Final Report – Team 18

Project Title:

“Codenames” Game AI

Project Sponsor:

Colin Johnson
(University of Nottingham)

Academic Sponsor:

Colin Johnson

Contributors:

Alexandru Stoica [psyas13, 20274825]

Daniel Robinson [psydr2, 20220149]

Hongjia Xue [scyhx5, 20216915]

Shahil Pramodkumar [psysp7, 20270365]

Ing Sam Yin [hfysi2, 20189487]

Ao Li [scyla3, 20217306]

Tianxiang Song [scyts1, 20217424]

Abstract

Boardgames have always been an important way to bring people together. Digital boardgames have been growing in popularity and are especially relevant now given recent global events. They allow people to connect despite being at a distance from one-another.

Our digital implementation of the boardgame *Codenames* enables people on different machines to play with each other as well-as with AI players. The project has focuses on AI and Networking and was approached through multiple AGILE sprints.

In future, the project could be built upon with new features such as allowing more players per team or by adding a player account system.

Changes from the Inter-rim report

Most of the report has been re-written to reflect the changes that we have undertaken since December. Some sections remain similar, such as Management which the team decided only needed minor changes.

Other sections, such as the analysis of libraries we've used have changed completely as we decided to take other libraries in favour of ones that were researched earlier in the project.

The "Initial implementation and testing" section present in the interim report is not present in the final report because the content has been repurposed into the Software Manual and because it was not on the marking rubric.

Lastly, there are less diagrams in the final report itself because they have been moved to the Software Manual and the Appendices where we found them more appropriate.

For links to our Gitlab Kanban Board, Code & Document Repository please see the Appendix here.

Table of Contents

MAIN REPORT 5

INTRODUCTION 5

PROJECT BACKGROUND AND UNDERSTANDING 5

Background Research and Importance 5
Understanding of the Project Brief 6
Related Work and Novel Additions 6

REQUIREMENTS AND CRITICAL ANALYSIS 7

Requirements and Specifications 7
Analysis of Approaches 10
Technical Reflection 12
Realisation Reflection 14

PROJECT MANAGEMENT AND PROGRESS 15

Project Management Approaches and Efficiency 15
Project Management Tools 15
Team Skills, Coordination and Sub Teams 17
Risk Planning 17
Management Reflection 17

CONCLUSION 18

APPENDIX 18

SOFTWARE MANUAL 21

GOALS AND DESCRIPTION 21

Terminology Definition 21

DESIGN AND ARCHITECTURE 22

Structure Overview 22
Server Architecture 26
Client Architecture 28
Design Patterns 31

PREDICTION ALGORITHMS 31

Data 31
Pre-processing 31
Algorithms 32
Parameter Setting 34

CODING CONVENTIONS 34

TESTING 35

JavaScript (JS) unit tests 35
Python unit tests 36
CI/CD 37

ADDITIONAL DOCUMENTATION 40

Dependencies 40

Deployment 40
USER MANUAL 41
PRODUCT GOALS 41
REQUIREMENTS 41
Hardware 41
Software 41
GETTING STARTED 42
Start the server 42
Single player 42
Multiplayer 46
Observer mode 49
RULE OF CODENAMES 49
MISCELLANEOUS 50
Options 50
Accessibility 50
Troubleshooting 51
Privacy Protection 52
FAQs 52

Part I

Main Report

Introduction

Throughout this last year our team has transformed the boardgame "Codenames" into a digital format allowing play between multiple people across the world while being supported by artificial intelligence. As a group we faced many obstacles along the way but through each one we united as a strong team to produce a project we are extremely proud of.

This report outlines the key stages of the development of our project, including project understanding, project development, management and reflection. The report is then followed by an appendix, a software manual and a user manual.

Project Background and Understanding

Background Research and Importance

Boardgames are a very popular pastime and are great at bringing light-hearted fun and enjoyment. One of the most well-known examples of this is Monopoly, which has sold over 275 million units since its inception (Moneyinc, 2019)[1].

Regular boardgame players are also likely to keep purchasing many new boardgames each year, a survey from 2017 indicated that approximately 65% of players bought five or more games in the year. This shows that the player-base is dynamic and consistently open to new titles and formats such as ours. (Printninja, 2017)[2]. More specific to our game, since "puzzles have remained an exciting and most dominating sources of board game products", our product is likely to be relevant to a majority of existing boardgame players. (Businesswire, 2021)[3].

Following the covid-19 pandemic, boardgames have been more important than ever at bringing people together. The digital boardgames industry has expanded greatly, exemplified by the 37.5% increase in mobile boardgame revenue world-wide over the previous year in 2020. This year, in 2022 the market is projected to make US$3.30bn (~£2.62bn) in revenue (Statista 2022)[4].

This massive interest in digital boardgames demonstrates how important they are to people's well-being. While the UK is finally easing back into normality, the pandemic has shown how useful technology can be used to bring people together over long distances or in situations where they cannot otherwise see each other. The drastic increase in interest in the games industry supports this.

---

[1] Lee, A., 2019. *The 20 Highest Selling Board Games of All Time*. [online] Money Inc. Available at: <https://moneyinc.com/highest-selling-board-games-of-all-time/> [Accessed 30 April 2022].

[2] PrintNinja. 2017. *Board Game Industry Statistics*. [online] Available at: <https://printninja.com/board-game-industry-statistics/> [Accessed 30 April 2022].

[3] Businesswire. 2021. *Global Board Games Market (2021 to 2026) - Outlook and Forecast - ResearchAndMarkets.com*. [online] Available at: <https://www.businesswire.com/news/home/20210105005724/en/Global-Board-Games-Market-2021-to-2026---Outlook-and-Forecast---ResearchAndMarkets.com> [Accessed 30 April 2022].

[4] Statista. 2022. *Board Games - Worldwide | Statista Market Forecast*. [online] Available at: <https://www.statista.com/outlook/dmo/app/games/board-games/worldwide> [Accessed 30 April 2022].

Understanding of the Project Brief

As a team we had several discussions over a few days regarding our choice of project and what it entailed. We read the project brief given to us and made sure to iron out the details of what our primary objective was.

After some deliberation, we agreed that our main objective was to create a digital version of the game "Codenames" that could sustain both Artificial Intelligence (AI) players and/or multiple players remotely.

After using the real board game version of the game provided by our stakeholder, we were able to fully understand the game rules and mechanics needed to play the game. This was especially beneficial for those on our team who have never played the game before.

Codenames is a game whereby two teams with one spymaster each try to get all their words guessed first. The spymasters who can see the colours of all the cards must provide one word and one number that their teammates must use to guess which cards are to be chosen. A randomly chosen bomb card instantly causes the team that picked it to lose. The skill needed as a spymaster is to correlate the different words that are of their team's colour and to provide a single word that can be assumed to relate to those words, whereas the spies must deduce what words are meant to be guessed based on the clues given by their spymaster.

As a team, we also had some ideas on how the game could be improved. On an online platform, there are many more things that can be done compared to an actual physical board game. One of the prime examples is the setup time, which takes far longer to do in-person than what could be done on an online platform. Taking advantage of the digital status of our project, we made our game extremely accessible and easy to play and set-up from a new player's perspective.

Related Work and Novel Additions

During the initial stages of our projects, we undertook research into existing online versions of the game (see **appendix K**) so that we could understand existing strengths and weaknesses with existing games.

Additions inspired by those found in existing works

After investigating other works, we had identified several important features that were valuable in their own versions and could be implemented similarly in our own:

We found that keeping the original style and layout of the board game made the games more enjoyable for existing and returning players.

We also found that having key elements that are easily identifiable made the games easier and more enjoyable to play as you always know the state of the game.

We have seen the use of animations in a few successful versions of the game and so we decided to incorporate the use of animations in our version to indicate key effects such as card selection.

Music and sound effects have been added to help players feel the impact of their actions, such as quiet clicks when the cursor passes over a card or when tapped. These sound options can be turned off or made quieter at the players' discretion via the settings menu.

Additions missing from existing works

We also studied features that were missing from existing projects and considered how they could make our projects stand out.

The primary feature that sets our game apart from the others is the availability of AI agents that can act as both spies and spymasters making it seem as though you are playing with real people. It allows for situations where you want to just play a quick game on your own to pass the time. AI players can fill the role vacancy of human players, making the game playable even if there are not enough human players in the multiplayer mode.

We also had the idea to build on our AI system to add a hint using natural language processing to help struggling players by giving them additional words that relate to the cards on the game board.

Additionally, we included an observer mode, whereby AI players duke it out while human players watch the gameplay. We strongly believe that this mode can help newcomers understand how to play far faster than reading from a rulebook.

Moreover, we found that other versions of the game did not truly accommodate for people with eyesight issues, so we also made the game significantly easier to play for them by letting players adjust the font size for their instance of the game. We also added an option to adjust the colours of the game to account for players who are colour-blind. These options are completely local and do not affect the other players in any way yet help improve the quality of gameplay for the players who need these options.

Requirements and Critical Analysis

Requirements and Specifications

Requirements Gathering

Even before the initial project bidding, we analysed basic requirements and applicable methods. Firstly, we decided to make a web application rather than mobile app because we are more familiar with web programming (having completed relevant modules in our first year) and because web applications can be compatible with both desktop and mobile devices instead of one or the other.

We concluded that there were three main components in the scope, namely:

- The game mechanics
- Web development for online play
- AI players

To get a breadth of requirements covering these components part of our team wrote User Stories a Use Case Diagram (please see the document in **appendix A**). The Use Case Diagram and the User Stories were written at the beginning to help us produce initial requirements. These helped us think about the different type of players and their needs.

To ensure that our requirements had depth and clarity, the other part of our team produced Personas and an Activity Diagram.

We spent just over a fortnight in this phase. We took care not to rush this phase, since the results of requirements analysis would shape our entire project from then onwards. Our entire team took part in requirements analysis, before two sub-teams were formed to draw up a Persona Board and Use Case Diagrams for the various actors. We then discussed our findings and gave our thoughts on what requirements there should be. As a result, we had breadth with 31 initial requirements needed.

To validate our requirements, we had an Internal Review to make some updates and clarifications on requirements. We also have had Focus Groups to make sure that our implementation of the game was

correct, and we have not missed any steps or rules of the game. The results are available **in appendix A.** These results allowed us to adapt our game to suit our target audience.

Prioritisation

We decided to split our requirements into Functional and Non-functional, each being subcategorised between High-priority and Low-priority. These were written into the top of the requirements document.

The high priority requirements contained the core requirements such as the game rules and the features needed for the game to function. The low priority requirements, things like saving/restoring the game etc, were not critical for the games functioning and so could be completed later when there was sufficient time.

Initially we focused on the high priority requirements since this was the foundation of our game and then during later stages of development, we were able to implement all requirements of lower priority since we had managed our time well throughout the project. Due to the frequent meetings between the sub teams, we were able to establish key components we wanted to have completed before the end of each sprint.

Scope Reduction

Due to the constant communication with our sponsor during the requirements gathering process we had a suitable scope for the project in which all requirements were able to be met. In part due to this, and our thorough requirements analysis, we did not end up having to do much in terms of scope reduction, as we felt that the planned project was satisfactory as it was. The scope was detailed at the very top of the requirements document to make sure we did not stray from it.

When defining our scope, we considered making our game playable for more than 4 players. Upon discussion amongst ourselves and with our sponsor, we concluded that this was not within the scope, and that our team should focus on making the game work with 4 players, each being AI or human, no more and no less. Note that the number of “guests” that can watch a game should be not limited however, and there could be multiple games running in different “rooms” at the same time.

Since requirements regarding user data introduced unnecessary complexity, we decided to remove it from the scope of the project. User data in this context refers more specifically to its collection. Since we did not plan on having our AI agents learn from every game played, in addition to not having players make accounts to play, we concluded that the collection of user data irrelevant and outside the scope of the project.

Considering that we are going to make a web application, in addition to the 3 main components, the scope includes:

- Customisation options
- PC and mobile Compatibility

Specifications Gathering

The Specifications have been thoroughly gathered from these requirements to make sure the game is in line with the client's requirements. By the end of our specifications, we had solidified the scope of our project and what needed to be completed in more concrete and software-oriented terms.

The game can be split up into three sections for development, so that each area is encapsulated and can be worked on independently if needed. These areas are *web development, game mechanics* and *AI development*, and for each area there are several specifications. We also made some low fidelity

prototypes to test the consistency of specifications and present how all specifications work together. (Specifications and Prototypes are available in **appendix B**).

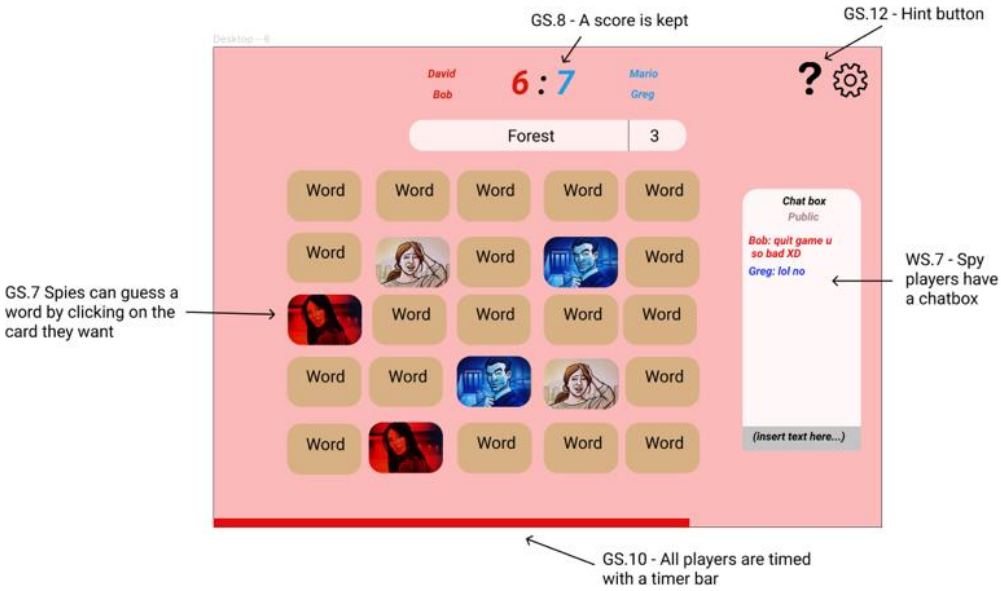
Moreover, we have done an Activity Diagram, Sequence Diagram and State Diagram for Requirements Modelling and Specification Visualizing (initial version in **appendix A&B**, and final version in *Software Manual*). Going in depth with the methodologies, the use of the state diagram allowed us to analyse and understand the different states of the game for our clients. This was especially important since the spies and spy masters both have a different view of the game and play different roles. This alongside the sequence diagram allowed us to identify and describe the communications between the different clients and the server.

We also made use of low fidelity prototypes in order to have a starting point for when we developed the software. It also helped us gain feedback and acceptance from our sponsor based on the design choices made since were able to adapt the looks and mechanics of the web pages early allowing for a more suited program for our stakeholders. We carried out external reviews which can be found in the requirements and specification documents stating our finding and the changes that can be made.

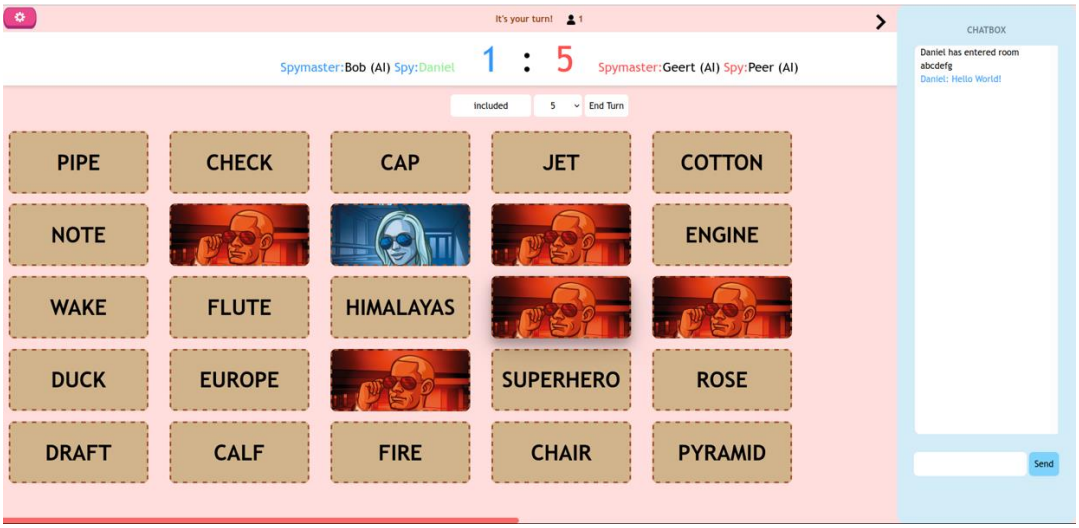
The prototype was very useful for showing our User Interface (UI) design. The board had a traditional layout so that the game was recognisable and easy to understand. The score and clue word were positioned near the top because it was an obvious location. The combination of colours and images on different parts of the board was also instrumental in our design.

The UI design of our final product was similar to the prototype but with some changes. Originally, we decided to have a constant chat box displayed on the screen but later we realised that it would cause issues on smaller devices and hence after talking to our sponsor we later decided to include a sliding chat box. This allowed for the chat to be displayed only when the user required it. We also found that the standard board size on other devices did not fit the screen size and so made the gameboard and scoreboard adaptable to fit the screen.

Despite the changes listed above, the many parts of the UI stayed consistent throughout the development of the project due to our extensive prototype validation.



An example of our low fidelity prototype



An example of our resulting game
(Note the similarities and differences discussed)

Analysis of Approaches

We considered many approaches for implementing this project, the discussion and justification is shown in this section.

Firstly, regarding software engineering methods, we went with the agile approach using sprints as it was the most appropriate as with the consistent input from our stakeholder, we can reduce the need for refactoring and maintenance. Since a traditional method would mean any changes or fixes would take place at the end of the methodology making it more expensive and time consuming to resolve compared to agile development, we are able to amend and change features at the end of the sprint. We also mitigate the risk of not meeting the required goals of the stakeholder as we receive feedback constantly not just from our stakeholder but also among the sub teams as we test the software as we go along.

As previously mentioned, we decided to make a web application instead of one installed on PC or mobile devices, which is a most convenient way for users to try our game on any devices. Besides, we are more familiar with web programming.

For hardware to test our program, we will use PCs, tablets, and other mobile devices as we decided to make this project run in the web browser. A large consideration is the layout of the webpage on all these devices.

With the approach to development, we made a quality manual (available in **appendix D**) to ensure development was consistent and of a professional standard to ensure code legibility and maintainability.

The tools and methods we planned to build this project include:

Programming Languages

- **Python** – a backend language suitable for game mechanics, prediction models, machine learning frameworks and server interfacing. Its abundant libraries provided useful tools for the project.
    - There are other options for backend languages such as *Java* and *NodeJS*, however, *Python* syntax is often simpler and some of us are able to employ strong *python* skills.

Regarding the AI part, *python* is also suitable for data analysis, making it easier to implement prediction algorithms.

- **JavaScript** – a language for user interaction with graphical interfaces and connections between frontend and backend. It is also an event-driven client-side scripting language with relative security, widely used for client-side development.
    - There are other options such as *TypeScript*, which extends functionalities of *JavaScript*, but for this project it is not necessary, and we have used *JavaScript* before.
- **HTML/CSS** - essential languages for Web Development.

With the use of these languages, we are able to fulfil our non-functional requirements such as playing the game in a web browser.

Web Framework

- **Flask** - a micro python web framework, it is lightweight and designed to make getting started quick and easy, with the ability to scale up to complex applications.
    - There are other options like *Django* and *CherryPy*, but as this project is relatively small and some of us have used *Flask* before.
- **Socket.IO** - a library that enables real-time, bidirectional, and event-based communication between multiple clients and a single server (can be implemented in both *JavaScript* and *Flask* to enable the communication).
    - There are other options in python libraries like *socket* and *websockets*, but we chose *Flask-SocketIO*, as *Socket.IO* builds on top of *HTTP* and *WebSocket*, providing extra features such as automatic reconnection and event-based notifications, and *Flask-SocketIO* is an implementation of Socket.IO using *Flask*.
- **Apache** – a website publishing framework that allows HTML/CSS/JavaScript stored in /var/www/html to be accessed from the internet. This worked simultaneously beside *Flask/SocketIO* and provides the pages and code for the web-browser to communicate with it.
    - We did not use *NGINX* because load-balancing was out of scope for our project, though we did discuss it being a possible future direction.

AI Techniques

- **GloVe** - Methods to convert word to vectors (using cosine similarity on word vectors to get word similarity). It mainly focusses on context to get words similar in meaning. We implemented two kinds of AI (Spy and Spymaster) by extracting suitable words from vocabulary using similarity on word vectors.
    - We did not use *fastText* since it focuses more on the *form* of words.
    - We have considered *word2vec* as well, the difference is that *GloVe* does not rely just on local statistics but incorporates global statistics (word co-occurrence) to obtain word vectors (Towardsdatascience 2019)[5], which could be an advantage on finding similar words.
    - We used pre-trained data from https://nlp.stanford.edu/data/glove.42B.300d.zip rather than training by ourselves since it's time consuming and the accuracy could not be as expected.

[5] Ganegedara, T., 2019. *Light on Math ML: Intuitive Guide to Understanding GloVe Embeddings*. [online] Medium. Available at: <https://towardsdatascience.com/light-on-math-ml-intuitive-guide-to-understanding-glove-embeddings-b13b4f19c010> [Accessed 30 April 2022].

- **Scikit-learn** – A python library providing simple and efficient tools for predictive data analysis. Used in AI prediction algorithm.
    - As the primary work of our AI algorithm is computing the pre-trained word vectors, most of work could done by *NumPy* library. However, as we decided to add *DBSCAN* clustering algorithm to the spymaster AI, *Scikit-learn* could be a good choice that provides efficient ways to implement the algorithm.

See appendix L for hyperlinks with more information. Also see the Software Manual.

Hardware

Initially we developed this project on our local machines and ran the server with *localhost* which worked well, however, running locally could only simulate multiplayer by opening multiple browser windows.

To realise the objective, one of us began developing and hosting our project remotely on an *AWS server*, while others continued working on their local machines. This made it possible to test the functionalities on the hosted website anytime with any devices.

Technical Reflection

Throughout the whole project we utilised all the tools mentioned above and found them to be effective for our project. We made use of existing libraries while also using new ones throughout the project (which we write about in more depth in the *Software Manual*).

We decided to use the programming languages we did as they were the most suitable for web development and with the simplicity of *Python* it was much easier to implement our game. We believe this was a good technical approach for our project as we had experience using these languages and were enabled to achieve our project goals. Also, Python works well alongside our choice of framework. Since these languages are most used in web development, future development of our project can continue using technologies we have used without the need for rewriting existing code.

Reflecting on our choice of web frameworks we strongly believe that *Flask* was a strong choice as it helped create web applications alongside *Python* much easier. Also given the time frame for this project with the nature of *Flask* being a small and lightweight framework it was much easier to understand and fully utilise in our project. *Socket.IO* is also easy to understand and it works well on communication between server and clients.

Regarding our coding method and the use of classes for structure, this aided our software development as it was a modular approach making it easier to complete separate classes during each sprint without the need for other classes. We were all also most familiar with this style of programming and so this sped up development. It also allows for easier understanding of the software and code reusability.

On further reflection of our implementation of the game, we have successfully achieved our goal of creating a fully functioning version of the board game with novel additions that sets us apart while hitting the project requirements. We understand that there are many ways we could have gone about solving this task but our method most suited our groups skills and given the time we had.

In terms of tests, there was continuous testing done to ensure functions and methods performed correctly without any logical errors as well as ensuring the looks of the user face were correct across all platforms. We later introduced some frameworks such as Chai and Mocha to do some further in-depth testing of the software. Later we had some CI/CD tests to ensure all pushes passed the tests correctly (see **Appendix M**). Initially there was some struggles to get these frameworks and the

pipelines working but we sorted it in the end. There may be some undiscovered uncertainties regarding these since they were completed quickly soon after they were introduced.

Initially during web development, the most difficult part was the communication between the clients and the server. All functions regarding sending and receiving messages must have been made to be consistent otherwise logical errors would occur as wrong information would be sent in the wrong order. However, with the planning and class descriptions prior to writing the code we were able to get everything working smoothly.

Considering our approach to implementing the AI of our game there were various methods and approaches we could have applied, and we chose to use *GloVe* as word vectors to compute similarity. With extensive research done during the initial stages we were able to understand and apply it with confidence. As we chose to use pre-trained word vectors, the relevant words can be effectively found by calculating cosine similarity of word vectors, which saved much time during the development process. The python library *Scikit-learn* has also provided very efficient ways to implement the clustering algorithms we need.

For AI prediction algorithms, we think it is designed successfully and not difficult to implement. The reason could be that we skipped the training process of word vectors which requires much time and knowledge. We followed the proposed algorithm (available in **appendix C**) at first, but we found that sometimes human players found it hard to guess based on the clue given by AI spymaster. We soon improved the algorithms by adding *DBSCAN* clustering to the clue generator, which made the clue more reasonably related to a set of words (see *Software Manual* for details).

Regarding the whole development process, initially we split that into two parts, one for webpage design and client-server communication, and another for game mechanics & AI algorithm, each implemented by a sub team. This is a success by the end of first semester, each part is implemented well as the skills needed for each side fully matches the members in each sub team.

However, we found it difficult to integrate those two parts, as this is a totally new field for all of us. As the Game-Dev team implemented a game input/output to terminal, how to call *python* script from client's web browser and send responses back becomes most important. At that time, the Web-Dev team realised client-to-client communication using Socket.IO, which also needs to start a python *Flask* server, giving us ideas that *python* scripts and *JavaScript* can be connected by the *Flask* application. Following that we successfully integrated those two parts together, making all the game mechanisms work on a website.

Another difficulty is the responsive web design, which allows web contents display well in different window sizes. We have been improving that for around three months and made a specific "mobile mode" that has new board layout on mobile phones. The layout appears nicely in a wide range of window sizes.

There are also some functionalities implemented by methods not planned before, for example, we introduced a function to store current game state to local browser and restore the board from that, by *localStorage* API in *JavaScript.* As the development process is also a learning process, we learned and applied many new technical methods during the year.

We also realised after managing to combine the Game-Dev branch with the Web-Dev branch and getting everything to work, that the AI was responding a little too quickly, making it difficult to react to. To fix it, we made the AI wait for a small period of time before it makes a move and selects a card so that it looks more human when someone is playing against it.

If we continued developing this project in the future, we would be able to easily accommodate any changes or future additions since the technologies we have used allow for easy extension. Regarding the framework it can be easily used for when our software undergoes extreme changes and extra complexities. The software is well maintained and contains clear and concise information through inline comments and documentation providing easily accessible future development for any developer.

With respect to future directions, it is possible to incorporate new libraries and technologies if required due to structured and secure development process.

As for game functionalities, we could make more than one human spies in one team, allowing more people to get involved rather than just watch. The requirements outside the scope due to unnecessary complexity could also be implemented. For example, a user account system could be established allowing players to make online friends, seeing play records, or even joining a random-matching game when there are many players trying to match others at the same time.

Making the software a mobile/PC application is also worth a try, which could provide more customisation options and complex animations that enhance user experience to a new level.

Realisation Reflection

As this project ends, we believe there are many things we have achieved as individuals but also as a group to accomplish this project. One thing we did well was conducting a thorough requirements gathering. This helped solidify a strong set of achievable requirements and specifications. This meant that there was less change in requirements later which could have slowed down the process. With the methodologies we used to gather these Requirements as seen in **appendix A,** we were able to gather requirements tailored to our target audience. This allowed us to create a game which is more likely to be played by many people.

We believe one of main achievements we have accomplished which separates us from many other groups is creating a solid foundation before we begin development. We ensured that we planned our project properly in order to mitigate any uncertainties or changes throughout the project. Hence when it came to development everyone knew what had to be done which sped up development while creating legible and coherent code. We conducted adequate preparation and analysed feasible methods for the project from modelling and validation to visualisation and prototyping to ensure we completed the project in a proper way.

Another thing that went well was during development, splitting the group into subgroups to work on the web page and game functionality meant there was less conflict during the whole project.

One thing that was slightly more difficult to achieve was implementing multiplayer with AI. Though we achieved this goal without any difficulties it took more time planning the structure of the program where clients interact with the server and other clients. However, keeping structured protocol messages and libraries we managed to achieve fast and reliable communication between clients and the server.

However, there are things not done very well in the whole process. One is that the team of Game-Dev coding for the backend did not follow the class diagram made before, probably because at the time of making class diagram, we did not do a deep search on the framework we planned to use, which causes the structure for backend classes a bit unorganised and additional time redoing the class diagram.

Another is that we write tests in the near end of development process, which does not follow the concept of *Test-driven Development*, making the coverage of test cases not very high and more unexpected problems during the development.

What we have learnt from them is that we still need to do a deeper search on the structure of our planned methods, write test cases to cover requirements before software is developed, and tracking all software development by repeatedly testing the software against all test cases.

One uncertainty we may face is how well our website performs under heavy stress and usage as we were only able to test a couple instances of the room at a time which performed very well on the server. Additionally, since the performance of AI players can only be measured subjectively, we are not able to fully confirm it could satisfy all the possible users.

Project Management and Progress

Project Management Approaches and Efficiency

At the beginning of this project, we decided to use a hybrid approach of both Traditional and Agile methods. Our supervisor and sponsor agreed that our requirements were unlikely to change greatly throughout the project and hence we made sure to benefit from a thoroughly documented requirements and specification process throughout our project. This was especially important at the beginning of the software development process as everyone was clear on how to develop the foundations for our project and how these foundations could be built upon.

As part of our hybrid approach, we have used a variety of Agile methods where appropriate. We have used SCRUM with 2-week long sprints and retrospectives. During these retrospectives we have discussed whether we have met our deliverables, how the team has found the work and how effective our methods and tools have been in meeting our targets. Our retrospective documentation is available in **appendix F**. Further to this we have had bi-weekly meetings with our supervisor, placed either just before or after our retrospectives, where we have shown him our sprint deliverables and written any feedback in **appendix E.**

From these supervisor meetings, we have been continually working on feedback. Some examples include incorporating a sliding side-panel into our design, adding additional mobile compatibility, adding clearer user feedback (through animations etc) and adding a “How to Play” section on the front page. Each of these things were suggested in meetings and implemented.

It is worth noting that while our supervisor meetings provided incredibly important advice, we did not put too much pressure on the supervisor. A goal from the start of the project was to be as self-managed as possible and so we tried not to rely too much on guidance as the project progressed.

In addition to our bi-weekly sprint and supervisor meetings, we have held regular stand-up-style meetings each week. At the beginning of our project, we did this 2-3 days a week on average with the goal of working on the design process and sharing ideas to get a clear and well-rounded project foundation. Nearer to the middle and end of the project these meetings only needed to be held once a week as the goal shifted to assigning coding roles and reflecting on different areas of development. We found that more could be done at this stage if we spent more time directly coding instead of having meetings.

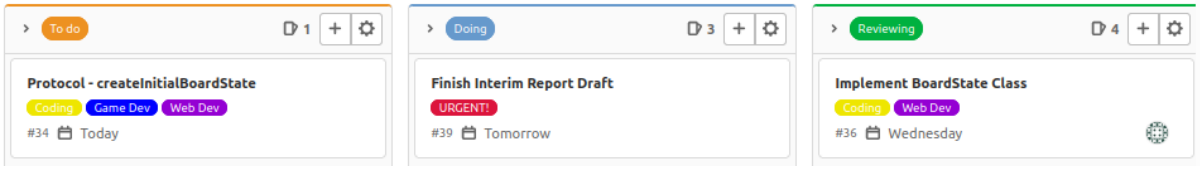
Project Management Tools

Our team decided on three different tools:

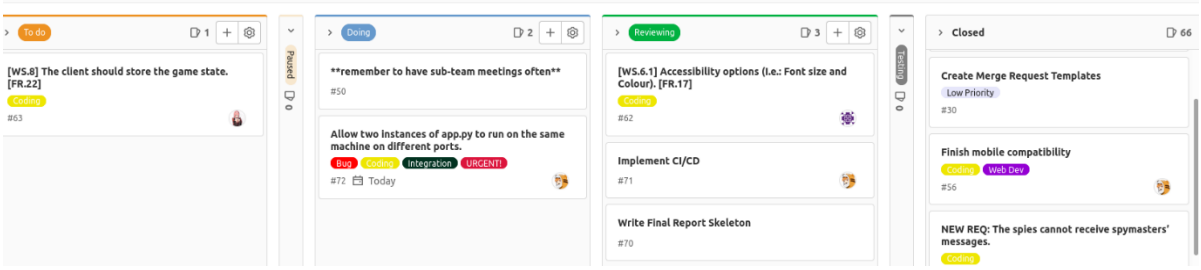
- **Discord** for online communication and messaging
- **Microsoft Teams** for file (documentation) sharing
- **Gitlab** for planning and managing workloads (see below)

Gitlab was the most crucial management tool used in this project. We decided to use it because most of the team have had experience with it in the past and because it is tightly integrated with our project repository, meaning we could link issues and milestones with git commits. Gitlab also offers additional features such as CI/CD (see *Software Manual*).

During the project we had split the workload into issues that were assigned between different members of the team (see *Team Skills and Coordination* below). Each issue was tagged to identify their features and was placed onto a board to show the progress made on them, similar to Kanban. A full list of issues, tags and milestones are available in **appendix H**, below are some examples.

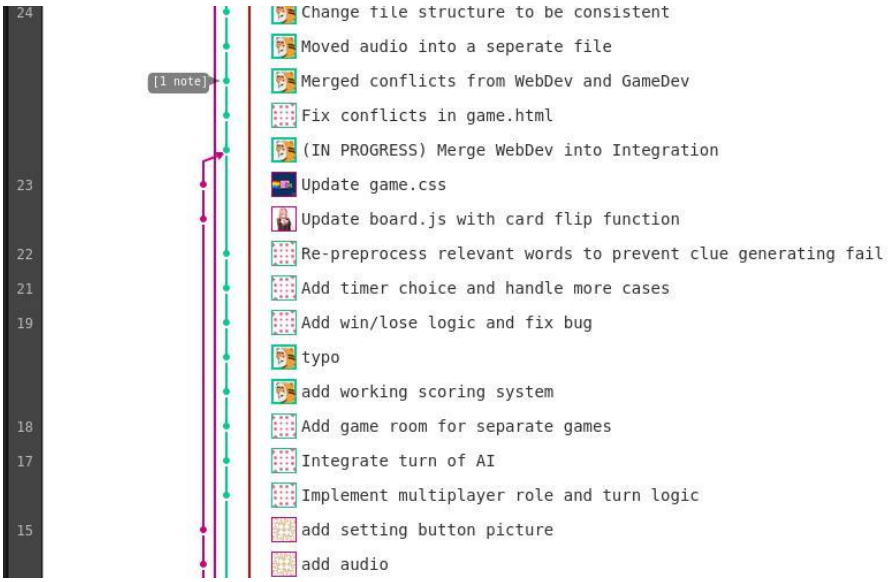


An early example of our GitLab Board



A later example of our GitLab Board

An additional feature of Gitlab that was used frequently near the end of our project was the Graph. This was shown in meetings alongside the board to see where each git branch is in relation to each other and how many commits were made to each. This was helpful in coordinating the final integration of the project and making git merges.



Part of our GitLab Graph

Team Skills, Coordination and Sub Teams

Members of the project have been active and eager to work as a team, demonstrated by a total of over 4000 individual messages in our discord server (as of 06/04/22) and by the frequency of our meetings.

Throughout the coding part of our project there have been two sub-teams, one handling the client-side (*"web-dev"*) and another handling the server-side (*"game-dev*"). The groups were initially split based on what people were most interested in and on what people were most experienced with, meaning that both teams were well-rounded.

As the project continued, some members of either team were cross trained to help with the other part of the project. This was essential as we had to integrate the two parts of our project together to make a tightly integrated and cohesive game. One example is that when developing the unit testing framework, those developing the framework needed an understanding of how both parts of the project would interact.

Risk Planning

In terms of risk-planning and contingency management, members can switch between sub-teams with ease because of the same code conventions between both teams, making newly viewed code quickly readable (again refer to **Appendix D** for more details). Additionally, members of one team should already have a good idea of what is happening on the other team due to the reviews we have in meetings, where both sub-teams share what they have done.

Early in our project one of our team members unfortunately could not work due to injury for a brief period (but has caught up and contributed very well afterwards). During this time, another team member was easily able to temporarily change sub-teams to cover the team which would have otherwise been left with a very imbalanced number of workers. The project was able to continue as normal. While a very unfortunate situation, the silver lining is that this does prove that our risk-mitigation is practical and successful.

Management Reflection

The team has worked very cohesively together and has followed everything that has been discussed in meetings. Our project management approach has been consistent, and the combination of traditional and agile methods has allowed us to have a thorough and well-documented understanding of the requirements. The one drawback of this is that we started coding relatively late, in hindsight we should have started a week earlier to start the coded prototype.

As the project continued into the later stages, we benefitted from shorter meetings and having more time to individually work on each of our assigned tasks.

The use of project management tools was appropriate for the task. We made good use of the tight integration between our issues board and our project repository on GitLab with issues being created, updated, and marked as completed in project meetings. One improvement could've been to use more GitLab milestones towards the middle and end of the project and to think about the project in a more long-term perspective more often. Having the issues board led to great productivity, but it did tend to make us think in the short-term.

Excellent team skills were employed throughout the project and there are few reservations with this. Every member had a different skill set and passion they brought to the table, and each worked hard to share this with the entire team. Perhaps an improvement could've been to have more focussed

sub-team meetings instead of general team meetings, though the perspective of the entire team meetings was helpful for cross-training.

Overall, the management style seems to have been successful as the code has been finished 3 weeks ahead of the submission deadline with *all* requirements (and specifications) met with the Supervisor's Approval.

Conclusion

In the end, we can safely say that the project went along smoothly, and we all learned quite a lot from it. For one, everyone had their strong suits, whether it was web development, game design or AI work and the way we worked around, with us talking frequently and giving each other updates, made us learn more about the processes that we were not as trained for. The way we organised the sprints and the group work was extremely well done so we did not run into any big conflicts or problems along the way. Although, the few issues we ran into we resolved them really quickly every time. In terms of the feedback that we received from our supervisor, almost all of it was positive and we took into account the additions that he suggested or better ways to do some things. When it comes to our requirements and specifications, we decided on really good ones from the very beginning so by the time we did requirements review, we only had to add a few relatively small ones which were finished in no time. In conclusion the project was a huge success, and we are really proud of the work we achieved in the time we had, finishing everything we wanted to. From a bunch of individuals with different views and ideas by the end of the project we felt like a united group with a powerful set of novel skills which we will use in the future.

Appendix

APPENDIX A – REQUIREMENTS DOCUMENT

During the first sprint of our project, we wrote a thorough requirements document listing the requirements and organising them using diagrams.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/blob/main/docs/Requirements%20Document%20V2.pdf

APPENDIX B – SPECIFICATIONS DOCUMENT

Similarly, to the requirements document above, we wrote a thorough specifications document listing the specifications and organising them using diagrams.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/blob/main/docs/Specifications%20Document%20V3.pdf

APPENDIX C – CLASS SPECIFICATIONS

In our second sprint we created a class specification document that defines each class that we would like to program and their associated methods. This also includes two class diagrams.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/blob/main/docs/Class%20Descriptions.pdf

Note that these are the initial class diagrams. With further sprints these classes evolved as shown in the Software Manual.

APPENDIX D – QUALITY MANUAL

The quality manual outlines how code will be written, reviewed, and committed to the git repository.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/blob/main/docs/Quality%20Manual.pdf

APPENDIX E – MEETING MINUTES

The notes of every meeting we had are recorded in the meeting minutes document.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/blob/main/docs/Meeting%20Minutes.pdf

APPENDIX F – RETROSPECTIVE EXAMPLES

Here are some examples of notes taken from our sprint retrospectives.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/blob/main/docs/Sprint%20Retros.pdf

APPENDIX G – SOFTWARE REPOSITORY

The code is stored in a GitLab repository.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/tree/main

APPENDIX H – GITLAB ISSUES AND BOARD

The GitLab issues, board, and milestones that we used to keep track of our project are available below.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/boards/61

APPENDIX I – MIDTERM REVIEW

Nearer to the end of our development cycle, we wrote a mid-term review to help us work out what still needed doing.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/blob/main/docs/Midterm%20Review.pdf

APPENDIX J – GITLAB GRAPH

The GitLab graph depicting git commits and their associated branches.

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/network/main

APPENDIX K – BACKGROUND RESEARCH

We looked at other existing solutions before starting development of our project. Here are some existing games:

https://codenames.game/

https://www.codegame.cards/

APPENDIX L – LIBRARY LINKS

Libraries noted in the Analysis section are detailed here:

https://flask.palletsprojects.com/en/2.1.x/

https://flask-socketio.readthedocs.io/en/latest/

https://nlp.stanford.edu/projects/glove/

https://scikit-learn.org/stable/user_guide.html

APPENDIX M – CONTINUOUS INTEGRATION

As mentioned in part of our reflection, CI/CD supported the testing process:

https://projects.cs.nott.ac.uk/comp2002/2021-2022/team18_project/-/pipelines

Part II

Software Manual

Goals and Description

The project goal is to create an online version of board game "Codenames", allowing multiple players to play against each other taking the roles of either "spy" or "spymaster". Players can also challenge AI opponents in different role and difficulty configurations.

Thus, the scope of the project includes but is not limited to:

- The game mechanics (see *Rules of Codenames* in User Manual)
- A website supporting multiplayer real-time online play
- Multiple separate game hosting (known as different "rooms")
- Game state store and recover (when playing as single player)
- Two types of AI player ("spy" and "spymaster") with configurable skill level
- Customisation options (e.g., timers, volume, colour schemes, font size, etc.)
- PC and mobile compatibility (responsive web design)

Terminology Definition

Some terminologies are defined to help readers understand this manual better.

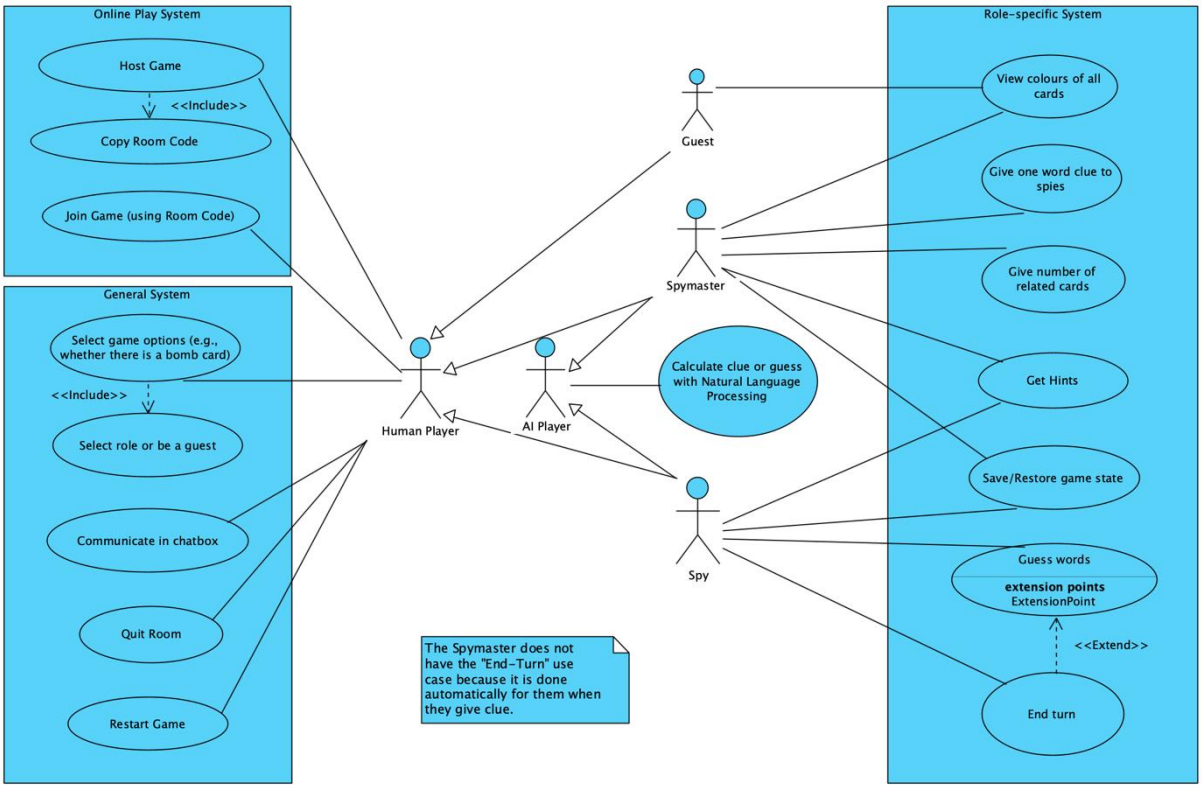
- **Codenames** – a board game, see https://en.wikipedia.org/wiki/Codenames_(board_game)
- **Spy, Spymaster** – roles of Codenames, see the link above for details (note that *spy* is the same to *field operatives* in this link)
- **Guest / Observer** – users that do not have a role but can watch a human/AI game and send chat messages
- **AI / AI player / AI opponent** – computer program that taking roles of "spy" or "spymaster"
- **AI Difficulty / AI skill level** – AI accuracy for selecting suitable cards. A harder difficulty means higher AI accuracy.
- **Room** – a specific channel for a set of players to play games, there is no limit on the number of people in a room, but at most 4 players can get a role, while others being guests. No communication between different rooms.
- **Host / Host User** – player that hosts a game room, can invite others to join, can configure game options and can start/restart game
- **Chat box** – a side panel for players to send messages to each other within a room
- **Timer** – timer for a player's turn that can be configured by host user. If the timer runs out in the turn of one team, that team fails.
- **Bomb Card** – a board card that can be configured by host user. If a spy picks it, that team fails.
- **Clue** – a word given by spymaster that relates to a couple of board words
- **Target number / Max guesses** – a number given by spymaster that indicates how many board words are related to the clue, and limits the max number of guesses by spy in this turn
- **Game state** – the information of an ongoing game (board card, current score, role assignment etc.)

Design and Architecture

Structure Overview

Use Case Diagram

Below is the Use Case Diagram for the different types of players that will play this game. This diagram demonstrates how the different types of actors will interact with this game functionally and how many use-case related requirements are needed by actors.



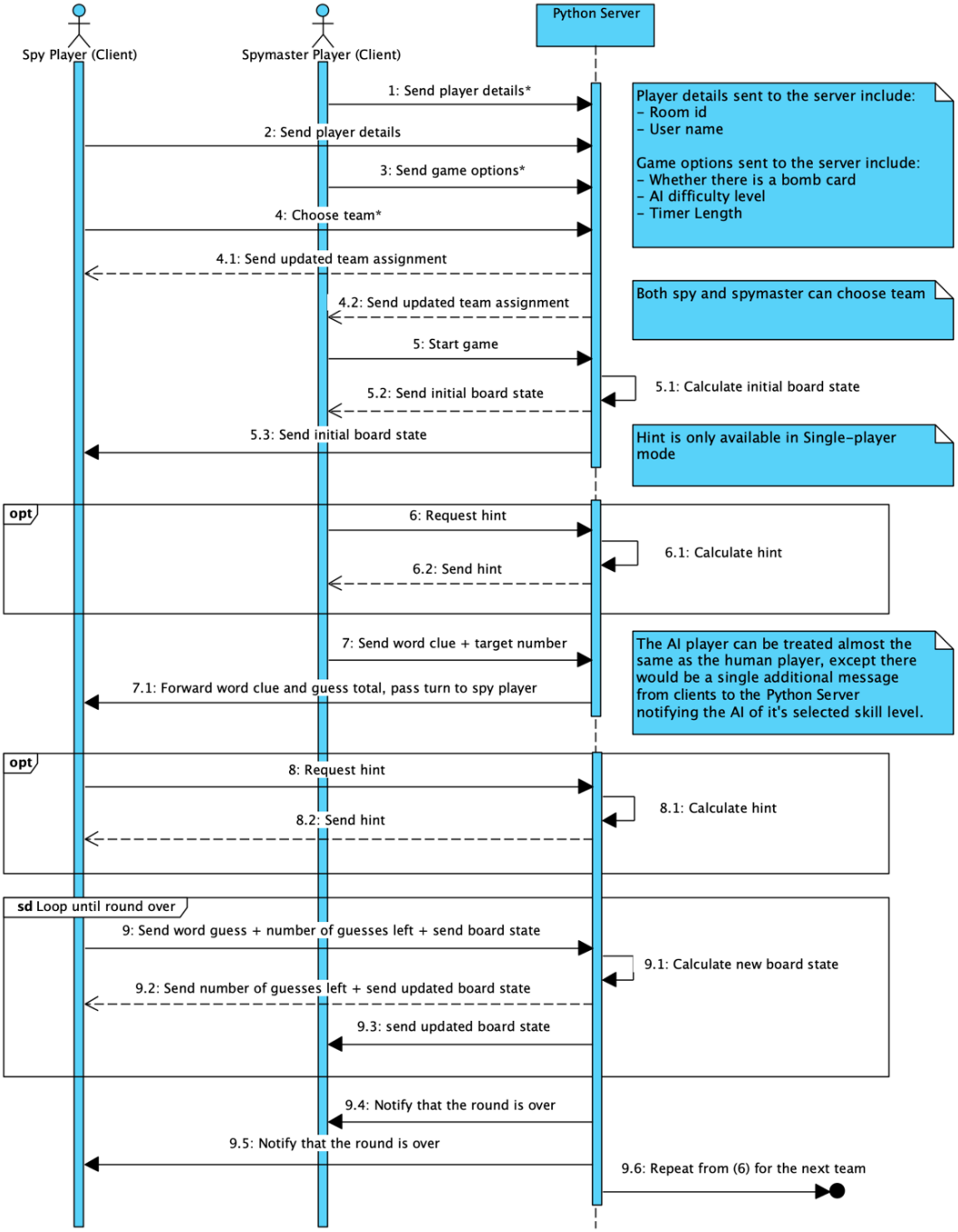
The Actors used are *Human Player, AI Player, Spymaster*, *Spy* and *Guest*. The Spy and Spymaster have access to all the use cases of either the AI player or the Human player depending on which type of player they are. The Guest can only watch the play and chat, having the view of Spymaster.

There are three main systems, the *Role-specific system*, the *Online play system,* and the *General system.* The role-specific system includes the specific use cases of human players only. The online player system includes the online-multiplayer functionality, such as creating and joining a game. This system will also run on a web-browser. Finally, the general system includes functions available to all human players, such as writing in the chat box.

Note that some use cases are not available in both single-player mode and multi-player mode. For example, in single-player mode, players cannot see room code and use chat box, while in multi-player mode, players cannot get hints and save/restore game state.

Sequence Diagram

Below are two Sequence Diagrams showing how the server interacts with clients. Note that the spymaster is assumed to be the host in the first diagram.

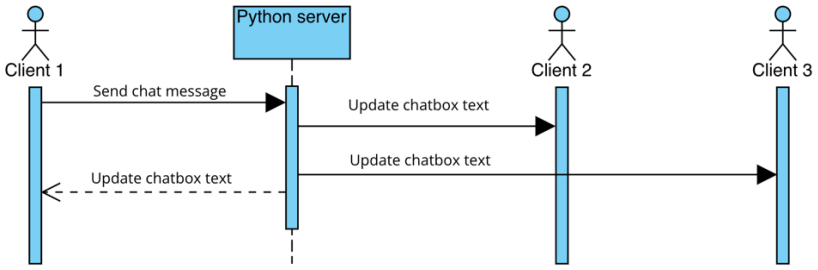


The first Sequence Diagram shows a series of events during the game. Firstly, the game starts with the host selecting the options for the game and then all players choosing their team and roles. After host user confirm to start, the server will send the initial board state to the players, which will be stored on the client's side.

After a move is made, the client sends the changed board to the server, and the server will send it back to all the clients. The hint request is optional and limited to 3 times in the whole game. If a turn is taken by AI player, a request will be sent to server to call the prediction function to make a move and send the new board state back.

When the game is over, the host user can choose to either quit the room (and other users in the room are forced to quit) or restart the game (and other users in the room are forced to restart), while other clients can only choose to quit the room (and all clients are notified).

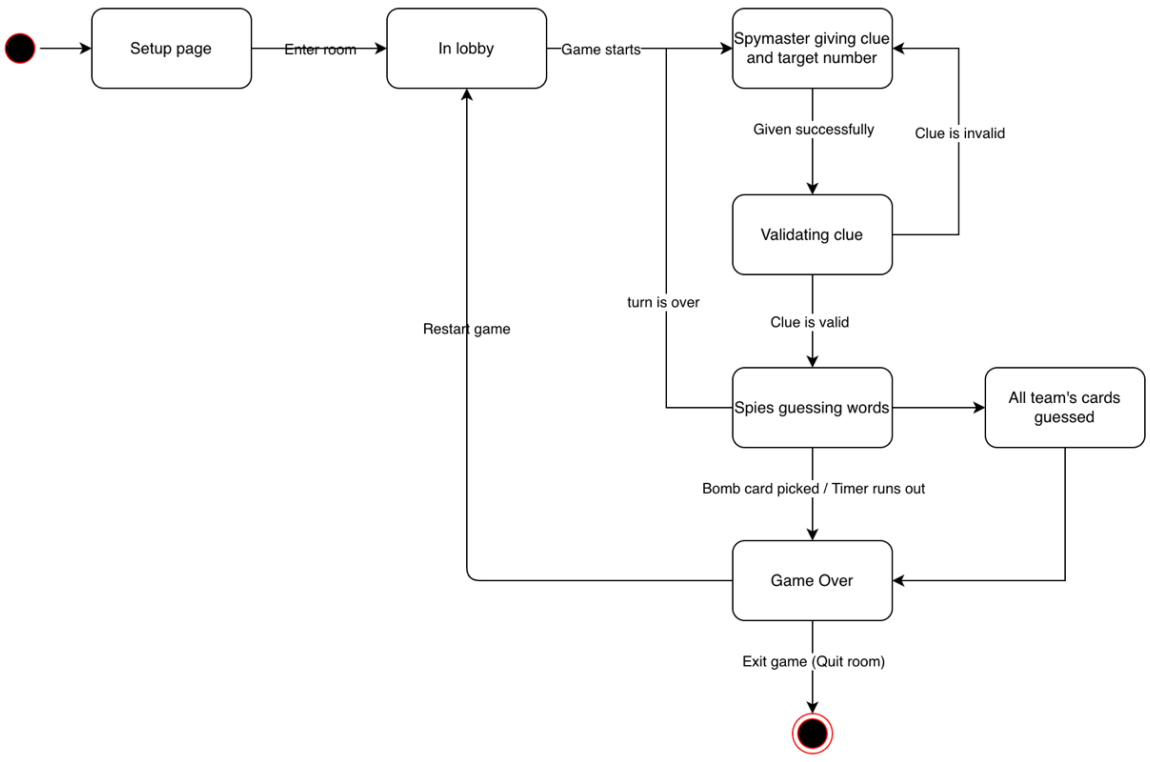
The functionality of chat box is shown in the second Sequence Diagram below (the number of clients is not limited to 3). Chatting can happen during role-selection and any time after game starts. Any chat message is received by all clients in the room. Note that although players and guests can both use the chat box, spies cannot see messages (replace message with something else) coming from spymasters or guests due to the risk of clue leak, while spymasters and guests can see all messages.



State Diagram

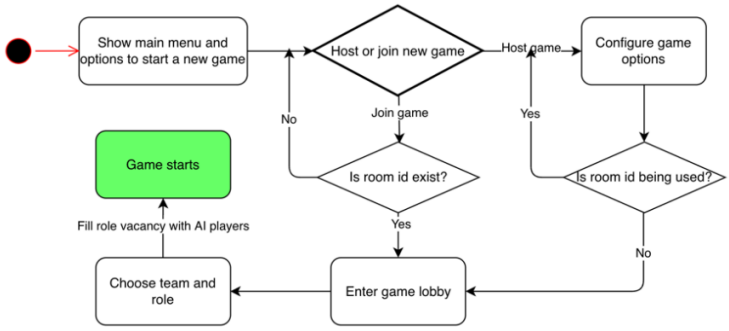
The state diagram shows the main states that the game can reasonably be expected to be in. This state diagram starts when either single player or multiplayer is selected from the main menu. In setup page, host user can configure game options (e.g., room name, bomb card, timer length and difficulty in single player mode), while other users can use room name to join the room. In the lobby, players can select role for themselves, and the AI players will fill the role vacancy. When the game starts, the spymaster of one team will input a clue to be given to their spies, the game will validate it to check if the spymaster gave a legal word, being that it cannot be a word on the board. If the clue is valid, the spies will try to guess the word(s) hinted at. Once the spy has chosen some card(s) or ends their turn early, the game will go back to an idle state as the other spymaster begins their turn.

The game carries on until a bomb card is picked or timer runs out, in which case the game is instantly thrown into the game over state, or until a team has no words remaining to pick, in which case they win, and proceed to the game over state as well.



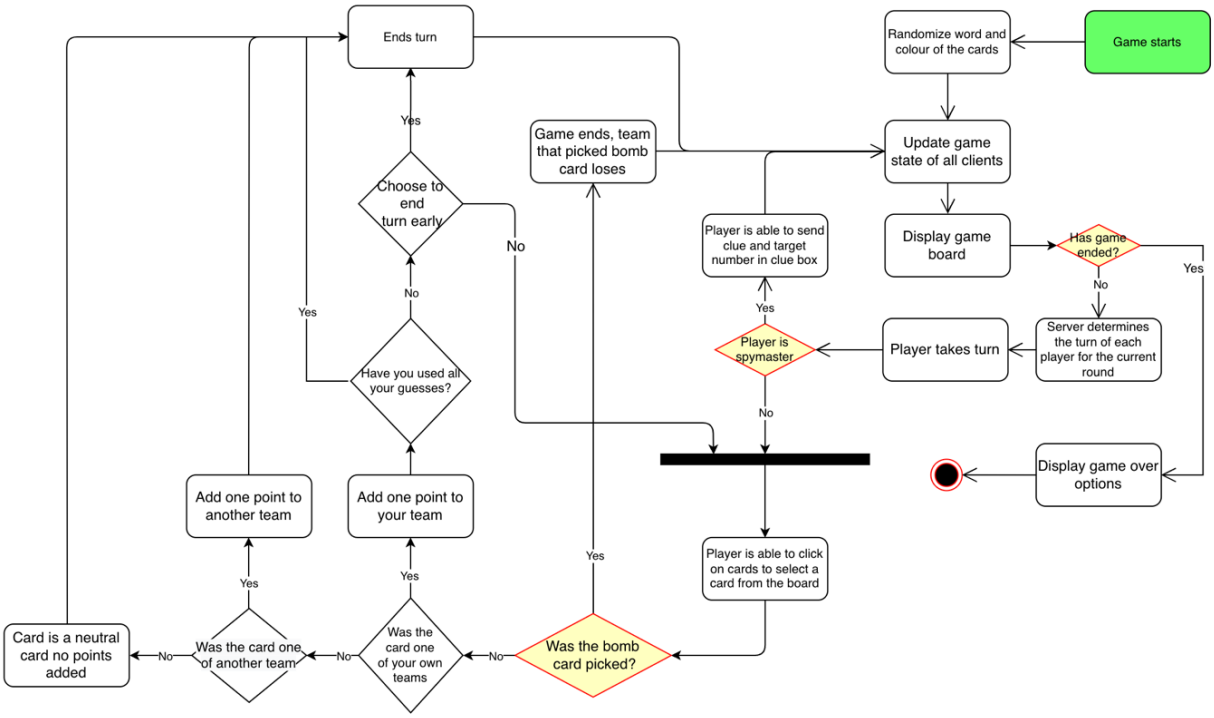
Activity Diagram

Below is the activity diagram representing the progression and actions a user would take when using the web app to join/create and play a game. The first part represents flow and actions the player can make before the game (corresponding to "Setup" and "In lobby" states in State Diagram). In the step of "Host game / Join game", the host user cannot name a room that is the same to another room already hosted, and other users cannot join a room that is not hosted (invalid room id). In the step of role selection, at most 4 players can be assigned a role (two teams, one spy and one spymaster per team), and if one does not select a role, he/she will be guest in the game.



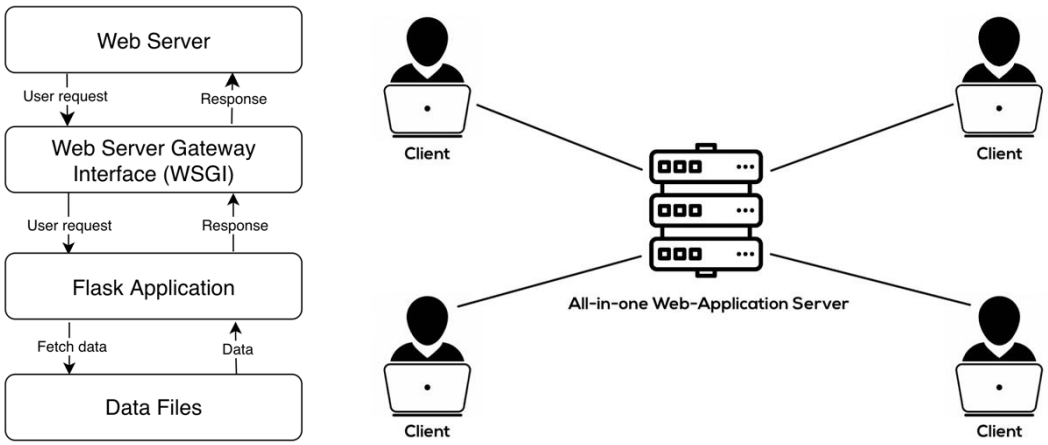
Continuing to the second half of the activity diagram corresponding to the rest of the State Diagram, which represents actions a player can make after the game starts. Note that only common activities in both single-player and multi-player mode are included (i.e., chat and hint activity are not included). Additionally, if a timer is set and runs out before turn ends for both roles, the game ends as well.

In the step of "Get hints" (in single player mode), the spymaster can get a set of clustered words (words with relation) which can help them conclude a suitable clue and the number of targets, while the spy can get two words, one is a correct word relating to the clue, and the other is a word randomly picked from cards not belonging to one's team. When the hint limit is reached, players can no longer request hint in the rest of turns.



Server Architecture

Logical and Physical Architecture



The four-tier logical architecture of the server is shown above, and all those functional elements execute on one physical device (known as the All-in-one Web-Application Server). Thus, the physical architecture is simply the machine itself connected to all the clients.

The *Apache* web server serves contents (e.g., *html*, *CSS*, *JavaScript*, and images) to load on client's web browser, and the *Gunicorn* (or *Flask-embedded*) WSGI server accepts client requests and sends it to a *Flask* Application producing functionalities of the game. For requests that call the prediction function (invoke AI player), the application will get pretrained word vectors stored locally to make prediction and give responses back to the web server.

Libraries

The external libraries and framework used in the server side is all python-related and their functionalities are as follows:

- **Flask** – A micro web framework. https://flask.palletsprojects.com/en/2.1.x/
- **Flask-SocketIO** - Gives Flask applications access to communications between the clients and the server. https://flask-socketio.readthedocs.io/en/latest/
- **NumPy** – Provides an array object of arbitrary homogeneous items, fast mathematical operations over arrays and random number generation. https://numpy.org/
- **Scikit-learn** – Provides simple and efficient tools for predictive data analysis. Used in AI prediction algorithm. https://scikit-learn.org/stable/
- **Gunicorn** – A Python WSGI HTTP Server (for production). https://gunicorn.org/
- **Eventlet** – A concurrent networking library for Python that allows you to change how you run your code, not how you write it (for production). https://eventlet.net/

And used python standard libraries are:

- **pickle** - Read byte stream in a file/database and convert it into a Python object. Used in loading word vectors and relative words.
- **itertools** - Functions creating iterators for efficient looping.
- **random** – Random variable generator.
- **time** – time access and conversions. Used in getting current time when saving game state.
- **sys** - System-specific parameters and functions. Used in specifying server port when starting python server using command line.

File Structure

The directory structure for server side is:

- src
  - game
    - boardGenerator.py
    - predictor.py
- rsc
  - data
    - codenames_words
    - relevant_words
    - relevant_vectors
    - word_dict
    - rooms
- app.py

Where the files in blue are python files running in the server that handle requests from the clients and send messages back. The files in *rsc/data/* are to store relevant information for the ongoing game, *rsc/data/rooms* will be written during the game while others are read-only. The Flask Application starts by running *app.py* and execute other python files when necessary.

Data Storage

As an instant game, no user data will be stored on the server. The only file written during the game records list of room id, while other data generated in the use of application will be stored in clients' web browsers. The usage of those server-related data files is:

- **rooms** (read and write): store names of current hosted rooms as strings (one name per line). Read as `list<string>` in python. When a host user enters a new room id to host, a request will be sent to Flask Application to read the *rooms* file to check if the room id exists. The same process happens for other users that join a room.
- **codenames_words** (read only): store 400 candidate words for board cards. Read as `list<string>` in python.
- **word_dict** (read only): store 9592 words (not including candidate card words) for AI spymaster to select clue from. Read as `list<string>` in python.
- **relevant_vectors** (read only): store words from *word_dict* and their corresponding word vectors in *binary format.* Read as `dictionary<string, numpy.ndarray>` in python.
- **relevant_words** (read only): store words from *codenames_words* and their relevant words from *word_dict* in *binary format.* Read as `dictionary<string,list<string>>` in python.

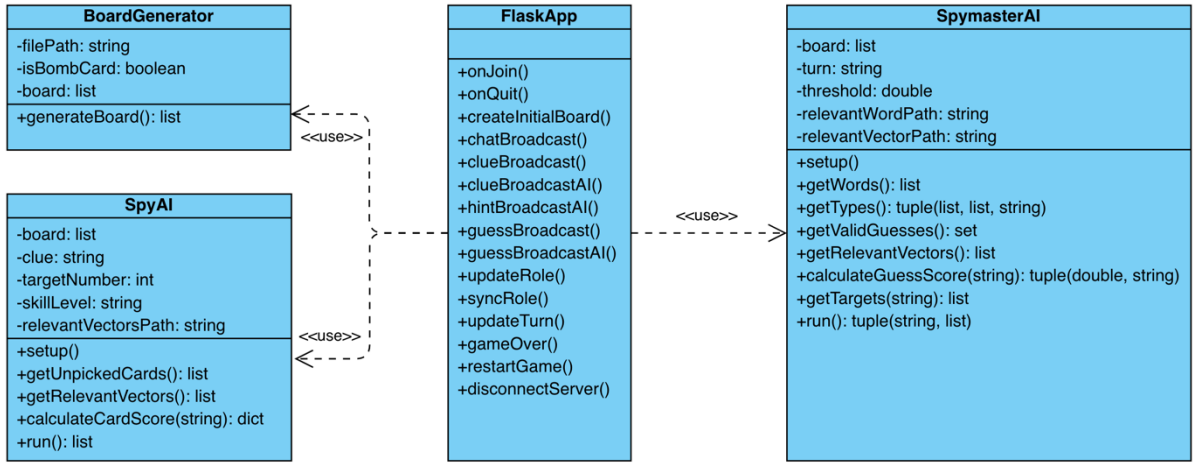
Classes and Functionality

The functionality of server-side code is mainly as follows:

- Accept join room request from clients and grouping clients to different channels.
- Accept synchronisation request from one client and send game state to all clients in the room.
- Accept start game request from host user and initialise a random board to send to all clients in the room.
- Accept AI invoking request from host user and invoke AI player to make a move and send new board state to all clients in the room.

- Accept hint request from one client and invoke AI player to generate a hint to send directly to that client.
- Accept quit room request from clients and disconnect them from the channel.

Below is the class diagram for the server-side.



Where *FlaskApp* is in app.py, *BoardGenerator* is in boardGenerator.py, and other two classes are in predictor.py.

Running *app.py* will start the Flask application (by default listening on port 5000). The methods defined in FlaskApp are invoked by requests sent from client to application via web server and WSGI server, and results are sent back to clients. Please refer to inline documentation for detailed information of these methods.

A *BoardGenerator* instance will be initialised in method *createInitialBoard()* in FlaskApp, which creates a list of cards on the board in the format of `list<dictionary<string,string>>`. The two keys of dictionary represent the word on card and the type of card respectively.

A *SpyAI* instance will be initialised in method *guessBroadcastAI()* in FlaskApp. By calling *run()* method, a list of words will be returned as guesses based on *board, clue, targetNumber* and *skillLevel* parameters passed to the instance at initialisation.

A *SpymasterAI* instance will be initialised in both methods *clueBroadcastAI()* and *hintBroadcastAI()* in FlaskApp. By calling *run()* method, the generated clue and a list of target words will be returned based on *board, turn* and *threshold* parameters passed to the instance at initialisation.

Client Architecture

The client-side code runs on user's web browsers. Thus, this part will mainly focus on the data storage and class/file functionalities.

Libraries

Socket.IO - a event-driven JavaScript library for real-time web applications. It enables real-time, bi-directional communication between web clients and servers. https://socket.io/

File Structure

The directory structure for the client side is:

- pages
  - game.html
  - instructions.html
  - setup.html
- rsc
  - audio
  - images
- src
  - web
    - audio.js
    - board.js
    - chatbox.js
    - randomId.js
    - server.js
    - game.css
    - main.css
- index.html

Where *html*, *CSS* and *JavaScript* files are marked different colours. The main functionalities of these files are:

- **index.html** – page that the browser loads first when visiting the website of project directory, which has links to other *html* files in *pages/.*
- **instructions.html** – page of instructions on how to play the game.
- **setup.html** – page of game configurations for three modes (single player, host multiplayer and join multiplayer) before entering the room, has link to *game.html.*
- **game.html** – page for the game, including role selection and game process.
- **board.js** – main functionalities of the game, including game mechanism, client synchronisation, local state storage and local settings.
- **audio.js** – loading audio files and setting volumes.
- **chatbox.js** – functionalities of the chat box.
- **randomId.js** – random room id and nickname generator.
- **server.js** – handle client-server communications.
- **game.css** – style sheet mainly for game board.
- **main.css** – style sheet for elements outside the game board.

Data Storage

Two methods are used for data transfer, store and recover in client side:

- **Query strings** - A query string is a part of a uniform resource locator (URL) that assigns values to specified parameters. The query string is parsed by *URLSearchParams()*. It is used in data transfer between webpages, for example:
  - **index.html -> setup.html** – a query string `choice=n` is appended to the URL (e.g., https://codenames.uk/pages/setup.html?choice=0), where *n* is decided by game mode chosen in *index.html.* And in *setup.html*, the value of *choice* is parsed to decide which mode to display.
  - **setup.html -> game.html** – a query string `game.html?choice=n&room=_&nickname=_&isBomb=_&timer=_&difficulty=_` is appended to the URL, where values in '_' are decided by game

configuration in *setup.html.* And in *game.html*, these values are parsed to setup the game.

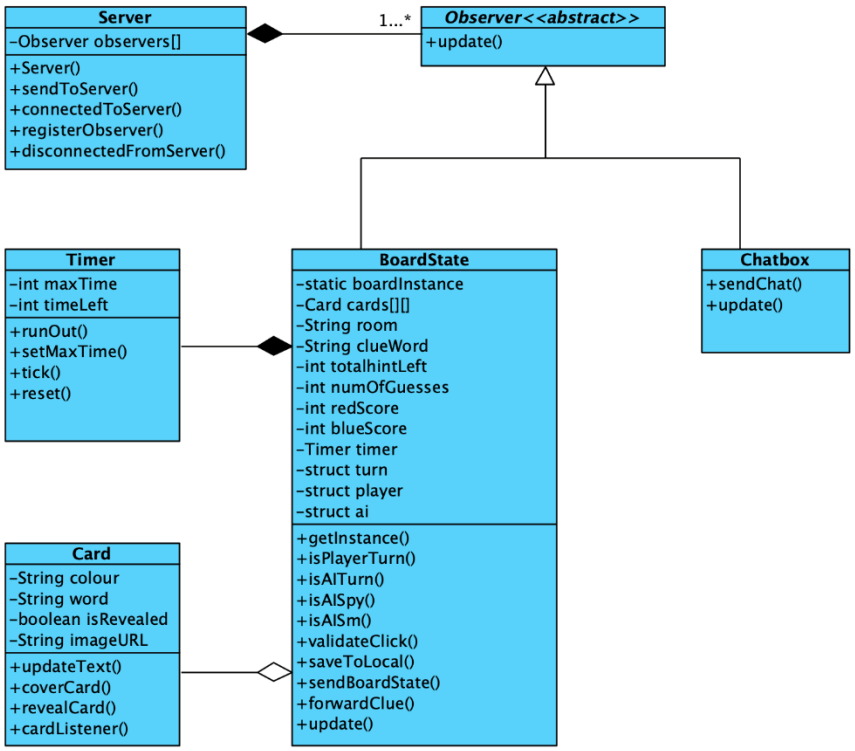
- **localStorage** – A read-only property of the *window* interface allowing users to access a *Storage* object for the *Document's* origin. It is used in Single-player Mode to store/recover game state in/from the browser. For example:
    - **store** – when users request to save game state, the syntax
      `localStorage.setItem('state', JSON.stringify(data));`
      stores board information to a current domain's local *Storage* object named "state", where *data* is in the *JSON* format of
      `{time: string; nickname: string; vocabulary: any; difficulty: string; timerLength: any; clue: any; numberOfGuesses: any; totalHintsLeft: number; redScore: number; blueScore: number; currentTurn: {...;}; ... 9 more ...; redSm: string;}`
    - **recover** – when users request to recover game state, the syntax
      `const data = JSON.parse(localStorage.getItem('state'));`
      loads board information from a current domain's local *Storage* object named "state", and the board will be recovered to the last saved state by reading properties from *data*.

Classes and Functionality

The functionality of client-side code is mainly as follows:

- Show contents of webpages
- Provide interface for user to interact with the application
- Perform game logic and mechanism
- Handle communications with the python server

Below is the class diagram for JavaScript:



Where *BoardState*, *Timer* and *Card* are in *board.js*, *Server, Observer* are in *server.js*, and *Chatbox* is in *chatbox.js*. Note that the classes listed above only implement part of functionalities. For information of other functionality, please refer to *File Structure* section.

When starting the game, a *Server* object will be initialised first, which handles communications to the python server. Following that a list of *Observer* objects (including *BoardState* and *Chatbox*) will be added to *Observer List* listening to updates from the *Server*.

BoardState is a class for storing and accessing properties of the game board, all attributes (excluding *player* which stores team/role of each client) between clients will be synchronised every time a move to the board is made. The *update()* method handles a list of events that python server sends to clients.

A list of *Card* objects is associated to *BoardState* to store information on each board cards. A *Timer* class implements the timer for countdown in every turn. *Chatbox* is another *Observer* class that listening from python server and updating text in chat box.

Besides, there are many methods written outside classes providing functionalities like handling changes to settings, sending data to server, saving board state and so on, please refer to inline documentation for more information.

Design Patterns

Two design patterns are adopted in the project:

- **Observer Pattern** – as demonstrated in client-side class diagram, two classes *BoardState* and *Chatbox* extend the abstract *Observer* class by implementing *update()* method, and *Server* class has a list of *Observers.* When *Server* object received messages from the python server, all its *Observers* will be notified and updated.
- **Singleton Pattern** – implemented within *BoardState* class, which has a static instance of itself, and a static method to get the instance.

Prediction Algorithms

Data

The prediction algorithms are built upon cosine similarity between word vectors. The pre-trained *GloVe* word vectors are downloaded from https://nlp.stanford.edu/data/glove.42B.300d.zip (42B tokens, 1.9M vocabulary, uncased, 300-dimension vectors, 1.75 GB download). For more information about *GloVe* and training methods, please refer to https://nlp.stanford.edu/projects/glove/.

Pre-processing

The 400 board words in *rsc/data/codenames_words* are all commonly seen and chosen from https://boardgamegeek.com/thread/1413932/word-list. For reducing computational load, words with relation to each board word have been extracted from the key set of word vectors and stored in *rsc/data/relevant_words*. The threshold of relevance is set to 0.4 (i.e., words having cosine similarity of word vectors higher than 0.4 is considered relevant), so that each board word will have a list of relevant words stored in a file.

Next, the set of these relevant words are stored in *rsc/data/word_dict* and their word vectors are stored in *rsc/data/relevant_vectors*. The relevant vocabulary and vector set make prediction faster and more precise.

Algorithms

There are two kinds of AI players in Codenames so two corresponding prediction algorithms have been implemented.

Spymaster AI

Spymasters have the view of all colours of board words, when invoking spymaster AI to make a prediction, the parameters given are the *board cards* and the *team playing during the current turn*.

The goal of spymaster is to find a clue word with relation to a set of board words, while preventing the clue being relevant to words with negative effect. Based on that, the first thing is to categorise **unpicked** board words into:

- ***goodWords*** – words for the spymaster's team (as indicated by *turn* parameter)
- ***badWords*** – words other than *goodWords*
- ***bombWord*** – the word on the bomb card (if applicable)
- ***goodClusterWords*** – the clustered words in *goodWords* obtained by *DBSCAN* algorithm applying on word vectors of *goodWords*, which have relation between each other. Note that the cluster chosen should have more than one word and have most words among all clusters. There could be no cluster in some cases.

Besides, the set of **candidate clue words** is obtained by reading the *relevant_words* file, taking the union of relevant word set of each ***goodWord.*** Then a score of each clue word will be calculated.

Since two words with cosine similarity of word vectors higher than *threshold* (set to 0.4) are considered *relevant*, **each time** when calculating a clue score, we get two new word sets:

- ***bestGoodWords*** – the subset of *goodWords* that each word is *relevant* to the clue
- ***bestBadWords*** – the subset of *badWords* that each word is *relevant* to the clue

thus, the score of each candidate clue word is calculated by:

$$score_{clue[i]} = \sum_{j=0}^{m} sim(clue[i], bestGoodWords[j]) + \sum_{j=0}^{n} sim(clue[i], goodClusterWords[j]) - \sum_{j=0}^{p} sim(clue[i], bestBadWords[j]) - sim(clue[i], bombWord)$$

Where m, n, p are length of ***bestGoodWords, goodClusterWords*** and ***bestBadWords*** respectively, and $sim(v, w)$ represents cosine similarity of vector representation of words *v* and *w*. Note that the parts regarding ***goodClusterWords*** or ***bombWord*** could be 0 if there is no word cluster or bomb card is not set. The equation follows the idea that is:

"The best clue should be relevant to as many good words as possible (particularly if these relevant words have relations between each other) and should be opposite to as many bad words as possible (particularly the word in bomb card)."

After calculating score of each candidate clue, the one with highest score is chosen as *clue* to return. Finally, the set of unpicked board words is sorted by similarity to the *clue*, and the largest contiguous subset starting from most similar word with all words being the same team as this spymaster is returned as *targetWords*, which has the length of *target number* the spymaster is supposed to give.

Thus, the pseudocode of spymaster prediction algorithm is:

```
INPUTS: board, turn, threshold

FOR card IN board.getUnpickedCards():
          IF card[team] == turn THEN
                    goodWords.append(card[word])
          ELSE
                    badWords.append(card[word])
                    IF card[team] == “bomb” THEN
                              bombWord = card[word]

goodClusterWords = max cluster of DBSCAN(samples=[vectors of goodWords], minCluster=2)

DICTIONARY relevantWords<string, list> = getRelevantWordsFromFile()
FOR word IN goodWords:
          candidateClue += relevantWords[word]

FOR clue IN candidateClue:
          bestGoodWords = [word FOR word IN goodWords AND similarity(clue, word)>threshold]
          bestBadWords = [word FOR word IN badWords AND similarity(clue, word)>threshold]
          score[clue] = sum([similarity(clue, word) FOR word IN bestGoodWords AND
                              goodClusterWords])
                    - sum([similarity(clue, word) FOR word IN bestBadWords AND bombCard])

chosenClue = KEY FROM score.keyset() WITH MAX VALUE

unpickedWords = [card[word] FOR card IN board.getUnpickedCards()]
sortedWordList = SORT unpickedWords BY similarity(chosenClue, word) DESCENDINGLY

FOR word IN sortedWordList:
          IF board.getTeam(word) == turn THEN
                    targetWords.append(word)
          ELSE
                    break

RETURN chosenClue, targetWords
```

Note that for requesting hints, spymaster AI is also invoked in the same way. The only difference is that for returned values, only *targetWords* is used as hint for both spy and spymaster human players.

Spy AI

Spies cannot see the colour of board cards so the spy AI should only give guesses based on the clue word and the target number given by spymaster. Additionally, the AI skill level is based on prediction accuracy of the spy AI (in the opposing team of player in single player mode). Therefore, when invoking the spy AI, the parameters given are *board words*, *clue word*, *target number* and *skill level*.

The spy prediction algorithm is thus simply choosing a set of similar words with the clue limited by the *target number* and affected by the *skill level*. Firstly, sort the **unpicked** board words based on cosine similarity of word vectors between clue and board word from high to low as the *guess list*, and take the first (*target number*) words as guesses.

Typically, for the AI spymaster with the highest skill level, the generated guesses could be all correct since the *target number* given by AI spymaster is exactly the max number of correct guesses starting from the top of *guess list*. The only exception is that when length of *targetWords* is 0, the AI spymaster will give 1 as *target number*, making it possible to guess a word not belonging to spy’s team.

As the normal setting of the AI team in single player mode is nearly unbeatable, in the step of choosing guesses from *guess list*, there is a possibility (decided by *skill level*) of choosing words outside of the first (*target number*) words of *guess list*, making it possible to include words not belonging to spy’s team.

Thus, the pseudocode of spymaster prediction algorithm is:

```
INPUTS: board, clue, targetNumber, skillLevel

unpickedWords = [card[word] FOR card IN board.getUnpickedCards()]
sortedGuessList = SORT unpickedWords BY similarity(clue, word) DESCENDINGLY

guesses = sortedGuessList[:targetNumber]

IF length(sortedGuessList) >= 2*targetNumber THEN
          FOR index IN range(targetNumber):
                    IF skillLevel == “Easy” AND random() < 0.3 OR
                       skillLevel == “Medium” AND random() < 0.2 THEN
                             guesses[index] = sortedGuessList[targetNumber+index]
RETURN guesses
```

Note that the configuration of *skillLevel* only applies on spy AI in the opposing team of player in single player mode (i.e., for multiplayer mode or AI teammates of human player, *skillLevel* is always set to “Hard”, which means highest accuracy).

Parameter Setting

There are some parameters that have effect on the performance of the prediction algorithm. The most important one is *threshold* for spymaster AI.

As stated, *threshold* represents the similarity acceptance. It is set to 0.4 by default, and the effect of its adjustment are:

- **Increasing** – words with higher similarity will be considered relevant, the size of the *word cluster* by DBSCAN algorithm could be **smaller**, and the length of *bestGoodWords* and *bestBadWords* could be **shorter**, making the samples for calculating clue score **fewer**.
- **Decreasing** – the opposite effect.

There is a trade-off. Having more samples for calculating the clue score does not mean the result is better, since large numbers of irrelevant words could make the best clue “averaged”, reducing the relevance between the best clue and its target words, making them less clear for human spy to guess. However, having less samples could make the best clue not relevant to any word at all.

A further two parameters are *random number thresholds* for different skill levels of spy AI. The default value for “Medium” level spy is 0.2 (i.e., 20% of times making a mistake) and 0.3 for “Easy” level spy. Increasing the values will make them less accurate.

Coding Conventions

For all code:

1. Variables are **declared at the top** of the method/function.
2. **Comment above** every function/method with:
    a. What the function **does.**
    b. What the function takes as **parameters.**
    c. What the function **returns.**
3. (Where possible), keep functions shorter than **75 lines** long.
4. (Where possible), write functions with less than **5 parameters.**
5. **Spaces** between operators.
6. Use **constants** instead of literal numbers.
7. Classes are written in **PascalCase.**

For the server-side code (python, as following conventions of *flask_socketio*):

- Methods, functions, and variables are written in **snake_case.**

For the client-side code (*.html, .css, .js*):

- Methods, functions, and variables are written in **camelCase.**

Testing

The unit testing covers the JavaScript client-side and the Python server-side and should be run before each git push. These tests can be easily extended to cover new functionality.

JavaScript (JS) unit tests

Dependencies

The JS tests use *Mocha* (https://mochajs.org/) for the testing framework and *Chai* (https://www.chaijs.com/) for syntactic sugar. If not already installed globally, these can be installed locally by navigating to the `/test/js/` directory and installing them with the node package manager.

```
$ cd /$repo_dir/test/js
$ npm install mocha
$ npm install chai
```

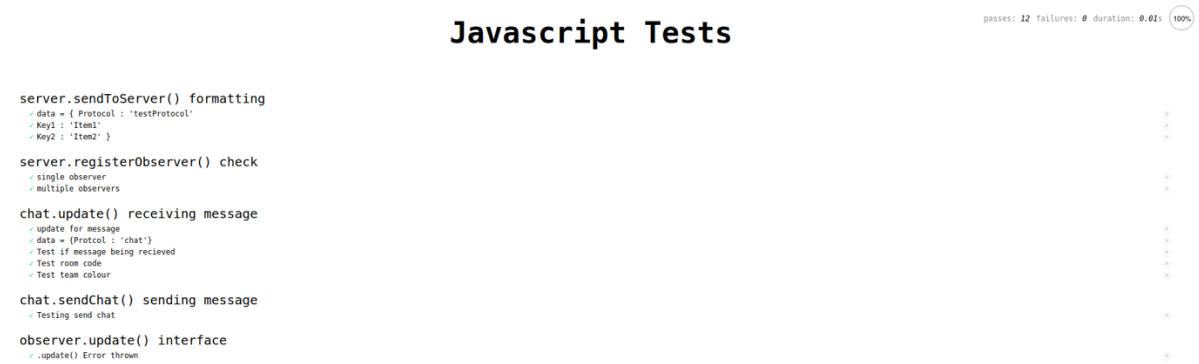
If there are any issues with the installation, consult the official *Mocha* and *Chai* documentation linked above.

How To run

The framework runs in a HTML file, and it tests functions from the client-side JS such as chatbox.js and server.js. To run the tests open `/test/js/runner.html` in a browser such as Firefox or Chrome.

```
$ firefox /$repo_dir/test/js/runner.html &
```

The resulting page should look like this:



How to extend

The tests are in `/test/js/test.js`. They are very simple to write and use the `assert` syntax described in the chai documentation. A simple test is written as follows:

```
describe("chat.sendChat() sending message", function () {
    var chatTest = chatbox.sendChat();
    it("Testing send chat", function () {
        assert.equal(chatTest.chatText, '');
    });
});
```

A group of tests begins with a *describe* function call where the first parameter is the description of the group, and the second parameter is a call-back function.

The call-back function contains general setup code for the tests followed by *it* functions that themselves have a first parameter describing the test and a second parameter with another call-back function.

The most inner call-back function has *assert* statements that determine whether the test has passed or failed.

The above example tests that the chatbox.sendChat() method returns a blank string when given no parameters.

Since JavaScript is an interpreted language, added tests will be automatically included in `runner.html` once written.

Python unit tests

Dependencies

There are no dependencies for the python tests themselves.

For the python server dependencies see *Server Architecture.*

How to run

There are two different methods of running the tests.

It is recommended to run them on a Linux machine with `/test/python/runTests.sh`. This BASH script will automatically run the tests and the python server at the same time and will report the output:

```
$ cd /$repo_dir/test/python/
$ ./runTest.sh
Running app.py on port 5001...

Running tests...
Template Test: Pass
Chat Protocol Test: Pass
Forward-clue word test: Pass
Forward-clue number test: Pass
Forward-clue turn change test: Pass

Closing app.py...
[1]+  Terminated              ( cd ../../; python3 app.py $TESTING_PORT >
/dev/null 2>&1 )

***
Done, 5/5 passed.
***
```

If you do not have access to a Linux machine, then the tests will need to be run manually without a runner script (or you can create your own windows/mac compatible equivalent).

1. Run the python server manually on a different port to the default (using the first command line argument).
2. Run the test script manually on that same port concurrently.
3. Close the test script with an interrupt after all tests have come back or there is a timeout.

4. Close the python server.

How to extend

The tests are located at `/test/python/test.py`. They are designed to test the server protocols and so are split into two parts.

The first part is the server input. The server is sent a message by using `client_.emit()` where the first parameter is the protocol name, the second is the input payload and the third is always `namespace='/'`.

Also note that most inputs require a *room.* This is set up as a global variable of the same name in the test script.

```
input = {
    "Protocol" : "chat",
    "message" : "Test Name: Hello World",
    "room" : room,
    "team" : "red",
    "role" : "spy"
}
client_.emit('chat', input, namespace='/')
```

Test Input Example

The second part of the test attempts to receive the server's reply with `@client_.on()` where the first parameter is the protocol name and the second is always `namespace='/'`.

The expected output can then be tested in a function where the first parameter is the data that the server has sent in reply.

Assert statements are used to track whether the test fails and should be written in the following format syntax:

`print("NAME_OF_TEST: ", end="")`

`Assert (TEST_STATEMENT), tFail("FAIL_MESSAGE"); tPass()`

```
@client_.on("chat", namespace='/')
def chat_test(data):

    expectedOutput = {
    "Protocol" : "chat",
    "message" : "Test Name: Hello World",
    "team" : "red",
    "role" : "spy"
    }

    print("Chat Protocol Test: ", end="")
    assert (data == expectedOutput), tFail("The chat should be in the expected format!"); tPass()
```

Test Output Example

CI/CD

GitLab Continuous Integration and Development allows the unit tests and any additional environment tests to be run automatically whenever a commit is pushed to the GitLab "main" and "Integration" branches.

Setting up a GitLab runner

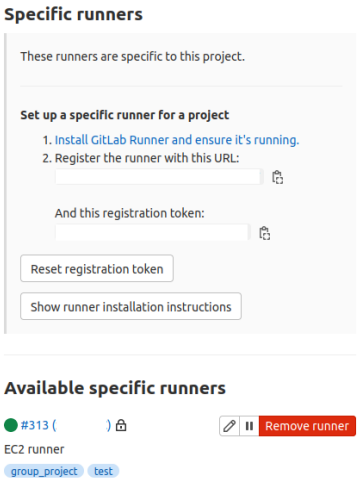
To run tests autonomously, a “runner” needs to be configured. This requires a Linux machine that has good connection to the internet and has a good and reliable uptime. If you do not have access to this, skip CI/CD.

To configure a runner:

1. Install the runner onto the target machine through a public repository (or otherwise, see https://docs.gitlab.com/runner/install/linux-repository.html). You will need superuser privileges to do this.

```
$ curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh" | sudo bash
$ sudo yum install gitlab-runner
# as of 06/04/22
```

2. Register the runner on GitLab using the data provided in your GitLab repository settings. (see https://docs.gitlab.com/runner/register/)




```
$ sudo gitlab-runner register
```

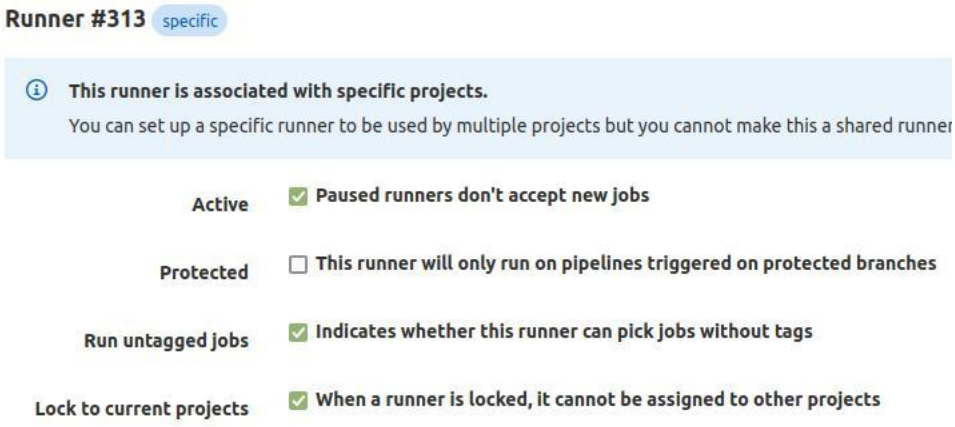
3. (optional) If you have dependencies for this project that are installed globally on a specific user, make sure that the runner runs in that user-mode instead of as root. (see https://stackoverflow.com/questions/37187899/change-gitlab-ci-runner-user)

```
$ sudo gitlab-runner uninstall
$ sudo gitlab-runner install --user $your_username
$ sudo reboot
```

4. Make sure all dependencies described in this document are installed on this machine.
5. Start the runner

```
$ sudo gitlab-runner start
```

6. Configure the runner so it can run untagged jobs.



The runner should now be up and running.

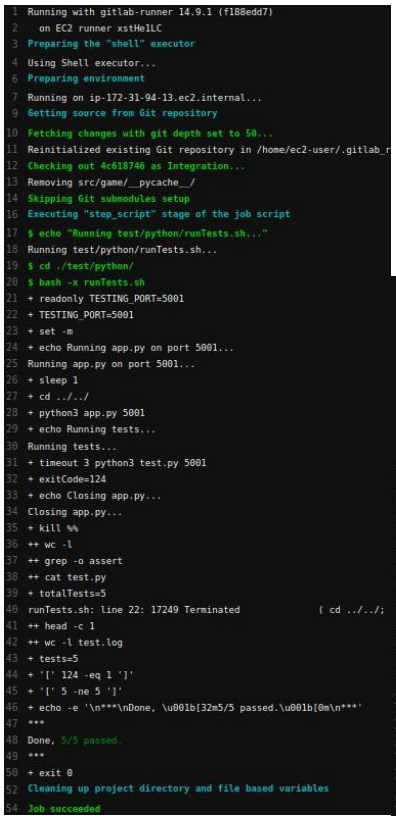
For more help on installing GitLab runners, see the GitLab documentation at https://docs.gitlab.com/runner/.

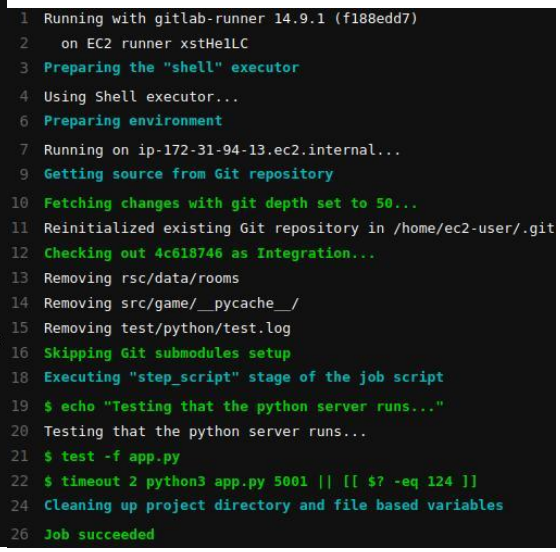
Configurations

After being setup, the GitLab runner will run everything described in `/.gitlab_ci.yml`. Statements are formed using GitLab's syntax YAML found here https://docs.gitlab.com/ee/ci/yaml/.

The file is divided into stages that are run as GitLab jobs. Standard Linux commands can be provided in each `script` section.

The *Server_Start* stage uses the `test` and `timeout` commands to see if the server starts up properly. The *Protocol_Tests* server runs the `/test/python/runTests.sh` script. More stages can easily be tested when extending the project. The output after pushing a commit should look similar to the images below.





Note that as the test scripts are run on a different port to the default, the runner can be installed on the same machine as where the main game is being hosted.

Additional Documentation

Dependencies

Compulsory

- Flask - https://flask.palletsprojects.com/en/2.1.x/
- Flask-SocketIO - https://flask-socketio.readthedocs.io/en/latest/
- Gunicorn - https://gunicorn.org/
- NumPy - https://numpy.org/
- Scikit-learn - https://scikit-learn.org/stable/

Test

- MochaJS - https://mochajs.org/api/
- ChaiJS - https://www.chaijs.com/guide/styles/#assert

Deploy

- Gunicorn - https://gunicorn.org/
- Eventlet - https://eventlet.net/

Deployment

There are many options to deploy the Flask-SocketIO server. In this section, the most used options are described. Suppose the private IP address of the server is *172.31.94.13,* and the public IP address is *3.83.45.21*. Make sure the *Apache Web Server* is started first by `sudo service httpd start` or equivalent.

Embedded Server

The simplest deployment strategy is to use Flask development server, which is not intended to be used in a production deployment.

Firstly, open *app.py* and scroll down to the bottom, edit the last line to be:

```
socket_.run(app, debug=True, host="172.31.94.13", port=serv_port)
```

then, open *src/web/server.js* and edit the second line to be:

```
const IP_ADDRESS = "3.83.45.21";
```

finally, in command line at project root, run:

```
python3 app.py
```

the Flask development server will start.

Gunicorn Web Server

To use Gunicorn WSGI server for production deployment, make sure *gunicorn* and *eventlet* are installed using pip or other tools.

Following the same procedure in deploying embedded server, and run:

```
gunicorn --worker-class eventlet -w 1 app:app --bind 172.31.94.13:5000 --daemon
```

in command line instead, the Gunicorn web server will start. Note that without `--daemon` it works as well and will have output to the terminal.

Part III

User Manual

Product Goals

Our project's goal is to build a web-based game called 'Codenames', a popular word-guessing game including both multiplayer and single player modes. The core of the game is to give clues and guess the corresponding words, and during the game, players can have the pleasure of guessing what the other players meant.

Requirements

The easiest way to try this game is visiting our project website http://www.codenames.uk/, where you can play on a stable online version providing full functionalities and learn more about our project. As the webpage is optimised for both mobile and PC, all you need is a device with internet connection and a modern web browser.

For users who want to install and run our project locally, please refer to the following part of this section. Note that the local version only allows you to play on one physical machine (i.e., you can only play single player, or play as "multiplayer" on different browser windows on the same machine).

Hardware

There is no exact hardware requirement for running the project. Any modern computer supporting *python3* and having a modern web browser should be able to run it locally.

Software

For *python* environment, it is recommended to use python version higher than *3.7.* The required external libraries are (by default the latest version installed by *pip3* or equivalent):

- Flask - https://flask.palletsprojects.com/en/2.1.x/
- Flask-SocketIO - https://flask-socketio.readthedocs.io/en/latest/
- Gunicorn - https://gunicorn.org/
- NumPy - https://numpy.org/
- Scikit-learn - https://scikit-learn.org/stable/

It is supported by following mainstream web browsers with higher version than listed:

Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
4	12	3.5	8	10.5	4	37	18	4	11	3.2	1.0

Getting Started

Start the server

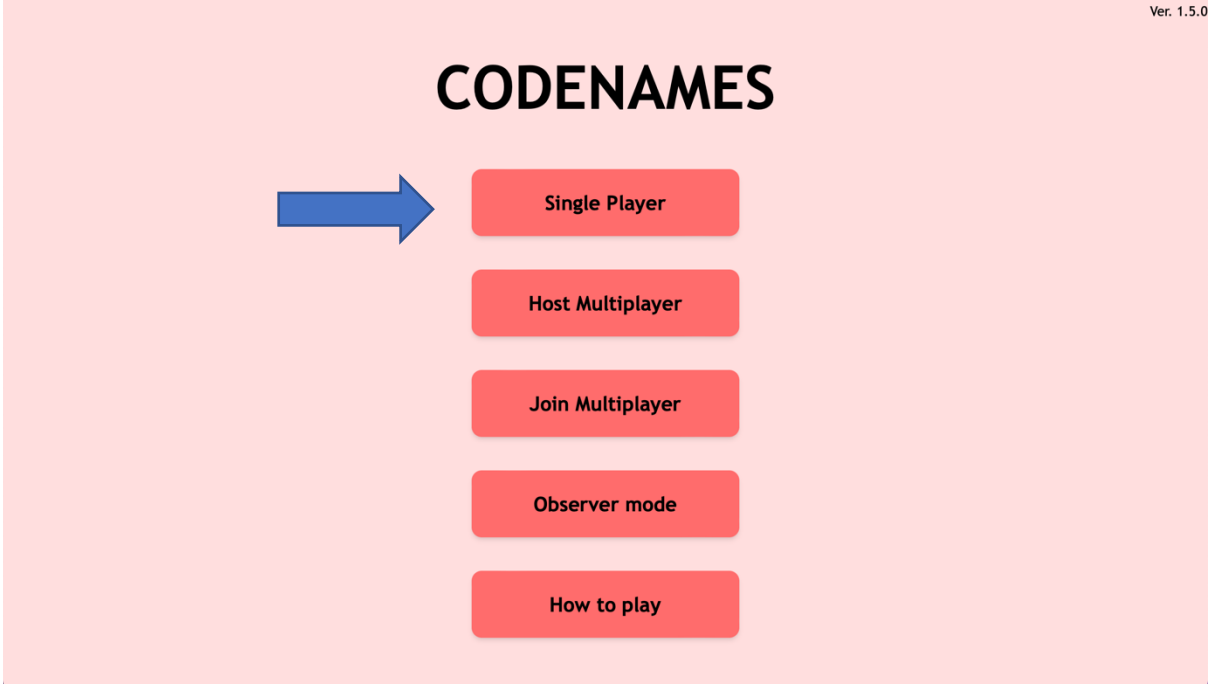
After installing the required libraries, go to project root directory, and start the server by running `python3 app.py` in command line. Open *index.html* and you will be able to see the game menu.

Single player

For single player, you're able to join whichever team you prefer and challenge our AI agents. Besides, there are multiple AI difficulty levels for you to choose.

Starting a game

Assuming you're using a PC as a running example, after entering our game, here is the welcome page shown below:



Here you click the 'Single player' to begin with. You will be directed to game setup page.

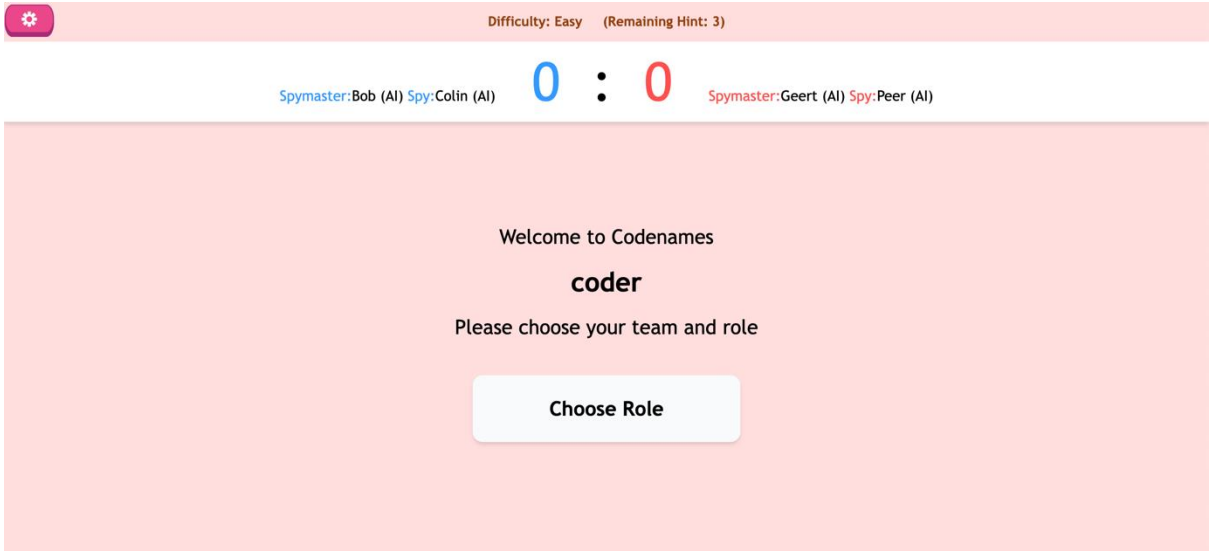
The first line “Name yourself” is to name your own game id. If you do not want to set it manually, the system will automatically generate one for you.

The second line “Difficulty” is to set the skill level of AI opponents from low to high (easy to hard).

The third line is to decide whether there is a Bomb Card on the board, which will end the game immediately if someone picks that card.

The last line is to decide whether to set a “timer” which will limit how long you have before your turn ends.

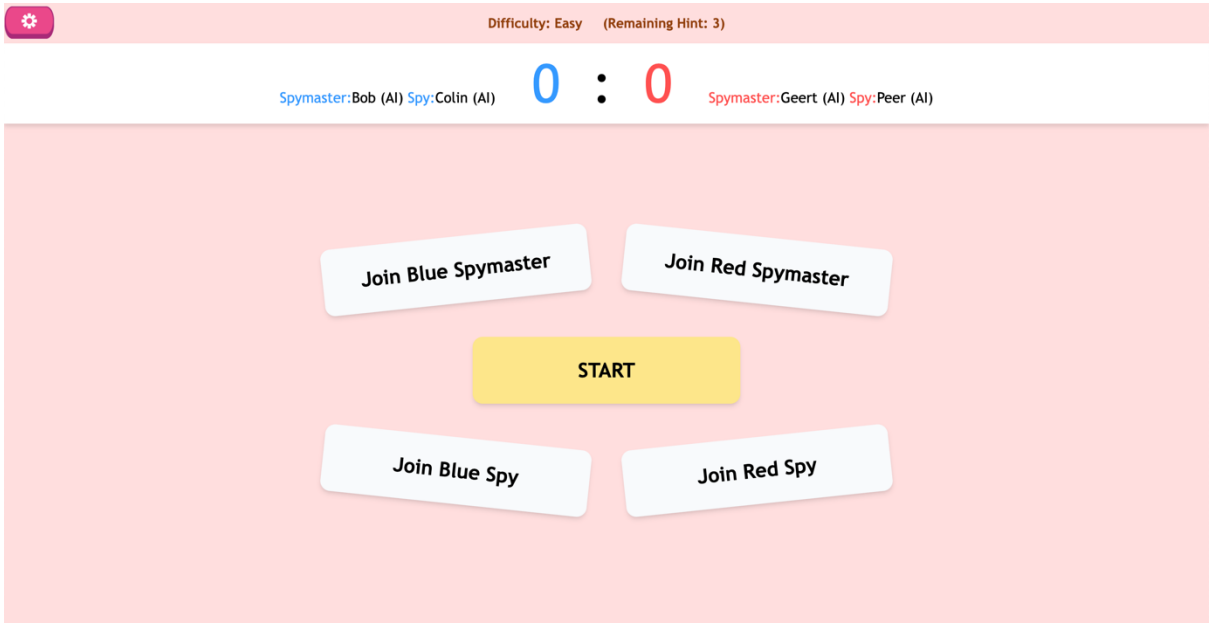
You will be directed to game lobby shown below when you confirm the setup.



Here you choose your role for the coming game.

There are 2 teams, red and blue. Each team has 2 roles: spy and spymaster. Their detailed rules will be expanded upon later.

After picking your role, you can click ‘START’ and finally enjoy your single game against our AI gamers.



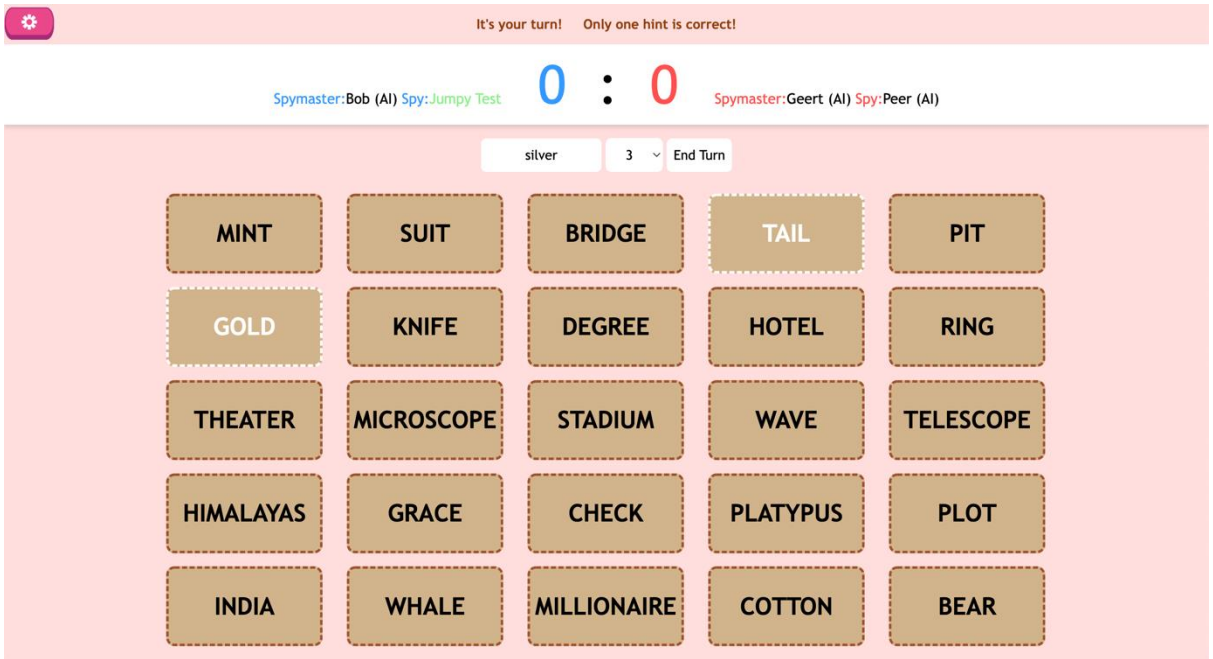
Below is the common layout of game board (spy) in Single Player mode:

Note that the Timer Bar becomes green in other player's turn. "End Turn" button will skip your turn immediately before you have made all your guesses.

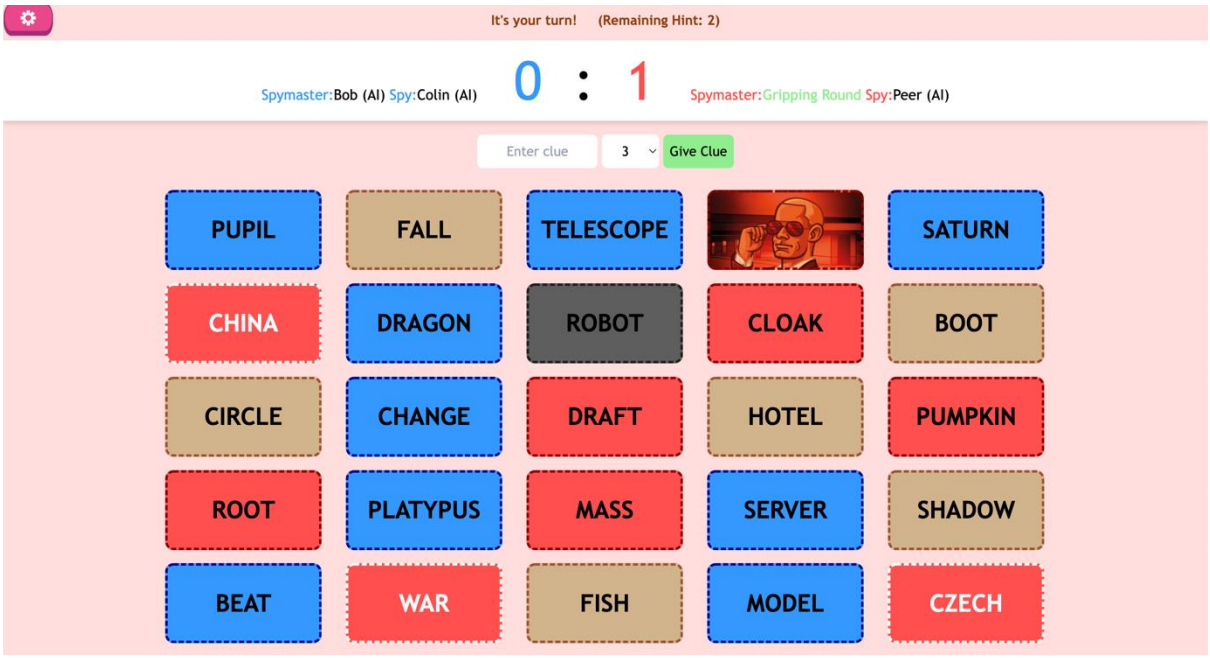
Getting hints

Players could get confused when the given clue word is vague, or the board words are difficult for them to come up with a clue. In single player mode, there are totally 3 hints for either spy or spymaster players in one game, and only one hint could be used per round.

When play as spy, after pressing "Hint" button, two cards in the board will be highlighted, and **only one** card is in your team, the other could be any card outside your team (e.g., in the case shown below, "Gold" is the correct card which is related to the clue "silver").



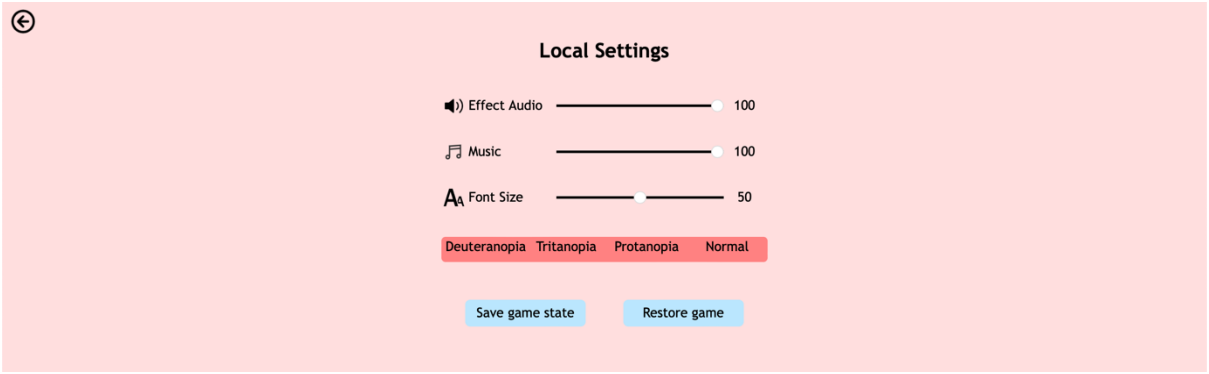
When play as spymaster and request a hint, a couple of related words will be highlighted to help you come up with a clue and target number.



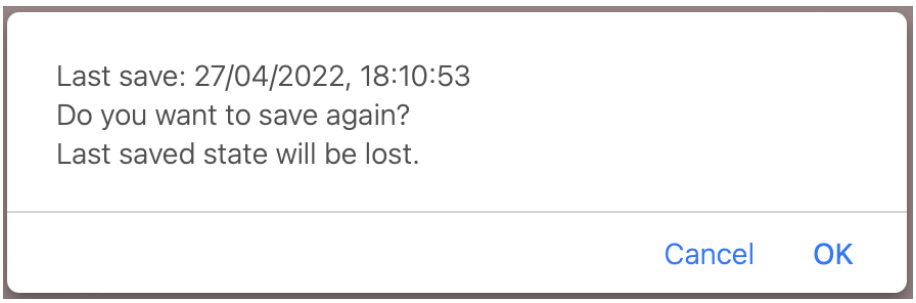
Saving and restoring the game state

Sometimes gamers don't have that much time to play the whole game and may quit for a while. Thus, we introduced a saving and restoring function to support that.

First, click the settings icon on the upper left corner, you will enter the Local Settings page shown below:

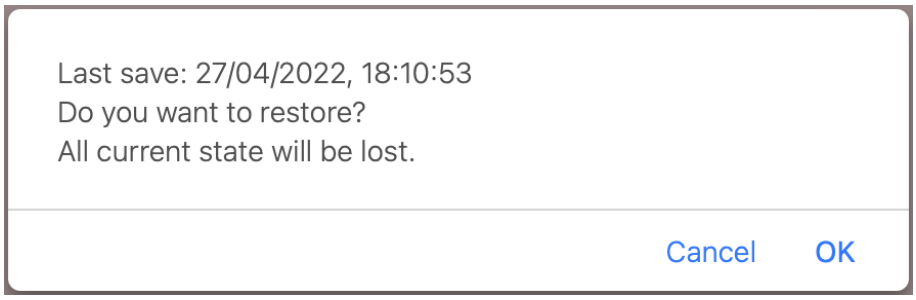


If you want to save the game, click "Save game state" button and a popup window will appear:



Note that you are only allowed to save game **in your turn**.

After this, if you want to pick up from where you left off, you can just go to the same settings page and click the “Restore game” button. A window will pop up as:

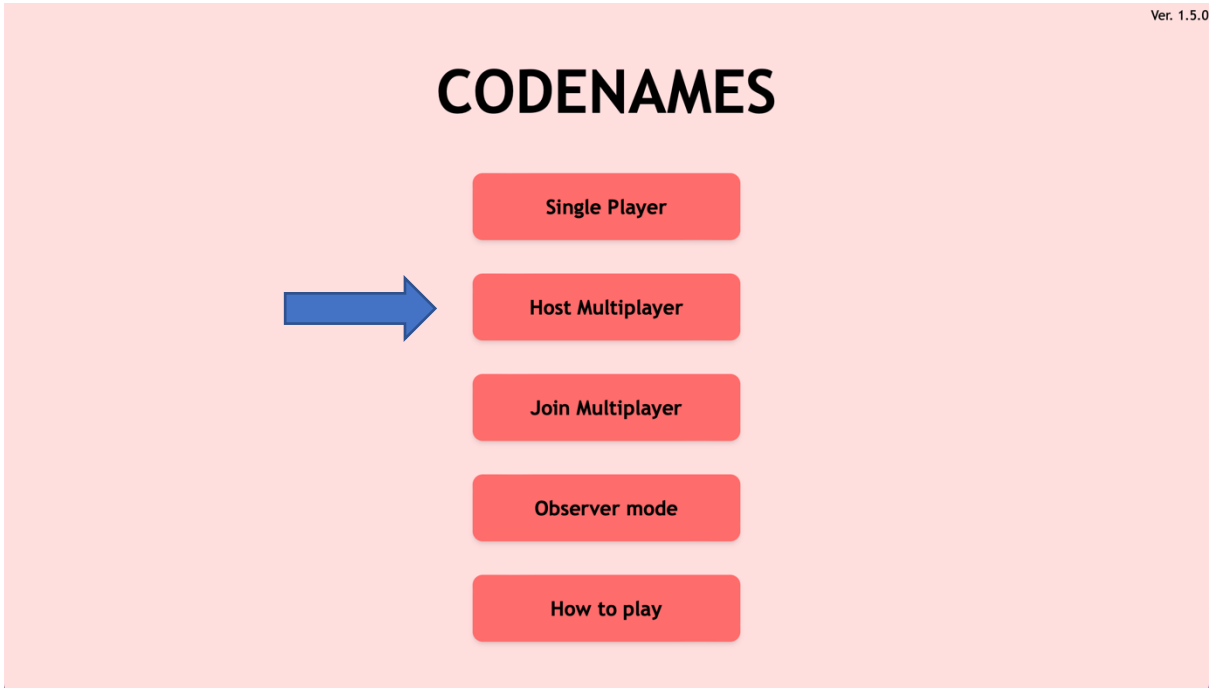


Note that you are only allowed to restore game **in your turn** or **before the game starts**.

Multiplayer

To play the multiplayer mode, you should make sure you and your friends’ devices are connected to internet. Click “Host Multiplayer” to host a game or “Join Multiplayer” to join a hosted room.

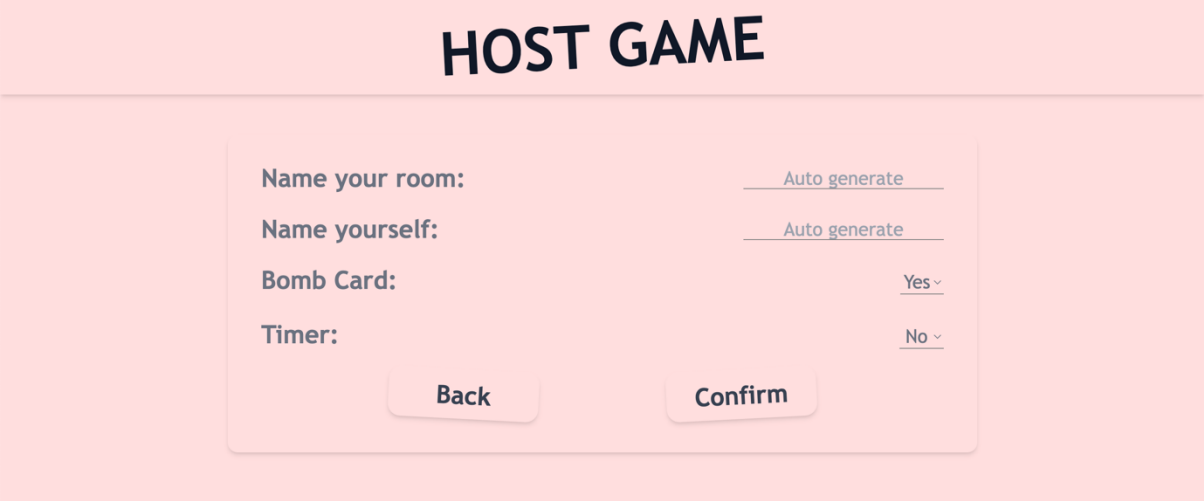
Hosting a game



After clicking “Host Multiplayer”, you will be directed to a page for host user as shown below.

The first line is to name your room. You can also let the program to generate a random room id by leaving it blank. Other gamers can join your room with your room id.

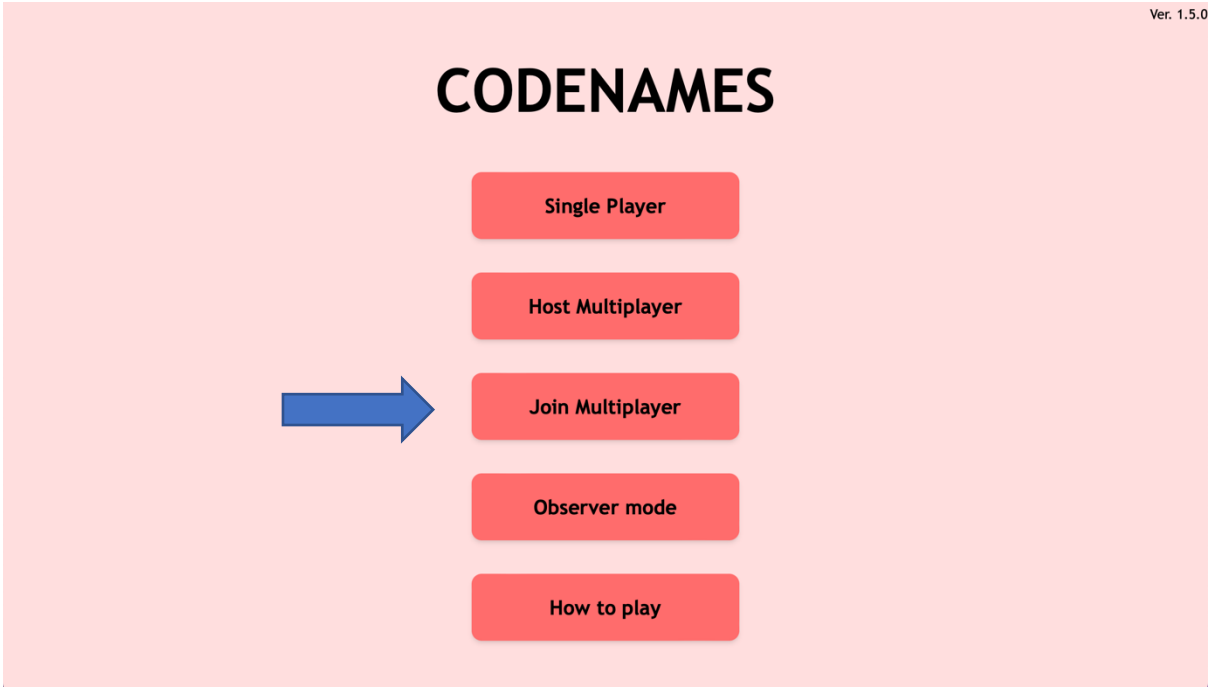
The rest of the options are well introduced in *Single Player* section, and when everything is done you can click “Confirm” to enter your room. Everything afterwards is almost the same as single player mode, waiting for everyone to choose their roles and then start the game.



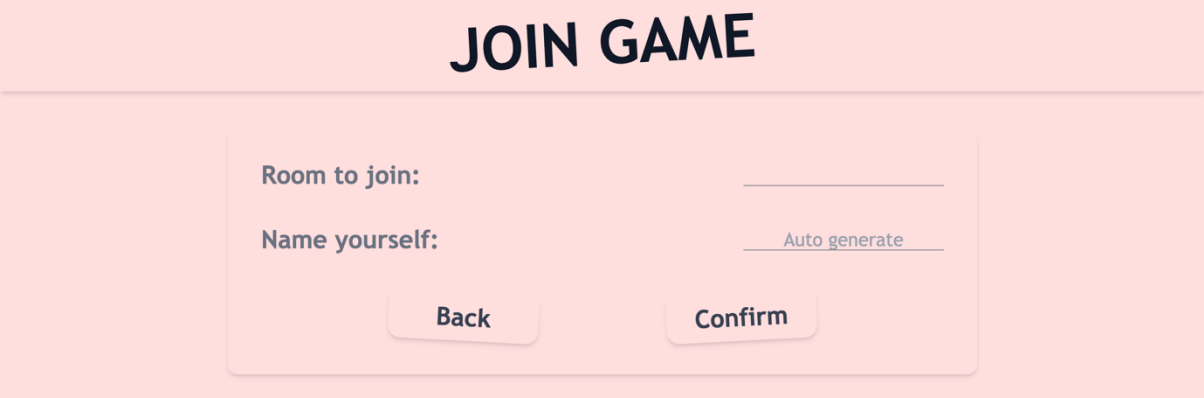
Note that the role vacancy will be filled by AI players, so the multiplayer mode could have player number ranging from 1 to 4.

When the game is finished, you can choose “Quit Room” to go back to main menu or “Restart” to go back to role selection page, and either choice will be synchronised to other players in the room.

Joining a game



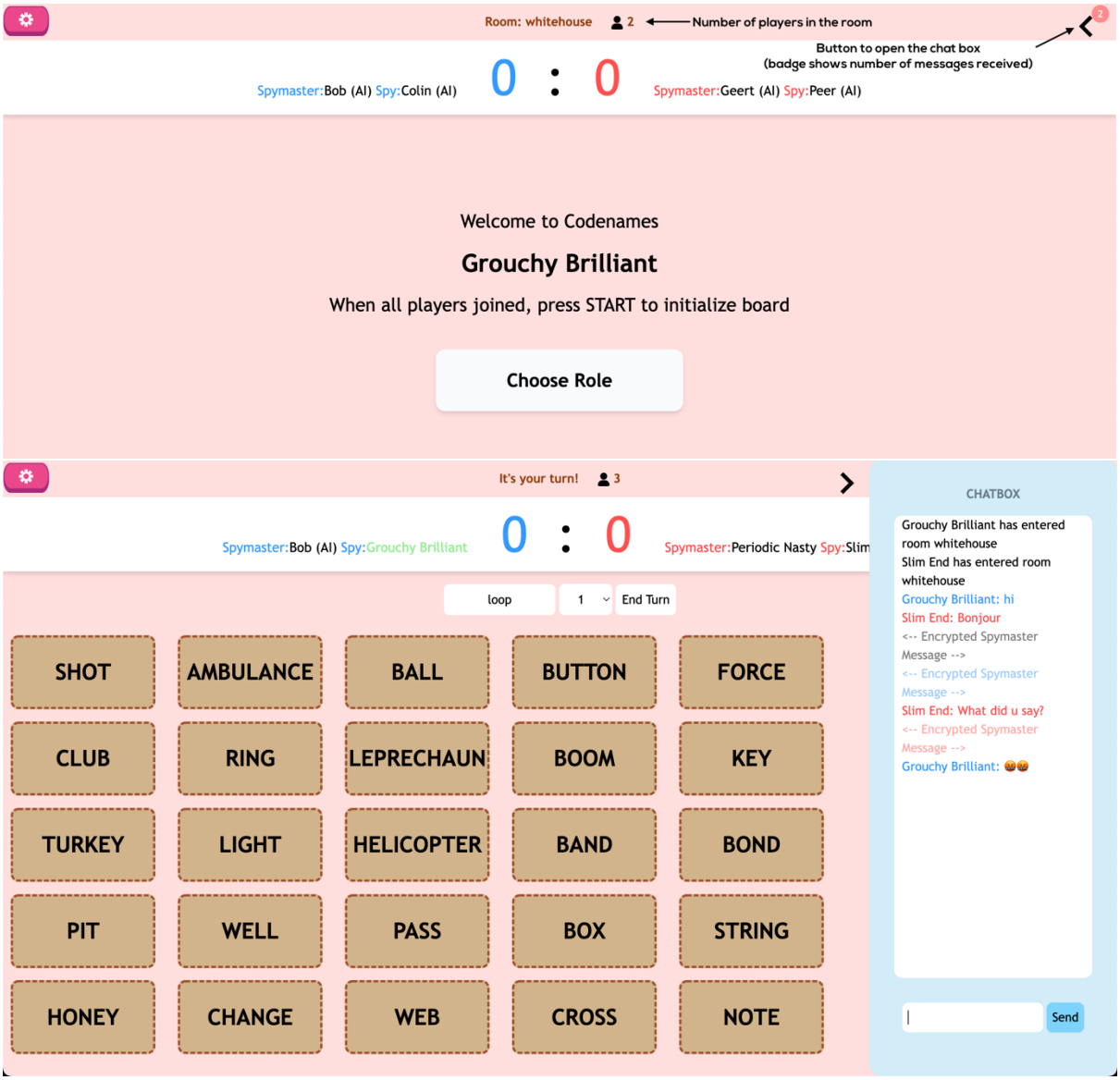
After clicking “Join Multiplayer”, you only need to enter the room id to join.



The following steps are almost the same as host user, except that you need to wait for the host to start or restart game. You can quit the room when game ends and other players in the room will be notified.

Using the chat

In multiplayer mode, our game allows users to chat with others in the room (don't forget to chat friendly ☺). The chat box is available once you enter the game lobby as follows:

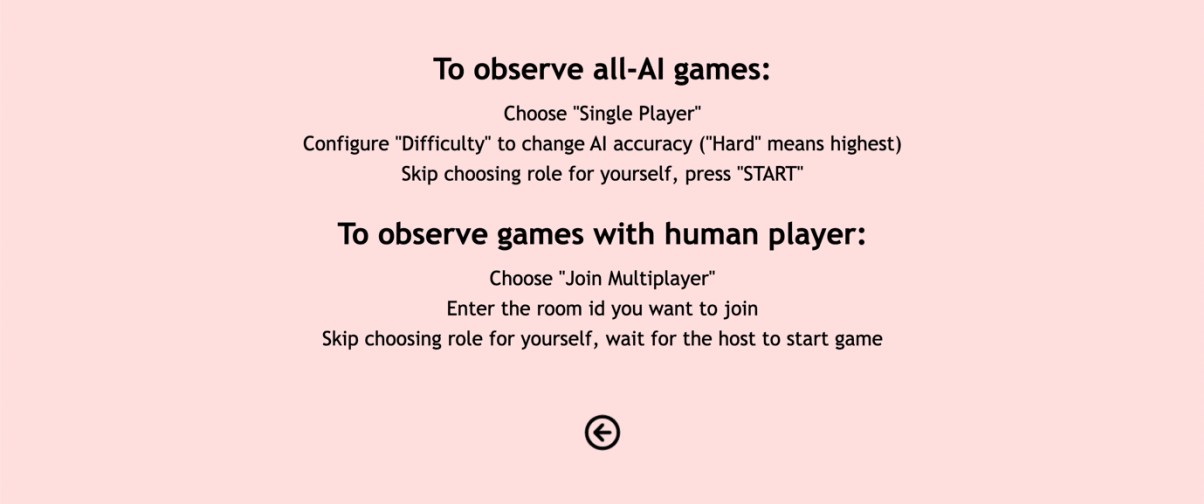


After opening the chat box, an example is shown above. The chat message contains username, and its colour represents user's team (text in black means the user has not chosen any role, known as "Guest" which will be introduced later).

As shown in the example, the spy player cannot see message from spymaster or guest due to the risk of clue leak, while spymasters and guests can see all messages. All players in the room can send chat and will be notified once someone joins or quits room.

Observer mode

Observer (also known as *Guest*) is the user without choosing any roles in the game room. You can be an observer in two ways:



Note that the number of guests to watch a multiplayer game is not limited (although max number of real players is 4). Guests could have the view of spymaster and chat to others but cannot make changes to the game board.

Rule of Codenames

The rule of game is also introduced by clicking "How to play" in the main menu. Here we simply copy that as follows:

The game is split between two teams, red and blue, over a 5x5 board of word cards and the objective is to be the first to get to 9 points. Each team has a spy and a spymaster.

Picking one of a team's card will add 1 point to that team. Picking a "neutral" card (card in brown) will not add any point. Picking a "bomb" card (card in black) or the timer runs out will end the game and your team loses.

For spymaster:

- You will see the team-colour of every card on the board.
- Enter a clue word that relates to several cards on the board. (Try to make it related to your team's cards rather than other cards)
- The clue cannot be the same as a word on the board.
- Enter the number of related cards and now it becomes your spy's turn.

For spy:

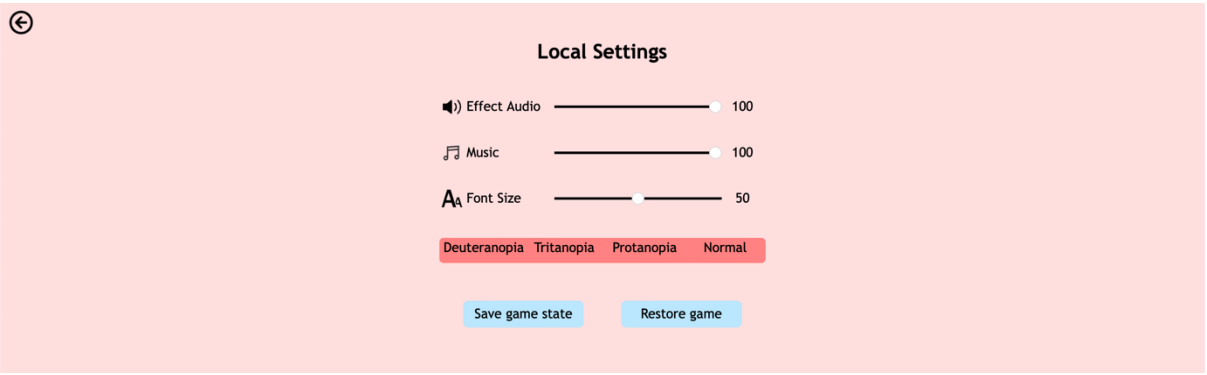
- You will get a board without team-colours. You will get the spymaster's word clue and number of related cards.
- Pick the right cards relating to the word clue. (e.g., If you get the clue "ocean-2" then the 2 cards "surf" and "water" might be correct.
- Your turn ends and the other team's turn begins when you either find a given number of correct cards or pick a card not belonging to your team.

The turn begins from blue spymaster, and loop until game ends.

Miscellaneous

Options

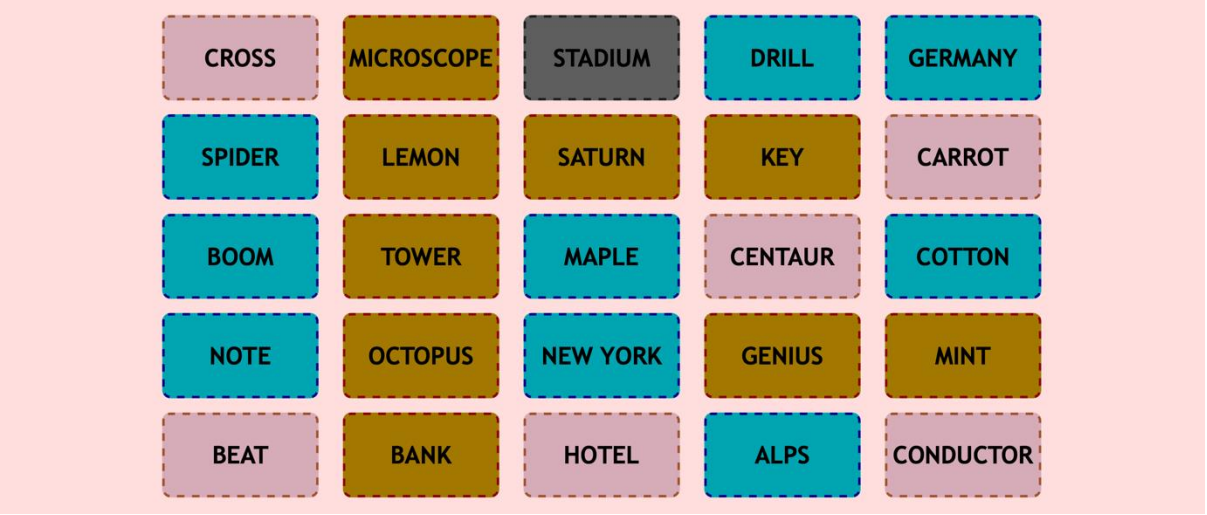
Our game provides a set of custom options as follows:



The first line is about to set the volume of the effect audio such as card-flipping and winning score, the second line is about to set the volume of the background music and the third line is about to change the font size of board cards.

Accessibility

Our product is also designed for users with visual impairment. As shown above in *Options* section, the fourth line provides a set of options for different colour blindness. When applied, the colour of board card will fit corresponding colour scheme. For example, below is the board for tritanopia:



In addition, the webpage layout is optimised for both mobile and PC, for any window size.

The game board on different devices is shown below:

These screenshots came from our demo video which is also available on http://www.codenames.uk/.

Troubleshooting

During the use of our product there could be some unexpected errors. Those most common are listed below and they appear as popup:

- ***Connection from server lost*** – please check whether the python server is running, if it is not then just start it, if it is then refreshing the webpage.
- ***Word not recognised by AI spy*** – please check the spelling of the word, make sure no spaces or other non-letter characters are in the input, or use another word.

- ***Illegal room id or username*** – please make sure the room id or username do not contain '?', '=', '&' or "(AI)".
- ***No saved state*** – please save the game before trying to restore the game. Since this is saved in the browser, clearing the browser cache can also delete the save data so be careful when doing that.
- ***Too few words to give a hint*** – please try request hint when less cards are picked.

Privacy Protection

As a browser game, we do not store any user data on the server, the data for restoring game state is stored locally on user's web browsers, and all chat messages are deleted from server once they are sent to all clients. The only data we save and evaluate is the room id so we can prevent multiple users from hosting rooms with the same name.

FAQs

- How can I get in touch with project developers?
    - Please go to our project website http://www.codenames.uk/ and find "Contact" section at bottom, click on any name of contributors.

- Is the project containing any open-source code from GitHub or somewhere else?
    - No, except for open-source external python libraries.
    - Additionally, we thank the authors of https://github.com/Pbatch/Codenames and https://github.com/thomasahle/codenames who have given us some ideas during our background research.