**CSCI 391 – ST: Microservices, Midterm Exam 1**

Name: Key

Student Id: Key

*Success is just around the corner—trust in your preparation and the journey ahead.* **🍀**

**What are Microservices? (16 Points)**

1. Which of the following descriptions best characterizes what the traditional three-tiered architecture (depicted below) prioritizes?



1. High cohesion of technical functionality over high cohesion of business functionality.
2. High cohesion of business functionality over high cohesion of technical functionality.
3. Which of the following are *advantages* of microservices (circle all that apply)?
4. Technology Heterogeneity.
5. Developer Experience.
6. Latency.
7. Ease of Deployment.
8. Scaling.
9. Which of the following are *disadvantages* of microservices (circle all that apply)?
10. Robustness.
11. Cost.
12. Organizational Alignment.
13. Reporting.
14. Monitoring & Troubleshooting.

*For questions 4 to 6, assess whether each scenario represents a good or bad fit for microservices adoption.*

1. A new product or startup where the domain is rapidly evolving and undergoing frequent changes as they refine the core elements of what they are trying to build.
   1. Bad fit for microservices adoption.
   2. Good fit for microservices adoption.
2. A hundred-person scale up that is growing rapidly and wants to allow more people to work independently of each other, reducing delivery contention.
   1. Bad fit for microservices adoption.
   2. Good fit for microservices adoption.
3. An organization creating software that will be deployed and managed by their customers.
   1. Bad fit for microservices adoption.
   2. Good fit for microservices adoption.
4. What are the three different monolithic architectures (where monolith refers to a single unit of deployment)?

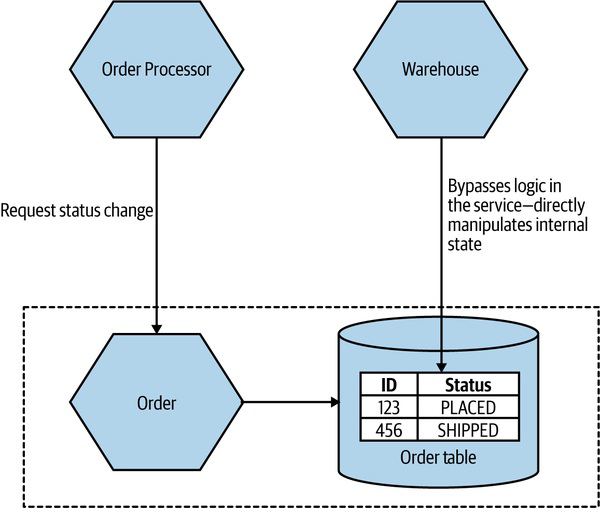
Single process, modular, and distributed monoliths.

1. Which of the following is an advantage of a monolithic architecture?
   1. Simpler deployment topologies.
   2. Simplified code reuse.
   3. Simplified monitoring and troubleshooting.
   4. a, b, and c are all advantages of a monolithic architecture.

**How to Model Microservices (30 Points).**

*For questions 9 to 12, circle the form of coupling that is the lowest, loosest, or most desirable.*

1. Pass-through coupling or Domain coupling.
2. Content coupling or Common coupling.
3. Domain coupling or Common coupling.
4. Pass-through coupling or Content coupling.
5. Which type of coupling is pictured below?



* 1. Domain coupling.
  2. Content coupling.
  3. Common coupling.
  4. Pass-through coupling.

*An order processor microservice makes requests to three different pricing microservices to obtain the best price to order new stock of a specific item. The latency of the response times for the pricing services are as follows:*

|  |  |
| --- | --- |
| **Pricing Microservice** | **Time (ms)** |
| *A* | *1* |
| *B* | *3* |
| *C* | *5* |

1. What is the total latency (duration) of the calls if they happen sequentially?

9 ms

1. What is the total latency (duration) of the calls if they happen parallelly?

5 ms

1. Common coupling can be acceptable in some cases, but not always. Under what conditions might it be acceptable to have common coupling in your system?

Unidirectional flow of data (i.e., one microservice is writing data, one is reading), static reference data.

1. True or false, one aggregate should be managed by more than one microservice?
2. How can we reduce the amount of pass-through coupling and its impact in MusicCorp’s order processing (as described below)? In your response, explain one of the three approaches discussed in the textbook and class, along with potential drawbacks and advantages of the approach you select. Additionally, provide an updated architectural diagram which describes the responsibilities of each microservice in MusicCorp’s updated order process.

*As an example of Pass-through coupling, let’s look more closely now at part of how MusicCorp’s order processing works. In the figure below we have an Order Processor, which is sending a request to Warehouse to prepare an order for dispatch. As part of the request payload, we send along a Shipping Manifest. This Shipping Manifest contains not only the address of the customer but also the shipping type. The Warehouse just passes this manifest on to the downstream Shipping microservice.*



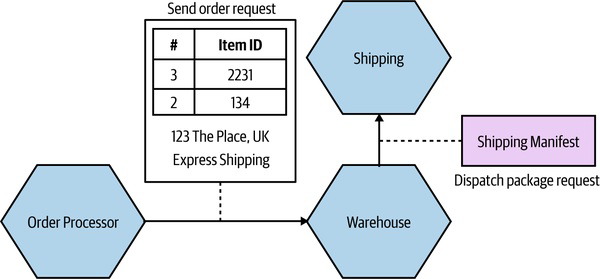
**YOUR ANSWER HERE:**

The major issue with pass-through coupling is that a change to the required data downstream can cause a more significant upstream change. In our example, if Shipping now needs the format or content of the data to be changed, then both Warehouse and Order Processor would likely need to change. There are a few ways this can be fixed. The first is to consider whether it makes sense for the calling microservice to just bypass the intermediary. In our example, this might mean Order Processor speaks directly to Shipping, as in Figure 2-5. However, this causes some other headaches. Our Order Processor is increasing its domain coupling, as Shipping is yet another microservice it needs to know about—if that was the only issue, this might still be fine, as domain coupling is of course a looser form of coupling. This solution gets more complex here, though, as stock must be reserved with Warehouse before we dispatch the package using Shipping, and after the shipping has been done, we need to update the stock accordingly. This pushes more complexity and logic into Order Processor that was previously hidden inside Warehouse.



*Figure 2-5. One way to work around pass-through coupling involves communicating directly with the downstream service.*

For this specific example, I might consider a simpler (albeit more nuanced) change—namely, to totally hide the requirement for a Shipping Manifest from Order Processor. The idea of delegating the work of both managing stock and arranging for dispatch of the package to our Warehouse service makes sense, but we don’t like the fact that we have leaked some lower-level implementation— namely, the fact that the Shipping microservice wants a Shipping Manifest. One way to hide this detail would be to have Warehouse take in the required information as part of its contract, and then have it construct the Shipping Manifest locally, as we see in Figure 2-6. This means that if the Shipping service changes its service contract, this change will be invisible from the viewpoint of Order Processor, as long as Warehouse collects the required data.



*Figure 2-6. Hiding the need for a* Shipping Manifest *from the* Order Processor.

While this will help protect the Warehouse microservice from some changes to Shipping, there are some things that would still require all parties to change. Let’s consider the idea that we want to start shipping internationally. As part of this, the Shipping service needs a Customs Declaration to be included in the Shipping Manifest. If this is an optional parameter, then we could deploy a new version of the Shipping microservice without issue. If this is a required

parameter, however, then Warehouse would need to create one. It might be able to do this with existing information that it has (or is given), or it might require that additional information be passed to it by the Order Processor.

Although in this case we haven’t eliminated the need for changes to be made across all three microservices, we have been given much more power over when and how these changes could be made. If we had the tight (pass-through) coupling of the initial example, adding this new Customs Declaration might require a lockstep rollout of all three microservices. At least by hiding this detail we could much more easily phase deployment. One final approach that could help reduce the pass-through coupling would be for the Order Processor to still send the Shipping Manifest to the Shipping microservice via the Warehouse, but to have the Warehouse be totally unaware of the structure of the Shipping Manifest itself. The Order Processor sends the manifest as part of the order request, but the Warehouse makes no attempt to look at or process the field—it just treats it like a blob of data and doesn’t care about the contents. Instead, it just sends it along. A change in the format of the Shipping Manifest would still require a change to both the Order Processor and the Shipping microservice, but as the Warehouse doesn’t care about what is in the manifest, it doesn’t need to change.

*For questions 19 to 21, choose the strategy described in each scenario as an alternative to using the Business Domain as the boundary for microservice modeling.*

1. Identify the parts of the system that experience more frequent changes, and extract those functionalities into separate services, allowing for more effective development and maintenance.
   1. Technology.
   2. Data.
   3. Volatility.
   4. Organizational.
2. Services within the green zone (outlined by a dotted green line) never have access to credit card information. This sensitive information is restricted to processes and networks within the red zone (outlined by red dashes). The gateway directs calls to the appropriate services and zones, ensuring that when credit card information passes through the gateway, it is treated as being within the red zone.
   1. Technology.
   2. Data.
   3. Volatility.
   4. Organizational.
3. Some of your functionality requires implementation in a runtime like Rust, which allows you to achieve higher performance gains.
   1. Technology.
   2. Data.
   3. Volatility.
   4. Organizational.

**Microservices Communication Styles (20 Points).**

*For questions 22 to 24, determine whether each scenario best describes In-process or Inter-process communication.*

1. The underlying compiler and runtime can carry out a whole host of optimizations to reduce the impact of the call, including inlining the invocation so it’s as though there was never a call in the first place.
   1. In-process.
   2. Inter-process.
2. When making a backward-incompatible change to an interface, you either need to do a lockstep deployment with consumers, making sure they are updated to use the new interface, or else find some way to phase the rollout of the new contract.
   1. In-process.
   2. Inter-process.
3. You are vulnerable to a host of errors that are outside of your control i.e., network timeouts.
   1. In-process.
   2. Inter-process.
4. What is the difference between 400 and 500 series response codes (which are both reserved for errors) in the HTTP protocol?

400 – something is wrong with the request. 500 – something is wrong with the downstream microservice.

*For questions 26 to 28, determine whether each scenario best describes Synchronous blocking or Asynchronous nonblocking communication.*

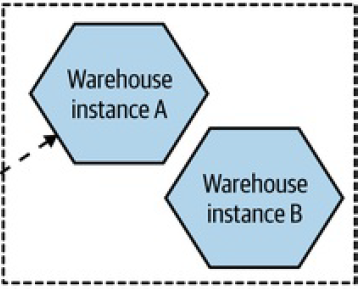
1. Becomes problematic when you start having more chains of calls.
   1. Synchronous blocking.
   2. Asynchronous nonblocking.
2. Every call would also be considered a request-response call.
   1. Synchronous blocking.
   2. Asynchronous nonblocking.
3. The microservice making the initial call and the microservice (or microservices) receiving the call are decoupled temporally.
   1. Synchronous blocking.
   2. Asynchronous nonblocking.

*For questions 29 to 31, determine the best Asynchronous nonblocking communication style to implement based upon the given scenario.*

1. There is a need for interoperability between different types of systems, including older mainframe applications or customizable off-the-shelf (COTS) software products.
   1. Common data.
   2. Event-driven.
   3. Request-response.
2. An upstream microservice need to make sure something gets done by a downstream microservice, and in fact asks it to.
   1. Common data.
   2. Event-driven.
   3. Request-response.
3. A service broadcasts information and assumes that interested parties will react accordingly.
   1. Common data.
   2. Event-driven.
   3. Request-response.

**Implementing Microservice Communication (10 Points).**

1. These serialization protocols are what you want to use if you start getting worried about payload size or about the efficiencies of writing and reading the payloads.
   1. Textual formats.
   2. Binary formats.
2. One or many instances of the same microservice (pictured below) that work together to process messages from a message broker, such as Kafka, RabbitMQ, or others is referred to as what?



Consumer Group

1. Which of the following would be used to implement point to point asynchronous communication between two microservices?
   1. Topic.
   2. Queue.
2. The OpenAPI specification (which originated from the Swagger project) now provides the ability to define enough information about endpoints to generate client-side code in various languages for which technology?
   1. Remote Procedure Calls.
   2. REST.
   3. GraphQL.
   4. Message Brokers.
3. Which of the following technology is also considered a *middleware*?
   1. Remote Procedure Calls.
   2. REST.
   3. GraphQL.
   4. Message Brokers.