# WASP S.E Assignment 2

Jonas Hansson, Lund University

January 3, 2023

The human civilisation is getting evermore connected, cellphones are virtually everywhere and even daily appliances are connected and communicate with each other. With the growing connectedness, it is important to understand the fundamental limitations of large-scale feedback systems. In my research I focus on the fundamental effects of interconnecting an ever-growing number of units. More specifically I investigate local and decentralised feedback and how this effects robustness and performance of large-scale systems. As an example we can consider a vehicle formation which attempts to coordinate velocity and relative positions between a large number of vehicles. In this setup a vehicle has a limited view, and thus limited information of the system. Under this constraint we develop and analyse different control schemes. For instance, the transient phase here is of great importance as it dictates whether the formation will have local collisions or not. The contributions of my work focus primarily on fundamental results in terms of theorems. The theorems are often the result of exploration which I do through simulations of various large-scale systems.

**Automated software testing** is a concept within S.E. (Software engineering) which can be used to ensure that the developed software acts as intended. In a large project this is of crucial importance as good tests makes sure that new deployments do not introduce bugs. Many of the tests that are run are so-called unit tests which make sure that the small code units behave as intended and thus can be used as modules to build a larger program. However, with the development of larger and interconnected systems it can be hard to test all the potential use cases. In the gaming industry ML techniques and AI have been shown useful in automated software testing as smart bots can mimic and generate behavior which resembles human behavior. I think similar setups may be discovered within traditional software engineering which could make use of developments in AI. For instance, it is common within academia that code is developed in conjunction with a paper, where the purpose of the code is to validate/verify theoretical results. Here I think, AI can be useful in a distant future. Here, the task of the AI would be to validate that the code does what the article states that it does and would thus require an understanding of both the article and the written code. This sounds like a quite complex task for an AI, but as history has shown, various ML methods are remarkably good at performing seemingly complex and abstract tasks such as NLP, playing games, and identifying objects in a picture.

Today there is a vast amount of code being developed for deploying ML methods on vastly different problems. Like in the gaming industry it is actually quite difficult to test this type of code. When it comes to ML there is often an inherent probabilistic nature of the results which is heavily influenced by the data used in training and how the training has been performed. Thus, an ML based predictor may perform bad due to various reasons and a good test should thus explain what, if anything, is wrong. Once again, this is a complex task which may need further developments of ML to be fully solved.

Another factor that a software engineer needs to think of is **Code sustainability**. This concept relates to the development of code which is easy to keep updated. This is important as dependencies may change over time, and small or big changes to the code may be desired in a later stage of the code development. However, for this to be possible the code must be written and designed in such a way that it is easier to reformulate than redoing the code. A key part of this is following the best coding practises such as commenting and naming conventions. This is also an area where ML can aid a software engineer. For instance, developing code together with a copilot (like Github copilot) may increase the use of good coding habits and in the case where a copilot can help generate good comment drafts, a big portion of the formal and tedious tasks may be alleviated from the software engineer. When it comes to my research, and academia overall I think these types of tools can be of great use. Unlike more pure software engineers, academics often develop code for performing simulations or simpler tasks and often the code is only shared between a limited number of people throughout development. But the targeted code reader are the peers and future article readers which thus calls for well-written code. Without good coding standards, the risk of potential errors increase, which is bad for the overall literature.

After the code has been written in a sustainable way, then it is time for **maintenance and evolution**. Every now and then the code needs to be updated to fit the needs of the users. In large coding ecosystems there are usually lots of co-dependencies that needs be taken care of while updating modules within. The same is true when new functionality is added. To create a maintainable ecosystem the key is often the same as for writing sustainable code: the code should follow the best practises and be clear. To evolve the code it is important that the code is written in a way that makes it easy to add and modify functionality as new ideas are formed. In the realm of ML and AI we are often concerned with large neural networks that we want to deploy for various tasks. A large portion of the code maintenance here is actually related to how to introduce and use new data as it becomes available. But for this to be possible we once again have a high level objective for an AI; keep the code maintainable and with potential for growth. There may be a possibility here for a good AI to suggest potential flaws or malpractices which the software engineer can take into account in the code development. With respect to my career as a researcher I think the concept of maintenance and evolution has an important role. A normal workflow for a PhD is to work on a project which is continuously grown into a thesis. Likewise, the code can often be reused and be generalised towards new results and ideas that are developed throughout the PhD.

As can be seen throughout this text I see many potential opportunities in using ML to improve S.E. in the future and this generates the natural question: will ML take over S.E.? To answer this I think we need to take a step back and ask a more fundamental question: for whom is the software engineered? I think it is safe to say that the answer here is a human, at least in most of the cases. Similarly, the reason why tests are made, we write code sustainably, and make the code maintainable is to enable other people to understand it. As we are inherently the oracle telling whether a game works, or if code does what we want I don't think it is possible to remove the human from the feedback loop of code development. But I think the rapid development of AI and ML will benefit S.E., and this even in a near future by automating many of the tedious tasks that are included in the software engineering practise today. A major part of the work of a software engineer today is to formulate abstract and concrete problems and then to solve them through coding, this I think will remain the same. Relating it more to my academic career, I think that ML already can play an important role. One of my main tasks is to make conjectures and disprove/prove them. A good counter example generator would be of great help here. The way that I would want to interact with an AI is through a co-developing environment where I can state my conjectures and where the agent tries its best to disprove them through logical steps. In many of my setups this can simply be made through well-chosen simulation experiments which a smart AI might be able to help me develop. In conclusion, I do not believe that ML will take over S.E. but will rather evolve alongside it and act as a tool for the software engineers to develop faster and more understandable code.