Micro-services

Polycode TAD Revision - Polycode Architecture

Simon LUCIDO

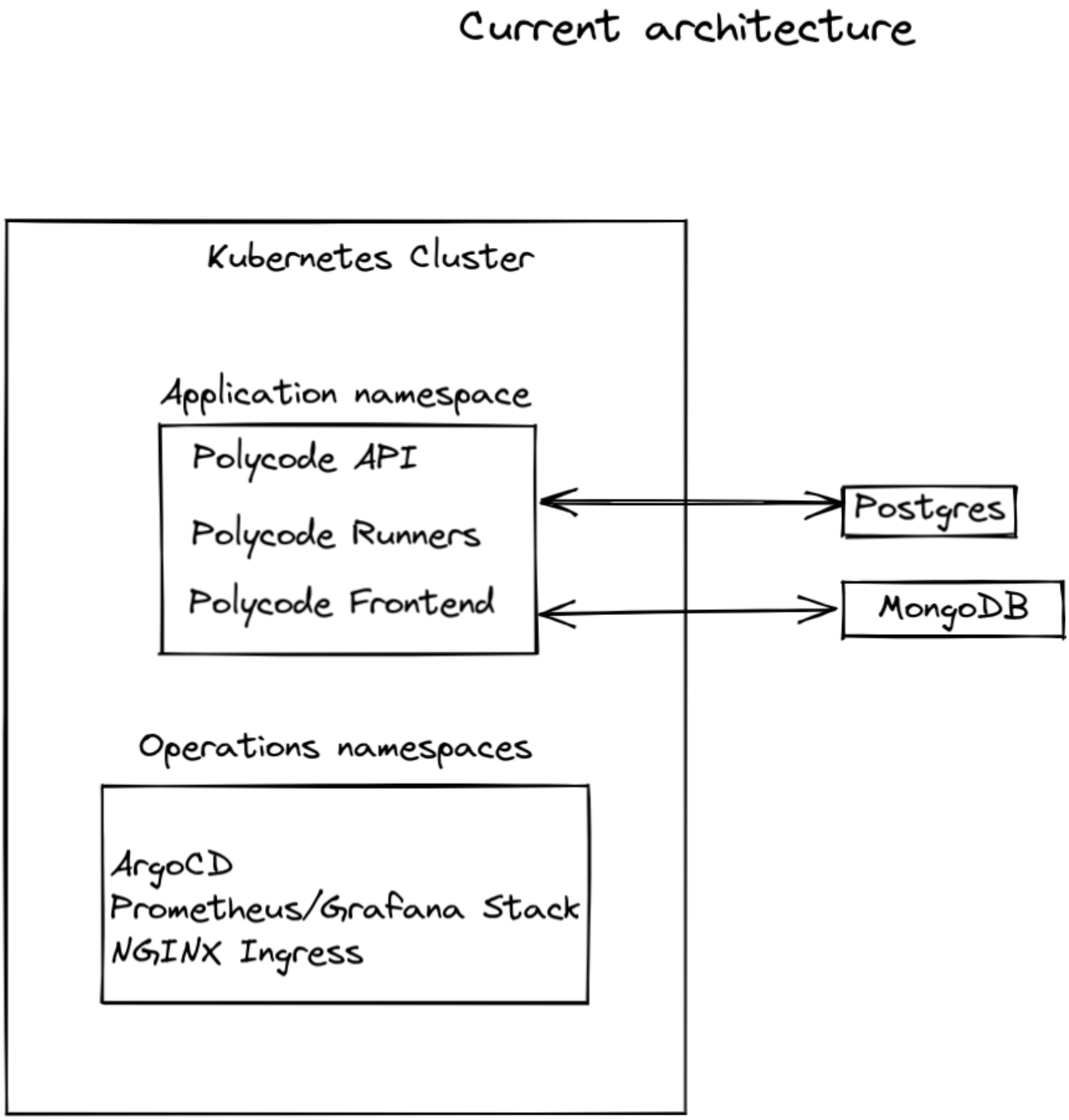
Version 1.0, 2023-05-23

Table of Contents

The existing

We first need to understand the current architecture of Polycode. To do so, I think looking at a (very) simplified diagram of the current architecture will best describes what is currently going on :

Current architecture



To give some contexts for those unfamiliar with the project, what I call "Polycode Runners", is the API that allow us to run the code that the user typed in.

Very quick pass on the stack we have :

- Kubernetes as our container manager.
- ArgoCD as our GitOps engine
- Nginx Ingress Controller as our Kubernetes ingress.

- The Prometheus/Grafana stack for our metrics and monitoring.

As you may have noticed, the current implementation is halfway between an old monolithic application, with a heavy native application, running on a bare-metal server. Instead, we are already in a Kubernetes context, which already allows for automatic scaling, if the application is made with the correct architecture decisions.

It is also worth noting that the current implementation mostly follows a service-oriented architecture.

From domains to microservices

It’s important to recognize the way you design you microservices will be reflected on the quality of your application. Poorly divided microservices will result in poor performance, poor maintainability, poor developer experience, and overall a bad user experience. If your data is poorly isolated, it will span multiple microservices and create an integrity and consistency mess. Make microservices too big and you lose the benefits. Make microservices too small and you begin to overload your network infrastructure, increase costs, decrease maintainability and increase the risk of bugs. Whatever your implementation, a poorly designed infrastructure will cause headaches. Let’s see how we can create a sensible microservice architecture, using Polycode as our use case.

Domain Driven Design (DDD)

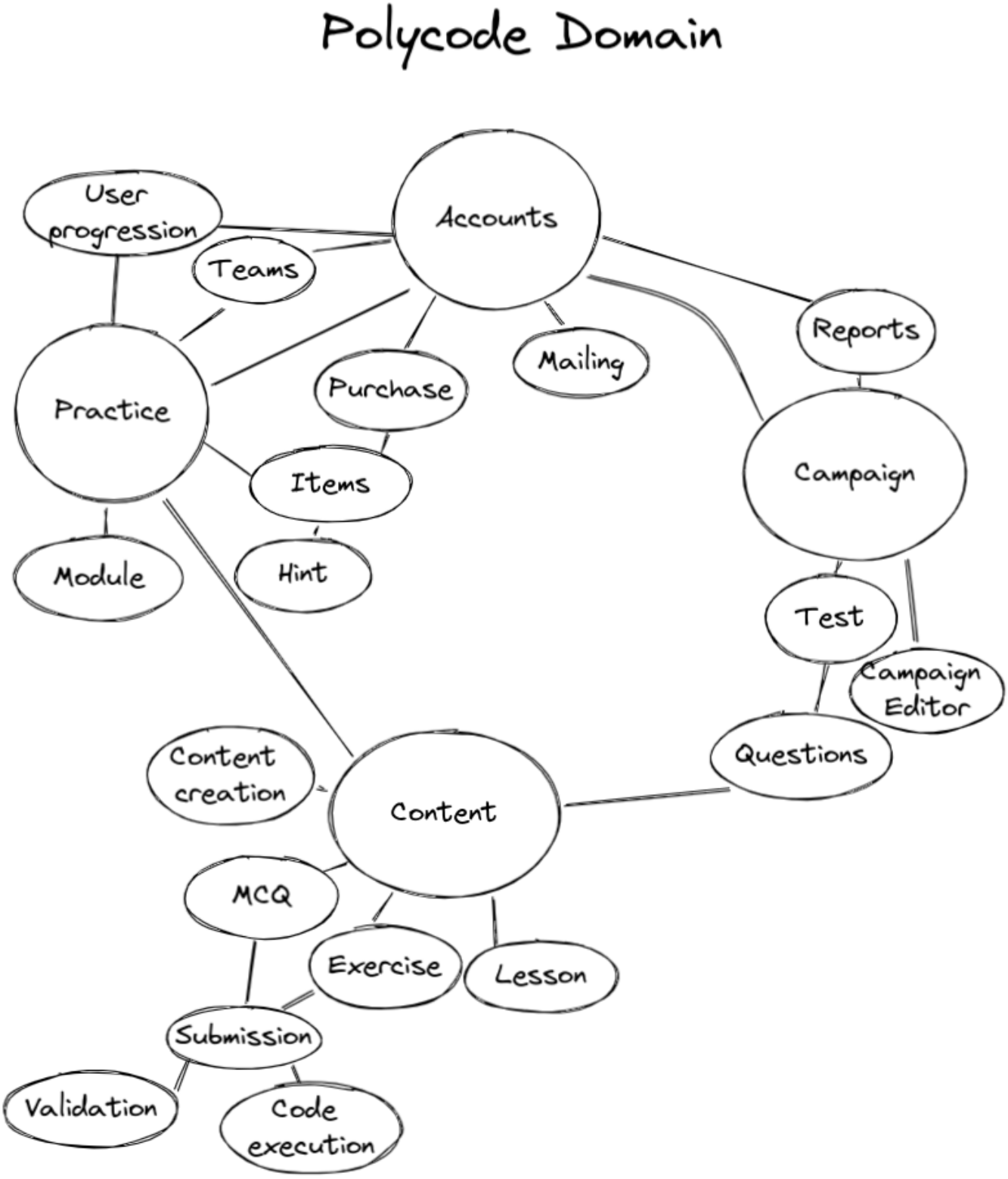
In order to make this goal more achievable, and to actually come up with a sensible approach, we are going to design our architecture based on the functional needs of our applications. Starting high up in the mental model of relations between components of our application, diving deeper and deeper until we come up with an actual microservice architecture. This is called Domain Driven Design.

For more information about DDD, please refer to my previous document.

Polycode Domains

Talking with domain experts, we can plot all our vocabulary on a diagram to better understand all the parts around Polycode. Here’s what I came up with:

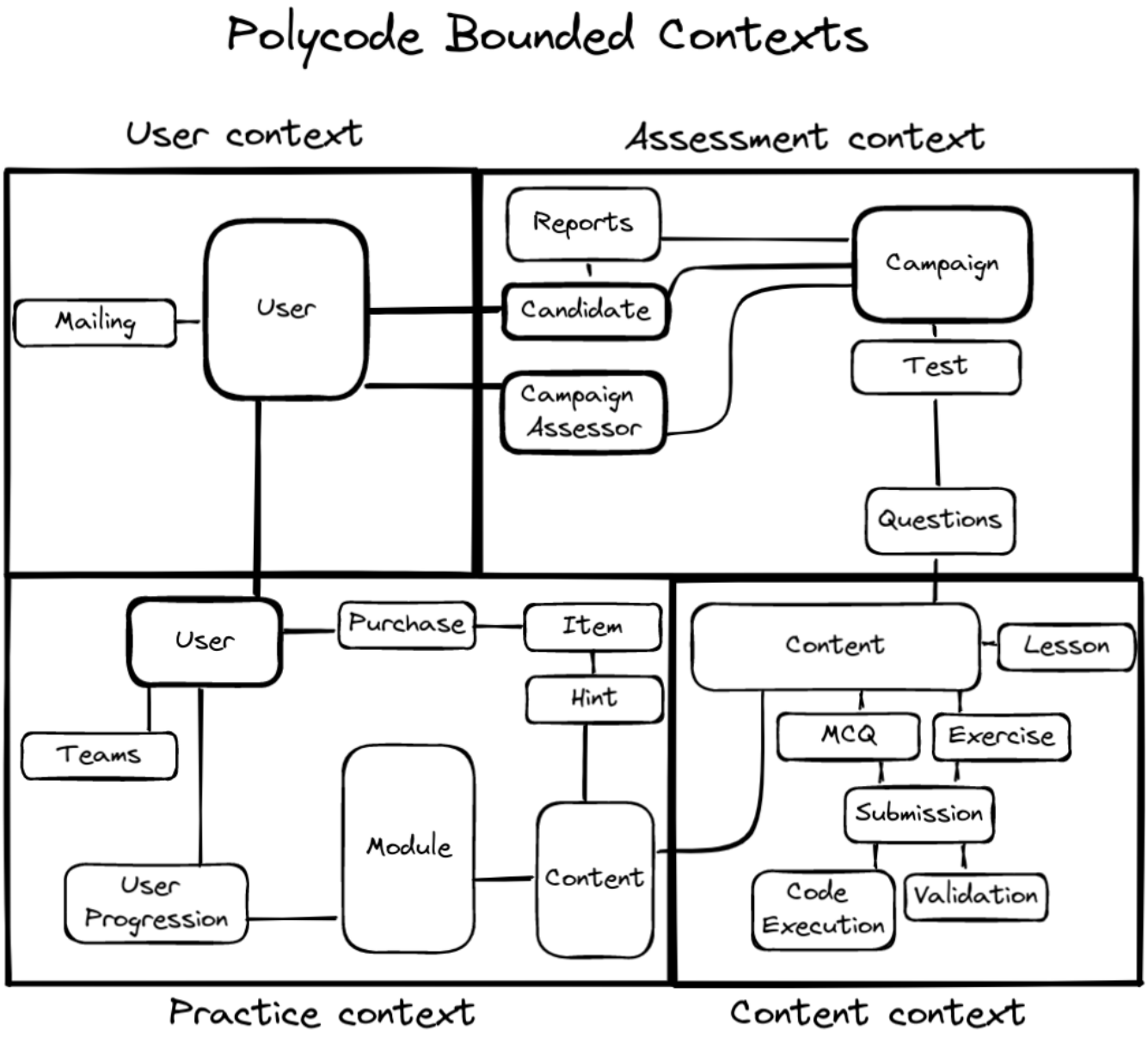
Polycode domains



Bounded Contexts

A bounded context is a specific way of defining the boundaries and scope of a domain. The goal of a bounded context is to create a clear separation between different parts of the domain and to explicitly define the ways in which they interact with one another:

Polycode bounded contexts



As you can see, I've divided the domain in 4 bounded contexts:

- The user context, which I also might refer as the account context, is the context where we have a deep knowledge about what a user is. This is where we handle his settings, account management, authentication. This is also the context responsible of sending emails, since an email is intrinsically linked to users.
- The practice context is responsible of handling the practice side of Polycode. This is where we have knowledge about teams and items, where we handle user progression and defines modules that the user will be able to follow. It is important to note that the notion of user in the practice context is not the same as the notion of user in the user context. A user, in the practice context, does not have a password or even an email. It is not needed in this context. This means that there will be translation layer in the communication between contexts, and this is normal. This is the cost we have to pay to make each of our context easy to work with and self-contained.
- The assessment context, in the same regards that the practice context, is responsible of handling the assessment side of Polycode. Separating it from the practice context makes sense to me, since we are handling a very different business logic. We need to grade candidates, with an invitation system on test that have a limit in time and that you can't retry.

- The content context. This is the harder one to justify, and it might look like I've let the technical implementation and details take over my thought process. I don't think this is the case. Wether you are in the practice or assessment context, you're not really concerned about the content itself, but more about the functionalities around it. You're concerned about what is a module, and that it has content within it, but whatever is the content. What is important to you is "Did the user finish this content ?" for example. In the content context, we don't really care where the content actually is. What we care about is inside this content, can we validate it, can we execute it ? How can we create new contents ? Contents exists in the assessment (as a question) and practice context, but they have a very different purpose than in the content context.

Now that we have defined our bounded contexts, let's move on to the next step.

Tactical DDD

We need to define our domain model with more precision. To do so, we will be using tactical DDD.

The user context

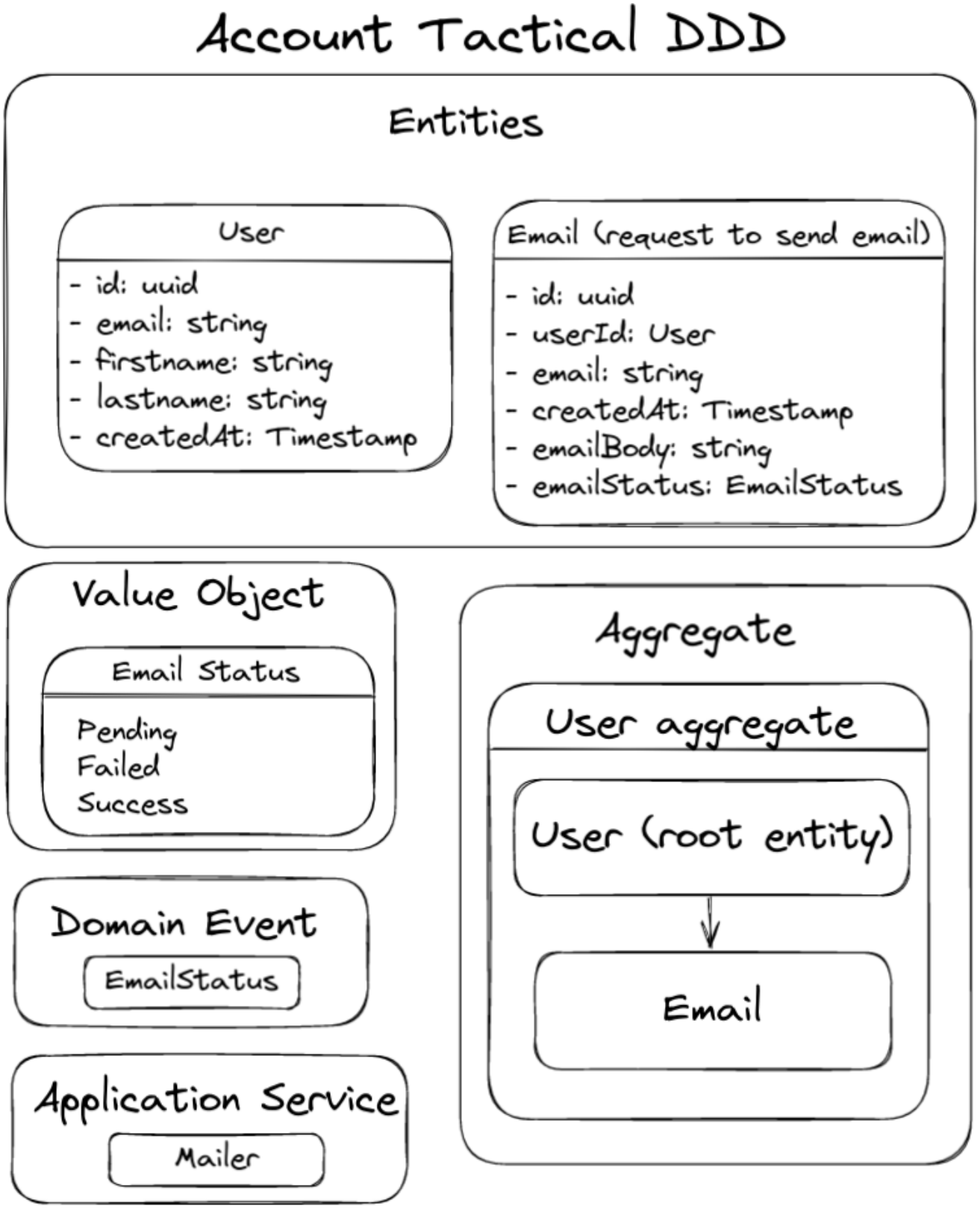


Figure 1. Account entities

We have the User entity and the Email entity. The user defines the basics of what is a user, in the account bounded context. A user should have a name, an email, and a unique identifier. We don't need to define anything else in this bounded context.

I've also defined a Email entity. This entity is here to represent and store emails sent to users (validation emails for example). The User is also an aggregate, composed of the user entity as the root, and the email entity as children of user. We want consistency between our user and email,

especially in this context, since we are dealing with user data. We want to make sure that when deleting a user, all its related sent email are also deleted. This is important to respect user data privacy, and to comply with GDPR.

Finally, I've also defined the Email Status domain event, which will notify the domain whenever a email changes status. This event should be sent when a email is created, when it has successfully been sent, or when it failed.

The practice context

Let's take a look at my take on the domain model for the practice bounded context:

Entities and value objects:

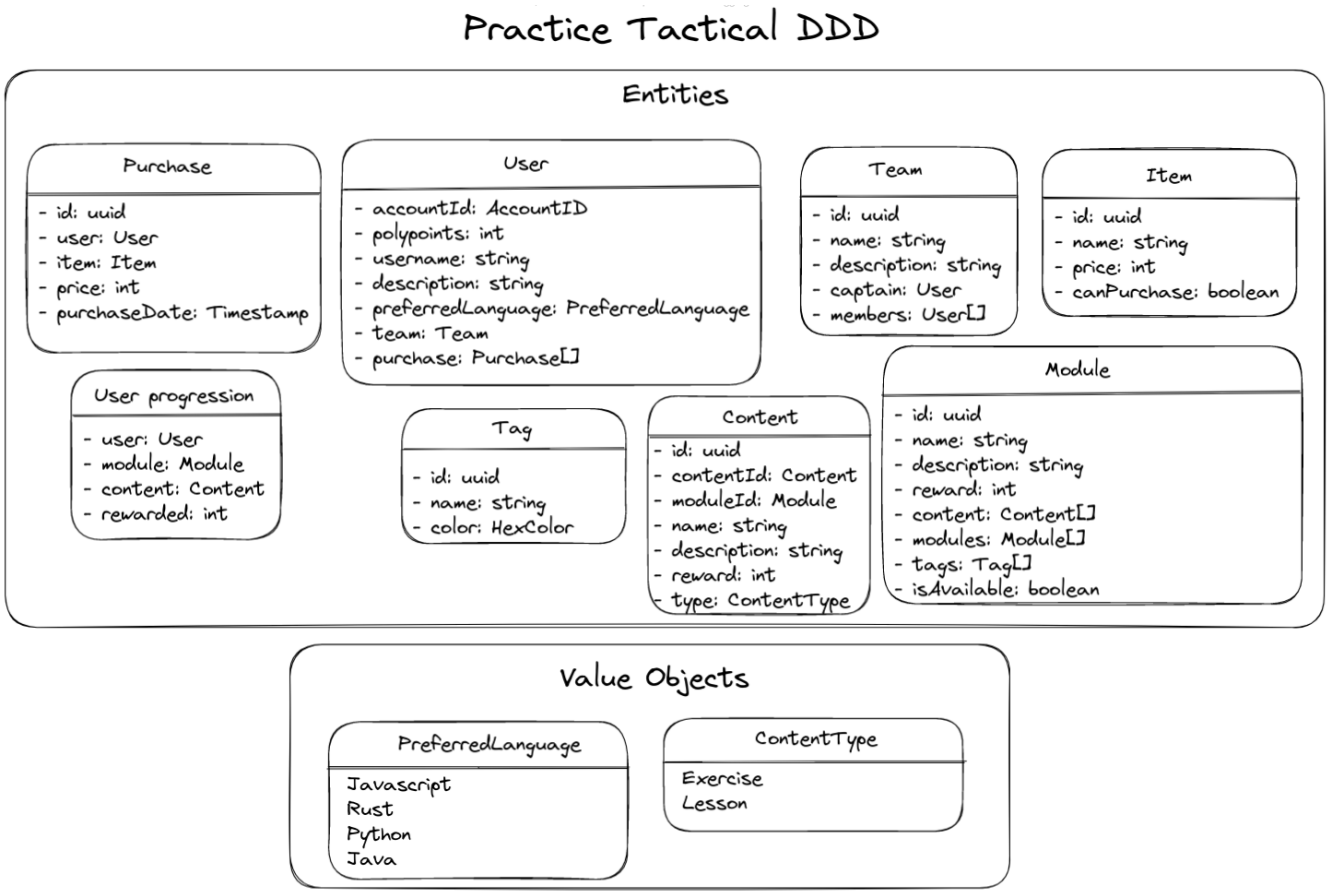


Figure 2. Practice entities and values objects

Aggregates:

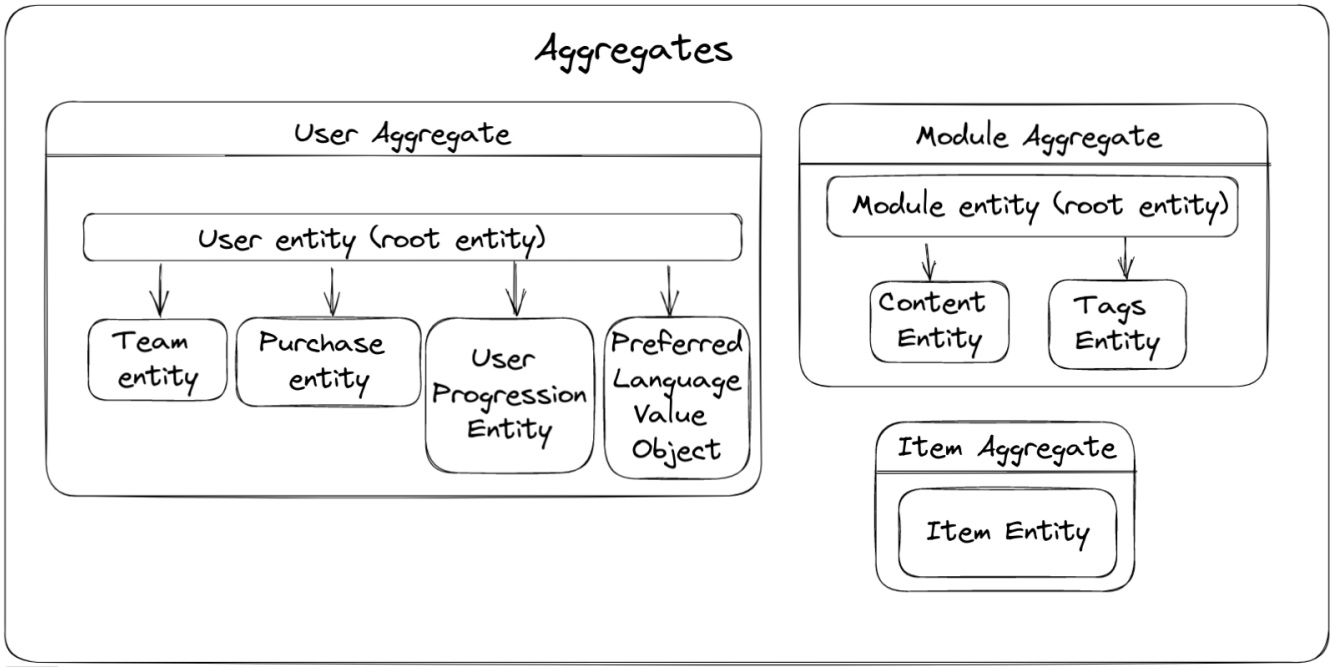


Figure 3. Practice aggregates

As you can see, I've defined the User entity, which is a different entity than the one in the Account context. However, the field `accountId` makes the link between the two. There is additional information that is relevant only within the practice context, and that's why there is a separate entity in it. User is also an aggregate, composed of itself as the root entity. It is linked to UserProgression, Team and Purchase, since I have identified a need for transactional consistency. In the case of Purchase, there is a need for consistency when the user buys an item. We need to make sure that he has enough polypoints, and the transaction should fail if there is an error in either updating its polypoints, or an error when registering the Purchase. There is a similar need for the user progression. Registering user progress and adding the corresponding rewards to its polypoints should be done in one transaction.

However, this kind of consistency check is not needed when we are not mutating the data. This is where patterns like CQRS shines, and allows for a better segmentation of your data access, and allow a much needed optimization in the case where your requirements for the command model and the query model are significantly different. We are not allowed to use CQRS in this paper, so I will not dive further into it.

I would now like to bring your focus to the Content entity. As we will discuss later, a Content is a totally different thing in the content context (content means a lot of different thing here, I'll try to make it clear). But in the practice context, we don't care about what is a content in the content context. We just now that OUR content has a name, description and a reward. This makes the content tightly linked with its parent module, this is why I've defined the module aggregate, with the module entity as the root and the contents as child entities. We need to maintain consistency here, when a module is updated or deleted, we need to make sure to propagate the needed actions to the contents.

The assessment context

I've identified the following entities, services, value objects and aggregates for the assessment context:

Entities, value objects and domain services:

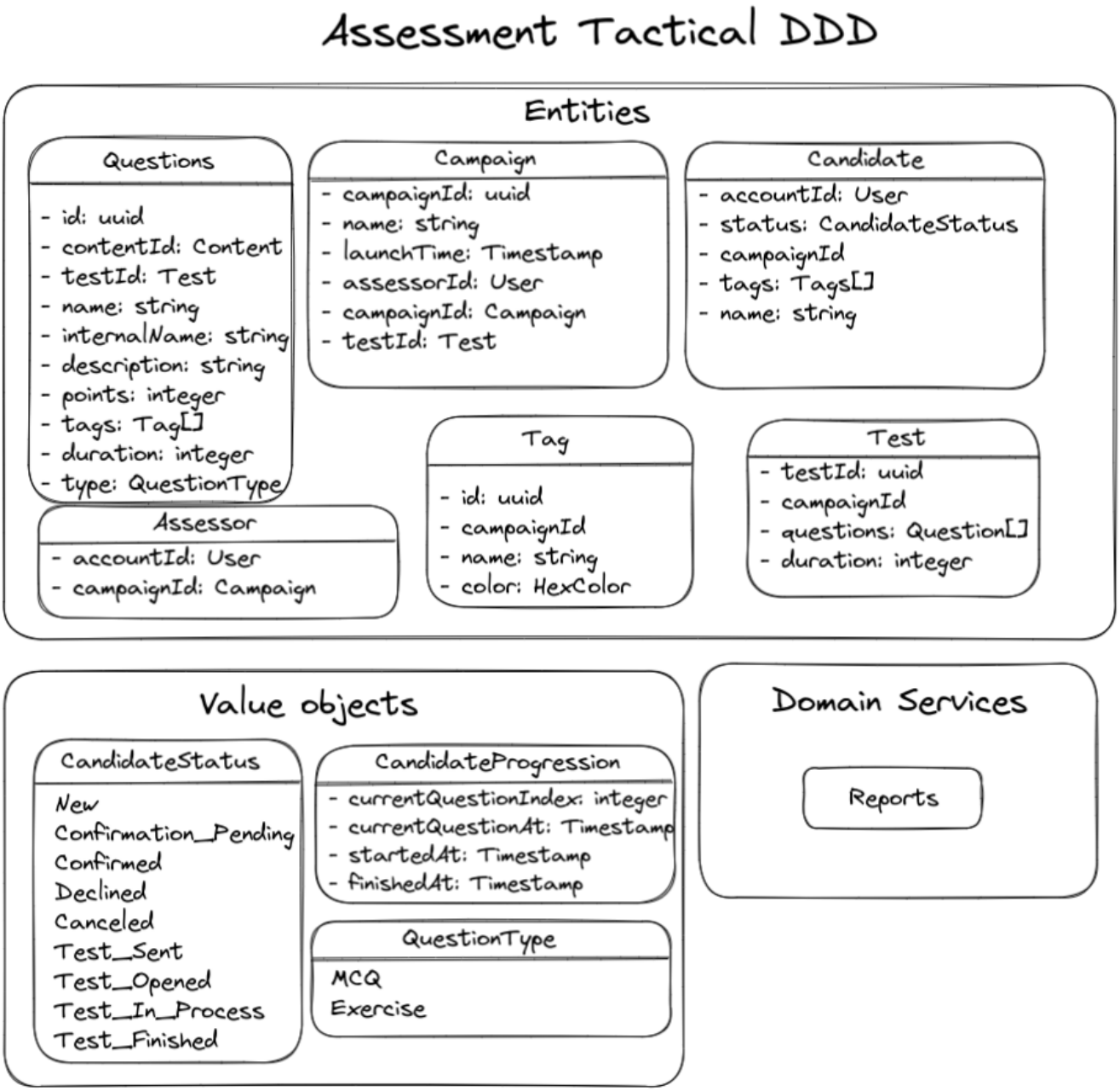


Figure 4. Assessment entities

Candidates have a status, which is an enumeration of value it can have. This has no identity, this is why CandidateStatus is a value object. As seen in the user stories, we need to send quite a lot of mails to our candidates. Even if this is a domain functionality, we delegated this part to our account context, which is why you don't see it appearing here. This will be an area of inter-domain communication.

Aggregates:

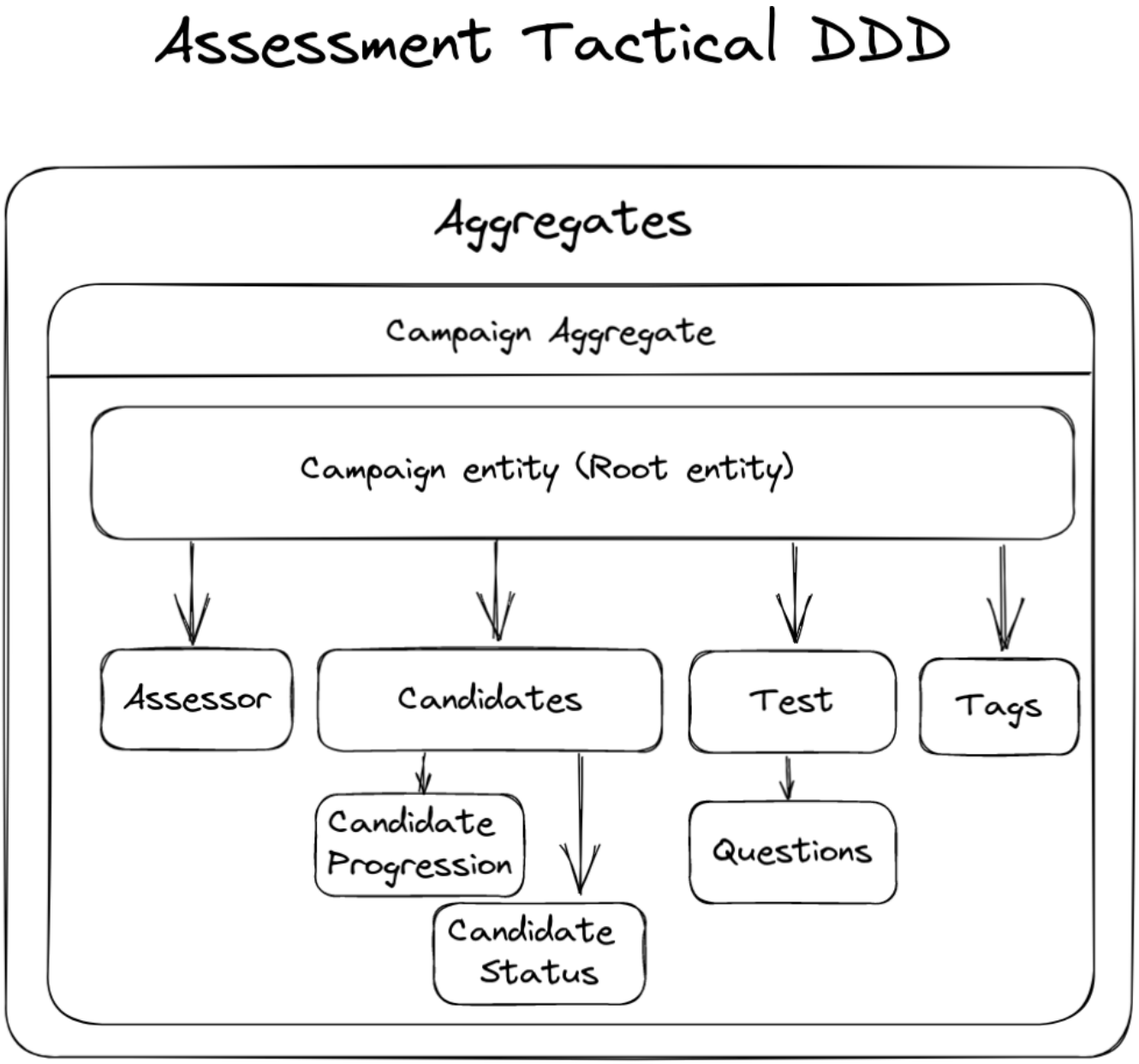


Figure 5. Assessment aggregates

I've defined only one aggregate, the campaign aggregate. Everything in the assessment bounded context is tied to a campaign, with a need for transactional consistency between all of them. The campaign entity will act like the gateway for all the children entities, making sure to take appropriate actions for each modifications and rolling back failed actions. This is a big aggregate and we will see how to deal with it later on. This is not a problem though, you have to accept that some transactional boundaries spans around a lot in your context. You might want to identify which of your dependencies forces you into this situation, and if there is something to be rethought, but for our assessment context, it's fine, not that big, and I don't think I misidentified links between my entities.

The content context

Finally, our final context. We have taken a look at all our bounded contexts but one: the content context. Let's dive right into it.

Entities, values objects and domain services:

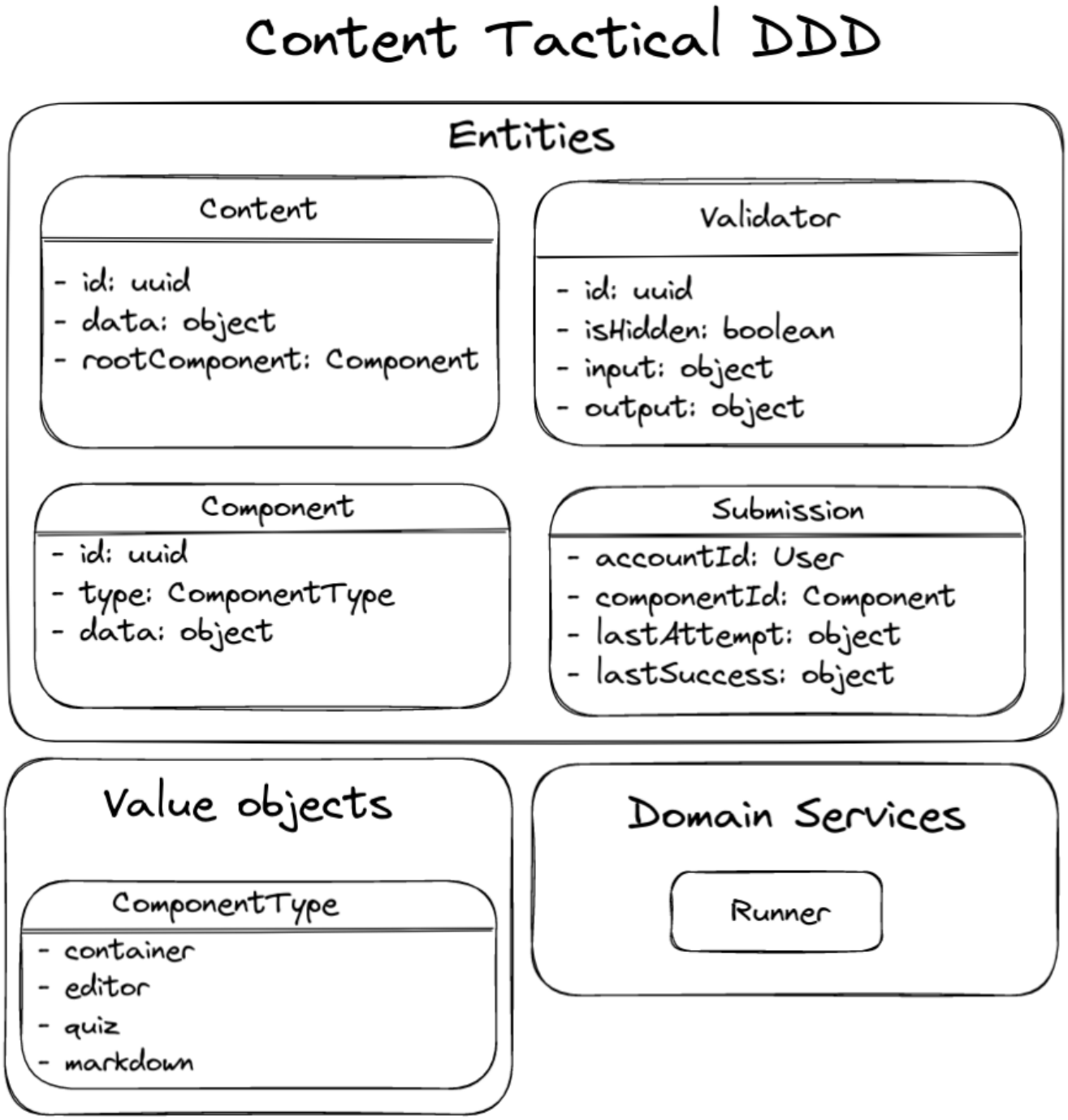


Figure 6. Content entities

I've defined 4 different entities. Let's start by talking about content.

Content is an entity we found in both the practice context (with its own content entity) and in the assessment context (the question entity). In this context, content does not mean the same thing as in the previous two contexts. Here, we care about how it works, how it is going to be displayed, and we don't care about the name it or the type it is given in other contexts.

I've defined a domain service: runner. It will be responsible for taking executable code, and running it, returning the results. It does not do any validation, it just executes code. Since this is at the core of the Polycode application, and a topic on its own, I will not dive into details about its implementation and inner-working here.

Let's now look at the aggregates I identified:

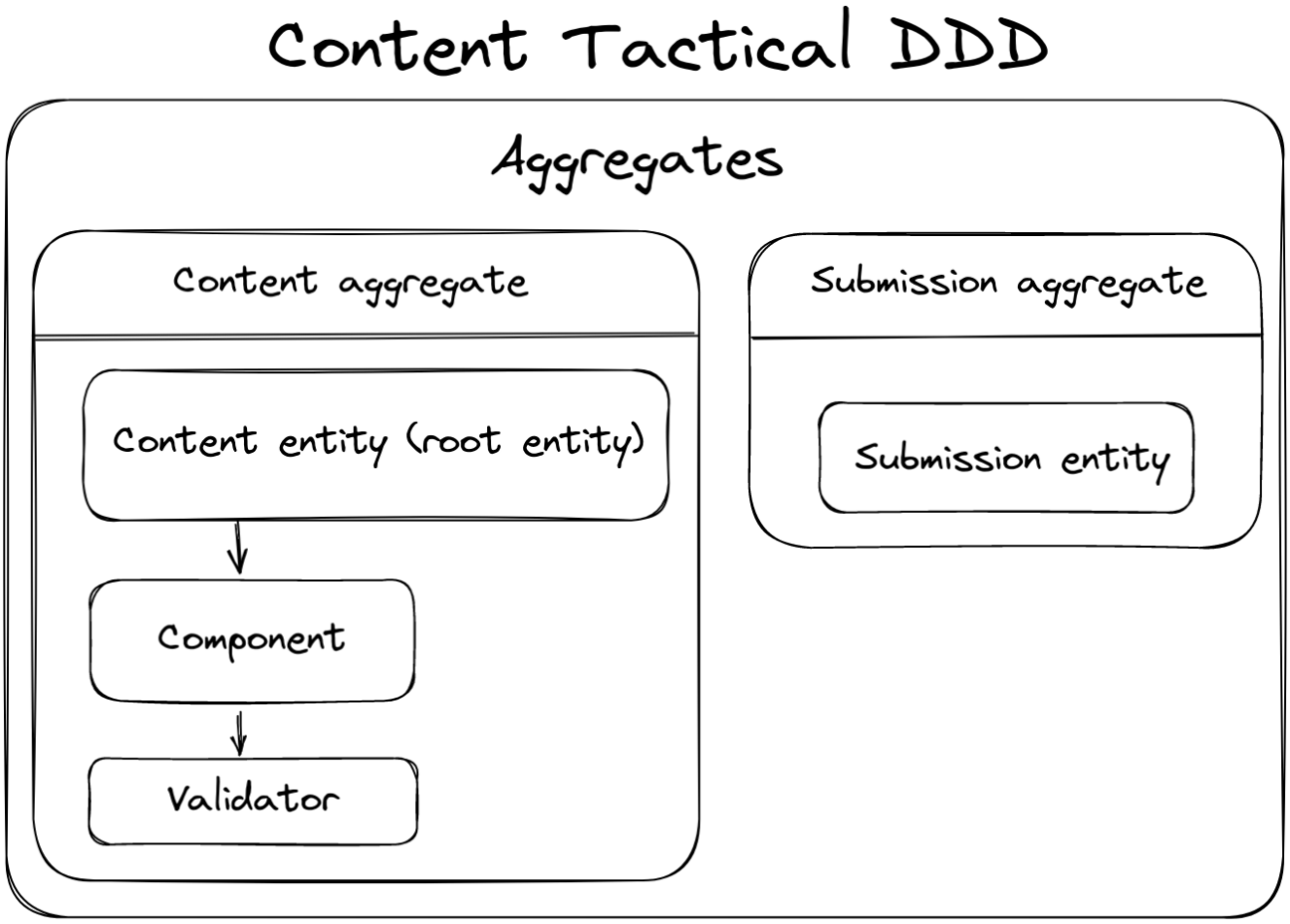


Figure 7. Content aggregates

I've defined two aggregates:

- The submission aggregate, which has the submission entity at its root. It is linked to a component, but it is not in the same aggregate as components. This is because, as seen before, we don't need consistency here, and we don't even want it. If the component were to be deleted, we want to keep track of all submissions that were linked to it, since users actually submitted something. Deleting a component doesn't delete what the user did. However, this means that we have no guarantee of the existence of the component when parsing submissions, neither do we know if the component was updated since the submission.
- The content aggregate. Here, we need consistency. Indeed, when modifying or deleting a component, we want to update all contents and components accordingly. We don't want to have contents with ghost components, and we must ensure this is not the case. Same things we components and validators. Validators are inherently linked to a component, and we need strong consistency, and check for business logic. An exercise must always have a validator linked to it, but a lesson should not. This consistency needs to be respected.

Microservices

We have now defined a domain model for each of our bounded contexts, and have a clear understanding of where our boundaries are. We now need to turn them into microservices.

From domain model to microservices

This section is heavily inspired of Microsoft's documentation on the subject.

The main criteria we are looking for in our architecture are:

- Each microservice has a single responsibility
- Microservices should not be too chatty
- A microservice should not be too big for a small team
- No hard dependencies on one another. A microservice does not rely on another to be deployed and to work effectively.
- They can evolve independently
- You respect data consistency when there is a need for one
- A microservice should not spread over your bounded contexts.

Let's apply the previous criteria to our Polycode application. I will be starting by going over each bounded contexts, and we will talk about inter-contexts communication later.

The schemas you will now see obey the following rules:

- Rectangle shape: Actual microservice
- Diamond shape: Message queue of some form
- Dotted arrow: Asynchronous listener

Account microservices

Let's discuss how I mapped out my account microservices:

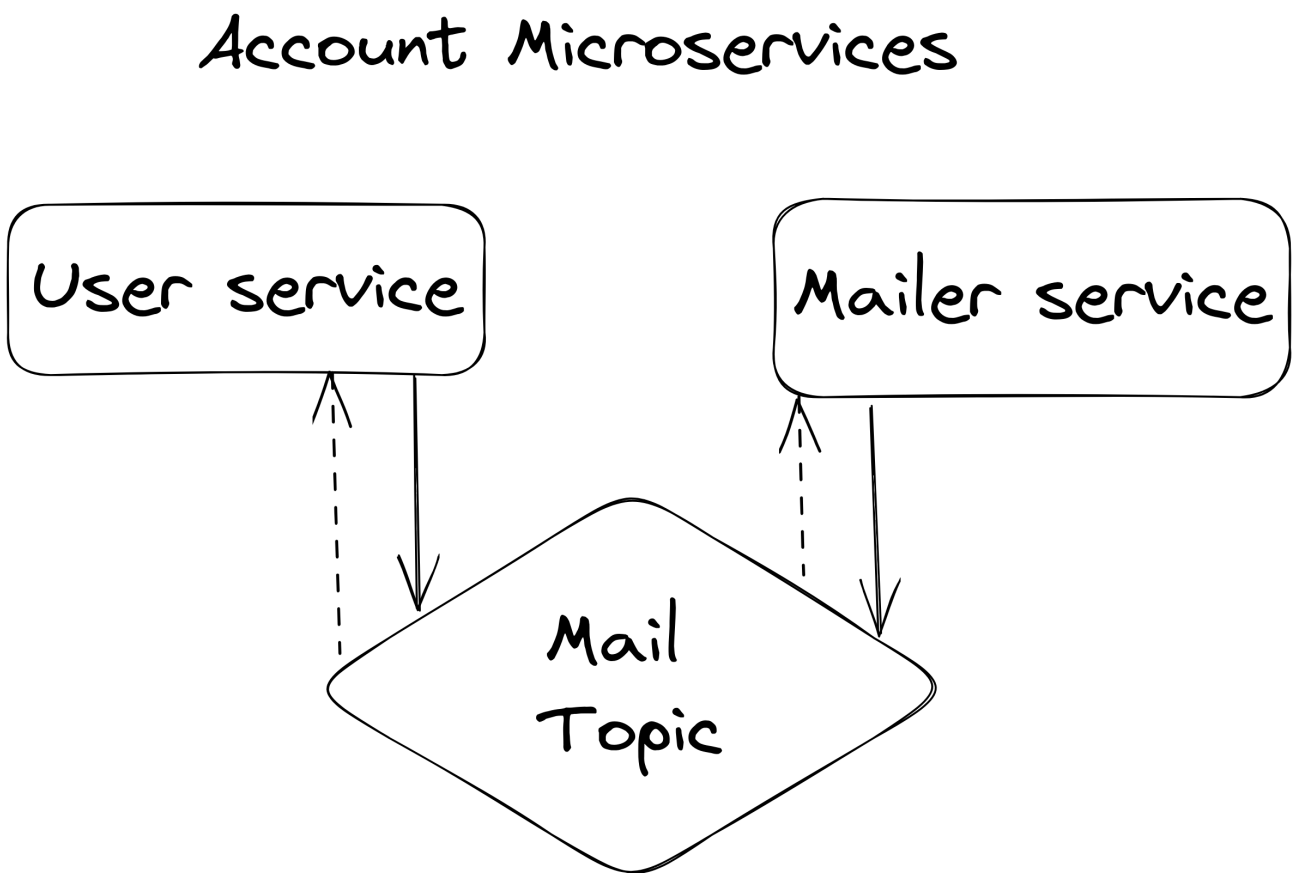


Figure 8. Account microservices

It is very simple and straightforward as you can see, since the domain model is very simple too. There are still some decisions that are not that obvious that were taken:

- I did not divide the mail entity within its own service. I think the cost of managing consistency between the two services overwhelms the benefits of splitting it in to microservices. The user microservice still has a single responsibility, which is to manage the users. Emails are part of this responsibility. The microservice is still sufficiently small to be handled by a small team.

Practice microservices

We identified 3 aggregates within our practice microservices: the user aggregate, the module aggregate and the item aggregate. I started by mapping each of them to one microservice, since this is where our transactional boundaries were identified. For the module and item microservices, I've estimated that they are small enough, and that they doesn't need to be divided. However, for our user microservice, which handled the user aggregates, encompassing user progression, teams and purchases entities, I think it is wiser to delegate some of the functionalities to some other microservices. Indeed, most of those entities are actually quite independent from each other, but also conveys a lot of business logic and intricacies with them.

Let's think about what are the trade-offs we make if we divides each of the sub-entities of the user aggregate in their own domain-driven microservices:

- Purchase entity:
  - Business logic closely related to the user entity, creating a new microservice for it might carry a burden of maintenance and communication across multiple teams.

    - Technically simple (more or less just a CRUD). This makes it hard to justify spinning up a new runtime and new containers for this little of code. This might make more sense if we are edging towards serverless computing, but this is not our case neither within the scope of this paper.
    - Needs very strong consistency with the user entity when updated.
    - Not chatty at all with the user entity. Making a purchase is a rare event. The purchase microservice would not rely on the user microservice for any of its operations. The user microservice would rely on the purchase microservice only when mutating purchases of an user.
- Team entity:
    - Business logic related but not entirely tied to the user entity.
    - Technically pretty straight-forward, but has a significant amount of business logic to implement, with a growing list of features.
    - Not too chatty with the user entity. Mutating a team is a relatively rare event.
- User-progression entity
    - Business logic closely related to the user entity, creating a new microservice for it might carry a burden of maintenance and communication across multiple teams.
    - Technically simple (more or less just a CRUD)
    - Needs strong consistency. We don't want the user to complete something without crediting the correct number of polypoints to its account.
    - Not too chatty. Only chats when an user finished a content, which is not a rare event by itself, but I would argue it is still not on an order of magnitude high enough to become worrisome.

I didn't choose the purchase entity because I think it's too small and its logic integrates well enough with the user's business logic to go into the same microservice. The same thing goes for the user-progression entity.

However, I think creating a microservice responsible for handling team's business logic makes much more sense: it is quite unrelated from the user entity from a business logic point of view, while not being too chatty. Both of the microservices will grow independently and can be worked on by different teams. This will also shrink down the size of the user's microservice by a significant amount, making it easily manageable by a small team, while still integrating the features of the user-progression and the purchase entity.

To make sure that we stay consistent in our state, I would recommend implementing the Saga pattern between the two of them. To keep them loosely coupled.

Here's the final diagram for our practice context:

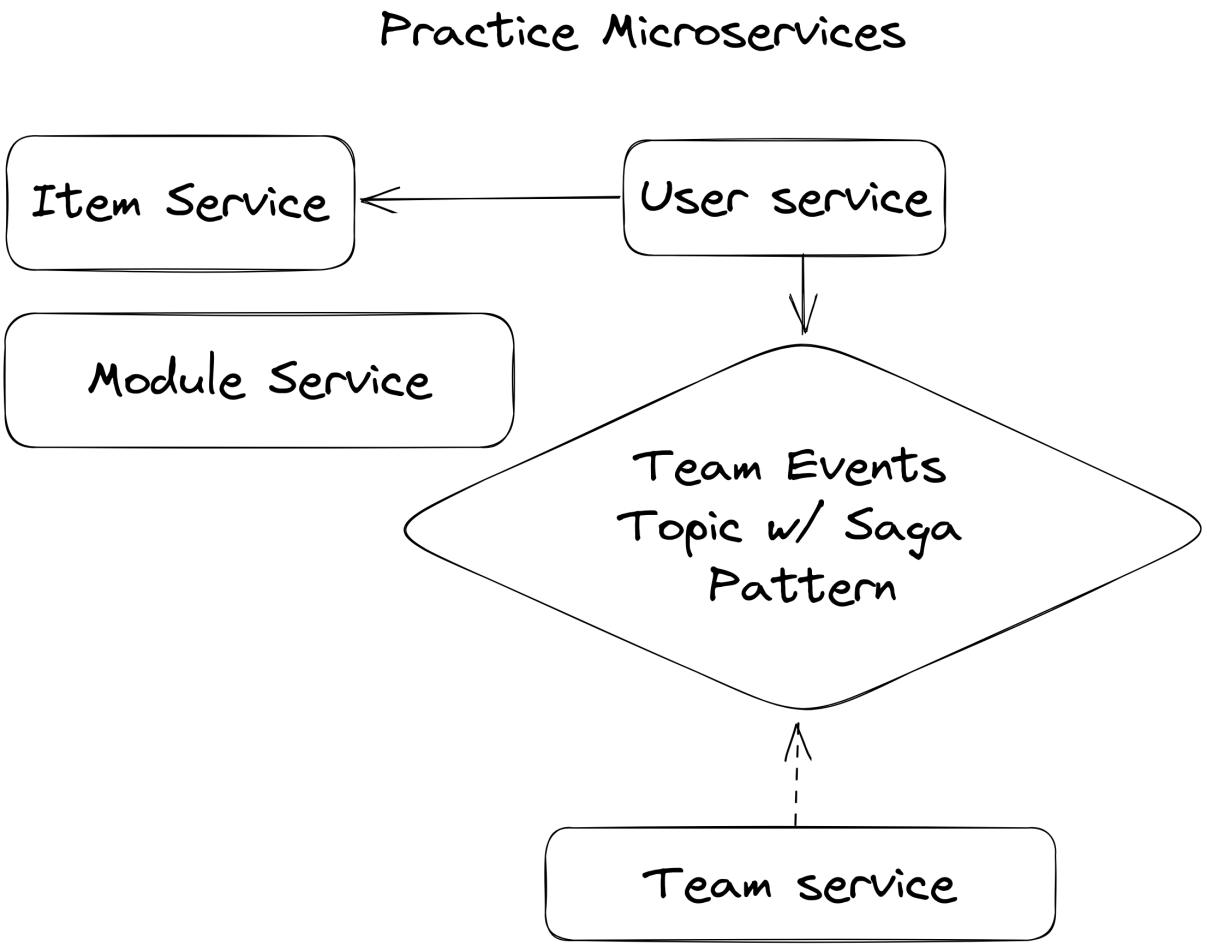


Figure 9. Practice microservices

Assessment microservices

As always, let's start with the aggregates identified in our domain model. In the assessment context, we found only one aggregate, the campaign aggregate, which encapsulates all our entities within our domain. As you might have realized, this is way to big for a small team to handle, and we need to identify where we can divide our microservice.

I'm going to keep the assessor and the tags entity within the campaign microservice, simply because they are way too small to be separated in their own service. We are left with the candidates and the test entities. Both are great candidates, since they handle logic that are quite decoupled from the campaign itself. This is why I've decided to create a microservice for both of them. Here's the diagram:

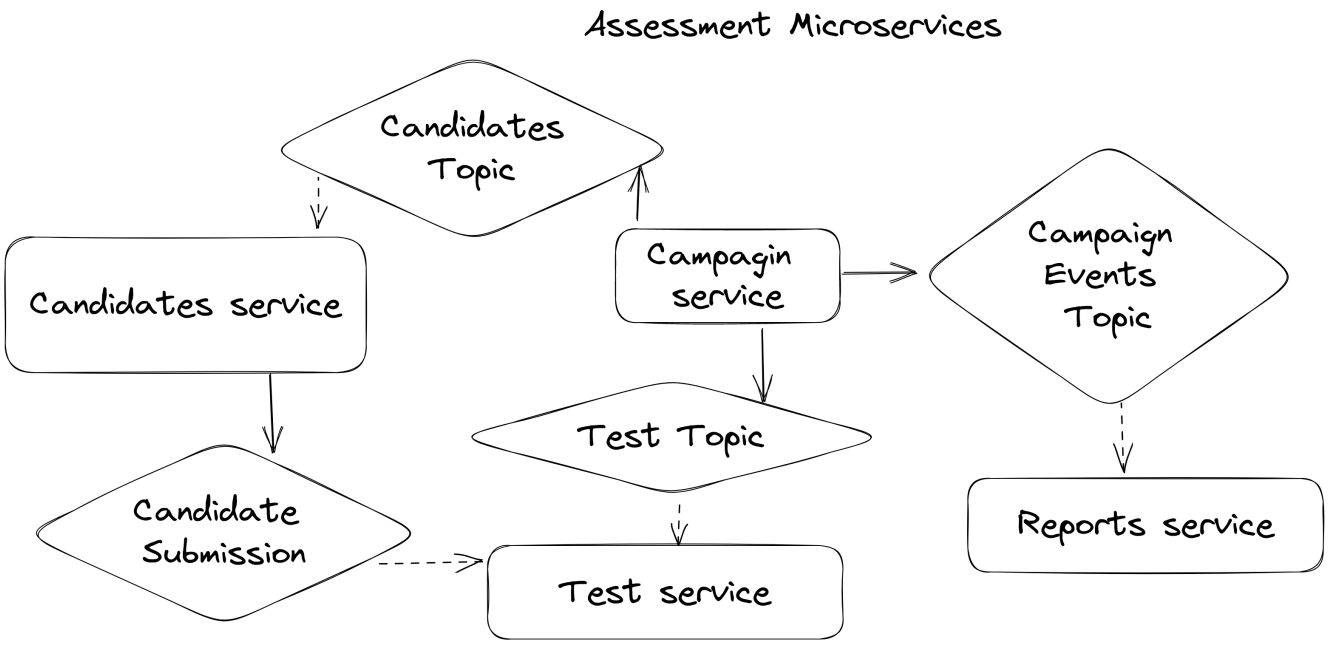


Figure 10. Assessment microservices

We also added our reports domain service as a microservice. It will be responsible for generating reports linked to a campaign. We are left with microservices that are small enough in size, but not too small. They are not too chatty, are not tightly coupled and can evolve independently.

Content microservices

Following the same logic as with the previous domains, we start from aggregates and we trim down each one of them, following some common sense and technical and business requirements:

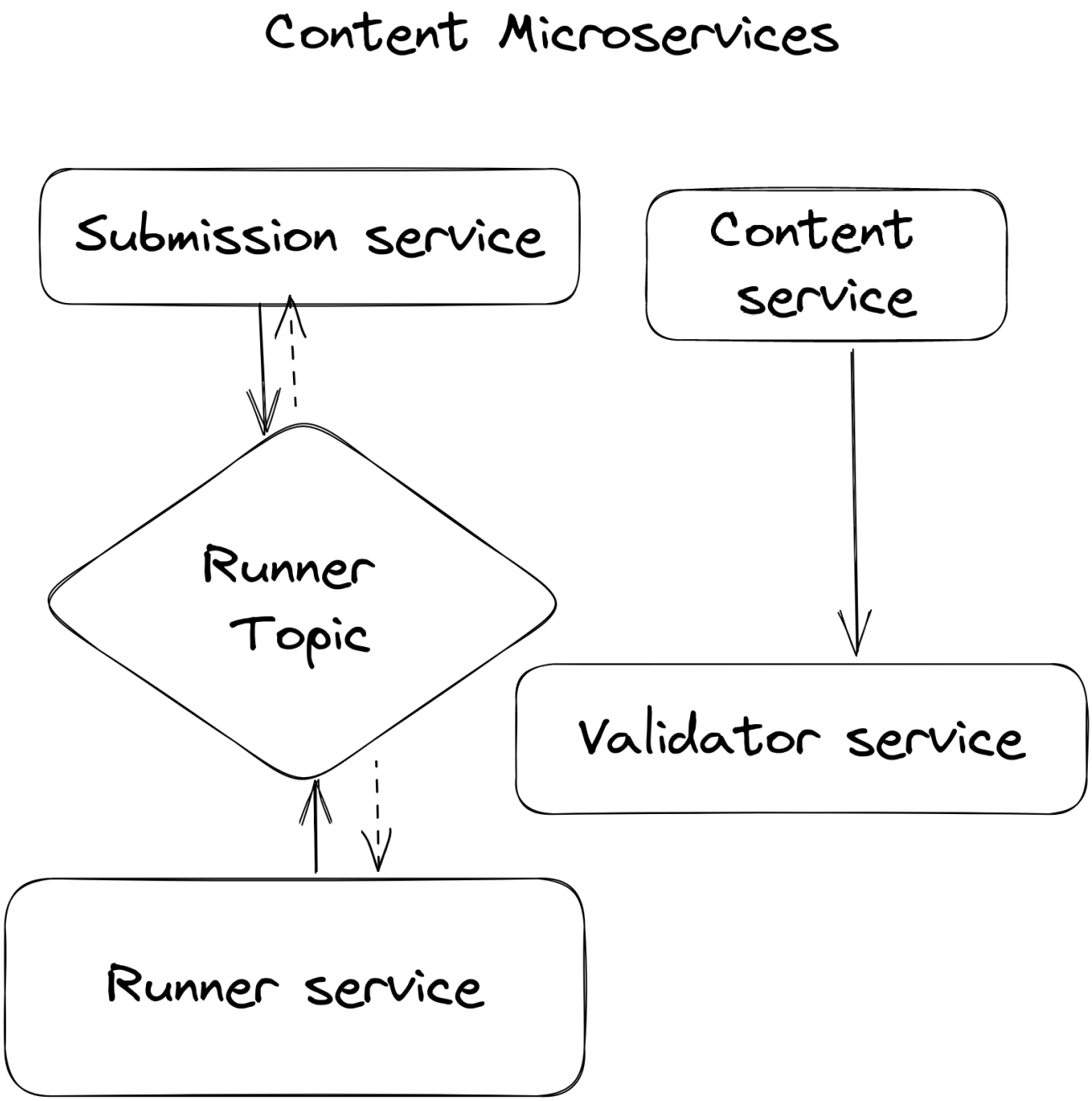


Figure 11. Content microservices

We can, and should, create a separate microservice for the validators. Validators might look like a simple CRUD at first glance, but actually is way more complicated than that, since it can take multiple shapes depending of the type of component being validated. I would argue that most of the complexity in this aggregate is within this entity. It is also separated enough for another team to work on, although a good communication between the teams responsible of creating the content microservice and validator microservice will be required.

Just like before, we must not forget our runner domain service, which will become a microservice by itself. A lot of complexity is hidden here. To dive deeper on the subject, please refer to the corresponding section in my previous paper.

Putting it all together

Now that we have defined all our microservices within each of our bounded contexts, let's take a

look at the greater picture of our architecture:

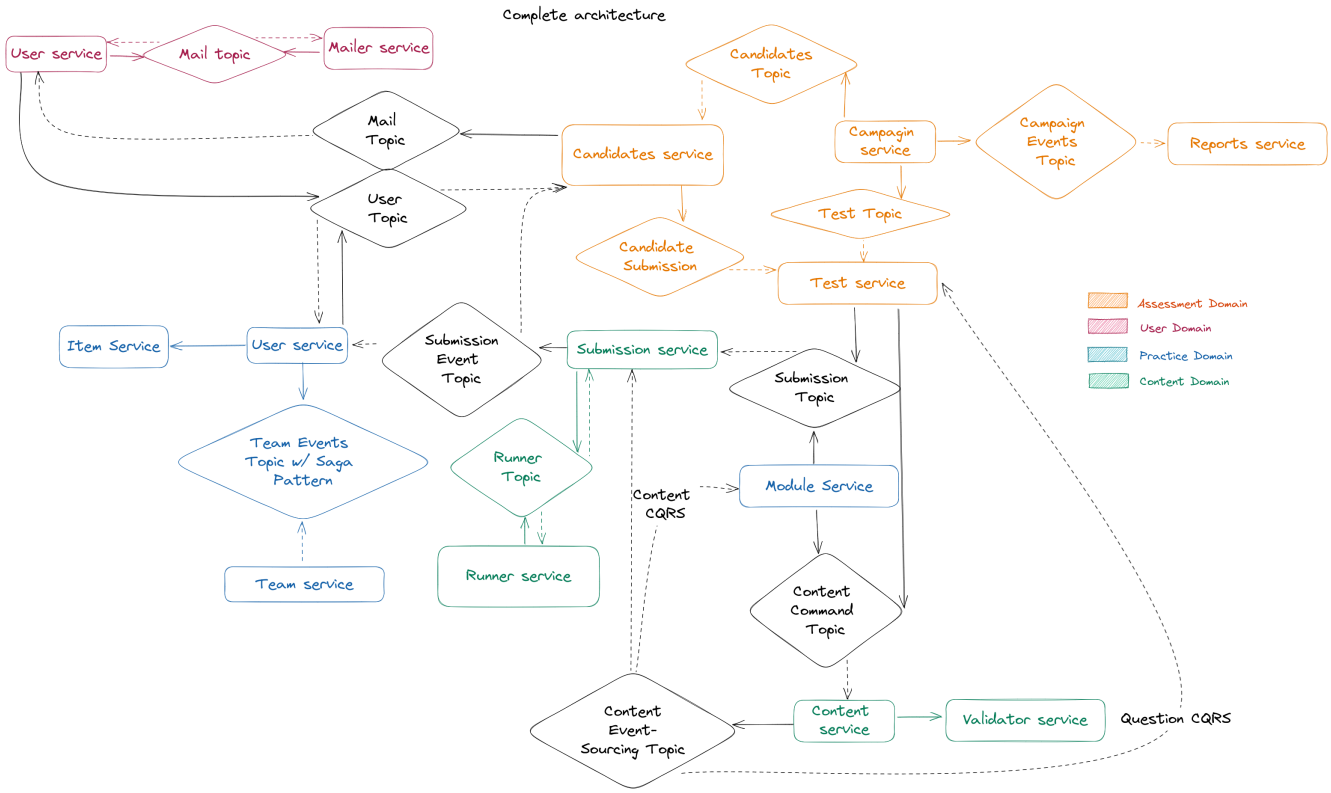


Figure 12. Complete architecture

Notable architecture decisions: * All cross-domain communication are asynchronous. * One domain should not rely on another to work. * One domain should not have to worry about who's consuming its events. * The User context sends events when a user in changed. This allows other domains to act within their boundaries to take the actions it deems needed. * CQRS Pattern on contents. When editing a content, commands are sent to the content microservice (within the content domain), which validates the command and create the appropriate event, sent into a topic. Both the module and the question microservices listens to these events, and update their copy of the content. * On a submission, both in the assessment and practice domain, the submission is first validated by the their respective user microservice. This is how we can check if a question was answered in time, for example.