**Institute of Systems Science**
**National University of Singapore**

# MASTER OF TECHNOLOGY IN ENTERPRISE SOFTWARE ENGINEERING

## Graduate Certificate Examination

## Subject: *Architecting Scalable Systems*

# Sample Examination Solutions

## SECTION A

**Question 1** *(Total: 25 Marks)*

a. Other answers will be considered in their own merits.

i. One architectural impact regarding the seed (Arrival Time):
It is required that the platform should be scalable to support 10,000 updates per minute in Phase 1 and the expected response time for updating the arrival times should be less than 3 seconds for 90% of the requests. The platform should also be prepared for the planned growth of having 3,000 buses as the fleets expand in future. This implies that the performance unit of the database in terms of CPU, memory and Data I/O need to be increased to handle higher input/output operations per seconds (IOPS). This will improve the read and write performance of the database.

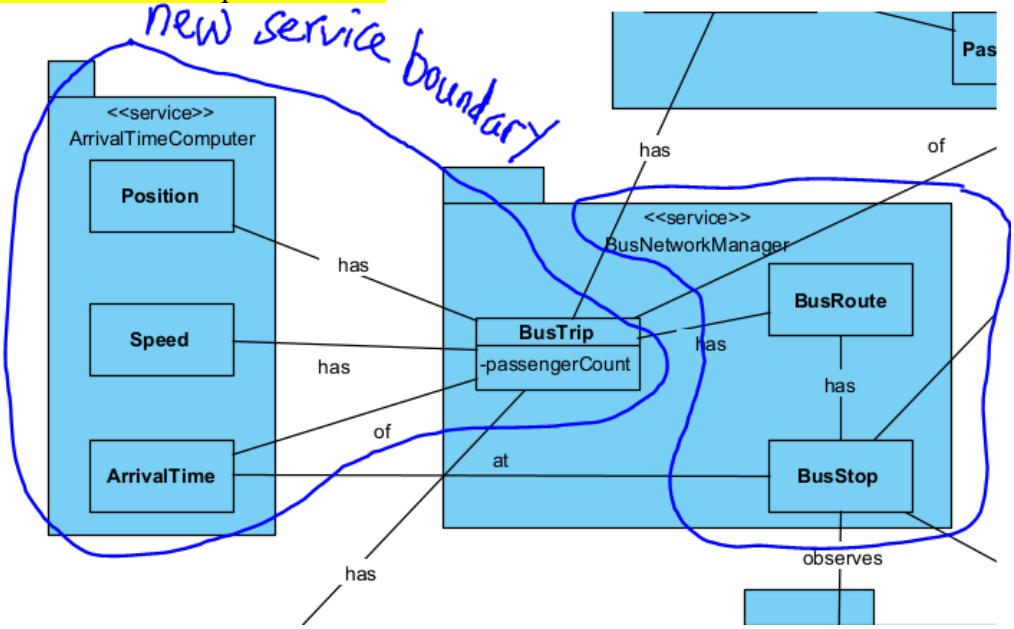
ii. Another architectural impact regarding 'View Arrival Times and Passenger Loads at Favourite Location' and 'View Arrival Times and Passenger Loads at Adhoc Location' interactions is:
The volume of users currently, in Phase 1 is 100,000 public users and 10,000 registered users and is projected to grow in future. For the computation of arrival times and passenger loads, updates are expected to be synchronized every 30 seconds. One crucial aspect of the architecture is to be able to concurrently perform the computation for each user in a timely manner before the updates to the users become stale. The update latency should not worsen as the volume of users changes.

*(2 Marks)*

b. Issues that arise due to the domain model.

*i.* The current domain design requires the ArrivalTimeComputer service to interact with the BusTrip entity of the BusNetworkManager service whenever it computes the arrival times for a given bus trip. This is relatively slow due to the need for inter-process, and likely inter-machine, communication between the two services. Given that the computation of the arrival times is likely the most frequently performed function of the platform, any such delay would be amplified. This issue can be effectively mitigated as below by relocating the BusTrip entity from the BusNetworkManager service to the ArrivalTimeComputer service.

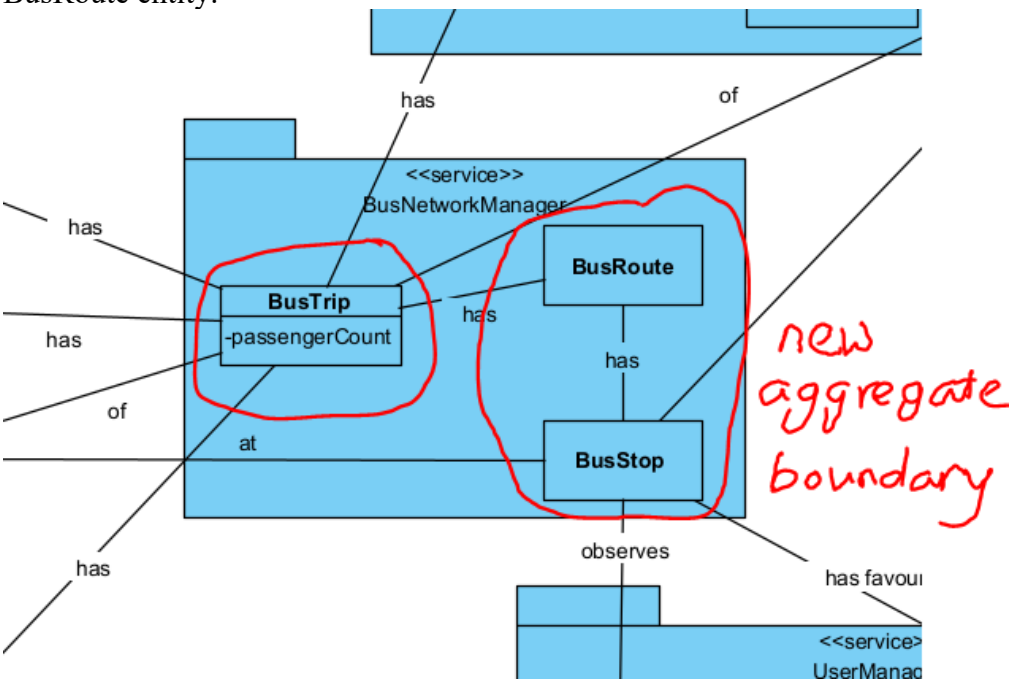
inter-process/inter-machine could be slow




Doing this would do away with the communication overhead between the two services. However, this countermeasure would compromise the functional cohesion of the ArrivalTimeComputer service. Perhaps it can be renamed to BusTripManager.

ii. The current domain design places the BusRoute and BusTrip entities in the same aggregate. This implies that all the bus trips of a given bus route would be loaded in memory at the same time. Assuming that this is only done on a per day basis, this could still result in tens of bus trips being loaded in memory for any given bus route. That is quite a large memory footprint for a bus route. Furthermore, there is not really a need for all the bus trips to be loaded in memory in the first place – the processing would probably require only the latest bus trip for a given bus route. The issue can be effectively mitigated by refactoring the aggregate design of the BusNetworkManager service as below so that the BusTrip entity is not residing in the same aggregate as the BusRoute entity.

In same aggregate means both entity will have to be loaded at same time




Doing this would not result in loading all the bus trips of a given bus route when processing a bus route. It is also convenient to place the BusStop entity in the same aggregate as the BusRoute entity – the bus stops of a bus route would be loaded when a bus route is load; the memory footprint is manageable as the number of bus stops for any given bus route is limited and static. There is no major drawback for this countermeasure.

*Marking Scheme for (i):*

- *2 marks for the most likely cause.*
- *2 marks for the countermeasure.*
- *1 mark for the drawback of the countermeasure.*
- *Up to a maximum of 5 marks.*

*(5 Marks)*

*Marking Scheme for (ii):*

- *1½ marks for the most likely cause.*
- *1½ marks for the countermeasure.*
- *1 mark for the drawback of the countermeasure.*
- *Up to a maximum of 4 marks.*

*(4 Marks)*

c. API Design.

A possible solution is as follows. Other solutions are acceptable with viable justifications.

The PUT method, instead of the PATCH method, was used in the APIs for vehicle onboard units to update the platform with vehicular and passenger information. Although the PUT and PATCH methods are both usable for updating a resource, they are technically different. The PUT method modifies a resource by providing the entire resource for update while the PATCH method provides only the part of resource that needs to be updated. API requests using the PATCH method would carry a smaller payload, resulting in better communication performance. Also, as the 'Bus Route' parameter is actually redundant, it should be removed from both of the APIs. This would also reduce the payload of the API requests, resulting in better communication performance.

The following example shows the payload of an API request using the PUT method to update the position and speed of a bus. It includes all the parameters.

```
{
        "bustrip": {
                tripId: "40911",
                …
        },
        "busroute": {
                routeId: "TP211AMK",
                …
        },
        "position": "27.8247427:-82.75040159999999",
        "speed": "30.255mph"
}
```

The following example shows the payload of the same API request using the PATCH method. It includes only the resource ID as well as the resource data that change.

```
{
        "bustrip": {
                tripId: "40911"
        },
        "position": "27.8247427:-82.75040159999999",
        "speed": "30.255mph"
}
```

In addition, API requests using the PUT method also require additional processing overhead to check for the existence of the resource before updating/creating it. Whereas, API requests with the PATCH method will simply update an existing resource. Doing away with existence check would also improve the performance of both APIs.

*Marking Scheme:*

- *2 marks for evaluating the current API design.1 mark for suggesting an alternate design*
- *3 marks for illustrating and articulating the alternate design.*
- *Up to a maximum of 6 marks.*

*(6 Marks)*

d. Two most important quality attributes for the storage/database technology.

The following are the top 2 quality attributes. Other quality attributes would be considered on their merits.

- High throughput: there is a need to support a high volume of telemetry data from vehicle onboard units. So, the ability to support high throughput is important.
- High availability: downtime would cause loss of data and loss of service.

*Marking Scheme:*

- *1 mark for each quality attribute with justification.*
- *Up to a maximum of 2 marks.*

*(2 Marks)*

e. Two cloud-based storage/database services.

The following are two possible services. Other services would be considered on their merits.

- HBase (or other column database) for storing telemetry data. HBase is scalable, highly available and support a high throughput of insert which is appropriate for storing telemetry data.
- Redis (or other in-memory cache) is suitable to store a high traffic query from users. In this platform, commonly requested information such as bus route and bus route-bus stop mapping can be stored in the cache. With careful design, even bus arrival times and passenger loads can be cached to improve performance.

*Marking Scheme:*

- *1 mark for each proposed service and 1 mark for its justification.*
- *Up to a maximum of 4 marks.*

*(4 Marks)*

f. Running two or more services in the same container instance.

The primary advantage of sharing the container instance is really the drastically reduced inter-service communication overhead. Those services that manifest high communication overhead between them would benefit greatly from sharing the same container instance. However, one salient constraint for sharing the same container instance is that the services would be scaled exactly the same way. Those services with similar scaling profile would be especially suitable.

Based on the domain model in Section 7 of the case study, it can be seen that the BusNetworkManager and ArrivalTimeComputer services are coupled with multiple associations. Whenever the new position and speed of a bus are received, the ArrivalTimeComputer and DesirableSpeedComputer would compute the arrival times and desirable speed. In a way, these two services would have relatively similar scaling profile. So, one suitable combination is to collocate the BusNetworkManager, ArrivalTimeComputer and DesirableSpeedComputer services in the same container instance.

*Marking Scheme:*

- *1 mark for the configuration.*
- *1 mark for suitable justifications.*
- *Up to a maximum of 2 marks.*

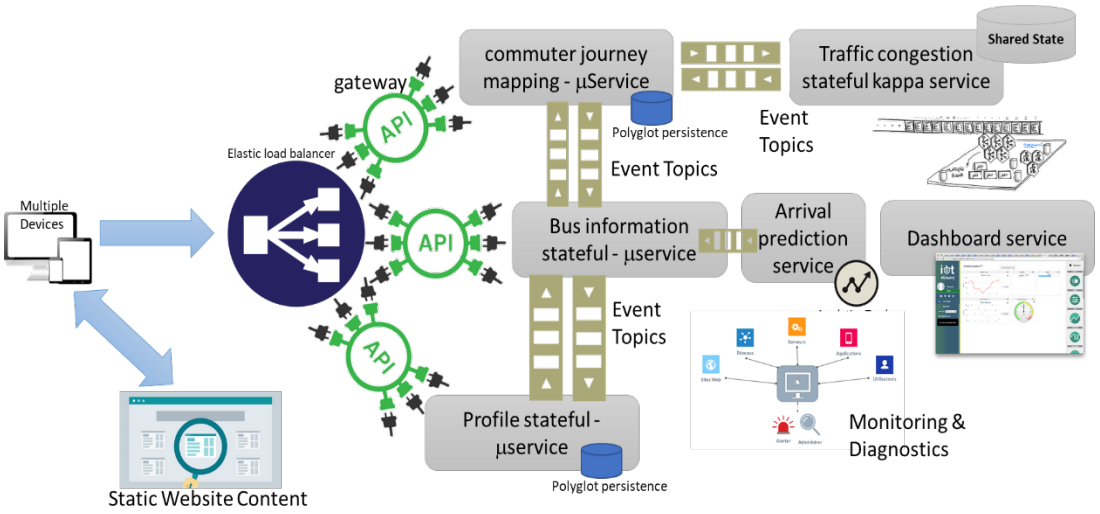
*(2 Marks)*

**Question 2** *(Total: 13 marks)*

a. Cloud native architecture.

Model Answer

Additional components: Event Driven Message Queues, Persistent State Store, NewSQL and NoSQL persistent store, Monitoring & Diagnostic Tools, Dashboard, Prediction Analytics Pipeline and Static Content Hosting.
Removed components: web servers, application servers and RDBMS,
Change of components: Traditional hardware-based load balancers converted to Elastic Load Balancers.
New Architecture would be:




More than pure microservices, a hybrid architecture involving microservices, serverless architecture and event driven architecture would be preferred.

*Marking Scheme – Component List: 3 Marks, Diagram and labelling: 2 marks and hybrid architecture: 2 marks.*

*(7 Marks)*

b. Virtual machines or containers for hosting.

Model Answer

Since the BusInfo4Me is always dealing with numeric, geospatial and text-based information, containers will be most suitable for hosting. Container automates transportation management by processing the data from sensors and connected IoT (Internet of Things) devices. The idea of leveraging container architecture to improve the efficiency of intelligent transportation systems is also getting traction. Kubernetes helps collect and analyse the traffic and transportation data in real-time. Devices such as automatic vehicle identifiers, GPS locators, sensors, and cameras are used to map the traffic and transit conditions. Some of the data the devices collect include travel speed, time delays and traffic count. Using containerization in a transport system involves including all the information an IoT device needs inside a container.

*Marking Scheme: Container choice: 1 mark and rational how it fits the proposed architecture: 2 marks.*

*(3 Marks)*

c. Cloud products/managed services.

Model Answer using AWS. Similar ones with GCP or Azure or Other will also be evaluated based on the merits of proposal.

Amazon Forecast is a fully managed service that uses machine learning to deliver highly accurate forecasts. Amazon Forecast uses machine learning to combine time series data with additional variables to build traffic prediction. Using machine learning, Amazon Forecast can work with any historical time series data and use a large library of built-in algorithms to determine the best fit for your particular forecast type automatically. The concurrent launch of CloudWatch Metrics, Auto Scaling, and Elastic Load Balancing helps in monitoring and diagnostics dashboard.

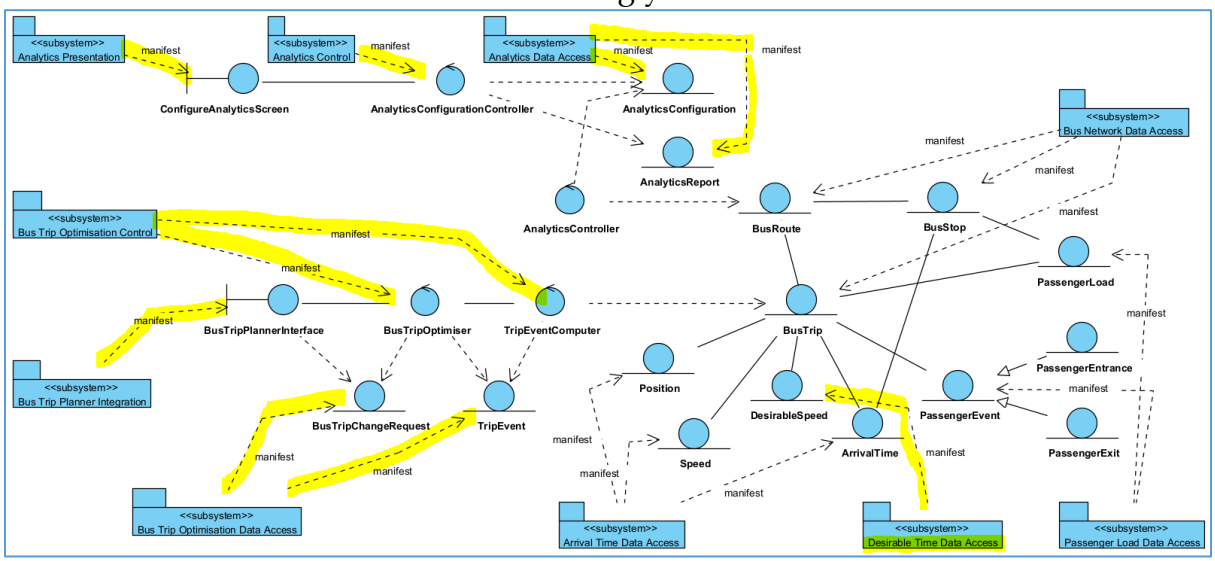

Alternatives could be using EC2 or Amazon Quick Sight /Athena. But forecast allows for flexible dynamic model building as a fully managed service, hence it is preferred.

*Marking Scheme: Product choice 1 mark, correct identification of managed services 1 mark and explaining the planned flow 1 mark.*

*(3 Marks)*

**Question 3** *(Total: 12 marks)*

a. Example of a correct detail logical architectural diagram. Other reasonable issues would be considered accordingly.




The following are examples of undesirable architectural decisions and their corrective actions:

- The packaging of the DesirableSpeed entity should be compatible with Figure 7 of the case study. That is, it should not be part of the Arrival Data Access subsystem which is part of the ArrivalTimeComputer service.
- The Desirable Time Data Access subsystem of the DesirableTimeComputer service is missing.

The above diagram also shows the correct packaging of the functional elements that were previously not yet packaged. The choice of the subsystem for a functional element should be complying with the architecture specified in section 6 of the case study.

*Marking Scheme:*

- *1 mark for the undesirable architectural decision*
- *1 mark for the corrective action*
- *4 marks for the packaging of the functional elements that were previously not yet packaged*
- *Up to a maximum of 6 marks*

*(6 Marks)*

b. One example of accepted answer is "Predicted Arrival Time Error". Other answers would be considered on their merits.

The above metric can be calculated based on 2 other metrics:

- Predicted Arrival Time (bus #, bus stop #, timestamp, predicted arrival)
  - Captured by the ArrivalTimeComputer service at the same sampling rate whenever the new position & speed of a bus is received.
  - Stored for a short time for metric calculation & troubleshooting only.
  - Can be captured centrally using centralized log shipping technology and kept as file in S3.
- Actual Bus Arrival Time
  - Captured by the telemetry in Phase 1.
  - Reuse whatever data storage used in Phase 1.

The calculation can be done as a batch process (e.g. daily) which would calculate the error for each of the prediction.

The error for each prediction can be kept if needed. This data can be summarized into daily metric for different bus stop/bus ID combination. The clean data can then be kept longer for long term tracking of the accuracy

*Marking Scheme:*

- *1 mark for identifying the metric(s)*
- *2 marks for the feasibility of the metrics*
- *3 marks for the appropriate detail of the design decisions (frequency, storage, etc.)*
- *Up to a maximum of 6 marks*

*(6 Marks)*