Liquid Democracy for Rating Systems

Hemanath Peddireddi
Department of Computer Science
University of Warwick

Supervised by Markus Brill

23 April 2025

WARWICK
THE UNIVERSITY OF WARWICK

Abstract

TODO

Keywords – liquid democracy, web application

Acknowledgements

TODO

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Decision-making is a central part of how groups operate; whether in political settings, organisations, or online communities. Traditionally, two primary models are used to make collective decisions: **direct democracy**, where every individual votes on each issue themselves, and **representative democracy**, where individuals elect others to vote on their behalf.

Both approaches have limitations. Direct democracy becomes impractical at scale, as it requires high levels of engagement from every participant. Representative systems, on the other hand, often concentrate decision-making power in a few individuals, and offer little flexibility once representatives are chosen.

Vodle is a web-based decision-making platform that aims to explore alternatives to these traditional models. It allows users to participate in polls by rating each option from 0 to 100, with the final result calculated using the MaxParC rating aggregation method. The platform is designed for open, consensus-oriented group decisions, and places strong emphasis on accessibility and user control.

However, the platform initially only supported direct participation – users could submit their own ratings but could not delegate their vote to others. This limitation made it less useful in contexts where users lacked time, interest, or expertise to vote in every poll. The motivation for this project is to address this gap by integrating support for liquid democracy into vodle.

1.2 Liquid Democracy

Liquid democracy is a hybrid approach that combines features of both direct and representative models. In a liquid democracy, users can choose to vote directly or delegate their vote to someone they trust. These delegations are transitive – if A delegates to B, and B to C, then C ultimately casts A's vote. Users may revoke or change their delegation at any time.

This system offers flexibility and scalability: engaged users can vote directly, while others can still influence outcomes through trusted delegates. It creates informal networks of influence, empowering individuals without forcing uniform participation.

Despite its benefits, liquid democracy introduces new technical challenges. Cycles in the delegation graph can trap votes, abstentions can unintentionally nullify entire chains, and certain individuals may accumulate disproportionate influence, becoming "super-voters." These risks require careful handling in any practical implementation, including the one proposed in this project.

1.3 Project Aims

This project aims to implement a flexible, robust liquid democracy system within vodle. In doing so, it extends the platform beyond simple direct voting and makes it more usable in realistic, large-scale decision-making settings.

The project addresses known theoretical weaknesses of liquid democracy, and introduces the following mechanisms:

- **Transitive delegation** – with consistent cycle prevention and real-time vote updates.
- **Ranked delegation** – where users specify fallback delegates.
- **Vote splitting** – allowing users to distribute parts of their vote to multiple people.
- **Per-option delegation** – letting users assign different delegates for different poll items.

All features are designed to work within vodle's server less, client-side architecture, and remain compatible with its underlying MaxParC voting model.

1.4 Structure of This Report – TODO when report is finished

The remainder of this report is structured as follows:

- Chapter 2
- Chapter 3
- Chapter 4
- Chapter 5
- Chapter 6
- Chapter 7

Chapter 2

Background Research

This chapter provides background context for the development of a liquid democracy system within vodle. It builds on the concepts introduced earlier, focusing on more detailed research into known limitations of liquid democracy and potential solutions proposed in academic literature. Additionally, the technical foundations and design philosophy of vodle as a platform are explored.

2.1 Liquid Democracy – TODO: FIX INTRO

Liquid democracy is a decision-making system that combines elements of both direct and representative democracy that offers a voter more flexibility than traditional voting models.

In direct democracy, every participant votes individually on each issue. This model offers the most individual input but can become impractical for large-scale decision-making due to the high level of participation required from each individual. As Ford (2002) states, direct democracy assumes that all individuals are both willing and able to engage meaningfully with every decision, which is often not the case in large groups due to the variance in both the interest and knowledge of voters. The cognitive demand of staying informed on all matters, combined with the time commitment necessary for constant participation, makes direct democracy unmanageable at scale.

In a representative democracy, citizens elect officials who make decisions on their behalf for the duration of a fixed term. While this model is scalable and practical for large populations, it introduces several limitations. Elected representatives often make decisions based on party lines, personal convictions, or external influences such as lobbying groups, which may not accurately reflect the preferences of their constituents (Blum and Zuber, 2016). In addition, because elections are infrequent, this system tends to be unresponsive to shifts in public opinion. Citizens are unable to easily adjust or retract their delegation, which limits their ability to influence decisions once representatives are in office (Blum and Zuber, 2016). As a result, participation is both indirect and inflexible, which can lead to disengagement and dissatisfaction among voters.

Liquid democracy addresses these limitations by allowing voters to: cast their votes directly, delegate them to someone that they trust, or abstain from voting entirely (Blum and Zuber, 2016). In comparison to a direct democracy, the bar for participation is lowered as voters no longer need to stay informed and engaged to pass a vote because they can trust a delegate to do it on their behalf. These delegations can also be updated or revoked at any time, giving users more control over how their vote is used in comparison to a traditional representational democracy where your representative can only be changed at certain points in time.

Despite its theoretical appeal, liquid democracy introduces new challenges. Transitive delegation can lead to unintended consequences such as delegation cycles, where a chain of delegations loops back on itself, or the emergence of super-voters, individuals who accumulate a disproportionate number of delegated votes. Additionally, if a voter delegates to someone who abstains from voting, their vote may be lost entirely. These issues raise concerns about fairness, transparency, and the integrity of representation within liquid democratic systems.

The remainder of this section explores these challenges in more detail and examines variations of liquid democracy designed to address them, including ranked delegation and vote splitting.

2.1.1 Issues with Liquid Democracy – TODO:INTRO

Throughout this section, several diagrams are used to illustrate how votes move through a liquid democracy system. To clarify the roles of different voters within these diagrams, the following symbols are used:

Symbol	**Role Description**
Circle	*Delegated voter* – has delegated their vote and does not cast one directly.
Square	*Casting voter* – casts their own vote and has not delegated.
Triangle	*Abstaining voter* – neither delegates nor casts their own vote.

Table 2.1: Diagram symbol legend used to represent different voter behaviours.

Delegation cycles

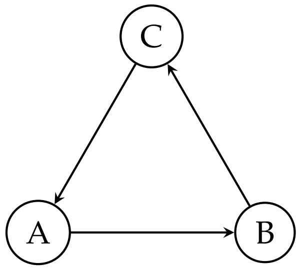

Figure 2.1: Delegation cycle: A delegates to B, B to C, and C back to A.

Delegation cycles occur when a vote is delegated in such a way that it ends up forming a loop (Brill et al., 2022), preventing the vote from reaching a final casting voter. For example, if Alice delegates her vote to Bob, Bob delegates to Charlie, and Charlie delegates back to Alice, the votes become trapped in a cycle (seen above) and can be treated as a loss of representation (Christoff and Grossi, 2017).

This issue is particularly problematic because it can nullify votes without the affected users ever realising. In systems where cycles are not explicitly detected and handled, these votes could be discarded silently, potentially changing the final outcome of the votes.

A simplistic method to prevent cycles is to check whether a delegation would create a cycle before allowing it. For example, if Alice tries to delegate her vote to Bob, the system checks whether Bob has already directly or indirectly delegated their vote to Alice. If so, the delegation is rejected. However, this approach can be cumbersome and may lead to a poor user experience, as users may not understand why their delegation was denied.

Delegation cycles are increasingly likely to emerge in dynamic voting systems, where delegations can be added, removed, or modified at any point in time. Delegations that initially did not form part of a cycle may later contribute to one as other voters add a new delegation or alter an existing one.

Abstentions

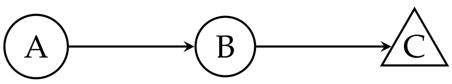

Figure 2.2: Delegation chain ending in abstention: A delegates to B, B to C. C abstains, causing the votes of A and B to be lost.

A voter abstains by neither casting a vote nor delegating it to another user (Brill et al., 2022). This includes both deliberate abstention, where a voter knowingly chooses not to participate, and passive abstention, where a voter may be unaware of an ongoing poll or are unable to engage with it.

Abstentions are especially impactful when they occur at the end of a larger delegation chain, as all votes passed along the chain to that voter are effectively lost (Brill et al., 2022). Additionally, the voters whose decisions were passed along the chain may also be unaware that their votes have been nullified, worsening the effect of the abstention.

Super-voters

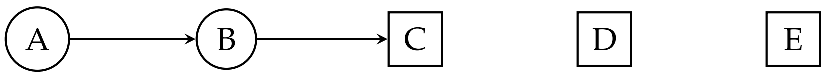

Figure 2.3: Super-voter: A delegates to B, B to C. No matter which vote D or E cast, C's vote will always determine the outcome as it has a weight of 3.

In liquid democracy, a super-voter is an individual who receives a large number of delegated votes, therefore gaining disproportionate influence over decisions (Kling et al., 2015). While this behaviour may reflect voters' genuine preferences, it can lead to a concentration of power that goes against the intended egalitarianism and democratic ideals of liquid democracy.

Even if a particular implementation of liquid democracy allows users to alter their delegation at any time, in practice, many voters may not actively monitor or even know how their vote is being used. This can allow a small number of super-voters to dominate outcomes, especially in systems that do nothing to prevent large delegation chains.

Real-world examples of this phenomenon have been documented. In the German Pirate Party's use of LiquidFeedback, certain users received so many delegations that their votes were like "decrees" (Sven Becker, 2012; Kling et al., 2015) even though they were not elected officials. Despite Kling et al. (2015) noting that super-voters generally voted in line with the majority and therefore did not drastically affect the outcome of the votes, the potential for individuals to single-handedly influence results still remains a concern.

This pattern is not limited to traditional online voting platforms. It can also be seen within decentralised autonomous organisations (DAOs) - blockchain-based entities where decisions are made collectively by token holders without central leadership. These organisations use token-based voting to decide on critical issues like protocol upgrades and funding allocations. Hall and Miyazaki (2024) studied 18 decentralised autonomous organisations (DAOs) and found that voting power was often concentrated in the hands of a few delegates. While most did not control a large share of all available tokens, low participation meant that their share of actual votes cast was disproportionately high. In several DAOs, the top five delegates accounted for over 50% of all votes cast, and in the DAO Gitcoin, this figure exceeded 90%.

2.1.2 Variations of Liquid Democracy

The challenges discussed in the previous section, such as delegation cycles, vote loss due to abstentions, and the emergence of super-voters, highlight inherent vulnerabilities in the standard liquid democracy model. To mitigate these issues, enhancements have been proposed that modify how delegations function. These include techniques that allow voters to specify multiple delegates or distribute their vote to multiple casting voters. Each approach introduces different trade-offs and requires algorithmic support to ensure sound and interpretable outcomes.

The following subsections present several such variations, along with the algorithms that can be used to implement them.

Ranked Delegation

Ranked delegation improves liquid democracy by allowing voters to list several trusted delegates in order of preference. Instead of choosing just one delegate, a voter can specify a ranked list so that if their top choice is unavailable (e.g. due to abstention or being a part of a delegation cycle) the system can use the next delegate specified.

Implementing ranked delegation requires a mechanism to decide among multiple possible delegation paths – a route that a vote can take through the delegation graph to reach a casting voter. This is done through a *delegation rule*, a function that, given a ranked delegation instance and a delegating voter, selects a unique path leading to a *casting voter* (Brill et al., 2022).

The following key properties help evaluate these delegation rules:

- **Guru Participation:** Ensures that a voter accepting delegated votes (a "guru") is never worse off by doing so. Receiving additional delegations should not decrease their influence over the final outcome (Kotsialou and Riley, 2020).
- **Confluence:** Guarantees that each delegating voter ends up with one clear and unambiguous delegation path. This property simplifies vote resolution and enhances transparency (Brill et al., 2022).
- **Copy Robustness:** Prevents strategic manipulation where a voter might mimic their delegate's vote outside the system to gain extra influence. A copy-robust rule makes sure that duplicating a vote externally does not yield more combined power than a proper delegation (Brill et al., 2022; Behrens and Swierczek, 2015).

The literature considers several delegation rules, each with distinct trade-offs:

Depth-First Delegation (DFD): Selects the path beginning with the highest-ranked delegate, even if the resulting chain is long. Although it prioritises individual trust preferences, DFD can violate guru participation (Kotsialou and Riley, 2020).

Breadth-First Delegation (BFD): Chooses the shortest available delegation path and uses rankings only to resolve ties. This approach usually produces direct, predictable chains and satisfies guru participation, although it might sometimes assign a vote to a lower-ranked delegate (Kotsialou and Riley, 2020; Brill et al., 2022).

MinSum: Balances path length and delegation quality by selecting the path with the lowest total sum of edge ranks. Due to this, MinSum avoids both unnecessarily long chains and poorly ranked delegations (Brill et al., 2022).

Diffusion: Constructs delegation paths in stages by assigning votes layer by layer based on the lowest available rank at each step. This method tends to avoid poor delegations but can sometimes produce unintuitive outcomes due to its tie-breaking procedure (Brill et al., 2022).

Leximax: Compares paths based on their worst-ranked edge. This ensures that especially low-ranked delegations are avoided early in the path while maintaining confluence (Brill et al., 2022).

BordaBranching: Takes a global view of the delegation graph by selecting a branching that minimises the total rank across all delegation edges. It satisfies both guru participation and copy robustness, though it is more computationally intensive (Brill et al., 2022).

In summary, ranked delegation enhances liquid democracy by reducing the risk of lost votes. The choice of delegation rule not only affects system efficiency but also influences fairness and robustness. While simpler methods such as DFD and BFD are easier to implement, advanced rules like MinSum, Leximax, and BordaBranching offer stronger guarantees and are better suited for practical deployment in platforms such as vodle.

For our implementation, MinSum will be chosen as the delegation rule because it offers a good trade-off between delegation quality, computational efficiency, and user interpretability. By selecting the path with the lowest total rank sum, MinSum prioritises higher ranked delegates while avoiding unnecessarily long or indirect delegation chains. This not only improves the quality of representation but also makes it clearer to users why a particular delegate was chosen as the path reflects their stated preferences in a straightforward way. Additionally, MinSum is more computationally efficient than alternatives like BordaBranching, making it a practical choice for deployment within the vodle platform.

Vote Splitting

Traditional delegation systems, which require voters to delegate their entire vote to a single individual, introduce significant risks such as vote loss through delegate abstentions, delegation cycles, and excessive concentration of voting power in the hands of a few super-voters. To mitigate these issues, vote splitting allows voters to distribute their voting power among multiple delegates. This approach provides greater flexibility and robustness while preserving voter intent more accurately.

Vote splitting offers several key advantages:

- **Increased resilience:** Distributing votes across multiple delegates reduces the impact of any single delegate abstaining or becoming unavailable, thus lowering the risk of vote loss.
- **Reduced concentration of power:** Allowing partial votes to different delegates decreases the likelihood of any single delegate becoming a super-voter.
- **Enhanced voter expression:** Voters can more precisely express their preferences and trust levels by allocating voting power proportionally to multiple individuals.

Several methodologies for implementing vote splitting have been explored in the literature, each with its strengths and weaknesses:

Equal Vote Distribution (Degrave, 2014)

Degrave's approach allows voters to distribute their votes evenly among multiple delegates. Voters select a group of delegates, and their vote is equally distributed amongst those that do not abstain. Although this system is intuitive and reduces the impact of abstentions, it lacks flexibility as voters cannot express differing trust levels towards each delegate. Additionally, a critical limitation is the inability for voters to allocate any portion of their vote to themselves, meaning voters are forced to either delegate their entire voting power or none of it, severely limiting personal control over a user's final vote.

Fractional Delegation (Bersetche, 2024)

Bersetche et al. introduce fractional delegation, allowing voters to explicitly assign different weights to each chosen delegate, including themselves. Delegates each receive a specified fraction of the voter's total voting power, reflecting the voter's nuanced trust and preference levels. This approach captures detailed voter preferences accurately and allows for greater personal agency compared to equal vote distribution. However, fractional delegation introduces additional complexity in managing and tracking these weighted delegations. Users must explicitly manage multiple numerical allocations, which may increase cognitive load and complicate user interfaces.

Trust Matrix Model

In the trust matrix model, voters define explicit trust values $\text{trust}_{i,j}$, representing how much voter i trusts voter j. Each voter also assigns their personal rating (self_i). The effective rating for each voter is computed iteratively using the equation:

$$\text{eff}_i = \text{trust}_{i,i} \cdot \text{self}_i + \sum_{j \neq i} \text{trust}_{i,j} \cdot \text{eff}_j$$

Here, each voter's trust values (including self-trust) must sum to at most 1. The iterative computation continues until the change in effective ratings between iterations falls below a predefined threshold ϵ. This approach offers the highest granularity and expressive power, allowing voters to precisely articulate nuanced trust relationships among multiple delegates. However, it comes with considerable computational complexity and potential convergence issues, especially in large networks with dense delegation relationships. Additionally, users may find it challenging to specify and manage such detailed trust matrices, negatively affecting usability.

Summary of Approaches

In summary, each vote-splitting method balances voter expressivity, computational complexity, user interface clarity, and resilience differently:

- **Equal Vote Distribution (Degrave)** excels in simplicity and ease of implementation, ensuring robustness through straightforward delegation. However, it significantly limits voter expression and prohibits voters from allocating votes to themselves.
- **Fractional Delegation (Bersetche)** provides greater flexibility, permitting detailed voter preference expression, including self-allocation of votes. This method increases both computational complexity and interface complexity.
- **Trust Matrix Iterative Model** offers the highest expressivity and detail in delegation relationships, capturing complex trust dynamics. However, this method entails substantial computational overhead and introduces complexity in terms of usability and understanding for voters.

Considering the priorities of the vodle platform–usability, intuitive interfaces, computational efficiency, and maintaining voter agency– . . .

2.2 Existing Implementations of Liquid Democracy

To understand how liquid democracy can be integrated into vodle, it is important to examine how similar systems have been implemented in real-world contexts. This section explores two implementations, LiquidFeedback and Google Votes, that offer valuable insights into the technical, social, and usability challenges associated with applying liquid democracy at scale.

2.2.1 LiquidFeedback

LiquidFeedback is one of the earliest and most influential real-world implementations of liquid democracy. Developed as an open-source platform, it was notably adopted by the German Pirate Party in 2010 to facilitate internal policy-making through online participation (Behrens et al., 2014). The platform allowed members to submit proposals, debate them in structured phases, and vote either directly or via transitive delegation.

In LiquidFeedback, users could choose different delegates for different topics, allowing them to assign their vote to someone they trusted on a specific issue. These choices remained in place until the user changed them, which meant that certain individuals could gradually accumulate more influence if others did not update their delegations. When multiple proposals were put forward, the system used a ranking-based voting method (such as the Schulze method) to decide which one should win. This approach compares each proposal against the others and selects the one that would win the most head-to-head match ups. Importantly, the system only accepted a proposal if it clearly beat the alternative of doing nothing, helping to avoid unnecessary or unpopular changes.

In practice, the Pirate Party's use of LiquidFeedback revealed several key dynamics relevant to this project. The platform was successful in enabling large-scale participation and crowd sourced policy formation, but it also demonstrated common risks of liquid democracy. Such as the existence of super-voters, as discussed previously.

Another practical issue was the complexity of the system. LiquidFeedback was difficult to understand for many users, especially those unfamiliar with concepts like transitive delegation or multi-stage voting which limited its accessibility and contributed to declining engagement over time (Kling et al., 2015).

For a platform like vodle, the experience of LiquidFeedback highlights several important design considerations. First, user interfaces must be intuitive enough to allow voters to participate without needing deep technical knowledge. Second, the user must be know the status of their delegation at a glance - improving the understanding of the platform. Finally, ensuring that votes lead to visible and actionable outcomes is critical for maintaining user engagement.

2.2.2 Google Votes

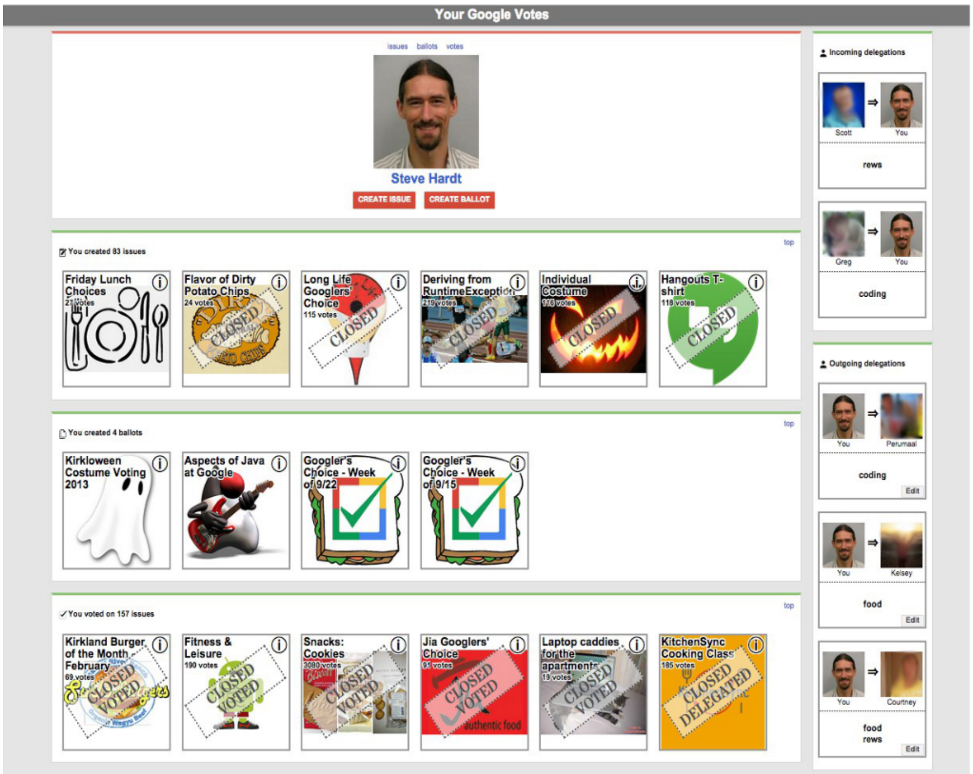

Figure 2.4: Screenshot taken from Hardt and Lopes (2015) showing the user interface of Google Votes.

Google Votes was an internal experiment at Google designed to explore the practical application of liquid democracy within a corporate environment. Built on top of the company's internal Google+ social network, it operated between 2012 and 2015 and allowed employees to participate in decision-making by either voting directly or delegating their vote to a colleague (Hardt and Lopes, 2015).

Delegations in Google Votes were category-specific, meaning that users could choose different delegates for different areas of interest, such as food, events, or technical infrastructure. These delegations were persistent but could be overridden at any time, giving users flexibility to either rely on trusted experts or vote independently as needed. The system supported transitive delegation and allowed users to reclaim control by casting their own vote, even after delegating.

The platform placed strong emphasis on usability and transparency. Delegation features were rolled out incrementally, with additional tools such as voting power estimates and delegation advertisements helping users understand their influence. One key design principle was what the authors called the "Golden Rule of Liquid Democracy": if a user delegates their vote, they should be able to see how it is being used. To accomplish this, users received notifications when their delegate voted, and all votes were visible to the relevant group. This encouraged accountability and gave voters confidence that their delegated votes were being used appropriately.

While Google Votes was never made publicly available, it served as a successful demonstration of liquid democracy in a structured, real-world setting. It showed that being able to delegate votes could improve engagement and decision-making within large organisations, especially when designed with attention to user experience. For vodle, the system provides a concrete example of how features like topic-specific delegation, transparency tools, and real-time voting feedback can make liquid democracy more practical and accessible.

2.3 Agent Based Modelling

Agent-based modelling (ABM) is a computational approach used to simulate the actions and interactions of autonomous agents in order to assess their effects on a system as a whole. It is particularly suited for exploring complex, dynamic systems where behaviour emerges from local interactions between individual entities (agents) rather than being dictated by central control. ABM has been widely applied in domains such as economics, sociology, and ecology to study decentralised systems, market dynamics, and collective behaviours (Bonabeau, 2002).

The need to explore ABM arises due to the project's goal of introducing a vote-splitting mechanism that hasn't been explored before into vodle. Traditional analysis alone may not effectively capture the dynamic interactions or unintended consequences that can emerge from this novel feature. Through ABM, it is possible to simulate realistic voting scenarios, track delegation chains, identify potential power imbalances, and anticipate challenges. These simulations can reveal performance insights and inform design decisions before implementing the mechanisms within the live platform.

Several widely used ABM frameworks exist, each with their own strengths and drawbacks relevant to this project:

- **NetLogo** (Tisue and Wilensky, 2004) is a highly accessible and widely adopted modelling platform known for its user-friendly graphical interface and ease of learning. It offers rapid prototyping capabilities and excellent visualisation features, allowing clear communication of results. However, very complicated models are not compatible with it.

- **Repast** (Collier, 2003) provides a powerful and versatile suite of tools for building large-scale, computationally intensive simulations. It supports distributed computing, which is beneficial for extensive delegation networks with potentially thousands of agents. However, Repast has a steep learning curve, which could hinder its compatibility with this heavily time restricted project.

- **Mesa** (Kazil et al., 2020) is an open-source framework written in Python and specifically designed for agent-based modelling. Its advantage lies in its integration with Python's ecosystem of data science libraries. Simulations built with Mesa can easily make use of tools such as NumPy and pandas for efficient data processing, and Matplotlib or Seaborn for visualising model outputs. This compatibility allows for rapid analysis and iteration, while also significantly lowering the learning curve for developers already familiar with Python. Mesa offers a practical balance between usability and computational flexibility, making it well-suited for customisable and moderately large simulations.

- **Agents.jl** (Vahdati, 2019) is a high-performance agent-based modelling framework written in Julia. Due to Julia's speed and efficiency, it is suitable for large-scale and computationally demanding simulations. The framework is designed to be user-friendly, with a syntax that is approachable for those familiar with scientific computing. However, the Julia ecosystem is less mature compared to Python's, which may limit the availability of additional libraries and resources.

Given the time constraints of this project, Mesa offers a practical and efficient solution. Its Python-based interface and straightforward setup allow for rapid development without the overhead of learning a new framework. This ease of use enables more time to be spent designing meaningful experiments and analysing results, rather than configuring tooling.

2.4 Vodle

Vodle is a web-based platform for participatory group decision-making. Users participate in polls that allow them to rate a set of options using sliders. When the poll ends, these ratings are aggregated and the MaxParC rating system is used to determine the final result of the poll.

2.4.1 MaxParC - TODO: rework the summary

MaxParC (Maximum Partial Consensus) is the core rating system used in vodle to aggregate user preferences and determine poll outcomes. Introduced by Heitzig et al. (2024), MaxParC was designed to address common limitations of traditional voting systems, particularly the tendency for majority rule to overlook minority preferences. Its goal is to balance fairness, consensus, and efficiency in collective decision-making.

In MaxParC, each user rates an option on a scale from 0 to 100. This rating reflects the user's willingness to approve that option based on how many other users also support it. Specifically, a rating of x means that the voter will approve the option if fewer than x% of participants disapprove. A rating of 0 means the option is never approved, while 100 means it is always approved regardless of others' opinions. This structure transforms a simple rating into a conditional approval, allowing for a more nuanced expression of preferences with the potential for compromise.

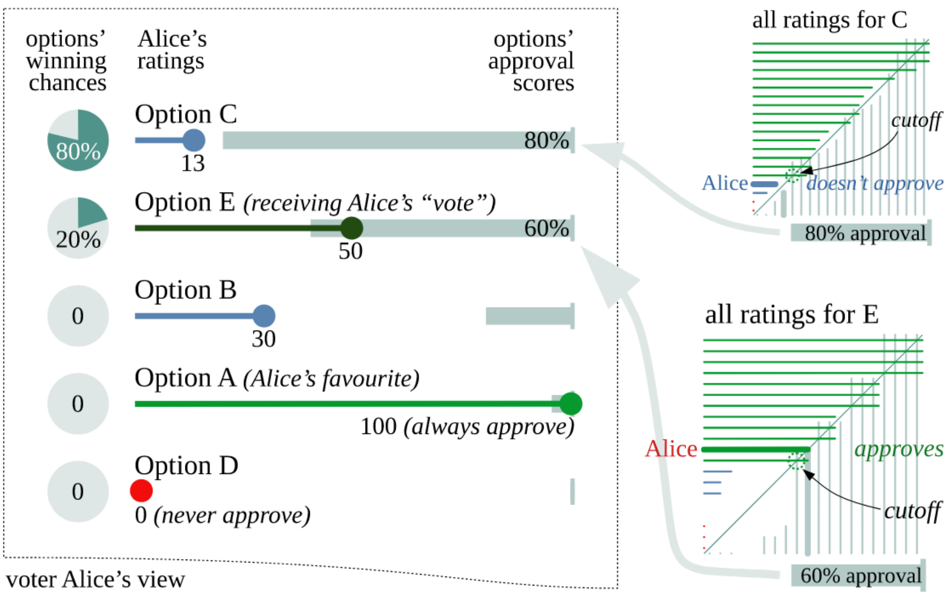

Figure 2.5: Visual representation of MaxParC from the perspective of a voter (Alice). Ratings represent conditional approval thresholds. An option is counted as approved by Alice if the approval bar (light grey) overlaps with her rating needle. Graphic from Heitzig et al. (2024).

TODO: rework this section to be more concise and clear or remove Understanding how MaxParC processes ratings is essential for this project, as the proposed vote splitting mechanism must operate within its conditional approval framework. When a user splits their vote among multiple delegates, the system must ensure that their own rating continues to contribute appropriately. Specifically, if a user delegates x% of their rating to others, their final rating must not fall below $(100 - x)$% of their original input. This constraint guarantees that the user's approval remains proportionally represented, even when part of their voting power is passed on to others.

Integrating liquid democracy into vodle therefore requires careful design to align with MaxParC's logic, ensuring both technical compatibility and conceptual consistency.

2.4.2 Technologies Used

Understanding vodle's technology stack is crucial to successfully integrate liquid democracy features into the existing platform. Since the project involves adding complex delegation and voting logic, it's important to appreciate the constraints and benefits of the technologies currently used in vodle, as they directly influence the design and implementation choices.

Angular

Vodle is built with Angular (Angular Team, 2024), a TypeScript based frontend framework created by Google. Angular's modularity and structured component system provides a strong foundation for incremental development, essential when introducing new features such as ranked delegation that build upon existing components. Its clear separation of concerns helps maintain readable and maintainable code, simplifying debugging and future enhancements. This is particularly beneficial as the delegation logic is expected to grow in complexity and build upon existing components as the project progresses.

Ionic Framework

The Ionic (Ionic Team, 2024) framework complements Angular by enabling the creation of responsive, mobile-compatible applications from a single codebase. Given vodle's goal of broad user participation, Ionic ensures that its functionalities remain consistent and accessible across both desktop and mobile devices. For this project, new delegation features must be designed to work seamlessly within the existing Ionic framework, ensuring that they are visually appealing and user-friendly on all platforms, including mobile devices.

CouchDB

CouchDB (Apache CouchDB Project, 2024) is vodle's primary data storage method and communicates directly with the client through HTTP requests, meaning there is no dedicated backend. This architecture places significant computational responsibilities on the client-side Angular application, including the handling of delegation chains, cycle detection, and the computation of final vote outcomes. Furthermore, since CouchDB stores data exclusively as JSON-formatted strings, complex delegation structures and voting relationships must be serialised and de-serialised on the client side.

The lack of server-side computation means the delegation algorithms must be designed with client-side efficiency in mind, ensuring performance remains acceptable even as delegation complexity increases. Thus, the choice of algorithms for liquid democracy features, such as those to resolve conflicting delegation paths, is directly influenced by CouchDB's architectural constraints.

These technological considerations – which will be covered in more detail in Section 4.1 – shape the practical implementation of liquid democracy within vodle, highlighting the need for efficient client-side processing, careful data management, and cross-platform consistency.

2.4.3 Partially Implemented Delegation in Vodle

The current version of vodle includes a partially implemented delegation feature that allows users to delegate their vote to another participant in a given poll. Whilst this implementation provides the basic structure required for delegation, it lacks consistency in handling more complex delegation scenarios such as cycles.

include delegation workflow + forward ref to implementation chapter

2.4.4 Design Philosophy

2.5 Summary

The background research presented in this chapter has provided the necessary foundation for designing and implementing advanced delegation features within vodle. Initially, the research highlighted critical limitations in traditional liquid democracy systems, such as the formation of delegation cycles, the risks associated with abstentions, and the disproportionate influence of super-voters. These insights showed the need for implementing delegation mechanisms that are capable of addressing these challenges effectively.

Research into these mechanisms revealed several promising methods. Ranked delegation was found to be an effective approach for reducing the risk of lost votes, with the MinSum delegation rule being particularly suitable due to its clear balance of efficiency, interpretability, and fairness. Vote splitting was identified as a valuable strategy to allow voters greater flexibility by distributing their influence among multiple trusted delegates. Additionally, the concept of delegating different options to distinct delegates was supported by practical experiences from Google Votes, where topic specific delegations improved user engagement and representation accuracy.

To ensure a thorough evaluation of these delegation mechanisms before full-scale integration, agent-based modelling was selected as a suitable method, with the Mesa framework identified as the most practical choice due to its ease of use, Python compatibility, and flexibility for rapid experimentation.

The technological constraints of vodle itself, especially the reliance on client-side processing due to the CouchDB architecture, demonstrated the need for efficient, lightweight implementation strategies.

These insights collectively define the project's objectives, which are formalised in the following chapter. The objectives are designed explicitly to address the limitations uncovered in the research, ensuring the integration of liquid democracy into vodle is practical, user-friendly, and aligned with established best practices.

Chapter 3

Project Objectives

This chapter outlines the milestones of the project. These objectives were derived from the background research and are designed to address the technical and theoretical challenges identified with traditional delegation systems.

To manage the project effectively, the objectives are divided into two categories: **core objectives**, which form the backbone of the implementation and are essential to meeting the project's main goals, and **extension objectives**, which provide additional insight or value.

Later in the chapter, each objective is broken down into specific functional and non-functional requirements. This structure helps to clarify expectations, guide implementation, and provide clear criteria for evaluating whether each objective has been met.

Core Objectives:

1. **Implement a Core Delegation Model into Vodle:** Build upon the existing, partially implemented delegation code within the vodle platform to create a fully functional system, including resolving key challenges such as cyclic delegations.

2. **Implement Ranked Delegation into Vodle:** Add a backup delegation mechanism to vodle, allowing users to specify up to 3 delegates.

3. **Implement a Vote Splitting Delegation Mechanism into Vodle:** Add functionality to vodle to delegate fractions of their rating to different delegates. Use the *will come back to when research written up* system to calculate final ratings.

4. **Implement the Ability to Delegate Individual Options to Different Users**: Allow users to delegate the ratings of specific options to different delegates.

Extension Objective:

1. **Simulate Delegation Mechanisms:** Perform agent-based modelling to analyse the effectiveness of various delegation systems, including those outlined in Objectives 1, 2 and 3.

3.1 Project Requirements

The project objectives represent high-level goals that must be translated into specific, actionable requirements. This process is essential for clarifying the scope of the work, ensuring comprehensive coverage of each objective, and establishing a structured foundation for both implementation and evaluation.

To support this, requirements are organised into two categories: functional (F) and non-functional (NF). Functional requirements describe the core behaviours and features the system must support, while non-functional requirements define performance, usability, and other quality-related constraints (Sommerville, 2016). Distinguishing between these categories helps ensure that both the system's functionality and overall user experience are properly addressed.

Each requirement is formulated to be measurable and testable. This allows for objective evaluation during development, facilitates verification against the project goals, and helps identify areas for improvement as the system evolves.

3.1.1 Implement a Core Delegation Model into Vodle

Functional Requirements

- **FR1:** The system shall correctly handle the delegation process from invitation to acceptance.
  - **FR1.1:** The system shall allow users to invite others to act as their delegate.
  - **FR1.2:** The system shall allow invited users to accept delegation requests.
  - **FR1.3:** The system shall prevent users from accepting their own delegation invitations.
  - **FR1.4:** The system shall detect and prevent the formation of delegation cycles.
- **FR2:** The system shall provide users with a clear view of their current delegation, including the ability to revoke it at any time.
- **FR3:** The system shall resolve delegations transitively, such that if User A delegates to B and B delegates to C, User C is the final casting voter for A's vote.

- **FR4:** The system shall allow users to override a delegate's decision for specific poll options by submitting a direct vote.
- **FR5:** The user interface for delegation shall be intuitive and accessible, with clear instructions and minimal friction to perform delegation actions.

Non-Functional Requirements

- **NFR1:** All delegation-related data must be stored in a JSON-encoded format, ensuring compatibility with the existing vodle CouchDB database.
- **NFR2:** Any changes to the database schema must be backward compatible, ensuring that existing data is not lost or corrupted during the upgrade process.
- **NFR3:** Any additional data stored in the database must be encrypted, using the same encryption method as the existing data, to ensure user privacy and consistency with the existing system.
- **NFR3:** The system shall preserve user privacy by ensuring that individual voting preferences and delegation choices are not visible to other users. The only information visible to a delegated user shall be the final vote cast on their behalf.

3.1.2 Implement Ranked Delegation into Vodle

Functional Requirements

- **FR1:** The system shall allow users to specify a ranked list of up to 3 delegates for each poll, with the ranking applying to all options within that poll.
- **FR2:** The system shall apply the MinSum delegation rule to resolve each voter's delegation path based on their ranked list of delegates.
- **FR3:** The system shall allow users to override the ranked delegation by submitting a direct vote for specific poll options.
- **FR4:** The system shall provide users with a clear view of their ranked delegation choices, including the ability to alter their rankings or revoke them at any time.

Non-Functional Requirements

- **NFR1:** The system shall ensure that the ranked delegation process does not introduce significant latency in the voting process, maintaining a response time of less than 2 seconds for delegation-related actions when the number of delegates is less than 100.

- **NFR2:** The user interface for ranked delegation shall be intuitive and accessible, with clear instructions and minimal friction to perform delegation actions.
- **NFR3:** The system shall ensure that any data related to ranked delegation is stored in a JSON-encoded format, ensuring compatibility with the existing vodle CouchDB database.

3.1.3 Implement a Vote Splitting Delegation Mechanism into Vodle

Functional Requirements

- **FR1:** The system shall allow users to delegate their vote to multiple delegates simultaneously for a single poll.
- **FR2:** The system shall allow users to assign a weight to each delegate such that the total weight does not exceed 0.99.
- **FR3:** The system shall use the algorithm described in subsection **??** to calculate the final rating for each option based on the weights assigned to each delegate.
- **FR5:** The system shall provide users with a visual interface to edit the weights assigned to each delegate, with either sliders or numeric inputs for easy adjustment.

Non-Functional Requirements

- **NFR1:** The system shall ensure that vote-splitting calculations are performed entirely on the client side to comply with vodle's CouchDB architecture.
- **NFR2:** All related data must be serialised as JSON strings to ensure compatibility with the CouchDB backend.
- **NFR3:** The user interface for vote splitting shall be intuitive and allow users to adjust weights easily, using sliders or numeric inputs.

3.1.4 Implement the Ability to Delegate Individual Options to Different Users

Functional Requirements

- **FR1:** The system shall allow users to assign different delegates for each individual option or subset of options within a poll.

- **FR2:** The system shall ensure that each delegated option is resolved independently, using the appropriate delegate's vote for that option.
- **FR3:** The system shall allow users to override a delegate's vote for a specific option by submitting their own rating.
- **FR4:** The system shall provide a user interface for viewing or revoking each individual delegation.

Non-Functional Requirements

- **NFR1:** The delegation interface must be intuitive and clearly indicate which delegate is assigned to each option, ensuring ease of use.
- **NFR2:** The delegation data must be serialised in a format compatible with CouchDB (e.g., JSON-encoded) to maintain compatibility with vodle's storage system.

3.1.5 Simulate Delegation Mechanisms

Functional Requirements

- **FR1:** The simulation system shall model individual agents representing voters, each capable of voting, abstaining, or delegating their vote according to a selected delegation rule.
- **FR2:** The system shall support multiple delegation mechanisms, including standard transitive delegation, ranked delegation (with the MinSum delegation rule), and vote splitting.
- **FR3:** The system shall allow configuration of simulation parameters such as number of agents, delegation probabilities and abstention rates.
- **FR4:** The system shall track and record key metrics such as vote concentration, number of super-voters, average chain length, vote loss due to abstentions or cycles, and decision quality.
- **FR5:** The system shall output simulation results in a structured format (e.g. CSV or JSON) for further analysis.

Non-Functional Requirements

- **NFR1:** The simulation framework must be lightweight and easy to extend, enabling rapid experimentation with new delegation rules or metrics.
- **NFR2:** The system shall be developed using Mesa to take advantage of existing data science libraries such as NumPy, Pandas, and Matplotlib for analysis and visualisation.
- **NFR4:** The simulation design shall support reproducibility by enabling fixed random seeds and storing configuration settings alongside output data.

Chapter 4

Design and Implementation

This chapter describes the design and implementation of the delegation mechanisms integrated into vodle, detailing both the technical approach and practical decisions made throughout development. It begins by outlining vodle's existing system architecture, clarifying how this influenced the integration of new delegation features.

Each subsequent section aligns directly with one of the project objectives defined previously, explaining the rationale behind key design choices, algorithms, and interface elements. Emphasis is placed on the critical design trade-offs and challenges encountered, highlighting how constraints such as the serverless architecture and client-side computation informed implementation decisions.

4.1 System Architecture Overview

Vodle is built as a serverless web application that emphasises accessibility, client-side performance, and ease of deployment. Its architecture comprises two components:

1. **Frontend:** Implemented using Angular and the Ionic framework, the frontend provides a responsive and modular interface that works across both desktop and mobile devices. The use of Angular facilitates the creation of component-based user interfaces, essential for introducing interactive features such as the ranked delegation UI and vote splitting sliders.
2. **Backend:** Vodle uses CouchDB as its database. There is no custom backend logic or middleware; instead, the frontend application communicates directly with CouchDB over HTTP.

Implications of This Architecture

Vodle's serverless architecture has several implications for the design and implementation of the delegation mechanisms, especially due to the absence of a traditional data processing backend. The following points summarise the key considerations:

- All vote delegation logic, including transitive resolution, cycle detection, and vote splitting calculations, must be executed in the browser. This places constraints on performance and requires careful optimisation of algorithms used.
- CouchDB's document-based storage model means that all data must be serialised and deserialised in JSON format. This affects how data structures are designed and manipulated, as well as how they are stored and retrieved from the database.

4.1.1 CouchDB Storage and Write Constraints

Vodle store all data in CouchDB in two types of databases: the _users database and poll databases. The _users database is a standard CouchDB database that stores user documents, while the poll databases are created dynamically for each poll and contain all relevant data for that poll.

1. **_users Database.**
   - Each document ID is `org.couchdb.user:<username>`.
   - Only the account owner or an administrator may modify or delete their user document.
2. **Poll Databases.**
   - Databases are named `poll-<POLLID>`.
   - Stores:
     - The immutable poll definition (`poll.json`).
     - One vote document per voter (`vote-<user>`).

Document-Level Security CouchDB enforces security at the level of entire documents. This means that access control decisions are made based on the identity of the user attempting to write a document and the document's ID – there is no support for restricting access to individual fields within a document. Additionally, CouchDB does not support merging concurrent changes; updates must replace the entire document in a single write operation. As a result, all write operations are either fully accepted or fully rejected by the database's `validate_doc_update` function. This strict model simplifies validation logic but introduces important constraints in the context of implemented liquid democracy, which are discussed in the remainder of this section.

Poll-DB Validation

When a client writes to a poll database, the `validate_doc_update` enforces:

1. **User Documents:** IDs are prefixed with `vodle.user.`, and only the owner may modify them:

```
if (!id.startsWith(`~{userCtx.name}Â§`)) {
  throw({ forbidden: 'Only the document owner may modify this user doc.' });
}
```

Figure 4.1: Code to prevent a user from modifying another user's document.

2. **Immutable Poll Artefacts:** Documents like `poll.json` or results cannot be updated once created:

```
if (oldDoc && isPollArtifact(id)) {
  throw({ forbidden: 'Poll documents are immutable once created.' });
}
```

Figure 4.2: Code to prevent modification of poll artefacts.

_users-DB Validation

In the _users database, a similar validation function is used to prevent users from modifying other users' documents, including the `vodle` service account.

```
if (id !== `org.couchdb.user:{userCtx.name}`) {
  throw({ forbidden: 'Users may only modify their own user document.' });
}
if (userCtx.name === 'vodle' || isPollService(userCtx.name)) {
  throw({ forbidden: 'Service accounts are immutable.' });
}
```

Figure 4.3: Code to prevent a user from modifying another user's document.

4.1.2 Summary of Storage and Validation Constraints

The architecture of vodle, particularly its reliance on CouchDB and the absence of a custom backend, imposes important constraints on how delegation features are designed and implemented.

- **User autonomy is strictly enforced.** Each user can only modify documents that are explicitly associated with their own identity. This guarantees that vote and delegation data cannot be tampered with by other clients but also eliminates the possibility of directly setting or managing another user's vote.
- **Shared state must be decomposed.** The current database design does not support the modification of a single document by multiple users. As a result, features that require a global view – such as a delegation graph – require a rework of the database schema.
- **Validation logic is structural, not contextual.** Since CouchDB validation functions can only inspect the document being written, they cannot reason about relationships across documents. This prohibits logic such as resolving delegations server-side, enforcing uniqueness of votes, or validating delegation cycles at the point of write.
- **Client-side logic carries the burden.** Due to the point above, all logic for delegation resolution, cycle checking, and vote splitting must be implemented in the client. This requires careful design to ensure that the frontend can handle complex delegation scenarios without overwhelming the user or causing performance issues.

Together, these constraints shape some of the design and implementation choices of the delegation features in vodle, which will be discussed in detail in the following sections.

4.2 Implement a Core Delegation Model into vodle

This section describes the design and implementation of the core delegation mechanism in vodle, which allows users to delegate their votes to others. It covers the issues with the original implementation and the challenges solved in the redesign.

4.2.1 Delegation Interaction Flow

Figure 4.4 depicts the life-cycle of a delegation. The diagram is intentionally simple–only the four message exchanges required to set up or tear down a delegation are shown–so the reader can map each arrow to a concrete API call or UI control.

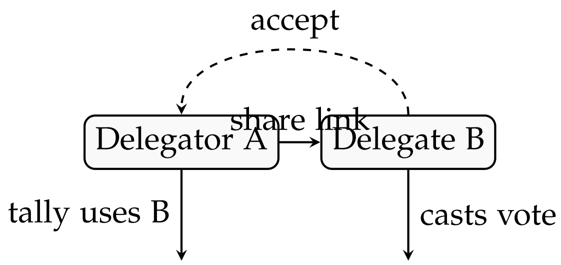

Figure 4.4: Minimal message sequence for creating a delegation. Only four interactions are required: (1) A shares a link, (2) B accepts, (3) B may cast or delegate further, (4) A's vote resolves through B.

At runtime the app must therefore support:

1. Generating and copying a poll-scoped, user-scoped invitation link.
2. Polling CouchDB for acceptance (or rejection) of that link.
3. A client-side resolution routine that follows the chain of `delegateFor` relationships until a casting voter is found.
4. Revocation that is immediate yet conflict-free if multiple clients are open.

4.2.2 Issues Prior to Redesign - unnecessary?

The original implementation of liquid democracy in vodle functions as a proof-of-concept, but it has several limitations that hinder its usability and reliability. The following sections outline some of the key issues that were identified whilst planning the redesign.

Inconsistency in data between clients delegation graph not synched – leads to cycle checking issue . . .

Out-of-sync state Each browser session cached its own copy of the delegation graph; no push notifications existed to bring other clients up-to-date.

Inconsistent cycle checking Because state diverged, the local "no-loop" tests sometimes produced different answers on different machines.

UI drift Controls such as "delegate none/some/all" and the "your vote is used for n others" badge used stale data and became misleading.

4.2.3 Cycle Checking

Robust cycle detection is a core requirement for any system implementing transitive delegation. In liquid democracy, cycles render votes unresolved and potentially lost, directly undermining the integrity of the outcome. Given vodle's dynamic and client-driven architecture, it was crucial to implement an efficient, client-side mechanism that could detect and prevent cycles in real time, without requiring server-side intervention or excessive computation.

The original implementation of cycle prevention in vodle was incomplete and inconsistently enforced. Because delegation data was locally cached and not always synchronised across clients, users could unknowingly create cycles that would only be caught (or missed) depending on the state of their browser session. This led to erratic user experiences, where a delegation might succeed on one device but be rejected on another.

The redesigned cycle checking approach addresses this inconsistency through the introduction of a shared data structure, stored in CouchDB and updated collaboratively by all clients. This ensures consistent validation of new delegations and enables immediate feedback to users attempting to create an invalid delegation path. The remainder of this section details the algorithmic choices, data structures, and UI modifications used to implement reliable cycle detection in vodle.

Algorithm Design

The current delegations in a system can be represented as a directed graph where each user is represented as a node and each delegation is represented as a directed edge (u, v), where u is the delegator and v is the delegate. The goal of the cycle-checking algorithm is to ensure that a proposed delegation does not create a cycle in this directed graph.

A new delegation $X \to Y$ is valid if and only if Y is *not* reachable from X in that DAG – if Y is not a descendant of X.

Instead of checking for this condition directly using a depth-first search (DFS) or breadth-first search (BFS), a more efficient approach is to maintain a list of all descendants for each user. This allows us to check if Y is in the list of descendants of X in constant time. The implementation of this algorithm is detailed in the next section.

Implementation Details

A hashmap is used to store the descendants of each user. The keys are user IDs, and the values are sets of user IDs representing the direct delegates of that user. In the code, this hashmap is referred to as "`inverse_indirect_map`".

```
"inverse_indirect_map": {
  "B": ["A"],
  "C": ["B", "A"],
  "D": ["C", "B", "A"]
}
```

Figure 4.5: Example of a hashmap for users A, B, C, and D. User A has delegated to B, user B has delegated to C, and user C has delegated to D. Consequently, the descendants of user D are A, B and C.

This map enables several key operations required for maintaining a consistent and cycle-free delegation graph:

- **Check Delegation Validity:** To determine whether a delegation $X \to Y$ would create a cycle, the system checks if Y already appears in the set of descendants of X. If so, the new delegation is invalid. This check takes $O(1)$ time.

```
const inverse_indirect_map = this.G.D.get_inverse_indirect_map(pid);
const descendant_set = inverse_indirect_map.get(delegate_vid);
if (descendant_set.has(myvid)) {
  cycle = true;
}
```

Figure 4.6: Code for checking if a delegation is valid. This check is triggered when a user clicks on a delegate link. The map is retrieved from the synchronised local cache, and the set of descendants is used to confirm that a cycle would not be formed.

- **Add Delegation Edge:** When a new delegation $X \to Y$ is accepted, the system must ensure that the descendant relationship is updated consistently. Specifically, for Y and every user u such that $Y \in \texttt{desc}(u)$, their descendants must be updated to include both X and all of X's current descendants.
- **Remove Delegation Edge:** When a delegation $X \to Y$ is removed, the system must ensure that the descendant relationship is updated consistently. Specifically, for Y and every user u such that $Y \in \texttt{desc}(u)$, their descendants must be updated to remove both X and all of X's current descendants.

Synchronisation – TODO

Explain CouchDB problem with poll db and user db. Every user needs to be able to fetch and push `inverse_indirect_map`.

Practical Challenges – TODO

4.2.4 User Interface Changes – TODO

- Re-bound the “delegate none/some/all” switch so it toggles *only* the current option and refreshes its label after server confirmation.
- Re-implemented the “vote is used for n others” badge so it recomputes from the latest `inverse_indirect_map` rather than client guesses.
- Added unobtrusive toast notifications when a delegation request is rejected (e.g. cycle detected) so the user can immediately choose another delegate.

4.2.5 Summary – TODO

4.3 Implement Ranked Delegation into Vodle

This feature introduced ranked delegation using the MinSum rule, allowing users to list fallback delegates in case their primary choice was unavailable.

- UI for setting delegate rankings
- New UI when making a poll to allow user to select ranked delegation.
- `direct_delegation_map`: maps user IDs to list of ranked delegates [[delegationid, rank, status]...]
- Explanation and application of the MinSum rule
- How do we determine who is a casting voter?
- Implementation of ranked path resolution
- Illustrations and code snippets

Challenges

The MinSum rule had to be implemented efficiently using only browser-based resources. Ranking resolution had to preserve user intent while avoiding delegation ambiguity. Providing visual feedback to help users understand how rankings would resolve added an additional layer of design complexity.

4.4 Implement a Vote Splitting Delegation Mechanism into Vodle

Vote splitting was implemented to allow users to distribute fractional influence to multiple delegates.

- UI for assigning weights
- modify`direct_delegation_map`to include weights [[delegationid, weight, status]...]
- Computation of weighted vote outcomes
- Constraints:
  - Weight sum limit (`< 1.0`)
  - Error handling
- Optimisations to limit database writes.
- Algorithm integration and frontend testing

Challenges

The vote splitting logic needed to maintain consistency with the MaxParC aggregation model, while ensuring intuitive user experience. Edge cases (e.g., partially overlapping delegate chains or missing data) introduced complexity during testing. Rendering weight distributions clearly in the UI while keeping the interface lightweight was a recurring challenge.

4.5 Implement the Ability to Delegate Individual Options to Different Users

This feature enabled per-option delegation, allowing users to assign a different delegate for each item in a poll.

- Per-option delegate selection interface
- Independent resolution of each delegated option
- talk about nested map - need to take care to serialise.
- `direct_delegation_map`: `option_id` -> `user_id` -> [delegationid, null, status]
- `inverse_indirect_map`: `option_id` -> `user_id` -> list of users who have delegated to them, either directly or indirectly.
- Storage schema modifications

Challenges

This mechanism required updates to the internal delegation logic to handle resolution at the option level. The user interface also had to be adapted to display multiple concurrent delegate selections without overwhelming the user. Debugging resolution logic for hybrid delegation modes (e.g., one direct, one split, one ranked) was non-trivial.

4.6 Simulate Delegation Mechanisms – Can Remove?

The simulation objective was de-scoped due to time constraints and prioritisation of implementation work. While initial planning and framework selection (Mesa) were completed, no functional simulation code was delivered. The decision to drop this extension is discussed further in the Project Management chapter.

4.7 Design Decisions and Trade-offs

- All logic had to run client-side due to the serverless CouchDB architecture, limiting complexity and computational resources.
- A consistent JSON format was required for all data models, impacting flexibility in data design.
- Trade-offs were made between expressive delegation types and usability, particularly in the option-specific and vote splitting interfaces.

4.8 Summary

- Each objective was successfully implemented within the constraints of the vodle platform.
- Challenges were primarily technical (client-side performance, real-time resolution) and design-oriented (clarity and control for users).
- The final implementation offers a modular, extensible delegation system that addresses the key theoretical and practical limitations outlined in earlier chapters.

Chapter 5

Evaluation

The following

5.1 Testing

- unit testing for delegation algorithms (minsum, vote splitting and test cycle checking for standard delegation)
-
- make sure all requirements are done (can format as table)

5.2 Requirements Evaluation - TODO After Requirements are 100% Done

The following section evaluates the project against the requirements set out in section 3. For each objective, their corresponding requirements are listed, along with a brief description of how they were/ were not met.

5.2.1 Core Objective 1: Implement a Core Delegation Model into Vodle

5.2.2 Core Objective 2: Implement Ranked Delegation into Vodle

5.2.3 Core Objective 3: Implement a Vote Splitting Delegation Mechanism into Vodle

5.2.4 Core Objective 4: Implement the Ability to Delegate Individual Options to Different Users

5.2.5 Extension Objective 1: Simulate Delegation Mechanisms

5.3 Feedback From Customer

get quote from Jobst: what he likes and dislikes about the implementation. also include specifications about what the final delegation implementation will be (vote splitting).

Chapter 6

Project Management

This chapter outlines the project's management approach, including the development methodology, planning, and reflections on the process. It also considers legal and ethical issues and assesses key risks associated with the project.

6.1 Methodology

The project adopted an agile methodology, chosen for its flexibility, iterative development cycle, and emphasis on frequent customer feedback. The work involved incrementally building a series of interdependent features into vodle – starting with a core delegation mechanism, and progressively expanding functionality to include ranked delegation, vote splitting, and finally, per-option delegation. This iterative approach allowed each new feature to build directly upon the last, ensuring ongoing compatibility and adaptability in design decisions as the system evolved.

Agile methodology was particularly suitable for this project due to the involvement of an active "customer" figure: Jobst Heitzig, co-supervisor and original creator of vodle. Heitzig played a crucial role in defining system expectations and guiding design decisions based on practical, real-world considerations. Regular meetings, held fortnightly with both Jobst Heitzig and Markus Brill, facilitated continuous feedback and review of progress, enabling rapid adaptation of development plans. This feedback cycle closely reflects the Agile Manifesto's principles of early and continuous delivery, as well as close collaboration between developers and stakeholders (Beck et al., 2001).

Other project management approaches, such as Waterfall, were also evaluated but ultimately dismissed due to their inherent rigidity. Although Waterfall initially appeared attractive due to clearly defined phases and comprehensive documentation at each stage, its requirement to specify the complete project scope upfront was incompatible with the evolving nature of the project. Given the shorter time frame and the dynamic nature of feature requirements, the flexibility afforded by agile was critical to the project's success.

Scrum, one of the most widely adopted agile frameworks (used by approximately 63% of agile teams (VersionOne, 2020)), was also considered. Its structured, sprint-based cycles, clear team roles, and structured ceremonies such as sprint planning and reviews were appealing for maintaining focused implementation and streamlined communication. However, the project's constraints – limited availability due to academic commitments and a small team size – made Scrum's daily stand-up meetings and fixed sprint lengths impractical. As a result, the project adopted an adapted agile approach: progress was reviewed every two weeks, effectively maintaining the advantages of frequent feedback without the constraints and scheduling pressures imposed by full Scrum ceremonies.

Each iteration of development produced a functional, testable feature that could be immediately evaluated and integrated into the broader system. This method significantly reduced the risk of late-stage integration issues and ensured steady, measurable progress throughout the project's duration. Overall, the agile methodology's iterative, feedback-oriented structure proved highly effective, meeting both the technical complexity and collaborative needs inherent in this work.

6.2 Plan

The project plan was organised into objectives (see Section 3) that built on one another in sequence:

- **Core Objective 1:** Implement a Core Delegation Model into Vodle.
- **Core Objective 2:** Implement Ranked Delegation into Vodle.
- **Core Objective 3:** Implement a Vote Splitting Delegation Mechanism into Vodle.
- **Core Objective 4:** Implement the Ability to Delegate Individual Options to Different Users.
- **Extension Objective 1:** Simulate Delegation Mechanisms.

This objective-led structure was well-suited to the agile approach, allowing each milestone to be treated as an iteration with a deliverable at the end. A Gantt chart (see below) was created to visualise the project timeline and to track dependencies and progress.

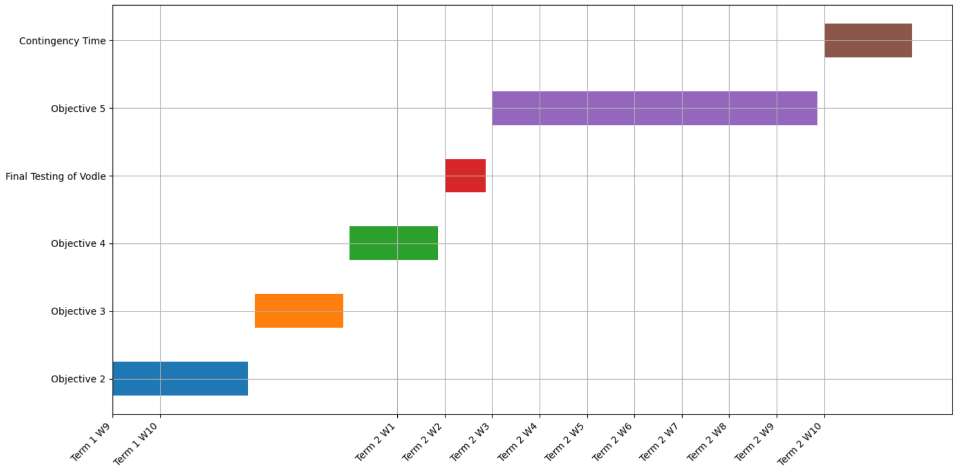

Figure 6.1: Gantt chart illustrating the project plan from the progress report.

6.3 Changes to the Project Plan

The original project plan, illustrated in the Gantt chart (Figure 6.1), outlined a linear progression through the core objectives. However, several adjustments were made during the course of the project to reflect evolving priorities and unforeseen technical challenges.

The most significant change was the decision to de-scope the extension objective (simulating delegation mechanisms). This was prompted by two main factors. First, implementing the core objectives proved more technically demanding than initially anticipated – particularly ranked delegation and vote splitting. These challenges required more time and attention than expected, leaving limited capacity to complete the extension objective without compromising the quality of the core deliverables.

Second, during the background research phase, it became clear that similar investigations into delegation behaviour had already been conducted – most notably by Brill et al. (2021). Although their study did not use agent based modelling, instead using networks from synthetic and real-world networks (such as partial networks from Facebook, Twitter, Slashdot, etc.), it provided a comprehensive empirical evaluation of ranked delegation rules using various metrics such as maximum vote path length, average vote path rank, the number of isolated voters (voters without a delegation path) and many more.

Given the depth and relevance of these findings, replicating the analysis through agent-based modelling, especially within the project's limited timeframe, was deemed unnecessary. Instead, the project focused on fully delivering and refining the core objectives, which aligned more directly with vodle's platform goals and would have a better impact on the user experience of vodle.

A second change involved reversing the development order of objectives 3 and 4. Originally, objective 3 (vote splitting) was scheduled to follow objective 2. However, during the Christmas development period, it became clear that objective 4 (per-option delegation) could be completed more quickly and required fewer algorithmic dependencies. To maintain development momentum, objective 4 was brought forward.

This adjustment helped mitigate project risk. Objective 4 involved minimal changes to the database schema and integrated easily with UI components developed for earlier objectives. In contrast, objective 3 introduced more complex computational logic and performance concerns, which demanded additional design, testing and changes to the database. Tackling objective 4 earlier helped avoid potential cascading delays and ensured a smoother integration process later in development.

6.3.1 Actual Timeline vs Planned Timeline

While the original project plan provided a clear sequence for implementing each objective, the actual progression deviated in several key areas due to technical challenges, interface considerations, and evolving priorities. Figure 6.2 (below) shows the actual timeline of the project, which can be compared to the original plan (Figure 6.1).

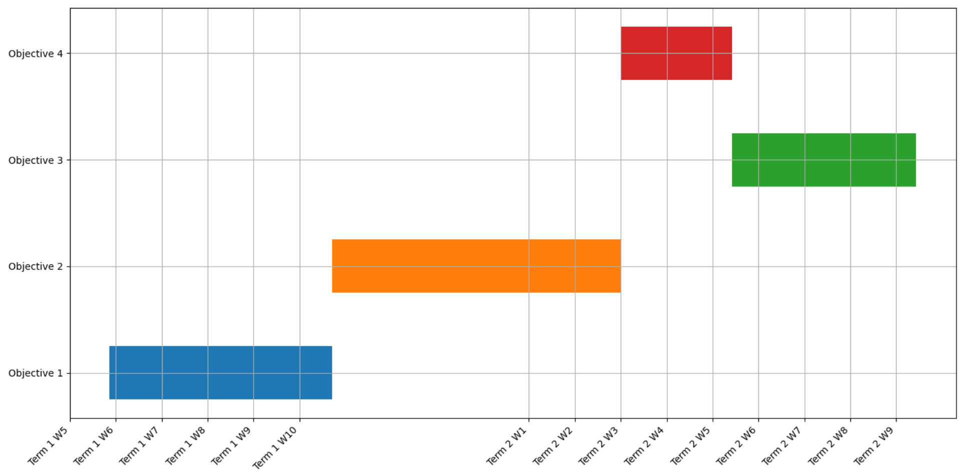

Figure 6.2: Gantt chart illustrating the actual timeline of the project.

The most significant deviations from the original plan included:

- **Objective 1 (Core Delegation Model)** commenced approximately one week later than planned. This delay stemmed from the need for deeper familiarization with vodle's existing codebase, particularly its database schema and frontend architecture. Understanding these components was essential to ensure that new delegation features could integrate seamlessly without disrupting existing functionality. This initial exploration phase, while time consuming, was crucial for establishing a solid foundation for subsequent development.

- **Objective 2 (Ranked Delegation)** extended significantly beyond its initial timeframe. Originally scheduled for completion during the Christmas break, development continued until Term 2, Week 3. This extension was primarily due to the unforeseen complexities (as discussed in Section 4.3) which required both sophisticated transitive resolution logic and intuitive user interface feedback mechanisms, which proved more challenging than initially estimated.

- **Objectives 3 and 4**'s order was swapped, as discussed earlier in this section. While the original plan positioned Objective 3 (Vote Splitting) to follow Objective 2, development logistics and dependency considerations prompted a shift to implement Objective 4 (Per-Option Delegation) first. This decision minimised integration risks, as Objective 4 required fewer schema modifications and built more directly on existing UI components, in particular those developed for Objective 2 (Ranked Delegation).

- **Extension Objective (Simulation)** was ultimately descoped from the project plan. This decision reflected both time constraints imposed by the extended development periods for core objectives and the discovery of existing comprehensive research on delegation behavior by Brill et al. (2021). Focusing resources on delivering robust implementations of the core objectives was deemed more valuable than duplicating existing research.

Despite these deviations, the overall project structure remained intact and successful. The buffer period allocated at the end of Term 2 served its intended purpose as contingency time, effectively absorbing the delays in Objectives 1 and 2. This strategic planning ensured that despite timeline adjustments, the project remained on track for completion with all core objectives delivered to high quality standards.

The ability to adapt the project plan while maintaining focus on delivering the core functionality demonstrates the value of the chosen agile methodology. Rather than rigidly adhering to potentially unrealistic timelines, the flexible approach allowed for continuous reassessment and prioritisation based on evolving technical insights and stakeholder feedback.

6.4 Risk Assessment

The following section provides a detailed analysis of the risks identified in the risk assessment table below (see Table 6.1). Each risk is discussed individually in detailed later on in the section, outlining its implications and the strategies proposed to mitigate potential issues.

Risk	**Likelihood**	**Mitigation Strategy**
Breaking the live vodle site during development	Medium	Use Git branching to isolate development from production environments. Conduct local testing before deployment.
Feature complexity exceeds estimates	High	Prioritise core objectives and maintain flexibility in scope.
Lack of engagement from supervisors or stakeholders	Low	Maintain regular communication through scheduled meetings.
Data loss or corruption	Low	Use Git for version control and take regular local backups.

Table 6.1: Key risks identified and their mitigation strategies

Breaking the Live Vodle Site During Development

Likelihood: Medium

Description: Modifications to the existing vodle platform could unintentionally introduce downtime or impair existing functionality on the live website. Any disruptions could negatively impact real users' interactions, leading to dissatisfaction and loss of trust in the platform.

Mitigation: Development activities will utilise Git branching to isolate new code from the production environment. Features will be developed and rigorously tested in local or staging environments before integration with the live deployment. Incremental rollouts and thorough pre-deployment testing will further help identify potential problems early, allowing quick remediation or rollback.

Feature Complexity Exceeds Estimates

Likelihood: High

Description: Advanced features such as ranked delegation and vote splitting may prove more complex than initially anticipated. Unexpected complexity can lead to delays, reduced functionality, or incomplete implementations, potentially affecting the project's timeline and deliverables.

Mitigation: Core objectives have been clearly defined and prioritised, ensuring focus remains on essential functionality. In cases of higher than anticipated complexity, resources will be redirected towards completing critical core features first, while the extension objective (agent based modelling) can be scaled back or postponed as needed.

Regular agile reviews will monitor progress closely, facilitating early identification and management of complexity-related issues.

Lack of Engagement from Supervisors or Stakeholders

Likelihood: Low

Description: Regular feedback and engagement from supervisors and stakeholders are crucial to ensure alignment with project goals, requirements, and user expectations. Insufficient feedback could result in misaligned implementations or objectives that do not fully meet user needs.

Mitigation: Fortnightly meetings have been scheduled with both the primary supervisor (Markus Brill) and the co-supervisor (Jobst Heitzig), who also fulfils the role of the project customer. This structured schedule ensures consistent opportunities for input and feedback. Additionally, a Telegram group chat is available to handle urgent queries and maintain ongoing dialogue.

Data Loss or Corruption – Code

Likelihood: Low

Description: Development activities pose a risk of code loss or corruption due to accidental deletion, unintended changes, or version conflicts. Such incidents could significantly delay development and necessitate additional time for recovery.

Mitigation: Version control will be rigorously maintained using Git, with frequent commits and descriptive commit messages ensuring traceability. Regular backups of the repository will be taken to safeguard against accidental loss, providing straightforward recovery paths when needed.

Data Loss or Corruption – Database

Likelihood: Low

Description: While unlikely, corruption of the CouchDB database during development could occur due to improper schema modifications or accidental changes.

Mitigation: No mitigation – in the event of data corruption, the development database will be reset to its original state. As all data in the database is poll and user specific, it does not impact development as no important data is stored in the production environment.

6.5 Risk Management Reflection

This section evaluates how effectively the project's risk management strategies addressed both anticipated and unforeseen challenges. It examines which risks were realised, how mitigation strategies performed in practice, and identifies lessons learned that could inform future projects.

Breaking the Live Vodle Site During Development

The use of Git branching strategies and comprehensive local testing successfully prevented any disruption to the live site. The separation between development and production environments ensured stability throughout the project lifecycle. This disciplined approach proved valuable despite adding some overhead to the development process.

Feature Complexity Exceeds Estimates

This materialised as the most significant risk, particularly during the implementation of ranked delegation (Objective 2) and vote splitting mechanisms (Objective 3). The algorithmic complexity and UI considerations extended development beyond initial estimates. The strategy of prioritising core objectives proved invaluable, allowing the project to successfully deliver all essential functionality. The decision to descope the extension objective demonstrates effective risk management balancing ambition with practical constraints. In hindsight, breaking complex features into smaller subtasks during planning might have improved estimation accuracy, though the agile methodology compensated through its inherent flexibility.

Lack of Engagement from Supervisors or Stakeholders

This risk did not take place. The fortnightly meetings and additional communication channels maintained strong engagement throughout the project. Jobst Heitzig's dual role as co-supervisor and customer representative provided crucial domain expertise and timely feedback on design decisions.

Data Loss or Corruption No significant data loss incidents occurred. Git version control provided reliable tracking of code changes, while the lightweight approach to database management proved appropriate as no critical data was compromised.

Summary Overall, the risk management approach proved effective in supporting project delivery despite several challenges. The most significant risk – feature complexity exceeding estimates – did occur and required timeline adjustments, but the contingency buffer and flexible scope management successfully mitigated its impact. The disciplined development approach prevented any disruption to the live site, while strong stakeholder engagement and version control systems effectively addressed the remaining identified risks. The experience highlighted the importance of integrating contingency time into project schedules and maintaining flexibility when dealing with technically complex features.

6.6 Legal and Ethical Considerations

As vodle may eventually be used to gather votes on sensitive topics, particular attention was paid to ensuring user privacy and system fairness throughout development.

Delegation chains are resolved internally within the browser and are never publicly exposed. A key design feature is that a delegation only becomes active when the invited user explicitly accepts the invitation. This ensures that no information about a user's voting intentions or delegation preferences is shared without their consent. The delegate only becomes aware of the relationship once they actively confirm it, and the delegator retains full control to revoke or modify the delegation at any time.

This mechanism protects the confidentiality of voter relationships and ensures that vote flows remain private unless both parties agree to the delegation. As such, even in a scenario where a vote is passed through multiple users, no individual along the chain gains access to the full path unless explicitly authorised.

Furthermore, no personal data was collected or processed for the purposes of this project. All stored information relates strictly to poll participation and delegation structures, with no link to identifiable personal attributes. As a result, no changes to vodle's terms of service were required, and the project remains compliant with relevant data protection and ethical standards.

6.7 Overall/Self Reflection - TODO

Chapter 7

Conclusions

7.1 Author's Assessment of the Project

"It can be a useful exercise for you (and a point of consolidation for the reader) to put together a brief summary of what you have achieved. This is not a compulsory section, but a self-assessment is welcome. A suggested format for this is to include a short section entitled 'Author's Assessment of the Project' consisting of brief (up to 100 words) answers to each of the following questions.

- What is the (technical) contribution of this project?
- Why should this contribution be considered relevant and important for the subject of your degree?
- How can others make use of the work in this project?
- Why should this project be considered an achievement?
- What are the limitations of this project?

"

What is the (technical) contribution of this project?

Why should this contribution be considered relevant and important for the subject of your degree?

How can others make use of the work in this project?

Why should this project be considered an achievement?

What are the limitations of this project?

7.2 Future Work

- More algorithms for ranked delegation
- Global Delegations – like liquid feedback

Bibliography

Angular Team, . *Angular - Web Framework*. Google, 2024. URL `https://angular.io/`. `https://angular.io/`.

Apache CouchDB Project, . *Apache CouchDB*. The Apache Software Foundation, 2024. URL `https://couchdb.apache.org/`. Accessed April 2025. `https://couchdb.apache.org/`.

Beck, Kent & Beedle, Mike & van Bennekum, Arie & Cockburn, Alistair & Cunningham, Ward & Fowler, Martin & Grenning, James & Highsmith, Jim & Hunt, Andrew & Jeffries, Ron & Kern, Jon & Marick, Brian & Martin, Robert C. & Mellor, Steve & Schwaber, Ken & Sutherland, Jeff & Thomas, Dave. Manifesto for agile software development, 2001. URL `https://agilemanifesto.org`. Accessed: 2025-04-21.

Behrens, Jan & Swierczek, Björn. Preferential delegation and the problem of negative voting weight. *The Liquid Democracy Journal*, 3:6–34, 2015.

Behrens, Jan & Kistner, Axel & Nitsche, Andreas & Swierczek, Björn. *The Principles of LiquidFeedback*. Interacktive Demokratie, 2014.

Bersetche, Francisco M. Generalizing Liquid Democracy to multi-agent delegation: A Voting Power Measure and Equilibrium Analysis, April 2024.

Blum, Christian & Zuber, Christina Isabel. Liquid Democracy: Potentials, Problems, and Perspectives. *Journal of Political Philosophy*, 24(2):162–182, June 2016. ISSN 0963-8016, 1467-9760. doi: 10.1111/jopp.12065. URL `https://onlinelibrary.wiley.com/doi/10.1111/jopp.12065`.

Bonabeau, Eric. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99:7280–7287, 2002.

Brill, Markus & Delemazure, Théo & George, Anne-Marie & Lackner, Martin & Schmidt-Kraepelin, Ulrike. Liquid Democracy with Ranked Delegations, December 2021.

Brill, Markus & Delemazure, Théo & George, Anne-Marie & Lackner, Martin & Schmidt-Kraepelin, Ulrike. Liquid Democracy with Ranked Delegations. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(5):4884–4891, June 2022. ISSN 2374-3468. doi: 10.1609/aaai.v36i5.20417. URL `https://ojs.aaai.org/index.php/AAAI/article/view/20417`. Number: 5.

Christoff, Zoé & Grossi, Davide. Liquid democracy: An analysis in binary aggregation and diffusion, 2017. URL `https://arxiv.org/abs/1612.08048`.

Collier, Nick. Repast: An extensible framework for agent simulation. *The University of Chicago's social science research*, 36:2003, 2003.

Degrave, Jonas. Resolving multi-proxy transitive vote delegation. (arXiv:1412.4039), December 2014. doi: 10.48550/arXiv.1412.4039.

Ford, Bryan Alexander. Delegative Democracy. May 2002. URL `https://infoscience.epfl.ch/handle/20.500.14299/156450`.

Hall, Andrew B. & Miyazaki, Sho. What Happens When Anyone Can Be Your Representative? Studying the Use of Liquid Democracy for High-Stakes Decisions in Online Platforms. Technical report, 2024.

Hardt, Steve & Lopes, Lia. Google Votes: A Liquid Democracy Experiment on a Corporate Social Network. *Defensive Publications Series*, June 2015.

Heitzig, Jobst & Simmons, Forest W. & Constantino, Sara M. Fair group decisions via non-deterministic proportional consensus. *Social Choice and Welfare*, May 2024. ISSN 1432-217X. doi: 10.1007/s00355-024-01524-3. URL `http://dx.doi.org/10.1007/s00355-024-01524-3`. Publisher: Springer Science and Business Media LLC.

Ionic Team, . *Ionic Framework - Cross-Platform Mobile App Development*. Ionic, 2024. URL `https://ionicframework.com/`. Accessed April 2025. `https://ionicframework.com/`.

Kazil, Jackie & Masad, David & Crooks, Andrew. Utilizing Python for Agent-Based Modeling: The Mesa Framework. In Thomson, Robert & Bisgin, Halil & Dancy, Christopher & Hyder, Ayaz & Hussain, Muhammad, editors, *Social, Cultural, and Behavioral Modeling*, pages 308–317, Cham, 2020. Springer International Publishing. ISBN 978-3-030-61255-9.

Kling, Christoph Carl & Kunegis, Jerome & Hartmann, Heinrich & Strohmaier, Markus & Staab, Steffen. Voting behaviour and power in online democracy: A study of liquidfeedback in germany's pirate party, 2015. URL `https://arxiv.org/abs/1503.07723`.

Kotsialou, Grammateia & Riley, Luke. Incentivising Participation in Liquid Democracy with Breadth-First Delegation. *New Zealand*, 2020.

Sommerville, Ian. *Software Engineering*. Always Learning. Pearson, Boston Columbus Indianapolis New York San Francisco Hoboken Amsterdam Cape Town Dubai London, tenth edition edition, 2016. ISBN 978-1-292-09613-1 978-1-292-09614-8.

Sven Becker, DER SPIEGEL. Liquid Democracy: Web Platform Makes Professor Most Powerful Pirate - spiegel.de, 2012. URL `https://www.spiegel.de/international/germany/liquid-democracy-web-platform-makes-professor-most-powerful-pirate-a-818683.html`.

Tisue, Seth & Wilensky, Uri. Netlogo: A simple environment for modeling complexity. In *International Conference on Complex Systems*, volume 21, pages 16–21. Citeseer, 2004.

Vahdati, Ali. Agents.jl: Agent-based modeling framework in Julia. *Journal of Open Source Software*, 4(42):1611, October 2019. ISSN 2475-9066. doi: 10.21105/joss.01611.

VersionOne, . 14th annual state of agile report. `https://stateofagile.com`, 2020. Accessed: 2025-04-09.