Cover sheet

NATIONAL UNIVERSITY OF SINGAPORE

DSA3101: Data Science in Practice

Assignment: Project Technical Report

Name: Gupta Ananya Vikas

Matriculation No.: A0226576W

Major(s): Data Science and Analytics with Second Major in Innovation and Design programme

Contents: (12 pages from Pg. 2 to 13 – to exclude cover and appendix page)

Overview

This project is in collaboration with Constrcushare, which provides a dealing platform for construction industry. Dr.Murugesan, the Chief Data Officer of Constrcushare, shared his vision to spread 'Greater business connectivity and rewards', setting in motion the mission of Constrcushare to connect users via the platform for optimising businesses and building sustainable solutions.

Through this project, Constrcushare was looking for an 'indication' of how the volume of transactions will grow, which features to prioritise. Therefore, our motivation was to apply agent-based simulation models as a basis in evaluation and improvement of existing e-commerce strategies to further obtain data that can be used in business decision analysis for Constrcushare's progress. Following our motivation and their expectations, after several interactions with the website and the Constrcushare team, our team felt that the website rolls out several features supported by small volume of data currently. Hence, there is a need to fill the gap and provide a medium which can show the effect of changes in different features and collect data based for different users as they interact with the website.

Based on the above, we built our problem statement to be 'Understanding how each of the Constrcushare website features affect site-engagement in terms of user interactions', which will guide us in simulating the right

Simulation Model: Agent Based Model (ABM)

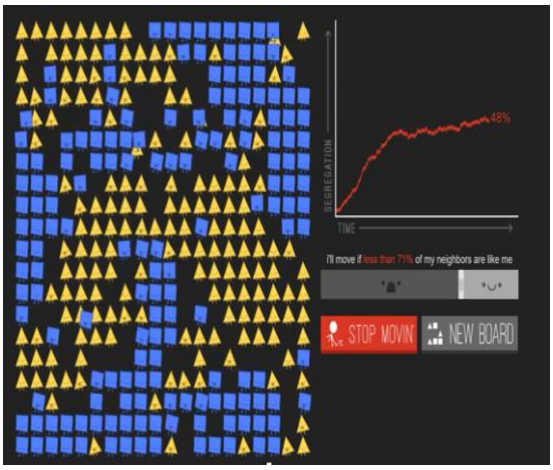


Fig 1: Example of Agent-Based Model

The insights from Bokor et al.'s paper helped narrow the approach further to Agent-based modelling as it weighed different simulations for construction specifically like discrete-event simulation (DES), agent-based modelling (ABM), and system dynamics (SD) under different scenarios. ABM was described much advantageous as there is no set global system behaviour hence the system's behaviour is influenced by individual agents' interaction and with their defined environment rules(Bokor et al., 2019). Such a case relates quite well with the environment of any e-commerce business, which made the replication of Constrcushare's environment via simulation easier and more adapted to their framework.

Since, the focus of the project is on the interaction and behaviour rather than the process, an agent-based model was adopted to construct the simulation, giving a bird's eye view into the effect of different actions on each aspect of the system as well as the system as a whole. While analysing the website features, our team observed that modelling the relationship between a specific feature on Constrcushare and the users, requires the right understanding of the behaviour of the agents within the limited resources. Hence, ABM acts as a cost-effective medium to model the challenges of the online market for optimising the profit and minimising any financial risk(Farooq et al., 2021).

One of the important aspects of ABM which is useful in e-commerce setting is that it provides flexibility in understanding how the system behaves over time even when it doesn't reach a steady or stable state, which is described as "transient and non-equilibrium" behaviour in technical terms(Kazil et al., 2020). E-commerce being an extremely uncertain market, consumer needs are changing constantly as external factors affect their buying and selling behaviour. There is a need to track interactions of sellers and buyers as well as monitor their social behaviours involving spatial or network attributes, which is difficult to map using mathematical equations or any simple machine learning models so ABM captures the path with the solution, simplifying the complex dynamics of the system for the users(Masad & Kazil, 2015). Therefore, ABM facilitates realistic simulation of these behaviours to see quantitatively how they evolve over time and how different factors can have an effect on the agents, that is the users in an e-commerce setting.

Medium of Model: Mesa Python

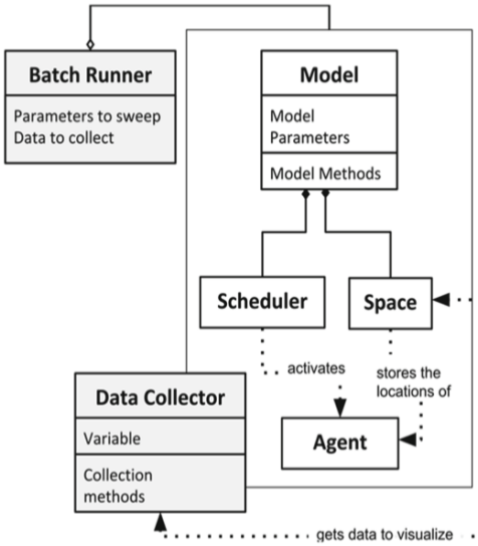


Fig 2: Mesa Architecture

Mesa is an open-source and python-based framework, which provides a fast and versatile framework for building agent based models. One of the main reasons to choose Mesa was that it was python based so it was easy-to-understand, accessible and provides the flexibility to integrate the simulation model with different data science tools like Jupyter Notebooks, Pandas. Moreover, the architecture is independent so replacing components is very convenient and it can be customised for different scale of models as the e-commerce businesses require models to be easy to scale up and scale out whenever needed(Kazil et al., 2020). With the end-goal of Construcshare extending on the model, it would be easier for the team to reproduce the model implementation, which is a major requirement of the project. Also, Mesa has an in-built component for visualising results with multiple agents and it better optimised for large-scale simulations which is required to showcase the outcome of complex e-commerce activities(Masad & Kazil, 2015). Hence, Mesa resonated more with our project objective and suitable for future use too.

Defining variables for Model Implementation

Audience: Chief Data Officer and the development team of Construcshare

Model Purpose: Understand how the relationship between a feature on Construcshare and the users affects members' website engagement and interaction like clicks, user log-ins, post creation, etc.

Explaining the Rationale: After looking at different objectives, the main focus being on 'maximising user onboarding', there was an observation on correlation between the consumer interactions. Since, user interaction drives user onboarding, rather it influences the rate of user onboarding, our team decided to delve deeper into investigating how this engagement of existing users affects the growth of Construcshare and its members over time. This links back directly to how an increased site engagement would achieve the objective of new users through different reasons like similar interests, word-of-mouth or similar dealings in the past, achieving the end goal set for the scope of the project.

Simulation Background and Parameters:
An environment for Construcshare website is set, where the agents pose the role of buyer or seller during an action on the website which relates to interacting with the website in terms of clicking on a post, viewing posts multiple times, negotiating for an item, creating a sell or lease post, creating a buyer request and most of all, completing a purchase, lease or sale transaction. These interactions are monitored over the course of changing platform pricing, which is an additional cost applied to sellers when they complete a sale or lease through the Construcshare platform.

- Dependent Variable: Website Activity and Interaction

Website activity and interaction was chosen mainly as it is a strong predictor of online buying behaviour, displaying demand and relevance to consumers so it can give an insight into what makes consumers choose Construcshare. Monitoring interaction can also help them acknowledge their strengths like features that are valued by consumers and driving engagement on the website. Additionally, interaction has a positive impact on customer experience, which influences customer retention, customer loyalty and most of all, word-of-mouth to enhance new user onboarding. Modelling via this metric can give a more wholistic view on their performance which can be difficult to gauge in the construction industry for Construcshare(Urdea & Constantin, 2021).

- Independent Variable: Altering Platform Fee

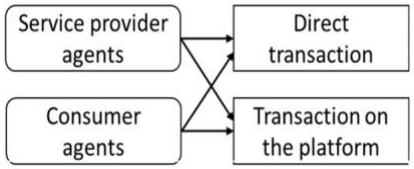


Fig 3: Effect of platform fee

Among different features like pricing, marketing strategies, payment mode and others, platform fee is the one that affects the consumer behaviour closely and directly(Fig 3). It influences the platform user acquisition and the consumers' decision-making on their mode of payment and also for future transactions which is an important consideration for maintaining customer base. More importantly, altering platform fee is necessary as the model can show the effect of change in its value, giving Construcshare an insight into the right platform fee as it affects the overall competitiveness of the platform. Also, it will help them achieve sustainability across the platform ecosystem with profitability for service providers and consumers(Inoue et al., 2019). It can enhance the consumer experience to ensure long-term sustainability and success for Construcshare, which is their main aim as they grow in the e-commerce market.

Model Rules & Assumptions:

Prior to diving into model assumptions, one of the clarifications about the overall focus is that only the website interface was assessed as part of this project and not the application that Construcshare operates. Moving to the model rules and assumptions:

1. All agents under the model are rational
2. The agents are assumed to follow the rule: Higher similarity based on initial relationship between agents, means they are more likely to interact more
3. Each agent has limited patience during a transaction so like in negotiations, the transaction will terminate after 4 negotiations (2 times back and forth)
4. The hypothesis for platform cost is that as platform cost changes, the probability of interaction between agents will change

Model Classes & Methods: ConstrucshareModel, MemberAgent, Post, Item, Network Grid, Relationship

Overview of all the classes and methods defined for the simulation model

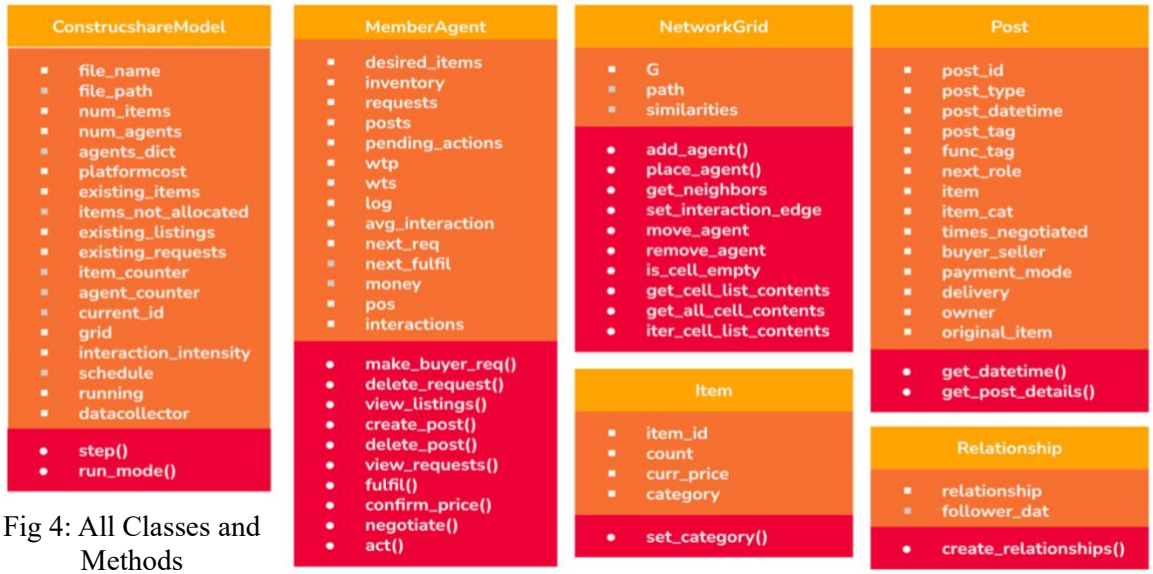


Fig 4: All Classes and Methods

Libraries Used: Mesa, NumPy, Math, Matplotlib, UUID, Pandas, Copy, Collections, Itertools, Numbers, Typing, Warnings, Networkx, Random, Datetime.

Primary resource classes like 'Post' and 'Item' created to ease the process of modelling agent behaviours later for the main model and agent classes.

'**Post**' class object has basic variables defined under the constructor as shown below with two additional methods: get_datetime() for retrieving the date and time of post creation and get_post_details() to get a list of the post particulars like id, type, item, category, etc.

Post Attributes	Initial Set Value	Use case
post_id	*Input when a post object is defined	Unique ID assigned to each post for identifying
post_type	*Input when a post object is defined	Kind of post like Purchase/Rent for buyer request post and Sale/Lease for seller post
post_datetime	Current date, time	Assign timestamp for the date and time of post creation
post_tag	None	If a seller post matches buyer request, the request id is tagged here
func_tag	None	Next function to be called is tagged for correct execution of actions
next_role	None	Assigned as 1 for buyer or 2 for seller for next agent's action
item	*Input when a post object is defined	Stores item object which pertains to this post object
item_cat	*Input when a post object is defined	Category of the item assigned to this post object
times_negotiated	0	Counter for negotiation cycle
buyer_seller	None	Tuple (buyer, seller) to keep track of the agents for this post object
payment_mode	None	Tracks direct deal or via platform dealing
delivery	None	Tracks method of delivery – self-collect or on-site delivery
owner	None	Subordinate variable to storing seller id of the post creator
original_item	None	Subordinate variable for storing item to avoid duplication

'Item' class object has 4 basic variables defined under the constructor as shown below with a category method: set_category() which choose a random category from the 8 categories under Construcshare.

Item Attributes	Initial Set Value	Use case
item_id	*Input when item object is defined	Unique ID assigned to each item for identifying
count	*Input when item object is defined	Stores available number of units of this item
curr_price	*Input when item object is defined	Stores the price assigned for the item
category	Assigned via set_category method()	Stores the category that the item belongs to

Creating the environment of the simulation model, a subclass of mesa.Model is initiated as '**ConstrucshareModel**' class which takes 4 major inputs (marked in green) while running the model to standardise some of the dynamic aspects of the website and make it scalable for the model. Under the constructor method, different attributes of the model shown below are initialised as well as agent classes are initialised within a grid network(space) with added along a scheduler(time) calling the agent behaviours in staged order of execution. The methods defined are:

- step() which calls the scheduler's step method executing the agents' behaviours in the stages order using 'act' function, mentioned in model_stages while initialising the model. Updating the state of the simulation at each time step, looping through the nodes in the grid and using the neighbouring nodes, average interaction intensity their interaction is updated in the network. Also, the step method updates the interaction intensity of each agent's edges in the network by reducing them uniformly by 20% to take into account repeated interaction intensity, which depreciates.
- run_model() performs the step function cycle for 200 iterations to manage the agent actions for 200 steps maximum so that interaction data can be collected in the data collector set up, which can be used to drive more consumer-specific decisions for Construcshare.

Model Attributes	Initial Set Value	Use case
file_name	*Input when model executed	Name of file used for similarity mapping of agents: follower_data.csv
file_path	Path to data folder	Stores path of file for extraction of data

num_items	*Input when model executed	Stores initial number of items introduced in model
num_agents	*Input when model executed	Stores initial number of agents introduced in model
agents_dict	Empty Dictionary	Maps agent's unique id as key and agent object as value
platformcost	*Input when model executed or default as 0.04	Stores proportion of amount charged to sellers when selling or leasing via the platform – assessed through model
existing_items	Empty List	List of all item objects
items_not_allocated	Empty List	Supplementary list of items not allocated or dealt under the model
existing_listings	Empty List	List of current seller post listings
existing_requests	Empty List	List of current buyer request posts
item_counter	0	Increments with item's count for carrying out the model steps
agent_counter	num_agents	Earlier set to initial number of agents – changes as more agents added
current_id	0	Arbitrarily initialising of current model for next model to be called
grid	NetworkGrid object	Creates grid instance to monitor model's space component
interaction_intensity	Sum of grid edge interaction	Using grid, calculating sum of interaction over the edges and between agent nodes in the network
schedule	Time scheduler	StagedActivation scheduler used for model's time component to execute each stage of action one after another – here implemented using act() function of the agents, not time component entirely
running	True	Boolean variable to manage current state of model - running or not
datacollector	DataCollector	Mesa's Data collector component to record interaction and average interaction data as the agent actions are executed

'**Relationship**' class was created to specifically apply the follower data provided by Construcshare to map the initial relationship between agent nodes in the network. It follows the rationale that if a user follows another user: the edge weight is higher showing more similarity of the nodes. Initially, the idea was to use calculated Jaccard similarity for this purpose but after feedback, a shift was made for using their follower data as a basis for the Jaccard similarity indicator for the node relation. The method defined is: create_relationships() which uses the input follower data frame to initialise a dictionary mapping the follower user id to following user ids after string manipulations in the followings dictionary created to extract the ids from the list. The relationship score is then calculated and assigned in the relationship dictionary of the class based on the following two conditions:

- if agent 1 follows agent 2 then the score is increased by 5 arbitrarily.
- if both agent 1 and 2 follow each other, the score is updated by sum of each other's score to signify a stronger initial relationship hence greater similarity between the agent nodes.

Relationship Attributes	Initial Set Value	Use case
relationship	Empty Dictionary	Stores each user's ID as key and their relationship score based on the users that they are following
follower_dat	*Input when relationship is defined	Stores the data frame from reading the input followers_data_path

'**NetworkGrid**' class is like the space component of the model implemented for simulating Construcshare as a network of agent nodes shown as a graph where each node represents an agent and edge represents their interaction. There are three attributes defined under the constructor as seen below which are integral in assigning nodes to their position in the grid based on the 'NetworkX' architecture.

Grid Attributes	Initial Set Value	Use case
G	*Input when new network initialised	NetworkX graph instance stored

path	*Input when new network initialised	Stores the edge paths of the graph
similarities	Assigned to Relationship class based on given path	Stores the Relationship class for similarities of nodes based on their follow behaviour

Some methods which have conventional and pre-defined functions are shown below which are mainly used for either updating the grid via adding, removing, moving agents to recalibrate the interaction.

Grid Method	Input	Purpose	Description	Output
add_agent()	Agent class	Adds a new node to the graph with its neighbours	New node with jaccard value set to 0 is added then iterating through all the nodes in the graph, edge is created for new node and jaccard value is changed if other nodes have a relationship with new node else jaccard value set randomly between 0 and 1 with interaction set to 0.	Calls place_agent() function
place_agent()	Agent and node_id	Positions a given agent in a node	Sets agent position to the node_id and add the agent to the agents list under node of graph instance G	None
get_neighbors	node_id, include_center, threshold	Gives all neighboring nodes within a threshold	Among the list of neighbours of the node_id, if distance between them is within a threshold then node is added to the neighbour's list with decision to including the centre or not	List of neighbouring nodes
set_interaction_edge	Start and end node, new_val	Set interaction of an edge between two nodes	Set the interaction value of the edge from start to end and end to start node as the new_val given	None
move_agent	Agent and node_id	Moves agent to new node	Calls remove and place agent for fresh initialised placement using node_id	None
remove_agent	Agent	Discard the agent from the network	Removes agent from list of agents under G and sets the position as None	None
is_cell_empty	node_id	Checks the cell	Checks if contents of the cell is empty or not	True/False
get_cell_list_contents	List of cells	Getting the agents from cell contents	Lists agents contained in the cells of cells_list	List of the agents
get_all_cell_contents	List of agents	Getting all agents	Lists all agents in the network	List of all agents

To model the complete Constrcushare environment, an agent class is defined called '**MemberAgent**' to signify that each agent represents a member of the Constrcushare community as they can only start dealing after registering on the platform. This agent class uses the pre-defined class 'mesa.Agent', which is commonly used in agent based simulations with the pre-fed variables unique_id and model, distinguishing each agent from another using the id and model defined for the setup environment, along with other attributes explicitly defined in the constructor method as seen below.

Agent Attributes	Agent Role	Initial Set Value	Use case
desired_items	Buyer	*Input when a buyer is defined	List of items that the buyer needs or has requested for
inventory	Seller	*Input when a seller is defined	List of items available for sale or lease
requests	Buyer	Empty List	List of all buyer requests made
posts	Seller	Empty List	List of all seller posts created
pending_actions	Buyer & Seller	Empty List	List to monitor and keep track of pending actions for the user so they can fulfil these before moving to next action
wtp	Buyer	*Input when a buyer is defined	Dictionary using item id as key and buyer's respective willingness-to-pay for that particular item as a value
wts	Seller	*Input when seller is defined	Dictionary using item id as key and seller's respective willingness-to-sell for that particular item as a value

log	Buyer & Seller	Empty List	List of all past actions taken by agent
avg_interaction	Buyer & Seller	0	Integer to represent the interactions based on actions taken by the agent
next_req	Buyer & Seller	None	Stores request post if linked with a listed seller post
next_fulfil	Buyer & Seller	None	Stores the post which the agent interacted with and confirmed price for fulfilment of order
money	Buyer & Seller	100	Placeholder value used under fulfilling
pos	Buyer & Seller	0	Shows position of agent
interactions	Buyer & Seller	Empty Dictionary	Dictionary to record the interactions

Moving to the methods which tie around these attributes together for carrying out each agent behaviour seen across the platform from the perspective of buyer and seller.

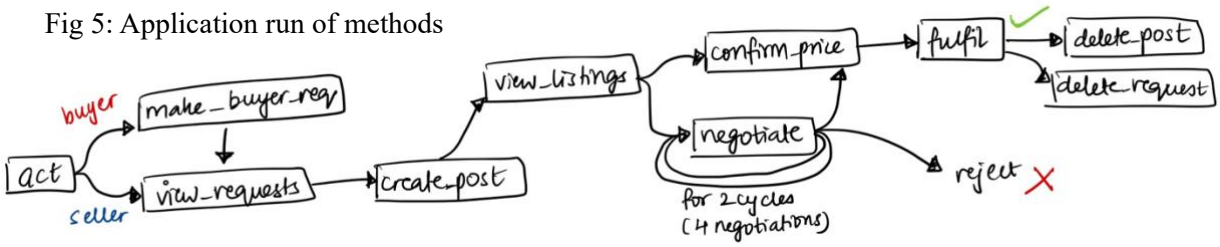
Method	Input	Purpose	Description	Output
make_buyer_req	None	Creates a new buyer request post	If the desired items list is not empty, buyer chooses an item object randomly from its desired items which does not have an existing request post. A post object is created using title, request type (purchase, rent or both) and unique ID assigned to the created request and the buyer_seller tuple is set to indicate self as the buyer. This request is then added to the buyer's requests as well as the model.existing_requests and returned.	Created request (Post object)
delete_request	Request	Discard request	Loop over the model.existing_requests and buyer's requests to match the required post_id and remove it from both the model and agent's request list.	None
view_listings	None	Shows buyer's browsing action	First, the agent checks for the presence of neighbouring node posts, randomly selecting one for viewing. If there are no neighboring posts, it selects a post randomly from model's existing_listings. Then the item associated with the selected post is checked against the buyer's desired items and a copy of the post is created to express interest in that listed seller post, setting the agent's buyer_seller attribute as self and seller who created the post. Lastly, if the buyer's willingness-to-pay for that item is less than the item price, need to move to negotiation so the func_tag is set to negotiate. Else, the price is suitable so the func_tag is set to confirm_price method, and the post is added in the pending_actions list of the buyer and returned.	Listed post copy (Post object)
create_post	None	Creates new seller post	Firstly, if the next_req attributes is not None, means the post creation is triggered by an existing request. Else, if not previously linked to buyer request, it is a fresh post triggered by seller. The particular details of the post like item, category, post type, payment mode, delivery method, etc are set randomly from the options available on the website or set arbitrarily for text entries based on it being a sale or lease post accordingly. New post object is defined and looped over the existing requests to check if the new post matches a request, which is added as post_tag and the agent's buyer_seller attribute is set based on the seller who created the post and buyer if it matches a request else set as None. Finally, this new post is added under existing_listings and seller's posts list with post's owner set as the seller id.	None
delete_post	Seller Post	Discard a post	Matches the seller post id with the contents of existing listings and seller posts and removes it from both the lists	None
view_requests	None	Shows seller's browsing action of buyer requests	Seller chooses a request post randomly from the existing_requests for viewing. Then, this selected request is checked if it has an existing seller post using post_tag else, the item indicated in the selected request is checked if it has enough quantity in the seller's inventory followed by calling create_post(), setting next_req to the selected request – used to model potential posts scenario After this, the seller is added to the buyer's interaction dictionary and the grid, increasing the interaction value by 2 to represent edge between the buyer and seller as they interact over a post.	None

fulfil	None	Completes a purchase, sale or lease transaction	The post sent for fulfilment of order is retrieved using the agent's next_fulfil attribute and then the post is deleted for both agents, item removed from their desired items and their respective willingness to pay and sell for that item is removed as order is being processed so post and item is no longer active. Also, the item is updated in each agent's inventory which allows circular trading scenario so buyer adds the purchased item to their inventory for resale and removed from seller's inventory. The interaction for this fulfilment action is increased by 5 for the seller with regard to the edge in the grid between the seller and buyer. Lastly, another round of checking is done to ensure post and item are removed from all pending_actions, other agent's existing posts to finish this transaction.	None
confirm_price	None	Implements the final quoted costing for an item	Retrieves the last added post from pending_actions and based on the next_role attribute of this post, buyer and seller action for final price quoting is done. If the role is buyer, the buyer will confirm the price and irrespective of the payment mode, post will be removed from requests and func_tag will be set to fulfil for next_role as seller to move to fulfilment. If the role is seller, seller's choice in payment mode is considered so simple deduction from placeholder value in 'money' attribute else addition of platform fee to the final price if via Construcshare payment gateway and func_tag set to fulfil for next_fulfil set as the current post to move to fulfilment of order.	
negotiate	None	Conduct negotiation cycles for two rounds	Models the negotiation cycle of the transaction wherein the last added post from pending_actions is selected and negotiation based on the next_role attribute to represent which agent called the action of negotiation. When called by buyer, there are three states available: 1. If buyer's willingness-to-pay is greater than or equal to the item's price then confirm_price() is called by the buyer - Accepting the transaction 2. If 4 negotiations (2 cycles) performed then, transaction terminated with respect to low patience level of agents – Termination of transaction 3. Else when buyer's willingness-to-pay is lower than price and negotiation limit has not reached 4 yet, then a round of negotiation is initiated. In a negotiation round, the price is reduced based on a 10% difference as a buyer offer and then the role switches to seller to approve/disapprove of this buyer offer based on their willingness-to-sell, setting func_tag to negotiate for initiating next round of negotiation. When called by seller, there are three states available: 1. If seller's willingness-to-sell is lesser than or equal to price then accepted and confirm_price() is called by the seller 2. If 4 negotiations (2 cycles of back and forth) done and yet the price differences are not suitable, then terminated 3. Else, seller offer is proposed using the same rules as before to set next_role as buyer for them to approve/disapprove of the new seller offer, assigning func_tag to negotiate for initiation of next round of negotiation to be called. Similar to other methods, negotiation interaction for buyer is added in the grid of seller interactions dictionary and score increased by 5 to update the edge between the buyer and seller performing negotiation action.	None

act	None	Responsible for calling function in order for a complete cycle	The idea of this function is to make sure all the actions are called in sequence and it initiates the stages of actions. The current role and action_type is randomly called to start the transaction. 1. If curr_role is buyer: - If curr_action_type is 2 (previous action) and the model has existing listings, the first action called is view_listings() – mainly for starting a new transaction from scratch from buyer's side. - If curr_action_type is 3 (previous action) but the buyer has pending action, the post with pending action will be extracted and the previous action which was interrupted or left idle will be finished first, calling the previous action method to be called – continuing transaction. - If curr_action_type is 1 (new action) and no existing listings or pending actions then buyer makes a request calling make_buyer_req() – facilitating the seller to act in the transaction. 2. If curr_role is seller: - If curr_action_type is 2 (previous action) and the model has existing requests, the first action called is view_requests() – mainly for start reaction from seller's side. - If curr_action_type is 3 (previous action) but the seller has pending action, the post with pending action will be extracted and the previous action which was interrupted or left idle will be finished first, calling the previous action method to be called – continuing transaction. - If curr_action_type is 1 (new action) and no existing requests or pending actions then seller creates a new post calling create_post() – facilitating the buyer to act in the transaction.	None

Rough drawing for an example run applying all defined methods
Example Run: Buyer 'b' browses the website and wants to buy an item 't'

Fig 5: Application run of methods



1. The act function triggers make_buyer_req() for item 't' and this request is added to b's requests.
2. A seller 's' calls view_requests() and a new post for it is created using create_post().
3. This post is added as a listing so 'b' views the listings via view_listings().
4. Suppose b's willingness to buy is lower than price of 't', there is a trigger to negotiate() from 'b'
5. 'b' suggests an offer to 's', which is assessed by 's' but let's say willingness to sell for 's' is higher so buyer offer is not accepted and an offer is sent by seller 's' for 'b' to assess with a new negotiate() call.
6. The new offer is within the willing to pay so b accepts it within negotiate, calling confirm_price().
7. The buyer gets the final price quotation and the order moves to fulfilment via fulfil() called.
8. The post and request for item 't' is discarded and other variables are updated for the respective buyer 'b' and seller 's'.

Agent Behaviour and Interaction Methods

Agents: Member of the Construcshare website, who can be a buyer, seller or both
Agent State(s): Assessing current members and new members, there is a state of active and inactive users wherein, initially each member is considered active. Also, the agent can be at multiple stages of the transaction flow and the end state is usually a fulfilled transaction or a rejected transaction, which will be explained further as agent behaviours.

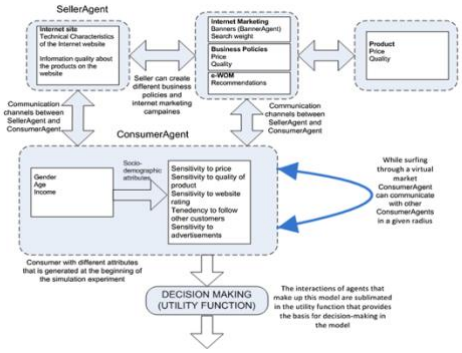
Agent Behaviours: Website interaction and engagement is monitored and the specific behaviours assessed are expressing interest by viewing posts, requests, negotiation, successful transaction. The order of actions considered for interaction are:

1. Expressing interest in post – request is viewed for interaction so value updated by 2
2. Negotiation – value updated by 5
3. Successful purchase – order fulfilled so value updated by 5

Flowchart of agent behaviour:

The methodology followed in this paper set the basis for our project execution where they use agent based simulation to test how different business strategies affect the behaviour of sellers, suppliers and consumers on a B2C sales website. Their source idea for developing the interactions between 'ConsumerAgent' and 'SellerAgent' (as seen in figure 6) gives a detailed analysis factors influencing user behaviour by modelling the actions taken by consumers as they interact with e-commerce platforms. These interactions are fed into a utility function to quantitatively monitor the effects and identify consumers with similar behaviours or needs for targeted strategy planning(Čavoški & Marković, 2017).

Fig 6: Reference Implementation



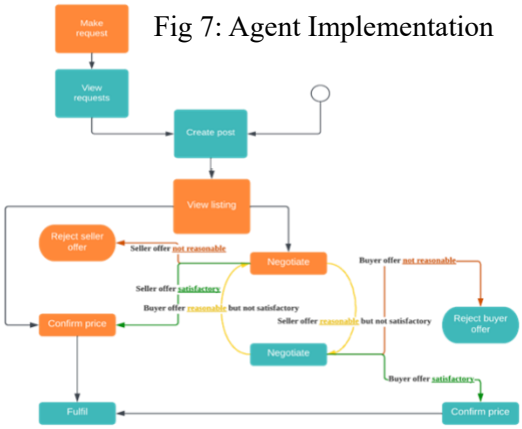


Fig 7: Agent Implementation

Using the above idea, we modelled the actions specific to Construcshare to quantify their interaction instead of modelling all the social behaviours described in the figure 6. Similar to utility function, the interaction calculations were used, where each action has a different utility (interaction score increment) hence a different effect on their engagement. This engagement was shown as a network of agents which can be visually comprehensive rather than just tabular calculations. The figure 7 shows the cycle of completing a transaction involving buyer(orange icons) and seller(blue icons) actions where agent has the freedom to accept(green arrows) or reject(red arrows) or continue negotiating to maximum two cycles (yellow arrows) during a transaction.

Flowchart of all components:

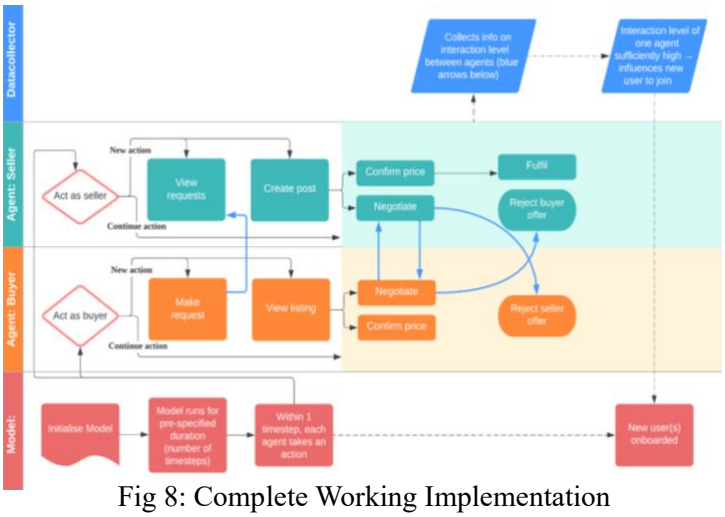


Fig 8: Complete Working Implementation

1. **Model** is initialised with the required 4 inputs(agents, items, platform fee, follower_data) and run some iterations where each agent performs an action and they interact with each other – giving insight into which users interact more and have more chance of bringing a new user to onboard.
2. **Agents** acting as Buyer or Seller and interacting with one another to conduct a transaction leading to fulfilment or rejection
4. **Datacollector** works during every transaction to collect data based on actions like viewing requests, negotiating and successfully completing a transaction (captured parts shown with blue arrows), which add to the interaction values for plotting in the network grid.

Docker & API Component (*model_api.py, Dockerfile, docker-compose.yml, requirements in 'code'*)

The API works using flask application to host the results on port 5000 using mesa container. It extracts the network grid using network_plots() to output two graphs: one using followers data, which can be uploaded using upload(), for initial relationship values, second using actions taken by agent for interaction values and a table, showing the interaction and similarity values over 50 iteration steps.

Running in docker: Firstly, set up working directory to the code folder then use 'docker compose up -d' to get the container up and running. Then call 'export FLASK_APP=model_api' followed by 'flask run' and 4 inputs required to start flask application on http://127.0.0.1:5000

Data processing (Script:*code/DataCleaning.ipynb*)(csv file: *code/Final_code/data/follower_data.csv*)
The main source of data provided were the follower, order and post logs, along with some audit logs however, with repetitions and test entries. Using pandas and numpy, some basic manipulations and column mappings were done then cleaned up using group by, filter, merge and irrelevant columns were dropped to combine only the relevant ones. The pre-processed data table contained 7 unique transactions out of which, the 11 follower user ids were filtered with their respective list of following user ids, which is used as an input for the model. The model can use dynamic data tables to alter the relationship of nodes based on the follower ids, making it suitable for the users to see the difference over time.

Impact of solution for User

Construcshare can use these graphs as an indicator for their future predictions and strategies to gauge the customer behaviour so they can tailor features for each user.

Theoretical Output Network Grid Analysis

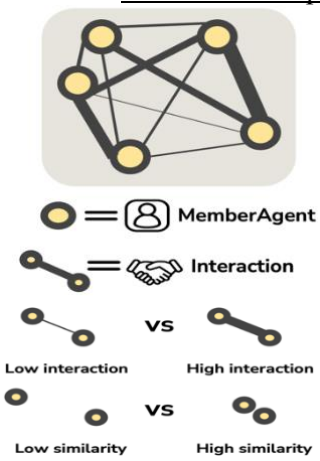


Fig 9: Grid Understanding

Network grid shown as the outcome because the relation between nodes is affected by multiple factors so graph is a better medium to display them.
There are two major aspects shown in the graph below:
Similarity based on Initial Relationship: Using follower user data, if two users followed each other then they are highly related – higher similarity while those that don't will be further from each other as less related – lower similarity. This relationship is updated by value 5 so if one follows another then value added by 5 and if both follow each other (two-way) then value is twice that is 10 is added so higher score, lesser distance between nodes so higher similarity.
Interaction between nodes: Using datacollector and actions taken by agents, the interaction is calculated for each user so if higher interaction value, thicker edge between the nodes while for lower interaction, thinner edge. When a user registers, a thin edge is attached with other agents, and as they perform actions, this edge gets thicker, which differs over time based on their activity on the platform.

Model Results

```
1 as seller making post
post first created s8b1ce2c5-eacb-43bc-8560-bcecc0896a77
2 as seller making post
post first created sa4cd1ea9-74e0-414a-8809-91fe0d748368
1 as buyer has existing listings
post interacted with is s8b1ce2c5-eacb-43bc-8560-bcecc0896a77 (first viewed)
post negotiated
2 as buyer has existing listings
post interacted with is sa4cd1ea9-74e0-414a-8809-91fe0d748368 (first viewed)
post negotiated
1 has pending actions as a 1, next action is <bound method MemberAgent.negotiate of <__main__.MemberAgent object at 0
x7fb90548cfd0>>
buyer info: <__main__.MemberAgent object at 0x7fb90548cfd0> 1
seller info: <__main__.MemberAgent object at 0x7fb90548dbd0> 1
post interacted with is s8b1ce2c5-eacb-43bc-8560-bcecc0896a77
2 as buyer has existing listings
post interacted with is sa4cd1ea9-74e0-414a-8809-91fe0d748368 (first viewed)
post negotiated
1 would have brought in new user
1 as seller making post
post first created sbde51c8a-d2c8-4e41-96f7-a9276eb0e26d
2 as buyer making req
1 would have brought in new user
1 as seller making post
post first created sa4031971-bf0e-4598-8c66-b0e3f71eefda
2 has pending actions as a 1, next action is <bound method MemberAgent.negotiate of <__main__.MemberAgent object at 0
x7fb90548ec80>>
buyer info: <__main__.MemberAgent object at 0x7fb90548ec80> 2
seller info: <__main__.MemberAgent object at 0x7fb90548dea0> 2
post interacted with is sa4cd1ea9-74e0-414a-8809-91fe0d748368
1 would have brought in new user
2 would have brought in new user
```

Step	AgentID	avg_interaction
0	1	0.0
	2	0.0
1	1	0.0
	2	0.0
2	1	0.0
	2	0.0
3	1	2.5
	2	0.0
4	1	2.0
	2	0.0
5	1	1.6
	2	2.5

Fig 10: Intermediate Runs Breakdown by steps

The intermediate results for test_model set using 2 agents, 5 items, run for 5 steps, which traces the transition from one action to another to see how the agents act at each timestep when platform fee is 0.04. The table even shows average interactions over the 5 steps.

The final results that can be used for insights source from the network grid graph and the table shown below (displayed in flask application). The below graphs were generated for 8 items, 5 agents run for 50 iteration steps with platform fee set to 0.02 and the results show the agent interaction values and relationship similarity values are marked on the edges in figures 11 and 12 respectively. In the implementation below, the thickness and similarity distance between nodes cannot be shown visually as seen in figure 9 above so the respective values are rounded and assigned over the edges like figure 11: Interaction values on edges and figure 12: Relationship similarity values on the edges, and their actual calculated values can be seen for each agent and each edge in the figures 13 and 14 respectively.

Fig 11: Agent Interaction Graph

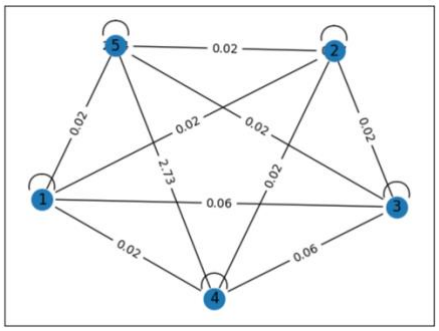


Fig 12: Relationship-Similarity Graph

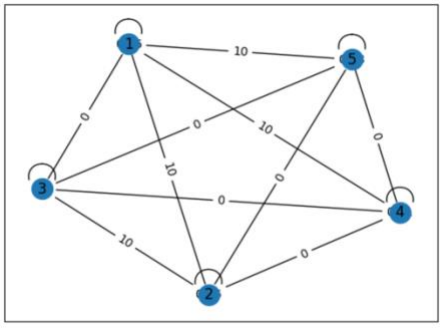


Fig 13: Agent-wise similarity and interaction

avg_interaction

Step	AgentID	
0	1	0.000
	2	0.000
	3	0.000
	4	0.000
	5	0.000
...	...	...
50	1	0.030
	2	0.088
	3	0.040
	4	0.874
	5	1.432

Agent Interactions Summary

agent_A	agent_B	jaccard	interactions
1	2	10.0	0.00
1	3	0.0	0.00
1	4	10.0	0.00
1	5	10.0	0.00
1	6	0.0	0.00
1	7	0.0	0.00
1	8	0.0	0.00
1	9	0.0	0.00
1	10	0.0	0.00
2	3	10.0	0.00
2	4	0.0	0.00
2	5	0.0	0.00
2	6	0.0	0.00
2	7	0.0	0.00
2	8	0.0	0.83
2	9	0.0	0.00
2	10	0.0	0.02
3	4	0.0	0.00
3	5	0.0	0.00
3	6	0.0	0.00
3	7	0.0	0.00
3	8	0.0	0.00
3	9	5.0	0.00
3	10	0.0	2.06
4	5	0.0	0.00
4	6	0.0	0.00
4	7	0.0	0.00
4	8	0.0	0.00
4	9	0.0	0.00
4	10	0.0	0.06
5	6	0.0	0.00
5	7	0.0	0.00
5	8	0.0	0.00
5	9	0.0	0.00
5	10	0.0	0.02

```
test_model.grid.G.edges.data("interaction")
```

```
EdgeDataView([(1, 1, 0.0), (1, 2, 0.02), (1, 3, 0.06), (1, 4, 0.02), (1, 5, 0.02), (2, 2, 0.27), (2, 3, 0.02), (2, 4,
0.02), (2, 5, 0.02), (3, 3, 0.0), (3, 4, 0.06), (3, 5, 0.02), (4, 4, 0.0), (4, 5, 2.73), (5, 5, 2.94)])
```

```
test_model.grid.G.edges.data("jaccard")
```

```
EdgeDataView([(1, 1, 0.86), (1, 2, 10), (1, 3, 0), (1, 4, 10), (1, 5, 10), (2, 2, 0.26), (2, 3, 10), (2, 4, 0), (2,
5, 0), (3, 3, 0.2), (3, 4, 0), (3, 5, 0), (4, 4, 0.91), (4, 5, 0), (5, 5, 0.48)])
```

Fig 14: Edge-wise Interaction and Similarity (jaccard)

These graphs can help drive insights like agents 1 and 4 having prior follow relation (10) caused most interaction (2.73), so these are more likely to have greater activity and more chance of bringing new users to the platform. Between agents 3 & 4 and agents 2 & 4, although both don't have a strong follow relationship with each other, their interaction levels differed, which means there is an external factor like word-of-mouth or historical data connection which has caused a difference.

Value-added for Constructshare

• Regular Monitoring: the platform can monitor the impact of changing a feature on existing users' engagement and user onboarding. Like platform fee, other features can be altered to see the effect.
• Predict & Plan: Based on simulation, the team can deploy the features that draw value without additional spending on research and enhance the speed of decision making.
• Competitive Advantage: Being aware of the 'cause and effect', features more suited to consumer needs and wants based on our low-cost and low-risk simulation model, aligning the logistics and marketing well in time to be ahead of competitors.

Limitations

Project Limitations: The scale of data was not pre-defined; the initial datasets contained mainly test entries and some were irrelevant for the model scope, so internal model testing with synthetic data or arbitrability set values was done as there was no real-time data to validate the model with actual users.
Model Limitations: The graph shows the interactions between fixed number of agents and items, which can only be changed when the model is initialised so the user onboarding for new users can only be seen as an insight on current users but cannot be shown visually in the graph by adding new nodes. Since, the development team at Constructshare rolled out new changes and features, we did not have enough time to model all changes. The element of uncertainty in the project helped us explore different aspects of the model but also brought ambiguity on the model satisfying the need of the company.

Future Considerations and Improvements

1. Once there is enough data available, the model can be changed to show updates in real-time, which is a popular requirement by e-commerce websites. Also, additional data can help in the inclusion of other two objectives of maximising reach and transactions.
2. Improvement in the way the graph edges change with each feature change can be enriched with more parameters in model for a more hands-free and visual experience for users via the native app.
3. Similarity metric can be enhanced from initial follow relationship to using agent's interest during sign up, company domain, inventory, items purchased, etc to define similarity between nodes.
4. Integrating such graph approaches for a recommendation system for buyers and sellers like many popular e-commerce websites so easier navigation across the website.

Appendix: References (APA style)

1. Bokor, O., Florez, L. T., Osborne, A. G., & Gledson, B. (2019). Overview of construction simulation approaches to model construction processes. *Organization, Technology & Management in Construction*, *11*(1), 1853–1861. https://doi.org/10.2478/otmcj-2018-0018

2. Kazil, J., Masad, D., & Crooks, A. (2020). Utilizing Python for Agent-Based Modeling: The Mesa Framework. Lecture Notes in Computer Science, 308–317. https://doi.org/10.1007/978-3-030-61255-9_30

3. Masad, D., & Kazil, J. (2015). Mesa: An Agent-Based Modeling Framework. Proceedings of the Python in Science Conferences. https://doi.org/10.25080/majora-7b98e3ed-009

4. Farooq, M., Shakoor, A., & Siddique, A. (2021). Agent-based modeling and simulation in the analysis of e-commerce market. 2021 International Conference on Frontiers of Information Technology (FIT). https://doi.org/10.1109/fit53504.2021.00060

5. Urdea, A., & Constantin, C. P. (2021). Exploring the impact of customer experience on customer loyalty in e-commerce. Proceedings of the . . . International Conference on Business Excellence, 15(1), 672–682. https://doi.org/10.2478/picbe-2021-0063

6. Inoue, Y., Takenaka, T., & Kurumatani, K. (2019). Sustainability of Service Intermediary Platform Ecosystems: Analysis and Simulation of Japanese Hotel Booking Platform-Based Markets. Sustainability, 11(17), 4563. https://doi.org/10.3390/su11174563

7. Čavoški, S., & Marković, A. S. (2017). Agent-based modelling and simulation in the analysis of customer behaviour on B2C e-commerce sites. Journal of Simulation, 11(4), 335–345. https://doi.org/10.1057/s41273-016-0034-9

8. Code References: https://github.com/projectmesa