# Requirements Analysis

The requirements for this project are listed below. The nature of the project required the development of theoretical knowledge before implementation began, and, as such, the requirements subtly evolved as understanding matured. The original requirements are given in Appendix [PROPOSAL APPENDIX] for comparison. The final requirements have been grouped into two categories. The first, labelled $A$, are the core requirements deemed essential to the project’s success. The second, labelled $B$, are considered extensions, aiming to improve understanding of the components used in the core implementation or further the investigation into the applications of HE in surveillance.

## Core

* \textit{implement a client-server application allowing videos to be homomorphically encrypted and transmitted in both directions.}

This component is vital because it provides the foundation for implementing and integrating all other components. It is also essential to emulate the MLaaS software stack. While conceptually simple, the nature of HE data adds many challenges to transferring videos, forming a significant portion of the investigation into HE applicability.

* \textit{implement background subtraction models that can extract moving objects from homomorphically encrypted videos.}

This requires designing and implementing the five moving object detection algorithms detailed in §\ref{sec:movingObjectDetection} so that they can act on HE data.

* \textit{evaluate the accuracy of HE inference to investigate its applicability to real systems.}

This involves analysing different metrics to understand the efficacy of moving object detection on HE data. Comparisons can be made between inference methods and between plain and encrypted data.

## Extensions

* \textit{implement a bespoke HE scheme and integrate it into the core application, providing the same functionality as the established scheme already used.}

While this implementation will likely have worse performance than existing implementations, it will offer helpful insight into the inner workings of HE. Also, it may provide opportunities for optimisations for this specific application.

* \textit{analyse the security of the encryption schemes used in the project.}

This will be useful in ensuring HE can overcome both security and privacy concerns of existing surveillance solutions and doesn’t accidentally introduce insecurities that could allow adversaries to extract information.

* \textit{implement an object recognition algorithm acting on HE data using neural networks.}

Implementations like Cryptonets [DOWLIN] have demonstrated the application of neural networks to HE. This would allow further services offered by surveillance companies to be emulated and reveal a greater insight into the limitations of HE in video analysis.

# Methodology

The different stages of the project were best suited to different development methodologies. For the core components, a waterfall development methodology was adopted [WATERFALL]. The requirements were detailed and unambiguous, so the project lent itself to a structured methodology, not requiring the flexibility of an iterative approach. The stages of the model are detailed below.

DIAGRAM

The results of the \textit{requirements analysis} stage have been detailed in §\ref{sec:requirements}. This stage is where most of the research was performed so that the project’s design would be better informed.

The \textit{design} phase incorporates expanding on the requirements into a physical project; this includes, for example, creating the class diagram shown in Figure \ref{fig:class}.

The \textit{implementation} and \textit{testing} stages were intertwined where possible in order to promote a test-driven approach to development. This was made easier by the object-oriented methodology (see §\ref{sec:OOP}) and unit testing practices (see §\ref{sec:testing}) adopted during the design stage.

The \textit{evaluation} stage replaces the \textit{maintenance} stage of the traditional Waterfall model because it is unsuitable for this project. This stage involves running experiments to evaluate the project.

When working on the extensions, an iterative model was more appropriate. The main reason for this was this section of the project was less understood so riskier than the core components. Consequently, a rapid cyclical development process requiring components to be broken down into smaller units would allow any problems to be discovered sooner, preventing the project from experiencing any costly setbacks. Therefore, development transitioned to an Agile model, depicted in Figure \ref{fig:agile}. Moreover, regular supervisor meetings allowed Agile’s sprint system to be utilised so that a prototype of the project could be presented in each meeting to ensure progress was thoroughly tracked.

A Gantt chart of the project’s timeline is shown in Figure \ref{fig:gantt}.

## Testing

### Challenges

Unlike traditional software engineering, machine learning does not provide precise criteria against which the correctness of an implementation can be verified. The models used for background subtraction are probabilistic, so the outputs cannot be precisely predicted. Consequently, a variety of testing methodologies were required.

### Unit Tests

Designed for testing atomic units of source code, unit testing takes advantage of the independent nature of components written following object-oriented principles to test functions in isolation. It does this by providing known expected, boundary, and erroneous data and ensuring the results of a function match the expected. Unit tests can be automated, making them easy to run repeatedly as changes to the source code are made, ensuring errors aren’t introduced.

Unit tests were particularly useful when developing a HE scheme from scratch. They verified the correctness of the encoding, encryption, and decryption functions and the HE Boolean circuits automatically and repeatedly.

### Integration Tests

While similar to unit tests, integration tests increase the scope of functionality covered by each test. The goal of integration testing is to ensure separate modules interact correctly. Once unit testing has been completed, these tests take the verified modules, group them into larger aggregates, and provide expected, boundary, erroneous data to ensure the output is correct.

Integration testing was useful in verifying that the software stack functioned correctly. For example, ensuring the client and server communicated correctly. While some integration testing can be automated, more complex engineering work was prioritised over creating a more comprehensive testing suite, so manual integration testing was the primary technique used.

### Manual Verification

Manual verification was used to overcome the challenges of testing the background subtraction models. Since the project involves video data, human inspection provides a good intuition of whether or not a background has been correctly removed. If a more detailed analysis is required, pixel values can be compared to check for expected results, or verify consistency across multiple tests\footnote{While humans are able to perform this verification, simple Python scripts are usually written to make testing more efficient.}.