WA2325 Solution Architecture (SA) Practitioner's Guide (Extended)

Solution book

Web Age Solutions Inc.

Table of Contents

Lab 1 - Defining the Problem

The sample solutions make use of the hypothetical Acme Paint scenario described in the appendix at the end of the Lab Book – no further solution is provided for this lab.

Part 1 - Preparation for using the provided example

If your group has chosen to use the provided example,

- Individually, read Appendix L1
- As a group, discuss your understanding of the problem

Your instructor is available if you have any questions or would like clarification.

Part 2 - Preparation for using a system of your own

If your group has chosen to use a system of your own

- As a group, discuss the problem that you would like to address
- Write down, at a high level, key aspects of the problem; make sure you capture
  - The business problem (what is the motivation)
  - How the business problem will be addressed (what the solution is, in business terms - stick to a high level description but call out the key features of the solution if applicable)

Use the provided example as a guide to what you may want to think about; it is not necessary to try to write down a problem with the level of detail used in the example!

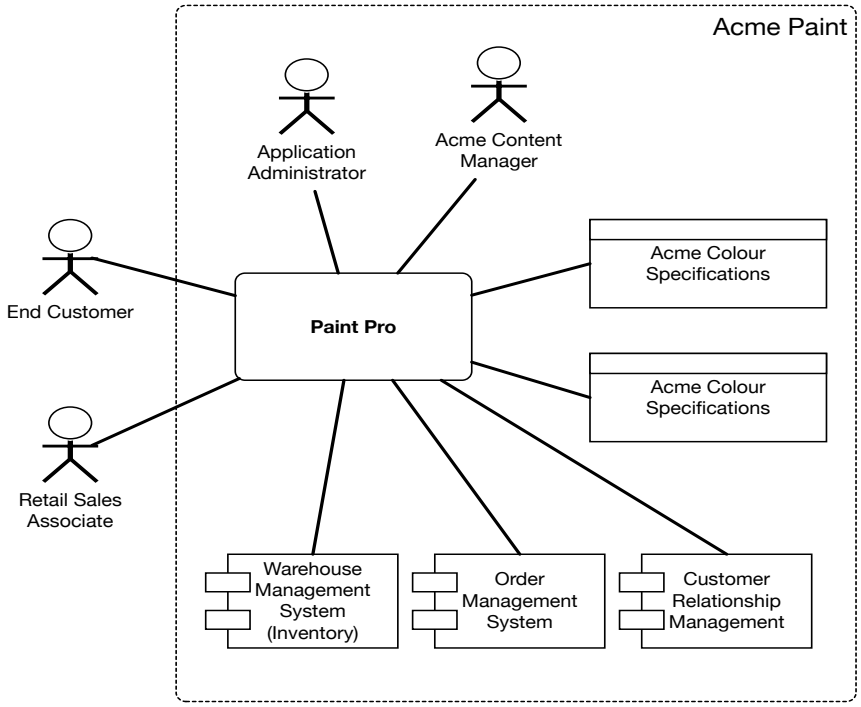
Part 3 - Wrap-up

Once you are satisfied that you understand the problem you are going to address, you are done. If you have chosen to make up your own problem to address, you will be asked to briefly describe the problem at the take-up of the next lab.

Lab 2 - Defining the Boundary

Part 1 - Context Diagram

The problem statement gives some information about the solution context, but we are left to read between the lines for the details. One possible context diagram for the solution described is this:



In a real solution architecture initiative, we would discover the context - through documentation, through discussion, and through common sense. What do we mean by "common sense?" There are often things that we can recognize will be needed, even if we do not know wha they are yet. This is part of the value of the early work to develop a context diagram.

For example, if we know that our solution needs to keep track of customers, and the organization does not have a customer master file or CRM system or something similar, then it indicates that some means of keeping customer information is required. A solution architect would likely suggest that this is a fundamental function that companies do, so it probably should be outside the solution. That reasoning could lead to an important discussion about the organization's enterprise vision and roadmap.

A note about notation: Traditionally the system is a box in the centre, and actors are stick figures around the edges. It is OK to use whatever notation makes sense to you and your audience. The author likes to use

- Stick figures for human actors
- Boxes for system actors (other systems)
- Stylized buildings for other organizations (such as a business partner)
- Data store symbols for things that are exclusively used to save or retrieve information(So they are conceptually a data store, regardless of how they actually work)

Part 2 - Actors

Some actors we might expect in our solution include

Actor	Location	Description
Acme content manager	Acme office	Manages content that is made available to retail staff and end customers
Retail store associate	Reseller store	Reseller employee who sells and supports end customers
End customer	Reseller store (1)	Customer who purchases Acme Paint products in a reseller store
Acme administrator	Acme head office	

(1) Notice that we located end customer in the reseller store. That's because from our point of view, interactions with the end customer are all in the reseller store itself. There's an assumption here; we are assuming that Acme Paint is not interacting with the customer outside of the retail store (through a Web site, for example), at least in the context of this system. These details can be important because the will be relied on by designers and implementors to make detailed decisions later.

Part 3 - Interfaces

We generally focus on machine-to-machine interfaces. The interface with the customer management system (CMS) may be described as

External system: CMS

- Source of master data about retailers (who are "customers" to Acme Paint)
- Source and sink of information about end customers (which the solution records in CMS under some circumstances)

Interface description:

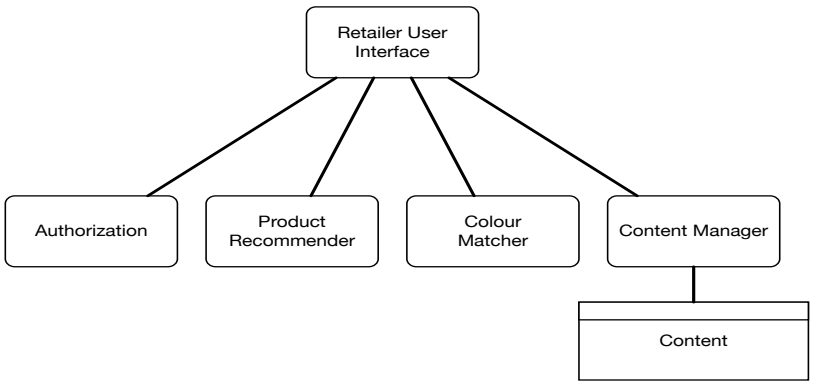
- HTTPS, REST
- Endpoint application is Salesforce
- Endpoint is https://acmepaint.salesforce.com/api/customer
- Interface specification (OpenAPI) is https://interface.salesforce.com/api/specs/customer

(Endpoints are made up.)

In this case, we are making use of a pre-existing, well documented interface. That may not be the case - the solution architect should find as much information about an interface as is practical, and expect that it will evolve over time.

Part 4 - Subsystems

Reading through the problem statement, we can identify a number of likely subsystems. Remember that this is just a starting point; as the solution architecture is developed further, it is likely that that subsystems will be refined - probably more added, sometimes initial subsystems refined.



Lab 3 - Stakeholders, Views, and Viewpoints

Part 1 - Identify stakeholders

There are many potential stakeholders; we won't try to list them all in this exercise, but here are a few; we will expand on some of them later. When identifying stakeholders, it is best to be broad and think of as many as you can, then prune the list afterwards. Just don't go crazy; when you start to have trouble finding new stakeholders, it's probably time to stop.

Retail store sales associates

End customers

Acme Paint users

Business Sponsor

Enterprise Architect

Marketing

Development

Part 2 - Classify stakeholders

Not all stakeholders are considered equally important for the purposes of a solution architecture. Here's a quick evaluation of the stakeholders we have identified:

There are some that are certainly stakeholders, in that they have an interest in the solution, but they are also classic "users" and their needs (should be) covered by functional requirements. For the purposes of the solution architecture and stakeholder analysis, we can probably set them aside:

Retail store sales associates A "user" - No further analysis

End customers A "user" - No further analysis

Acme Paint users A "user" - No further analysis

The rest have varying needs, and are worth thinking about a bit more:

Business Sponsor The person who at the end of the day has put the weight of their position behind the project, and is (usually) also funding the project. A sponsor is typically concerned with the overall project objectives, direction, progress, and cost. The key concern of a sponsor is whether the project will achieve its business

objectives and be worth the investment.

Enterprise Architect The person or role responsible for the overall architecture of the enterprise, who will be typically interested in how the project fits into the overall enterprise and whether it seeks deviations from architectural standards.

Marketing In the case of PaintPro, the application supports marketing efforts through its ability to deliver relevant information to retail sales associates. Marketing would likely be most concerned with what flexibility they have in the materials that can be delivered, how they can ensure that retail sales associates get the messaging that marketing wants, and what mechanisms there are (if any) to get feedback from retail sales associates.

Development Development (here we lump all development together, from design to development to test to deployment) will have technical concerns and will want to have material that supports their understanding and helps them develop the product.

The example of marketing above illustrates why stakeholder identification and analysis can be a valuable activity for solution architects. Notice that we identified a desire of marketing to get feedback from retail sales associates - they want to be able to understand whether their marketing programs are working. This may not have been an identified requirement for the project, but there is a good business reason to consider it. A good solution architect would make a judgement call, based on good business sense, whether to raise this potential new need. If it turned out to be something that the project should consider, it would likely lead to architecturally significant changes. Solution architects should feel comfortable interacting with business, with project management, with sponsors or a steering committee, to raise new concerns.

We can classify stakeholders according to the two-by-two matrix described in the lecture:

Business Sponsor	High power, low interest
Enterprise Architect	High power, high interest (1)
Marketing	Low power, low interest
Development	Low power, high interest

(1) The enterprise architect has high power with respect to architectural standards, but likely low power regarding other aspects of the project. If it's not clear, be conservative and use "high".

(The evaluation of these stakeholders is just illustrative - your stakeholders could very well be evaluated differently.)

Part 3 - Identify some views and viewpoints

Some views that would be appropriate for our stakeholders are

Business Sponsor

- Project charter
- Project budget, timeline
- Context diagram
- High-level requirements (depending on the level of their interest)

(Notice that most of these are not things that a solution architect would typically be responsible for.)

Enterprise Architect

- Context diagram
- Data architecture (conceptual)
- Application architecture (overview)
- Technical architecture (overview)
- Summary of compliance with enterprise standards
- Submission to Architecture Review Board, if applicable
- Detailed documentation on any requested deviations from architectural standards, if applicable

Marketing

- Likely interested in high level requirements related to management of collateral material, delivery of collateral material to retail sales associates and customers, and capabilities for delivering messaging to retail sales associates.

Development

1. Pretty much everything technical

Lab 4 - Architectural Requirements

Part 1 - Identify Quality of Service requirements

Two Quality of Service requirements: Availability, and Deployability

An architecturally significant non-functional requirements: Localizability

(There are others, obviously, but we will explore these three in the solution)

Availability This is a fairly obvious quality of service need for a customer-facing application, but a solution architect should not take it for granted. There is a very significant cost difference between a solution that is available “almost all the time” and one that “never goes down”, and a solution architect needs to consider what is really required by the business. For motivation, think of the difference in business need (and solution cost) between a marketing site for a retailer and an online banking application for a major bank.

Another concern with availability should be whether an application should be available all the time (24/7) or whether there are periods it can be offline. If it has to be continuously available, that has an architecturally significant implication on the application architecture and the technical architecture, which must be able to support rolling updates. If there are periods where the application can be unavailable, releases can be done during those times and the application and technical architecture might be simpler.

For this problem: “The solution shall be available 99.9% of the time between 7am Eastern and 10pm Pacific, every day.”

This leaves a maintenance window between 1am and 7am Eastern time each day; a capability to perform rolling updates would be nice, but is not required.

Note that this is an area where a good solution architect will be thinking about the future; are we reasonably sure that we will never operate in Europe, for example? Are we reasonably sure that the solution will never need to be extended with a customer-facing component? If we’re not sure, then we should consider the possibility that we should architect for continuous availability and rolling deployment now, as that can be painfully hard to retrofit later.

Deployability As the application needs to be deployed in retail locations, potentially a lot of them, and those location will have a wide variety of technology installed, the solution should be easy to deploy. We could define this as “deployment of the application to a retail location should require as little effort and have as few technical dependencies as practical.” This is true, but rather non-specific; at this point in the development of the solution architecture, we may not want to be more specific until we have looked at tactics

for achieving the requirement.

Localizability The solution will be used all over North America, so will need to be able to present interfaces in several languages, and in both metric and English units.

Part 2 - Brainstorm tactics for a quality of service requirement

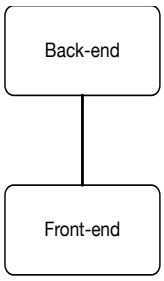
Availability

So, we should quickly realize that any portion of the application inside the retailer is pretty much out of our control. If we deploy on retailer hardware, we are subject to whatever availability they have.

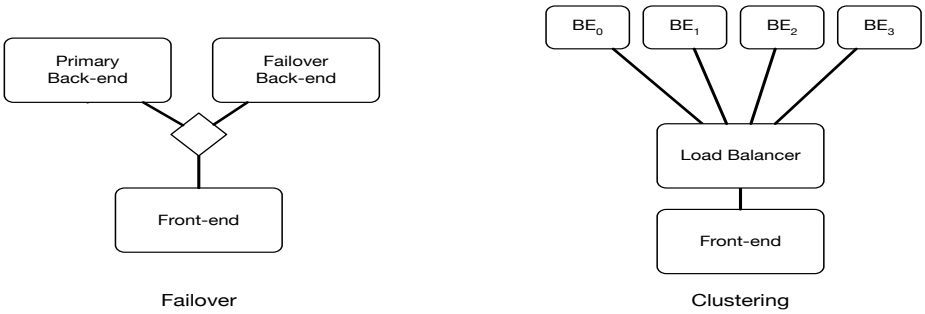
(That may not be bad - outages are isolated to a single location, and if systems are down that location likely has bigger problems than not being able to run our application.)

What we can do for availability is be flexible on what resources we need of the retail location - if all we need is a browser, then in principle the retailer could just switch to another device if needed.

Notice that we're starting to implicitly think about some fundamental things about what the solution is, so let's start sketching it out. If feels like there's going to be a back-end and a front-end, with the front-end (the user interface) at the retailer:



What about the back-end, then? We know we need some redundancy to meet an availability requirement. Two basic approaches are failover and clustering:

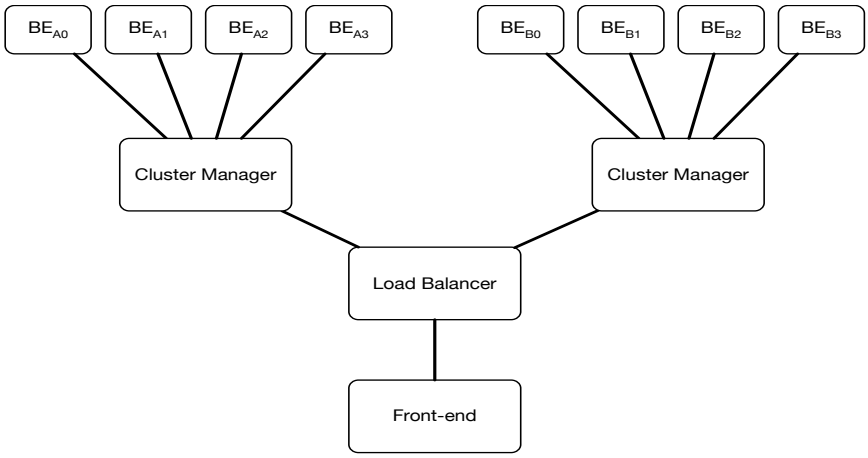


(There are other approaches, but we'll stick to these for the purposes of the solution.)

Failover (as drawn here) is a well-known, trusted technique, and fits well in a classic datacenter environment. Acme Paint probably has a bunch of enterprise applications that work this way.

Clustering is a little more challenging - there are more parts, and how do we keep the data in all theses back-end instances consistent? We didn't have to worry about that with failover, as there was only one instance active at at time.

Additionally, clustering as drawn here misses something important - what if we loose a whole data centre, or communications to a data centre? Ooops, no cluster! So we really need to think about two kinds of redundancy - redundant instances in a cluster, and redundancy across data centres:



That seems pretty complicated. Failover is looking pretty good.

There are QoS differences between the two (or three, depending on how you look at it)

approaches, but we won't go into that here - let's just stay that if we looked further, we would find redundant clusters are "better" then failover in a number of different ways, but are more complicated and require more infrastructure, so there's a tradeoff.

Perhaps another QoS requirement can help us decide.

(If you're really paying attention, you may notice that clusters make rolling deployment a lot easier - so if we're leaning toward needing rolling deployment, that's a strong push in the way of clustering instead of failover. We point this out to show how QoS requirements are often intertwined.)

Deployability

Deployability will be a significant concern for Acme Paints. The Paint Pro app will get deployed in retail locations that Acme as no control over, they will be scattered all around multiple countries, and they will be (hopefully) added in large numbers fairly regularly as Acme's retail partnerships grow.

Deployment considers lead to an obvious conclusion - the front-end should be web-based, so that there really isn't any deployment. Problem solved!

(We should validate that this is a viable option - there may be something we have not considered - but we're going to go with this decision for now.)

That decision has a whole bunch of implications, which we won't go into now but will run into later.

What about deployability for the back-end portion? Well, deployability is pretty straightforward for a failover pattern - we've been doing enterprise apps this way for years.

But deployments in the failover pattern are usually pretty large - we deploy an instance, and it's not easy to scale up or scale down. We probably want to be able to scale easily because of all those retailers that we hope to add over time.

Deployment in a cluster model is much more granular - the application does not have to be one large deployable unit, it could be a number of different deployment units each deployable to the cluster individually. Furthermore, if we need more capacity, we deploy another instance of the back-end component(s) and add it to the cluster.

We could explore a number of ways to do this, but for the sake of avoiding going down a rabbit-hole, let's jump to a tactic that has a number of nice properties - deployment with containers.

Containers are like light-weight virtual machines - you can think of them as a virtual machine without the overpaying system. Containers are not burdened by the overhead of having their own copy fo the operating system but they isolate processes inside the container as if those processes are running on a virtual machine all on their own.

Modern container tools like Docker have a whole ecosystem of development and runtime tools, and Docker containers can be run in containers by runtime orchestration tools like

Kubernetes.

(As an aside, one would not generally expect a solution architect to use tools like Docker or Kubernetes hands-on, but one would expect a that a solution architect working in an enterprise would know what they can do, their pros and cons, and the implications on the architecture. Knowing what choices are available, and which ones are right for a particular solution, is part of a solution architects role.)

We should point out one thing that's going on here - our choices for satisfying an architectural requirements are going to have a significant influence a little later when we consider other aspects of the architecture. For example, it's good practice when using containers to have a container for each major subsystem of an application. We avoid putting too much functionality into a single container - just as we try to avoid building monoliths.

Additionally, containers communicate with the outside world, including other containers, using a networking model. So, the application architecture of a containerized application is a collection of loosely coupled services (each container can be thought of as a service). This comes with its own advantages (and complications).

Sometimes, that complexity isn't worth it - we may work through other parts of the architecture and decide "that was the wrong choice" and go back and change our minds. In general, we would like to have solutions that are as simple as practical, and complexity should be there for a reason.

The take-home point is that architectural decisions and their implications are frequently interrelated, and a seemingly straightforward choice at one point of the architecture process can lead to significant implications elsewhere.

In the real world (outside of a lab exercise), we do our best to see the big picture, and we are OK with revising a previous decision as we learn more.

Localizability

Localizability is actually fairly straightforward to deal with, now that we know we have a web-based front end.

- In the application architecture, note that the design should use a user interface framework that supports externalized localization ("externalized" in the sense that the localization information is stored in a file or some such data store external from the application itself). The fact that there is an external data store for localization information is architecturally significant.
- In the application architecture, note that unit conversion will be required and hint that it should be implemented once, in a class or something similar. Unit conversion is a SMOP ("simply a matter of programming") and would usually be left to the discretion of the designer.
- Ensure that the user profile tracks localization preferences per user. This is presumably a functional requirement, and is straightforward, but we would expect

that it shows up the the data model for a user later in the solution architecture.

- The fact that there is a user profile also implies that there is a user manager; if we have not thought about that yet, we add it to the emerging architecture. See how one requirement leads to a recognition of other things that are needed?

Part 3 - Assess the architectural impact of your tactics

For the two architectural requirements that you addressed, what tactics are the best options for your particular solution? Why?

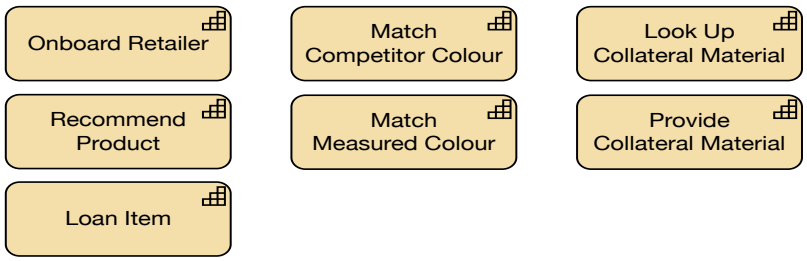
Are there trade-offs that you may have to think about? Two different approaches might have different advantages, or one approach might improve one quality of service while having some other detrimental effect.

(Cost is frequently a limiting factor in determining which tactics are practical for a given situation. Cost can be dollars, but effort is another form of cost; some tactics might be impractical simply because there is not time to implement them.)

Lab 5 - Business and Data Architecture

Part 1 - Capabilities

We can describe capabilities at many different levels of detail, depending on what is appropriate for our needs at the time. At a high level, some of the business capabilities our solution must address include



(The notation used here is ArchiMate - use whatever makes sense for your audience.)

There is some discretion about what capabilities to list and what to leave off. For example, we could list “Manage Users”. The author chose not to, feeling that this is not really a business capability per se and should be captured as a functional requirement anyway.

There is also some judgement about capability names. For example, why “Loan Item” when we know the thing being loaned is a colorimeter - why not “Loan Colorimeter”? Well, it’s a judgement call, but the author felt that it was fairly certain that the capability was loaning things, and the fact that the thing happens to be a colorimeter is incidental. If some specific and some general names appear equally applicable, we feel picking the more general is almost always the right thing to do.

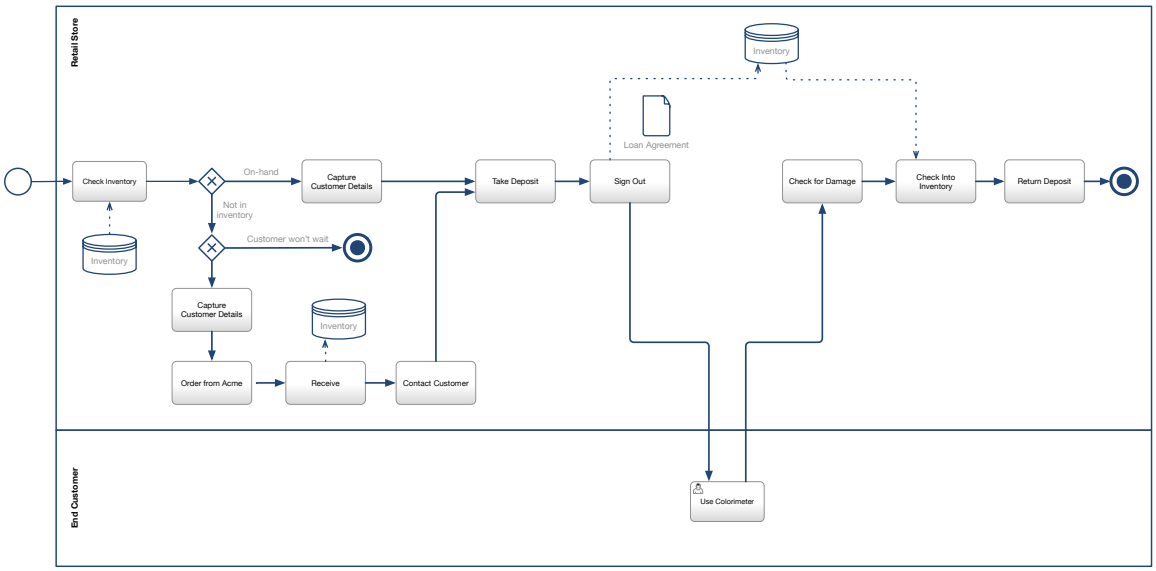
We also use judgement in how finely we break up a capability. We could have, for example, said “Look up and provide collateral material” - why was it broken in two? We feel, intuitively at this point, that these are actually two different things, and we have a suspicion that there may be use cases where we provide collateral material without looking it up first.

With this amount of judgement and intuition guiding us, we should be OK to recognize that our capability descriptions evolve over time as our understanding of the problem develops.

Part 2 - Business process

We look at the process of loaning a colorimeter to a customer. There are many ways one might do this - we make some assumptions in this process diagram. If your solution is different, it’s likely also “correct”, in the sense that it’s a possible solution that would meet requirements. In a real solution architecture development, we would likely develop

a pro forma process then refine it.



Note that the notation used for this process diagram is BPMN (Business Process Modeling Notation)

Notice that while the process is relatively straightforward, there are a number of exceptions and alternate paths - this is typical of real-world business processes.

How much detail we go into as a solution architect should be driven by what our needs are - and that is driven by our desire to discovering things that are architecturally significant. How do we know? Judgement and experience, and recognizing that we may get it wrong and need to come back and refresh our work later.

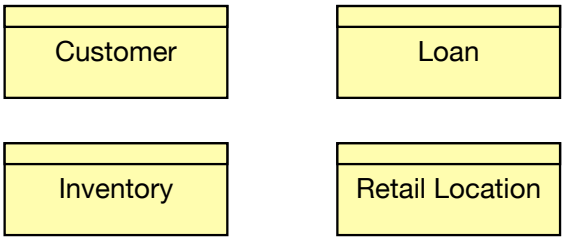
Pro tip Pay attention to the business information used by a business process. Processes often require that we keep track of state, which implies using business information. Not only do we want to pay attention to business information, we want to watch what parts of the process use that information. These observations often lead to identification of data stores and data flow later in the application architecture.

For example, consider the process steps where we take a deposit and later return the deposit. It should be obvious that we need to keep track of the deposit. It may not be obvious (hopefully we notice it in time) that if we want to return a deposit to the original method of payment, we will need to keep track of some identifier from the credit card processor when the deposit was taken and use that identifier later when we process the refund (but not too much later - credit card processors limit how long an identifier is valid for).

See how paying attention to business information can expose a bunch of new considerations?

Part 3 - Business information

A few of the pieces of business information we identify from our business process are

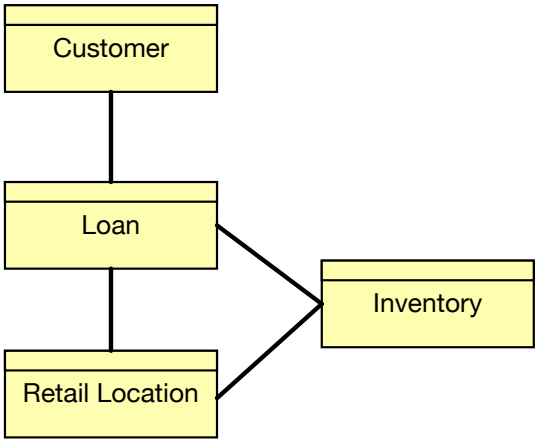


Are there trade-offs that you may have to think about? Two different approaches might have different advantages, or one approach might improve one quality of service while having some other detrimental effect.

(Cost is frequently a limiting factor in determining which tactics are practical for a given situation. Cost can be dollars, but effort is another form of cost; some tactics might be impractical simply because there is not time to implement them.)

Part 4 - Conceptual data model

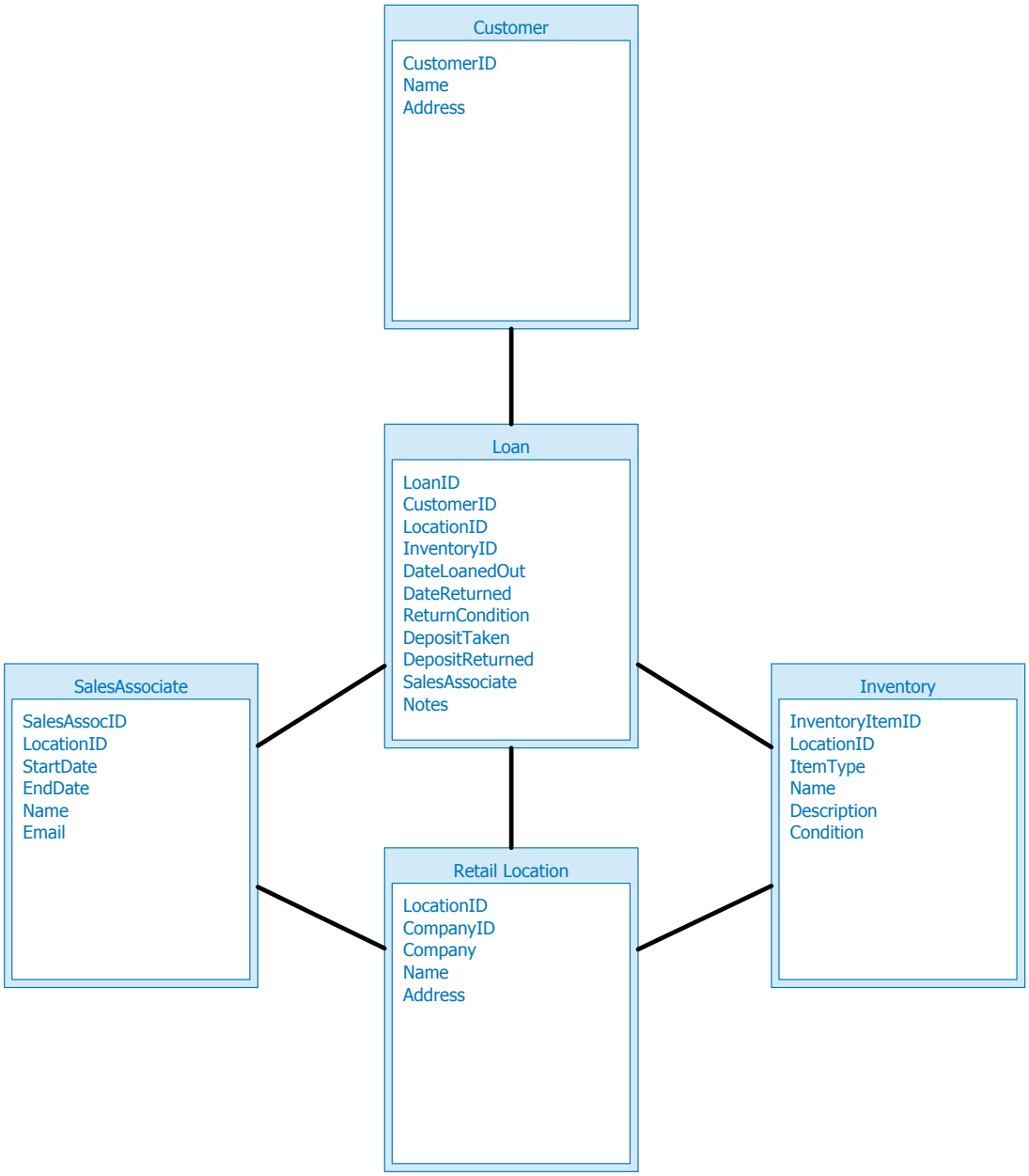
We can start to formalize our business information in a conceptual data model:



(There is a 1:1 correspondence between business information and entities - that's not always the case, and usually as we develop our data model we find it's much richer than the business information it is related to. It just so happens the two are similar at this point in the analysis.)

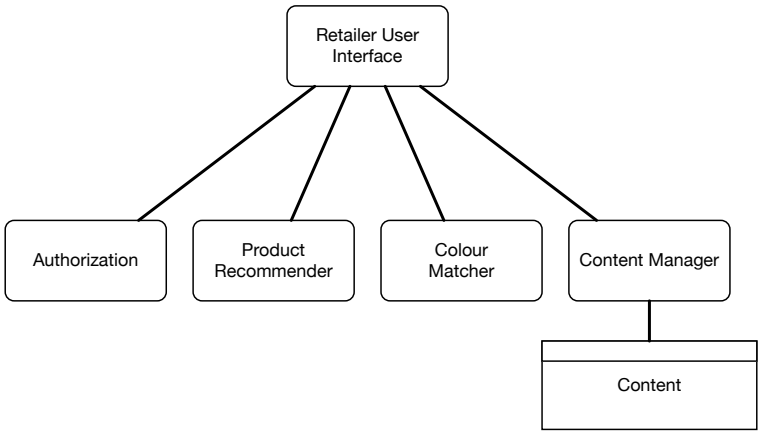
Part 5 - Logical data model

At this point in the development of a solution architecture, we likely don't have a lot of detail for a logical data model, but we can start to record simple attributes and relationships. This logical data model would be expected to evolve quite a bit as the solution architecture develops, but is an important starting point and keeps track of what we think we know about our entities.



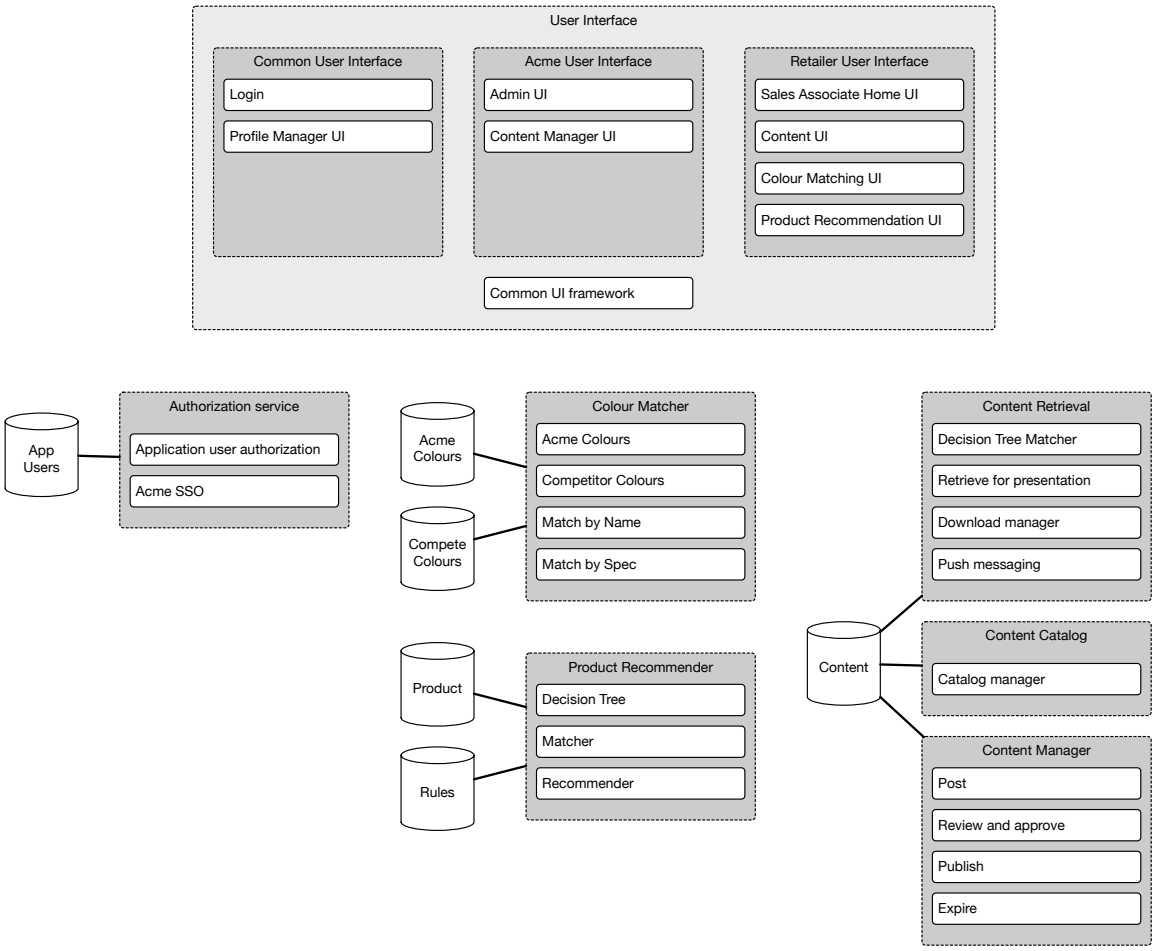
Lab 6 - Application Architecture

We start by repeating the subsystems that we identified back in Lab 2 - that's a good starting point:



Now, we think about those subsystems; some are pretty big, so we break a few of them up into somewhat smaller bits. We may bring some of the bits back together later; at this point we are looking for what feel like good ways to factor the application into parts.

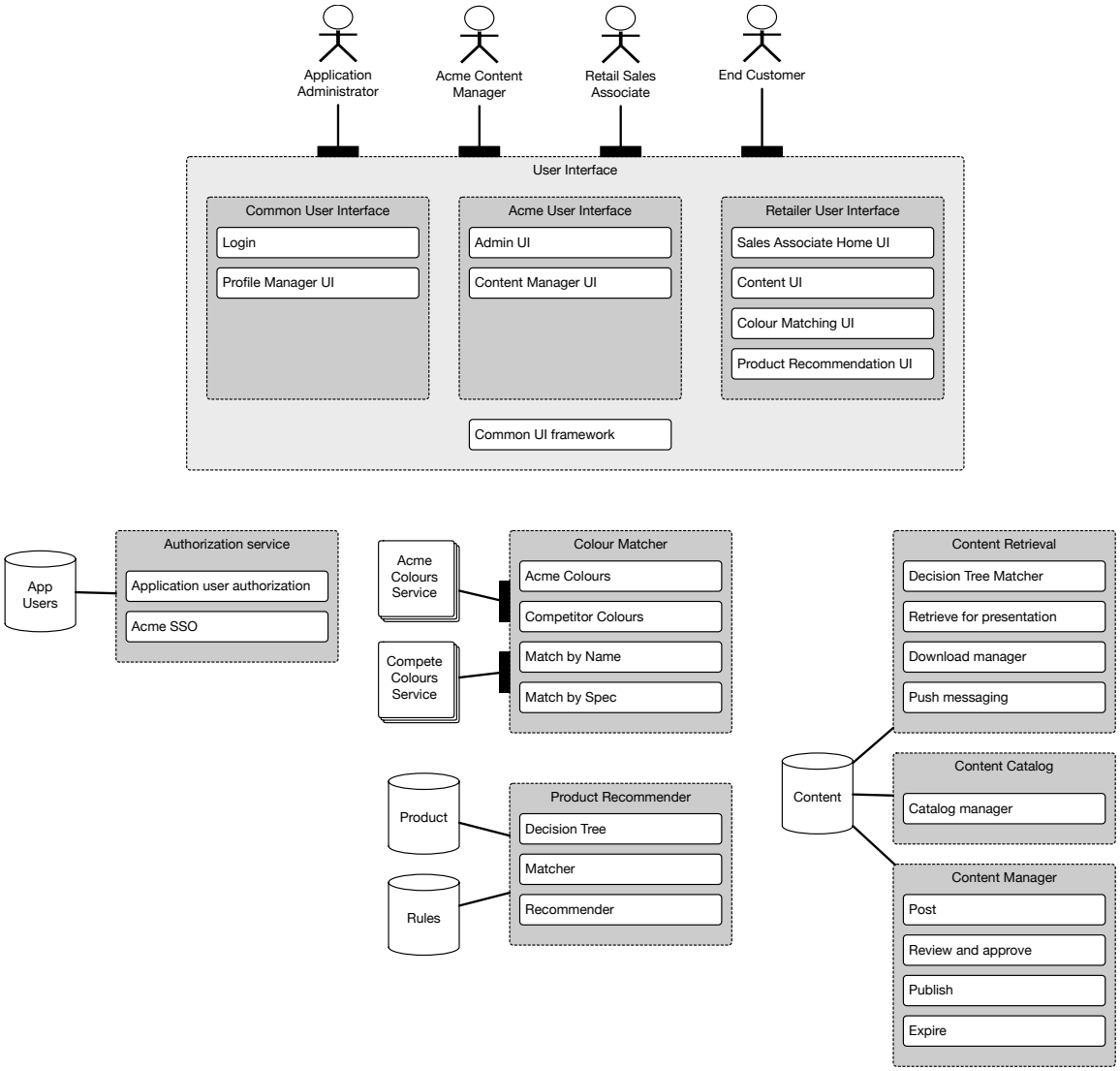
(This diagram is simplified for the purpose of the exercise; we omit a lot of the connections and additional explanatory detail we would have in a real-world solution.)



Considering the interfaces that we identified in Lab 2, we can make sure that each interface handled by some component in the application architecture. The author would fiddle around with the component diagram for a while at this point, thinking about what arrangement made the most sense; this diagram is the result of that and doesn't show intermediate steps.

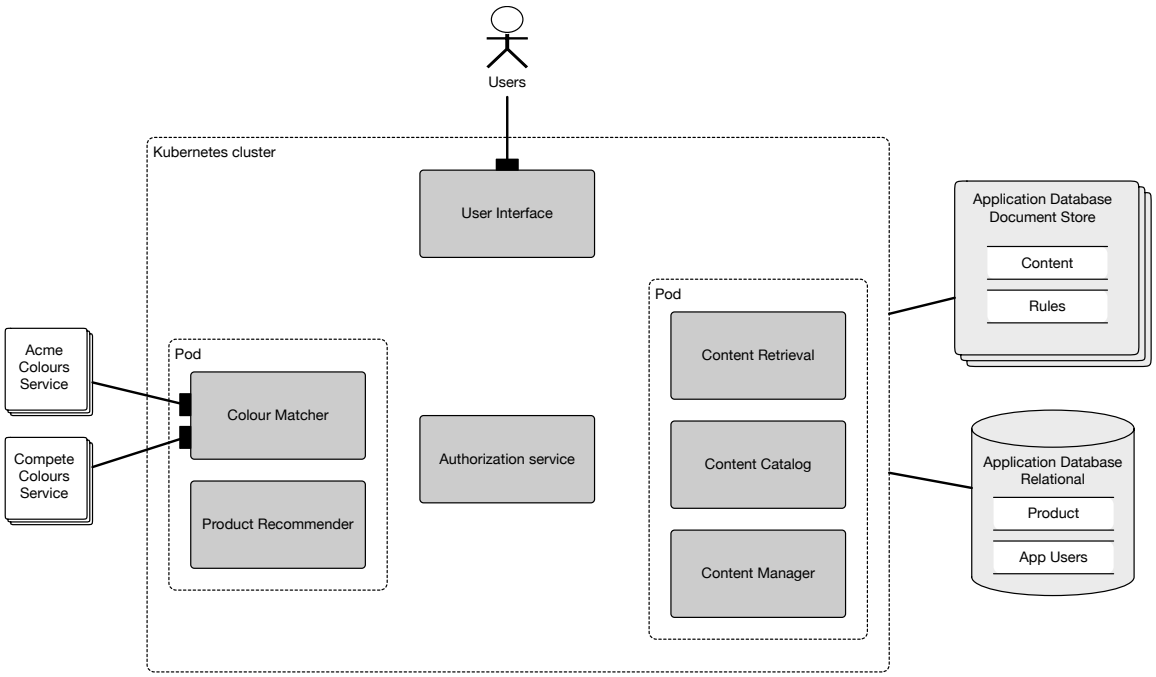
(Not all interfaces are shown here, to keep the diagram manageable on one page.)

Notice that we have changed the Acme Colours and Competitive Colours data stores from a database to a service, used through an interface; this reflects the context diagram which indicates colours are external to the application.



We explored the use of clustering as a tactic for availability (and to some extent deployment) back in Lab 4; we are going to assume that the application will be built using Docker containers. Under that assumption, we can think about how groups of components in our application architecture would be built and deployed together as a container.

We add data, and classify the data stores by type. For the application architecture, we want to identify what entities are managed by each data store, but we don't need to go into any more detail; details about data are already captured in the data architecture we did before.



A few notes about this diagram, which is considerably simplified from a complete application architecture:

- Each of the grey boxes in this diagram represents a container. Each container implements the functionality shown in the previous diagram. At runtime, there would be several instances of each container - by default at least three for redundancy, but more if there is sufficient load.
- Different containers could be implemented with different technology. For example, the user interface could all be in NodeJS, the authorization service could be written in Java, and the product recommender could be in Python. Since all interaction happens over the network, each container can be thought of as a black box.
- The internal relationships between containers (the connections between them) are not shown here for simplicity, but are an important detail.
- Some of the containers are grouped into “pods”. A pod is a set of related containers that are always run as a group; there are also some efficiencies in inter-container communication within a pod.
- This diagram does not show the container orchestrator (Kubernetes); this is an application architecture, after all.
- Databases are shown outside of the cluster; databases are not usually run inside containers, for reasons that are beyond the scope of this discussion. Notice that there are two types of databases - a traditional relational database and a document store - a document store is a type of database that stores “documents” (usually JSON objects) and can retrieve them based on their contents. Document stores have some interesting properties that make them a good alternative for relational

databases in certain cases.

- We have assumed that we can store several subjects in each database, which would usually be the case unless there are very different access patterns for different subjects.
- This diagram does not show which containers use which databases; this is an omission for simplicity but is an important detail.

Our application architecture satisfies the availability requirement; more detail is needed in the technical architecture to show just how that works, but the application part of it has been taken care of. Deployability will be largely taken care of by containers; again, there will be more detail in the technical architecture. What about localization? For our application, localization is expected to be implemented through a UI framework that will be accounted for in the design of the user interface controller components, and we don't really need to show that explicitly in the application architecture diagrams. A note to the designers that it is required is in order, but if we are comfortable leaving the implementation details to the designers we can leave it at that.

Lab 7 - Technical Architecture

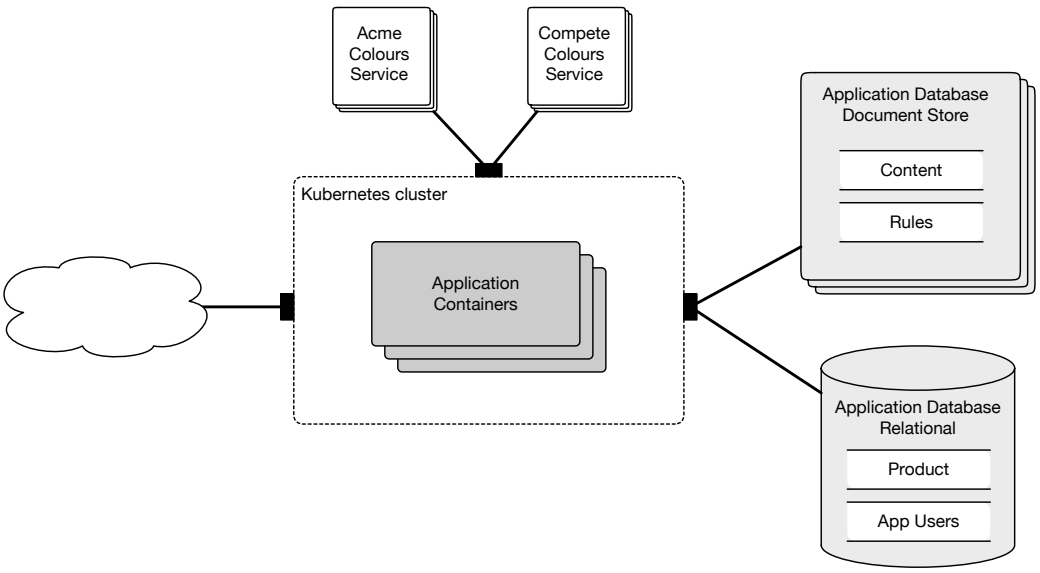
Given our quality of service requirements, a technical architecture using Docker containers seems appropriate. There are (many) other valid approaches, but for the sake of this exercise assume this approach is approved by Acme Paint and is the one we will pursue.

Let's also assume that we are going to deploy in our own data centre.

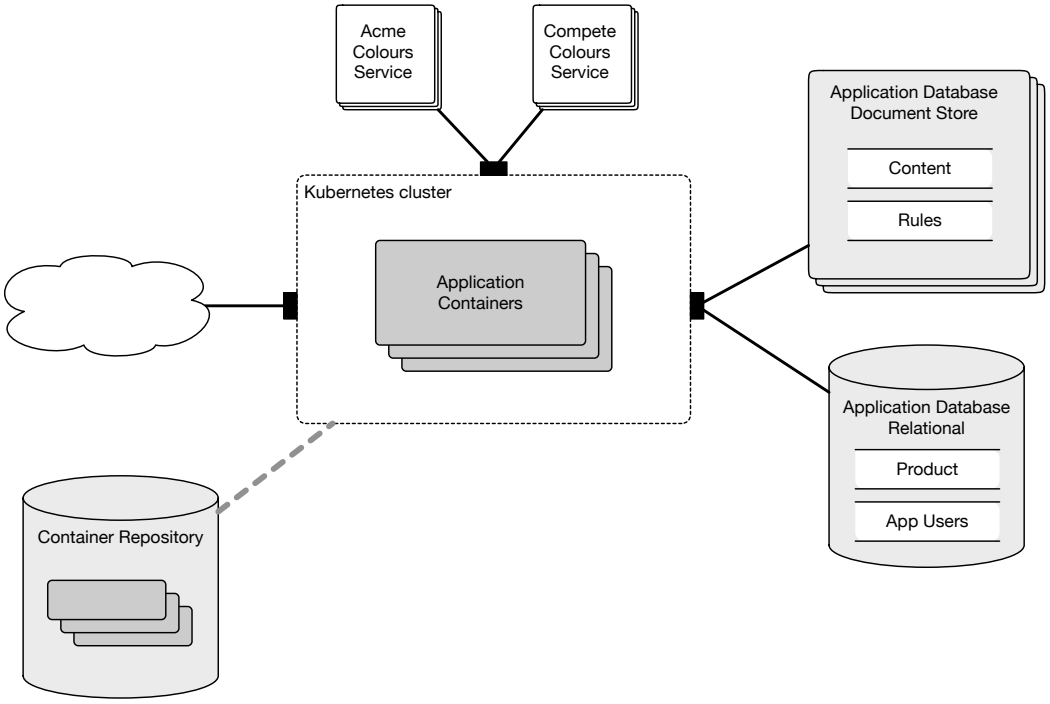
The basic components of our technical architecture, then, are

- Container (application components as described in the application architecture will be packaged in containers)
- Container runtime (what we have previously called the "cluster")
- Databases
- Related enterprise applications

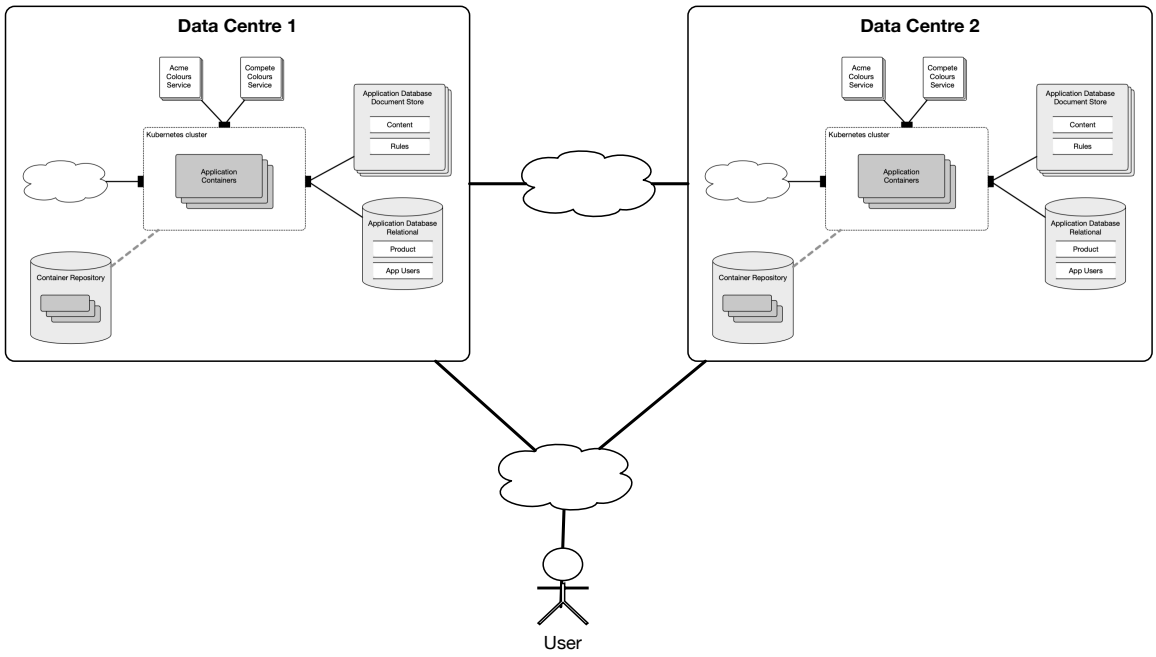
(Databases, if you are not familiar with containers, are not usually run inside containers - they will be separate technical objects in our architecture.)



That's not quite the whole picture - containers don't appear out of thin air when they are needed, they are retrieved from a repository. The repository hold container images; just like a VM image, a container image is a binary object that can be copied na instantiated to become a running container.



That's still not quite the whole picture, as everything is in one data centre and that doesn't meet our availability requirements. We need to add a second data centre and communications between them.



Here’s an interesting observation - if the container repository and runtime infrastructure is already in place, what do we actually have to deploy for this application? Remarkably little - to a first approximation only the application databases are deployed just for the application (and that’s assuming we need new database instances - if we can piggyback on an existing instance we may not need any physical infrastructure deployed for the application, as long as there is sufficient capacity.)

Where does the application come from, then? It’s in the containers, and the containers are software objects that can be built from source code on demand. Effectively, we can build and deploy our technical infrastructure pretty much completely in software. That’s a real plus for containers.