LUISS

Department of Business and Management
Teaching: Games and Strategies

Randomized FIFO and Relaxed Rationality in Ride-Sharing Platforms: Design and Performance of Multiple Acceptance Rules

SUPERVISOR

Prof. Xavier Mathieu Raymond Venel

CANDIDATE

Francesca Iannacci

ID: 281431

Academic Year 2024/2025

Abstract

This thesis investigates deviations from perfect rationality within the Randomized FIFO dispatch mechanism developed by Castro et al. (2021), aimed at mitigating driver "cherry-picking" in ride-sharing platforms. Specifically, it analyzes how realistic driver acceptance behaviors—modeled through unconditional, threshold-based, probabilistic, and patience-based heuristics—affect platform efficiency, fairness, queue stability, and revenue outcomes.

To evaluate these behavioral deviations, two event-driven simulations are employed. The first simulation explores the impact of varying acceptance heuristics on system performance compared to the Nash Equilibrium baseline. The second simulation specifically identifies optimal queue dynamics, such as trip partitioning and bin configurations, to maximize efficiency under each behavioral heuristic. Results quantify performance trade-offs introduced by bounded rationality, driver impatience, and simplified decision-making, providing practical insights and actionable guidance for designing robust dispatch mechanisms under realistic constraints.

Acknowledgements

Game theory is one of the courses that most captured my interest in my academic path, primarily thanks to Professor Venel's brilliant teaching. I want to thank him for inspiring me to deepen my knowledge of the subject and to structure my thesis around it. His unwavering support and insightful guidance shaped both the direction and rigor of this study. His expertise in game-theoretic models was invaluable at every stage—from formulating the initial research questions to refining the simulation frameworks and interpreting the results.

I am deeply grateful for my years at LUISS University. Its vibrant academic community nurtured my intellectual curiosity and inspired me to think critically. Through diverse experiences—including international opportunities—I grew both personally and academically into a version of myself I am truly proud to be. I want to thank all my professors for their excellent instruction, constant availability, and support, which enabled me to explore the subjects that truly inspire me and to bring them together in this work.

Lastly, I wish to express my heartfelt gratitude to my family and friends for their constant love and support throughout the years. I owe a large part of my achievements to their encouragement and moral sustenance, which gave me the strength to persevere when everything felt overwhelming. In particular, I am profoundly grateful to my parents, Olga and Michelangelo, whose sacrifices allowed me to pursue this rewarding path and to appreciate the wonders that higher education has revealed to me.

Thank you

Contents

Chapter 1

Introduction

Urban ride-sharing platforms have reshaped the way people move through cities by dynamically matching rider requests with available drivers via smartphone apps. These systems provide advantages such as shorter waiting times, fewer cases of empty vehicle trips, and flexible earning opportunities for drivers. Nevertheless, the provision of these services relies on an advanced dispatch engine that decides which driver will serve each ride request. A dispatch mechanism is an algorithmic rule that considers available drivers and assigns each incoming trip request to one of them. The design of such a system directly affects passenger waiting times, efficiency of vehicle utilization, drivers' earnings, and the financial sustainability of the platform. Therefore, finding an optimal balance between efficiency, fairness, and simplicity is critical to delivering sustainable and quality service.

1.1 Overview

This thesis focuses on the randomized FIFO dispatch mechanism, originally introduced by Castro et al. (2021)[1], designed to reduce the "cherry-picking" behavior that arises under strict First-In-First-Out (FIFO) assignment. Under a strict FIFO system, every incoming trip request is assigned to the driver who has been waiting the longest, regardless of the value or distance of the trip. This

setting can incentivize drivers to cherry-pick—actively decline low-fare or short-distance requests in hopes of getting more lucrative rides—thus resulting in unfulfilled trip requests and operational inefficiencies. The randomized FIFO method overcomes this by grouping drivers into probabilistic "bins" and allowing the interleaving of offers among drivers with longer and shorter wait times, rather than imposing a strict chronological ordering by waiting time. We begin by characterizing the Nash equilibrium for this theoretical dispatch (Castro et al., 2021[1]), and then introduce a range of bounded-rational acceptance behaviors that better model the actual decision-making processes of drivers in the real world. We simulate these deviations, compare them against the equilibrium, and clarify the performance of various queue configurations in terms of throughput, net revenue, and fairness when drivers use non-optimally rational decision-making rules.

1.2 Structure

The structure of this thesis is as follows: Chapter 2 formalizes the mathematics underlying our analysis, including $M/M/c$ queueing formulations, Erlang–C formulas, basic concepts of Game Theory, and concepts related to Nash equilibrium; Chapter 3 defines and formally introduces three dispatch mechanisms—Strict FIFO, Direct FIFO, and Randomized FIFO—and analyzes their equilibrium properties, summarizing the key theoretical insights from Castro et al. (2021)[1]; Chapter 4 describes our first discrete-event simulation, which examines how each acceptance rule behaves under the Randomized FIFO mechanism and highlights the trade-offs that arise; Chapter 5 then presents the second simulation framework used to benchmark these acceptance rules against the Nash Equilibrium by testing every possible combination of trip partitions and queue segmentations to explore how different Randomized FIFO structures impact efficiency under each rule; Chapter 6 concludes with a synthesized overview of findings and discusses practical implications for platform design.

Chapter 2

Mathematical Background

Ride-sharing platforms operate at the intersection of demand and supply dynamics, where passenger requests must be matched with available drivers in real-time under varying market conditions. To analyze and optimize these interactions, it is essential to employ formal mathematical models that can accurately represent system behaviors, predict equilibrium outcomes, and inform decision-making processes. This chapter presents the theoretical foundations necessary to understand the Randomized FIFO dispatch model, focusing on the underlying queuing structures and the game-theoretic modeling of strategic behavior among drivers operating within dispatch mechanisms.

2.1 Chapter overview

Chapter 2 introduces the theoretical tools that support the modeling work developed later in the thesis. Section 2.2 focuses on queueing theory, presenting the $M/M/c$ model and key metrics needed to evaluate system performance. This material will later inform our analysis of rider and driver dynamics. Section 2.3 turns to game theory, covering fundamental concepts such as Nash equilibrium and subgame-perfect equilibrium. Together, these sections equip us with the

frameworks necessary to formulate and analyze the dispatch dynamics addressed in the following chapters.

2.2 Queueing theory: essential metrics and concepts

Queueing theory tackles one of the most common challenges of daily life: the experience of waiting. The formal study of queueing systems began in the early 20th century with Danish mathematician and engineer Agner Krarup Erlang, who introduced probabilistic methods to analyze congestion in telephone networks.[1] His groundbreaking work laid the foundation for the broader field of queueing theory, inspiring the development of increasingly sophisticated models to manage delays and improve service efficiency. Later in this chapter, we will reference a queueing theory framework known as the Erlang-C or $M/M/c$ model.

A queueing model possesses the characteristics listed below.[2] These attributes are at the core of the dispatch mechanisms analyzed in Chapter 3, including Randomized FIFO.

The arrival process of customers. It is commonly assumed that interarrival times are independent and identically distributed (i.i.d.). In many real-world systems, customers arrive at service facilities in a random fashion. This variability in arrivals is often well-modeled by a Poisson process, where the time between arrivals follows an exponential distribution. Therefore, the probability that the time between two consecutive arrivals is less than or equal to T, given a Poisson arrival process with rate λ is:

$$P(t \leq T) = 1 - e^{-\lambda T}. \tag{2.1}$$

Here, λ represents the average number of arrivals per unit of time. Since the Poisson distribution is discrete, it also provides the probability of observing an exact number of arrivals within a set period,

[1] Erlang, A. The theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik B 20* (1909), 33–39.

[2] Adan, Ivo, and Jacques Resing. *Queueing Systems*, Department of Mathematics and Computing Science Eindhoven University of Technology P.O. Box 513, 5600 MB Eindhoven, The Netherlands, 26 Mar. 2015.

$$P(X = n) = \frac{(\lambda T)^n}{n!} e^{-\lambda T},\ n = 0,1,2,... \tag{2.2}$$

In this context, n denotes a non-negative integer representing the specific number of occurrences for which we calculate the probability. λT is the Poisson mean (or expected value), indicating the average number of occurrences within the given time interval. A Poisson process is characterized by three fundamental assumptions:

1. Customers arrive individually, not in groups.
2. Each arrival is independent of all the others.
3. The likelihood of an arrival is uniform across time—it does not vary depending on the time of observation.

The behavior of customers. Customers can vary in their willingness to wait—some may remain in the queue indefinitely, while others may abandon it after a certain period due to impatience. For example, a person calling a customer service center might decide to hang up and try again later if the wait time becomes too long. In our simulation, this customer behavior, specifically in the context of drivers, is modeled and adjusted through what we define as the "acceptance rule".

The service times. We assume that service times are independent and identically distributed (i.i.d.) and independent from the interarrival times. However, in some systems, service times may vary depending on the current queue length. For instance, in a production environment, machines might operate at higher speeds when the backlog of jobs becomes excessive. The exponential distribution is often used to model service times, capturing the probability that a service will be completed within a given time interval T. The probability can be calculated by using the following expression:

$$P(t \leq T) = 1 - e^{-\mu T}. \tag{2.3}$$

Here, μ denotes the average service rate—that is, the number of customers served per unit of time. The variable t represents the service duration for an individual customer, and T is the time threshold we are evaluating.

The service discipline. The service discipline defines how the system manages its resources, including the number of servers and the system's capacity, the maximum number of customers allowed in the system at any time, including those currently being served. It also specifies the rule used to determine which customer is selected next for service. Common service disciplines include:

- **FIFO (First In First Out)**: customers are served in the order they arrive.
- **LIFO (Last Come First Out)**: the most recently arrived customer is served first.
- **RS (Random Service)**: the next customer is chosen at random.
- **PS (Processor Sharing)**: service capacity is equally shared among all active customers—commonly used in computing systems.
- **Priority-Based**: customers with higher priority (e.g., urgent requests or shorter service times) are served before others.

The service capacity. There may be either a single server or multiple servers simultaneously providing service to customers.

The waiting room. Queueing systems may have constraints on the number of customers that can wait within the system. For instance, in a small coffee shop, only a certain number of people can line up inside before the space becomes full, forcing others to wait outside or leave.

2.2.1 Kendall's notation

This notation will be particularly helpful in our discussion of the queueing model underlying Randomized FIFO. The framework was introduced by mathematician David G. Kendall to classify a wide variety of queueing systems. We represent a queueing system using the notation:

$$A \,/\, B \,/\, m \,/\, K \,/\, n \,/\, D$$

Each element in the system notation is defined in the following way. A represents the distribution of interarrival times, while B describes the distribution of service times. The parameter m indicates the number of servers, and K refers to the total system capacity, meaning the maximum number of customers allowed in the system, including those being served. The variable n specifies the

population size, or the total number of potential customers. Finally, D denotes the service discipline, such as FIFO, LIFO, or RS.

The first two positions describe the statistical distributions of interarrival and service times. Common abbreviations include D (Deterministic distribution), M (Markovian—Poisson arrivals or Exponential services), G (General), GI (General and Independent), Geom (Geometric). The fourth position indicates the total capacity, including service points and buffer space. For instance, if there are K servers and no additional waiting area, K appears in this position. If the queue has unlimited capacity, this element is typically omitted. The sixth position denotes the queue discipline. It is only included when the discipline is something other than the default FIFO.[3]

2.2.2 The $M/M/c$ queue

We now return to the work of mathematician A.K. Erlang to explore the Erlang-C model. This model describes a system in which customers arrive, with a constant rate, at a queue served by c identical servers and are assumed to have infinite patience. Given that arrivals follow a Poisson process and service times are exponentially distributed, the model is denoted as $M/M/c/\infty$, or more commonly, $M/M/c$. To better understand the need to extend a single-server queueing model to a multi-server one—such as the one we analyze in our ride-sharing context—we introduce the concept of server utilization, denoted by ρ. This metric, also called the occupation rate, is defined as the ratio between the mean service time and the mean interarrival time:

$$\rho \;=\; \frac{\text{mean service time}}{\text{mean interarrival time}}\,.$$

In the context of ride-sharing, ρ is referred to as traffic intensity and indicates the fraction of time a driver is actively engaged in serving riders. Assuming an infinite population and a Poisson arrival process with rate λ, the mean interarrival time becomes $1/\lambda$, and the mean service time $1/\mu$.[4] Substituting into the formula, we obtain for a single-server queue:

[3] Sztrik, János, et al. *Basic Queueing Theory*, University of Debrecen, Faculty of Informatics, Dec. 2012.

[4] If λ is the average number of arrivals per time unit, then on average one arrival happens every $1/\lambda$ time units. The same logic applies to μ: if a server can handle μ customers per unit time, each service takes on average $1/\mu$ time units.

$$\rho \;=\; \lambda \;\cdot\; \frac{1}{\mu} = \frac{\lambda}{\mu}\,. \tag{2.4}$$

If $\rho > 1$, the system is considered overloaded, meaning requests are arriving faster than they can be handled by a single server. This indicates a need for additional service capacity (e.g., more servers) to maintain system stability. In our scenario, an increase in rider arrivals is necessary to match driver availability and ensure timely service.[5] Hence, we refer to the $M/M/c$ queue to build an initial understanding of the system dynamics we will explore in the chapters that follow. Under $M/M/c$, indicators such as utilization, throughput, waiting time, and queue length can be derived analytically. However, in our study, the system behavior deviates from the classic multi-server queue, and we rely on simulation to assess system performance. We now shift our focus to the key performance metrics of an $M/M/c$ queue.[6]

Occupation rate. The traffic intensity previously defined for a single-server queue decreases when more servers are available, improving the system's overall efficiency. Therefore, we have:

$$\rho = \frac{\lambda}{c\;\mu}\,. \tag{2.5}$$

For the system to remain stable and avoid becoming overloaded, it is required that $\rho < 1$, meaning the arrival rate of customers must not exceed the total service capacity of all workers per unit of time.

Steady state probabilities. The state of the system is characterized by the number of customers in the system. We denote with p_n the equilibrium probability that there are n customers in the system. These probabilities are typically determined by modeling the system as a continuous-time Markov chain, where each state represents the number of customers present, and transitions occur based on arrival and service rates. These probabilities lay the basis for deriving the following performance metrics.

$$P_0 = \left[\; \sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{c!\,(1-\rho)} \;\right]^{-1} \tag{2.6}$$

[5] While it is logically reasonable to define drivers as servers, in the context of our model, the queueing dynamics analyzed in the following chapters focus on the queue of drivers. Therefore, we adopt a reversed perspective: drivers are treated as customers in the system, and riders are considered the servers.

[6] Fiveable. "8.3 *$M/M/1$* and $M/M/c$ queues – Stochastic Processes." Edited by Becky Bahr, Fiveable, 2024.

$$P_n = \frac{(c\rho)^n}{n!}\ P_0, \quad 0 < n < c \tag{2.7}$$

$$P_n = \frac{(c\rho)^n}{c!\, c^{n-c}}\ P_0, \quad n \geq c \tag{2.8}$$

Probability of waiting (Erlang-C). The Erlang's C formula, also known as Erlang's delay formula, expresses the probability that an arriving customer must wait for service. A lower value of this probability indicates a more efficient and responsive system.

$$P_w = \frac{\frac{(c\rho)^n}{n!}\frac{n}{n-(c\rho)}}{\sum_{k=0}^{n-1}\frac{\rho^k}{k!} + \frac{(c\rho)^n}{n!}\frac{n}{n-(c\rho)}} \tag{2.9}$$

Average number of customers in the queue. We define the expected number of customers in the queue as the average number of customers waiting to be served.

$$L_q = \frac{P_0(c\rho)^n\rho}{n!\,(1-\rho)^2} \tag{2.10}$$

A lower value of L_q indicates that customers experience shorter waits before receiving service, contributing to higher system efficiency and customer satisfaction.

Average number of customers in the system. We now introduce the expected number of customers in the system, L, which accounts for both those waiting in the queue (L_q) and those currently being served. This measure is directly connected to Little's Law, a fundamental relationship in queueing theory, which states that the average number of customers in the system equals the arrival rate (λ) multiplied by the average time a customer spends in the system (W).[7] At this stage, we express L in the following way:

$$L = L_q + \frac{\lambda}{\mu}\,, \tag{2.11}$$

which is simply a reformulation of Little's Law,

$$L = \lambda W. \tag{2.12}$$

[7] We will delve deeper into Little's Law in the upcoming chapters, as it forms the foundation for many of the models we study.

A lower value of this metric indicates that the system is less congested, resulting in shorter delays for customers. Unlike L_q, the average number of customers in the system offers a more comprehensive view of system congestion.

Expected waiting time in the queue. The expected waiting time in the queue, W_q, measures the average amount of time a customer spends waiting in the queue before receiving service. The formula for this metric is a direct application of Little's Law to the queue component of a system ($L_q = \lambda W_q$). It is expressed as:

$$W_q = \frac{L_q}{\lambda} . \tag{2.13}$$

As one would expect, a lower value of W_q reflects a more efficient queueing system, with customers experiencing reduced waiting times. Moreover, if a customer abandonment mechanism were introduced—an assumption outside the standard $M/M/c$ framework where customer patience is infinite—a lower W_q would also contribute to minimizing customer abandonment rates, thus enhancing overall system performance.

Expected waiting time in the system. The expected waiting time in the system, W, represents the total amount of time a customer spends between entering the system and completing service. It combines both the waiting time in the queue and the service time itself. This expression is directly derived from Little's Law—$L = \lambda W$, previously introduced when discussing the average number of customers in the system. Accordingly, we have:

$$W = \frac{L}{\lambda} = \frac{L_q}{\lambda} + \frac{1}{\mu} = W_q + \frac{1}{\mu} . \tag{2.14}$$

In essence, we are summing the average time a customer spends waiting in the queue with average service time. A smaller value of this metric translates into shorter delays for customers throughout the system. Therefore, minimizing W leads to a more efficient, responsive, and satisfying service experience.

While the metrics derived from the $M/M/c$ model provide essential insights into system performance, they are not fully sufficient for capturing the dynamics of our specific scenario. In

our case, we introduce a fixed abandonment mechanism for rider patience, meaning that customers may leave the system if their wait exceeds a certain threshold. Therefore, while many of the multi-server queue results remain highly informative, they will be carefully adapted to account for this additional behavioral dimension.

2.3 Game theory: principles and equilibrium concepts

Game theory offers a unified framework for modeling strategic interactions. It examines situations of conflict, the interactions between agents, and the decisions they make. A game is characterized by a set of players (typically finite) and the strategies available to them within given rules.[8] The players in a game engage with one another in an interdependent manner—the choices made by one player influence not only their own outcome but also the outcomes of others. Hence, game theory studies situations where a player's outcome is determined not only by their own choices but also by the decisions of other players. Game theory helps us analyze how drivers and riders in a ride-sharing queue make strategic accept-or-decline decisions—modeling each driver's acceptance rule as a strategy and predicting the equilibrium outcomes (i.e., throughput, wait times, payoff distributions) of different dispatch mechanisms. We begin with a concise overview of game-theory fundamentals.

2.3.1 Game structures and information dynamics

Normal form (strategic) games. A game in normal form is a formal representation of a strategic situation where all players choose their actions simultaneously and independently . It is defined by the following components (Hotz H., 2006):

1. A finite set of players $M = \{a_1, ..., a_n\}$.
2. For each player, a set of possible actions or strategies $S_i,\ i \in M$.

[8] Hotz, Heiko. A Short Introduction to Game Theory, LMU Munich, 2006, www.theorie.physik.uni-muenchen.de/lsfrey/teaching/archiv/sose_06/softmatter/talks/Heiko_Hotz-Spieltheorie-Handout.pdf.

3. A payoff function (u_i) for each player ($i \in M$), mapping every profile of strategies into the reward (or cost) that player receives based on their own strategy and the choices made by others.

When the game involves only two players and each has a limited set of strategies, the payoffs can be conveniently represented in a payoff matrix. This matrix visually displays the players, their available strategies, and the corresponding payoffs for each combination of choices. In the example below (Table 1), Player 1 chooses between two strategies: "T" (Top) or "B" (Bottom), while Player 2 selects either "L" (Left) or "R" (Right). The resulting payoffs for each combination of strategies are displayed within the cells of the table.

Player 1\Player 2	L	R
T	(2, 5)	(3, 7)
B	(2, 0)	(5, 5)

Table 1: A normal form game

Each payoff is a pair (x, y), where x is the payoff for Player 1 and y is the payoff for Player 2. For instance, if Player 1 selects T and Player 2 selects R, the resulting outcome is (3, 7), meaning Player 1 receives a payoff of 3, while Player 2 receives a payoff of 7.

Extensive form games. Unlike a normal form game, where players choose their strategies simultaneously, an extensive form game is structured so that players make their moves sequentially. This type of game is represented using a game tree, where each node corresponds to a specific stage. The initial node marks the beginning of the game, while terminal nodes, which have only one connected edge, signify the end of the game and define a complete strategy profile. Each non-terminal node is assigned to a particular player, indicating that it is their turn to make a decision. The edges connecting the nodes represent the possible actions available at each decision point. At the end of the game, each terminal node is associated with a payoff for every player, reflecting the outcome if the sequence of actions leading to that node is followed.

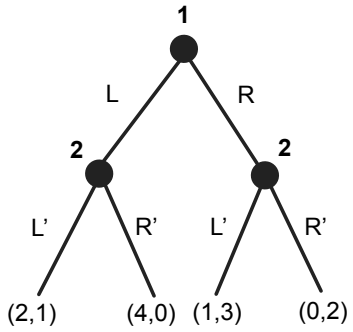


Figure 1: An extensive form game

In the figure, if Player 1 chooses strategy L and Player 2 selects strategy R', the resulting payoff will be 4 for Player 1 and 0 for Player 2.

In analyzing strategic interactions, it is important to distinguish between games based on the information available to the players.

- **Perfect Information.** A game is said to have perfect information if, at every decision point, each player knows the full history of previous moves; classic examples include chess and checkers.
- **Imperfect Information.** In a game of imperfect information, players must make decisions without full knowledge of past actions, as in simultaneous-move games where players choose strategies at the same time.
- **Complete Information.** A game involves complete information if all players are sully aware of the structure of the game, including the available strategies and payoff functions of all participants.
- **Incomplete Information.** If any of the information necessary to have a complete information game is missing or uncertain, the game is classified as one of incomplete information, often requiring players to form beliefs about known elements.

2.3.2 Strategic stability: Nash and Subgame-perfect equilibria

Understanding how players' strategies interact and stabilize is central to analyzing game-theoretic models. Two key solution concepts help capture this idea of stability: the Nash equilibrium and the Subgame-perfect equilibrium. While a Nash equilibrium ensures that no player has an incentive

to deviate given the strategies of others, Subgame-perfect equilibria strengthen this notion by requiring credibility and optimality at every stage of the game. In particular, we delve into Subgame-perfect equilibrium because it is a fundamental concept for analyzing extensive form games—such as the dynamic scenario studied in our case—where decisions unfold over time. In this section, we introduce and compare these concepts to better understand strategic stability in both simultaneous and sequential games.

Nash Equilibrium

In a strategic game, each player is assumed to act rationally, choosing the best available action based on their expectations about the actions of others. Players must, therefore, form beliefs about how their opponents will behave. These beliefs are shaped by players' prior experiences in playing the game, which are assumed to be extensive enough for them to accurately anticipate others' actions. Importantly, players treat each game as an isolated event: they do not adapt their strategies based on familiarity with particular opponents, nor do they expect current actions to influence future behaviors. We consider a structured setting where players face a wide and changing pool of opponents, selected randomly for each play. This repeated interaction fosters beliefs about the behavior of "typical" opponents rather than specific individuals. At its core, the framework is built on two key components: (1) players act rationally based on their beliefs about others, and (2) players' beliefs about others' actions are correct.

A *Nash equilibrium* is an action profile a^* where no player i can improve their outcome by unilaterally deviating from their chosen action a_i^*, assuming other player j adheres to their strategy a_j^* (Osborne M. J., 2000).

In a setting where players are drawn randomly from populations, a Nash equilibrium represents a steady state; if players repeatedly engage in the game and the same action profile, a^*, consistently arises, no player has any incentive to change it. Another important aspect of Nash equilibrium is the assumption that players' beliefs about each other's actions are accurate. Because of this, Nash equilibrium is often described as a situation where players' expectations are coordinated.

We let a represent an action profile where each player i chooses an action a_i. Representing with (a_i', a_{-i}) the action profile in which all players implement strategy a while player i deviates to

strategy a_i'. If $a_i' = a_i$, there is no deviation, and $(a_i', a_{-i}) = a$. Using this notation, we provide the formal definition of a Nash equilibrium.

Definition 1 (*Nash equilibrium of strategic game with ordinal preferences*; Osborne, M. J. (2000), An Introduction to Game Theory, Ch. 2, Def. 2.1, p. 21). An action profile a^* in a strategic game with ordinal preferences is a Nash equilibrium if, for every player i and every alternative action a_i available to player i, a^* is at least as preferred by player i as the action profile (a_i, a^*_{-i}), where player i unilaterally deviates to a_i while every other player j continues to play a^*_j. Formally, for every player i,

$$u_i(a^*) \geq u_i(a_i, a^*_{-i}) \text{ for every action } a_i \text{ of player } i, \tag{2.15}$$

where u_i denotes the payoff function that represents player i's preferences.[9]

While Nash equilibrium is a key concept in strategic games, it does not ensure existence or uniqueness—some games have one, many, or no equilibria. Traditionally, it models steady-state behavior among experienced players, but an alternative view sees it as the outcome of rational players deducing others' actions without prior experience.

To properly analyze the strategic interactions in dynamic settings, we extend the concept of Nash equilibrium to extensive form games. Since the scenario we study can be modeled as a dynamic, extensive form game with perfect and complete information, it is necessary to characterize equilibrium strategies across the entire sequence of moves. In this context, a Nash equilibrium specifies a strategy profile where each player's choices are optimal, given the observed history and the strategies of others.

Definition 2 (*Nash equilibrium of extensive game with perfect information*; Osborne, M. J. (2000), An Introduction to Game Theory, Ch. 5, Def. 159.2, p. 159). A strategy profile s^* in an extensive game with perfect information is a Nash equilibrium if, for every player i and every alternative strategy r_i of player i, the terminal history $O(s^*)$ generated by s^* is at least as preferred by player

[9] Osborne, Martin J. *An Introduction to Game Theory by Martin J. Osborne*, 6 Nov. 2000, mathematicalolympiads.wordpress.com/wp-content/uploads/2012/08/martin_j-_osborne-an_introduction_to_game_theory-oxford_university_press_usa2003.pdf.

i as the terminal history $\mathrm{O}(r_i, s^*_{-i})$ generated by the strategy profile (r_i, s^*_{-i}) where player i deviates to r_i while every other player j continues to follow s^*_j. Formally, for every player i,

$$u_i(\mathrm{O}(s^*)) \geq u_i(\mathrm{O}(r_i, s^*_{-i})) \text{ for every strategy } r_i \text{ of player } i, \tag{2.16}$$

where u_i is a payoff function that represents player i's preferences and O is the outcome function of the game.

Subgame-perfect Equilibrium

The notion of Nash equilibrium does not account for the sequential nature of extensive games. As a result, the steady states captured by Nash equilibrium may not always be robust when decisions are made over time. To better model sequential decision-making, we introduce a stronger concept: a solution that requires players' strategies to be optimal after every possible history, not just at the beginning of the game. To define this refinement, we first introduce the idea of a subgame.

Definition 3 (*Subgame*; Osborne, M. J. (2000), An Introduction to Game Theory, Ch. 5, Def. 162.1, p. 162). Let Γ be an extensive game with perfect information and player function P. For any nonterminal history h, the subgame $\Gamma(h)$ following the history h is defined as the following extensive game.

> *Players.* The players in Γ.
>
> *Terminal histories.* All action sequences h' such that (h,h') forms a complete terminal history in Γ.
>
> *Player function.* Each proper sub-history h' is assigned to a player according to $P(h,h')$.
>
> *Preferences.* Players' preferences are consistent with their preferences over the corresponding full histories (h,h') in Γ.

Put it simply, for any nonterminal history h, the subgame following h refers to the part of the game that remains after h occurs. The subgame following the initial empty history $\emptyset$ is simply the entire game. All other subgames are referred to as proper subgames. Since every nonterminal history defines a subgame, the number of subgames equals the number of nonterminal histories.

To formally define subgame-perfect equilibrium, we introduce new notation. Let h be a history and s a strategy profile. If h occurs—even if it may not align with s—players thereafter follow the strategy profile s. We denote by $\mathrm{O}_h(s)$ the terminal history resulting from h followed by actions according to s. Notably, when h is the initial history, $\mathrm{O}_\emptyset(s)$=$\mathrm{O}(s)$.

Definition 4 (*Subgame perfect equilibrium*; Osborne, M. J. (2000), An Introduction to Game Theory, Ch. 5, Def. 164.1, p. 164). A strategy profile s^* in an extensive game with perfect information is called a subgame perfect equilibrium if, for every player i and every history h where it is player i's turn to move ($P(h) = i$), the following holds: the terminal history $\mathrm{O}_h(s^*)$ resulting from following s^* after h is at least as good for player i as the terminal history resulting from any deviation r_i by player i, with all other players sticking to s^*. Formally,

$$u_i(\mathrm{O_h}(s^*)) \geq u_i(\mathrm{O_h}(r_i, s^*_{-i})) \text{ for every strategy } r_i \text{ of player } i, \tag{2.17}$$

Here, u_i represents player i's payoff function, and $\mathrm{O}_h(s)$ denotes the sequence of actions starting from history h under strategy profile s.

The key idea is that each player's strategy must be optimal not only at the beginning of the game but after every possible history where the player is called to move. This ensures that strategies are credible and robust throughout the entire game. Given the definitions of Nash equilibrium and Subgame-perfect equilibrium, we now introduce a fundamental result

> A Subgame-perfect equilibrium is a strategy profile that induces a Nash equilibrium in every subgame (Osborne M. J., 2000).

2.4 Summary and transition

In this chapter, we introduced the mathematical background necessary for the analysis of the dispatch mechanisms. We began by presenting key elements of queueing theory, with a focus on the $M/M/c$, multi-server model and its associated performance metrics. We also discussed the limitations of the classical model for our purposes, noting the need to account for fixed abandonment behavior to better reflect rider patience.

In the second part of the chapter, we turned to game theory. We reviewed fundamental concepts such as Nash equilibrium and Subgame-perfect equilibrium, emphasizing their role in capturing strategic decision-making in dynamic settings. Although these concepts were introduced in a general framework, without direct application to drivers and riders, they will be essential in modeling the strategic interactions we explore in the next chapters.

The mathematical tools discussed here provide the foundation for the analysis that follows. We are now ready to formally define the dispatch models under study and examine their equilibrium properties.

2.4.1 Symbols and formulas

We next summarize the main symbols and mathematical expressions introduced throughout the chapter. These definitions provide a concise reference for the core concepts discussed, ranging from queueing system metrics to equilibrium conditions in strategic and extensive form games.

Queueing Theory—$M/M/c$ queue

λ	Arrival rate (customers per unit time)
μ	Service rate (customers served per unit time)
c	Number of servers
L	Expected number of customers in the system
L_q	Expected number of customers in the queue
W	Expected waiting time in the system
W_q	Expected waiting time in the queue
$\rho = \dfrac{\lambda}{c\mu}$	Traffic intensity
$P_0 = \left[\sum_{n=0}^{c-1} \dfrac{(c\rho)^n}{n!} + \dfrac{(c\rho)^c}{c!\,(1-\rho)} \right]^{-1}$	Probability of having zero customers in the system
$P_n = \begin{cases} \dfrac{(c\rho)^n}{n!}\ P_0, & 0 < n < c \\ \dfrac{(c\rho)^n}{c!\,c^{n-c}}\ P_0, & n \geq c \end{cases}$	Probability of having n customers in the system
$P_w = \dfrac{\dfrac{(c\rho)^n}{n!}\dfrac{n}{n-(c\rho)}}{\sum_{k=0}^{n-1}\dfrac{\rho^k}{k!} + \dfrac{(c\rho)^n}{n!}\dfrac{n}{n-(c\rho)}}$	(Erlang-C) Probability that an arriving customer has to wait
$L_q = \dfrac{P_0(c\rho)^n\rho}{n!\,(1-\rho)^2}$	Average number of customers in the queue
$L = L_q + \dfrac{\lambda}{\mu}$	Average number of customers in the system
$W_q = \dfrac{L_q}{\lambda}$	Average waiting time in line
$W = W_q + \dfrac{1}{\mu}$	Average time spent in the system

Game Theory

M	Set of players
S_i	Set of strategies available to player i
$s = (s_1, \ldots, s_n)$	Strategy profile (one strategy for each player)
w_i	Payoff function for player i
a_i	Action chosen by player i
a_{-i}	Actions of all players other than i
a^*	Nash equilibrium action profile
s^*	Subgame-perfect equilibrium strategy profile
Γ	Extensive form game
$\Gamma(h)$	Subgame following history h
$\mathrm{O}(s)$	Outcome of the game when strategy profile s is played
$\mathrm{O}_h(s)$	Outcome of the subgame starting at history h under strategy s

Chapter 3

Dispatch Frameworks

Ride-sharing platforms rely on efficient dispatch mechanisms to match drivers with riders while balancing fairness, reliability, and revenue optimization. Traditional First-In-First-Out (FIFO) dispatching, ensures that drivers who have waited the longest are prioritized. However, heterogeneous trip earnings incentivize drivers to cherry-pick (i.e., accepting only higher-fare trips while skipping lower-value ones), leading to longer rider wait times and inefficient matches. In the article "*Randomized FIFO Mechanisms"*, Francisco Castro, Hongyao Ma, Hamid Nazerzadeh, and Chiwei Yan propose a family of dispatch mechanisms that blend FIFO priority with controlled randomness.[10] This chapter explores the following as presented in the article: Strict FIFO, Direct FIFO, and Randomized FIFO mechanisms. We analyze how each variant shapes driver behavior, influences platform earnings, and alters system-wide performance. In Chapter 4 and Chapter 5, we simulate the Randomized FIFO framework under more realistic conditions—specifically when drivers are not flawlessly strategic agents in a dynamic game of perfect and complete information.

[10] Castro, Francisco, et al. "Randomized FIFO Mechanisms." arXiv.Org, 21 Nov. 2021, arxiv.org/abs/2111.10706.

3.1 Problem formulation and model design

The article studies dispatch strategies for ride-sharing at high-demand locations (like airports), where trips differ significantly in earnings and riders have limited patience. Because platforms can't easily adjust fares, they rely on driver waiting times as incentives to improve efficiency and fairness. Randomized FIFO aims to optimize the allocation of drivers to rider requests while maximizing the platform's net revenue.

A continuous-time, non-atomic queueing model is developed, with steady arrivals of drivers and riders. Riders request trips with varying earnings, may cancel after limited rejections, and drivers strategically balance waiting costs against trip earnings. The "first-best" solution, achievable without driver strategic behavior, serves as a performance benchmark. Strict FIFO dispatching—offering trips to the longest-waiting driver—results in strategic "cherry-picking", leaving lower-paying trips unfulfilled, causing excessive wait times and revenue loss. Two alternative mechanisms address this. Direct FIFO sends lower-value trips directly to drivers further back in the queue, improving throughput but raising fairness concerns, as drivers who waited longer might miss out on trip offers. Randomized FIFO dispatches trips probabilistically within segments of the queue, encouraging acceptance while preserving fairness, maximizing revenue, and reducing income disparities. The study highlights queue-based incentive mechanisms as powerful tools for balancing fairness, reliability, and revenue without altering prices or imposing rigid penalties.

3.1.1 Model setup and assumptions

The study examines a continuous-time, non-atomic queueing model for dispatching trips from a single location (e.g., an airport). Riders request trips continuously to multiple destinations, represented as $\mathcal{L} = \{1, 2, \ldots, \ell\}$. Each destination $i \in \mathcal{L}$, has an arrival rate of riders of $\mu_i > 0$ and an earnings value of w_i. Riders cancel requests after being declined P times, reflecting limited patience. Drivers arrive at a rate of $\lambda > 0$, strategically balancing trip earnings against opportunity costs $c > 0$ incurred while waiting in queue; the platform also incurs an opportunity cost of $c_p \in [0, c]$.

The platform dispatches trip requests sequentially to drivers in a queue, who may accept or decline offers. A declined trip continues to be dispatched until it's either accepted by another driver, canceled by the rider after P rejections, or withdrawn by the platform. The platform fully and transparently shares information on demand, supply, earnings, and opportunity costs with drivers. Drivers know their queue position, can decline trips without penalty, and are free to leave, rejoin, or exit the queue altogether. Under ideal conditions—no driver strategic behavior and perfect platform control—an optimal dispatch solution, the first-best, prioritizes trips in descending order of earnings to maximize revenue and throughput. The optimal set is determined by identifying the lowest-earning trip type dispatched, denoted as:

$$i^* = \max \left\{ i \in \mathcal{L} \mid \lambda > \sum_{j=1}^{i-1} \mu_j \right\} \tag{3.1}$$

An optimal platform strategy does not maintain a non-zero driver queue. It dispatches drivers upon arrival to destinations in decreasing order of w_i until all riders are served or all drivers are utilized. The platform's trip throughput is the mass of trips completed per unit time, while its net revenue is the total net earnings from completed trips minus the opportunity costs due to waiting drivers.

Proposition 1 (The first best; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 2, p. 8, Proposition 1). The steady state first best outcome has zero drivers in the queue. The first best trip throughput is represented by

$$T_{FB} = \min \left\{ \lambda, \sum_{i \in \mathcal{L}} \mu_i \right\} \tag{3.2}$$

If driver supply is greater than demand, then all rider requests are fulfilled, and the throughput is simply $\sum_{i \in \mathcal{L}} \mu_i$. However, if driver supply is lower than demand, the throughput is constrained by λ, meaning some riders will not be served. The first-best net revenue represents the total earnings from completed trips after accounting for driver constraints. The formula

$$R_{FB} = \sum_{i \in \mathcal{L}} \mu_i \sum_{i=1}^{i^*-1} w_i \mu_i + w_{i^*} \min \left\{ \lambda - \sum_{j=1}^{i^*-1} \mu_j, \mu_{i^*} \right\}, \tag{3.3}$$

breaks revenue into two components. The first term accounts for fully dispatched trips, where all requests for trip types 1 through $i^* - 1$ are fulfilled. The second term addresses the lowest-priority trip i^*, where only a fraction of requests may be served, depending on the remaining available drivers.

3.2 Mathematical formulation of the three models

3.2.1 Strict FIFO: default queue-based dispatching

Strict FIFO is generally perceived as fair because it ensures that each driver gets a chance to receive a trip in the order they arrived. However, when drivers have the flexibility to decline trips, they may choose to wait for more profitable ones, leading to cherry-picking. This behavior can result in poor outcomes for riders, drivers, and the platform.

A driver will accept a trip to location 2 only if the additional waiting cost for a trip to location 1 outweighs the earnings difference between the two trips $(w_1 - w_2)$. If $\tau_{1,2}$ is the maximum time a driver is willing to wait for a trip to location 1, we have

$$\tau_{1,2}c = w_1 - w_2 \Rightarrow \tau_{1,2} = \frac{(w_1 - w_2)}{c}\,. \tag{3.4}$$

Little's Law relates the average number of items in a queue to the arrival rate and the average time spent in the system ($L = \lambda W$). Therefore, the first driver willing to accept a trip to location 2 will be at position $n_2 \triangleq \mu_1\tau_{1,2} = \mu_1(w_1 - w_2)/c$ with a continuation payoff of w_2, representing indifference. Given this and assuming infinite rider patience, the first position in the queue where a driver is willing to accept a trip to each location $i \in \mathcal{L}$ as opposed to wait and obtain a trip to location $i + 1$ is given by the following lemma.

Lemma 1. A strict FIFO dispatching system reaches equilibrium when a driver accepts a trip to destination $i \in \mathcal{L}$ only if their queue position q satisfies $q \geq n_i$, where $n_1 \triangleq 0$ and for all $i \geq 2$,

$$n_i \triangleq \sum_{j=1}^{i-1} \left(\frac{w_j - w_{j+1}}{c} \sum_{k=1}^{j} \mu_k \right). \tag{3.5}$$

Since a driver is indifferent at position n_i, their continuation payoff—the driver's net earnings from a trip minus the future waiting costs from their current position onward—is w_i regardless of accepting a trip to location i or not. Drivers at earlier positions wait for trips with higher earnings.

In reality, riders have finite patience, meaning that they will cancel their trip requests if it is declined too many times. If the patience level P is lower than the required queue positions n_i, the trip request will never reach a driver who is willing to accept it. As a result, many trips may go unfulfilled, significantly reducing system efficiency. Strict FIFO dispatching leads to highly inefficient driver allocation, reducing both driver earnings and platform revenue.

3.2.2 A dispatching mechanism

While strict FIFO is itself a dispatch mechanism, its simplicity allows it to be described informally. A formal definition becomes necessary when analyzing more complex strategies which modify queue order and require a generalized framework. The article formally describes a dispatching mechanism that determines how trip requests are assigned to drivers. The mechanism can either dispatch the trip to a driver or choose not to dispatch it, denoted by ϕ.

Definition 1 (Dispatching Mechanism; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 3, p. 11, Definition 1). A dispatching mechanism determines how trips are allocated based on the queue length Q, the dispatching history h, and the trip's destination. The mechanism selects a probability distribution over position in the queue $[0, Q] \cup \{\phi\}$. This means the platform decides whether to assign a rider's trip to a driver at position $q \in [0, Q]$ or to not dispatch it (ϕ).

The mechanism's decisions depend only on the queue length Q, the system's current state, and a trip's past dispatch history—not on past driver actions. Similarly, drivers' decisions depend on their current queue position and the current queue length. A driver's strategy in the queue is represented as a tuple $\sigma = (\alpha, \beta, \gamma)$, where:

(i) $\alpha(q, Q, i) \in [0,1]$: probability that a driver accepts a trip dispatch to location i.

(ii) $\beta(q, Q) \in [0,1]$: probability that a driver re-enters the queue at the tail after declining.

(iii) $\gamma(q, Q) \in [0,1]$: probability that a driver leaves the queue without taking a trip.

To evaluate the decision-making process of a driver, we define the continuation payoff, which captures the expected net benefit of remaining in the queue. Let $U(q, Q, \sigma, \sigma')$ be a random variable representing the continuation payoff of a driver at position q when the queue length is Q. This payoff depends on the driver adopting strategy σ while all other drivers follow strategy σ', including those who will enter the queue in the future. The continuation payoff consists of the net earnings from trips the driver may complete in the future, and the total opportunity cost incurred from waiting in the queue. The expected continuation payoff from position q is denoted as:

$$\pi(q, Q, \sigma, \sigma') \triangleq \mathbb{E}[U(q, Q, \sigma, \sigma')] .$$

This formulation allows the model to predict how drivers evaluate the trade-offs between accepting a trip, waiting for a better trip, or leaving the queue entirely. We now define several equilibrium properties that characterize an optimal dispatching mechanism. These properties ensure that drivers act rationally, do not attempt to manipulate their position in the queue, and make decisions that align with the system's efficiency goals.

Definition 2 (Subgame-Perfect Equilibrium; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 3, p. 12, Definition 2). A strategy σ^* is said to be a subgame-perfect equilibrium (SPE) if, for any feasible strategy σ, a driver following σ^* receives at least the same expected payoff as under σ:

$$\pi(q, Q, \sigma^*, \sigma^*) \geq \pi(q, Q, \sigma, \sigma^*), \quad \forall Q \geq 0, \forall q \in [0, Q]. \tag{3.6}$$

This ensures that drivers have no incentive to deviate from the equilibrium strategy, as doing so would not improve their expected payoff. While Osborne's definition of subgame-perfect equilibrium applies to extensive-form games, the version by Castro et al. adapts the same principle to dynamic queues. In both cases, players have no incentive to deviate at any stage, ensuring strategies remain optimal throughout. This conceptual overlap justifies applying subgame-perfection to ride-sharing dispatch models.[11]

[11] Although not a classical extensive-form game, the dynamic queue in our model shares core features: sequential decision-making, state-dependent payoffs, and strategic interactions over time. As such, subgame-perfection can naturally be extended to this stochastic, dynamic setting, as in Castro et al. (2021).

Definition 3 (Individual Rationality; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 3, p. 12, Definition 3). A dispatching mechanism is individually rational in SPE if drivers expect a non-negative payoff upon joining the queue.

$$\pi(q, Q, \sigma^*, \sigma^*) \geq 0, \quad \forall Q \geq 0, \forall q \in [0, Q]. \tag{3.7}$$

This condition guarantees that drivers do not enter the queue unless doing so yields an expected benefit, ensuring participation in the system remains viable.

Definition 4 (Envy-Freeness; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 3, p. 12, Definition 4). A mechanism is envy-free in SPE if no driver prefers the expected continuation payoff of another driver who has waited for less time.

$$\pi(q_1, Q, \sigma^*, \sigma^*) \geq \pi(q_2, Q, \sigma^*, \sigma^*), \quad \forall q_1, q_2 \in [0, Q] \text{ s. t. } q_1 \leq q_2. \tag{3.8}$$

This condition implies that drivers do not attempt to reposition themselves within the queue, ensuring fairness across different queue positions.

When a dispatching mechanism $\mathcal{M}$ reaches steady state under strategy σ^*, the length of the queue, denoted as Q^*, remains stable. This occurs when the rate at which drivers join the queue matches the rate at which they are dispatched. In this state, every driver follows the strategy σ^*, ensuring predictable queue dynamics. Let $\varkappa_i(\sigma^*)$ represent the fraction of trips to location i that are completed in equilibrium. The trip throughput of the mechanism is given by

$$T_{\mathcal{M}}(\sigma^*) = \sum_{i \in \mathcal{L}} \varkappa_i(\sigma^*) \mu_i. \tag{3.9}$$

The total earnings of drivers from completed trips must be balanced against the opportunity costs associated with waiting in the queue. The net revenue generated by the platform under the mechanism is expressed as

$$R_{\mathcal{M}}(\sigma^*) = \sum_{i \in \mathcal{L}} \varkappa_i(\sigma^*) \mu_i w_i - Q^* c_p. \tag{3.10}$$

The primary objective of a dispatching mechanism is to maximize trip throughput and net revenue while maintaining an efficient and stable equilibrium. A mechanism is considered optimal if, in

equilibrium (i) it achieves the first-best trip throughput, meaning that all feasible trips are completed with minimal inefficiency, (ii) it attains the second-best net revenue, which is the highest achievable net revenue under a dispatching system that is flexible, transparent, and does not penalize drivers for their choices.

3.2.3 Direct FIFO: selective matching

The direct FIFO mechanism builds upon FIFO dispatching by assigning lower-earning trips to drivers positioned further down the queue. This increases the likelihood of acceptance, as drivers further back are incentivized to take these trips in exchange for skipping the rest of the queue. When all drivers follow this strategy, the system reaches a subgame perfect equilibrium, ensuring maximum revenue and trip throughput within a flexible and transparent framework.

Definition 5 (Direct FIFO Dispatching; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 3, p. 13, Definition 5). Under the direct FIFO Dispatching, trips to each location $i \in \mathcal{L}$ are dispatched sequentially in a FIFO manner, but only from a specific queue position n_i. If the queue length Q meets or exceeds n_i, trips to location i are assigned. However, if $Q < n_i$, then no trips to i are dispatched.

Higher-earning trips are assigned to drivers at the front of the queue, as they have incurred the highest waiting costs. For trips to lower-earning destinations ($i > 1$), the mechanism bypasses drivers who are unlikely to accept them and instead starts dispatching from the first queue position where a driver is willing to accept. This assumes infinite rider patience.

Theorem 1 (Incentive Compatibility of Direct FIFO; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 3, p. 13, Theorem 1). *In a subgame-perfect equilibrium (SPE) under the direct FIFO mechanism, drivers accept all trips dispatched and enter the queue only if the queue length does not exceed*

$$\overline{Q} \triangleq n_\ell + \frac{w_\ell}{c} \sum_{i \in \mathcal{L}} \mu_i. \tag{3.11}$$

The equilibrium outcome is individually rational and envy-free.

The formula for $\overline{Q}$ reflects that, in a subgame perfect equilibrium (SPE), the queue is sufficiently long to accommodate all trip requests, including those to the lowest-paying destination ℓ. The segment of the queue given by $\frac{w_\ell}{c}\sum_{i\in\mathcal{L}}\mu_i$ represents the final positions after n_ℓ where drivers are still incentivized to accept trips to location ℓ rather than exit the queue.

If there are more drivers than required to complete all high-earning trips, direct FIFO does not achieve the first-best net revenue. Some drivers engage in strategic waiting, declining lower-earning trips in favor of trips with higher earnings. This results in a nonzero queue length, reducing net revenue. While strategic waiting cannot be entirely eliminated, it can be minimized under direct FIFO. The following theorem establishes that direct FIFO achieves the highest possible equilibrium net revenue under any transparent and flexible dispatching mechanism that does not penalize drivers for rejecting trips.

Theorem 2 (Optimality of direct FIFO; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 3, p. 14, Theorem 2). *For any economy, the direct FIFO mechanism satisfies in SPE (i) first-best trip throughput, meaning all feasible trips are completed, (ii) first-best net revenue when the platform's opportunity cost $c_p = 0$, (iii) second-best net revenue when $c_p \in (0, c]$, meaning it maximizes net revenue among mechanisms that do not penalize drivers for declining trips.*

When driver supply does not exceed total rider demand, $\lambda \leq \sum_{i\in\mathcal{L}}\mu_i$, the queue forms up to position n_{i^*}, and the lowest-earning trip i^* is partially completed. All completed trips match the first best outcome, and each driver earns a payoff of w_{i^*}. When driver supply exceeds total rider demand, $\lambda > \sum_{i\in\mathcal{L}}\mu_i$, the system is oversupplied, all trips are completed, and the queue reaches its maximum equilibrium length $\overline{Q}$, with drivers indifferent between queueing or leaving, yielding zero payoff.

Direct FIFO improves efficiency and revenue compared to strict FIFO but can be unfair, as it may allocate high-paying trips to drivers further back in the queue. To address this, Randomized FIFO is introduced—an approach that adds controlled randomness to trip assignments to reduce strategic waiting and promote fairer trip distribution among drivers.

3.2.4 Randomized FIFO: probabilistic dispatch to align incentives

The family of randomized FIFO mechanisms achieves optimal throughput and near-optimal revenue in equilibrium without unfair prioritization. Randomized FIFO dispatches trips uniformly at random to drivers in the queue, aligning incentives and reducing cherry-picking. Drivers are less likely to reject low-paying trips because this implies waiting significantly longer for the next opportunity. To illustrate the impact of randomization on incentive alignment, the article presents the steady-state Nash equilibrium under random dispatching, where every trip request is assigned to drivers uniformly at random within the queue.

Definition 6 (Nash Equilibrium in Steady State; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 4, p. 15, Definition 6). A strategy σ^* is said to form a Nash equilibrium among drivers in steady state if there exists a queue length $Q^* \geq 0$ such that for any feasible strategy σ and any queue position $q \in [0, Q^*]$,

$$\pi(q, Q^*, \sigma^*, \sigma^*) \geq \pi(q, Q^*, \sigma, \sigma^*) \ , \tag{3.12}$$

when all drivers adopt strategy σ^*, the steady-state queue length remains Q^*.

This condition ensures that no driver has an incentive to deviate from strategy σ^*, and the queue length stabilizes in equilibrium.

The concept of Nash equilibrium in steady state presented in Definition 6 builds on the classical notion of Nash equilibrium as introduced by Osborne (2000, Definition 2). In both formulations, a player (or drivers) has no incentive to unilaterally deviate from their strategy given the strategies of others. While Osborne's definition is framed in terms of preferences over terminal histories in extensive-form games, the Castro model translates this to expected payoffs at different queue positions under a fixed queue length Q^*. The common principle is that strategic optimality is preserved under unilateral deviations, whether in discrete histories or continuous queue positions, ensuring equilibrium stability in both settings.

Proposition 2 (Optimality of random dispatching; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 4, p. 15, Proposition 2). Under random dispatching, every trip is assigned uniformly at random across all drivers in the queue. In steady-state Nash equilibrium, this

mechanism achieves (i) first-best trip throughput, ensuring all feasible trips are completed, (ii) second-best net revenue when $c_p > 0$, (iii) first-best net revenue when $c_p = 0$.
This leads to two key considerations.

- A driver under random dispatching must wait significantly longer after declining a trip than under strict FIFO, increasing the cost of cherry-picking.
- Trip assignments are less predictable, increasing the variance in both waiting times and net earnings among drivers.

While pure randomization introduces substantial uncertainty, a well-structured randomized FIFO mechanism can align incentives while preserving fairness.

Definition 7 (Randomized FIFO; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 4, p. 16, Definition 7). A randomized FIFO mechanism is defined by dividing the queue into $m \geq 1$ bins, denoted as $\Big([\underline{b}^{(1)},\ \overline{b}^{(1)}], [\underline{b}^{(2)},\ \overline{b}^{(2)}], \dots, [\underline{b}^{(m)},\ \overline{b}^{(m)}]\Big)$. When a trip is dispatched for the k^{th} time, the mechanism randomly assigns it to a driver within the corresponding bin $[\underline{b}^{(k)},\ \overline{b}^{(k)}]$. If a trip is declined, it moves sequentially into the next bin until it is accepted or canceled. In essence, trip requests are initially assigned to drivers in the first bin ($[\underline{b}^{(1)},\ \overline{b}^{(1)}]$) uniformly at random. If a dispatch is rejected, the system moves the trip to the next bin, continuing the process until all bins are exhausted.

Given that riders have a patience level of P, a trip may be dispatched up to P times before the request is canceled. Let i^* denote the lowest-earning trip type that is partially completed under the first-best outcome (as in 3.1). The top i^* destinations are partitioned into $m \leq \min\{i^*, P\}$ ordered sets $\mathcal{L}^{(1)}, \dots, \mathcal{L}^{(m)}$, satisfying three conditions:

(i) (exhaustiveness) the union of all partitions covers the top i^* destinations: $\bigcup_{k=1}^{m} \mathcal{L}^{(k)} = \{1,2,\dots,i^*\} \subseteq \mathcal{L}$

(ii) (mutual exclusivity) trip i belongs to only one partition: $\mathcal{L}^{(k_1)} \cap \mathcal{L}^{(k_2)} = \emptyset, \quad \forall k_1 \neq k_2$

(iii) (monotonicity) later partitions contain lower-earning trips, such that for all k_1, k_2 where $k_1 < k_2$, the trips satisfy: $i < j, \quad \forall i \in \mathcal{L}^{(k_1)}, \forall j \in \mathcal{L}^{(k_2)}$

This guarantees that the higher-earning trips are assigned to earlier partitions, while lower-earning trips are assigned to later partitions. Each bin k is defined using the payoff gap from the minimum in its partition. The upper ($\underline{b}^{(k)}$) and lower ($\overline{b}^{(k)}$) bin bounds are calculated as

$$\underline{b}^{(k)} \triangleq \sum_{i \in \cup_{k'<k} \mathcal{L}^{(k')}} \left(w_i - \min_{i' \in \mathcal{L}^{(k)}} \{w_{i'}\} \right) \mu_i / c, \tag{3.13}$$

$$\overline{b}^{(k)} \triangleq \sum_{i \in \cup_{k' \leq k} \mathcal{L}^{(k')}} \left(w_i - \min_{i' \in \mathcal{L}^{(k)}} \{w_{i'}\} \right) \mu_i / c. \tag{3.14}$$

The formulas are derived using Little's Law, similarly to 3.5. The article proves that the bins start from the head of the queue (i.e., $\underline{b}^{(1)} = 0$), and that they do not overlap.

The primary result of this study is that the randomized FIFO mechanisms structured in the manner described achieve the optimal steady-state outcome in Nash equilibrium. Drivers are incentivized to accept trips in a way that maximizes trip throughput and net revenue without introducing unfair dispatching practices.

Theorem 3 (Optimality of randomized FIFO; Castro et. al. (2021), Randomized FIFO Mechanisms, Ch. 4, p. 16, Theorem 3). *For any given economy and any ordered partition of the top i^* destinations denoted as $(\mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(m)})$, where $m \leq min\{i^*, P\}$, a randomized FIFO mechanism that follows the structure outlined in equations (3.13) and (3.14) achieves first-best trip throughput and second-best net revenue in Nash equilibrium. When the platform incurs no opportunity cost (i.e., $c_p = 0$), the equilibrium also achieves the first-best net revenue.*

The equilibrium properties of randomized FIFO are fundamental for the upcoming analysis presented in Chapter 4. Under a randomized FIFO mechanism, a steady-state Nash equilibrium is achieved when (i) all driver in the k^{th} bin accept only trips within the top k partitions $\bigcup_{k'=1}^{k} \mathcal{L}^{(k')}$, (ii) no driver exits the queue without a trip or rejoins at the tail, (iii) drivers join the queue with probability $\min\{1, \sum_{i \in \mathcal{L}} \mu_i / \lambda\}$ upon arrival, and (iv) the queue length remains constant at Q^*. Furthermore, the continuation payoff for any driver in the k^{th} bin is equal to the net earnings of the lowest-paying trip in the k^{th} partition, i.e. $\pi^*(q) = \min_{i \in \mathcal{L}^{(k)}} \{w_i\}$ for all $q \in [\underline{b}^{(k)}, \ \overline{b}^{(k)}]$ for each

$k \leq m$. Being $\pi^*(q)$ non-negative and monotonically non-increasing in q implies that randomized FIFO ensures *individual rationality and envy-freeness in steady-state Nash equilibrium.*

With higher levels of rider patience, P, the randomized FIFO mechanism uses more bins to better match high-paying trips with longer-waiting drivers. If $(P \geq i^*)$, the mechanism assigns one trip to each partition and dispatches it to the driver at position n_k in the queue on its k^{th} attempt. In equilibrium, trips to all locations $k \leq i^*$ are accepted by drivers at n_k, leading to the same equilibrium outcome as in direct FIFO with all drivers having equal total payoffs.

3.3 Final remarks: relaxation of equilibrium

While the Randomized FIFO mechanism is theoretically efficient—achieving optimal throughput and near-optimal revenue in Nash equilibrium—these results rely on the assumption of perfectly rational driver behavior. In real-world settings, drivers may act unpredictably, deviate from equilibrium strategies, or respond to short-term incentives. To explore how the mechanism performs under such realistic conditions, Chapter 4 presents simulation results that test Randomized FIFO in scenarios where drivers are not fully rational. This provides insights into the robustness and practical limitations of the mechanism when applied outside the idealized assumptions of game-theoretic models.

Chapter 4

Behavioral Simulation Framework

Building upon the theoretical models and equilibrium analyses discussed in the previous chapters, we conduct two simulations with distinct objectives. The first evaluates the performance of acceptance rules that deviate from perfect rationality to understand their impact on system metrics. The second benchmarks each acceptance rule against the Nash equilibrium optimum, developed by Castro et. al. (2021), by testing all bin and trip partitions to identify the most efficient configuration for each rule. In this chapter, we present the first simulation framework, designed to recreate the dynamics of the Randomized FIFO dispatch mechanism under more realistic behavioral conditions. While prior work assumes drivers are perfectly rational agents, real-world settings often deviate from such idealized behavior. Drivers may exhibit bounded rationality, inconsistent patience levels, or heuristics-driven decision-making. To bridge this gap between theory and practice, the simulation implements variations of Randomized FIFO-based dispatch rules in an event-based environment where drivers make decisions according to predefined but imperfect acceptance strategies. The model builds on the $M/M/c$ queueing framework introduced in Chapter 3, and incorporates a dynamic game in which drivers act as players deciding whether to accept incoming trip offers based on queue position and perceived opportunity cost.

4.1 Simulation purpose and scope

The simulation serves as a computational testbed to evaluate the practical viability and robustness of Randomized FIFO mechanisms under a range of behavioral assumptions. Inspired by the original article's focus on high-demand ride-sharing locations—such as airports—it recreates a single-location dispatch environment with steady rider demand, varying trip earnings, and drivers making real-time decisions under pressure. These settings are prone to strategic behavior and fairness concerns due to limited rider patience and the high cost of missed offers. Instead of assuming fully rational agents, the simulation models a spectrum of driver behaviors, from deterministic cutoff rules to probabilistic and patience-based heuristics, to explore how real-world frictions affect system performance.

Crucially, the simulation not only replicates the structural components of the theoretical model (such as partitioned queue bins and rider patience thresholds), but also enables controlled experimentation with behavioral assumptions. This flexibility makes it possible to benchmark alternative acceptance rules against key performance metrics—such as service rate, revenue, fairness, and queue efficiency—highlighting the trade-offs that platforms may face when rationality is no longer guaranteed. Rather than predicting exact outcomes, we seek to identify trends vulnerabilities, and strengths in the mechanism's design across a diverse space of practical conditions.

4.2 Code structure and acceptance rules

The simulation code is organized into modular sections, each handling a distinct aspect of the Randomized FIFO environment. The simulation operates through a discrete-event object-driven framework, managing driver and rider arrivals and the dispatching process over discrete event timelines. Drivers and riders are modeled as agents arriving randomly, and their interactions are captured in a structured queue environment. Dispatches are managed through partitions and bins that are computed based on trip earnings and waiting costs. This structure allows clear separation between the setup of the simulation (parameters, partitions, bin boundaries) and its dynamic execution (agent interactions, event processing, metric calculations).

As mentioned above, the simulation tests a range of acceptance rules, each representing a distinct approach that drivers may take when deciding whether to accept or reject trip offers. The acceptance rules vary from simple unconditional acceptance to more sophisticated strategies that integrate probabilistic or patience-based decision-making. Specifically, the simulation considers the following rules (names are expressed as they appear in the code): AlwaysAccept, StrictCut, NE, ProbNE, ProbBinNE, DrivPNE, TimePNE. Each of these rules introduces different assumptions about driver rationality, patience, and responsiveness to queue conditions, enabling a comprehensive evaluation of how behavioral variation impacts the Randomized FIFO dispatch mechanism's overall performance. Notably, rules ending in NE are direct modifications of the Nash Equilibrium rule, each relaxing its assumptions in a specific way to simulate more realistic behavior. These rules can be viewed along a behavioral spectrum, with some—like ProbNE and ProbBinNE—closely mirroring the rational behavior expected in equilibrium, and others—like AlwaysAccept or StrictCut—representing more extreme or implausible heuristics. This ranking allows us to assess how incremental deviations from optimal behavior influence system-level outcomes.

4.2.1 Unconditional acceptance of all trips (AlwaysAccept)

The AlwaysAccept rule represents the simplest possible driver behavior: unconditional acceptance of every trip offer, regardless of queue position, destination, or expected earnings. Drivers following this rule do not engage in any strategic evaluation; they accept the first trip presented to them without delay. This rule serves as a baseline in the simulation, illustrating system performance in the absence of selective behavior or incentive-driven decision-making. While unrealistic in practice, AlwaysAccept provides a useful benchmark for understanding how much strategic filtering affects metrics like throughput, queue length, driver payoff, and platform revenue. It effectively models a setting where drivers are fully compliant and indifferent to the variability in trip values or waiting costs.

4.2.2 Deterministic queue-position threshold for acceptance (StrictCut)

The StrictCut, or Strict Cutoff, rule introduces a simple deterministic decision mechanism based on a driver's position in the queue. Under this rule, drivers accept a trip only if their current

queue position is less than or equal to a predefined threshold C (given that 0 represents the head of the queue, or the driver who has waited the longest in the queue). If a driver's position exceeds this cutoff, the offer is automatically declined, regardless of the trip's destination or potential earnings. This rule mimics a form of bounded rationality, where drivers follow a fixed heuristic, accepting offers only when they believe they have waited "long enough" to justify taking any trip. StrictCut reflects behavior that is not fully strategic but incorporates a basic sense of waiting cost and fairness. Drivers do not evaluate trip value but use their position in the queue as a proxy for opportunity cost—the longer they wait, the more likely they are to accept. This rule helps assess how threshold-based policies affect system performance, particularly regarding fairness, average payoff, and trip completion rates compared to more flexible or strategic acceptance rules.

4.2.3 Nash-based acceptance logic (NE)

The NE, or Nash equilibrium, acceptance rule represents the most strategic and rational behavior modeled in the simulation. Under this rule, a driver accepts a trip only if it belongs to a partition that offers at least as much expected utility as their current position in the queue. This logic directly follows from the equilibrium solution derived in the theoretical model, where each partition corresponds to a set of trips that rational drivers would accept based on their continuation value. The NE rule assumes full information and perfect rationality; drivers are aware of the trip partitions and can accurately assess whether an offer meets or exceeds their expected payoff. As such, it serves as the benchmark for optimal decision-making within the Randomized FIFO framework.

4.2.4 Probabilistic Nash equilibrium (ProbNE)

The ProbNE acceptance rule introduces a probabilistic variation of the Nash Equilibrium strategy. While it retains the core logic of partition-based decision-making, drivers following this rule no longer behave with perfect consistency. Consequently, drivers accept trips with 80% probability if $k_{\mathcal{L}} \leq k_b$, and 20% when $k_{\mathcal{L}} > k_b$, where $k_{\mathcal{L}}$ is the trip partition index and k_b is the driver's bin index. This models bounded rationality, where drivers generally act strategically but occasionally make suboptimal decisions due to uncertainty or error.

4.2.5 Bin-sensitive probabilistic acceptance (ProbBinNE)

The ProbBinNE, or Probability By Bin, rule refines the ProbNE strategy by tying acceptance probabilities to both the trip's partition and the bin from which the offer is made, incorporating a cost-aware behavioral logic. Specifically, when the partition index of the trip ($k_{\mathcal{L}}$) is less than or equal to the driver's assigned bin (k_b), the driver accepts the offer with a relatively high probability. This probability decreases progressively across bins (e.g., 0.9 for bin 1, 0.75 for bin 2, 0.6 for bin 3), reflecting the increased cost of waiting in earlier positions. Conversely, if $k_{\mathcal{L}} > k_b$, the acceptance probability is lower (e.g., 0.1 for bin 1, 0.3 for bin 2, 0.5 for bin 3), and increases with the bin index, indicating a willingness to compromise as the driver's position worsens. In effect, ProbBinNE models acceptance behavior as a probabilistic function of perceived queueing costs: drivers in earlier bins require more attractive offers to justify the cost of continuing to wait, while those further back are more tolerant of less optimal trips. This rule captures cost-responsive behavior that sits between fully rational strategy and human-like compromise.

4.2.6 Trip-specific patience-driven acceptance (DrivPNE)

The DrivPNE, or Driver Patience, rule introduces a form of partition-aware, trip-specific patience into the driver's decision-making process. Under this rule, each driver is assigned a fixed patience threshold for each combination of trip type and bin. Rather than making an immediate decision, the driver may reject an offer a predefined number of times before ultimately accepting it. For instance, a driver may accept a type-3 trip (lowest-paying trip on a scale of only 3 trip types) from bin 1 only after rejecting it twice, while accepting the same trip from bin 3 immediately. To keep things consistent with the other acceptance rules, we focus on two bins only in this simulation (we are holding rider patience fixed at $P = 2$). This implies that trips to location 3 are rejected at least once before being accepted. In the next chapter, we'll evaluate this rule across a three bins scheme as well. This structure reflects human-like hesitation: drivers are willing to wait in the hope of receiving a better offer, but their willingness is bounded and context-dependent. DrivPNE thus blends elements of the Nash Equilibrium framework with individual behavioral inertia, allowing for the analysis of how differentiated patience affects service rates, matching efficiency, and fairness across the queue.

4.2.7 Time-constrained queueing (TimePNE)

The TimePNE, or Time Patience, rule represents a time-based constraint on driver participation in the queue. Under this rule, each drivers are assigned a fixed patience threshold—defined in terms of time spent waiting—after which they leave the queue if they have not been matched with a rider. Unlike other rules that evaluate individual trip offers, TimePNE does not consider the characteristics of specific trips or partitions; instead, drivers follow a simple temporal cutoff, exiting the system once their wait exceeds a predefined duration (e.g., 30 time units). This models a realistic behavioral limit, where drivers are only willing to idle for so long before choosing to abandon the platform. TimePNE allows the simulation to capture the effects of queue abandonment on system efficiency, service probability, and platform revenue, particularly under high-demand conditions where delays are common.

4.3 Outcome analysis across acceptance strategies

The simulation results provide a comprehensive view of how each acceptance rule influences key performance metrics within the Randomized FIFO dispatch system. Before presenting the outcomes, we first introduce the metrics analyzed—clarifying how each indicator captures different aspects of system efficiency. The results are then presented individually for each rule, highlighting how different behavioral assumptions—ranging from unconditional acceptance to partition-sensitive patience—affect indicators such as service probability, driver utilization, average payoff, queue length, and platform revenue. This is followed by a comparative analysis across all acceptance rules, enabling the identification of consistent patterns, trade-offs, and outliers. Together, these results illustrate how deviations from equilibrium behavior, whether minor or extreme, can significantly shape system-level outcomes, offering practical insights into the robustness and flexibility of the dispatch mechanism under varied real-world conditions.

4.3.1 Key metrics for system evaluation

To evaluate how different acceptance rules influence the performance of the Randomized FIFO dispatch mechanism, the simulation tracks a set of key metrics that reflect efficiency, fairness, and platform profitability.

These metrics are computed at the end of each simulation run:

- **TP** (Throughput). The number of riders successfully matched and served per unit of time. It reflects the system's ability to fulfill demand.[12]
- **Bin$_k$_p**. The probability of successfully matching a rider in bin k, calculated as the ratio of accepted to offered trips in that bin.
- **P_match**. The overall rider matching probability, accounting for fallback across bins.
- **DrivUtil** (Driver Utilization). The proportion of drivers who complete at least one trip.
- **ServProb** (Service Probability). The fraction of rider requests that result in a successful match.[13]
- **CRate** (Cancellation Rate). Defined as 1 – ServProb, measuring how often requests go unserved.
- **ExRate** (Patience Exhaustion Rate). The share of riders who abandon after reaching their rejection limit.
- **AvgQLen** (Average Queue Length). The mean number of drivers waiting over time.
- **NetRev** (Net Revenue). Platform earnings net of drivers' aggregate waiting costs.
- **AvgDrvPay** (Average Driver Payoff). Mean driver payoff (earnings minus waiting costs).
- **VarDrvPay** (Variance of Driver Payoffs). Statistical variance of driver payoffs, serving as a proxy for income inequality.

Together, these metrics enable a multi-dimensional assessment of system performance, facilitating direct comparison across acceptance rules in terms of efficiency, fairness, and platform incentives. Below, we present the key findings from our simulation runs. For full code listings and detailed metric definitions, please refer to Appendix A.

[12] Intuitively, this measure aligns with the cumulative matching probability (P_match) multiplied by the total rider demand ($\sum \mu_i$). While computed directly from the simulation, the validity of throughput can be intuitively confirmed by this relationship, highlighting its consistency as an efficiency indicator.

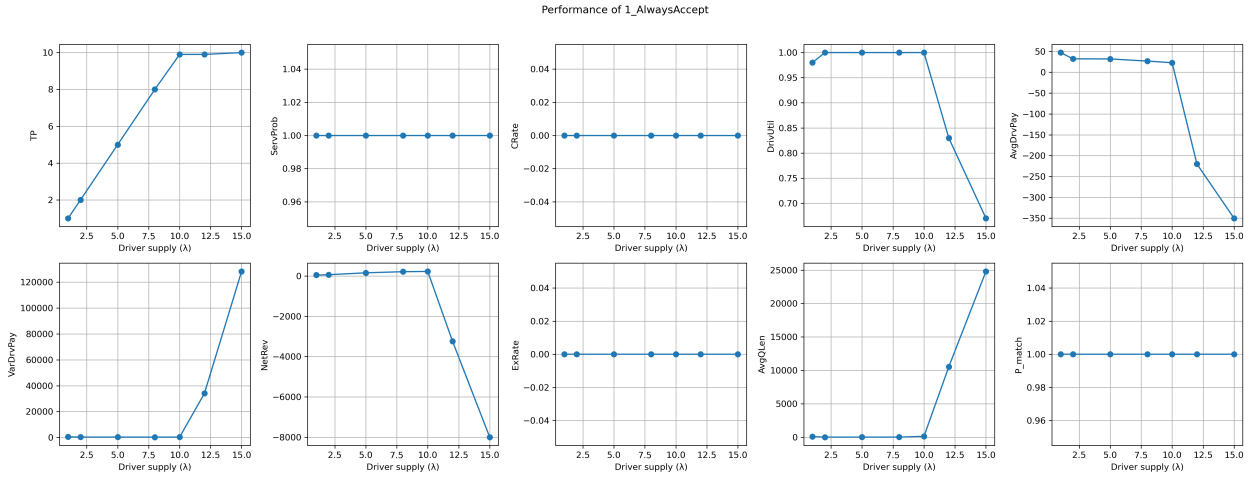
[13] As bins are progressively reached, this metric converges to the ratio of total rider demand to total offers per rider—i.e., $\sum \mu / \sum o$, where μ is the location demand and o the average offers needed for acceptance.

4.3.2 Results by acceptance rule

The results below are organized by acceptance rule, using the code names assigned in the simulation (e.g., 3_NE, 4_ProbNE). For each rule, we examine how system performance evolves with changes in driver supply. The simulations are based on a stylized urban environment with three destination types [1, 2, 3], each offering different rewards and arrival rates: \$75, \$25, and \$15, with relative demand rates of [1, 6, 3]. Driver cost per unit time is set to 1/3. Riders are assumed to have a patience level of 2, meaning they will attempt up to two matches before exiting the system. Driver supply (λ) varies from 1 to 15 to explore undersupply, balanced, and oversupply conditions. Each simulation runs for 10,000 time units to approximate steady-state behavior. To isolate the effects of the acceptance mechanism, the queue is left unbounded, reflecting driver patience and allowing the system to capture extreme queue growth in oversupplied conditions.

1_AlwaysAccept. We first present the main results under the AlwaysAccept rule, which serves as a baseline scenario where all trip offers are unconditionally accepted.

Rule	P	λ	i*	Partitions	Bins	TP	Bin1_p	Bin2_p	P_match	DrivUtil	ServProb	CRate	ExRate	AvgQLen	NetRev	AvgDrvPay	VarDrvPay
1_AlwaysAccept	2	1	1	[[1]]	[(0, 0)]	1	1	NaN	1	0.98	1	0	0	80.67	46.75	47.25	539.81
1_AlwaysAccept	2	2	2	[[1], [2]]	[(0, 0), (150, 150)]	2	1	NaN	1	1	1	0	0	0.4	65.06	32.27	313.01
1_AlwaysAccept	2	5	2	[[1], [2]]	[(0, 0), (150, 150)]	5	1	NaN	1	1	1	0	0	2.41	158.67	31.98	306.76
1_AlwaysAccept	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	8	1	NaN	1	1	1	0	0	4.09	213.84	26.86	278.17
1_AlwaysAccept	2	10	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.9	1	NaN	1	1	1	0	0	121.7	227.9	22.87	298.57
1_AlwaysAccept	2	12	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.9	1	NaN	1	0.83	1	0	0	10517.26	-3238.18	-219.84	34034.67
1_AlwaysAccept	2	15	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	10	1	NaN	1	0.67	1	0	0	24802.14	-7997.95	-350.26	128262.24



As driver supply (λ) increases, throughput (TP) rises linearly until it plateaus at the demand ceiling (10 trips per unit time), confirming full utilization of demand.

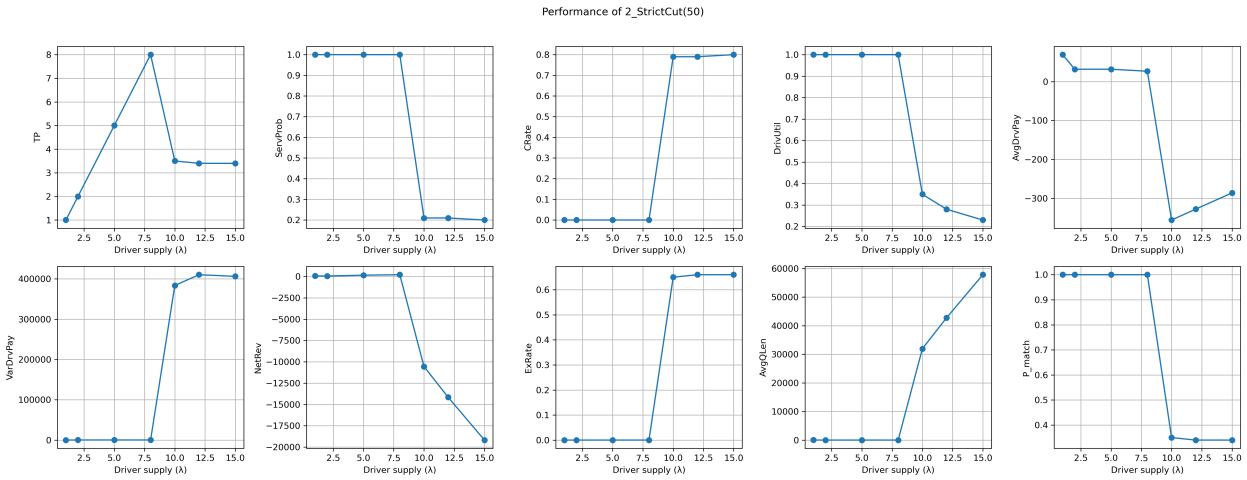
Rule	λ	L1_Bin1	L1_Bin2	L2_Bin1	L2_Bin2
1_AlwaysAccept	1	100	0	0	0
1_AlwaysAccept	2	100	0	100	0
1_AlwaysAccept	5	100	0	100	0
1_AlwaysAccept	8	100	0	100	0
1_AlwaysAccept	10	100	0	100	0
1_AlwaysAccept	12	100	0	100	0
1_AlwaysAccept	15	100	0	100	0

These patterns are consistent with the bin-level acceptance data, where all trips to Location 1 are always matched in Bin 1, and all trips to Location 2 are also matched in Bin 1 once $i^* > 1$, resulting in a 100% acceptance rate concentrated in a single bin.

However, the absence of queue control or selective acceptance causes queue length and driver payoff variance to explode in oversupplied regimes ($\lambda \geq 10$). Average driver payoff becomes strongly negative, reflecting high waiting costs due to long idle times. Net revenue collapses into deeply negative territory, indicating inefficiency from platform overspending on driver wait compensation. Driver utilization steadily declines beyond $\lambda = 10$, while service probability remains at 1.0 throughout—since all riders are matched eventually, albeit with excessive driver waiting. This masks the poor platform performance and worsening driver experience, evidenced by sharply increasing queue length and income dispersion. Overall, while AlwaysAccept ensures zero cancellation and maximal service levels, it leads to extreme inefficiencies and platform losses in high-supply scenarios, highlighting the need for smarter acceptance rules to manage market balance.

2_StrictCut(50). The results for StrictCut at position 50 illustrate how enforcing a fixed positional threshold impacts key performance metrics across varying levels of driver supply.

Rule	P	λ	i*	Partitions	Bins	TP	Bin1_p	Bin2_p	P_match	DrivUtil	ServProb	CRate	ExRate	AvgQLen	NetRev	AvgDrvPay	VarDrvPay
2_StrictCut(50)	2	1	1	[[1]]	[(0, 0)]	1	1	NaN	1	1	1	0	0	15.98	67.75	69.34	33.53
2_StrictCut(50)	2	2	2	[[1], [2]]	[(0, 0), (150, 150)]	2	1	NaN	1	1	1	0	0	0.39	63.55	31.84	297.64
2_StrictCut(50)	2	5	2	[[1], [2]]	[(0, 0), (150, 150)]	5	1	NaN	1	1	1	0	0	2.44	158.38	31.85	301.57
2_StrictCut(50)	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	8	1	NaN	1	1	1	0	0	3.97	214.43	26.79	272.98
2_StrictCut(50)	2	10	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	3.5	0.35	0	0.35	0.35	0.21	0.79	0.65	31940.92	-10552.77	-355.78	383605.81
2_StrictCut(50)	2	12	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	3.4	0.34	0	0.34	0.28	0.21	0.79	0.66	42758.42	-14161.42	-327.68	410364.52
2_StrictCut(50)	2	15	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	3.4	0.34	0	0.34	0.23	0.2	0.8	0.66	57812.32	-19179.06	-286.04	406229.24



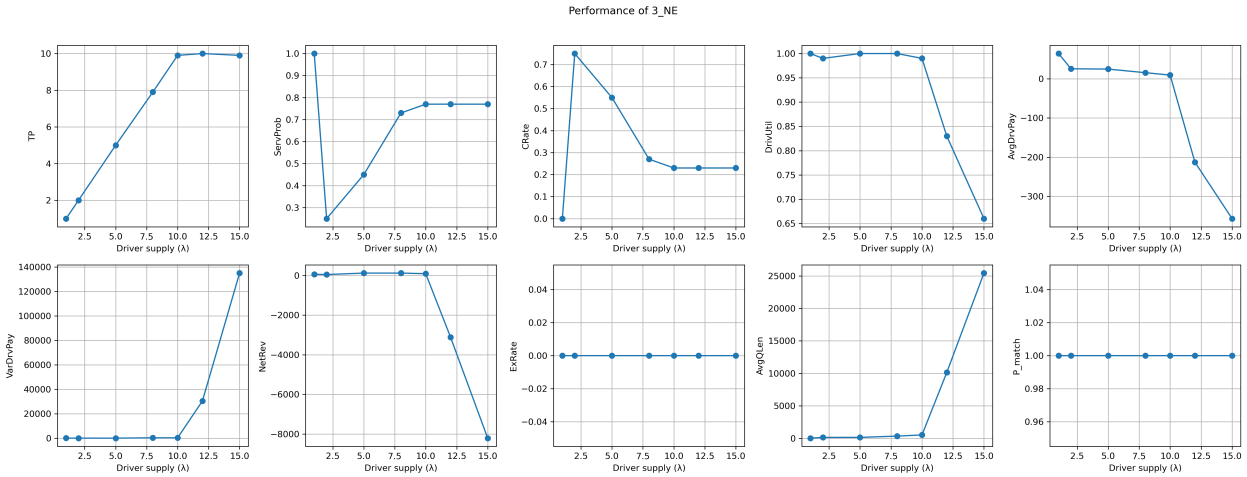
At low to moderate driver supply levels ($\lambda \leq 8$), StrictCut(50) performs similarly to the AlwaysAccept benchmark—throughput (TP) rises steadily, service probability is perfect, and driver utilization is complete. This is confirmed by the distribution of accepted matches: from $\lambda = 2$ to $\lambda = 15$, 100% of L1 and L2 trips are accepted in Bin1, with zero matches from downstream positions.

Rule	λ	L1_Bin1	L1_Bin2	L2_Bin1	L2_Bin2
2_StrictCut(50)	1	100	0	0	0
2_StrictCut(50)	2	100	0	100	0
2_StrictCut(50)	5	100	0	100	0
2_StrictCut(50)	8	100	0	100	0
2_StrictCut(50)	10	100	0	100	0
2_StrictCut(50)	12	100	0	100	0
2_StrictCut(50)	15	100	0	100	0

However, once λ exceeds 8, the rigid position threshold at queue index 50 becomes a bottleneck. TP quickly caps at around 3.5, Bin1_p stagnates near 0.35, and P_match also stabilizes at 0.34–0.35, signaling that only early-position drivers are consistently matched while the rest are bypassed. As a result, service probability drops, despite many rider requests, since offers beyond the strict cutoff are wasted. This inefficiency leads to a sharp rise in cancellation and exhaustion rates, and driver utilization plummets to just 23% at high λ. Economic outcomes reflect these mismatches: net revenue becomes deeply negative, and average driver payoffs collapse into losses with extremely high variance (over 400,000), driven by idle time and growing inequality in driver opportunities. Average queue lengths explode (over 57,000 at $\lambda = 15$), revealing systemic congestion. In sum, while StrictCut(50) enforces queue fairness at low supply, it fails to adapt under oversupply, creating performance ceilings and instability across key metrics.

3_NE. The NE rule captures strategic driver behavior under equilibrium conditions, where acceptance decisions reflect utility-maximizing responses to rider destinations and queue positioning. The following results illustrate how this rational behavior shapes overall system dynamics across varying levels of driver supply.

Rule	P	λ	i*	Partitions	Bins	TP	Bin1_p	Bin2_p	P_match	DrivUtil	ServProb	CRate	ExRate	AvgQLen	NetRev	AvgDrvPay	VarDrvPay
3_NE	2	1	1	[[1]]	[(0, 0)]	1	1	NaN	1	1	1	0	0	31.5	63.34	64.32	61.97
3_NE	2	2	2	[[1], [2]]	[(0, 0), (150, 150)]	2	0.14	1	1	0.99	0.25	0.75	0	148.25	50.21	25.42	25.08
3_NE	2	5	2	[[1], [2]]	[(0, 0), (150, 150)]	5	0.14	1	1	1	0.45	0.55	0	152.01	123.22	24.7	6.61
3_NE	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7.9	0.7	1	1	1	0.73	0.27	0	347.69	123.21	15.49	318.52
3_NE	2	10	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.9	0.7	1	1	0.99	0.77	0.23	0	527.63	93.28	9.41	274.69
3_NE	2	12	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	10	0.7	1	1	0.83	0.77	0.23	0	10140.6	-3109.9	-212.69	30384.74
3_NE	2	15	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.9	0.7	1	1	0.66	0.77	0.23	0	25429.77	-8207.55	-356.72	134965.02



As driver supply (λ) increases, throughput (TP) grows quickly and stabilizes around 10 riders per unit of time, corresponding to the total passenger arrival rate. This suggests the system successfully fulfills nearly all demand once an adequate number of drivers are available, validating the effectiveness of equilibrium-based matching. The service probability follows a more nuanced trajectory and eventually stabilizes at approximately 0.77—matching the theoretical limit derived from the ratio of total accepted arrival rates to total offers: $1+6+3/1+6+2\cdot 3 = 10/13 \approx 0.77$.

Rule	λ	L1_Bin1	L1_Bin2	L2_Bin1	L2_Bin2
3_NE	1	100	0	0	0
3_NE	2	100	0	0	100
3_NE	5	100	0	0	100
3_NE	8	100	0	0	100
3_NE	10	100	0	0	100
3_NE	12	100	0	0	100
3_NE	15	100	0	0	100

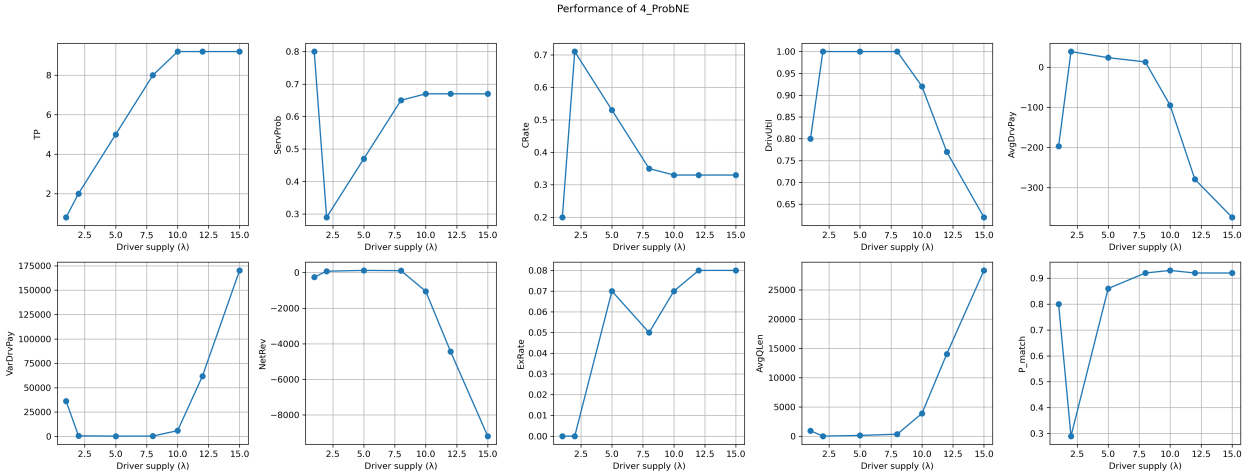
This cap reflects that only trips that reach a willing bin contribute to matches under NE behavior. Bin 1 acceptance probability (Bin1_p) remains flat at ~0.7 once the queue reliably reaches beyond the Bin 1 boundary. The data in the table above supports this: offers to location 1 (L1) are always

fully accepted in Bin1, while L2 is only ever accepted in Bin2, which begins receiving and accepting offers starting at $\lambda = 2$ and continues with 100% acceptance through $\lambda = 15$.

This behavior stabilizes the cancellation rate (Crate), which drops significantly once location 2 trips consistently reach Bin2, reducing mismatches. However, despite high throughput and stable matching efficiency, economic performance deteriorates sharply beyond equilibrium levels. As λ exceeds 10, driver utilization drops from 0.99 to 0.66, average driver payoffs plunge into negative territory, and net revenue becomes increasingly negative. This decline stems from inflated queue lengths (over 25,000 at $\lambda = 15$), long idle times, and growing compensation costs. These results underscore that while NE matching is efficient within equilibrium bounds, it suffers substantially from performance losses when supply far exceeds demand.

4_ProbNE. The results for ProbNE highlight how introducing probabilistic acceptance affects system behavior across varying driver supply levels. Specifically, drivers accept requests with an 80% probability when the rider's destination lies within a location group (partition) with index corresponding to their own bin or a lower, higher-paying one, and with a 20% probability otherwise. Compared to deterministic rules, this approach introduces smoother performance transitions and softens queue dynamics, particularly in intermediate supply regimes.

Rule	P	λ	i*	Partitions	Bins	TP	Bin1_p	Bin2_p	P_match	DrivUtil	ServProb	CRate	ExRate	AvgQLen	NetRev	AvgDrvPay	VarDrvPay
4_ProbNE	2	1	1	[[1]]	[(0, 0)]	0.8	0.8	NaN	0.8	0.8	0.8	0.2	0	948.58	-256.43	-197.01	36214.75
4_ProbNE	2	2	2	[[1], [2]]	[(0, 0), (150, 150)]	2	0.29	NaN	0.29	1	0.29	0.71	0	33.95	77.58	39.21	619.38
4_ProbNE	2	5	2	[[1], [2]]	[(0, 0), (150, 150)]	5	0.29	0.8	0.86	1	0.47	0.53	0.07	153.89	121.19	24.01	308.7
4_ProbNE	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	8	0.62	0.8	0.92	1	0.65	0.35	0.05	360.19	107.77	13.44	342.75
4_ProbNE	2	10	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.2	0.62	0.8	0.93	0.92	0.67	0.33	0.07	3916.66	-1051.54	-94.75	5953.86
4_ProbNE	2	12	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.2	0.62	0.8	0.92	0.77	0.67	0.33	0.08	14060.48	-4432.94	-279.45	61720.53
4_ProbNE	2	15	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.2	0.62	0.8	0.92	0.62	0.67	0.33	0.08	28327.44	-9189.5	-374.14	170226.09



Throughput (TP) increases steadily with driver supply and saturates just below 10, closely tracking the rise in cumulative match probability (P_match), which converges around 0.92. This cap stems from the fixed probabilities embedded in the rule: trips to higher bins are only accepted with 0.8 and 0.2 probability depending on their relative bin position, which limits maximum achievable matches regardless of queue size.

Rule	λ	L1_Bin1	L1_Bin2	L2_Bin1	L2_Bin2
4_ProbNE	1	100	0	0	0
4_ProbNE	2	100	0	100	0
4_ProbNE	5	87.4	12.6	29.1	70.9
4_ProbNE	8	89.4	10.6	33.8	66.2
4_ProbNE	10	83.7	16.3	24.4	75.6
4_ProbNE	12	83.5	16.5	23.8	76.2
4_ProbNE	15	83.2	16.8	23.7	76.3

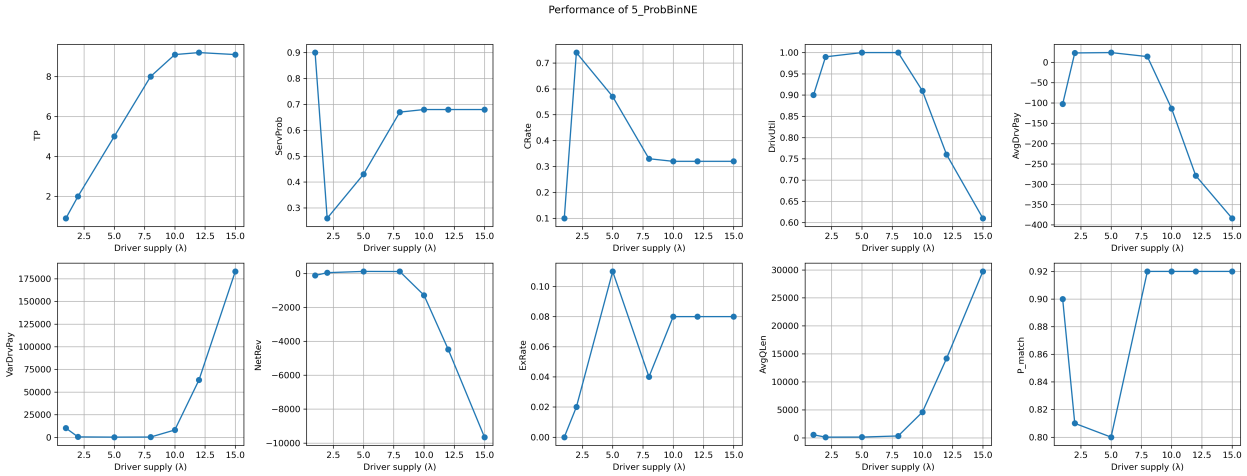
This pattern is further confirmed in the table above, which shows how trips to Location 1 and 2 are split across bins under oversupply: only ~83% of L1 trips and ~76% of L2 trips are matched, consistent with the 0.8 acceptance cap.

Service probability stabilizes near 0.67 in high-supply regimes, reflecting the equilibrium between increasing offers and the bounded acceptance chances. This is lower than NE's 0.77 cap because some offers—even to drivers in matching bins—are probabilistically rejected. Cancellation rate and patience exhaustion grow with λ due to this stochastic rejection, though more mildly than in StrictCut(50). Driver utilization begins at 1.0 (with the exception of the single-bin scenario) and then gently declines with oversupply. Economic metrics like net revenue and average driver payoff deteriorate in high λ, though not as drastically as with more rigid rules. Overall, ProbNE achieves a more resilient balance between match rates and system flexibility but remains constrained by its inherent probability ceilings.

5_ProbBinNE. We now examine the performance of ProbBinNE, which refines the probabilistic decision logic by assigning different acceptance probabilities across bins. This rule increases realism by capturing the idea that drivers may be more likely to accept requests in higher-priority bins. The results reveal how this differentiated behavior affects efficiency, utilization, and fairness across varying levels of driver supply. Drivers accept requests with a 90%, 75%, or 60% (ordered by bin 1-3) probability when the rider's destination lies within a location group (partition) with index

corresponding to their own bin or a lower, higher-paying one, and with a 10%, 30%, or 50% probability otherwise.

Rule	P	λ	i*	Partitions	Bins	TP	Bin1_p	Bin2_p	P_match	DrivUtil	ServProb	CRate	ExRate	AvgQLen	NetRev	AvgDrvPay	VarDrvPay
5_ProbBinNE	2	1	1	[[1]]	[(0, 0)]	0.9	0.9	NaN	0.9	0.9	0.9	0.1	0	563.53	-120.85	-102.59	10174.23
5_ProbBinNE	2	2	2	[[1], [2]]	[(0, 0), (150, 150)]	2	0.21	0.75	0.81	0.99	0.26	0.74	0.02	145.62	46.14	23.26	480.98
5_ProbBinNE	2	5	2	[[1], [2]]	[(0, 0), (150, 150)]	5	0.21	0.75	0.8	1	0.43	0.57	0.11	156.28	119.9	24.15	222.41
5_ProbBinNE	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	8	0.66	0.75	0.92	1	0.67	0.33	0.04	355.43	113.59	14.26	323.14
5_ProbBinNE	2	10	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.1	0.66	0.75	0.92	0.91	0.68	0.32	0.08	4604.18	-1281.17	-113.54	8095.56
5_ProbBinNE	2	12	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.2	0.66	0.75	0.92	0.76	0.68	0.32	0.08	14192.15	-4475.32	-279.04	63311.81
5_ProbBinNE	2	15	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	9.1	0.66	0.75	0.92	0.61	0.68	0.32	0.08	29722.92	-9654.46	-383.93	183019.17



The performance of ProbBinNE closely resembles that of ProbNE, with key differences emerging from the bin-specific acceptance probabilities. Throughput (TP) rises predictably with driver supply and plateaus at 9.2 due to the limit set by cumulative match probability (P_match = 0.92) multiplied by total rider demand ($\mu = 10$). The service probability curve similarly levels off around 0.68, reflecting the weighted effect of acceptance caps per bin—bin 1 capped at 0.66, bin 2 at 0.75—and their respective exposure to ride offers. For example, in oversupplied settings, bin 1 is saturated with offers to L1 and L2 but rejects the latter ~33% of the time, which inflates the offer denominator while limiting matches.

Rule	λ	L1_Bin1	L1_Bin2	L2_Bin1	L2_Bin2
5_ProbBinNE	1	100	0	0	0
5_ProbBinNE	2	99	1	55.5	44.5
5_ProbBinNE	5	93.3	6.7	15.2	84.8
5_ProbBinNE	8	95.8	4.2	22.4	77.6
5_ProbBinNE	10	92.5	7.5	13.2	86.8
5_ProbBinNE	12	92.2	7.8	12.8	87.2
5_ProbBinNE	15	92.3	7.7	12.8	87.2

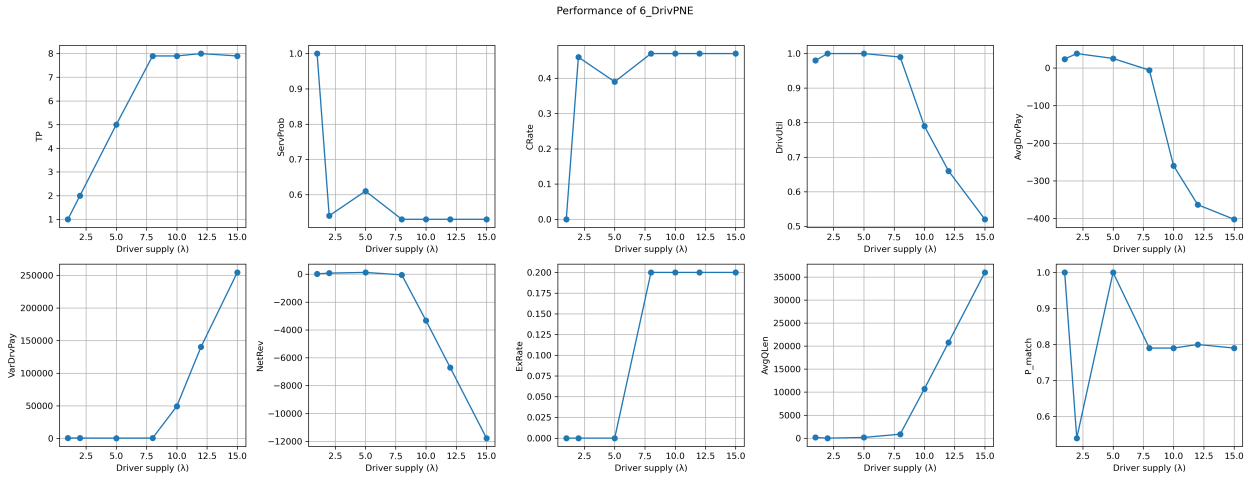
These patterns are evident in the match distribution table: Bin 1 consistently captures over 92% of L1 matches across $\lambda \geq 5$, while Bin 2 handles most L2 matches—87.2% at $\lambda = 15$. Yet, Bin 1

also regularly receives L2 offers (e.g., 12.8% at $\lambda = 15$) and rejects many, dragging down the service rate.

Cancellation and patience exhaustion rates stabilize at 0.32 and 0.08 respectively, similarly to the caps observed in ProbNE. Driver utilization drops steadily with λ, from 1.0 to 0.61, as many drivers remain idle for longer periods. This leads to worsening driver payoffs and rising variance, culminating in high queue lengths (nearly 30,000 at $\lambda = 15$) and platform losses. Notably, caps in P_match (0.92) and Bin1_p (0.66) arise directly from the probabilistic structure of the rule—these are not results of behavioral changes but rather fixed acceptance ceilings. Overall, ProbBinNE introduces more nuanced rejection behavior while preserving throughput efficiency up to high supply levels, albeit at the cost of fairness and revenue stability in oversaturated conditions.

6_DrivPNE. DrivPNE incorporates targeted patience thresholds based on both bin index and destination. This design reflects the idea that drivers may be willing to wait through a certain number of mismatches before accepting trips to lower-valued locations. The willingness to wait is proportional to the waiting costs drivers incur while remaining in the queue. Drivers in Bin 1 accept trips to Location 1 immediately, to Location 2 after 1 decline, and to Location 3 after 2 declines. Drivers in Bin 2 accept Location 1 and 2 immediately and Location 3 after 1 decline.

Rule	P	λ	i*	Partitions	Bins	TP	Bin1_p	Bin2_p	P_match	DrivUtil	ServProb	CRate	ExRate	AvgQLen	NetRev	AvgDrvPay	VarDrvPay
6_DrivPNE	2	1	1	[[1]]	[(0, 0)]	1	1	NaN	1	0.98	1	0	0	154.64	22.96	23.54	660.67
6_DrivPNE	2	2	2	[[1], [2]]	[(0, 0), (150, 150)]	2	0.54	NaN	0.54	1	0.54	0.46	0	0.99	76.36	38.14	488.88
6_DrivPNE	2	5	2	[[1], [2]]	[(0, 0), (150, 150)]	5	0.54	1	1	1	0.61	0.39	0	148.96	125.39	25.18	376.52
6_DrivPNE	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7.9	0.5	0.59	0.79	0.99	0.53	0.47	0.2	862.39	-49.29	-6.01	447.71
6_DrivPNE	2	10	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7.9	0.5	0.59	0.79	0.79	0.53	0.47	0.2	10708.58	-3330.37	-259.89	49229.13
6_DrivPNE	2	12	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	8	0.5	0.59	0.8	0.66	0.53	0.47	0.2	20809.77	-6696.99	-363.66	140165.14
6_DrivPNE	2	15	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7.9	0.5	0.59	0.79	0.52	0.53	0.47	0.2	36018.76	-11767.73	-402.33	254469.93



The performance of DrivPNE exhibits several distinctive patterns driven by its patience-based queue logic. Throughput (TP) increases with driver supply and stabilizes around 7.9 trips per unit of time at $\lambda \geq 8$. This cap stems from the limited patience thresholds that restrict when drivers are willing to accept lower-priority trips. Despite having the full set of drivers in queue, trips to lower-valued locations are delayed until enough mismatches occur, limiting instantaneous matching and suppressing P_match, which caps at 0.79. Service probability stabilizes around 0.53, significantly lower than that of AlwaysAccept or NE. This reflects the high number of rider offers that are rejected when they arrive before reaching a driver's acceptance threshold.

Rule	λ	L1_Bin1	L1_Bin2	L2_Bin1	L2_Bin2
6_DrivPNE	1	100	0	0	0
6_DrivPNE	2	100	0	100	0
6_DrivPNE	5	100	0	70.2	29.8
6_DrivPNE	8	66.1	33.9	44.4	55.6
6_DrivPNE	10	65.7	34.3	44.8	55.2
6_DrivPNE	12	65.8	34.2	44.6	55.4
6_DrivPNE	15	65.8	34.2	44.4	55.6

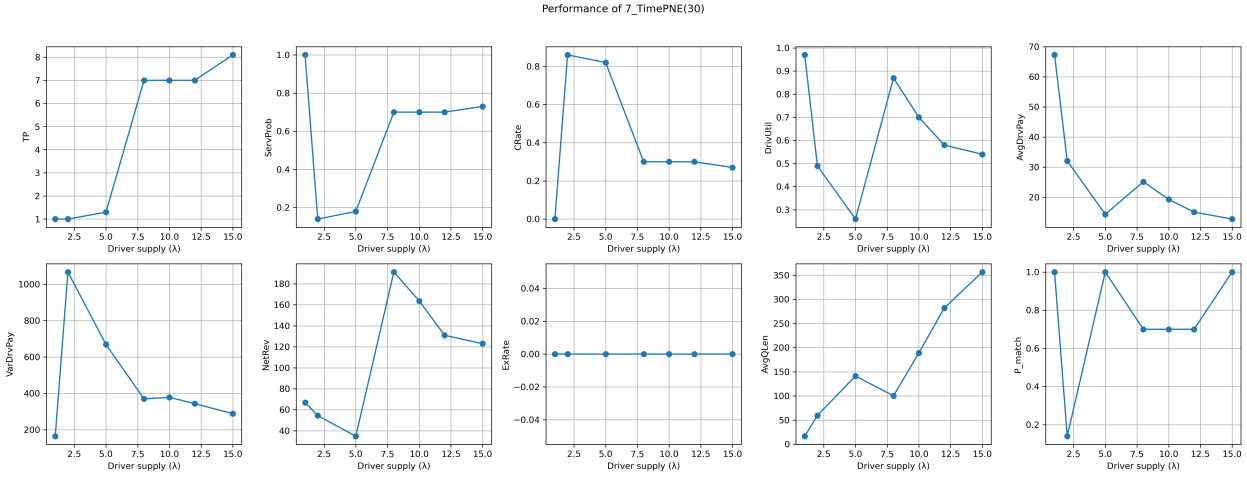
As seen in the data, only about 66% of L1 trips are matched to Bin 1 at $\lambda \geq 8$, while substantial offers are routed to Bin 2. For example, at $\lambda = 15$, L1 matches are split as 65.8% to Bin 1, 34.2% to Bin 2, reflecting early mismatches and driver drop-off points.

The impact of these thresholds is visible in the stability of Bin1_p and Bin2_p values, which plateau at 0.5 and 0.59 respectively. This confirms that despite increased queue length, driver willingness to serve only arrives after specific declines, capping their participation rate. As a result, cancellation and exhaustion rates rise and flatten near 0.47 and 0.2, respectively. Economic performance deteriorates sharply as supply increases. Average driver payoff declines from 23.5 to – 402.3, with variance exploding to over 250,000. Queue length follows a steep upward trajectory, reaching over 36,000 at $\lambda = 15$, confirming substantial delays and system congestion. Driver utilization falls from 0.98 to 0.52, reflecting the long idle times caused by drivers waiting to reach acceptable positions in the queue.

Overall, DrivPNE introduces realistic behavioral heterogeneity by allowing variable patience, but this comes at the cost of substantial performance trade-offs. While it avoids strict rejection rules, its inherent delays in acceptance drive inefficiencies under high-supply scenarios, leading to capped throughput, suppressed service probability, and considerable platform losses.

7_TimePNE. TimePNE applies a uniform time-based cutoff, where drivers exit the queue if unmatched after a fixed threshold of T = 30 time units. Unlike spatial or probabilistic rules, it does not consider trip characteristics—only wait duration. This models realistic abandonment behavior and reveals how limited patience impacts system efficiency and service under high-delay conditions.

Rule	P	λ	i*	Partitions	Bins	TP	Bin1_p	Bin2_p	P_match	DrivUtil	ServProb	CRate	ExRate	AvgQLen	NetRev	AvgDrvPay	VarDrvPay
7_TimePNE(30)	2	1	1	[[1]]	[(0, 0)]	1	1	NaN	1	0.97	1	0	0	16.54	66.95	67.3	163.9
7_TimePNE(30)	2	2	2	[[1], [2]]	[(0, 0), (150, 150)]	1	0.14	NaN	0.14	0.49	0.14	0.86	0	59.43	54.49	32.08	1067.18
7_TimePNE(30)	2	5	2	[[1], [2]]	[(0, 0), (150, 150)]	1.3	0.14	1	1	0.26	0.18	0.82	0	141.2	34.83	14.35	668.27
7_TimePNE(30)	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7	0.7	NaN	0.7	0.87	0.7	0.3	0	100.16	191.14	25.17	369.25
7_TimePNE(30)	2	10	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7	0.7	NaN	0.7	0.7	0.7	0.3	0	188.84	163.64	19.3	377.74
7_TimePNE(30)	2	12	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7	0.7	NaN	0.7	0.58	0.7	0.3	0	281.87	130.92	15.08	343.05
7_TimePNE(30)	2	15	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	8.1	0.7	1	1	0.54	0.73	0.27	0	356.51	122.91	12.77	288.15



By forcing drivers out after 30 time units, TimePNE keeps the queue from stretching into lower-priority bins, so overall throughput is governed by how quickly drivers move through Bin 1 rather than by total demand. As supply rises, TP jumps from about 1 at $\lambda = 5$ to around 7 under the [1, 2],[3] split, reflecting the larger share of drivers in Bin 1. However, because few drivers ever reach the Bin 2 cutoff before abandoning, TP stays capped and never fully exploits excess supply. Consequently, service probability levels off around 0.7 and only edges up slightly once supply is high enough to cover Bin 2. Rider patience exhaustion remains at zero, which is encouraging, but drivers still cancel roughly 30% of requests whenever $\lambda \geq 8$.

Rule	λ	L1_Bin1	L1_Bin2	L2_Bin1	L2_Bin2
7_TimePNE(30)	1	100	0	0	0
7_TimePNE(30)	2	100	0	0	0
7_TimePNE(30)	5	100	0	0	100
7_TimePNE(30)	8	100	0	0	0
7_TimePNE(30)	10	100	0	0	0
7_TimePNE(30)	12	100	0	0	0
7_TimePNE(30)	15	100	0	0	100

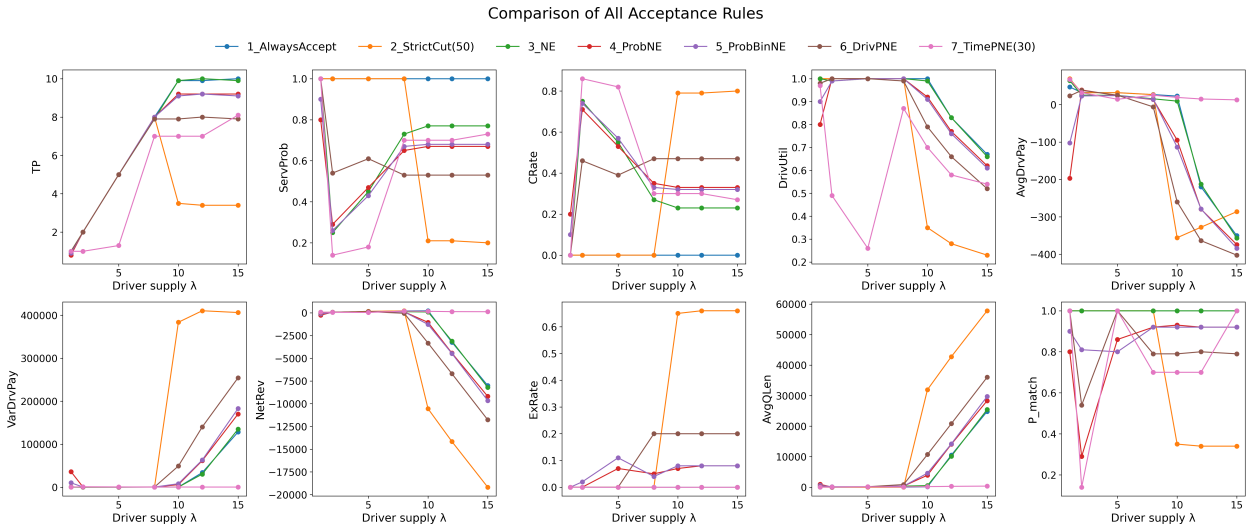
As the table illustrates, TimePNE mirrors NE's bin-based acceptance patterns exactly—drivers accept the same trip partitions in each bin. The only deviation arises when the queue never grows large enough to reach Partition 2, in which case those trips simply go unoffered.

This rule's most dramatic impact is on net revenue: by capping driver wait times, the platform retains nearly the full fare for every match, driving revenue well above that of other rules. Although surpassing the Nash benchmark may appear counterintuitive, it reflects lower driver cost burdens, not better matching. Revenues reach their highest point at $\lambda = 8$ and then decline when the system becomes oversupplied. Correspondingly, payoff variance plunges from over 1,000 at $\lambda = 2$ down to about 200 at high supply, indicating that the rule also stabilizes driver earnings by limiting extreme wait-time swings.

Overall, TimePNE injects realistic impatience by removing drivers after a fixed wait, which sharply reduces waiting costs and boosts net revenue—but this comes at the expense of total matches. While it stabilizes payoffs and caps queue lengths, throughput and service probability remain bounded below the demand rate, and excess supply simply increases abandonment rather than completed trips.

4.4 Cross-rule performance comparison and final insights

Having examined each acceptance rule in isolation, we now turn to a side-by-side comparison that highlights how drivers' behavioral strategies shape system-level outcomes across varying supply regimes. By plotting key metrics—throughput (TP), service probability (ServProb), average driver payoff (AvgDrvPay), queue length (AvgQLen), net revenue (NetRev), payoff variance (VarDrvPay), and others—against driver supply (λ), we can identify systematic trade-offs between efficiency, fairness, and platform profitability. In what follows, we organize our discussion around three market conditions (undersupply, balanced supply, oversupply) and contrast groups of rules by their "filtering aggressiveness" (AlwaysAccept vs. StrictCut), strategy sophistication (NE and its probabilistic relaxations), and patience constraints (DrivPNE, TimePNE).



Undersupply ($\lambda \leq 5$). When driver supply is low relative to demand, all rules achieve nearly the same throughput—with the exception of TimePNE(30), which matches slightly fewer trips overall. However, service probabilities diverge sharply from NE under several rules. Both AlwaysAccept and StrictCut(50) always match every trip, driving their service rates to 1.0, whereas NE deliberately holds service probability down. In contrast, DrivPNE overserves relative to NE (about 0.54 at $\lambda = 2$ and 0.61 at $\lambda = 5$ versus NE's 0.25 and 0.55), while TimePNE(30) markedly underserves (only about 0.14 at $\lambda = 2$ and 0.18 at $\lambda = 5$). The two probabilistic-threshold rules—ProbNE and ProbBinNE—track NE most closely, deviating by only a few percentage points. Overserving rules like AlwaysAccept and DrivPNE boost total earnings but increase variability, while underserving rules like TimePNE sacrifice overall matches and driver income in exchange for more consistent (lower-variance) pay.

Balanced supply ($\lambda \approx 8 - 10$). AlwaysAccept continues to overserve—matching nearly every trip even as λ rises—while NE holds service around 0.70–0.75. This drives down average driver pay and spikes payoff variance. StrictCut(50), by contrast, severely underserves, yielding rising queues and dramatically lower earnings and efficiency. Among the bounded-rational rules, ProbNE and ProbBinNE stay closest to NE, sacrificing a few percentage points of service and resulting in slightly higher payoff dispersion. DrivPNE and TimePNE(30) also underserve versus NE but far

less drastically than StrictCut. These rules strike intermediate trade-offs, keeping driver pay and queue lengths nearer to NE levels while tolerating slightly lower throughput.

Oversupply ($\lambda > 10$). StrictCut(50) serves a small portion of demand, inducing prohibitively long queues and huge negative net revenues, while AlwaysAccept maximizes matches and keeps queues minimal with revenues plunging slightly in the negatives. Equilibrium-based rules (NE, ProbNE, ProbBinNE) sustain moderate queue sizes and negative revenue due to oversupply, service probability stabilizing at cap values; introducing randomness or bin-sensitivity helps contain extreme disparities in driver earnings. DrivPNE's individualized patience consistently curbs queue growth and smooths payoff distributions, while TimePNE achieves the tightest control on both queue length and income inequality by expelling drivers who exceed their wait threshold.

No single acceptance rule universally dominates across all metrics and supply regimes. Our simulations reaffirm that NE serves as the efficiency benchmark—it maximizes throughput and revenue under balanced supply, though oversupply amplifies inefficiencies. In extreme undersupply or oversupply, AlwaysAccept ensures full coverage with minimal queues, while StrictCut(50) rigidly sacrifices many matches once driver availability crosses its cutoff. Hybrid strategies—ProbNE, ProbBinNE, DrivPNE, and TimePNE—closely track NE's performance, each introducing controlled randomness or patience constraints to manage queues and stabilize pay without substantially lowering overall throughput. These deviations from perfect rationality reveal where efficiency, stability, and cost controls intersect.

Chapter 5

Benchmarking Behavioral Strategies

In this chapter, we present the second simulation, which aims to benchmark each relaxation of Nash Equilibrium behavior against the Nash equilibrium optimum. The goal is to identify the optimal bin and partition setups for each acceptance rule under realistic behavioral constraints. Building on the acceptance rules introduced earlier, we systematically compare each behavioral variation—ProbNE, ProbBinNE, DrivPNE, and TimePNE—with the NE baseline. Unlike previous simulations that varied the level of driver supply, this analysis keeps supply fixed and instead explores different patience levels and queue configurations. For each rule, we test all bin and partition designs to identify the most efficient combination. Key performance indicators such as throughput, and net revenue are recorded, along with the Price of Anarchy (PoA), which captures the cost of deviating from optimal NE behavior. This chapter provides a comparative framework for understanding which configuration minimizes efficiency losses for each behavioral rule.

5.1 Simulation focus and design

This chapter builds upon the behavioral simulation framework introduced earlier, using the same event-driven, randomized FIFO architecture to model interactions between drivers and riders. As before, drivers are queued and matched based on real-time system dynamics, and the platform's performance is assessed through key operational and economic metrics.

However, the scope of this simulation is narrower and more focused. Here, we test only four acceptance rules—ProbNE, ProbBinNE, DrivPNE, and TimePNE—each representing a principled relaxation of the Nash Equilibrium (NE) benchmark. The objective is to benchmark these near-rational strategies under controlled conditions: specifically, fixed driver supply and varying patience levels. By doing so, we evaluate how close each rule remains to NE outcomes, quantify their performance losses via Price of Anarchy metrics, and identify the most effective bin and partition configurations for each acceptance strategy. The goal is to identify for each acceptance rule which scenario combination achieves the highest system efficiency while remaining grounded in realistic driver behavior.

5.2 Algorithmic variations

This simulation builds on the same event-driven randomized FIFO framework developed in the previous chapter, but incorporates several key adjustments to align more closely with the equilibrium conditions described in the theoretical model and to better assess variations of the NE rule under more realistic system constraints.

5.2.1 Partition focus on [1, 2, 3]

A central focus of this simulation is on testing the partition configuration [1, 2, 3], which activates a randomized dispatch mechanism when $i^* > 1$ and $P = 1$. In the [1,2,3] partition with a single bin, any rule based on NE will lead all drivers to accept all trips, because every trip belongs to the single partition assigned to that bin. So from a trip selection perspective, there's no filtering or prioritization happening—all trips are equally valid targets.

This setup departs from traditional partitioning logic by grouping all locations into a single partition, thereby removing any spatial prioritization. The resulting configuration is supported under the randomized FIFO equilibrium framework and is known to significantly raise the cost of cherry-picking behavior: drivers can no longer rely on targeted acceptance to isolate high-value trips, as all offers are now uniformly randomized across locations. These fairness gains come at the cost of predictability and control. Drivers may face increased uncertainty in earnings and idle time since their ability to pick profitable matches is reduced. This can lead to higher dispersion in payoffs and longer average waiting times, particularly in oversupplied conditions.

Because of these trade-offs, the partition [1, 2, 3] is interesting to explore. It includes the extreme end of randomized dispatch design and allows for examination of how robust each acceptance rule is to uncertainty and reduced control. Comparing this partition against more traditional scenarios such as [[1], [2, 3]] or [[1], [2], [3]] helps clarify how far near-rational strategies can go in terms of performance.

5.2.2 Queue length bounding to approximate steady-state

To better replicate steady-state equilibrium behavior, we bound the maximum queue length. In theory, the queue stabilizes at a constant level Q^* when the system reaches equilibrium. Specifically, for our parameters where $i^* = 3$, $Q^* = n_3 = 360$ under the direct FIFO model (see Theorem 2, Sec. 3.2.3). However, in practice, to ensure consistent simulation coverage and reduce variability, we set the queue length cap to 400. This ensures that the system reliably reaches the target length without excessive sensitivity to short-term fluctuations.[14]

5.2.3 Equilibrium randomization for trips to i^*

A final algorithmic refinement concerns the treatment of trips to the threshold location i^* under the NE rule. In equilibrium, drivers in the final bin randomize over whether to accept a trip to

[14] The equilibrium queue length Q^* ensures that the queue neither grows indefinitely nor empties out, allowing throughput and payoffs to stabilize. In the oversupplied regime ($\lambda > \sum_i \mu_i$), the system reaches a steady state where all trips are completed, and the queue length stabilizes at $Q^* = n_{i^*}$. For $i^* = 3$, this means $Q^* = n_3$, where n_3 is the number of drivers needed to serve location 3 demand in equilibrium.

location i^* with a probability calibrated to match the system's feasible service rate. Specifically, if $d = i^*$ (where d is the requested destination), the probability of accepting the trip is given by:

$$\frac{\min\left(\lambda - \sum_{i<i^*} \mu_i, \mu_{i^*}\right)}{\mu_{i^*}}.$$

This logic is encoded directly in the simulation code and controls whether a drivers serves a rider at i^* or not.[15]

5.3 Results and analysis

The following section presents the results of the refined simulation under fixed drivers supply and varying patience levels. We analyze system performance across five acceptance rules—NE, ProbNE, ProbBinNE, DrivPNE, and TimePNE(30)—using a consistent setup with three location types [1, 2, 3], trip rewards [75, 25, 15], and a uniform cost of waiting set to $c = 1/3$. The rider demand vector is fixed at [1, 6, 3], representing heterogeneous destination popularity, while the arrival rate of drivers is held at $\lambda = 8$. For each acceptance rule, we compare results across different bin configurations and location partitions to identify which scenario yields the best balance of throughput, efficiency, and fairness. We begin by introducing the key performance metrics used to evaluate each scenario.

5.3.1 Performance metrics

The core performance indicators capture both operational efficiency and driver experience. These include Throughput (TP), Net Revenue, Average Waiting Time, and Average Driver Payoff. In addition, we compute Price of Anarchy metrics—PoA_TP and PoA_NetRev—to measure efficiency loss relative to the NE benchmark.

- **TP** (Throughput). Measures the number of successfully served ride requests per unit time.

[15] This follows the equilibrium mixed strategy described in the paper: drivers in the k^{th} bin accept all trips in partitions $\bigcup_{k'=1}^{k} \mathcal{L}^{(k')}$, but randomize over i^* when in the final bin, ensuring the acceptance rate matches the residual system demand.

- **NetRev** (Net Revenue). captures platform earnings after deducting driver compensation based on queue time.
- **AvgWait** (Average Waiting Time). reflects how long drivers wait in the queue before being matched.
- **AvgPayoff** (Average Driver Payoff). represents the average net earnings of drivers, accounting for both rewards and waiting costs.
- **PoA_TP** (Price of Anarchy – TP). evaluates the efficiency loss in throughput compared to the NE benchmark. Calculated as: PoA_{TP} = (TP under rule)/(TP under NE).
- **PoA_NetRev** (Price of Anarchy – Net Revenue). evaluates revenue efficiency loss compared to NE. Calculated as: $\text{PoA}_{\text{NetRev}}$ = (NetRev under rule)/(NetRev under NE).

Together, these metrics enable a clear benchmarking of each acceptance rule's efficiency.

5.3.2 Nash equilibrium benchmark

The first step in our simulation was to validate the Nash Equilibrium (NE) rule under all tested configurations. This ensures a solid benchmark for comparing the efficiency of alternative acceptance strategies. As shown, NE consistently delivers the first-best throughput and the second-best net revenue across all partition types and bin designs.

Rule	P	λ	i*	Partitions	Bins	TP	NetRev	AvgWait	AvgPayoff	PoATP	PoANR
NE	1	8	3	[[1, 2, 3]]	[(0, 360)]	7.99	123.67	43.2	15.47	1	1
NE	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7.83	113	47.92	14.37	1	1
NE	2	8	3	[[1], [2, 3]]	[(0, 0), (180, 360)]	7.98	153.47	31.99	19.21	1	1
NE	3	8	3	[[1], [2], [3]]	[(0, 0), (150, 150), (360, 360)]	7.83	111.49	47.83	14.17	1	1

These results are consistent with the theoretical findings: NE maximizes completed trips while maintaining fairness and individual rationality. The net revenue achieved is optimal under positive waiting costs—specifically, the second-best outcome when $c > 0$, as drivers incur opportunity costs while queuing. The effective NE behavior is expected to yield throughput (TP) values as close as possible to 8, which represents the total rider arrival rate. For net revenue comparisons, we can take the average of NE net revenues across partitions (≈ 125.4) as a reference point when evaluating whether other acceptance rules appear to outperform NE due to stochastic fluctuations . The small discrepancies observed (e.g., TP slightly below 8) are attributed to simulation noise, not deviations from equilibrium behavior. Confirming the NE benchmark is essential for evaluating

the Price of Anarchy, as it represents the upper bound of performance under rational decision-making.

5.3.3 Comparative performance by acceptance rule

This section presents a comparative analysis of the four NE-based acceptance rules—ProbNE, ProbBinNE, DrivPNE, and TimePNE(30). Using consistent simulation settings, we evaluate how these rules perform across various bin and partition schemes. Results are assessed in terms of throughput, net revenue, waiting times, and driver payoffs, with deviations from NE outcomes quantified using Price of Anarchy metrics. This comparison highlights the operational and economic trade-offs introduced by each rule and identifies which configurations most closely approach NE efficiency.

ProbNE. Behaves identically to the rule introduced in the previous chapter, where drivers accept trips with 80% probability if $k_{\mathcal{L}} \leq k_b$, and 20% when $k_{\mathcal{L}} > k_b$, where $k_{\mathcal{L}}$ is the trip partition index and k_b is the driver's bin index.

Rule	P	λ	i*	Partitions	Bins	TP	NetRev	AvgWait	AvgPayoff	PoATP	PoANR
ProbNE	1	8	3	[[1, 2, 3]]	[(0, 360)]	6.4	58.69	61.74	9.23	0.8	0.47
ProbNE	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7.52	97.85	51.38	12.99	0.96	0.87
ProbNE	2	8	3	[[1], [2, 3]]	[(0, 0), (180, 360)]	6.78	76.65	58.03	11.28	0.85	0.5
ProbNE	3	8	3	[[1], [2], [3]]	[(0, 0), (150, 150), (360, 360)]	7.6	102.02	50.65	13.37	0.97	0.92

Throughput (TP) remains strong overall, peaking at 7.6–7.52. The PoA_TP values confirm this, ranging from 0.8 to 0.97. Net revenue declines compared to NE, with values between 58.69 and 102.02, reflecting losses due to mismatches and partially accepted trips. The corresponding PoA_NetRev ranges from 0.47 to 0.92, underscoring that while ProbNE maintains respectable efficiency, it can incur steep opportunity costs under less favorable partitioning (e.g., [[1, 2, 3]]). AvgWait stays elevated (above 50), indicating that the probabilistic rejection of even desirable trips leads to longer queuing times. AvgPayoff is also reduced relative to NE, staying in the 9.23–13.37 range.

Among all tested configurations, the [1, 2, 3] partition consistently performs worst across every metric. This outcome stems from the absence of spatial structure: drivers in all bins face the same wide distribution of destinations, leading to more frequent probabilistic rejections. Since ProbNE

does not prioritize specific trip types within such a flat partition, riders are frequently mismatched or delayed, inflating wait times (61.74), lowering net revenue (58.69), and reducing match rates (TP = 6.4). These inefficiencies are directly linked to the design of the partition, where the rule's probabilistic nature offers no corrective mechanism to counterbalance the lack of trip segmentation.

By contrast, the most efficient outcomes emerge under the most granular configuration—[[1], [2], [3]]—which yields the highest TP (7.6), NetRev (102.02), and highest AvgPayoff (13.37) among ProbNE setups. This structure aligns well with the rule's probabilistic logic: because drivers are assigned to specific bins and those bins are narrowly focused on particular destinations, the 80% acceptance probability is mostly directed toward relevant, high-incentive matches. Mismatches are minimized, and the system avoids the inefficient randomness seen in flatter partitions. In essence, ProbNE benefits most from clear spatial separation, where each bin is tightly associated with specific rider types. Without this structure—as in [1, 2, 3]—its stochastic nature becomes a liability, amplifying queue delays and reducing economic efficiency.

ProbBinNE. This rule builds on the logic of ProbNE but assigns bin-specific acceptance probabilities—90%, 75%, or 60% (bin 1 to 3 respectively) for trips where $k_{\mathcal{L}} \leq k_b$, and 10%, 30%, or 50% for trips where $k_{\mathcal{L}} > k_b$. While it retains the stochastic foundation of ProbNE, this added differentiation is intended to better align acceptance behavior with incentive gradients across bins.

Rule	P	λ	i*	Partitions	Bins	TP	NetRev	AvgWait	AvgPayoff	PoATP	PoANR
ProbBinNE	1	8	3	[[1, 2, 3]]	[(0, 360)]	7.21	85.01	54.14	11.8	0.9	0.69
ProbBinNE	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7.58	102.14	50.79	13.43	0.97	0.9
ProbBinNE	2	8	3	[[1], [2, 3]]	[(0, 0), (180, 360)]	6.37	67.81	61.89	10.65	0.8	0.44
ProbBinNE	3	8	3	[[1], [2], [3]]	[(0, 0), (150, 150), (360, 360)]	7.2	91.23	54.31	12.63	0.92	0.82

Throughput (TP) remains robust, ranging from 6.37 to 7.58, mirroring the performance of ProbNE. However, Net Revenue shows a slightly narrower and more stable range—from 67.81 to 102.14—suggesting improved alignment between rider offers and driver preferences due to bin-specific probabilities. Price of Anarchy metrics similarly reflect moderate efficiency loss, with PoA_TP between 0.80 and 0.97 and PoA_NetRev from 0.44 to 0.90. While AvgWait remains elevated across all scenarios, it is slightly more consistent than in ProbNE, indicating more predictable queue dynamics. AvgPayoff also improves marginally in most partitions, pointing to better payoff distribution among drivers.

Across the four tested partitions, the split [1, 2], [3] emerges as the clear optimum: grouping the two most popular locations (1 2) into Bin 1 aligns the 90 % acceptance rate with the bulk of demand, resulting in the highest throughput (TP = 7.58), net revenue (102.14), and average driver payoff (13.43). This configuration also minimizes wasted offers to lower-value trips, keeping average wait times relatively low (50.79) and preserving 97 % of the NE throughput benchmark.

By contrast, the partition [1], [2, 3] performs worst: Bin 2 must serve both trips to locations 2 and 3 under a 75 % acceptance rule, inflating mismatches. Here, throughput falls to 6.37 (PoA_TP = 0.80), net revenue collapses to 67.81 (PoA_NetRev = 0.44), and drivers endure the longest queues (AvgWait = 61.89). The flat single-bin case [1, 2, 3] and the fully granular split [1], [2], [3] produce intermediate outcomes—better than [1], [2, 3] but not as strong as [1, 2], [3]—highlighting that ProbBinNE's stochastic acceptance benefits most from partitions that concentrate high-probability matches on the largest demand clusters.

DrivPNE. This rule models bounded rationality through a deterministic patience structure. Drivers in Bin 1 accept trips to Location 1 immediately, to Location 2 after 1 decline, and to Location 3 after 2 declines. Drivers in Bin 2 accept Location 1 and 2 immediately and Location 3 after 1 decline, while drivers in Bin 3 accept Location 1, 2, and 3 immediately. Unlike the probabilistic rules, DrivPNE enforces acceptance through accumulated patience depletion rather than acceptance probability, reflecting time-sensitive decision-making.

Rule	P	λ	i*	Partitions	Bins	TP	NetRev	AvgWait	AvgPayoff	PoATP	PoANR
DrivPNE	1	8	3	[[1, 2, 3]]	[(0, 360)]	4.45	25.92	89.33	5.96	0.56	0.21
DrivPNE	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7.32	98.39	53.39	13.42	0.93	0.87
DrivPNE	2	8	3	[[1], [2, 3]]	[(0, 0), (180, 360)]	7.31	99.69	53.34	13.58	0.92	0.65
DrivPNE	3	8	3	[[1], [2], [3]]	[(0, 0), (150, 150), (360, 360)]	7.87	114.23	47.3	14.44	1.01	1.02

DrivPNE displays a more behaviorally constrained performance profile than the probabilistic rules, with Throughput (TP) ranging from 4.45 to 7.87 and Net Revenue between 25.92 and 114.23. These wider spreads indicate that performance is highly sensitive to partition structure. The Price of Anarchy metrics also reflect this volatility: PoA_TP spans from a low 0.56 to near-parity at 1.01, while PoA_NetRev ranges from just 0.21 to 1.02 (PoA measures > 1 are due to NE measurements not being in perfect steady-state, e.g. TP $\neq 8.0$, because of simulation noise). Compared to ProbNE and ProbBinNE, AvgWait is considerably more variable—reaching as high

as 89.33 when queue dynamics degrade—while AvgPayoff spans from 5.96 to 14.44. This confirms that under certain conditions, the rule's logic of incrementally relaxing driver constraints can either sharply hinder or closely match NE-like performance.

The configuration with the fully separated partition [[1],[2],[3]] emerges as the most effective, achieving a throughput (TP) of 7.87 and a net revenue of 114.23—slightly surpassing the NE benchmark due to minor simulation noise. This marginal overshoot in Price of Anarchy metrics (PoA_TP = 1.01, PoA_NetRev = 1.02) does not indicate a rule superiority over NE, but rather that the rule successfully mimics rational equilibrium behavior under favorable structural alignment. Drivers progressively align with less preferred but still viable offers, similarly to the behavior observed under NE, avoiding long idle periods and wasted opportunities. The result is a low average wait time (47.3) and the highest AvgPayoff (14.44) among all rules tested. In contrast, the single-bin configuration [1,2,3] performs worst across every metric, yielding the lowest TP (4.45), net revenue (25.92), and highest AvgWait (89.33). Here, all drivers face the same pool of trips and utilize the full patience window before accepting most offers, producing systemic delays and inefficient matching. Without structured trip differentiation, the mechanism fails to leverage its behavioral strengths.

DrivPNE performs best when bins are aligned with destination types, where its deterministic decline logic facilitates match efficiency. Its rule design not only maintains spatial rationality but also achieves almost-peak system efficiency under the right partitioning.

TimePNE(30). Under this rule, drivers remain in the queue for up to 30 time units, after which they exit the system if not matched. This introduces a realistic constraint, finite willingness to wait, which captures urgency-driven behavior and naturally penalizes dispatching delays.

Rule	P	λ	i*	Partitions	Bins	TP	NetRev	AvgWait	AvgPayoff	PoATP	PoANR
TimePNE(30)	1	8	3	[[1, 2, 3]]	[(0, 360)]	7.89	220.77	5.29	27.67	0.99	1.79
TimePNE(30)	2	8	3	[[1, 2], [3]]	[(0, 150), (360, 360)]	7	191.44	9.99	25.18	0.89	1.69
TimePNE(30)	2	8	3	[[1], [2, 3]]	[(0, 0), (180, 360)]	3.02	62.19	9.82	13.96	0.38	0.41
TimePNE(30)	3	8	3	[[1], [2], [3]]	[(0, 0), (150, 150), (360, 360)]	4	100.26	7.82	17.51	0.51	0.9

The TimePNE bounded queuing behavior leads to strikingly different system dynamics, especially in terms of Net Revenue. While Throughput (TP) varies from 3.02 to 7.89—less consistent than under NE-based rules—the standout pattern is the extraordinary surge in Net Revenue, which

spans from 62.19 to 220.77. This is not the result of better matching, but rather of drivers leaving the system before incurring significant wait costs, minimizing opportunity losses and boosting platform profit. Price of Anarchy (PoA) values capture this shift: PoA_NetRev peaks at 1.79, an unusual outcome justified by reduced waiting rather than increased trip efficiency. AvgWait is low across most setups (as low as 5.29), while AvgPayoff varies between 13.96 and 27.67—the highest range among all acceptance rules.

Among all partition setups, the [1, 2, 3] configuration produces the most striking results across every key metric. With drivers all eligible to serve the full trip set, the system maintains maximum flexibility in matching while ensuring drivers do not wait longer than 30 units. This setup achieves near-optimal throughput (TP = 7.89), minimal average waiting time (5.29), and the highest net revenue observed in the entire simulation (220.77). The exceptional PoA_NetRev of 1.79—greater than the NE benchmark—is fully justified here: it's not due to better matching but stems from reduced waiting losses under hard time caps. Drivers who are not matched promptly exit the system before incurring costly delays, allowing the platform to retain more trip value. This makes TimePNE(30) under [1, 2, 3] uniquely suited for maximizing revenue in oversupplied environments.

In contrast, the [1], [2, 3] partition performs worst across nearly all dimensions. Here, Bin 2 must absorb the bulk of demand and serve multiple destinations—yet with limited flexibility due to spatial separation. With strict time limits, many drivers time out before being matched efficiently, especially in Bin 2, which faces destination mismatches and sparse opportunities. This leads to the lowest throughput (TP = 3.02) and net revenue (62.19) among all scenarios tested for TimePNE(30). AvgWait rises (9.82), and while net revenue doesn't collapse entirely, the system's inefficiency is clear: PoA_TP falls to 0.38, and PoA_NetRev drops to 0.41, indicating severe underperformance relative to NE. In summary, TimePNE(30) thrives under unified partitions like [1, 2, 3], where flexibility compensates for early driver exits. In fragmented partitions like [1], [2, 3], however, the strict time constraints amplify mismatches and block trip completion, drastically reducing system efficiency.

5.4 Summary and implications

This chapter has examined a set of near-rational acceptance rules under a refined simulation model grounded in the randomized FIFO dispatch mechanism. By holding driver supply fixed and varying patience, we assessed results for different behavioral constraints—ranging from stochastic acceptance to deterministic patience thresholds—under all possible bin and trip configurations. Across all scenarios, Nash Equilibrium served as a performance ceiling, consistently achieving first-best throughput and second-best net revenue. Simulation results show that queue segmentation plays a pivotal role in driving efficiency, but the optimal partitioning depends on the specific logic of each acceptance rule. ProbNE and DrivPNE share the same optimal partition—the fully separated structure [[1], [2], [3]]—which best supports their respective probabilistic and patience-based decision rules by aligning bins with distinct trip types. ProbBinNE, however, behaves less predictably: when a majority of high-value trips fall into Bin 1, where acceptance probabilities are highest, the system performs well—but this efficiency hinges on the distribution of demand aligning with the bin hierarchy. In contrast, TimePNE(30) consistently performs best under unified partitions like [1, 2, 3], where full flexibility compensates for limited driver patience, allowing the platform to retain more trip value under strict time constraints. Overall, efficiency emerges not solely from the rule, but from the fit between acceptance behavior and partition structure.

Chapter 6

Conclusions

In this dissertation, we developed and analyzed a family of near-rational acceptance rules for ride-sharing queues under a Randomized FIFO dispatch mechanism, originally proposed by Castro et al. (2021)[1], that attain a significant fraction of the efficiency under the Nash equilibrium and yet comply with realistic behavioral constraints. The mathematical basis was set in place in Chapter 2: we modeled the $M/M/c$ queue of strategic agents, used Erlang–C formulae to derive steady-state wait time distributions, and clarified the notions of Nash and subgame-perfect equilibrium in the context of the dispatch mechanisms. Chapter 3 outlined three particular dispatch variants—Strict FIFO, Direct FIFO, and Randomized FIFO—and carefully detailed the equilibrium operating aspects of each mechanism, summarizing the foundations presented in Castro et. al. (2021)[1], as well as the queue partitioning or threshold parameterizing technique for use in simulations. For the purposes of checking these findings in more realistic scenarios, Chapter 4 developed an event-driven simulation setting in which drivers follow bounded-rational acceptance rules. We evaluated the behavior of the main indicators relative to each strategy over a significant timescale across various ratios of supplies to demands parameters. Then, in Chapter 5, we set standards for the whole family of bounded-rational variants against the optimal Nash equilibrium

rule, studying each variant to determine its optimal queue partitions or thresholds relative to throughput and net revenue.

In every experimental setting, the Nash equilibrium represents the upper bound—attaining throughputs near the first-best and revenues close to the second-best (with deviations due to simulation noise). Alternative mechanisms introduce systematic variations reflecting bounded rationality and impatience: ProbNE/ProbBinNE replace equilibrium calculations with simple random choices, DrivPNE limits the number of rejections allowed to prevent endless cherry-picking, and TimePNE imposes time limits to capture real-time impatience. Though each alternative sacrifices a small amount of NE efficiency, TimePNE achieves greater net revenue than Nash equilibrium by substantially curtailing waiting-cost losses incurred by passengers—at the cost of higher abandonment rates, however. Underlying all of these results is the fact that Randomized FIFO was proven optimal only when drivers follow exact NE strategies—because its bin-and-partition structure perfectly aligns continuation values—yet real drivers often deviate from full rationality, using heuristics, misestimating wait times, or exhibiting inconsistent patience. To bridge that gap and preserve near-NE performance, the platform can anticipate boundedly rational behavior by adjusting bin thresholds, collapsing or expanding partitions, or even providing "nudges" in the driver interface to guide acceptance decisions. For instance, if many drivers accept high-value trips probabilistically (as in ProbBinNE), merging lower-value bins ensures more offers land in bins where acceptance is likely, preserving throughput and revenue. Consequently, the efficacy of these mechanisms is reinforced through strategic partitioning: under tight time constraints, a single undifferentiated pool (e.g., [1, 2, 3]) maximizes flexibility and revenue, but when drivers depart from ideal rationality, carefully segmented, destination-oriented categories (e.g., [1], [2, 3]) nudge them toward underserved trips, dampen supply-demand imbalances, and almost restore all of the Nash equilibrium's throughput while reducing volatility. Ultimately, thoughtful partition design is crucial for maximizing efficiency when drivers deviate from perfect rationality under a Randomized FIFO dispatch system.

Bibliography

[1] Castro, Francisco, et al. "Randomized FIFO Mechanisms." arXiv.Org, 21 Nov. 2021.

[2] Erlang, A. The theory of probabilities and telephone conversations. Nyt Tidsskrift for Matematik B 20 (1909), 33–39.

[3] Adan, Ivo, and Jacques Resing. Queueing Systems, Department of Mathematics and Computing Science Eindhoven University of Technology, 26 Mar. 2015.

[4] Green, Linda. Queueing Theory and Modeling, Graduate School of Business, Columbia University, New York.

[5] Fiveable. 8.3 M/M/1 and M/M/c queues – Stochastic Processes. Edited by Becky Bahr, Fiveable, 2024.

[6] Sztrik, János, et al. Basic Queueing Theory, University of Debrecen, Faculty of Informatics, Dec. 2012.

[7] Zukerman, Moshe. "Introduction to Queueing Theory and Stochastic Teletraffic Models." EE Department City University of Hong Kong.

[8] Rubino, G. "Basics on Queues." INRIA / IRISA, Rennes, France, Feb. 2006.

[9] Holler, Manfred J. Classical, Modern and New Game Theory, 4 Aug. 2001.

[10] Hotz, Heiko. A Short Introduction to Game Theory, LMU Munich, 2006.

[11] Gurrien, Bernard. "On the Current State of Game Theory." Real-World Economics Review, [l'Université Paris 1, France], 2018.

[12] Cave, Jonathan. Introduction to Game Theory, The Rand Graduate School, Apr. 1987.

[13] Karlin, Anna R., and Yuval Peres. Game Theory, Alive, 13 Dec. 2016.

[14] Osborne, Martin J., and Ariel Rubinstein. A Course in Game Theory, Massachusetts Institute of Technology, 1994.

[15] Bonanno, Giacomo. Game Theory, Third Edition, 2024.

[16] Angelopoulos, Konstantinos, et al. First-and Second-Best Allocations Under Economic and Environmental Uncertainty, University of Glasgow, Athens University of Economics and Business, CESifo, Munich., 29 Oct. 2010.

[17] Sethi, Rajiv, and Jörgen Weibull. "What Is Nash Equilibrium?" Edited by Cesar E. Silva, The Graduate Student Section.

[18] Osborne, Martin J. An Introduction to Game Theory, Department of Economics, University of Toronto Toronto, Canada , 6 Nov. 2000.

Appendix A

Simulation Codes and Metrics

In this appendix, we present the full Python code for both simulation frameworks and the key performance metrics used to evaluate their outputs. First, we introduce each metric's mathematical definition and interpretation, and then the complete simulation scripts that generate these metrics under the different acceptance strategies.

Performance metrics

TP	$\frac{\#\{\text{matched riders}\}}{T}$	Riders served per unit time.
Bin_k_p	$\frac{\#\{\text{accepted offers in bin } k\}}{\#\{\text{offers sent in bin } k\}}$	Acceptance rate within the k^{th} bin.
P_match	$\frac{\#\{\text{served riders}\}}{\#\{\text{offers sent (all bins)}\}}$	Overall probability an offer leads to a match.
DrivUtil	$\frac{1}{NT}\sum_{d=1}^{N}\text{busy\_time}_d$	Fraction of total driver-time spent on trips.

ServProb	$\frac{\#\{\text{accepted offers}\}}{\#\{\text{offers sent}\}}$	Share of offers that drivers accept.
CRate	$1 - \text{ServProb}$	Share of offers explicitly declined.
ExRate	$\frac{\#\{\text{expired offers}\}}{\#\{\text{offers sent}\}}$	Share of offers that time out (no response).
AvgQLen	$\frac{1}{T}\int_0^T Q(t)dt$	Time-average queue length (riders waiting).
NetRev	$\frac{\sum_{\text{served}} w_{\text{dest}}}{T} - c_p(\text{AvgQLen})$	Fare revenue rate minus waiting-cost rate.
AvgDrvPay	$\frac{1}{N}\sum_{d=1}^{N}\left(\sum_{\text{matches}_d} w_{\text{dest}} - c\tau_d\right)$	Mean net payoff per driver (earnings minus waiting cost).
VarDrvPay	$\frac{1}{N}\sum_{d=1}^{N}(\pi_d - \bar{\pi})^2$ where $\pi_d = \sum w_{\text{dest}} - c\tau_d, \quad \bar{\pi} = \frac{1}{N}\sum \pi_d$	Population variance of driver payoffs (inequality measure).

Below is the Python code for the first simulation, which tests the behavior of the various acceptance rules (AlwaysAccept, StrictCut, NE, ProbNE, ProbBinNE, DrivPNE, and TimePNE) and records the associated performance metrics.

```python
import heapq
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display
from IPython.display import FileLink

# ── 1) Partition helper (returns partitions, i_star) ──────────────────────

def generate_partitions(mu, lam, P):
    cum_mu = np.cumsum(mu)
    i_star = np.searchsorted(cum_mu, lam, side='left') + 1
    i_star = min(i_star, len(mu))
    m = min(i_star, P)
    base, extra = divmod(i_star, m)

    parts, start = [], 1
    for _ in range(m):
        size = base + (1 if extra>0 else 0)
        extra -= 1
        parts.append(list(range(start, start+size)))
        start += size
```

```python

    if len(parts) == 2:
        flat = parts[0] + parts[1]
        parts = [
            flat[:-1],
            [flat[-1]]
        ]

    return parts, i_star

# ——— 2) Bin-boundary computation ———————————————————

def compute_bin_bounds(mu, w, c, partitions):
    mu = np.array(mu, float)
    w  = np.array(w,  float)
    bins = []
    for k, part in enumerate(partitions):
        w_min = min(w[i-1] for i in part)
        lo = sum((w[i-1]-w_min)*mu[i-1]
                 for j in range(k) for i in partitions[j]) / c
        hi = sum((w[i-1]-w_min)*mu[i-1]
                 for j in range(k+1) for i in partitions[j]) / c
        lo = int(np.floor(lo))
        hi = int(np.ceil (hi))
        bins.append((lo, hi))
    return bins

# ——— 3) Acceptance-rule classes ———————————————————————————

class AcceptanceRule:
    def accept(self, driver, rider, bin_idx, pos, now=None):
        raise NotImplementedError

class AlwaysAccept(AcceptanceRule):
    def accept(self, driver, rider, bin_idx, pos, now=None):
        return True

class StrictCut(AcceptanceRule):
    def __init__(self, C): self.C = C
    def accept(self, driver, rider, bin_idx, pos, now=None):
        return pos <= self.C

class NE(AcceptanceRule):
    def accept(self, driver, rider, bin_idx, pos, now=None):
        for p_idx, part in enumerate(driver.partitions):
            if (rider.dest + 1) in part:
                break
        else:
            return False

        return p_idx <= bin_idx

class ProbNE(AcceptanceRule):
    def accept(self, driver, rider, bin_idx, pos, now=None):
        for p_idx, part in enumerate(driver.partitions):
            if (rider.dest + 1) in part:
                break
        else:
            return False

        p_accept = 0.8 if p_idx <= bin_idx else 0.2
        return np.random.rand() < p_accept

class ProbBinNE(AcceptanceRule):
    def accept(self, driver, rider, bin_idx, pos, now=None):
        for p_idx, part in enumerate(driver.partitions):
            if (rider.dest + 1) in part:
                break
        else:
            return False
        high_probs = [0.9, 0.75, 0.6]
        low_probs  = [0.1, 0.30, 0.5]
        p_accept = high_probs[bin_idx] if p_idx <= bin_idx else low_probs[bin_idx]
        return np.random.rand() < p_accept

class DrivPNE(AcceptanceRule):
    def __init__(self):
        self.patience = {
```

```
            0: {0:0, 1:1, 2:2},
            1: {1:0, 2:1},
            2: {2:0}
        }
    def accept(self, driver, rider, bin_idx, pos, now=None):
        loc = rider.dest
        max_declines = self.patience.get(bin_idx, {}).get(loc,0)
        if driver.declines < max_declines:
            driver.declines += 1
            return False
        return True

class TimePNE(AcceptanceRule):
    def __init__(self, T=30):
        self.T = T

    def accept(self, driver, rider, bin_idx, pos, now=None):
        if now is not None and (now - driver.join_time) > self.T:
            return None
        for p_idx, part in enumerate(driver.partitions):
            if (rider.dest + 1) in part:
                return p_idx <= bin_idx
        return False

# ——— 4) Event-driven randomized-FIFO simulator ———————————————————

class Driver:
    def __init__(self, join_time, w, bins, partitions):
        self.join_time = join_time
        self.earnings  = 0.0
        self.wait_cost = 0.0
        self.declines  = 0
        self.bins       = bins
        self.partitions = partitions
        self.w          = w

    def serve(self, dest, now, w, c):
        wait = now - self.join_time
        self.wait_cost += c * wait
        self.earnings  += w[dest]

class Rider:
    def __init__(self, dest, P):
        self.dest     = dest
        self.patience = P
        self.declines = 0

class Event:
    def __init__(self, t, kind):
        self.t, self.kind = t, kind
    def __lt__(self, other):
        return self.t < other.t

def simulate_random_fifo(mu, w, c, cp, lam, P,
                         partitions, bin_bounds,
                         rule: AcceptanceRule,
                         T_max=50000):

    now = 0.0
    prev = 0.0
    queue, drivers = [], []
    m = len(partitions)
    total_reqs    = np.zeros(m, int)
    served_by_bin = np.zeros((m, m), int)
    offers_by_bin = np.zeros(m, int)
    dest_to_part = { d1-1: p_idx
                for p_idx, part in enumerate(partitions)
                for d1 in part }
    ev = []

    heapq.heappush(ev, Event(np.random.exponential(1/lam),    'driver_arr'))
    heapq.heappush(ev, Event(np.random.exponential(1/sum(mu)), 'rider_arr'))

    warmup_horizon = 5.0
    while ev and now < warmup_horizon:
        e = heapq.heappop(ev)
        now = e.t
        if isinstance(rule, TimePNE):
```

```
            queue = [d for d in queue if now - d.join_time <= rule.T]

        if e.kind == 'driver_arr':
            d = Driver(join_time=now, w=w, bins=bin_bounds, partitions=partitions)
            drivers.append(d)
            queue.append(d)
            heapq.heappush(ev, Event(now + np.random.exponential(1/lam), 'driver_arr'))
        else:
            dest = np.random.choice(len(mu), p=np.array(mu)/sum(mu))
            rider = Rider(dest, P)
            heapq.heappush(ev, Event(now + np.random.exponential(1/sum(mu)), 'rider_arr'))
            for k,(lo,hi) in enumerate(bin_bounds):
                if rider.declines >= P:
                    break
                if not queue:
                    continue
                idxs = [i for i in range(len(queue)) if lo <= i <= hi]
                if not idxs:
                    continue
                pick = np.random.choice(idxs)
                d    = queue[pick]
                if rule.accept(d, rider, k, pick, now):
                    d.serve(dest, now, w, c)
                    queue.pop(pick)
                    break
                else:
                    rider.declines += 1

    len_prewarm = len(drivers)
    now = prev = warmup_horizon
    stats = {'served':0, 'riders':0, 'cancelled':0, 'offers':0, 'patience_exhausted': 0}
    total_earn = 0.0
    queue_time  = 0.0
    driver_arrivals = 0

    while ev and now < T_max:
        e = heapq.heappop(ev)
        now = e.t
        dt  = now - prev
        queue_time += len(queue)*dt
        prev = now
        if isinstance(rule, TimePNE):
            queue = [d for d in queue if now - d.join_time <= rule.T]

        if e.kind == 'driver_arr':
            d = Driver(now, w, bin_bounds, partitions)
            drivers.append(d)
            queue.append(d)
            driver_arrivals += 1
            heapq.heappush(ev, Event(now + np.random.exponential(1/lam), 'driver_arr'))

        else:
            stats['riders'] += 1
            dest = np.random.choice(len(mu), p=np.array(mu)/sum(mu))
            rider = Rider(dest, P)
            dest_part = dest_to_part.get(dest, None)
            if dest_part is None:
                stats['cancelled'] += 1
                heapq.heappush(ev, Event(now + np.random.exponential(1/sum(mu)), 'rider_arr'))
                continue
                total_reqs[dest_part] += 1
            heapq.heappush(ev, Event(now + np.random.exponential(1/sum(mu)), 'rider_arr'))

            for k,(lo,hi) in enumerate(bin_bounds):
                if not queue:
                    break
                idxs = [i for i in range(len(queue)) if lo <= i <= hi]
                if not idxs:
                    continue
                offers_by_bin[k] += 1
                stats['offers'] += 1
                pick = np.random.choice(idxs)
                d    = queue[pick]
                decision = rule.accept(d, rider, k, pick, now)
                if decision is None:
                    queue.pop(pick)
                    continue
```

```
                elif decision:
                    stats['served'] += 1
                    d.serve(dest, now, w, c)
                    total_earn     += w[dest]
                    queue.pop(pick)
                    if dest_part is not None:
                        served_by_bin[dest_part, k] += 1
                    break
                else:
                    rider.declines += 1
                    if rider.declines >= P:
                        stats['patience_exhausted'] += 1
                        break

    T       = T_max
    served  = stats['served']
    if stats['offers'] > 0:
        sp = served / stats['offers']
    else:
        sp = np.nan
    sp = float(np.clip(sp, 0.0, 1.0))
    cr = float(np.clip(1 - sp, 0.0, 1.0))
    drivers_meas = drivers[len_prewarm:]
    util_count = sum(1 for d in drivers_meas if d.earnings > 0)
    driver_util  = util_count / driver_arrivals
    patience_rate = stats['patience_exhausted'] / stats['riders']
    payoffs = np.array([d.earnings - d.wait_cost for d in drivers_meas])

    return {
        'throughput':        served / T,
        'service_prob':      sp,
        'cancel_rate':       cr,
        'driver_utilization':       driver_util,
        'patience_exhaustion_rate': patience_rate,
        'avg_driver_payoff': payoffs.mean() if payoffs.size else np.nan,
        'var_driver_payoff': payoffs.var()  if payoffs.size else np.nan,
        'avg_queue_len':     queue_time / T,
        'net_revenue':       (total_earn / T) - cp * (queue_time / T),
        'total_reqs':        total_reqs,
        'served_by_bin':     served_by_bin,
        'offers_by_bin':     offers_by_bin
    }

# ── 5) All rules test ──────────────────────────────────────────

np.random.seed(0)

mu         = [1,6,3]
w          = [75,25,15]
c, cp      = 1/3, 1/3
P          = 2
lam_values = [1,2,5,8,10,12,15]

rules = {
    '1_AlwaysAccept':        AlwaysAccept(),
    '2_StrictCut(50)':       StrictCut(50),
    '3_NE':                  NE(),
    '4_ProbNE':              ProbNE(),
    '5_ProbBinNE':           ProbBinNE(),
    '6_DrivPNE':             DrivPNE(),
    '7_TimePNE(30)':         TimePNE(T=30),
}

records = []
accept_records = []
for name, rule in rules.items():
    for lam in lam_values:
        parts, i_star    = generate_partitions(mu, lam, P)
        bin_bounds       = compute_bin_bounds(mu, w, c, parts)
        out              = simulate_random_fifo(mu, w, c, cp, lam, P, parts, bin_bounds,rule, T_max=10000)
        offers = out['offers_by_bin']
        served = out['served_by_bin']
        O1 = offers[0]
        A1 = served[:,0].sum()
        ps = []
        for k in range(len(offers)):
            Ok = offers[k]
            Ak = served[:, k].sum()
```

```
            pk = Ak / Ok if Ok > 0 else np.nan
            ps.append(pk)
        P_match = 0.0
        prod_fail = 1.0
        for pk in ps:
            P_match += prod_fail * (pk if not math.isnan(pk) else 0.0)
            prod_fail *= 1 - (pk if not math.isnan(pk) else 0.0)
        bin_info = [(lo, hi) for lo, hi in bin_bounds]

        records.append({
            'Rule':          name,
            'P':             P,
            'λ':             lam,
            'i*':            i_star,
            'Partitions':    parts,
            'Bins':          bin_info,
            'TP':            round(out['throughput'],1),
            'Bin1_p':        round(ps[0],2),
            'Bin2_p':        round(ps[1],2) if len(ps)>1 else float('nan'),
            'P_match':       round(P_match,2),
            'DrivUtil':      round(out['driver_utilization'],2),
            'ServProb':      round(out['service_prob'],2),
            'CRate':         round(out['cancel_rate'],2),
            'ExRate':        round(out['patience_exhaustion_rate'],2),
            'AvgQLen':       round(out['avg_queue_len'],2),
            'NetRev':        round(out['net_revenue'],2),
            'AvgDrvPay':     round(out['avg_driver_payoff'],2),
            'VarDrvPay':     round(out['var_driver_payoff'],2),
        })

        total_reqs    = out['total_reqs']
        for p_idx in range(len(parts)):
            total_accepted = served[p_idx].sum()
            for b_idx in range(len(parts)):
                pct = (served[p_idx, b_idx] / total_accepted * 100) \
                if total_accepted>0 else 0.0
                accept_records.append({
                    'Rule':      name,
                    'λ':         lam,
                    'Partition': f"L{p_idx+1}",
                    'Bin':       f"Bin{b_idx+1}",
                    'AcceptPct': round(pct,1)
                })

accept_df = pd.DataFrame(accept_records)
wide = (
    accept_df
    .pivot(index=['Rule','λ'], columns=['Partition','Bin'], values='AcceptPct')
    .fillna(0)
)
wide = wide.sort_index(axis=1, level=[0,1])
wide.columns = [f"{part}_{bin}" for part,bin in wide.columns]
wide = wide.reset_index()
display(wide)

df = pd.DataFrame(records)
display(df)

# ——— 6) Visual Comparison ——————————————————————————————————

metrics = ['TP', 'ServProb', 'CRate', 'DrivUtil', 'AvgDrvPay', 'VarDrvPay', 'NetRev', 'ExRate', 'AvgQLen',
'P_match']

n_metrics = len(metrics)
ncols     = 5
nrows     = math.ceil(n_metrics / ncols)

# Combined grid
fig, axes = plt.subplots(2, 5, figsize=(25, 11))
axes = axes.flatten()

TITLE_FS = 16
LABEL_FS = 16
TICK_FS  = 14
LEG_FS   = 10
SUPTITLE_FS = 22

for ax, metric in zip(axes, metrics):
```

```
    pivot = df.pivot(index='λ', columns='Rule', values=metric)
    pivot.plot(ax=ax, marker='o', legend=False)
    ax.set_xlabel('Driver supply λ', fontsize=LABEL_FS)
    ax.set_ylabel(metric,            fontsize=LABEL_FS)
    ax.tick_params(axis='both', labelsize=TICK_FS)

handles, labels = axes[0].get_legend_handles_labels()
fig.legend(
    handles,
    labels,
    loc="upper center",
    ncol=7,
    frameon=False,
    fontsize=16,
    bbox_to_anchor=(0.5, 0.935),
    bbox_transform=fig.transFigure
)

for ax in axes[len(metrics):]:
    ax.set_visible(False)

fig.suptitle("Comparison of All Acceptance Rules", fontsize=SUPTITLE_FS)
plt.tight_layout(rect=[0, 0.03, 1, 0.92])
plt.show()

# Separate graphs per acceptance rule
for rule in df['Rule'].unique():
    sub = df[df['Rule'] == rule]
    fig, axes = plt.subplots(nrows, ncols, figsize=(20, 4*nrows))
    axes = axes.flatten()
    for ax, metric in zip(axes, metrics):
        ax.plot(sub['λ'], sub[metric], marker='o', linestyle='-')
        ax.set_xlabel('Driver supply (λ)')
        ax.set_ylabel(metric)
        ax.grid(True)
    for ax in axes[len(metrics):]:
        ax.set_visible(False)
    fig.suptitle(f"Performance of {rule}")
    fig.tight_layout(rect=[0, 0.03, 1, 0.96])
    filename = f'performance_{rule.replace(" ","_")}.png'
    fig.savefig(filename, dpi=300, bbox_inches='tight')
    display(FileLink(filename))
    plt.show()
```

The following section presents the Python code for the second simulation, which benchmarks these behavioral strategies under different partitioning and queue-bounding settings.

```
import math
import heapq
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from   IPython.display import display
from   IPython.display import FileLink

# ——— 1) Partition helper (returns partitions, i_star) ————————————————————
def generate_partitions(mu, lam, P):
    cum_mu = np.cumsum(mu)
    i_star = np.searchsorted(cum_mu, lam, side='left') + 1
    i_star = min(i_star, len(mu))
    m = min(i_star, P)
    base, extra = divmod(i_star, m)

    parts, start = [], 1
    for _ in range(m):
        size = base + (1 if extra>0 else 0)
        extra -= 1
        parts.append(list(range(start, start+size)))
        start += size

    if len(parts) == 2:
        flat = parts[0] + parts[1]
        parts = [
            flat[:-1],
```

```python
            [flat[-1]]
        ]

    return parts, i_star

# ── 2) Bin-boundary computation ──────────────────────────
def compute_bin_bounds(mu, w, c, partitions):
    mu = np.array(mu, float)
    w  = np.array(w,  float)
    bins = []
    for k, part in enumerate(partitions):
        w_min = min(w[i-1] for i in part)
        lo = sum((w[i-1]-w_min)*mu[i-1]
                 for j in range(k) for i in partitions[j]) / c
        hi = sum((w[i-1]-w_min)*mu[i-1]
                 for j in range(k+1) for i in partitions[j]) / c
        lo = int(np.floor(lo))
        hi = int(np.ceil (hi))
        bins.append((lo, hi))
    return bins

# ── 3) Acceptance-rule classes ──────────────────────────────────
class AcceptanceRule:
    def accept(self, driver, rider, bin_idx, pos, now=None):
        raise NotImplementedError

class NE(AcceptanceRule):
    def accept(self, driver, rider, bin_idx, pos, now=None):
        for p_idx, part in enumerate(driver.partitions):
            if (rider.dest + 1) in part:
                break
        else:
            return False

        return p_idx <= bin_idx

class ProbNE(AcceptanceRule):
    def accept(self, driver, rider, bin_idx, pos, now=None):
        for p_idx, part in enumerate(driver.partitions):
            if (rider.dest + 1) in part:
                break
        else:
            return False

        p_accept = 0.8 if p_idx <= bin_idx else 0.2
        return np.random.rand() < p_accept

class ProbBinNE(AcceptanceRule):
    def accept(self, driver, rider, bin_idx, pos, now=None):
        for p_idx, part in enumerate(driver.partitions):
            if (rider.dest + 1) in part:
                break
        else:
            return False
        high_probs = [0.9, 0.75, 0.6]
        low_probs  = [0.1, 0.30, 0.5]
        p_accept = high_probs[bin_idx] if p_idx <= bin_idx else low_probs[bin_idx]
        return np.random.rand() < p_accept

class DrivPNE(AcceptanceRule):
    def __init__(self):
        self.patience = {
            0: {0:0, 1:1, 2:2},
            1: {1:0, 2:1},
            2: {2:0}
        }
    def accept(self, driver, rider, bin_idx, pos, now=None):
        loc = rider.dest
        max_declines = self.patience.get(bin_idx, {}).get(loc,0)
        if driver.declines < max_declines:
            driver.declines += 1
            return False
        return True

class TimePNE(AcceptanceRule):
    def __init__(self, T=30):
        self.T = T

    def accept(self, driver, rider, bin_idx, pos, now=None):
        if now is not None and (now - driver.join_time) > self.T:
            return None
        for p_idx, part in enumerate(driver.partitions):
            if (rider.dest + 1) in part:
                return p_idx <= bin_idx
        return False
```

```python

# ——— 4) Event-driven randomized-FIFO simulator ——————————————————————
class Driver:
    def __init__(self, join_time, w, bins, partitions):
        self.join_time = join_time
        self.earnings  = 0.0
        self.wait_cost = 0.0
        self.total_wait = 0.0
        self.declines  = 0
        self.bins       = bins
        self.partitions = partitions
        self.w          = w

    def serve(self, dest, now, w, c):
        wait = now - self.join_time
        self.total_wait += wait
        self.wait_cost += c * wait
        self.earnings  += w[dest]

    @property
    def payoff(self):
        return self.earnings - self.wait_cost

class Rider:
    def __init__(self, dest, P):
        self.dest     = dest
        self.patience = P
        self.declines = 0

class Event:
    def __init__(self, t, kind):
        self.t, self.kind = t, kind
    def __lt__(self, other):
        return self.t < other.t

def simulate_random_fifo(mu, w, c, cp, lam, P, partitions, bin_bounds, rule: AcceptanceRule, l, i_star, T_max=50000):

    random.seed(l)
    now = 0.0
    prev = 0.0
    queue, drivers = [], []
    m = len(partitions)
    total_reqs    = np.zeros(m, int)
    served_by_bin = np.zeros((m, m), int)
    offers_by_bin = np.zeros(m, int)
    dest_to_part = { d1-1: p_idx
                for p_idx, part in enumerate(partitions)
                for d1 in part }
    ev = []

    heapq.heappush(ev, Event(np.random.exponential(1/lam),    'driver_arr'))
    heapq.heappush(ev, Event(np.random.exponential(1/sum(mu)), 'rider_arr'))
    warmup_horizon = 0
    while ev and now < warmup_horizon:
        e = heapq.heappop(ev)
        now = e.t

        if e.kind == 'driver_arr':
            d = Driver(join_time=now, w=w, bins=bin_bounds, partitions=partitions)
            if random.random()<min(1):
                drivers.append(d)
                queue.append(d)
            heapq.heappush(ev, Event(now + np.random.exponential(1/lam), 'driver_arr'))
        else:
            dest = np.random.choice(len(mu), p=np.array(mu)/sum(mu))
            rider = Rider(dest, P)
            heapq.heappush(ev, Event(now + np.random.exponential(1/sum(mu)), 'rider_arr'))
            for k,(lo,hi) in enumerate(bin_bounds):
                if rider.declines >= P:
                    break
                if not queue:
                    continue
                    idxs = [i for i in range(len(queue)) if lo <= i <= hi]
                if not idxs:
                    continue
                pick = np.random.choice(idxs)
                d    = queue[pick]
                if rule.accept(d, rider, k, pick):
                    d.serve(dest, now, w, c)
                    queue.pop(pick)
                    break
                else:
                    rider.declines += 1

    len_prewarm = len(drivers)
```

```
    now = prev = warmup_horizon
    stats = {'served':0, 'riders':0, 'cancelled':0, 'offers':0}
    total_earn = 0.0
    queue_time  = 0.0
    driver_arrivals = 0

    for n in range(360):
        d = Driver(now, w, bin_bounds, partitions)
        queue.append(d)

    while ev and now < T_max:
        e = heapq.heappop(ev)
        now = e.t

        if isinstance(rule, TimePNE):
            queue = [d for d in queue if now - d.join_time <= rule.T]

        dt  = now - prev
        queue_time += len(queue)*dt
        prev = now

        if e.kind == 'driver_arr' and len(queue)>400:
            heapq.heappush(ev, Event(now + np.random.exponential(1/lam), 'driver_arr'))
        elif e.kind == 'driver_arr' and len(queue)<=400:
            d = Driver(now, w, bin_bounds, partitions)
            if random.random()<1:
                drivers.append(d)
                queue.append(d)
                driver_arrivals += 1
            heapq.heappush(ev, Event(now + np.random.exponential(1/lam), 'driver_arr'))

        else:
            stats['riders'] += 1
            dest = np.random.choice(len(mu), p=np.array(mu)/sum(mu))
            rider = Rider(dest, P)
            heapq.heappush(ev, Event(now + np.random.exponential(1/sum(mu)), 'rider_arr'))

            for k,(lo,hi) in enumerate(bin_bounds):
                if not queue:
                    break
                idxs = [i for i in range(len(queue)) if lo <= i <= hi]
                if not idxs:
                    continue
                offers_by_bin[k] += 1
                stats['offers'] += 1
                pick = np.random.choice(idxs)
                d    = queue[pick]

                decision = rule.accept(d, rider, k, pick, now)
                if decision is None:
                    queue.pop(pick)
                    continue
                elif decision:
                    sum_mu_before = sum(mu[:i_star-1])
                    mu_i_star = mu[i_star-1]
                    numerator = min(lam - sum_mu_before, mu_i_star)

                    if (dest==i_star-1) and random.random()>numerator/mu_i_star:
                        mu_i_star
                    else:
                        stats['served'] += 1
                        d.serve(dest, now, w, c)
                        total_earn     += w[dest]
                        queue.pop(pick)
                    break
                else:
                    rider.declines += 1
                    if rider.declines >= P:
                        break

    T       = T_max
    served  = stats['served']

    drivers_meas    = drivers[len_prewarm:]
    payoffs = np.array([d.earnings - d.wait_cost for d in drivers_meas])

    served_drivers = [d for d in drivers_meas if d.earnings > 0]

    if served_drivers:
        avg_wait   = np.mean([d.total_wait for d in served_drivers])
        avg_payoff = payoffs.mean() if payoffs.size else np.nan
    else:
        avg_wait = avg_payoff = np.nan

    throughput = served / T
```

```
    net_rev    = (total_earn/T) - cp * (queue_time/T)

    return throughput, net_rev, avg_wait, avg_payoff

# ——— 5) All rules test ———————————————————————————
np.random.seed(0)

mu, w, c, cp = [1,6,3], [75,25,15], 1/3, 1/3
lam = 8

rules = {
    'NE':            NE(),
    'ProbNE':        ProbNE(),
    'ProbBinNE':     ProbBinNE(),
    'TimePNE(30)':   TimePNE(T=30),
    'DrivPNE':       DrivPNE(),
}

scenarios = []
parts1, i1 = generate_partitions(mu, lam, 1)
scenarios.append((1, parts1, i1))
parts2a, i2a = generate_partitions(mu, lam, 2)
parts2b = [[1],[2,3]]
i2b = i2a
scenarios.append((2, parts2a, i2a))
scenarios.append((2, parts2b, i2b))
parts3, i3 = generate_partitions(mu, lam, 3)
scenarios.append((3, parts3, i3))

rows = []
for P, parts, i_star in scenarios:
    bins = compute_bin_bounds(mu, w, c, parts)

    results = {}
    for name, rule in rules.items():
        tp, nr, aw, ap = simulate_random_fifo(mu, w, c, cp, lam, P, parts, bins, rule, 5, i_star, T_max=10000)
        results[name] = (tp, nr, aw, ap)

    tp_NE, nr_NE, _, _ = results['NE']

    for name, (tp, nr, aw, ap) in results.items():
        rows.append({
            'Rule':       name,
            'P':          P,
            'λ':          lam,
            'i*':         i_star,
            'Partitions': parts,
            'Bins':       bins,
            'TP':         round(tp,2),
            'NetRev':     round(nr,2),
            'AvgWait':    round(aw,2),
            'AvgPayoff':  round(ap,2),
            'PoATP':      round(tp/tp_NE, 2),
            'PoANR':      round(nr/nr_NE, 2),
        })

df = pd.DataFrame(rows)
display(df)

rule_order = ['NE','ProbNE','ProbBinNE','DrivPNE','TimePNE(30)']
df['Rule'] = pd.Categorical(df['Rule'], categories=rule_order, ordered=True)
df_by_rule = df.sort_values(['Rule','P']).reset_index(drop=True)

display(df_by_rule)
```