**Architecture Test**

I don’t have practical experience in this area, but I have a high-level idea of the concepts involved. I would certainly look to discuss this with others who have experience, but if these decisions were mine alone to make, what I discuss below would be where I start.

Note, I’ve done some research online to understand a little better what the intention is.

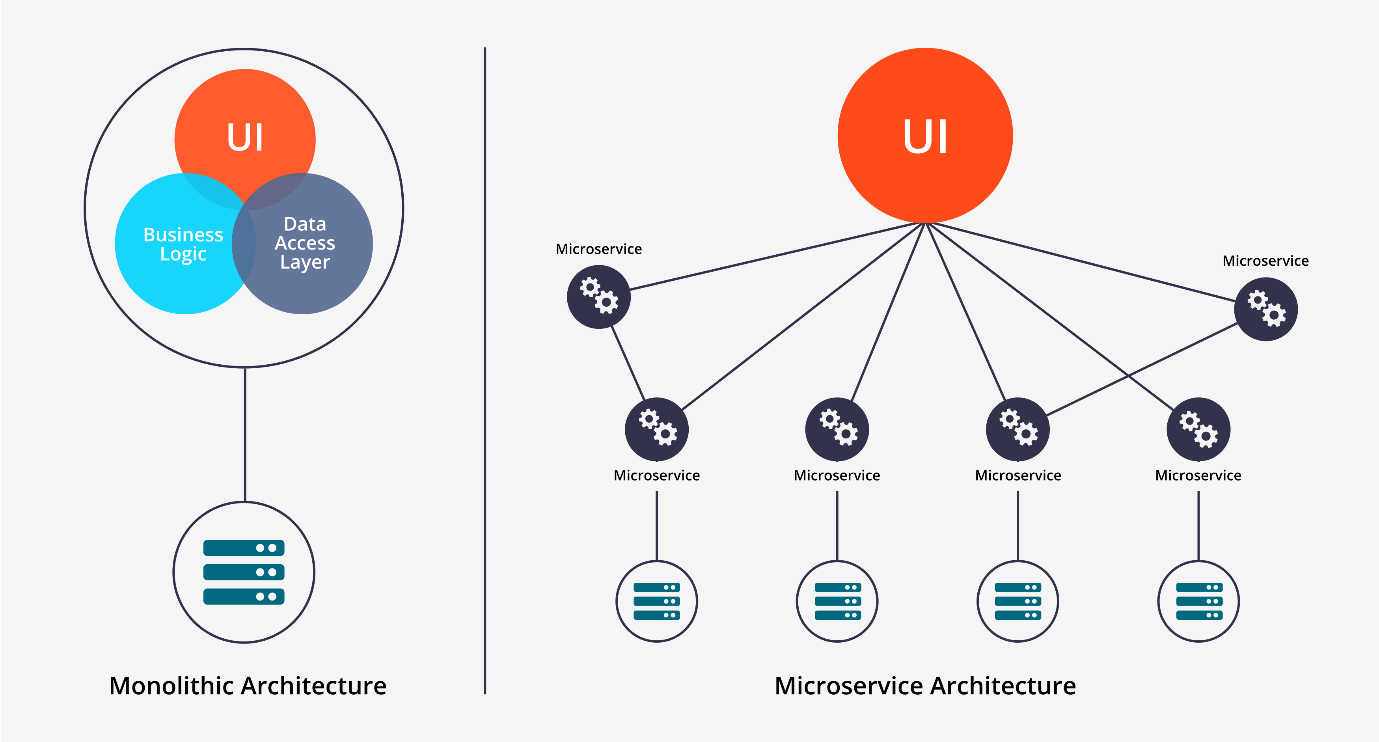
Something I want to make explicit that is somewhat implicit in my answers below: As with code, I try to break problems down into small, manageable, incremental changes. I try to find the next development that is independent of others that has the highest benefit. This is where I think pragmatism is important – valuing the effort that could be spent and sequencing the work most effectively in line with requirements and goals. The more I know about the goals and/or requirements of a project, and the software in question, the better decisions I can make with regards to prioritizing the work.

**Steps/Actions**

There are few important things to consider:

1. DevOps: While I’m not familiar with this as a concept in its own right, it appears to me to be the collaboration between the development aspects and operational aspects of the software. More specifically, this is working on both the software and the operational provision of the software at the same time, in the same designs, with an overall architecture that fits the operational needs.
2. Microservices: Not practically familiar with, but conceptually, this defines individual subsystems within the overall platform that work together with their own APIs. The benefit we look to exploit is separation of responsibilities and deployment – individual subsystems that are not dependent on each other (or on each other’s environments) that can be deployed independently and asynchronously.
3. Containers: Broadly familiar with the concept, this is used to package up software and deploy it to an environment in a consistent manner. Benefits including robustness, consistency, speed and portability.

The above will influence the architecture of the system we are looking to build or migrate to. I found this diagram which seems about right for what I’d expect (either developing the right or migrating left to right):



Assuming this is somewhat like the system we aim to build the next to consider is the approach:

1. Big bang: Develop the new architecture and when ready, in one or few steps, move/migrate from the current to the new.
   1. Advantages: Can be less effort to develop from scratch due to creating new practices and less dependencies/downstream impacts.
   2. Disadvantages: Should the project fail or progress slow, most or all of the effort could be lost if the work is not completed.
2. Iterative/incremental/refactoring: Take the existing system and refactor it into the new architecture in stages, then continue refactoring to improve the APIs within the system (such as subdividing into further microservices, and redefining their APIs).
   1. Advantages: Each change improves the existing system and builds on the value already held in the codebase.
   2. Disadvantages: Refactoring can be problematic, especially while trying to keep existing systems working and results consistent.

Either approach, an initial step would be a prototype of the final architecture. This would be a fully functional system, but a system that does nothing or something trivial (like sending messages between APIs). This prototype would be used to validate the steps in the project plan and design, and highlight concerns and/or issues before affecting the existing codebase.

For a big bang approach, the existing system’s most valued code could be taken and either rewritten or transposed into the new architecture. With more knowledge of the current architecture, we would identify the key aspects of the system and prioritize how we develop them in the new architecture, for example considering the most critical aspects to the business, the most complex code, etc.

For an iterative approach, we would (if possible) get the existing system working in the new architecture in as few steps as possible. For example, this could be to have one monolithic microservice. There may be steps prior to this to decouple layers of the software. Once this is done, then the system can be refactored and improved, subdividing it into the microservices and developing the containers as and when the microservices are ready. This would result in the existing codebase in a new-architecture compatible form.

**Azure Products and Services**

Honestly, I would need to understand more about the system and how appropriate the above ideas are before knowing what criteria these products and services need to meet. Having said that, I don’t think researching these products and services fresh is a bad idea considering the rate at which the industry changes.

**Technologies**

Same as above. I’m personally not experienced with technologies relevant here. While experience would give me a good starting point, it’s not a bad idea to see how relevant that experience would be and how outdated it might be.

**CI/CD**

Continuous integration is a common practice for code development in general. It is important to maintain an understanding of system stability and readiness at all times, especially with multiple developers working towards one or more products. Continuous delivery takes continuous integration further by providing the result of a successful ‘build’ as a release candidate. This means that if the ‘build’ is successful, it *could* be released if desired. ‘Build’ in this context could be any number of projects and/or libraries related to one or more products, and in some form would result in a built form of the software.

In terms of implementing this, some considerations:

1. Testing and coverage: While 100% test coverage would be ideal, likely it will be less. For continuous delivery (and a more confident continuous integration result) we would need test coverage over the most critical and complex areas of the system. This is important because to release a candidate we must be confident in its quality. Having said that, there is some grace with regards to bugs because with a continuous delivery pipeline, the bug can be patched much faster than without. If coverage is lacked where we feel it is needed, we would need to develop more tests. Of course, this can come at later stages to build confidence – it’s not a barrier to achieving this conversion.
2. Quality gates: If we don’t have already, we should use some code quality tools like SonarQube, checkstyle (maybe findbugs/spotbugs). There are a number of others, some that cover design patterns. This will help keep bugs lower (than without) and make maintenance easier in future.
3. Build tools and automation: We will need build tools to help us (such as ant, Maven, Gradle) configure our CI/CD to run our builds - run the tests and quality gates, build the binaries, etc.
4. Software change integration: We could have developments going straight onto the trunk however that could make successful release candidates less frequent. I think in my first interview we spoke about branch-per-task. That could fit well here as the feature/change would go through review and quality measures before making it to the trunk.

As for steps to take:

1. Get CI tool up and running. Jenkins is well known and in development still. There are others I’m not familiar with.
2. Get whatever tests we have running, preferably build on commit, if not polling every so often (say 15 mins). The frequency of builds will provide the team with knowledge of platform stability.
3. Take any automated processes already in place and exercise them from the CI platform.
4. Take any quick wins in terms of automating anything not already automated that is needed for product delivery (production of binaries, build product installers, etc).
5. Prioritize the remaining functions required to build the product and schedule their development.

DevOps is a new concept to me, but it feels very familiar as the scope of my involvement has covered the building and releasing of software such that I have scripted specifically for our clients, providing the software in the most appropriate form for them. While DevOps doesn’t appear to be limited to culture, I believe a team should feel responsibility for the products they work on. Having them contribute to this process and getting them involved is a way of empowering them and motivating them to think about the operations as well as the code. Not all developers will be interested in the process beyond the code, however by building the process into the continuous integration platform, they will be able to see the requirements they must satisfy with their code and designs - this being in the form of quality gates, tests, and potentially environment build tests (for testing deployments).

**Container Orchestration**

I don’t know much about this aspect. From my research it involves automation of the operational effort associated with running containerized services. This includes provisioning, deployment, load balancing, etc. Again, don’t know much of these topics, but I would investigate them to see what’s needed and prioritize them based on our current requirements.