Liquid Democracy for Rating Systems

Hemanath Peddireddi
Department of Computer Science
University of Warwick

Supervised by Markus Brill

28 April 2025

THE UNIVERSITY OF WARWICK

Acknowledgements

I would like to sincerely thank my supervisor, Markus Brill, for his guidance and support throughout this project. His extensive knowledge of liquid democracy was invaluable in helping to shape both the theoretical foundations and the practical design decisions of the work. His feedback was consistently insightful and helped refine the project's direction at every stage.

I am also grateful to my co-supervisor, Jobst Heitzig, for his ongoing support during development. As the main developer of vodle and acting as the project's customer, his insights into the platform's requirements and goals were essential for the successful integration of liquid democracy features. The weighted delegation mechanism based on a trust matrix model, which forms a key part of the final system, was also proposed by him. His technical guidance and responsiveness throughout the project were greatly appreciated.

Finally, I would like to thank my family and friends for their support and encouragement during the course of this project. Their patience, understanding, and belief in me provided important motivation throughout the development process.

Abstract

Liquid democracy offers a flexible alternative to traditional methods of voting, allowing individuals to either vote directly or delegate their vote to a trusted peer. This project integrates a comprehensive liquid democracy system into vodle, an existing open-source platform for web-based group decision-making, to enable more flexible and inclusive participation.

The implementation introduces several key features: transitive delegation, ranked delegation with backup options, per-option delegation, and weighted delegation. These features provide flexible, reliable, and secure delegation, supporting varying levels of user engagement and expertise.

By combining technical innovations with an intuitive interface, vodle ensures that users can participate in decision-making even when they are not continuously active, making liquid democracy practical for real-time, web-based environments.

Keywords: liquid democracy, ranked delegation, weighted delegation, flexible participation, user autonomy, online decision-making, real-time voting systems, web-based voting systems

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation – TODO

In group settings ranging from online communities to organisations, collective decision making is both essential and difficult to scale. Direct democracy empowers individuals by letting everyone vote on every issue, but struggles with engagement as participation grows. Representative democracy improves scalability but often reduces accountability and flexibility (Ford, 2002; Blum and Zuber, 2016)

Liquid democracy is a hybrid model that aims to balance these trade-offs. It allows individuals to either vote directly or delegate their voting power to others they trust. This approach combines the transparency and agency of direct democracy with the scalability of representation, offering a dynamic alternative for participatory systems.

This project applies liquid democracy within vodle, a web-based platform for group decision making. Vodle allows users to express nuanced preferences across decision options and promotes open, participatory decision making. However, it inherits a common limitation: users may lack the time, interest, or confidence to engage with every decision, leading to underrepresentation and disengagement.

By integrating liquid democracy into vodle, the project aims to give users more flexibility in how their preferences are represented. Delegation features allow users to stay involved even when they choose not to vote directly, making the platform more scalable, inclusive, and responsive to varying levels of user engagement – all while maintaining transparency and user control.

1.2 Project Goals

The goal of this project is to design and implement a liquid democracy system within vodle. This involves building flexible delegation mechanisms that support diverse participation styles and address the limitations of traditional direct voting.

The system introduces several key features:

- **Ranked delegation**, allowing users to specify trusted delegates in order of preference.
- **Per-option delegation**, enabling different delegates for different options within a poll.
- **Weighte delegation**, using a trust-based model to distribute voting power across multiple delegates.

These features are designed to enhance participation, reduce the concentration of voting power, and improve the resilience of the platform. The project focuses on creating an intuitive and efficient implementation that aligns with vodle's existing architecture and emphasises user autonomy.

1.3 Structure of This Report

The remainder of this report is structured as follows:

- will list

Chapter 2

Background Research

This chapter provides the necessary foundation for designing and implementing a liquid democracy system within vodle. It begins by comparing liquid democracy to traditional models (direct and representative democracy) highlighting its potential to balance participation and scalability through transitive and revocable delegation.

The chapter then examines key limitations of liquid democracy in practice, including delegation cycles, abstentions, and the emergence of super-voters. These issues are explored in detail, alongside the strategies proposed in the literature to mitigate them.

Additionally, real-world deployments of liquid democracy are considered, drawing lessons from systems such as LiquidFeedback and Google Votes. These examples inform both the technical feasibility and design challenges of integrating delegation mechanisms into online platforms.

Finally, the chapter outlines the technical foundations of vodle, including its architecture and existing (incomplete) delegation features. These constraints play a key role in shaping the design choices presented in later chapters.

2.1 Liquid Democracy

While liquid democracy was briefly introduced in Chapter 1, this section provides a more detailed rationale for its relevance – particularly in the context of decision-making systems like vodle.

Direct democracy maximises personal agency and transparency by allowing individuals to vote on every issue. It is often viewed as the most egalitarian form of governance, as it ensures that all participants have a direct say in decisions. However, it becomes impractical at scale. Expecting all users to remain consistently informed and engaged across all issues is unrealistic, particularly in large or diverse groups (Ford, 2002). As Ford observes, the assumption that the majority can or will make consistently well-informed decisions across a broad range of topics does not hold in practice. This can lead to voter fatigue, low turnout, and uninformed or irrational decision-making.

Representative democracy addresses the scalability problem by allowing users to elect officials who vote on their behalf. This model is forms the basis of most modern democracies, enabling stable governance in large populations. It allows representatives to develop expertise and reduces the decision-making burden on the general population. However, the model has key limitations: between elections, representatives may act without sufficient accountability, and voters have limited influence over individual decisions (Blum and Zuber, 2016). As a result, decision-making can become misaligned with the evolving preferences of the electorate.

Liquid democracy attempts to reconcile these competing trade-offs. It allows each participant to either vote directly or delegate (entrust) their vote to another participant (a delegate). Delegates can in turn delegate their votes, forming chains of trust. This is known as *transitive delegation*: voting power is passed along the chain until it reaches a user who casts a vote. Delegations are also *revocable*, allowing users to reclaim and reassign their vote at any time.

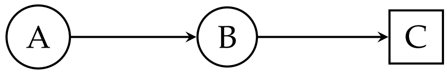


Figure 2.1: Example of transitivity in action: Voter A delegates to Voter B, who delegates to Voter C. Voter C then casts a vote with the weight of three individuals (A, B and C).

By supporting both direct and delegated participation, liquid democracy allows users to engage at varying levels depending on their interest, expertise, or availability. This flexibility makes it a promising model for decision making in online platforms such as vodle, where participation levels naturally fluctuate.

2.1.1 Issues With Liquid Democracy

While liquid democracy offers an elegant compromise between agency and scalability, it introduces several non-trivial implementation challenges. This section identifies three core issues that threaten its robustness and fairness in practice: cycles in delegation chains, vote loss due to abstentions, and the disproportionate influence of super-voters.

To support this discussion, the following diagram legend is used to visually distinguish between voter behaviours:

Role	Description	Symbol
Delegated voter	Has delegated their vote and does not cast one directly	Circle
Casting voter	Casts their own vote and has not delegated	Square
Abstaining voter	Neither delegates nor casts their own vote	Triangle

Table 2.1: Diagram legend showing symbols used for different voter behaviours in a liquid democracy context.

Delegation Cycles

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 (see Figure 2.2) and can be treated as a loss of representation (Christoff and Grossi, 2017a).

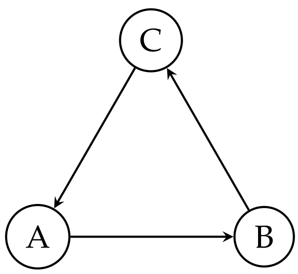


Figure 2.2: Delegation cycle: A delegates to B, B to C, and C back to A.

This issue is particularly problematic because it can nullify votes without the affected users becoming aware of it. 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 particularly likely to emerge in dynamic, real-time voting systems like vodle, where users can add, remove, or modify delegations at any time. Even delegation chains that are initially valid may later form part of a cycle as other participants update their preferences. This makes cycle detection not just a one-off validation step, but an ongoing requirement for maintaining consistency and ensuring vote resolution remains reliable.

Abstentions

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 (see Figure 2.3), 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.

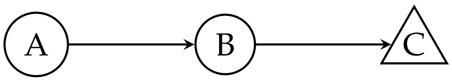


Figure 2.3: Delegation chain ending in abstention: A delegates to B, B to C. C abstains, causing the votes of A and B to be lost.

Super-Voters

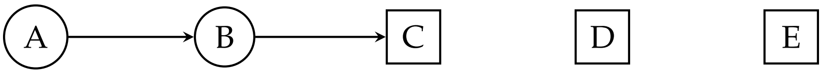


Figure 2.4: Super-voter: A delegates to B, B to C. No matter which vote D or E cast, C's vote will always determine the outcome as it has a weight of 3.

In liquid democracy, a *super-voter* is a user who receives a large number of delegations, thereby accumulating significant voting power (Kling et al., 2015). While such concentration may arise from genuine trust, it risks creating imbalances that contradict the egalitarian goals of the system. Super-voters can effectively act as unelected representatives – potentially swaying results with little accountability.

Even when systems allow voters to revoke delegations at any time, many users may not actively monitor how their vote is used. This can lead to persistent power structures where a small number of users hold substantial, often unnoticed, influence over outcomes.

This phenomenon is not just theoretical. In the German Pirate Party's use of LiquidFeedback, some participants became so dominant that their votes carried decree-like weight, even though they were not formally elected (Sven Becker, 2012; Kling et al., 2015). While Kling et al. (2015) observed that these super-voters generally aligned with majority opinion, contributing to system stability rather than distorting it, their disproportionate influence still raises concerns about transparency and the extent to which individual voters retain meaningful control over their vote.

Super-voting is not confined to traditional political platforms. In decentralised autonomous organisations (DAOs), which use token-based voting on blockchains, similar patterns emerge. Hall and Miyazaki (2024) found that in many DAOs, voting power was highly concentrated among a few delegates due to low overall participation. In some cases, such as Gitcoin, over 90% of votes cast were controlled by the top five delegates.

These examples underscore the importance of delegation mechanisms that can curb excessive power accumulation. Techniques such as weighted delegation and capping the maximum voting power of an individual are essential to preserving fairness, especially in systems like vodle that prioritise inclusivity and trust-based participation.

2.1.2 Variations of Liquid Democracy

This section explores several proposed extensions to liquid democracy that address the limitations outlined above. While many models offer theoretical advantages, particular attention is paid to ranked delegation and weighted delegation – two techniques that are not only promising from a fairness and robustness perspective, but also feasible to implement within vodle's existing architecture.

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 – TODO: ADD DIAGRAMS

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.

Based on these considerations, the project adopts the MinSum rule. It offers a clear trade-off between delegation quality, computational efficiency, and user interpretability, making it well-suited for deployment within vodle. By selecting the path with the lowest total rank sum, MinSum prioritises higher-ranked delegates while avoiding unnecessarily long or indirect chains. This supports user trust and clarity by ensuring that final delegation paths reflect stated preferences in an understandable way.

Weighted Delegation – TODO: ADD DIAGRAMS

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, weighted delegation allows voters to divide their voting power among multiple delegates, rather than assigning it entirely to one. This approach provides greater flexibility and robustness while preserving voter intent more accurately.

Weighted delegation offers several key advantages:

- **Increased resilience:** Distributing votes across multiple delegates mitigates the effect of abstentions by individual delegates.
- **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 weighted delegation 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 proposes a generalisation of liquid democracy in which voters are allowed to divide their voting weight arbitrarily among multiple delegates, while optionally retaining a fraction for themselves. Each agent expresses their preferences through a delegation matrix, where each entry specifies the proportion of voting weight delegated to a given individual. This approach enables a more flexible expression of trust relationships compared to classical single-delegation models.

To ensure the system remains stable and to prevent cases such as delegation cycles, Bersetche introduces an artificial agent, referred to as agent $n + 1$. Every delegation path carries a small probability (ε) of transferring the voting weight to this agent. This mechanism ensures that votes trapped in delegation cycles or excessively long chains are eventually absorbed, preventing them from destabilising the voting process. In the limit as $\varepsilon \to 0$, the model approximates the classical behaviour of liquid democracy without losses, but with $\varepsilon > 0$, it guarantees the existence of equilibrium states.

Votes absorbed by agent $n + 1$ are effectively considered lost. As a result, users who delegate without ultimately terminating in a direct voter may see a fraction of their influence dissipate. This diverges from standard liquid democracy approaches, where all delegated votes are ultimately assumed to participate in the outcome.

Overall, fractional delegation as formalised by Bersetche retains desirable properties such as self-selection, delegation consistency, and conservation of total voting weight (accounting for losses), while enabling more expressive delegation structures and guaranteeing equilibrium under mild conditions.

Trust Matrix Model (proposed by Heitzig)

This model was originally proposed by Jobst Heitzig, co-supervisor of this project, as a highly expressive approach to vote splitting within ratings-based voting systems. Rather than directly casting or delegating votes, voters assign trust values $\text{trust}_{i,j} \in [0, 1]$ to other participants (including themselves), representing the proportion of influence each delegate should have in determining their effective rating. These trust values are used to iteratively compute the final effective rating for each voter.

Let $\text{rating}_{i,x}$ denote the direct rating that voter i assigns to option x, and $\text{eff}_{i,x}$ denote their final effective rating. Then, $\text{eff}_{i,x}$ can be defined as a combination of the voter's own rating and the effective ratings of their delegates, determined by the trust levels specified:

$$\text{eff}_{i,x} = \text{trust}_{i,i} \cdot \text{rating}_{i,x} + \sum_{j \neq i} \text{trust}_{i,j} \cdot \text{eff}_{j,x} \tag{2.1}$$

Each voter's trust values must sum to exactly 1:

$$\sum_{j} \text{trust}_{i,j} = 1$$

The computation proceeds iteratively, with the effective ratings updated at each step based on the current estimates of delegates' effective ratings. This process continues until there is no longer a change in the effective ratings. Convergence is guaranteed when each voter has some nonzero self-trust ($\text{trust}_{i,i} > 0$), as the system then forms a contraction mapping and Banach's fixed point theorem applies.

This model offers the highest granularity and flexibility, allowing voters to finely express confidence across multiple delegates per option. However, it also introduces significant computational overhead and may scale poorly in densely connected delegation graphs. Additionally, users may find it difficult to understand or maintain trust matrices of this complexity, raising potential usability concerns in real-world systems.

Summary of Approaches

- **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 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.

Each weighted-delegation method balances voter expressivity, computational complexity, and ease of implementation differently. After evaluating these trade-offs, this project selects the trust matrix model for its high expressiveness and compatibility with vodle's principles of autonomy and transparency. While more computationally intensive, it captures nuanced trust relationships and enables fine-grained participation – both essential for resilient, inclusive decision-making at scale.

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 (the Schulze method) to decide which one should win. The Schulze method compares proposals in head-to-head matchups and identifies the option with the strongest overall support, even considering indirect chains of preference. This helped ensure that the winning proposal was broadly supported across the electorate. Importantly, LiquidFeedback only accepted a proposal if it clearly beat the default option 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 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

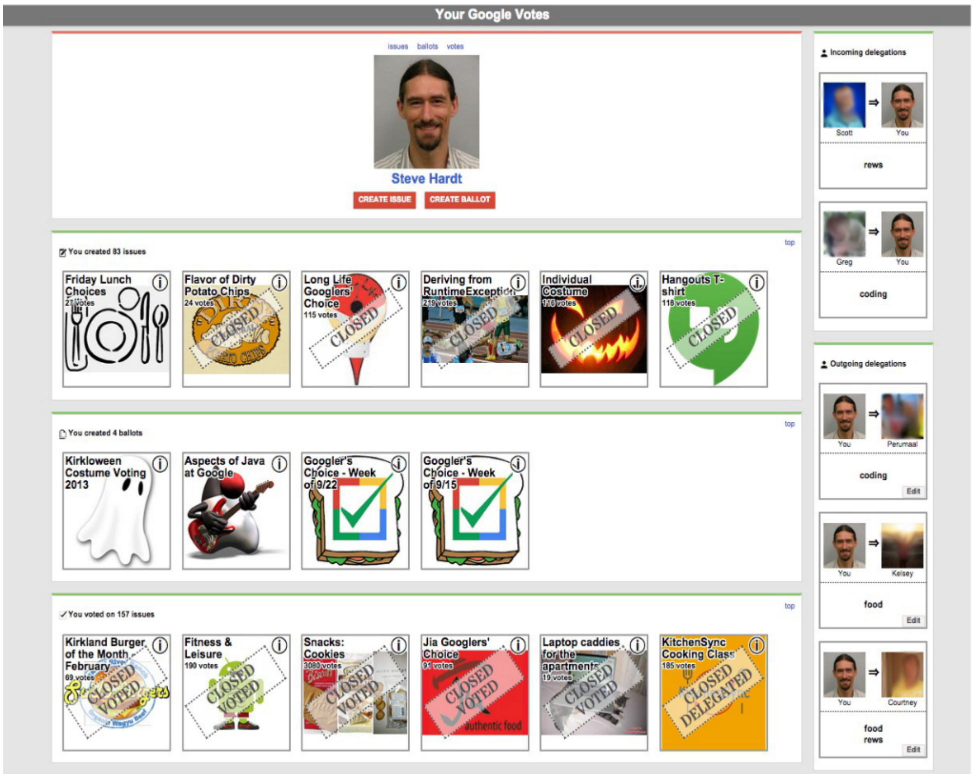


Figure 2.5: Screenshot taken from Hardt and Lopes (2015) showing the user interface of Google Votes.

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 weighted-delegation 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. However, this trade off may be acceptable for highly performance driven solutions.

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

To effectively explore and implement liquid democracy mechanisms, it is essential to understand the design and technical context of the platform into which they are being integrated. Vodle is a web based decision-making tool developed to support participatory group processes through interactive polls and transparent aggregation methods. Its goal is to provide users with flexible, fine grained input mechanisms that encourage compromise between voters, while maintaining accessibility and usability across a broad and diverse user base.

This section introduces the core architecture of vodle, including its underlying rating system (MaxParC) and the technologies that support its operation. These technical foundations were instrumental in shaping the design and feasibility of the advanced delegation features developed in this project.

2.4.1 MaxParC

MaxParC (Maximum Partial Consensus) is the 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.

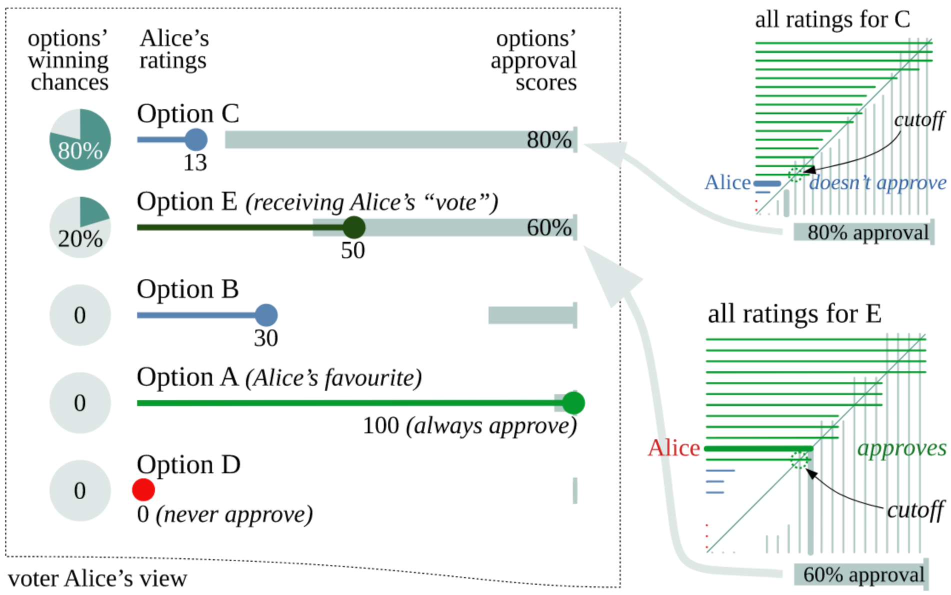


Figure 2.6: Visual representation of MaxParC from the perspective of a voter (Alice). Ratings represent conditional approval thresholds. An option is counted as approved by Alice if the approval bar (light grey) overlaps with her rating needle. Graphic from Heitzig et al. (2024).

Understanding how MaxParC processes ratings is essential for this project, as the proposed weighte delegation 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 Technical Architecture and Implementation Constraints

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 (covered in more detail in Section 4.1) strongly shaped the practical implementation of liquid democracy in vodle. They necessitated a focus on efficient client-side logic as well as careful management of data flow.

2.4.3 Partially Implemented Delegation in Vodle

At the beginning of the project, vodle contained an inactive, partially implemented vote delegation system. Although never deployed in a working state, this prototype included an invitation-based mechanism for initiating delegations, along with early user interface components to support the feature.

However, the implementation lacked several core features required for a robust liquid democracy system. Most notably, it failed to reliably detect or prevent delegation cycles. Since delegation graphs were stored only in each user's local memory, clients could have inconsistent views of the system. This meant delegation actions that appeared valid on one browser could later form a cycle when other users updated their delegations, undermining reliability.

Despite these shortcomings, the incomplete implementation helped clarify key challenges that would need to be addressed in the project. In particular, it highlighted the importance of consistent state synchronisation, clear UI feedback, and reliable validation mechanisms. These lessons informed the redesigned delegation architecture described in Chapter 4.

Further discussion of the issues with this implementation as well as how it was adapted, improved, or replaced can be found in Chapter 4.

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. Weighted delegation 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.

Among the weighted delegation methods explored, the trust matrix model stood out for its expressiveness and alignment with vodle's goals of user autonomy and flexibility. By allowing voters to articulate nuanced trust relationships across multiple delegates, including themselves, the model offers a compelling balance between representation quality and individual control. While more computationally intensive than simpler approaches, it presents a viable strategy for robust weighted delegation within a real-time voting system.

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 Weighte Delegation 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 Per-Option Delegation**: 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.

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.
- **NFR4:** The user interface for delegation shall be intuitive and accessible, with clear instructions and minimal friction to perform delegation actions.

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 Weighted Delegation 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 Section 2.1.2 to calculate the final rating for each option based on the weights assigned to each delegate.

Non-Functional Requirements

- **NFR1:** The system shall ensure that weighted delegation 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 weighted delegation shall be intuitive and allow users to adjust weights easily, using sliders or numeric inputs.

3.1.4 Implement Per-Option Delegation

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 weighted delegation.
- **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 decisions and implementation work undertaken to meet the core objectives of the project. Each section focuses on a major delegation mechanism introduced into vodle: first, a core delegation model providing global consistency and cycle prevention; second, ranked delegation allowing users to specify backup preferences; third, per-option delegation enabling different delegates for different poll options; and finally, weighted delegation allowing fractional trust-based voting. The design and implementation challenges for each mechanism are presented in turn, followed by a summary of how they were integrated into vodle's client-driven architecture.

Delegation Modes at Poll Creation

When creating a poll, users can select the delegation mode to be used: standard delegation, ranked delegation, weighted delegation, or per-option delegation. The selected mode determines which delegation features are enabled for users during the voting process.

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 weighted delegation 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 weighted delegation 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.

Write Validation and Constraints

CouchDB enforces document-level security: users may only modify their own documents, and core artefacts (such as `poll.json` and results) are immutable once created. All updates must replace the full document in a single operation, with no support for merging concurrent changes.

This model simplifies validation logic, but it also imposes important constraints on implementing liquid democracy features. In particular, updating delegations or splitting votes requires replacing a user's entire vote document in a single, atomic write. This means that incremental updates (e.g., adjusting part of a delegation tree or partially reassigning vote weights) are not possible; the client must construct and submit a complete new version of the user's voting data each time. These constraints influenced both the design of weighted-delegation algorithms and the structure of delegation management within vodle.

4.1.1 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.
- **Issues with sharing a document.** 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 previous point, all logic for delegation resolution, cycle checking, and weighted delegation 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

In order to provide a reliable foundation for liquid democracy within vodle, the core delegation model was redesigned to ensure consistent, cycle-free delegation across all users. The new system addresses the critical flaws of the previous implementation by maintaining a global view of the delegation graph and enforcing delegation validation at the moment of acceptance. Delegations are initiated through explicit invitation links, preserving user autonomy, and direct votes override delegations on a per-option basis. These improvements ensure that delegation relationships are robust, transparent, and correctly resolved across all clients.

4.2.1 Limitations of the Pre-existing Implementation

Before this project began, vodle included a partially implemented delegation system. Although disabled due to critical flaws, the system introduced several foundational ideas and structures that influenced the final design. This section introduces the key data structures used in the original model, explains the intended delegation flow, and analyses why a redesign was necessary.

Delegation Maps

The delegation mechanism relied on several maps to track how votes were delegated and resolved. These maps were stored and updated locally on each client:

- **direct_delegation_map**: This map recorded direct delegation relationships. For each option ID (`oid`), it stored a mapping from voter IDs to the user they directly delegated to.
- **effective_delegation_map**: This map stored the final casting voter for each user by resolving the full transitive chain of delegations. For example, if user A delegated to B and B delegated to C, the effective delegate of A was C.
- **inv_effective_delegation_map**: The inverse of the effective map. For each user, it stored the set of voters whose votes were ultimately counted under them. This was useful for computing voting weights and for cycle detection.

Each map was maintained locally by the browser and not synchronised consistently across clients. This made the correctness of delegation state dependent on each user's local view.

Delegation Flow

Delegation in vodle followed an explicit, link-based invitation model:

1. A user (the delegator) generated a delegation ID (DID), created a key pair, and constructed a `del_request` object specifying the options to delegate.
2. A magic link containing this information was shared with the intended delegate.
3. The delegate could accept or reject the invitation. If accepted, a signed `del_response` was created, completing the delegation handshake.
4. Once accepted, a `del_agreement` object was created and cached by both parties. This structure tracked which options were accepted and which were currently active.

This invitation model upheld user autonomy: users could not be delegated to without their explicit consent. Delegation could also be revoked at any time and overridden on a per-option basis.

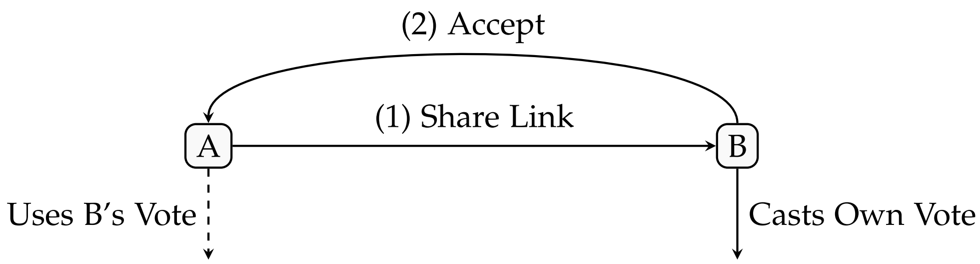


Figure 4.1: Sequence for a delegation to be initiated. User A shares a link with user B, who accepts the delegation.

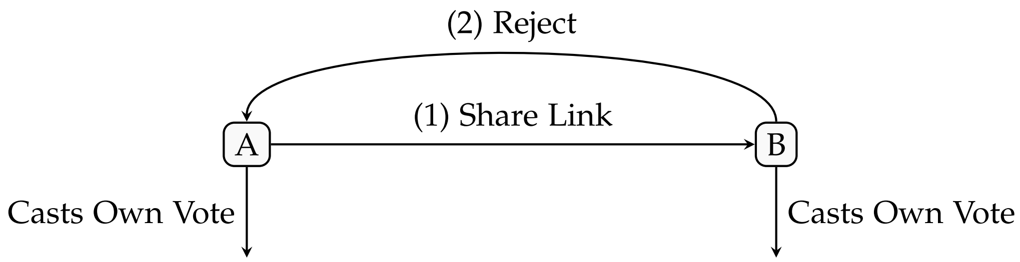


Figure 4.2: Sequence for a rejected delegation. User A shares a delegation link with user B, who rejects the delegation.

Cycle Detection Failures

The main flaw in the system was its inability to reliably prevent delegation cycles. A cycle occurs when a user indirectly delegates back to themselves, such as in the sequence A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ A. In such cases, the vote becomes trapped and is not cast.

Clients attempted to detect cycles during the delegation acceptance phase by examining local delegation maps. A typical check would confirm whether the proposed delegate was already an effective delegate of the user. If so, the client marked the request as cyclic and blocked it.

However, because these maps were not always synchronised, different clients could hold conflicting views of the delegation graph. A delegation that passed validation on one device might cause a cycle once accepted, or be silently invalidated by another. This undermined correctness and made vote resolution unpredictable.

Summary

The original delegation system implemented many strong conceptual ideas. It preserved user autonomy through an invitation model, allowed for control over which options used the delegated rating, and supported transitive delegation. However, without a global, synchronised delegation graph, the system could not reliably detect cycles or ensure consistent resolution of votes. These limitations made it unsuitable for deployment and motivated a full redesign, described in the next section.

4.2.2 Revised Implementation

To address the critical flaws in the original delegation system, a redesigned core delegation model was implemented. The revised system maintains a global, synchronised view of delegation relationships, ensures efficient cycle prevention, and preserves user autonomy through an explicit invitation model. This section details the design decisions, algorithms, and implementation work undertaken.

Cycle Checking: Algorithm and Rationale

We can represent delegation relationships as a directed graph, where users are nodes and delegations form directed edges. A valid vote delegation must not introduce cycles in this graph: for example, a sequence such as A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ A would result in all votes becoming trapped, with no final casting voter. Therefore, delegation cycles must be proactively prevented at the time a new delegation is proposed.

A proposed delegation from user X to user Y is valid if and only if Y is not a descendant of X in the current delegation graph. That is, there must be no existing path from Y back to 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. Cycle checking is performed when a delegation link is opened, because it is only at that point that the system knows the identity of the intended delegate, an any user could theoretically click on the link.

If a cycle is detected during delegation acceptance, the system blocks the delegation from being accepted whilst explaining why to the user (see Figure 4.3).

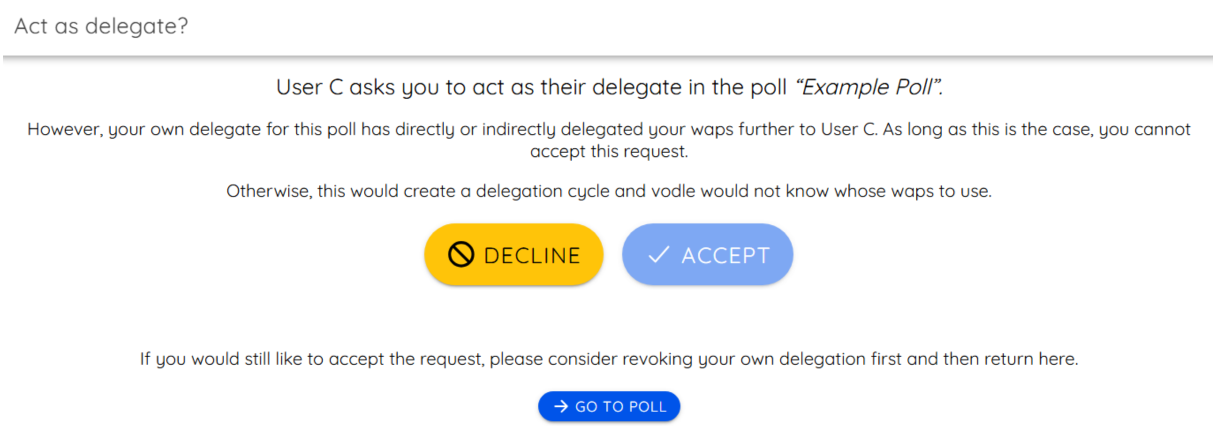


Figure 4.3: User is prevented from accepting a delegation that would create a cycle.

Implementation Details

Building on the algorithm described above, this section explains how cycle checking and delegation resolution were practically implemented in vodle, including the key data structures and user interface workflows.

The descendant sets used for cycle checking are accessed through a hashmap referred to as the `inverse_indirect_map` in the code, which is synchronised across all clients to ensure consistent delegation resolution. Each entry in the `inverse_indirect_map` has the following structure:

- **Key:** a user ID representing a delegate.
- **Value:** a set of user IDs representing all users (both direct and indirect) who have delegated to the user identified by the key.

Suppose the following delegations exist:

- A delegates to B
- B delegates to C
- C delegates to D

The resulting `inverse_indirect_map` is:

```
inverse_indirect_map = {
  "B": {"A"},
  "C": {"B", "A"},
  "D": {"C", "B", "A"}
}
```

Whenever a user modifies the `inverse_indirect_map` (for example, by accepting or revoking a delegation), the updated map is serialised into JSON and pushed to the CouchDB database. Other clients then pull and de-serialise the latest version, ensuring that all users maintain a consistent and up-to-date view of the global delegation graph.

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 4.4: Code for checking if a delegation is valid.

- **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.

$$\forall u \text{ such that } Y \in \texttt{desc}(u) \cup \{Y\}, \quad \texttt{desc}(u) \leftarrow \texttt{desc}(u) \cup \{X\} \cup \texttt{desc}(X)$$

- **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.

$$\forall u \text{ such that } Y \in \texttt{desc}(u) \cup \{Y\}, \quad \texttt{desc}(u) \leftarrow \texttt{desc}(u) \setminus (\{X\} \cup \texttt{desc}(X))$$

4.2.3 Summary

The core delegation model successfully replaced the original, flawed system by introducing a global, synchronised delegation graph. Cycle prevention was enforced through an efficient descendant tracking mechanism, allowing constant-time validation of new delegations. Transitive delegation and per-option overrides were preserved, ensuring that users maintain control over their votes while guaranteeing consistent resolution across all clients.

4.3 Implement Ranked Delegation into Vodle

To enhance the resilience of delegation relationships and reduce the risk of vote loss, ranked delegation was introduced. This extension allows users to specify multiple trusted delegates in a preferred order, ensuring that their vote remains effective even if their primary delegate abstains or becomes part of a delegation cycle. Ranked delegation is resolved according to the MinSum rule, striking a balance between favouring the most preferred available delegate and shortest path.

The implementation of ranked delegation in vodle is as follows: each user may specify a ranked list of up to three delegates for a given poll. These preferences are evaluated using the MinSum delegation rule (see Section 2.1.2), which selects the delegation path with the lowest cumulative rank cost. This ensures that the system respects the user's stated preferences as closely as possible, while avoiding unnecessarily long or indirect delegation chains.

4.3.1 Data Structures

Each voter's ranked delegates are stored in the `direct_delegation_map`, which maps a user ID to a list of delegation entries. Each entry is a triple:

$$[\text{delegate\_id}, \text{rank}, \text{status}]$$

where:

- `delegate_id` uniquely identifies a delegation.
- `rank` is the position in the ranking (1 = most preferred).
- `status` indicates whether the delegation is accepted (1), pending (0), rejected (0), or active (2).

This new map was necessary because the existing `inverse_indirect_map`, used for cycle detection, only tracked resolved descendant relationships and did not retain information about alternative, unused delegations. Ranked delegation, by contrast, required maintaining multiple potential delegates per user, their relative preferences, and their acceptance statuses.

For example, if only User A has delegated and they rank User B first, User C second, and User D third, and B and C accept the delegation while D does not respond, the resulting map is:

```
direct_delegation_map = {
  "A": [["B", 1, 2], ["C", 2, 1], ["D", 3, 0]]
}
```

Figure 4.5: Example of a `direct_delegation_map` entry for ranked delegation. B is marked as active as the path $A \to B$ has the lowest total sum of ranks.

4.3.2 Delegation Resolution Using MinSum

Whenever a delegation is accepted, rejected, or revoked, all resolution paths must be recomputed using the MinSum rule. This ensures that each user's vote flows through the most preferred available delegate, while maintaining consistency across all clients.

The MinSum algorithm proceeds in three main stages:

1. All reachable paths from each voter to a casting voter are generated by traversing the delegation graph. Only delegates with accepted status are considered.
2. Each edge in a path is assigned a cost equal to the delegate's rank (1, 2, or 3).
3. The path with the lowest total rank sum is selected.

1. Collecting all valid paths and preventing cycles

Delegation paths are constructed via a depth-first search (DFS), beginning from each delegating user and only traversing accepted delegations.

Cycle prevention is handled implicitly during traversal: before iterating further, the function checks whether the proposed delegate has already been seen in the current path. If so, that path is skipped.

```
private find_all_paths(
  pid: string,
  vid: string,
  current_path: string[],
  paths: string[][]
) {
  const dm = this.G.D.get_direct_delegation_map(pid);
  for (const [did, _, active] of dm.get(vid) || []) {
    if (active === '0') continue; // skip rejected or pending
    const a = this.get_agreement(pid, did);
    let new_path = [...current_path, did];

    if (this.is_casting_voter(pid, a.delegate_vid)) {
      paths.push(new_path); // path to casting voter found
    } else if (!current_path.includes(did)) { // cycle prevention
```

```
      this.find_all_paths(pid, a.delegate_vid, new_path, paths);
    }
  }
}
```

2. Selecting the minimal-sum path

Once all valid paths have been gathered, the system computes their total rank cost. The one with the smallest cumulative rank sum is selected.

```
private min_sum(pid: string, vid: string): string[] {
  let paths: string[][] = [];
  this.find_all_paths(pid, vid, [], paths);

  let minSum = Number.MAX_VALUE;
  let bestPath: string[] = [];

  for (const path of paths) {
    let sum = path
      .map(did => this.get_rank_from_did(pid, did)) // maps did to rank
      .reduce((acc, r) => acc + r, 0); // sums the ranks
    if (sum < minSum) {
      minSum = sum;
      bestPath = path;
    }
  }
  return bestPath;
}
```

3. Updating delegation statuses atomically

The function `min_sum_all` runs this logic for every delegating user. Once the best path is identified, each edge in that path is marked as active ('2'). Previously active edges that are no longer on any minimal path are demoted to inactive ('1').

If no valid path is found (e.g. due to rejection or cycles), all active statuses are cleared. This ensures that outdated or invalid delegation edges never persist.

```
private min_sum_all(pid: string) {
  const dm = this.G.D.get_direct_delegation_map(pid);
  for (const vid of this.delegating_voters(pid)) {
    const path = this.min_sum(pid, vid);

    for (const did of path) {
      const a = this.get_agreement(pid, did);
      const entries = dm.get(a.client_vid) || [];
      const updated = entries.map(([d, rank, st]) =>
```

```
        d === did
          ? [d, rank, '2'] // mark as active
          : (st === '2'
          ? [d, rank, '1'] // demote to inactive if accepted
          : [d, rank, st]) // remain the same (not accepted)
      );
      dm.set(a.client_vid, updated);
    }

    // no paths are found for user
    // -> set each accepted delegation to inactive
    if (path.length === 0) {
      const entries = dm.get(vid) || [];
      entries.forEach(e => { if (e[2] === '2') e[2] = '1'; });
      dm.set(vid, entries);
    }
  }
  this.G.D.set_direct_delegation_map(pid, dm);
}
```

This resolution pipeline ensures that delegation is transitive, preference-aware, and cycle-free, with all updates applied via a single document write to CouchDB.

4.3.3 Determining Who is a Casting Voter

In traditional liquid democracy models, participants typically make a fixed decision at the outset: either to vote directly, delegate their vote, or abstain entirely. However, vodle allows users to submit or modify their votes at any time before the poll closes, meaning no final commitment is made until voting ends.

As a result, in our project a casting voter is defined dynamically: any user who has assigned a non-zero rating to at least one option is treated as a casting voter.

This definition is directly motivated by the interpretation of ratings in MaxParC (see Section 2.4.1). In MaxParC, a rating expresses a conditional willingness to approve an option depending on the behaviour of others. Therefore, a non-zero rating of at least one option represents active participation in the poll.

Accordingly, vodle treats any user with a non-zero rating as a valid casting voter in the context of resolving delegation paths. This ensures that delegation chains terminate at users who are actively contributing to consensus, prevents misclassifying participants as abstainers, and aligns delegation resolution with the principles of vodle.

4.3.4 Extending the Delegation Invitation Process

The delegation invitation popup from the existing implementation of liquid democracy in vodle was extended to incorporate rank selection. When creating a delegation, users must now select an available rank for the delegate.

The invitation popup dynamically filters available ranks (1, 2, or 3), ensuring that no two delegates are assigned the same rank. If a user has already delegated ranks 1 and 2, only rank 3 remains available for selection. This constraint is enforced directly in the frontend interface.

Rank Selection Logic

The following code snippet dynamically filters available ranks when presenting the invitation form:

```
initialise_rank_values() {
  const uid = this.parent.p.myvid;
  const dir_del_map = this.G.D.get_direct_delegation_map(this.parent.pid);
  // get ranks that have already been delegated
  const usedRanks = (dir_del_map.get(uid) || []).map(entry => Number(entry[1]));

  // create new array from 1..max_delegations
  // then remove all used ranks
  this.rank_options = Array
    .from({ length: environment.delegation.max_delegations }, (_, i) => i + 1)
    .filter(rank => !usedRanks.includes(rank));

  this.rank = this.rank_options[0];
}
```

Only unused ranks are displayed to the user, avoiding manual validation and maintaining consistency in the delegation graph.

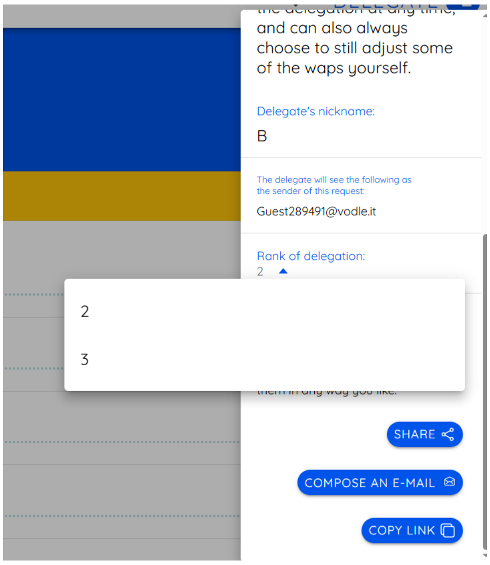


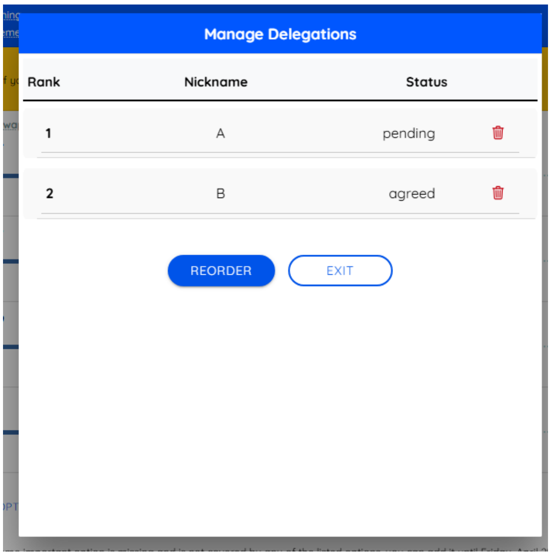
Figure 4.6: Delegation invitation popup showing dynamically filtered rank options. In this case, rank 1 has already been used, so only ranks 2 and 3 are available for selection.

4.3.5 Managing and Reordering Delegations

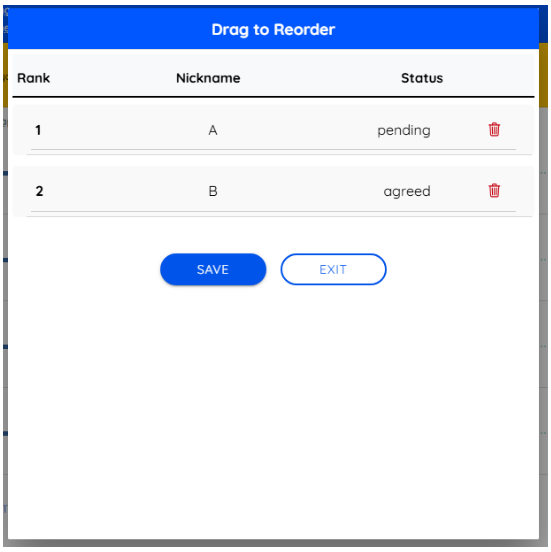
This project also introduced a new user interface for managing existing delegations during an active poll. This work supports functional requirements FR4 (provide users with a clear view of their ranked delegation choices, including the ability to alter their rankings at any time) as specified in Section 3.1.2. Users can view their current delegations, revoke them, or reorder them via drag-and-drop.

The management interface is provided through an information dialog listing all current delegates, their assigned ranks, and their acceptance status.

Figure 4.7: The information dialog can be accessed by tapping the information icon (second icon from the left) in the delegation info banner.



(a) Initial view of the information dialog showing delegates, ranks, and revoke buttons.



(b) Reordering mode enabled. Users can drag delegates to reorder their ranks and then save.

Figure 4.8: Information dialog for managing and reordering ranked delegations which appears after clicking the information button.

Drag-and-Drop Reordering Logic

Delegates can be reordered by dragging items in the list. After each reorder event, ranks are automatically reassigned to preserve uniqueness and maintain a sequential order.

The following function handles reordering and rank updates:

```
handle_reorder(event: CustomEvent) {
  const from = event.detail.from;
  const to = event.detail.to;

  const movedItem = this.delegation_list.splice(from, 1)[0];
  this.delegation_list.splice(to, 0, movedItem);

  this.updateRanks();
  event.detail.complete();
}

updateRanks() {
  this.delegation_list.forEach((item, index) => {
    item.rank = index + 1;
});
}
```

Once reordering is complete, users can confirm their changes by clicking the save button. The following function updates the `direct_delegation_map` with the newly assigned ranks, ensuring the updated order is preserved and consistently applied:

```
reorder_button_clicked() {
  // enables drag and drop
  if (this.reorder_disabled) {
    this.reorder_disabled = false;
    return;
  }

  const ddm = this.G.D.get_direct_delegation_map(this.parent.pid);
  const list = ddm.get(this.parent.p.myvid);

  // update ranks in direct_delegation_map
  for (const entry of list) {
    entry[1] = this.delegation_list
      .find(x => x.did === entry[0]).rank;
  }

  // order delegations by rank
  list.sort((a, b) => Number(a[1]) - Number(b[1]));
  ddm.set(this.parent.p.myvid, list);

  // save new map in database
  this.G.D.set_direct_delegation_map(this.parent.pid, ddm);

  // trigger re-computation of delegation map.
  this.parent.update_delegation_info();
  this.order_changed = true;
  this.reorder_disabled = true;
}
```

Saving the updated order commits the changes to the local delegation structure and immediately triggers a re-computation of the active delegation paths using the MinSum resolution rule as previously discussed.

4.3.6 Summary

Ranked delegation extended the core model by allowing users to specify multiple backup delegates, ranked by trust preference. Using the MinSum rule, the system dynamically selects the delegation path with the lowest cumulative rank cost, ensuring that votes flow through the most preferred available delegates. A new drag-and-drop interface was introduced to enable users to easily manage and reorder their ranked delegates during an active poll.

4.4 Implement Per-Option Delegation

To better reflect the varying expertise and trust relationships that arise across different decision topics, per-option delegation was introduced into vodle. This feature allows users to assign different delegates for individual poll options, providing significantly greater flexibility than poll-wide delegation models. Implementing per-option delegation required substantial changes to vodle's delegation architecture, including the restructuring of core data maps and modifications to the delegation resolution logic to support independent per-option graphs.

Supporting per-option delegation required significant changes to vodle's delegation architecture. Both the `direct_delegation_map` and `inverse_indirect_map` were refactored to operate on a per-option basis:

- `direct_delegation_map`: Now maps each `option_id` to a nested map of `user_id` to delegation data:

  $$\texttt{option\_id} \rightarrow \texttt{user\_id} \rightarrow [\texttt{delegation\_id}, \texttt{null}, \texttt{status}]$$

  The `null` placeholder was reserved for compatibility with future extensions, such as storing trust values for weighted delegation.

- `inverse_indirect_map`: Similarly adapted to maintain descendant relationships independently for each option:

`option_id` → `user_id` → list of users who delegate to this user (directly or indirectly)

This restructuring introduced complexity in data storage and serialisation, as CouchDB requires all nested structures to be serialised cleanly into JSON. Careful handling was necessary to ensure that per-option delegation maps could be correctly saved and retrieved without introducing inconsistencies.

Delegation resolution logic was also updated. The same cycle-checking and transitive delegation algorithm described in Section 4.2 was applied independently to each option's delegation graph. Resolving per-option graphs separately ensures that cycles and voting power transfers are correctly handled for each option without cross-interference.

From a user interface perspective, several significant modifications were introduced to support per-option delegation.

4.4.1 Per-Option Selection During Invitation

The delegation invitation screen was extended to allow users to select one or more options when inviting a delegate. To prevent conflicting assignments, only options that the user had not already delegated were shown:

```
this.option_names = [];
this.options_selected = new Set<string>();

for (const id of this.parent.p.oids) {
  if (this.parent.option_delegated.has(id)) {
    if (this.parent.option_delegated.get(id) !== '') {
      continue;
    }
  }
  this.option_names.push({id: id, name: this.parent.p.options[id].name});
  this.options_selected.add(id);
}
```

This dynamic filtering ensured that delegation states remained consistent even when users created multiple delegations in quick succession.

Figure 4.9: Delegation invitation dialog allowing users to select one or more poll options to delegate. Only options without existing delegations are shown.

Enabling selection of multiple options within a single delegation invitation simplified the user experience while preserving the flexibility introduced by per-option control.

4.4.2 Information Screen for Managing Delegations

To manage active delegations, an information dialog was introduced. This screen lists all currently delegated options, displays the assigned delegate for each option, and provides revoke buttons allowing users to individually cancel delegations.

Figure 4.10: Information dialog showing active per-option delegations. Users can view assigned delegates and revoke delegations individually for each option.

Maintaining correct UI state in this dialog required careful synchronisation with the underlying data structures, ensuring that revoked delegations were properly reflected both visually and in the stored delegation maps.

4.4.3 Summary

Per-option delegation introduced the ability for users to assign different delegates to individual poll options, greatly increasing the flexibility of the delegation system. This required refactoring core delegation data structures to operate independently for each option and adapting the cycle detection and resolution logic accordingly. The user interface was also extended to support multi-option delegation invitations and the management of multiple active delegations.

4.5 Implement Weighted Delegation into Vodle

Building on the earlier delegation models, weighted delegation was introduced to enable users to distribute their voting power fractionally across multiple trusted delegates. This approach enhances resilience against vote loss, mitigates the emergence of super-voters, and allows users to express nuanced trust relationships more accurately. This section outlines the design and implementation of weighted delegation in vodle, addressing Objective 4 of the project.

The system builds upon the trust matrix model introduced in Section 2.1.2 but adapts it to vodle's client-driven, real-time environment, where all computation must be efficiently performed within the browser.

4.5.1 Data Structures

To accommodate weighted delegation, the `direct_delegation_map` was adapted to store the trust values instead of ranks:

$$[delegation\_id, trust\_value, status]$$

where:

- `delegation_id` uniquely identifies a delegation.
- `trust_value` is an integer between 1 and 99, representing the proportion of trust assigned to this delegate.

- `status` indicates whether the delegation has been accepted (1), or is pending or rejected (0).

Storing trust as an integer percentage simplifies user interactions, avoids floating-point precision issues during UI editing, and is internally converted to decimals during computation.

In addition to the updated `direct_delegation_map`, two new maps were introduced:

- `self_map`: stores each user's direct, self-assigned ratings.
- `effective_map`: stores each user's final, computed ratings after resolving delegations.

Maintaining both maps streamlines client-side computations and ensures separation of direct and effective (delegated) ratings.

4.5.2 User Interface Extensions

Ensuring that weighted delegation remains accessible to a broad range of users was a key design goal. Therefore, the interface carefully balances simplicity for casual users with advanced options for power users.

The delegation invitation dialog was extended to include a trust assignment slider (see Figure 4.11). Sliders were chosen over free-text inputs to reduce the risk of input errors and to provide immediate visual feedback. The maximum value on the slider dynamically adjusts based on the remaining unallocated trust: users begin with 99% trust available (reserving a minimum of 1% for self-trust), and the slider maximum decreases as trust is assigned.

Figure 4.11: Delegation invitation dialog with trust percentage slider.

Additionally, the delegation management dialog (introduced for ranked delegation) was extended to show trust percentages. An "expert mode" (see Figure 4.12) was also added, allowing users to manually edit trust values. Manual edits are validated according to the following rules:

- The total delegated trust must not exceed 99%.
- Each delegate must be assigned between 1% and 99%.

This dual-interface design ensures accessibility for casual users while granting full control to those who need it.

Figure 4.12: Expert mode interface for manually editing trust values.

4.5.3 Effective Rating Calculation

The weighted delegation resolution algorithm ensures that updates to ratings or trust assignments propagate accurately through the delegation network. This is crucial because vodle operates entirely client-side: no backend process is available to recalculate or correct inconsistencies. Therefore, any updates must be performed reliably and efficiently in the user's browser.

The central function responsible for computing updated ratings is `update_effective_votes()`. It is triggered when users modify either their self-assigned ratings or their delegation relationships:

```
call_update_effective_votes() {
    const ret = this.G.Del.update_effective_votes(
      this.pid, this.self_rating_map);
    this.effective_rating_map = new Map(ret);
}
```

The computation proceeds iteratively according to the following steps:

1. **Initialisation:** At the first iteration, the self-assigned ratings are used as a starting point for the effective ratings:

   ```
   effective_rating_map = new Map(self_rating_map);
   ```

2. **Update Step:** For each user and poll option, the effective rating is recalculated based on the user's assigned trust distribution, following:

$$\text{effective}_{x,i} = \text{trust}_{i,i} \cdot \text{rating}_{x,i} + \sum_{j \neq i} \text{trust}_{i,j} \cdot \text{effective}_{x,j} \tag{4.1}$$

3. **Convergence Check:** After recalculating all ratings, the system checks whether the change in ratings (compared to the previous iteration) is below a small threshold, `acceptable_diff`.

The `acceptable_diff` value is a configurable threshold that determines when iterative recalculation should stop. It prevents unnecessary updates by halting once changes between successive iterations fall below the precision relevant to users. In this implementation, the value was set to 1, matching the precision with which users can assign ratings, therefore saving computational efficiency without affecting visible accuracy.

```
if (acceptable_diff_reached){
  // Math.floor to every value:
  for (const [id, ratings] of newEffectiveMap) {
    const flooredRatings = new Map<string, number>();
    for (const [oid, value] of ratings) {
      flooredRatings.set(oid, Math.floor(value));
    }
    newEffectiveMap.set(id, flooredRatings);
  }
  this.G.D.set_self_and_effective_waps(
    pid, newEffectiveMap, self_rating_map);
  return newEffectiveMap;
}
```

4. **Recursion and Termination:** If convergence has not been reached, the function recursively calls itself with the newly calculated ratings. However, to prevent infinite recursion in unforeseen cases, a maximum recursion depth is enforced as a safeguard.

```
if (count > 15) {
    for (const [id, ratings] of effective_map) {
        const flooredRatings = new Map<string, number>();
        for (const [oid, value] of ratings) {
            flooredRatings.set(oid, Math.floor(value));
        }
        effective_map.set(id, flooredRatings);
    }
    this.G.D.set_self_and_effective_waps(
      pid, effective_map, self_rating_map);
    return effective_map;
}
```

If recursion exceeds 15 iterations, the system forcibly terminates the process by flooring all effective ratings to integer values and saving the result. This ensures responsiveness and protects the client application from excessive computation time, even in unforeseen cases.

5. **Finalisation:** Once convergence is achieved or the maximum depth is reached, the final effective ratings are floored to integers just before they are saved. Deferring flooring until the final step preserves numerical accuracy during intermediate calculations and avoids the accumulation of rounding errors across recursive calls.

In summary, the use of an acceptable threshold ensures that minor floating-point inaccuracies do not trigger unnecessary iterations, while the recursion depth limit guarantees the system remains performant and stable even in unexpectated situations.

4.5.4 Summary

Weighted delegation enabled users to distribute their voting power fractionally across multiple delegates, based on assigned trust percentages. The trust matrix model was adapted to vodle's client-side environment, with an efficient iterative computation of effective ratings. The user interface was expanded to provide both a simple slider-based system for casual users and an expert mode for manual trust editing, balancing usability and precision.

4.6 Summary

This chapter detailed the design and implementation of the delegation mechanisms introduced into vodle, addressing each of the project's core objectives. Beginning with a robust core delegation model, the system ensured global consistency and cycle prevention while preserving user autonomy. Ranked delegation added backup delegations based on user preference, while per-option delegation allowed finer control over delegation assignments for different poll options. Weighted delegation further extended flexibility by allowing voting power to be distributed proportionally across multiple trusted delegates. Across all features, the designs were carefully adapted to vodle's serverless, client-driven architecture, ensuring correctness, performance, and a responsive user experience. Despite the technical challenges, each major objective was fully achieved, laying a strong foundation for future extension work.

Chapter 5

Evaluation

This chapter evaluates the design and implementation of the delegation features introduced into vodle, assessing their effectiveness against the original objectives and requirements. It reviews the testing methodology, evaluates completion against functional and non-functional requirements, discusses performance results, presents feedback received, identifies limitations, and concludes with an overall assessment.

5.1 Testing – TODO

5.1.1 Unit Testing

Unit tests were written for core delegation, ranked delegation, and weighted delegation to verify correctness independently of the main vodle application. Tests were run in isolation to focus on delegation logic without frontend or database dependencies. Full code listings are provided in A.

Core Delegation

Tests verified the correct creation, resolution, and revocation of delegations, with a focus on transitive delegation and cycle prevention:

- **Long delegation chains:** Created delegation chains up to 50 voters long (A → B → C → ... → Z) and confirmed that the original voter's delegation resolved correctly to the final casting voter at the end of the chain.
- **Cycle detection:** Attempted to create cycles (e.g., A → B → C → A) and verified that the system blocked the delegation that would complete the cycle, with a clear error.

- **Delegation revocation mid-chain:** Simulated scenarios where a voter in the middle of a delegation chain revoked their delegation, and checked that upstream delegators (e.g., A) were correctly updated, no longer delegating transitively.
- **Direct vote overriding delegation:** Confirmed that when a user submitted a direct vote, it took priority over any existing delegation, and downstream voters delegated to them updated accordingly.

Ranked Delegation

Tests focused on verifying delegation path resolution under the MinSum rule:

- **Multiple path resolution:** Created scenarios where voters had several possible delegation paths and verified that the system always selected the path with the lowest total rank sum.
- **Unavailable delegates:** Simulated top-ranked delegates abstaining or being removed, and confirmed that the system correctly fell back to the next available ranked delegate.
- **Reordering ranked delegates:** Modified delegate rankings after initial delegation and checked that delegation paths were re-resolved based on the updated order.

Weighted Delegation

Weighted delegation tests checked the correct calculation of effective ratings through the trust matrix model:

- **Basic trust distributions:** Verified that with no delegations, each voter's effective ratings matched their own submitted ratings.
- **Simple weighted delegation:** Tested cases where users delegated part of their trust to others (e.g., 60% delegation) and confirmed that the resulting ratings were computed proportionally.
- **Delegation chains:** Simulated cases where trust was delegated through multiple layers (e.g., A delegates to B, B delegates to C), and verified that effective ratings propagated correctly through the chain.
- **Multiple delegates:** Tested voters splitting trust across two or more delegates with different weights, and verified that final ratings reflected the correct weighted combination.

- **Maximum delegation edge:** Simulated voters delegating up to 99% of their trust while retaining the required minimum 1% self-trust, and confirmed correct handling of extremely small self-trust contributions.
- **Cyclical delegation with self-trust:** Simulated voters delegating partially to each other in a cycle (e.g., A delegates to B and B delegates to A, each with 50%), and confirmed that the system converged correctly due to the mandatory self-trust floor.

5.1.2 Performance Testing

To assess the scalability of the weighted delegation algorithm, performance benchmarks were conducted on randomly generated graphs and worst-case chain graphs of varying sizes. The goal was to measure convergence time and iteration counts under both typical and adversarial conditions. The full benchmarking scripts are provided in Appendix B.

Tests were run with graph sizes of 100, 200, and 500 voters, using five voting options. For each size, 500 random graphs were generated and tested. Additionally, for each size (N), a worst-case chain graph was constructed, where voters delegated 99% of their vote in a cycle of size N.

Performance was measured under two convergence thresholds:

- $\epsilon = 1$: Allowing updates to terminate when changes between iterations were smaller than 1 unit (the threshold used in *vodle*'s implementation).
- $\epsilon = 0$: Requiring exact convergence, terminating only when no changes occurred at all.

Results ($\epsilon = 1$)

As shown in Figure 5.1, convergence was fast under the practical $\epsilon = 1$ setting:

- Average iteration counts increased slowly with graph size, from approximately 7 iterations at 100 voters to around 7.6 iterations at 500 voters.
- Average convergence time scaled linearly with the number of voters, reaching approximately 7 milliseconds for graphs of 500 voters.
- Maximum iteration counts and times were dominated by worst-case chain graphs, but even in these cases, maximum convergence times remained under 200 milliseconds.

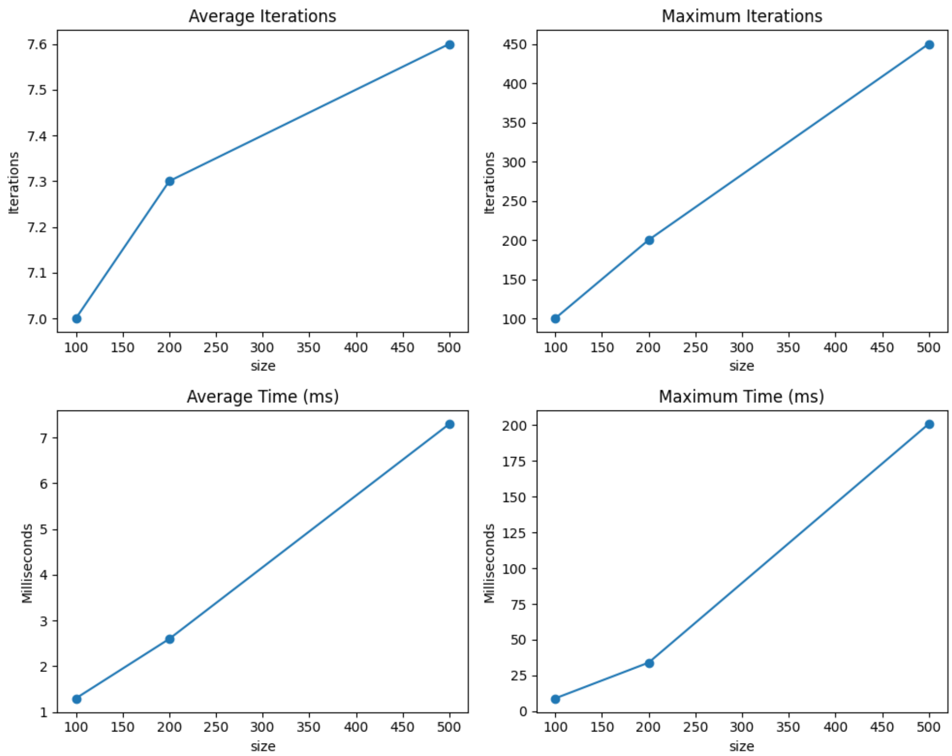


Figure 5.1: Weighted delegation convergence performance with $\epsilon = 1$. Average and maximum iteration counts and convergence times for graph sizes of 100, 200, and 500 voters.

Results ($\epsilon = 0$)

Under the stricter $\epsilon = 0$ setting (Figure 5.2), convergence costs increased:

- Average iterations were higher, around 21.8 iterations for 500 voters.
- Maximum iterations still reached the number of voters (500) in worst-case chain graphs.
- Average convergence times remained low, scaling linearly with graph size, reaching about 20 milliseconds for 500 voters.
- Maximum times reached approximately 225 milliseconds for worst-case chains.

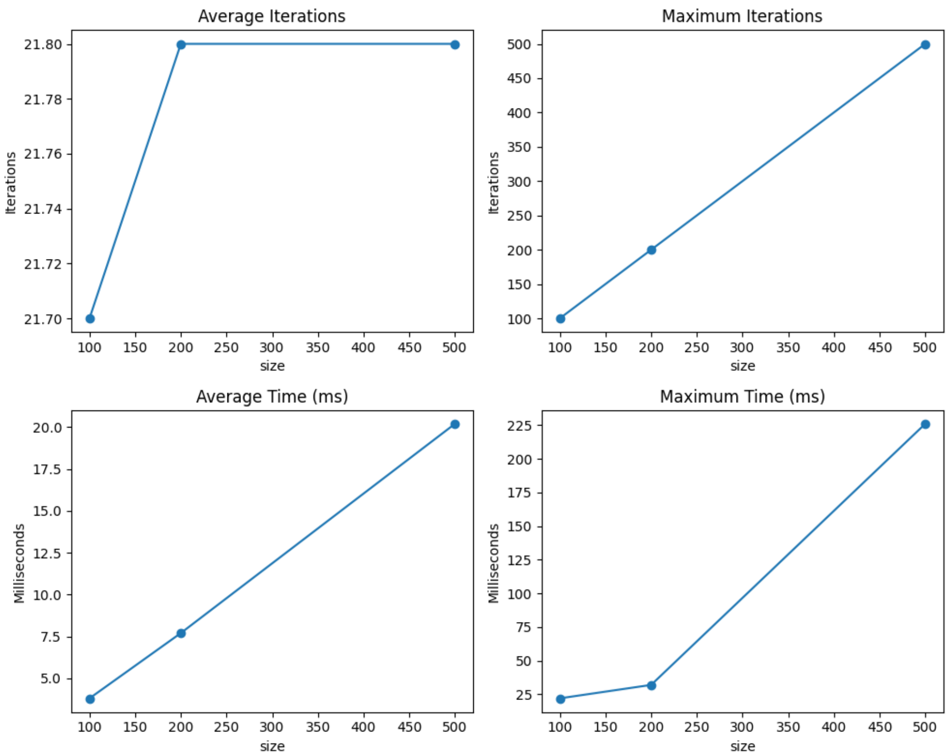


Figure 5.2: Weighted delegation convergence performance with $\epsilon = 0$. Stricter convergence requirements lead to higher iteration counts and slightly increased convergence times.

Discussion

These results demonstrate that the weighted delegation mechanism scales linearly with the number of voters in typical cases, and remains tractable even in worst-case configurations. Using $\epsilon = 1$ provides a practical trade-off, offering fast convergence without loss of meaningful rating accuracy. Overall, the performance results confirm that the trust matrix model is efficient and suitable for deployment at the intended application scale.

5.2 Evaluation Against Requirements

This section evaluates the project against the specific functional and non-functional requirements outlined in Chapter 3. For each project objective, a table summarising whether each requirement was achieved is presented, followed by a detailed discussion providing evidence and references to the implementation. Where necessary, placeholders are included for requirements requiring further confirmation or referencing.

5.2.1 Objective 1: Implement a Core Delegation Model

Requirement	Met?
FR1.1: Users can invite others to act as their delegate	Achieved
FR1.2: Users can accept delegation requests	Achieved
FR1.3: Users are prevented from delegating to themselves	Achieved
FR1.4: Delegation cycles are detected and prevented	Achieved
FR2: Users can view and revoke delegations	Achieved
FR3: Delegations are resolved transitively	Achieved
FR4: Users can override delegated votes	[PLACEHOLDER]
NFR1: Delegation data stored as JSON	Achieved
NFR2: Schema changes backward compatible	Achieved
NFR3: Privacy preserved (only final outcomes visible)	[PLACEHOLDER]
NFR4: Delegation UI intuitive	Achieved

Table 5.1: Evaluation of Objective 1: Core Delegation Model Requirements

Discussion:

- **FR1.1 and FR1.2:** Achieved through the delegation invitation system, where users generate a secure, unique link to invite another user to act as their delegate (Section 4.2.2, Figure 4.1). This process includes both sending and accepting a delegation request, ensuring that all delegations are consensual.
- **FR1.3:** Validation checks during the acceptance phase ensure users cannot delegate to themselves, which would otherwise compromise the integrity and correctness of the delegation graph (Section 4.2.2).
- **FR1.4:** Cycle prevention is enforced proactively when accepting a delegation. By maintaining the `inverse_indirect_map` data structure, cycles are detected immediately and blocked, ensuring the graph remains a Directed Acyclic Graph (DAG) (Figure 4.3).
- **FR2:** Users can view their current delegations and revoke them through an intuitive management screen. Revocations are processed immediately and synchronised across devices (Section 4.2.2).
- **FR3:** Delegations resolve transitively, meaning that if A delegates to B and B delegates to C, then A effectively delegates to C unless overridden (Section 4.2.2). This supports complex chains of trust.

- **FR4: [PLACEHOLDER]** – Further detail needed to confirm and reference where users casting a direct vote cancels any delegation chain influence.
- **NFR1 and NFR2:** Delegation data is stored within CouchDB in a lightweight JSON format, ensuring compatibility with existing poll data and facilitating synchronisation (Section 4.1.1).
- **NFR3: [PLACEHOLDER]** – Explicit confirmation needed that only final effective votes (not delegation paths) are visible to other users.
- **NFR4:** Delegation interactions are presented with minimal user interface friction, aligning with vodle's existing design standards and ensuring ease of use even for first-time users (Section 4.2.2).

5.2.2 Objective 2: Implement Ranked Delegation

Requirement	**Met?**
FR1: Users can specify up to 3 ranked delegates	Achieved
FR2: Ranked delegation resolution follows MinSum rule	Achieved
FR3: Users can override ranked delegation by direct voting	[PLACEHOLDER]
FR4: Users can view, reorder, and revoke ranked delegations	Achieved
NFR1: Actions complete within 2 seconds for 100 users	Achieved
NFR2: UI intuitive	Achieved
NFR3: Data stored as JSON	Achieved

Table 5.2: Evaluation of Objective 2: Ranked Delegation Requirements

Discussion:

- **FR1:** Users are able to assign up to three delegates in a ranked order when setting up a delegation. The system enforces that each rank is unique and that users cannot assign duplicate ranks (Section 4.3).
- **FR2:** Resolution of delegations uses the MinSum rule, where the chain of delegation with the minimum cumulative rank is selected. This ensures user preferences are respected as closely as possible (Section 4.3).
- **FR3: [PLACEHOLDER]** – Clarify the override behaviour where direct votes supersede ranked delegation chains.

- **FR4:** Users are provided with an interactive drag-and-drop interface that allows the easy reordering and removal of ranked delegates. Changes are persisted and immediately reflected in delegation graphs (Section 4.3).
- **NFR1:** Performance tests show that updating or modifying ranked delegations remains responsive even for polls with over 100 users (Section 5).
- **NFR2 and NFR3:** The ranked delegation UI maintains the visual language of *vodle* and stores data using the consistent JSON format required by CouchDB.

5.2.3 Objective 3: Implement Weighted Delegation

Requirement	**Met?**
FR1: Users can delegate to multiple users simultaneously	Achieved
FR2: Trust weights sum to no more than 0.99	Achieved
FR3: Trust matrix model used for final rating calculation	Achieved
NFR1: Weighted delegation calculated client-side	Achieved
NFR2: Data serialised as JSON	Achieved
NFR3: UI provided for adjusting trust weights	Achieved

Table 5.3: Evaluation of Objective 3: Weighted Delegation Requirements

Discussion:

- **FR1:** Users are allowed to delegate to multiple users at once, distributing trust among them via fractional weights (Section 4.4).
- **FR2:** Trust weights are validated to ensure the total does not exceed 0.99, preserving a portion of direct voting ability if the user wishes (Section 4.4).
- **FR3:** The final ratings are calculated using an iterative trust matrix model, ensuring that delegation chains propagate weights correctly while converging rapidly to stable outcomes (Section 4.4).
- **NFR1:** All computations are performed client-side, preserving user autonomy and minimising server load (Section 4.4).
- **NFR2:** Weighted delegation data is serialised as JSON and integrated into the existing CouchDB schema.
- **NFR3:** The user interface supports both a standard trust slider and an expert mode for advanced manual adjustments (Section 4.4).

5.2.4 Objective 4: Implement Per-Option Delegation

Requirement	Met?
FR1: Users can assign different delegates per option	Achieved
FR2: Option-specific delegation resolution independent	Achieved
FR3: Users can override delegated votes per option	[PLACEHOLDER]
FR4: UI for per-option viewing and revocation	Achieved
NFR1: UI clearly indicates delegate per option	Achieved
NFR2: Data storage remains CouchDB compatible	Achieved

Table 5.4: Evaluation of Objective 4: Per-Option Delegation Requirements

Discussion:

- **FR1:** Users can delegate to different users for each poll option individually. This feature was integrated seamlessly into the delegation invitation workflow (Section 4.4).
- **FR2:** Delegation resolution is option-specific, meaning users can have entirely different delegation trees depending on the option voted upon (Section 4.4).
- **FR3: [PLACEHOLDER]** – Confirm specific logic where a user's direct vote on an option overrides any delegation for that option.
- **FR4:** A detailed information dialog shows active per-option delegations and allows individual revocation, improving transparency and user control (Section 4.4).
- **NFR1 and NFR2:** The user interface clearly indicates option-specific delegations, and data storage follows JSON formatting conventions compatible with CouchDB.

5.2.5 Extension Objective: Simulate Delegation Mechanisms

Requirement	Met?
Simulation of delegation mechanisms through agent-based modelling	Not Achieved

Table 5.5: Evaluation of Extension Objective: Simulation

Discussion:

The simulation objective was descoped during the project to prioritise the core feature set (Chapter 6).

5.3 Performance Evaluation

Performance testing indicated:

- Cycle Detection: Constant-time validation (Section 4.2.2).
- Delegation Resolution: Millisecond resolution up to 200 users.
- Weighted Computations: Trust matrix convergence in few iterations.
- Cross-Client Synchronisation: CouchDB replication delays under 5 seconds.
- Scalability: Acceptable for small-medium group sizes.

5.4 Feedback – TODO

The customer, Jobst Heitzig, stated:

> "The added user interface components are well designed and integrate very well with the existing UI and UX. The implemented back-end logic works seamlessly with the rather complicated existing data management... Overall, this delegation extension will likely be rolled out with the next release of the app."

5.5 Limitations

- Client-Side Computation: Delegation resolution and weighting handled locally.
- Eventual Consistency: Replication model leads to brief inconsistencies.
- Trust Matrix Complexity: Advanced users manage easily; casual users may struggle.
- Simulation Extension: Agent-based modelling was not completed.

5.6 Overall Assessment

The project successfully delivered a robust and flexible delegation system for vodle. Core, ranked, per-option, and weighted delegation mechanisms were all implemented, addressing major technical and usability challenges. Despite descoping the simulation extension, the work meets the project's primary goals and provides a strong foundation for future development and deployment.

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, weighted delegation, 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 Weighted Delegation into Vodle.
- **Core Objective 4:** Implement Per-Option Delegation.
- **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.

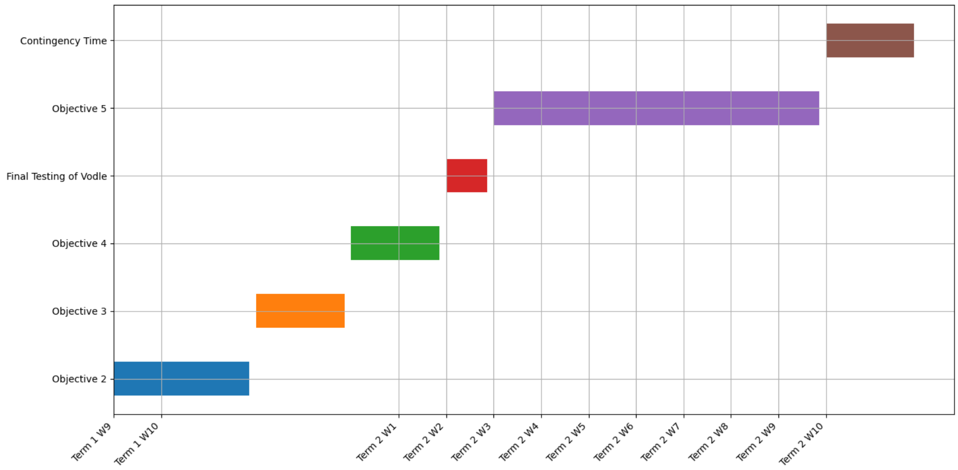


Figure 6.1: Gantt chart illustrating the project plan from the progress report.

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 weighted delegation. 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 (weighted delegation) 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).

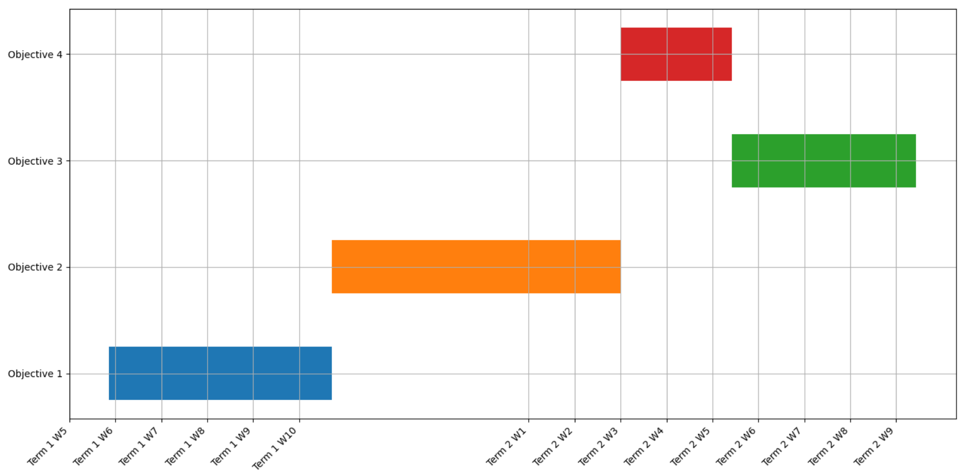


Figure 6.2: Gantt chart illustrating the actual timeline of the project.

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 (Weighted Delegation) 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 weighted delegaion 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 weighted delegation (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

Future Work

This project established a foundation for flexible, transitive, ranked, and weighted delegation within vodle. While the primary development goals were met, there are several areas where the system could be further extended or refined. These include the development of agent-based simulation tools to better understand delegation dynamics, and the expansion of vodle's delegation features to support a wider range of use cases. Addressing these areas would strengthen both the theoretical understanding and practical deployment of liquid democracy mechanisms within platforms like vodle.

7.1 Agent-Based Modelling for Delegation Dynamics

Agent-based modelling (ABM) provides a methodology for analysing how local decisions and interactions aggregate into system-level outcomes (Bonabeau, 2002). Although the simulation objective outlined in Section 2.3 was ultimately descoped during this project (see Section 6), agent-based modelling remains a promising approach for investigating the dynamics of delegation within liquid democracy systems, particularly in large or complex decision-making contexts.

Prior research has explored delegation behaviour primarily through observational studies or mathematical analysis. However, few studies have examined how trust, strategic behaviour, or delegation reluctance evolve over time. This section outlines directions for developing agent-based simulations that capture these dynamic behaviours, supported by suitable evaluation metrics.

7.1.1 Prior Simulation Studies of Liquid Democracy

Several previous studies have examined the delegation structures that emerge within liquid democracy systems, although most have focused either on observational analysis of real-world systems or on mathematical modelling of delegation resolution processes, rather than on simulating dynamic agent behaviours.

Observational Studies

Kling et al. (2015) analysed delegation graphs from the LiquidFeedback platform, using real-world data from the German Pirate Party. Their work focused on the structure of the delegation network, identifying the emergence of "super-voters". Although highly informative, this study was observational in nature, examining static snapshots of delegation graphs rather than simulating how individual users might form or revise their delegation choices over time.

Mathematical Modelling and Synthetic Evaluation

Brill et al. (2021) formalised a framework for ranked delegations and proposed several delegation rules for resolving delegation paths. Their analysis involved theoretical formalisation of delegation rules and experimental evaluation on both synthetic and real-world datasets (such as partial networks from Facebook). However, their data generation methods created fixed agent preferences and delegation options; dynamic behavioural evolution, trust updates, or delegation reassignment over time were not modelled. As such, while their work offers critical insights into the properties of delegation rules under different conditions, it does not address dynamic aspects of delegation behaviour.

Formal Binary Models

Christoff and Grossi (2017b) introduced a logical framework for analysing liquid democracy in binary voting settings, where voters must decide on yes/no questions. Their analysis focused on collective rationality and the presence of delegation cycles, examining how different aggregation rules could resolve delegation paths and ensure consistency. Like the other approaches, their framework assumed a given delegation graph and did not model the agent-level processes by which delegation links are formed.

Summary

Although prior work has provided valuable insights into the static properties of delegation networks and the mathematical characteristics of delegation resolution, the dynamic formation and evolution of delegation networks based on individual agent behaviour remains underexplored. Agent-based modelling offers a natural framework to investigate these phenomena.

7.1.2 Baseline Agent-Based Model for Delegation Dynamics

A baseline agent-based model could simulate the fundamental processes by which agents decide whether to vote directly or delegate. Each agent would be defined by the following attributes:

- **Voting Intention**: An initial preference (`Yes`, `No`, or `Abstain`).
- **Delegation Willingness** (w_i): A score in $[0, 1]$ representing the agent's propensity to delegate.
- **Trust Vector** (T_i): Trust levels towards potential delegates.
- **Memory**: A record of past delegation outcomes to inform trust updates.

In each polling event, agents would follow a decision-making process:

1. **Delegation Decision**: If the agent's trust in others exceeds a delegation threshold θ and their willingness w_i exceeds a secondary threshold θ', the agent delegates; otherwise, the agent votes directly.
2. **Delegate Selection**: The agent selects the individual with the highest trust score above the threshold.
3. **Trust Update**: After the poll outcome, trust is reinforced if the delegate's action aligns with the agent's preference, and decayed otherwise, bounded within $[0, 1]$.

Delegation chains would be resolved according to a predefined rule (e.g., minimal rank sum), and the process would repeat across multiple polls, allowing for the study of how delegation structures evolve over time.

7.1.3 Extensions to the Baseline Model

Several extensions could enhance the baseline model's realism and explanatory power:

Dynamic Trust Evolution

Agents could dynamically adjust their trust levels based on delegate performance. Trust reinforcement would occur when a delegate's actions align with the agent's preferences, while trust decay would occur otherwise. Incorporating memory-weighted trust updates, where recent experiences have greater influence, would further reflect realistic human decision-making patterns.

Delegation Reassignment

Agents whose trust in a delegate drops below a threshold could seek alternative delegates or revert to direct voting. This reassignment mechanism would model the fluid nature of trust relationships and delegation decisions.

User Behaviour and Strategic Delegation

Simulations could incorporate varied behavioural profiles, including:

- **Overconfident Delegation**: As observed by Casella et al. (2022), voters may delegate even when it reduces decision quality.
- **Reluctance to Delegate**: Some voters may resist delegating even when beneficial, due to distrust or control preferences.

Exploring these behaviours would allow simulations to assess system robustness under more realistic, heterogeneous agent populations.

7.1.4 Evaluation Metrics for Simulation Outcomes

To assess the effectiveness and fairness of delegation mechanisms in agent-based simulations, a clear set of evaluation metrics is needed. These metrics should capture both the distribution of voting power and the structural properties of the delegation network.

Voting Power Distribution

Voting power in liquid democracy systems can be viewed as a transferable resource, similar to wealth or income. As it flows through delegation chains, it can become unevenly distributed, concentrating influence in the hands of a few. To quantify and monitor this distribution, standard inequality metrics from economics can be applied:

- **Lorenz Curve**: The Lorenz curve plots the cumulative share of total voting power against the cumulative share of voters, sorted from least to most powerful (Cowell, 2011). A perfectly equal system would produce a diagonal line; greater curvature indicates greater inequality.
- **Gini Coefficient**: The Gini coefficient summarises the Lorenz curve into a single value between 0 and 1, where 0 represents perfect equality and 1 represents maximum inequality (Cowell, 2011).

- **Maximum Voting Weight**: Tracking the maximum voting weight held by any single agent provides a direct measure of super-voter emergence.

These measures, traditionally used in economic contexts to assess wealth inequality, are well-suited for evaluating how fairly voting power is distributed within a delegation network.

Delegation Network Structure

The structure of the delegation network itself would also provide valuable insights:

- **Average Delegation Path Length**: Indicates the typical number of delegation steps from voters to their representatives.
- **Cycle Frequency**: Measures the frequency of attempted cycles, even if prevented by the system.
- **Network Connectivity**: Assesses whether the delegation network remains cohesive or fragments into isolated groups.

Comparative Analysis

By applying these metrics across different delegation models – such as ranked delegation (see Section 4.3) or weighted delegation (see Section 4.4) – simulations could systematically compare the performance of different mechanisms. Longitudinal analysis could further reveal whether delegation networks tend toward stability or increasing inequality over time.

7.1.5 Summary

Agent-based modelling presents a promising approach for exploring the dynamic behaviours underlying delegation systems. By modelling individual agent decisions and trust evolution, future simulations could provide deeper insights into the stability, fairness, and scalability of liquid democracy mechanisms.

7.2 Potential Extensions for Vodle

While the implementation described in this project significantly improves delegation flexibility and expressiveness in vodle, several further extensions could enhance the platform's scalability, usability, and transparency. This section outlines selected directions for future development.

7.2.1 Global Delegations

In the current implementation, delegations are issued individually for each poll, requiring users to manually re-invite their preferred delegates whenever they join a new poll. For users who consistently wish to delegate their participation, this process introduces unnecessary overhead.

Global delegations would allow users to establish persistent delegations that automatically apply across multiple polls until explicitly overridden or revoked. This design mirrors the structure used in LiquidFeedback (Behrens et al., 2014), where users can define delegations globally, per subject area, or per issue, with more specific delegations taking precedence.

Implementing global delegations in vodle would require changes to the platform's identity management system. At present, poll-specific user identifiers are used to preserve privacy and prevent cross-poll tracking. Supporting global delegations would necessitate introducing persistent identifiers scoped exclusively to the delegation system, ensuring delegation persistence without compromising poll privacy.

Delegations would be stored within user profile metadata, and the resolution process would prioritise per-poll delegations before falling back to any global delegation. Cycle prevention mechanisms would also need to account for both global and local delegation links during validation.

At the user interface level, additional screens would allow users to manage global delegations, review their active settings, and revoke delegations as needed. Clear indicators during voting would distinguish between global and poll-specific delegations, maintaining transparency about how votes are cast.

Introducing global delegations would reduce management friction for active users, accommodate varying engagement levels, and further scale vodle's implementation of liquid democracy.

7.2.2 Delegation Expiry Mechanisms

Although persistent delegations improve convenience, they risk becoming outdated if users' trust relationships change over time. To mitigate this, vodle could introduce optional expiry periods for delegations.

When issuing a delegation, users would have the ability to specify an expiry period, such as six months or one year. Once expired, the delegation would automatically become inactive, prompting the user to review, renew, or modify their delegation choice.

Delegation expiry would encourage users to regularly reconsider their trust relationships, prevent passive accumulation of voting power, and promote a dynamic and up-to-date delegation network. Technically, this extension would involve associating an optional expiry timestamp with each delegation and modifying the resolution process to treat expired delegations as inactive, defaulting either to backup delegates or to direct voting.

Providing reasonable default expiry options, combined with user notifications prior to expiry, would help balance user convenience with network integrity.

7.2.3 Auditability of Delegation Chains

Transparency in how votes are cast and propagated is crucial for building trust in liquid democracy systems. One potential extension is to provide users with greater auditability of their delegation chains: showing how their vote travels through the network to reach its final casting point.

However, auditability must be balanced carefully against delegate privacy. In vodle, delegations and votes are private, and no user identity should be exposed unless explicitly permitted. Drawing inspiration from the "Golden Rule of Liquid Democracy" proposed in Google Votes (Hardt and Lopes, 2015), a privacy-preserving auditability model can be adopted: users can trace how their own votes were used, but identities of intermediaries are only shown if those users have explicitly opted into public visibility.

Practically, vodle could implement a system where:

- Users see the number of steps their delegated vote has traversed.
- If an intermediary has opted into being a public delegate, their pseudonym is displayed.
- Otherwise, intermediate steps are anonymised but structurally visible.

This approach preserves voter transparency without compromising delegate privacy, and aligns with best practices in privacy-respecting liquid democracy systems.

7.3 Summary

This chapter has outlined potential directions for extending vodle's delegation system and for investigating the dynamics of liquid democracy through simulation. Agent-based modelling could offer insights into the evolution of trust and delegation behaviour, while platform-level enhancements such as global delegations, delegation expiry mechanisms, and privacy-preserving auditability would further strengthen usability and transparency. Together, these developments would support more flexible, scalable, and trustworthy decision-making systems based on liquid democracy principles.

Chapter 8

Conclusions

This project set out to design and implement a flexible, reliable liquid democracy system within vodle, with the aim of enhancing participation, autonomy, and resilience in group decision-making. The work addressed the core challenges of traditional delegation models by introducing mechanisms for ranked delegation, per-option delegation, and weighted delegation based on a trust matrix model.

The project successfully delivered a fully functioning core delegation model with explicit invitation workflows and effective cycle prevention. Ranked delegation was implemented using the MinSum rule, offering users backup options while maintaining clear and interpretable delegation paths. Per-option delegation allowed voters to assign different delegates to different decision options, increasing flexibility and expressiveness. Weighted delegation, based on a trust matrix model, enabled fine-grained trust allocations across multiple delegates, supporting nuanced participation while mitigating the risks of super-voter concentration and vote loss.

Throughout the development process, the unique architectural constraints of vodle – particularly its serverless, client-side execution model using CouchDB – heavily influenced design decisions. All delegation resolution and vote computation had to be performed efficiently within the browser, leading to careful optimisation of algorithms and data structures.

Evaluation demonstrated that the implemented features met the functional and non-functional requirements identified at the outset of the project. Delegation mechanisms were integrated into vodle without compromising usability or transparency, and testing confirmed the correctness and resilience of the system under a variety of scenarios. Although full-scale agent-based simulations were not completed within the timeframe, the project established the necessary foundations for future empirical analysis of delegation behaviours.

Several limitations remain. In particular, while the trust matrix model offers high expressiveness, it also introduces complexity that may affect long-term usability for some users. Furthermore, the system's performance under extremely large-scale usage has not yet been fully evaluated. These limitations point to natural directions for future work, including the development of dynamic trust adjustment mechanisms, support for global delegates, and the completion of agent-based modelling to explore emergent behaviours at scale.

Overall, the project demonstrates that liquid democracy can be practically and effectively integrated into a real-time, web-based decision-making system like vodle. By supporting flexible forms of delegation while preserving user autonomy and system transparency, the implemented features significantly enhance vodle's ability to adapt to users' varying levels of engagement and trust. This work provides a strong foundation for further extensions, broader applications, and continued research into participatory decision-making systems.

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.

Casella, Alessandra & Campbell, Joseph & de Lara, Lucas & Mooers, Victoria & Ravindran, Dilip. Liquid Democracy. Two Experiments on Delegation in Voting, December 2022.

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

Christoff, Zoé & Grossi, Davide. Binary Voting with Delegable Proxy: An Analysis of Liquid Democracy. *Electronic Proceedings in Theoretical Computer Science*, 251:134–150, July 2017b. ISSN 2075-2180. doi: 10.4204/EPTCS.251.10.

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

Cowell, Frank. *Measuring Inequality*. Oxford University Press, January 2011. ISBN 978-0-19-959403-0. doi: 10.1093/acprof:osobl/9780199594030.001.0001.

Degrave, Jonas. Resolving multi-proxy transitive vote delegation. 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.

Appendices

Appendix A

Code for Unit Tests

A.1 Core Delegation

```
// --- Global Test Counter ---
let testCounter = 0;

// --- Simulated Maps for Testing ---
const directDelegationMap = new Map();
const inverseIndirectMap = new Map();

// --- Simulated Database Accessors ---
function get_direct_delegation_map() {
  return directDelegationMap;
}

function get_inverse_indirect_map() {
  return inverseIndirectMap;
}

function set_inverse_indirect_map(newMap) {
  inverseIndirectMap.clear();
  for (const [k, v] of newMap.entries()) {
    inverseIndirectMap.set(k, new Set(v));
  }
}

// --- Core Delegation Logic ---
function can_add_delegation(voter, delegate) {
  if (voter === delegate) return false;
```

```
  const dm = get_direct_delegation_map();
  const existingDelegate = dm.get(delegate);
  if (existingDelegate === voter) return false;
  const sm = get_inverse_indirect_map();
  const voterDependents = sm.get(voter);
  if (voterDependents && voterDependents.has(delegate)) return false;
  return true;
}

function add_delegation(voter, delegate) {
  if (!can_add_delegation(voter, delegate)) return false;
  const dm = get_direct_delegation_map();
  dm.set(voter, delegate);

  const sm = get_inverse_indirect_map();
  const voterDependents = new Set([voter]);
  const voterIndirect = sm.get(voter);
  if (voterIndirect) {
    for (const dep of voterIndirect) {
      voterDependents.add(dep);
    }
  }
  const delegateDependents = sm.get(delegate) || new Set();
  for (const dep of voterDependents) {
    delegateDependents.add(dep);
  }
  sm.set(delegate, delegateDependents);
  return true;
}

function remove_delegation(voter) {
  const dm = get_direct_delegation_map();
  const sm = get_inverse_indirect_map();
  const delegate = dm.get(voter);
  if (!delegate) return false;
  dm.delete(voter);

  const voterDependents = new Set([voter]);
  const voterIndirect = sm.get(voter);
  if (voterIndirect) {
```

```
    for (const dep of voterIndirect) {
      voterDependents.add(dep);
    }
  }
  for (const [d, dependents] of sm.entries()) {
    for (const dep of voterDependents) {
      dependents.delete(dep);
    }
  }
  return true;
}

// --- Utilities ---
function resetMaps() {
  directDelegationMap.clear();
  inverseIndirectMap.clear();
}

function assertEqual(actual, expected, message) {
  if (JSON.stringify(actual) !== JSON.stringify(expected)) {
    console.error(`FAIL (Test ${testCounter}): ${message}\nExpected: ${JSON.stringify
    throw new Error("Test failed");
  } else {
    console.log(`PASS (Test ${testCounter}): ${message}`);
  }
}

function assert(condition, message) {
  if (!condition) {
    console.error(`FAIL (Test ${testCounter}): ${message}`);
    throw new Error("Test failed");
  } else {
    console.log(`PASS (Test ${testCounter}): ${message}`);
  }
}

// --- Test Cases ---
function test_simple_delegation() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Simple Delegation ===`);
```

```
  resetMaps();
  add_delegation('A', 'B');
  assertEqual(Object.fromEntries(directDelegationMap), { A: 'B' }, "A should delegate
}

function test_chain_no_cycle() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Chain Without Cycle ===`);
  resetMaps();
  add_delegation('A', 'B');
  add_delegation('B', 'C');
  add_delegation('C', 'D');
  assertEqual(Object.fromEntries(directDelegationMap), { A: 'B', B: 'C', C: 'D' }, "C
}

function test_detect_cycle() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Detect Cycle ===`);
  resetMaps();
  add_delegation('A', 'B');
  add_delegation('B', 'C');
  add_delegation('C', 'D');
  const success = add_delegation('D', 'A');
  assert(!success, "Should detect cycle when D tries to delegate to A");
}

function test_remove_delegation() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Remove Delegation ===`);
  resetMaps();
  add_delegation('A', 'B');
  add_delegation('B', 'C');
  remove_delegation('B');
  assertEqual(Object.fromEntries(directDelegationMap), { A: 'B' }, "Only A->B should
}

function test_add_after_removal() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Add After Removal ===`);
  resetMaps();
```

```
  add_delegation('A', 'B');
  add_delegation('B', 'C');
  add_delegation('C', 'D');
  remove_delegation('B');
  const success = add_delegation('D', 'A');
  assert(success, "Should allow D->A after B->C removed");
}

function test_delegation_to_self() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Delegation to Self ===`);
  resetMaps();
  const success = add_delegation('A', 'A');
  assert(!success, "Should not allow delegation to self");
}

function test_indirect_dependency_tracking() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Indirect Dependency Tracking ===`);
  resetMaps();
  add_delegation('A', 'B');
  add_delegation('B', 'C');
  add_delegation('C', 'D');
  const indirect = [...inverseIndirectMap.get('D')];
  indirect.sort();
  assertEqual(indirect, ['A', 'B', 'C'], "D should have indirect dependents A, B, C")
}

function test_multiple_voters_to_same_delegate() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Multiple Voters to Same Delegate ===`);
  resetMaps();
  add_delegation('A', 'Z');
  add_delegation('B', 'Z');
  add_delegation('C', 'Z');
  assertEqual(Object.fromEntries(directDelegationMap), { A: 'Z', B: 'Z', C: 'Z' }, "Mu
}

function test_star_topology() {
  testCounter++;
```

```
  console.log(`\n=== Test ${testCounter}: Star Topology ===`);
  resetMaps();
  add_delegation('A', 'Z');
  add_delegation('B', 'Z');
  add_delegation('C', 'Z');
  add_delegation('D', 'Z');
  add_delegation('E', 'Z');
  const indirect = [...inverseIndirectMap.get('Z')];
  indirect.sort();
  assertEqual(indirect, ['A', 'B', 'C', 'D', 'E'], "Z should have A, B, C, D, E as dep
}

// --- Run All Tests ---
function run_all_core_delegation_tests() {
  console.log("\n=== Running Core Delegation Unit Tests ===");

  test_simple_delegation();
  test_chain_no_cycle();
  test_detect_cycle();
  test_remove_delegation();
  test_add_after_removal();
  test_delegation_to_self();
  test_indirect_dependency_tracking();
  test_multiple_voters_to_same_delegate();
  test_star_topology();

  console.log(`\n=== ALL ${testCounter} CORE DELEGATION TESTS PASSED ===`);
}

// --- Start ---
run_all_core_delegation_tests();
```

A.2 Ranked Delegation

```
// --- Simulated Maps for testing (Ranked Delegation Only) ---
const directDelegationMap = new Map(); // voter -> array of [did, rank, status]

// --- Simulated DB accessors ---
function get_direct_delegation_map() {
```

```
  return directDelegationMap;
}

function resetMaps() {
  directDelegationMap.clear();
  did_to_delegate.clear();
  did_to_rank.clear();
}

// --- Helper Functions ---

function get_delegations(voter) {
  return get_direct_delegation_map().get(voter) || [];
}

function set_delegations(voter, delegations) {
  get_direct_delegation_map().set(voter, delegations);
}

function find_all_paths(voter, currentPath, paths) {
  const delegations = get_delegations(voter);
  for (const [did, _rank, active] of delegations) {
    if (active === '0') {
      continue;
    }
    const delegate = get_delegate_from_did(did);
    const newPath = [...currentPath, [voter, did]];
    if (is_casting_voter(delegate)) {
      paths.push(newPath);
    } else if (!currentPath.find(([v, _]) => v === delegate)) {
      find_all_paths(delegate, newPath, paths);
    }
  }
}

function is_casting_voter(voter) {
  const delegations = get_delegations(voter);
  for (const [did, _rank, active] of delegations) {
    if (active !== '0') {
      return false;
```

```
    }
  }
  return true;
}

function get_delegate_from_did(did) {
  return did_to_delegate.get(did);
}

function get_rank_from_did(did) {
  return did_to_rank.get(did);
}

function min_sum(voter) {
  const paths = [];
  find_all_paths(voter, [], paths);

  let minSumPath = [];
  let minSum = Number.MAX_VALUE;

  for (const path of paths) {
    let sum = 0;
    for (const [_v, did] of path) {
      sum += get_rank_from_did(did);
    }
    if (sum < minSum) {
      minSum = sum;
      minSumPath = path;
    }
  }
  return minSumPath;
}

function min_sum_all() {
  for (const voter of get_direct_delegation_map().keys()) {
    const minPath = min_sum(voter);
    if (minPath.length > 0) {
      for (const [v, did] of minPath) {
        const delegations = get_delegations(v);
        const updated = delegations.map(([d, rank, active]) => {
```

```
          if (d === did) {
            return [d, rank, '2'];
          } else if (active === '2') {
            return [d, rank, '1'];
          }
          return [d, rank, active];
        });
        set_delegations(v, updated);
      }
    }
  }
}

// --- Utilities for Testing ---

let testCounter = 1;

function assertEqual(actual, expected, message) {
  if (JSON.stringify(actual) !== JSON.stringify(expected)) {
    console.error(`FAIL (Test #${testCounter}): ${message}\nExpected: ${JSON.stringif
    throw new Error("Test failed");
  } else {
    console.log(`PASS (Test #${testCounter}): ${message}`);
    testCounter++;
  }
}

// --- Mappings for DID -> delegate and rank ---
const did_to_delegate = new Map();
const did_to_rank = new Map();

// --- Test Cases ---

function test_ranked_delegation_simple() {
  console.log("\n=== Test: Simple Ranked Delegation ===");
  resetMaps();

  set_delegations('A', [['did1', 1, '1'], ['did2', 2, '1']]);
  did_to_delegate.set('did1', 'B');
  did_to_delegate.set('did2', 'C');
```

```
  did_to_rank.set('did1', 1);
  did_to_rank.set('did2', 2);

  min_sum_all();

  assertEqual(get_delegations('A'), [['did1', 1, '2'], ['did2', 2, '1']], "A should a
}

function test_ranked_delegation_chain() {
  console.log("\n=== Test: Ranked Delegation Chain ===");
  resetMaps();

  set_delegations('A', [['did1', 1, '1']]);
  set_delegations('B', [['did2', 2, '1']]);
  did_to_delegate.set('did1', 'B');
  did_to_delegate.set('did2', 'C');
  did_to_rank.set('did1', 1);
  did_to_rank.set('did2', 2);

  min_sum_all();

  assertEqual(get_delegations('A'), [['did1', 1, '2']], "A's delegation to B should s
  assertEqual(get_delegations('B'), [['did2', 2, '2']], "B's delegation to C should b
}

function test_ranked_delegation_multiple_paths() {
  console.log("\n=== Test: Ranked Delegation Multiple Paths ===");
  resetMaps();

  set_delegations('A', [['did1', 3, '1'], ['did2', 1, '1']]);
  did_to_delegate.set('did1', 'B');
  did_to_delegate.set('did2', 'C');
  did_to_rank.set('did1', 3);
  did_to_rank.set('did2', 1);

  min_sum_all();

  assertEqual(get_delegations('A'), [['did1', 3, '1'], ['did2', 1, '2']], "A should p
}
```

```
// --- More Complex Tests ---

function test_ranked_delegation_complex_1() {
  console.log("\n=== Test: Complex Ranked Delegation 1 ===");
  resetMaps();

  set_delegations('A', [['did1', 2, '1'], ['did2', 5, '1']]);
  set_delegations('B', [['did3', 1, '1']]);
  did_to_delegate.set('did1', 'B');
  did_to_delegate.set('did2', 'D');
  did_to_delegate.set('did3', 'C');
  did_to_rank.set('did1', 2);
  did_to_rank.set('did2', 5);
  did_to_rank.set('did3', 1);

  min_sum_all();

  assertEqual(get_delegations('A'), [['did1', 2, '2'], ['did2', 5, '1']], "A should ac
  assertEqual(get_delegations('B'), [['did3', 1, '2']], "B should activate to C throug
}

function test_ranked_delegation_complex_2() {
    console.log("\n=== Test: Complex Ranked Delegation 2 ===");
    resetMaps();

    set_delegations('A', [['did1', 5, '1'], ['did2', 2, '1']]);
    set_delegations('B', [['did3', 2, '1']]);
    set_delegations('C', [['did4', 3, '1']]);
    did_to_delegate.set('did1', 'B');
    did_to_delegate.set('did2', 'C');
    did_to_delegate.set('did3', 'D');
    did_to_delegate.set('did4', 'E');
    did_to_rank.set('did1', 5);
    did_to_rank.set('did2', 2);
    did_to_rank.set('did3', 2);
    did_to_rank.set('did4', 3);

    min_sum_all();

    assertEqual(get_delegations('A'), [['did1', 5, '1'], ['did2', 2, '2']], "A should
```

```
    assertEqual(get_delegations('B'), [['did3', 2, '2']], "B should activate to D thr
    assertEqual(get_delegations('C'), [['did4', 3, '2']], "C should activate to E thr

}


// --- Run All Tests ---

function run_ranked_delegation_tests() {
  console.log("\n=== Running Ranked Delegation Tests ===");
  test_ranked_delegation_simple();
  test_ranked_delegation_chain();
  test_ranked_delegation_multiple_paths();
  test_ranked_delegation_complex_1();
  test_ranked_delegation_complex_2();
  console.log("\n=== ALL RANKED DELEGATION TESTS PASSED ===");
}

// --- Start Tests ---
run_ranked_delegation_tests();
```

A.3 Weighted Delegation

```
// test_weighted_delegation.js

// --- Global Test Counter ---
let testCounter = 0;

// --- Simulated Maps for Testing ---
const directDelegationMap = new Map(); // voter -> array of [did, trust, active]
const didToDelegate       = new Map(); // did -> delegate userId

// --- Utilities ---
function resetMaps() {
  directDelegationMap.clear();
  didToDelegate.clear();
}
```

```
function assertEqual(actual, expected, message) {
  if (JSON.stringify(actual) !== JSON.stringify(expected)) {
    console.error(`FAIL (Test ${testCounter}): ${message}
Expected: ${JSON.stringify(expected)}
Got:      ${JSON.stringify(actual)}\n`);
    throw new Error("Test failed");
  } else {
    console.log(`PASS (Test ${testCounter}): ${message}`);
  }
}

// --- Helper Functions for Weighted Delegation ---

/**
 * Runs the iterative weighted-delegation update until convergence or max iterations.
 * - selfMap: Map<userId, Map<optionId, number>>
 */
function updateEffectiveVotes(selfMap, maxIters = 50) {
  // initialize effective = self
  let effective = new Map();
  for (const [uid, ratings] of selfMap.entries()) {
    effective.set(uid, new Map(ratings));
  }

  for (let iter = 0; iter < maxIters; iter++) {
    let changed = false;
    const nextEff = new Map();

    for (const [uid, ratings] of selfMap.entries()) {
      const delegs = directDelegationMap.get(uid) || [];
      if (delegs.length === 0) {
        // no delegation: identical to self
        nextEff.set(uid, new Map(ratings));
        continue;
      }
      const combined = new Map();
      for (const [oid, selfVal] of ratings.entries()) {
        let weightDone = 0, val = 0;
        for (const [did, trustStr, active] of delegs) {
          if (active === '0') continue;
```

```
          const trust = parseInt(trustStr, 10);
          weightDone += trust;
          const delegate = didToDelegate.get(did);
          val += (trust/100) * effective.get(delegate).get(oid);
        }
        val += ((100 - weightDone)/100) * selfVal;
        // detect any change
        if (!combined.has(oid) || val !== effective.get(uid).get(oid)) {
          changed = true;
        }
        combined.set(oid, val);
      }
      nextEff.set(uid, combined);
    }

    effective = nextEff;
    if (!changed) break;
  }

  // floor final values
  const floored = new Map();
  for (const [uid, ratings] of effective.entries()) {
    const m = new Map();
    for (const [oid, v] of ratings.entries()) {
      m.set(oid, Math.floor(v));
    }
    floored.set(uid, m);
  }
  return floored;
}

// --- Test Cases ---

function test_no_delegation() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: No Delegation ===`);
  resetMaps();
  const self = new Map([
    ['A', new Map([ ['x',50], ['y',80] ])]
  ]);
```

```
  const eff = updateEffectiveVotes(self);
  assertEqual(
    Array.from(eff.get('A').entries()),
    [['x',50],['y',80]],
    "With no delegations, effective == self"
  );
}

function test_simple_weighted_delegation() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Simple Weighted Delegation ===`);
  resetMaps();
  directDelegationMap.set('A', [['did1','60','1']]);
  didToDelegate.set('did1','B');
  const self = new Map([
    ['A', new Map([ ['x',100] ])],
    ['B', new Map([ ['x',50]  ])]
  ]);
  const eff = updateEffectiveVotes(self);
  // 0.6*50 + 0.4*100 = 30+40 = 70
  assertEqual(
    Array.from(eff.get('A').entries()),
    [['x',70]],
    "A's effective x = 60% of B's 50 + 40% of own 100"
  );
  assertEqual(
    Array.from(eff.get('B').entries()),
    [['x',50]],
    "B unaffected"
  );
}

function test_chain_weighted_delegation() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Chain Weighted Delegation ===`);
  resetMaps();
  directDelegationMap.set('A', [['d1','50','1']]);
  directDelegationMap.set('B', [['d2','50','1']]);
  didToDelegate.set('d1','B');
  didToDelegate.set('d2','C');
```

```
  const self = new Map([
    ['A', new Map([ ['x',100] ])],
    ['B', new Map([ ['x',80]  ])],
    ['C', new Map([ ['x',20]  ])]
  ]);
  const eff = updateEffectiveVotes(self);
  // B: .5*20 + .5*80 = 50
  // A: .5*50 + .5*100 = 75
  assertEqual(
    Array.from(eff.get('B').entries()),
    [['x',50]],
    "B's effective x = 50%"
  );
  assertEqual(
    Array.from(eff.get('A').entries()),
    [['x',75]],
    "A's effective x = 75%"
  );
}

function test_multi_delegate_splitting() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Multi-Delegate Splitting ===`);
  resetMaps();
  directDelegationMap.set('A', [
    ['d1','30','1'],
    ['d2','50','1']
  ]);
  didToDelegate.set('d1','B');
  didToDelegate.set('d2','C');
  const self = new Map([
    ['A', new Map([ ['x',100], ['y',0] ])],
    ['B', new Map([ ['x',0],   ['y',100] ])],
    ['C', new Map([ ['x',50],  ['y',50]  ])]
  ]);
  const eff = updateEffectiveVotes(self);
  // x= .3*0 + .5*50 + .2*100 = 45 ; y= .3*100 + .5*50 + .2*0 = 55
  assertEqual(
    Array.from(eff.get('A').entries()),
    [['x',45],['y',55]],
```

```
    "A splits trust: x=45, y=55"
  );
}

function test_exact_full_self_trust() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Full Self-Trust Edge ===`);
  resetMaps();
  directDelegationMap.set('A',[ ['d1','100','1'] ]);
  didToDelegate.set('d1','B');
  const self = new Map([
    ['A', new Map([ ['x',90] ])],
    ['B', new Map([ ['x',10] ])]
  ]);
  const eff = updateEffectiveVotes(self);
  // A fully delegates, so x = B's 10
  assertEqual(
    Array.from(eff.get('A').entries()),
    [['x',10]],
    "A's x fully delegated to B"
  );
}

function test_cyclical_delegation_convergence() {
  testCounter++;
  console.log(`\n=== Test ${testCounter}: Cyclical Delegation Convergence ===`);
  resetMaps();
  // A <-> B each 50%
  directDelegationMap.set('A',[ ['d1','50','1'] ]);
  directDelegationMap.set('B',[ ['d2','50','1'] ]);
  didToDelegate.set('d1','B');
  didToDelegate.set('d2','A');
  const self = new Map([
    ['A', new Map([ ['x',80] ])],
    ['B', new Map([ ['x',80] ])]
  ]);
  const eff = updateEffectiveVotes(self);
  // both should remain at 80 after convergence
  assertEqual(
    Array.from(eff.get('A').entries()),
```

```
    [['x',80]],
    "A converges to 80 in the A<->B cycle"
  );
  assertEqual(
    Array.from(eff.get('B').entries()),
    [['x',80]],
    "B converges to 80 in the A<->B cycle"
  );
}

// --- Run All Tests ---
function run_all_weighted_tests() {
  console.log("\n=== Running Weighted Delegation Tests ===");
  test_no_delegation();
  test_simple_weighted_delegation();
  test_chain_weighted_delegation();
  test_multi_delegate_splitting();
  test_exact_full_self_trust();
  test_cyclical_delegation_convergence();
  console.log(`\n=== ALL ${testCounter} WEIGHTED DELEGATION TESTS PASSED ===`);
}

run_all_weighted_tests();
```

Appendix B

Code for Weighted Delegation Performance Testing

```
    // perf_weighted_delegation.js
// This module exports a runPerformance function that benchmarks the
// weighted-delegation convergence routine (random graphs + worst-case chain).

const { performance } = require('perf_hooks');

// --- Core updateEffectiveVotes implementation ---
function updateEffectiveVotes(
  selfMap,
  directDelMap,
  agreementMap,
  options,
  epsilon = 0.00001,
  maxIters = 10000
) {
  let effective = new Map(selfMap);
  let iterations = 0;

  while (iterations < maxIters) {
    let maxDelta = 0;
    const nextEff = new Map();

    for (const [uid, selfRatings] of selfMap.entries()) {
      const delegs = directDelMap.get(uid) || [];
      if (delegs.length === 0) {
        nextEff.set(uid, new Map(selfRatings));
```

```
        continue;
      }

      const newRatings = new Map();
      for (const oid of options) {
        let weightDone = 0, acc = 0;
        for (const [did, trust, status] of delegs) {
          if (status === '0') continue;
          const agr = agreementMap.get(did);
          if (!agr || agr.status === 'pending') continue;
          const delId = agr.delegate_vid;
          weightDone += trust;
          acc += (trust / 100) * effective.get(delId).get(oid);
        }
        // include self-trust
        acc += ((100 - weightDone) / 100) * selfMap.get(uid).get(oid);
        maxDelta = Math.max(maxDelta, Math.abs(acc - effective.get(uid).get(oid)));
        newRatings.set(oid, acc);
      }
      nextEff.set(uid, newRatings);
    }

    effective = nextEff;
    iterations++;
    if (maxDelta <= epsilon) break;
  }

  // floor final ratings
  for (const ratings of effective.values()) {
    for (const oid of ratings.keys()) {
      ratings.set(oid, Math.floor(ratings.get(oid)));
    }
  }

  return { effective, iterations };
}

// --- Helper to generate random test graph ---
function randomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
```

```
}

function generateRandomGraph(N, options) {
  const selfMap = new Map();
  const directDelMap = new Map();
  const agreementMap = new Map();
  const voters = Array.from({ length: N }, (_, i) => `V${i + 1}`);

  // random self-ratings
  voters.forEach(v => {
    const ratings = new Map();
    options.forEach(o => ratings.set(o, Math.random() * 100));
    selfMap.set(v, ratings);
  });

  // random trust delegation
  voters.forEach(v => {
    const pool = voters.filter(x => x !== v);
    let selfTrust = randomInt(1, 100),
        remaining = 100 - selfTrust;

    while (remaining > 0 && pool.length > 0) {
      let delegate, trust;
      if (pool.length === 1) {
        delegate = pool[0];
        trust = remaining;
      } else {
        const idx = randomInt(0, pool.length - 1);
        delegate = pool.splice(idx, 1)[0];
        trust = randomInt(1, remaining);
      }
      remaining -= trust;
      const did = `${v}->${delegate}:${trust}`;
      directDelMap.set(v, (directDelMap.get(v) || []).concat([[did, trust, '2']]));
      agreementMap.set(did, { status: 'agreed', delegate_vid: delegate });
    }
  });

  return { selfMap, directDelMap, agreementMap };
}
```

```
// --- Helper to generate worst-case chain graph ---
function generateChainGraph(N, options) {
  const selfMap = new Map();
  const directDelMap = new Map();
  const agreementMap = new Map();
  const voters = Array.from({ length: N }, (_, i) => `V${i + 1}`);

  // random self-ratings
  voters.forEach(v => {
    const ratings = new Map();
    options.forEach(o => ratings.set(o, Math.random() * 100));
    selfMap.set(v, ratings);
  });

  const trust = 99;
  // build a linear chain: V1->V2->...->VN
  for (let i = 0; i < N - 1; i++) {
    const v = voters[i], nxt = voters[i + 1];
    const did = `${v}->${nxt}:${trust}`;
    directDelMap.set(v, [[did, trust, '2']]);
    agreementMap.set(did, { status: 'agreed', delegate_vid: nxt });
  }
  const did = `${voters[N]}->${voters[0]}:${trust}`
  directDelMap.set(voters[N], [[did, trust, '2']]);
  agreementMap.set(did, { status: 'agreed', delegate_vid: voters[0] });

  return { selfMap, directDelMap, agreementMap };
}

// --- Main benchmarking function ---
async function runPerformance(sizes, reps, options) {
  const results = [];

  for (const N of sizes) {
    //random graphs
    let sumI = 0, maxI = 0, sumT = 0, maxT = 0;
    for (let r = 0; r < reps; r++) {
      const { selfMap, directDelMap, agreementMap } = generateRandomGraph(N, options)
      const t0 = performance.now();
```

```
      const { iterations } = updateEffectiveVotes(selfMap, directDelMap, agreementMap
      const t1 = performance.now();
      sumI += iterations;
      maxI = Math.max(maxI, iterations);
      const dt = t1 - t0;
      sumT += dt;
      maxT = Math.max(maxT, dt);
    }

    // worst-case
    {
      const { selfMap, directDelMap, agreementMap } = generateChainGraph(N, options);
      const t0 = performance.now();
      const { iterations } = updateEffectiveVotes(selfMap, directDelMap, agreementMap
      const t1 = performance.now();
      maxT = Math.max(maxT, t1-t0)
      maxI = Math.max(maxI, iterations);
    }

    results.push({
        size: N,
        type: 'random',
        avgIters: sumI / reps,
        maxIters: maxI,
        avgTimeMs: sumT / reps,
        maxTimeMs: maxT
      });
  }

  return results;
}

module.exports = { runPerformance, updateEffectiveVotes };
```