1. **Think about the ISO/IEC/IEEE Standard 42010 definition of a system architecture that we saw in section 2.0 (and remember that we have discussed software as one kind of system or at least as part of a system). This definition in part defines an architecture as a system as it appears within its environment, related to the influences of the environment on the system. Think of an example where a software package exists in one environment and must be ported to a different environment, where the influences of the new environment on the system require changes to the system.**

   Let's take the case where one is moving an application from a desktop environment to the cloud. If it's basically a monolithic application such as, say, a statistics package then moving it from a desktop environment to the cloud is easy but you don't necessarily get a lot of bang for the buck—the most likely advantage is that if a lot of users access this package then the cloud could potentially fire off new copies of the application in a virtual machine. (But in a desktop environment (not using virtualization) you could maybe have just had the users access different machines.)

   Now let's consider a distributed application that you move from a desktop system to the cloud. If it's distributed using ONC Remote Procedure Calls then it might be difficult to use an ONC RPC from one cloud virtual machine to another cloud virtual machine—for one thing, it's difficult to go through firewalls with ONC RPCs. A better technology to use would be some kind of Web Service. (But in any case what you really need is a Service Oriented Architecture of some kind.)

2. **In regard to the discussion about Fielding's view of architecture from section 2.0, why does it say it is difficult to directly compare the architectures of different systems or even compare the same system in two different environments? Give an example.**

   Fielding says that quality attributes are different for different systems, or for different implementations of the same system. For example, let's consider a software application that is running on a very low powered processor, with not much memory and a low execution speed. In that case, it is very important for that software application to be very efficient (quality factor:

efficiency). Now let's port this same software application to a high powered processor, with a lot of memory and a high execution speed. In this case, being able to re-use portions of the source code of this software application to build new environments (quality factor: reusability) becomes more important to consider than efficiency, since the software doesn't have to be very efficient in order to run appropriately in this new environment.

3. **Does it really matter whether you call something an "architectural style" or an "architectural pattern"?**

   Nah. A rose by any other name would smell as sweet. Just define exactly what you mean.

4. **Fielding refers to an "Event-based integration" architectural style and Gomaa discusses group messaging notification patterns (broadcast and subscription/notification). In section 2.2 you were told that these overlap. How are they alike? How are they different?**

   From section 2.2.2, Fielding's Event-based integration is a peer to peer style that says:

   - Instead of invoking a component directly, a component can broadcast events
   - Components register interest in an event, and the system invokes the registered components

   From section 2.2.1, Gomaa discusses Group communication patterns:

   - Broadcast pattern
     - An unsolicited message is sent to all recipients.
   - Subscription notification pattern
     - This is a selective form of group communication where the same message is sent to members of a group. A component can subscribe and unsubscribe from a group, and can belong to more than one group.

   So they are very similar. Both allow broadcasting events, and both distinguish between solicited broadcasts and unsolicited broadcasts (with Fielding the concept of unsolicited broadcasts is implied rather than explicitly stated). Both allow a component to show interest in an event, and be

sent messages when the event occurs. Gomaa's is a broader concept in that it is group based instead of single user based.

5. **Why do you think Gomaa divides architectural patterns into structure patterns, which describe the static structure of an architecture, and communication patterns, which address dynamic communication among distributed components of an architecture. In particular, Gomaa specifies an example of a structural pattern, called multiple client/single service, in which one service fills requests from more than one client. But…but…but, client/server is an example of communication between distributed components. Why does Gomaa call this a structure pattern? Why isn't it just yet another example of a communication pattern?!?**

   Those of Gomaa's structure patterns discussed in section 2.2.1 are concerned with how client/server software should be organized in terms of ***what component*** is talking to ***what component***. For example, another architectural pattern he discussed is multiple client/multiple service, where a client communicates with several services but a service may also communicate with other services.

   Instead, Gomaa's communication patterns are concerned with ***how*** any kind of call from one component to a different component takes place. For example, he talks about asynchronous vs. synchronous calls. So *which* client talks to *which* server is a structure pattern, but *how* the client talks to the server is a communication pattern. There is some overlap, however, in that Gomaa includes broker patterns under the heading of communication patterns. This is reasonable in that it specifies the *how* a connection between client and server takes place in that when using a broker pattern the connection is asynchronous, and that the broker has the potential to add additional value added during the connection. However, it could be considered a structure pattern instead in that it specifies that a client talks to a broker which talks to a server, so it is also concerned with between *which* components the connection takes place. So to some degree the division between the structure and communication categories is arbitrarily drawn.

6. **In the message queue paradigm, what are the advantages and disadvantages of push compared to pull?**

   As we saw in section 2.2.1, a pull message queue makes more work in terms of polling for the recipient, but also means that the recipient would is never overwhelmed by having too many messages to process at the same time. However, a push queue works well when messages are fairly infrequent (so the possibility of being overwhelmed with too many messages at the same time is low), and when the recipient needs to know as soon as possible when the message is available.

7. **What really is the difference between completion time and latency in terms of user perception? Give an example that illustrates this.**

   For example, the user wants to download a very big file that will require half an hour on the user's slow internet link to complete. If the user has to wait until the complete file has downloaded before getting any feedback, then the latency (as perceived by the user) would be poor. That is, the user wouldn't know that anything was going on for half an hour, and wouldn't be sure that his/her command to download had even been recognized. However, perhaps the user could be presented with a status bar, which is updated as new chunks of the file come in. This reduces latency as perceived by the user. The *initial* latency would be the time required for the first chunk of data to come in.

   A more generic definition of user perceived latency would be the time it takes for the round trip between when the user enters the command, and when (any kind of) response to the command comes back.

   Note that the definition of user perceived latency is not the same definition of latency as, for example, one would use for network latency. With network latency you're thinking more about the round trip time required for a single packet.

8. **Can you think of any situation where scalability is important? Give an example.**

Perhaps a popular or perhaps a notorious event could occur at a university that due to news coverage would suddenly make the university's website much more popular than usual. The servers supporting the university's website could be overwhelmed. If the university's website were hosted on a cloud, however, the cloud could supply additional resources to handle the temporary overload. An example of a popular event might be winning a national championship in football. An example of a notorious event could be a shooting event at the university.

9. **Discuss how efficiency could be hurt when reusability is improved.**

   When you make a component more reusable, you normally have to make the component more generic. One way in which components are made more generic is to accept a wider set of input data. Since there are more different kinds of input data, checking input data for validity becomes harder and requires more code, which therefore hurts efficiency.

10. **Why might it be important to design application software such that different middleware technologies could be used at different times ?**

    Some middlewares are better suited to a particular environment than others, so if application software is ported to a different environment, it might make sense to swap out the middleware it uses. (One such example is moving some kinds of distributed systems to a cloud.)

    Also, different middlewares tend to come in and out of favor over time. To keep the software application so that it could work in new environments, swapping out the middleware could be useful.