**Evaluation**

**Performance:**

To talk about the performance of the system i will deconstruct the overall system into subsystems, this will help to differentiate the performance of one system to another. A design pattern that I utilised heavily was the subscriber/publisher pattern, this essentially allows things to only be updated when something that is relevant to them changes. For example, the resource UI will only update if one of the resource amounts has actually been changed, otherwise remains static.

Resources:

The resource system that i implemented was one that heavily relied on two components, event driven UI updates and dictionary access for resource caching. The way this was handled was to hold a dictionary of all of the resources in the game and their player specific amounts. This made it simple to update amounts, query amounts and update the UI efficiently. In saying this there really couldn’t have been much impact from the resource system itself to toal performance.

Agents/Actions:

The initial idea for the AIAgent and AIAction system was to allow for self deciding AI, however this was cut because of time related issues and focus being placed elsewhere. In preliminary testing I devised a system that would allow all AIAgents in the scene to be updated from a central AIManager. I later developed the AIManager to a more advanced state where it would be able to update a certain percentage of all the AI in the scene over a specified amount of time. This means that you could set 25% of the AIAgents to update their decisions every 0.1 seconds, per se.

Even though the self deciding AI had to be cut, the existence of the AIAgents and AIManager were still both beneficial to the design plans set forth for the AIActions and how they would work. Which is to say that on a button press callback the UnitHandler script would tell its associated AIAgent which AIAction to process and whether or not to add it to the list or set it as the current action.

Each AIAction has its own level of complexity so it is difficult to judge it as a whole, but I took precautions in order to make calculations efficient and quick. The default included AIActions mostly utilise some vector mathematics in their calculations, nothing too intensive. However, if the previously mentioned AIManager is used then any of the inefficiencies/performance hits of the AIActions should be waylaid due to the fact that not so many are being calculated at once. A way to improve complex calculations would be to utilise coroutines in order to break up functions over multiple frames.

Selection:

The way selection is handled is pretty simple, for single selection it uses a raycast into the world to see what is being hit, and whether it is one of the scene selectables (valid selectable object in the scene). However, a lot of RTS games have the ability to drag select over multiple units/buildings which is what is also implemented roughly. I split this into Selector and SelectorUI, the second of which merely displays a box over the area the player is dragging. The Selector script handles selecting and highlighting valid objects in the scene.

Selecting, deselecting, hovering and other such things are abstracted through the ISelectable interface that makes it easy to affect valid objects that the player selects and is hovering over, such as changing materials.

The only intensive part of the Selector script is the dragging selection, but in saying that it really isn’t that much of a performance hit. It uses viewport projection to get the objects within the dragging area and check if they are valid. For highlighting this happens every frame, but selecting only occurs on mouse down and is completed on mouse up. Another use of the subscriber/publisher pattern that minimises intensive per frame calculations, this is done through the use of Unity’s new and improved InputSystem.

Networking:

On the networking side of things I am not so confident that it is the most efficient it could be, i decided that the best (or perhaps easiest) solution would be to run the entire game through the server and have the server deal with all syncing and data updating. This means that as long as latency isn’t too high then all clients and server will be in sync no matter what, because the server pretty much dictates what all clients experience equally.

Given that the server handles all calculations (and clients generally just get told values to update, or in regards to AIActions will just be accessed through their string names in a static dictionary) this makes it less intensive for clients as they don’t have to deal with any maths or calculations. This does mean that the server will be bearing the burden, and in the same regards it may put more stress on networking due to messages being sent between client and server.

**Issues:**

Networking Issue:

The biggest issue that was observed during the process of creating this modular networked RTS system was the move to making it networked. That is an oversimplification of the issue, to elaborate:

The game was initially built as a single player experience in order to thoroughly test and polish each of the systems that needed to be implemented simply and easily. Once I began to convert the single player experience to an online multiplayer experience it quickly became apparent that the way I had previously designed my systems and structures did not exactly lend itself well to networking.

Networking Solution:

Due to the fact that I had designed the game as a single player experience first I was forced to redesign several of my original systems to be more networkable. For example I had to fix the resource management and seperate it so that there was a Mod\_ResourceManager and Mod\_ResourceUI so that the Mod\_ResourceManager could be put on the player and would update the scene’s Mod\_ResourceUI when a resource value changed.

In more general terms though, the networking issue was handled by essentially forcing the server to deal with everything, though this in itself caused issues. The issues caused were the fact that AIActions were executed on the server, so player specific effects were not properly propagated to all the clients. This was fixed by having AIAgents know the player that owns them, and could access the Mod\_ResourceMod or other needed data which would then handle its own propagation to clients to update them correctly.

**Possible Improvements:**

If I were to do this project again, it would be a good idea to start building the game as an online experience from the beginning rather than adding it in later. This has been a good learning experience for my own benefit in the regards of making networked games, or I suppose any game that has a unique aspect about it. Which is to start with the unique aspect and integrate the rest of the game into it, rather than building a game and then attempting to integrate the unique aspect in after.

Other improvements that I noted are mostly superficial things rather than system changing improvements, for example after some user feedback it came to my attention that having a separate button for each AIAction on an AIAgent is not very intuitive and would be preferred if you could just right click while hovering over something and have the AIAgent figure out what it can do from there.

In saying this, there aren't any real great system changing improvements that I could imagine making at the moment, it would be possible to make some performance increases but the system works efficiently as is. I can happily say that almost all of the planned systems have been implemented and polished to an adequate level, though no network lobby was added and the AI was not fully implemented unfortunately. Given more time these are the two things that i would have liked to add to the current core system.