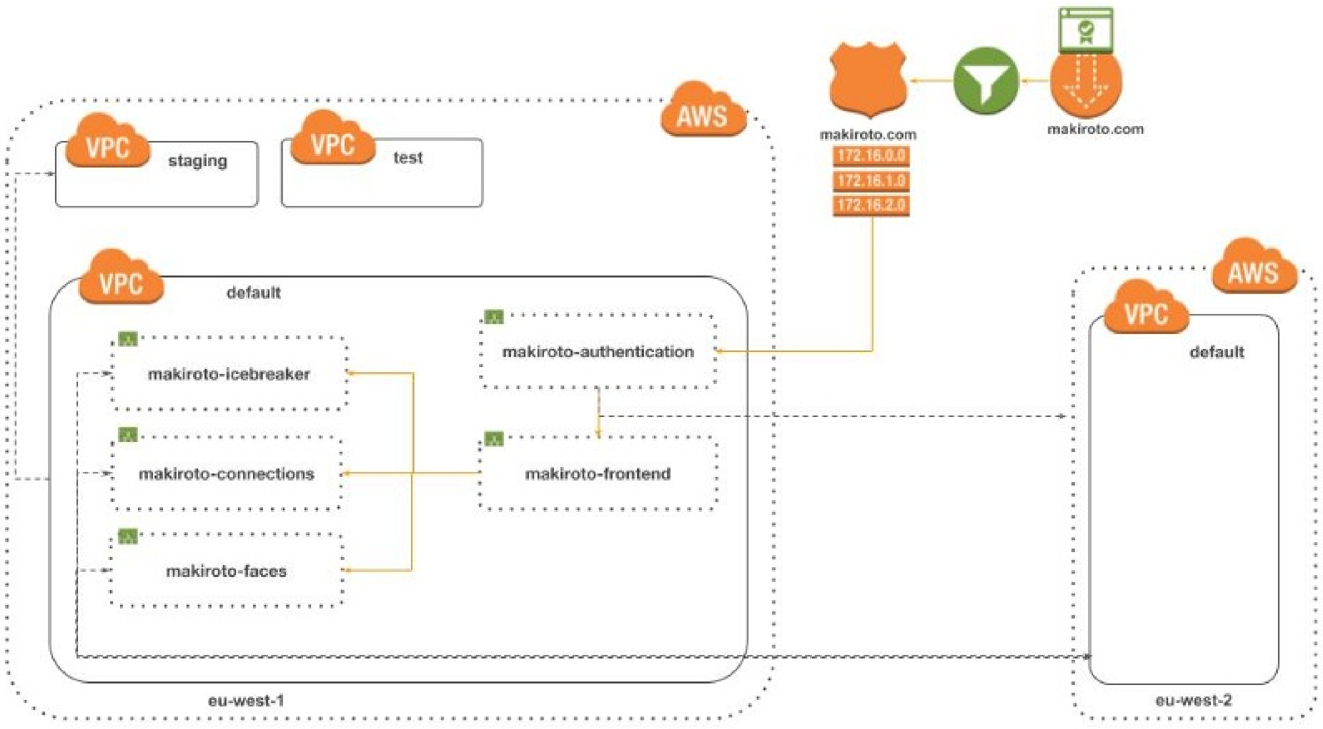
This diagram is interesting for a couple of reasons. First, to avoid clutter, I deliberately decided to omit many details, such as S3 bucket for upload, API Gateway resources, Rekognition image collection and DynamoDB table. As it very often happens, models of different types overlap. All those details have already been documented when we looked at the process structure of Faces microservice. Reproducing the same information manually would be too tiresome, and we do not have a suitable tool yet.

Second, this diagram reflects an important architectural decision regarding where ZIP files with Lambda functions code will be located. It might be tempting to keep sources within the upload bucket. However, this does not seem to be the right solution from security perspective. It is not a good idea to keep Lambda functions code in an S3 bucket, which is accessible by external users for upload. This particular decision is not specific for the Faces microservice only — it is common for all microservices.

Third, this diagram reveals something, which otherwise could be easily overlooked: all three Lambda functions need to be somehow coordinated with regard to the URI format. Keeping that coordination implicit would be the most problematic choice. In this case I decided to extract common logic into a Python egg package. An alternative solution could be to give up on three separate Lambda functions and put all computations together in one Lambda function. Also keep in mind — coordination over URI structure came up as a result of the decision to encode some important information in URI directly. If S3 meta-data were used instead, these internals would look differently.

Deployment Model

The last aspect of the system architecture, we need to analyze, is how our system is going to be deployed across multiple regions. For that purpose we will need to produce another model:



MakirOto Deployment Model

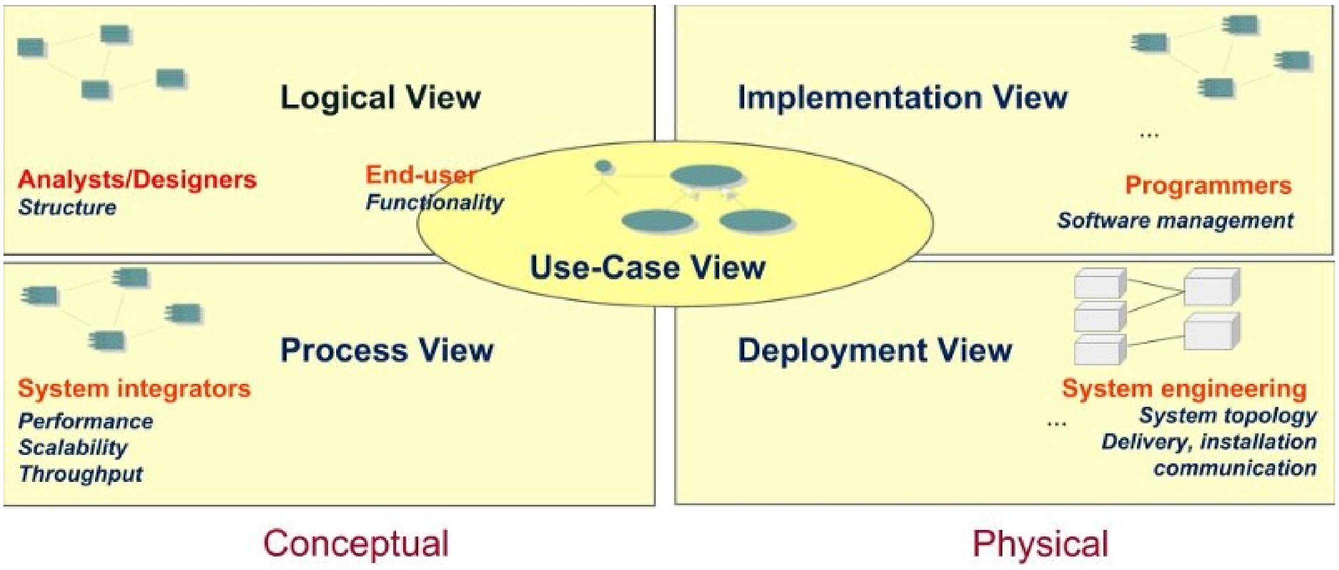
The Deployment Model diagram above presents a relatively simple setup with two regions: one for primary production operations, testing and staging, and another for disaster recovery backup. IP routing table points only to the Cognito authentication entry point, the rest is directly wired at region level. Every microservice, except for the Frontend, takes care for its global replication. For S3 buckets and DynamoDB tables, cross-region replication is supported out of the box; however for Cognito User Pool, some special care needs to be taken (e.g. via Lambda Function triggers). In addition, production data export into Staging area needs to be implemented somehow. This, in turn, might bring additional requirements to be addressed within other Models (Logical, Process, Implementation). And it is typically the case — architecture process is seldom linear.

The deployment strategy presented above might be debated over whether it is correct, or more likely, whether it provides an adequate solution for security, availability and cost requirements. Chances are**,** it is not optimal yet. However, what is important is that in its current state, it is reflected in an explicit and precise form making it more conducive for additional scrutiny.

Concluding Remarks

Software architecture is more about clear communication of simplest possible solutions, rather than about inventing something overly complex to digest. The current state of affairs with software architectures in general and Serverless architectures in particular is far from ideal—we still do not use a consistent architecture modeling language in daily practice. One of the main reasons seems to be that we are constantly trying to put into one diagram or picture more than it could bear. Maintaining semantic consistency is hard and is effectively possible only within a well-defined narrow context. Any non-trivial system will need more than one model, each one reflecting a particular system aspect. Furthermore, all these models will have to be logically mapped onto each other to preserve coherence across the whole system.

In this post I used 5 different models: Use Case, Logical, Process, Implementation and Deployment. In fact, I adopted so called “4+1 View of Software Architecture” approach developed by P. Krutchen. In the original paper different models are called Views, and the overall structure was presented using this diagram:



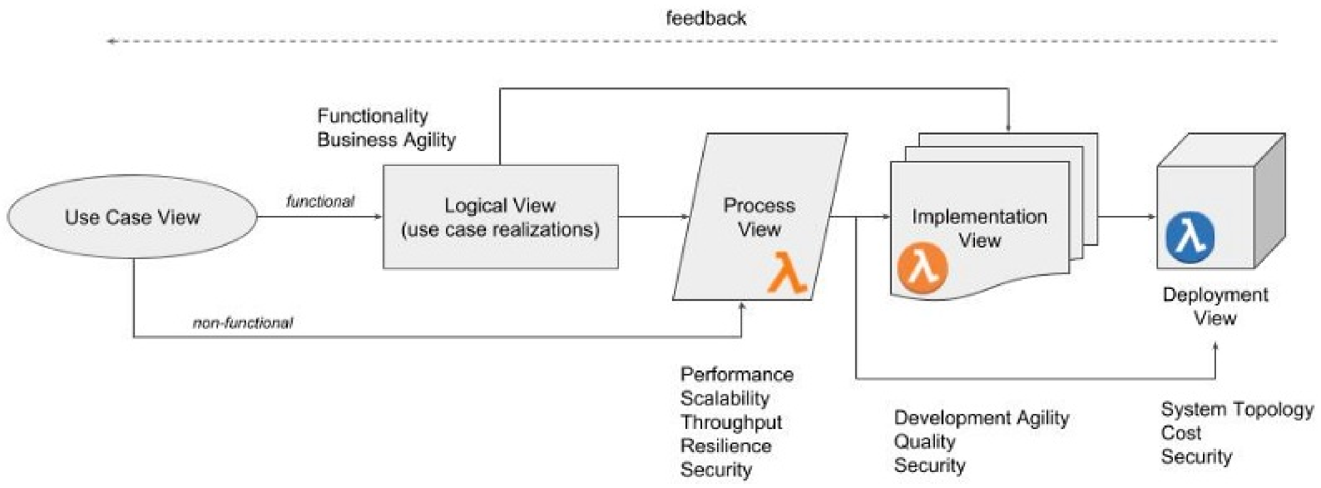
“4+1 Views of Software Architecture”

Views definition and responsibilities are summarized in the table below:

View	Reflects	Documentation
Use Case	HOW external actors interact with the system	User's Manual
Logical	WHAT the system does	Business Rules
Process	HOW the system runs internally	Operations Manual
Implementation	HOW the system is built	Developer's Guide
Deployment	HOW the system is deployed	Operations Manual

Views Definitions and Responsibilities

In real life the architecture process will unlikely be linear. More likely it will "oscillate" forth and back between multiple views. For example, when working on MakirOto Deployment View, we discovered that we need some custom solution for Cognito User Pool replication (AWS does not have it at the moment). We also discovered that we need to take care of production data export to Staging Zone. That will most likely bring us back to the Process View, and the whole process will likely take several iterations until it settles down. This forth and back dynamic between views is illustrated below:



"4+1 Views" Interactions

The "4+1 Views" model suggests a consistent language to describe software architecture from multiple angles and to map them onto each other for preserving coherence of the whole structure. It is indeed a very powerful modeling approach based on a solid scientific foundation.

Equipped with this language we now could reason systematically about various aspects of Serverless architecture be it Lambda Function granularity, observability, performance, security, productivity, APIs or Open Source strategy. We could also evaluate multiple architectural options without getting into overheated emotional debates “your vs mine architecture”. I plan to cover many of these topics in future posts. Stay tuned.