Abstract

Lo studio si concentra sulla modellazione e sulla simulazione del sistema di **ride-hailing** di Bolt, facendo ricorso ai principi fondamentali della teoria delle code. L'obiettivo principale è analizzare l'efficienza operativa dell'architettura del sistema attualmente adottata e valutare l'impatto dell'introduzione di un servizio di **ride-sharing**. Il lavoro si articola in due fasi principali:

1. **Analisi del sistema esistente di ride-hailing**, con particolare attenzione alla distribuzione delle richieste di servizio e all'impiego dei veicoli.

2. **Introduzione e valutazione di un servizio di ride-sharing**, concepito ridurre l'impatto ambientale del sistema, in linea con le normative emergenti in materia di mobilità sostenibile.

Quest'ultima fase, infatti, si inserisce nel contesto normativo definito dal **Regolamento (UE) 2019/631 del Parlamento Europeo e del Consiglio**[1][2], che stabilisce i livelli di prestazione in materia di emissioni di CO_2 per le nuove autovetture e i veicoli commerciali leggeri. Tale regolamento mira a incentivare soluzioni di trasporto più sostenibili, promuovendo l'adozione di soluzioni condivise, come il ride-sharing, capace di ridurre il numero di veicoli circolanti e, di conseguenza, le emissioni per passeggero.

Indice

1 Introduzione

Bolt è un'azienda estone attiva nel settore della mobilità urbana che, tra i servizi offerti, include il **ride-hailing** tramite una piattaforma digitale. La missione dichiarata dall'azienda è contribuire alla riduzione delle emissioni derivanti dal trasporto passeggeri, *responsabile di una quota significativa delle emissioni globali*, incentivando soluzioni di mobilità condivisa. Grazie all'integrazione di più modalità di trasporto in un'unica piattaforma — automobili, biciclette, monopattini elettrici, ecc. — e al supporto alla connessione con il trasporto pubblico, Bolt si propone di costruire un modello di mobilità urbana **più efficiente, accessibile e sostenibile** rispetto a quello tradizionale basato sull'uso dell'auto privata.

2 Oggetto di studio

Prima di descrivere nel dettaglio l'oggetto di studio, è opportuno chiarire alcune ipotesi semplificative adottate per la sua costruzione: il sistema studiato rappresenta una versione astratta e controllata del servizio di ride-hailing offerto da Bolt. In particolare, si assume:

1. Un **intervallo temporale fisso**, durante il quale si concentra l'analisi delle prestazioni.

2. Il **numero di veicoli**, sia **complessivamente** sia per **ciascuna tipologia**, rimane invariato per l'intero intervallo temporale considerato.

3. L'**assenza di una modellazione spaziale esplicita**: le distanze tra la posizione dei passeggeri e dei veicoli sono approssimate attraverso probabilità di abbinamento, senza considerare coordinate geografiche reali.

4. Un **domanda delle richieste di corsa stabile** durante tutto l'intervallo simulato.

Il sistema oggetto di studio è composto da **tre centri di servizio**, ciascuno dedicato alla gestione delle richieste per una specifica tipologia di veicolo tra quelle offerte da Bolt. Le tipologie selezionate sono:

1. **small** che rappresenta i veicoli con una capacità da 1 a 3 posti.

2. **medium** che rappresenta i veicoli con una capacità da 4 posti.

3. **large** che rappresenta i veicoli con una capacità da 5 a 8 posti.

Queste tre tipologie rappresentano le classi di veicoli più significative per il funzionamento del sistema, pur non esaurendo l'intera gamma di opzioni presenti sulla piattaforma. Un utente che desidera richiedere una corsa accede alla piattaforma digitale, seleziona la tipologia di veicolo desiderata e invia la richiesta di corsa. Nella realtà, l'assegnazione di una richiesta ad un veicolo avviene tramite un algoritmo multi-parametrico di *matching*, che considera vari fattori come distanza, priorità, disponibilità del conducente etc. Nel nostro studio tale processo è **semplificato** adottando una politica di tipo **selection in order**, in cui ogni richiesta viene assegnata al primo veicolo disponibile. Il veicolo assegnato rimane occupato per tutta la durata della corsa, al termine della quale torna automaticamente disponibile e può accettare nuove richieste. Inoltre, un utente potrebbe decidere di cancellare la propria richiesta di corsa prima che questa sia assegnata ad un veicolo.

3 Obiettivo dell'analisi

Lo studio si propone due obiettivi principali. Il primo è **gestire lo stesso carico di lavoro** con l'introduzione del servizio di ride-sharing, individuando il **numero ottimale di veicoli** da assegnare a ciascuna tipologia in modo da ottenere una distribuzione del carico tra i centri di servizio quanto più omogenea possibile. Il secondo obiettivo è la **riduzione del numero totale di veicoli** impiegati, resa possibile grazie all'introduzione del servizio di ride-sharing, senza compromettere le prestazioni osservate nel modello base.

4 Modello concettuale

Il servizio di Bolt è stato modellato come in Figura 1.

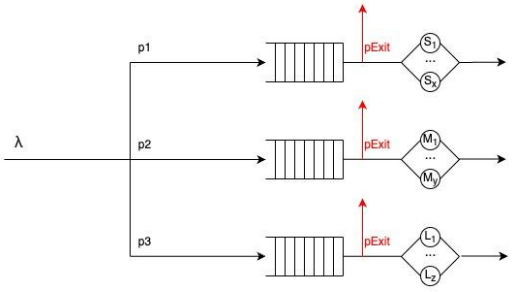


Figura 1: Modello concettuale del servizio di ride-hailing

Gli elementi principali di questo sistema sono:

1. **Utente**: colui che desidera richiedere una corsa.
2. **Auto**: il veicolo che può essere richiesto da un utente.

In particolare si ha:

1. Centro di **servizio tradizionale** per veicoli **small**: gestisce le corse effettuate da veicoli con capacità da **1 a 3 passeggeri**.
2. Centro di **servizio tradizionale** per veicoli **medium**: gestisce le corse effettuate da veicoli con capacità da **4 passeggeri**.
3. Centro di **servizio tradizionale** per veicoli **large**: gestisce le corse effettuate da veicoli con capacità da **5 a 8 passeggeri**.

4.1 Stato del sistema

Nel sistema **tradizionale**, lo stato del sistema in un determinato istante è definito dal vettore

$$(n_{small}, n_{medium}, n_{large})$$

- $n_{small}, n_{medium}, n_{large}$: numero di richieste attualmente presenti nei centri di servizio, suddivise in base alla tipologia di veicolo richiesta (small, medium, large).

4.2 Descrizione degli eventi

Nel sistema **tradizionale**, gli eventi che modificano lo stato del sistema sono:

1. L'**arrivo di una richiesta** di corsa per una specifica **tipologia** di veicolo.
2. Il **termine di una corsa** che rende un veicolo nuovamente **disponibile**.
3. L'**uscita dal sistema** di un utente che ha annullato la propria **richiesta di servizio**.

5 Modello delle specifiche

Gli arrivi al sistema sono modellati con un processo di **Poisson**, in cui i tempi di inter-arrivo, distribuiti *esponenzialmente*, risultano indipendenti e privi di memoria, descrivendo in modo realistico la casualità delle richieste di servizio.

Il **tempo medio di servizio** $E(S)$ e il **numero totale di veicoli** N sono stati ricavati dai dati presentati nello studio[3]. Il numero totale di veicoli N è stato poi suddiviso in tre partizioni N_1, N_2, N_3, proporzionalmente alla domanda di ciascuna tipologia di veicolo.

Poiché lo studio si concentra su una **fascia operativa a carico medio-alto**, il coefficiente di utilizzo ρ è stato fissato a **0.7**. A partire dai valori di $E(S)$, dal numero di veicoli disponibili per tipologia e dal valore di ρ, è possibile calcolare il **tasso di arrivo medio** λ tramite le seguenti formule:

- $\lambda_1 = \frac{\rho}{\frac{E(S_{server})}{N_1}}$
- $\lambda_1 = (1 - p_{Exit}) * p_1 * \lambda$

$\implies \lambda = \frac{\rho N_1}{E(S_{server})(1-p_{Exit})*p_1}$

5.1 Matrice di routing

Il modello impiega tre probabilità per ripartire le richieste tra le diverse tipologie di veicoli.

Inoltre, viene introdotta una probabilità di abbandono dalla coda, denotata con $\mathbf{P_{Exit}}$, che rappresenta la possibilità che un utente in attesa decida di annullare la propria richiesta di corsa. A tale probabilità è stato assegnato il valore **0.05**, ipotizzando che, qualora il sistema riesca a gestire ragionevolmente il carico, sia poco probabile che un utente scelga di cancellare la propria richiesta.

La tabella di routing è:

	Esterno	Centro small	Centro medium	Centro large
Esterno	0	p_1	p_2	p_3
Centro small	1	0	0	0
Centro medium	1	0	0	0
Centro large	1	0	0	0

Tabella 1: Tabella di routing modello base

5.2 Equazioni di traffico

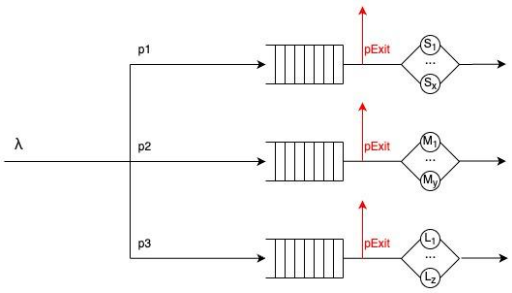


Figura 2: Modello concettuale del servizio di ride-hailing

Le equazioni di traffico del sistema sono:

$$\begin{cases} \lambda_1 = p_1(1 - p_{Exit})\lambda \\ \lambda_2 = p_2(1 - p_{Exit})\lambda \\ \lambda_3 = p_3(1 - p_{Exit})\lambda \end{cases}$$

5.3 Modellazione centri

I centri di servizio "classici" sono modellati come code **M/N/k**, caratterizzati da arrivi di tipo **Poisson**, un numero **finito** di server e da una **coda di attesa a capacità infinita**, alimentata da un pool di utenti. Nel sistema sono presenti tre centri di questo tipo, ciascuno dedicato a una specifica tipologia di veicolo: **small**, **medium** e **large**. Il numero di server in ciascun centro è determinato in funzione della categoria di veicolo che esso gestisce.

5.4 Scelta delle distribuzioni

In assenza di dati sufficienti per un'analisi statistica precisa, le scelte adottate per modellare i tempi di servizio nei diversi centri mirano a rappresentare in modo realistico il comportamento dei processi.

- **Distribuzione Gaussiana Troncata – Tempi di Servizio**

  Per modellare i tempi di servizio è stata adottata una distribuzione **Gaussiana troncata**, considerata adatta a descrivere un processo con forte concentrazione dei valori attorno alla media. Questa distribuzione permette, infatti, di catturare l'andamento centrato dei tempi di servizio, pur lasciando spazio a variazioni legate a corse più brevi o

più lunghe. La troncatura inferiore evita la generazione di tempi irrealisticamente bassi, mentre quella superiore pone un limite massimo coerente con le tempistiche reali del sistema.

- **Distribuzione Esponenziale – Tempi di Arrivo**

  Per modellare gli istanti di arrivo delle richieste si è scelta la distribuzione **esponenziale**, ampiamente impiegata in ambito modellistico per rappresentare processi di arrivo casuali. La scelta è giustificata dalla proprietà di assenza di memoria tipica di questa distribuzione: la probabilità di un prossimo arrivo non dipende dal tempo trascorso dall'ultimo evento. Questo riflette adeguatamente l'elevata aleatorietà del flusso di richieste e l'indipendenza tra arrivi successivi.

6 Modello computazionale

Come linguaggio di programmazione per la simulazione è stato scelto **Java**, principalmente per i vantaggi che offre, tra cui la gestione automatica della memoria e l'approccio *object-oriented*, particolarmente adatto alla modellazione di sistemi complessi. Per la generazione dei grafici è stato, invece, utilizzato **Python**, grazie alla disponibilità della libreria `matplotlib`, che risulta semplice ed efficace per la creazione di rappresentazioni grafiche.

Per garantire una struttura chiara, modulare e facilmente manutenibile, il codice è stato organizzato in **sei packages principali**:

- **Centers**: contiene le classi che implementano le due tipologie di centri (*base* e *migliorativo*), incapsulando la logica di creazione e di processamento degli eventi del sistema.
- **Configuration**: include la classe di supporto per la gestione e l'impostazione dei parametri di configurazione.
- **Controller**: raccoglie le classi che implementano i due sistemi (*base* e *migliorativo*) e che integrano al loro interno la logica simulativa sia per il caso ad **orizzonte finito** che ad **orizzonte infinito**.
- **Libs**: contiene il codice sviluppato da *Steve Park* e *Dave Geyer* che fornisce l'implementazione delle distribuzioni statistiche e dei generatori pseudo-casuali utilizzati nella simulazione.
- **Model**: comprende le classi di supporto alla simulazione *next-event* e alla raccolta delle statistiche.
- **Utils**: raccoglie le classi ausiliarie per l'elaborazione e l'analisi delle statistiche prodotte durante le diverse simulazioni.

Per la generazione dei numeri pseudo-casuali, il modello utilizza la libreria `rngs`, una versione multi-stream compatibile con la libreria `rng`. Questa libreria gestisce fino a 256 stream di numeri casuali, dei quali solo uno è attivo alla volta. Durante la simulazione, ogni volta che è necessario generare un numero casuale per una componente del modello, lo stream corrispondente viene attivato tramite `SelectStream` e il numero viene prodotto tramite `Random`. Lo stato aggiornato di ciascun stream viene salvato tramite `GetSeed` solo al termine di ogni run, garantendo l'indipendenza e la variabilità delle sequenze casuali tra repliche e assicurando risultati affidabili per le analisi statistiche.

Per inizializzare tutti gli stream è stata utilizzata la funzione `PlantSeeds`, che permette di "piantare" uno stato iniziale differente per ciascuno stream. Nel modello computazionale implementato, ciascun stream è associato a una specifica componente casuale del sistema. Lo **stream 0** è il **default stream**, utilizzato come punto di partenza per inizializzare tutti gli altri stream tramite `PlantSeeds`. Gli altri stream sono così assegnati:

- Lo **stream 1** è dedicato alla funzione `getNextArrivalTimeSimpleCenter()`;
- Lo **stream 2** è dedicato alla funzione `getNextArrivalTimeRideSharing()`;
- Lo **stream 3** è dedicato alla funzione `getServiceTimeSimple()`;
- Lo **stream 4** è dedicato alla funzione `getServiceTimeRideSharing()`;
- Lo **stream 5** gestisce la probabilità di uscita dal servizio tradizionale (`P_EXIT`);
- Lo **stream 6** gestisce la probabilità di uscita dal servizio ride-sharing (`P_EXIT`);
- Lo **stream 7** gestisce la probabilità di match con risorsa occupata (`P_MATCH_BUSY`);
- Lo **stream 8** è dedicato alla funzione `getNumeroPosti()`.

La simulazione adotta l'approccio della **Next-Event Simulation**, che consiste nel far avanzare il *clock* della simulazione elaborando gli eventi in ordine cronologico. In pratica, si identifica l'evento più imminente, si esegue l'azione corrispondente e si aggiornano di conseguenza lo stato del sistema e il *clock* della simulazione. Nei paragrafi seguenti verranno analizzate alcune delle componenti più significative di questo meccanismo.

6.1 Simulation Clock

Una delle principali strutture dati del simulatore è `MsqTime`, che rappresenta il *clock* della simulazione:

```
public class MsqTime {
    double current;
    double next;
}
```

Gli attributi della classe hanno il seguente significato:

- **current**: istante temporale corrente della simulazione;
- **next**: istante temporale in cui si verificherà il prossimo evento.

6.2 Eventi

Gli eventi del sistema sono modellati tramite la classe `MsqEvent`, definita come segue:

```
public class MsqEvent {
    double t;
    int x;
}
```

Gli attributi della classe sono:

- **t**: istante temporale in cui l'evento si verifica;
- **x**: variabile di stato che specifica se l'evento è attivo oppure no.

6.3 SimpleSystem

La logica di simulazione è implementata all'interno della classe `SimpleSystem`, che mette a disposizione due modalità di esecuzione:

- `runFiniteSimulation()`, che realizza una simulazione a orizzonte finito mediante la *tecnica delle repliche*. Tale modalità viene impiegata esclusivamente per l'analisi del transitorio, con l'obiettivo di verificare se e quando il sistema converge verso un regime stazionario;
- `runInfiniteSimulation()`, che implementa una simulazione a orizzonte infinito basata sulla *tecnica delle batch means*. Questa modalità costituisce l'approccio effettivamente adottato per il calcolo delle **statistiche prestazionali**, consentendo di ottenere stime affidabili in regime stazionario.

6.4 SimpleMultiServerNode

La classe `SimpleMultiServerNode` incapsula la logica operativa dei centri di servizio. Il metodo `processNextEvent()` si occupa della gestione dell'evento temporalmente più vicino nella coda degli eventi del nodo: esso aggiorna lo stato interno, esegue le azioni associate all'evento e provvede ad avanzare il tempo della simulazione.

```
{
  public int processNextEvent(double t) {
        int e = peekNextEventType();
        MsqEvent ev = event.get(e);
        clock.next = ev.t;

        clock.current = clock.next;

        if (e == ARRIVAL || e > numberOfServersInTheCenter) {
            if (e == ARRIVAL) {

                lastArrivalTimeInBatch = clock.current;

                numberJobInSystem++;

                MsqEvent arr = event.getFirst();
                arr.t = distrs.getNextArrivalTimeSimpleCenter(rng,
                    ↪ system, centerIndex, clock.current);
                arr.x = 1;

                rng.selectStream(5);
                double rnd = rng.random();


                if (rnd < P_EXIT) {
                    numberJobInSystem--;
                    return -1;
                }
            }

            int serverIndex = findOne();

            if (serverIndex != -1) {

                double serviceTimeSimple = distrs.getServiceTimeSimple
                    ↪ (rng);

                MsqEvent sEvent = event.get(serverIndex);
                sEvent.t = clock.current + serviceTimeSimple;

                serversCompletition[serverIndex].setLastCompletionTime
                    ↪ (sEvent.t);
```

```
                sEvent.x = 1;
                sum[serverIndex].service += serviceTimeSimple;

                if (e > numberOfServersInTheCenter) {
                    event.remove(e);
                }
                return serverIndex;
            }
        } else {
            numberJobInSystem--;
            sum[e].served++;

            if (numberJobInSystem >= numberOfServersInTheCenter) {
                double serviceTimeSimple = distrs.getServiceTimeSimple
                    ↪ (rng);

                MsqEvent sEvent = event.get(e);
                sEvent.t = clock.current + serviceTimeSimple;

                serversCompletition[e].setLastCompletionTime(sEvent.t)
                    ↪ ;
                sum[e].service += serviceTimeSimple;
                return e;


            } else {
                event.get(e).x = 0;
                }
            }
        return -1;
    }
}
```

Nel caso di elaborazione di un arrivo, il sistema verifica la possibilità di assegnare la richiesta a uno dei server disponibili. A tal fine viene adottata una politica di *selection in order*, implementata dal metodo `findOne()`: la richiesta viene quindi instradata al primo server libero individuato nell'ordine di scansione.

```
{
  private int findOne() {
        for (int i = 1; i < event.size(); i++) {
            if (event.get(i).x == 0) return i;
```

```
        }
        return -1;
    }
}
```

6.5 getNextArrivalTimeSimpleCenter() e getServiceTimeSimple()

Per generare i tempi di arrivo e di servizio, il modello sfrutta la classe `Rngs` con un approccio multi-stream, garantendo l'indipendenza tra le sequenze di numeri casuali utilizzate dalle diverse componenti del sistema.

Si mostra come esempio il codice di `getNextArrivalTimeSimpleCenter()`:

```
{
  public double getNextArrivalTimeSimpleCenter(Rngs r, Sistema system
      ↪ , int centerIndex, double sarrival) {
        r.selectStream(1);
        double lambda = config.getDouble("simulation","lambdasimple")
            ↪ ;

        if(system instanceof SimpleSystem) {
            switch (centerIndex) {
                case 0 -> lambda *= config.getDouble("simulation","
                    ↪ p_small");
                case 1 -> lambda *= config.getDouble("simulation","
                    ↪ p_medium");
                case 2 -> lambda *= config.getDouble("simulation","
                    ↪ p_large");
                default -> System.out.println("Centro inesistente!");
            }

        }else{

            switch (centerIndex) {
                case 0 -> lambda *= config.getDouble("simulation","
                    ↪ p_small") * config.getDouble("simulation","
                    ↪ psimple") ;
                case 1 -> lambda *= config.getDouble("simulation","
                    ↪ p_medium") * config.getDouble("simulation","
                    ↪ psimple") ;
```

```
                case 2 -> lambda *= config.getDouble("simulation","
                    ↪ p_large") * config.getDouble("simulation","
                    ↪ psimple") ;
                default -> System.out.println("Centro inesistente!");

            }
        }

        sarrival += exponential(1/lambda, r);
        return sarrival;
    }
}
```

Mentre `getServiceTimeSimple()` ha la seguente implementazione:

```
{
  public double getServiceTimeSimple(Rngs r) {
        r.selectStream(3);
        double esi = config.getDouble("simulation","esi");
        double alpha, beta;
        double a = 2;
        double b = 40;

        alpha = cdfNormal(esi, 4, a);
        beta = cdfNormal(esi, 4, b);

        double u = uniform(alpha, beta, r);
        return idfNormal(esi, 4, u);
    }
}
```

7 Design degli esperimenti

7.1 Intervalli di confidenza

All'interno della trattazione sono stati utilizzati *gli intervalli di confidenza*. Un intervallo di confidenza è un intervallo calcolato a partire dai dati campionari, che ha una certa probabilità di contenere il valore reale della media della popolazione. In altre parole, fornisce una stima dell'incertezza associata alla media campionaria, indicando un intervallo plausibile entro cui ci si può ragionevolmente aspettare che la media reale del sistema si trovi.

Per questo studio sono stati scelti intervalli con un *livello di confidenza* pari al **95%**. Gli intervalli di confidenza sono stati calcolati utilizzando la seguente formula:

$$Intervallo\ di\ Confidenza = t^* \left(\frac{s}{\sqrt{n-1}} \right)$$

dove

- **s** è la deviazione standard campionaria;
- **n** è la dimensione del campione;
- $\mathbf{t}^*$ è il valore critico della distribuzione t di *Student* (nel codice, il calcolo è stato effettuato utilizzando la funzione `idfStudent()` fornita dalla classe `Rvms`).

7.2 Analisi del transitorio

Prima di procedere con l'esecuzione degli esperimenti, è fondamentale analizzare la fase transitoria del sistema per verificare ***se*** e ***quando*** esso raggiunge uno stato stazionario. A tal fine, si esegue una simulazione a orizzonte finito (*finite-horizon*), osservando il sistema per un tempo limitato. Per ottenere stime affidabili delle statistiche transitorie, la simulazione viene replicata più volte a partire dallo stesso stato iniziale. In ciascuna replica, i generatori di numeri pseudo-casuali sono inizializzati utilizzando lo **stato finale** degli stream `rng` della replica precedente. Questo garantisce che le sequenze casuali siano indipendenti e non si sovrappongano tra le repliche, permettendo di catturare correttamente la variabilità naturale del sistema e di valutare con precisione il comportamento transitorio e la convergenza verso lo stato stazionario.

La variabile utilizzata come indicatore della convergenza transitoria è il tempo medio di risposta $E[Ts]$, come illustrato nelle figure sottostanti:

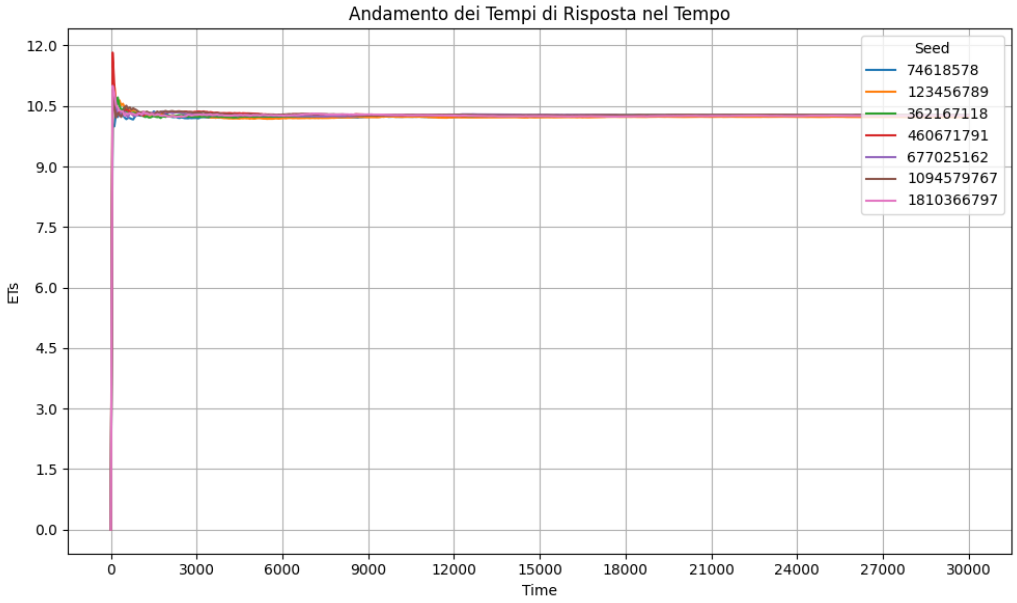


Figura 3: Centro small

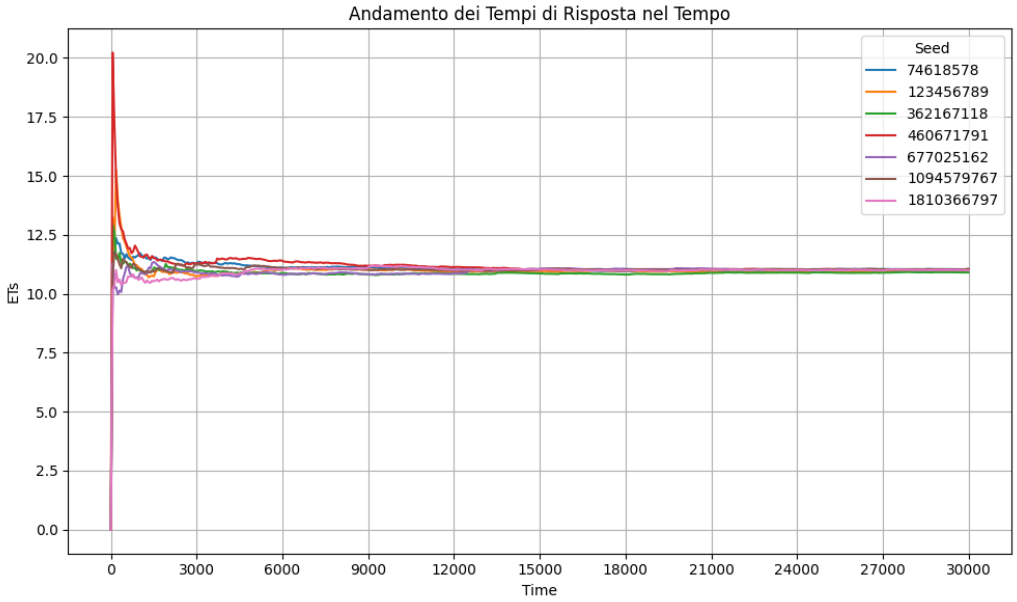


Figura 4: Centro medium

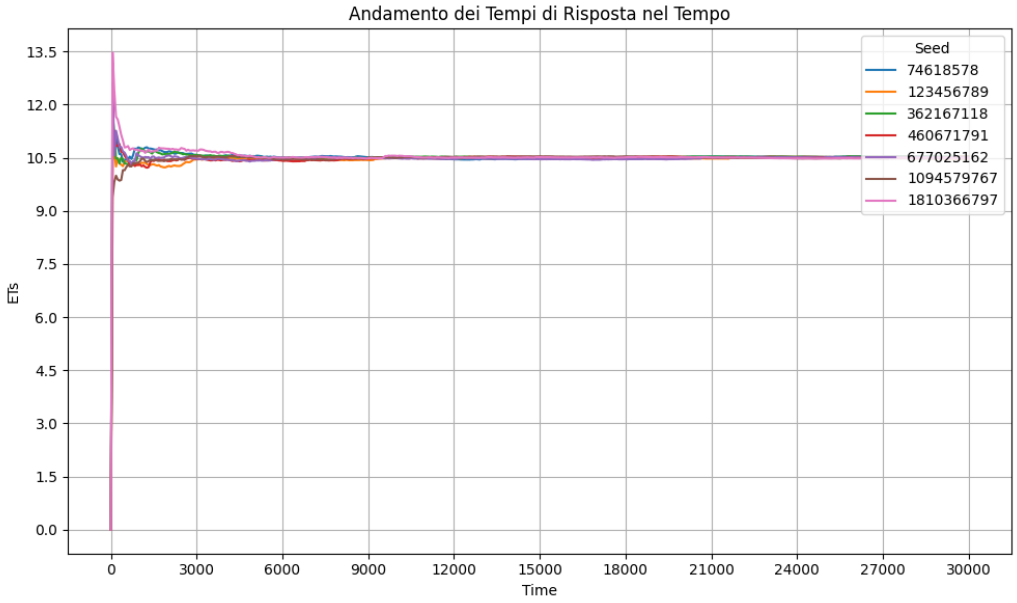


Figura 5: Centro large

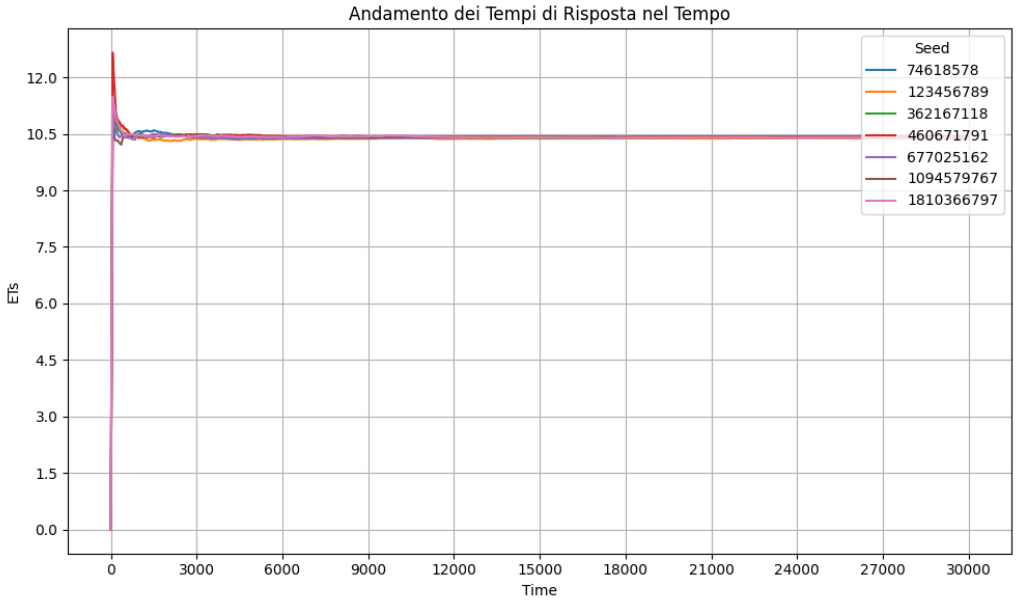


Figura 6: Risultati globali

7.3 Simulazione ad orizzonte infinito

Lo scopo di una simulazione a orizzonte infinito è stimare le caratteristiche del sistema in regime stazionario. Questo approccio è stato adottato per analizzare il comportamento asintotico del sistema, cioè il periodo in cui le metriche di interesse diventano indipendenti dalle condizioni iniziali. Considerando che Bolt è un'applicazione già ampiamente consolidata e adottata dagli utenti e che lo studio si concentra su una singola fascia operativa a partire da uno stato iniziale non vuoto, la simulazione a orizzonte infinito risulta particolarmente appropriata poiché consente di ottenere stime delle prestazioni operative prive del bias introdotto dalla fase transitoria.

Per stimare le statistiche di interesse, si è applicato il metodo delle ***Batch Means***. I parametri scelti sono stati $\mathbf{b = 4096}$ (dimensione di ciascun batch) e $\mathbf{k = 512}$ (numero complessivo di batch). Tali valori non sono stati fissati a priori: inizialmente si era optato per $b = 256$ e $k = 64$, ma entrambi i parametri sono stati aumentati progressivamente fino a garantire che la *lag-1 autocorrelazione* tra le medie di batch risultasse inferiore a **0.2**, in accordo con quanto raccomandato in [4]. In questo modo si è raggiunto un compromesso soddisfacente perché da un lato le medie di batch risultano sufficientemente indipendenti; dall'altro, il numero di batch rimane abbastanza ampio da permettere una stima affidabile della varianza.

È noto che, all'inizio della simulazione, le caratteristiche del sistema differiscono da quelle in regime stazionario, ad esempio quando il sistema viene avviato vuoto. È quindi necessario eseguire la simulazione per un certo periodo di tempo prima che il sistema raggiunga lo stato stazionario. Le osservazioni raccolte durante questo periodo iniziale potrebbero compromettere la precisione delle stime degli indici di prestazione, specialmente se il periodo di transizione è particolarmente lungo. Questo problema è noto come *initialization bias* o *start-up problem* [5]. Per mitigare tale problema, si è deciso di scartare le osservazioni corrispondenti ai primi $r \cdot b \cdot k$ job processati. Dopo alcune prove, il parametro $\mathbf{r}$ è stato fissato a **0.2** sia per il modello iniziale sia per quello migliorato. Non esiste una regola universale per determinare il valore ottimale di r; risulta, tuttavia, importante evitare valori troppo elevati, poiché questi ridurrebbero eccessivamente la quantità di dati disponibili e potrebbero aumentare l'autocorrelazione tra le osservazioni.

Eseguendo la simulazione si osservano i seguenti risultati:

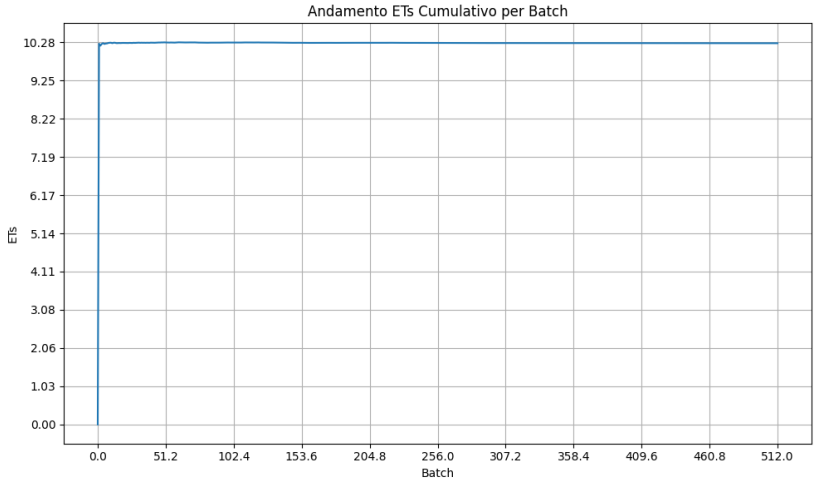


Figura 7: Centro small

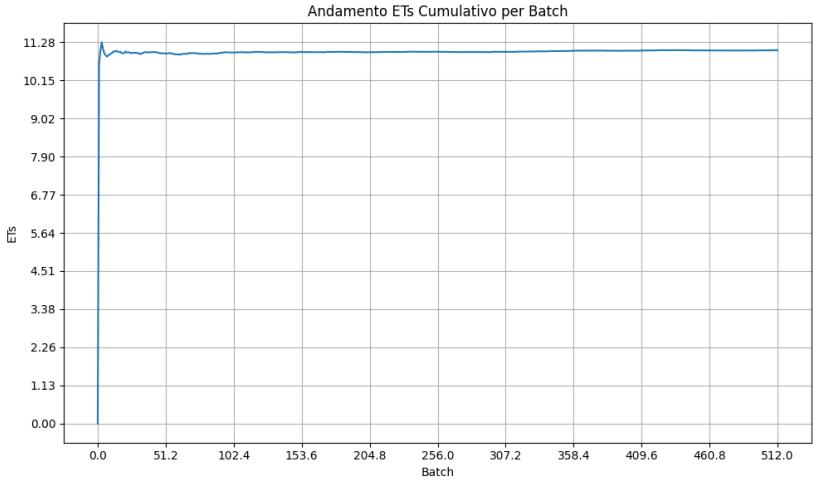


Figura 8: Centro medium

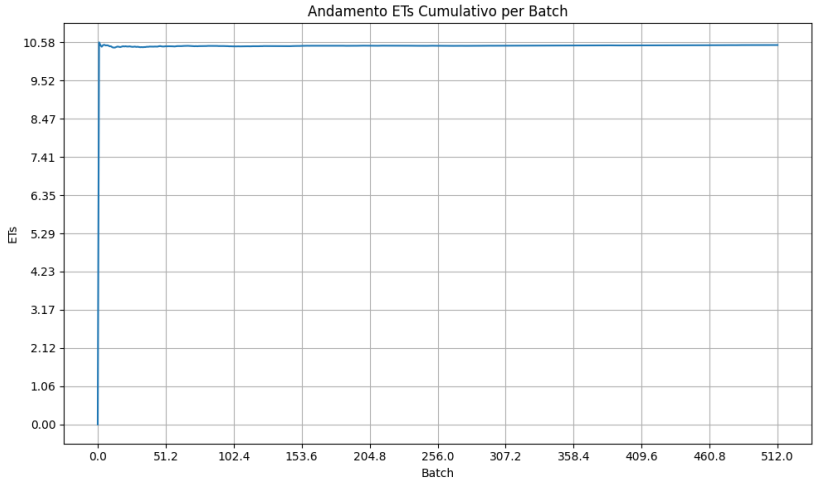


Figura 9: Centro large

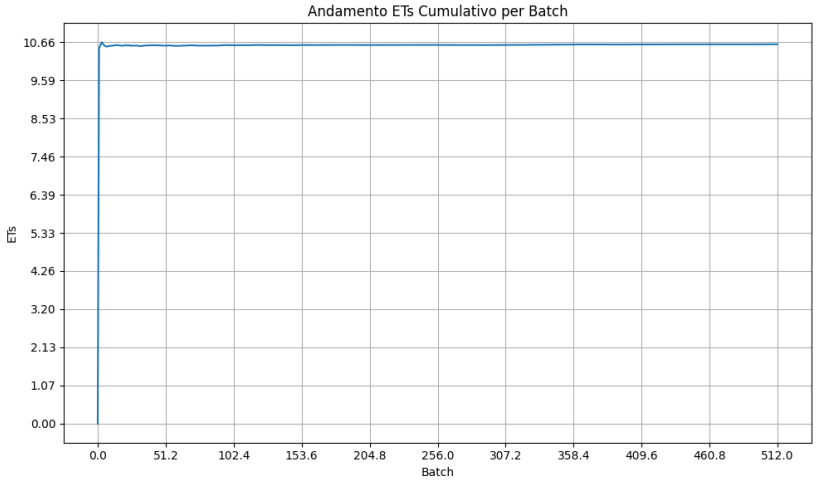


Figura 10: Risultati globali

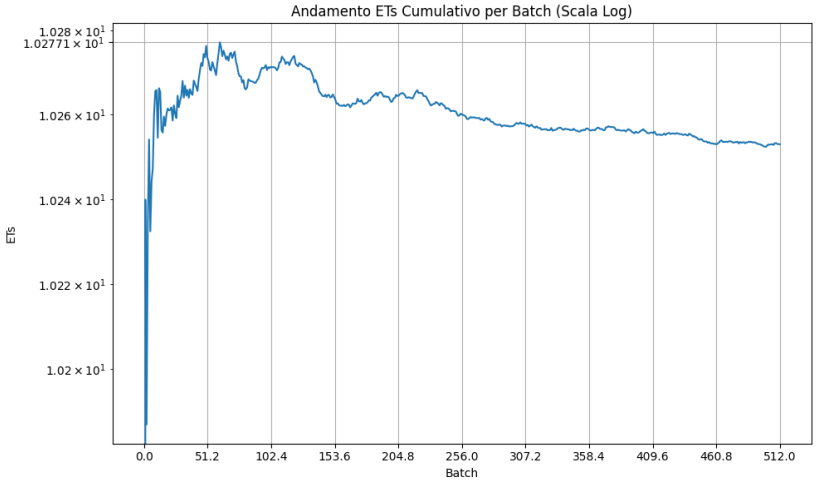


Figura 11: Centro small in scala logaritmica

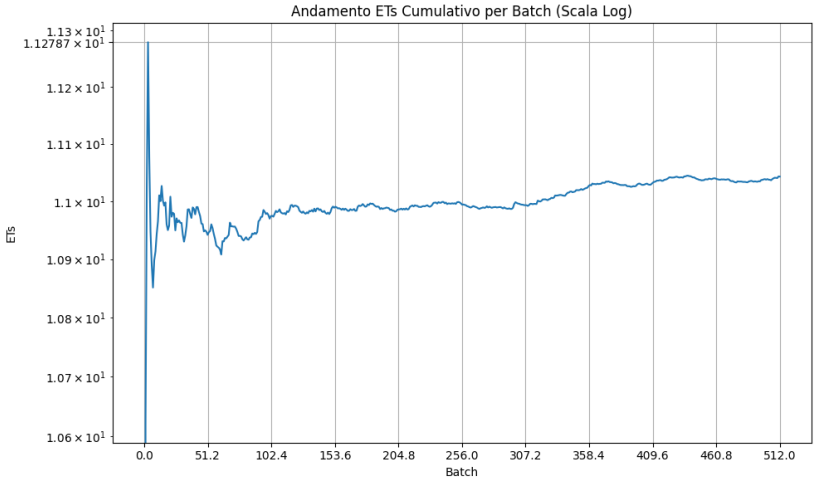


Figura 12: Centro medium in scala logaritmica

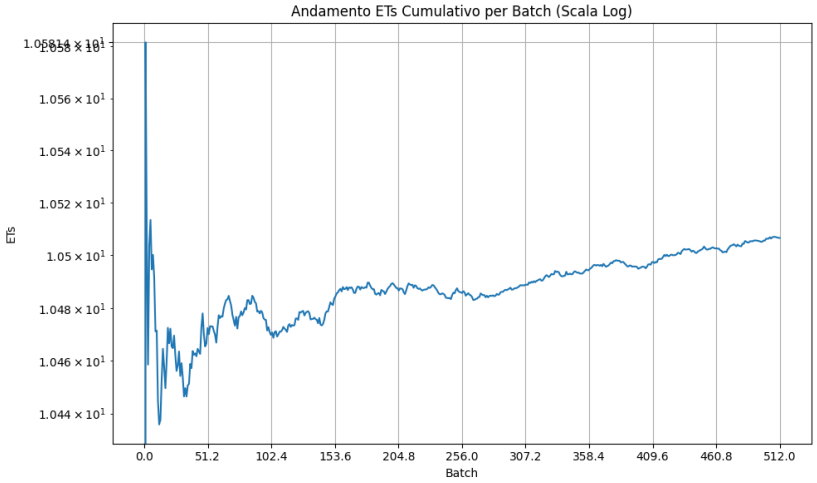


Figura 13: Centro large in scala logaritmica

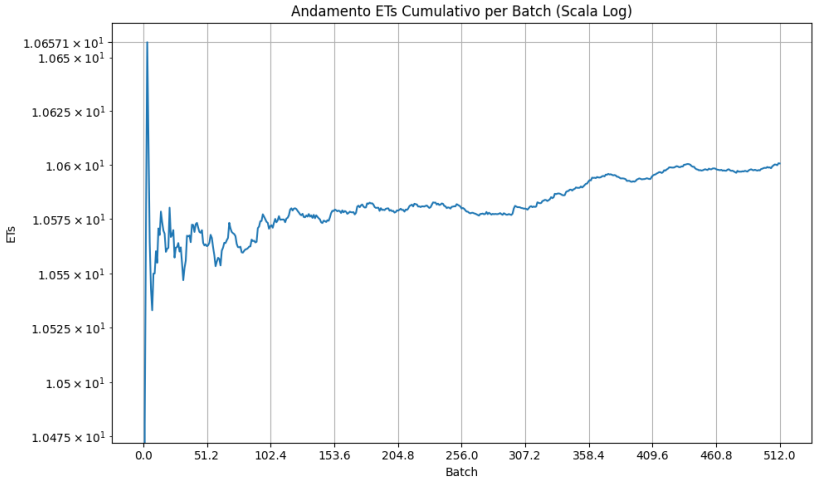


Figura 14: Risultati globali in scala logaritmica

Centro	$E[N_S]$	$E[T_S]$	$E[N_Q]$	$E[T_Q]$	ρ	E[S]
small	23.6684 ± 0.0433	10.2530 ± 0.0069	0.0795 ± 0.0047	0.0342 ± 0.0020	0.7148 ± 0.0012	10.2187 ± 0.0063
medium	4.2580 ± 0.0268	11.0426 ± 0.0371	0.3201 ± 0.0130	0.8211 ± 0.0310	0.6563 ± 0.0027	10.2216 ± 0.0143
large	12.1577 ± 0.0376	10.5066 ± 0.0144	0.3383 ± 0.0124	0.2907 ± 0.0103	0.7387 ± 0.0018	10.2160 ± 0.0083

Tabella 5: Statistiche riassuntive simulazione ad orizzonte infinito del modello base

7.4 Autocorrelazione

Di seguito sono riportati i valori dell'autocorrelazione delle statistiche forniti dal simulatore, che utilizza la classe di libreria `Acs.java`:

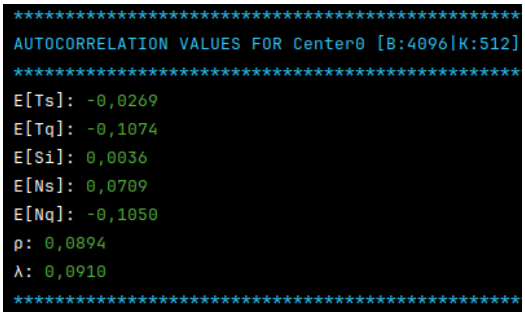


Figura 15: Valori di autocorrelazione del centro small

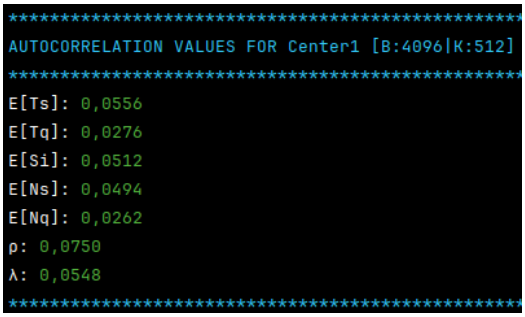


Figura 16: Valori di autocorrelazione del centro medium

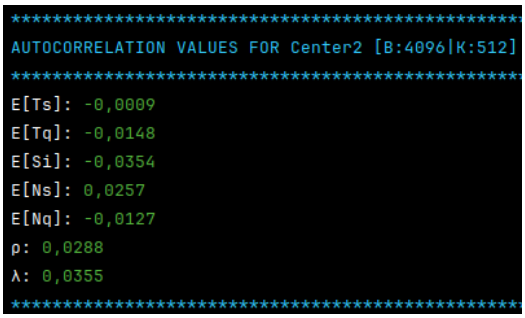


Figura 17: Valori di autocorrelazione del centro large

8 Verifica

La verifica mira a dimostrare la correttezza dell'implementazione confrontando i risultati di simulazione con soluzioni analitiche entro un intervallo di confidenza del 95% (vedi §9.1). Per rendere possibile il confronto, è stata cambiata la distribuzione dei tempi di servizio in una **esponenziale**. La verifica si concentra sul caso stazionario, utilizzando una simulazione ad **orizzonte infinito**. Per ogni centro vengono riportati i calcoli analitici e allegato lo screenshot della verifica automatica eseguita dal simulatore.

8.1 Centro automobili small M/M/33

$$\lambda = 4.05263161\,\text{richieste/min}$$

$$\lambda_1 = \lambda \cdot p_{small} \cdot (1 - p_{exit}) = 2.31000002\,\text{richieste/min}$$

$$E(S_i) = \frac{1}{\mu_i} = 10\,\text{min}$$

$$E(S) = \frac{E(S_i)}{33} = 0.\overline{30}\,\text{min}$$

$$\mu_i = \frac{1}{E(S_i)} = 0.1\,\text{job/min}$$

$$\rho = \frac{\lambda_1}{33\,\mu_i} = 0.70000001$$

$$m = 33$$

$$p_0 = \left[\sum_{i=0}^{m-1} \frac{(m\,\rho)^i}{i!} + \frac{(m\,\rho)^m}{m!\,(1-\rho)}\right]^{-1} = 9.23785633 \times 10^{-11}$$

$$P_Q = \frac{(m\,\rho)^m\ p_0}{m!\,(1-\rho)} = 0.03539644$$

$$E(T_q) = \frac{P_Q\ \cdot E(S)}{1-\rho} = 0.03575398\,\text{min}$$

$$E(N_q) = E(T_q)\,\lambda_1 = 0.0825917$$

$$E(T_s) = E(T_q) + E(S_i) = 10.03575398\,\text{min}$$

$$E(N_s) = E(T_s)\,\lambda_1 = 23.18259187$$

```
Center0
E[Ts]: mean 10,0267, diff 0,0091 is within ±0,0161
E[Tq]: mean 0,0330, diff 0,0028 is within ±0,0029
E[Si]: mean 9,9937, diff 0,0063 is within ±0,0152
E[Ns]: mean 23,1457, diff 0,0369 is within ±0,0539
E[Nq]: mean 0,0765, diff 0,0061 is within ±0,0067
ρ: mean 0,6991, diff 0,0009 is within ±0,0015
λ: mean 2,3084, diff 0,0016 is within ±0,0036
```

Figura 18: Valori di verifica automatica per il centro small

8.2 Centro automobili medium M/M/6

$$\lambda = 4.05263161 \,\text{richieste/min}$$

$$\lambda_2 = \lambda \cdot p_{medium} \cdot (1 - p_{exit}) = 0.385 \,\text{richieste/min}$$

$$E(S_i) = \frac{1}{\mu_i} = 10 \,\text{min}$$

$$E(S) = \frac{E(S_i)}{6} = 1.\overline{6} \,\text{min}$$

$$\mu_i = \frac{1}{E(S_i)} = 0.1 \,\text{job/min}$$

$$\rho = \frac{\lambda_2}{6\,\mu_i} = 0.641\overline{6}$$

$$m = 6$$

$$p_0 = \left[\sum_{i=0}^{m-1} \frac{(m\,\rho)^i}{i!} + \frac{(m\,\rho)^m}{m!\,(1-\rho)}\right]^{-1} = 0.01976355$$

$$P_Q = \frac{(m\,\rho)^m\, p_0}{m!\,(1-\rho)} = 0.24946504$$

$$E(T_q) = \frac{P_Q \cdot E(S)}{1-\rho} = 1.16030201 \,\text{min}$$

$$E(N_q) = E(T_q)\,\lambda_2 = 0.44671627$$

$$E(T_s) = E(T_q) + E(S_i) = 11.16030201 \,\text{min}$$

$$E(N_s) = E(T_s)\,\lambda_2 = 4.29671627$$

```
Center1
E[Ts]: mean 11,1257, diff 0,0346 is within ±0,0888
E[Tq]: mean 1,1448, diff 0,0155 is within ±0,0616
E[Si]: mean 9,9809, diff 0,0191 is within ±0,0399
E[Ns]: mean 4,2925, diff 0,0042 is within ±0,0435
E[Nq]: mean 0,4468, diff 0,0001 is within ±0,0253
ρ: mean 0,6409, diff 0,0007 is within ±0,0036
λ: mean 0,3853, diff 0,0003 is within ±0,0015
```

Figura 19: Valori di verifica automatica per il centro medium

8.3 Centro automobili large M/M/16

$$\lambda = 4.05263161 \,\text{richieste/min}$$

$$\lambda_3 = \lambda \cdot p_{large} \cdot (1 - p_{exit}) = 1.15500001 \,\text{richieste/min}$$

$$E(S_i) = \frac{1}{\mu_i} = 10 \,\text{min}$$

$$E(S) = \frac{E(S_i)}{16} = 0.625 \,\text{min}$$

$$\mu_i = \frac{1}{E(S_i)} = 0.1 \,\text{job/min}$$

$$\rho = \frac{\lambda_3}{16\,\mu_i} = 0.72187501$$

$$m = 16$$

$$p_0 = \left[\sum_{i=0}^{m-1} \frac{(m\,\rho)^i}{i!} \; + \; \frac{(m\,\rho)^m}{m!\,(1-\rho)} \right]^{-1} \; = \; 9.25463973 \times 10^{-6}$$

$$P_Q = \frac{(m\,\rho)^m \; p_0}{m!\,(1-\rho)} = 0.15951876$$

$$E(T_q) = \frac{P_Q \; \cdot E(S)}{1-\rho} = 0.35846913 \,\text{min}$$

$$E(N_q) = E(T_q)\,\lambda_3 = 0.41403185$$

$$E(T_s) = E(T_q) + E(S_i) = 10.35846913 \,\text{min}$$

$$E(N_s) = E(T_s)\,\lambda_3 = 11.96403195$$

```
Center2
E[Ts]: mean 10,3550, diff 0,0034 is within ±0,0348
E[Tq]: mean 0,3621, diff 0,0036 is within ±0,0185
E[Si]: mean 9,9929, diff 0,0071 is within ±0,0225
E[Ns]: mean 11,9825, diff 0,0185 is within ±0,0524
E[Nq]: mean 0,4211, diff 0,0071 is within ±0,0219
ρ: mean 0,7226, diff 0,0007 is within ±0,0023
λ: mean 1,1570, diff 0,0020 is within ±0,0026
```

Figura 20: Valori di verifica automatica per il centro large

Indice	Valore analitico	Valore empirico	Intervallo di confidenza
$E[N_S]$	23.1826	23.1457	± 0.0539
$E[T_S]$	10.0358	10.0267	± 0.0161
$E[N_Q]$	0.0826	0.0765	± 0.0067
$E[T_Q]$	0.0358	0.0330	± 0.0029
ρ	0.7	0.6991	± 0.0015

Tabella 2: medie analitiche, empiriche e intervalli di confidenza del centro small

Indice	Valore analitico	Valore empirico	Intervallo di confidenza
$E[N_S]$	4.2967	4.2925	± 0.0435
$E[T_S]$	11.1603	11.1257	± 0.0888
$E[N_Q]$	0.4467	0.4468	± 0.0253
$E[T_Q]$	1.1603	1.1448	± 0.0616
ρ	0.6417	0.6409	± 0.0036

Tabella 3: medie analitiche, empiriche e intervalli di confidenza del centro medium

Indice	Valore analitico	Valore empirico	Intervallo di confidenza
$E[N_S]$	11.9640	11.9825	± 0.0524
$E[T_S]$	10.3585	10.3550	± 0.0348
$E[N_Q]$	0.4140	0.4211	± 0.0219
$E[T_Q]$	0.3585	0.3621	± 0.0185
ρ	0.7218	0.7226	± 0.0023

Tabella 4: medie analitiche, empiriche e intervalli di confidenza del centro large

9 Validazione

La fase di validazione ha l'obiettivo di verificare che il modello computazionale sia coerente e rappresentativo del sistema reale che si intende simulare. In genere, questa fase prevede il confronto dei risultati del modello con dati empirici o con misurazioni del sistema reale, al fine di accettarne la correttezza e la capacità predittiva.

Nel nostro caso, tuttavia, non è stato possibile effettuare una validazione, in quanto non sono disponibili dati osservati del sistema in esame. Tale mancanza di informazioni ci impedisce di confrontare direttamente le prestazioni simulate con il sistema reale.

10 Modello migliorativo

Nella seconda fase dello studio, si ipotizza l'introduzione di un servizio di **ride-sharing** all'interno dell'applicazione Bolt, con l'obiettivo di consentire a più passeggeri di condividere lo stesso veicolo. In questo scenario, una richiesta può essere abbinata a un veicolo condiviso solo se entrambe le seguenti condizioni risultanto soddisfatte:

1. Il veicolo deve avere un numero **sufficiente** di **posti liberi** per ospitare tutti i passeggeri coinvolti.

2. L'itinerario della nuova richiesta deve essere **compatibile** con quelle già in corso.

Nella piattaforma in esame, l'assegnazione di una richiesta a un veicolo viene gestita tramite complessi algoritmi di matching che considerano molteplici fattori simultaneamente. Nel nostro studio, questo processo viene semplificato adottando un algoritmo che valuta principalmente le **due condizioni** sopra indicate, assegnando la richiesta al primo veicolo compatibile e disponibile secondo queste regole. Nel nostro modello, un veicolo adibito al ride-sharing può accettare più richieste finché ha posti liberi disponibili. Inoltre, come nel servizio tradizionale, un utente potrebbe decidere di cancellare la propria richiesta di corsa prima che questa venga assegnata ad un veicolo. Infine, nel caso in cui non sia stato possibile effettuare il matching, la richiesta di corsa verrà dirottata verso il servizio di trasporto tradizionale.

11 Modello concettuale

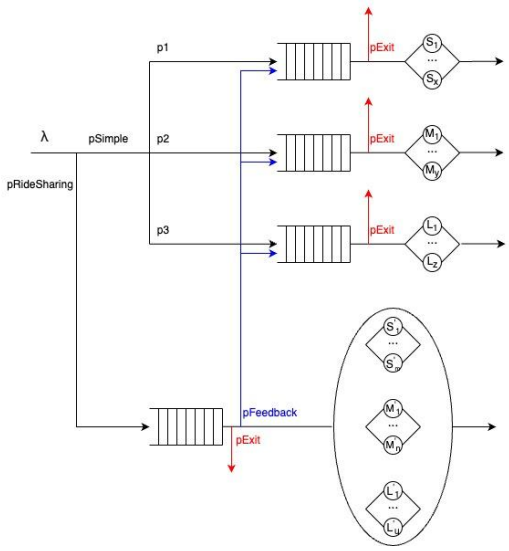


Figura 21: Modello concettuale del servizio di ride-hailing con aggiunta del servizio di ride-sharing

Gli elementi principali di questo sistema sono:

1. **Utente**: colui che desidera richiedere una corsa.
2. **Auto**: il veicolo che può essere richiesto da un utente.

In particolare si ha:

1. Centro di **servizio tradizionale** per veicoli **small**: gestisce le corse effettuate da veicoli con capacità da **1 a 3 passeggeri**.
2. Centro di **servizio tradizionale** per veicoli **medium**: gestisce le corse effettuate da veicoli con capacità da **4 passeggeri**.
3. Centro di **servizio tradizionale** per veicoli **large**: gestisce le corse effettuate da veicoli con capacità da **5 a 8 passeggeri**.
4. Centro di **servizio ride-sharing**: gestisce le corse effettuate da veicoli adibiti al **ride-sharing**.

11.1 Stato del sistema

Nel sistema **migliorativo**, lo stato del sistema in un determinato istante è definito dal vettore

$$(n_{small}, n_{medium}, n_{large}, n_{ride}, s_1, ..., s_m, L_{richieste})$$

- $n_{small}, n_{medium}, n_{large}, n_{ride}$: il **numero di richieste** nei **centri di servizio**.
- $s_1, ..., s_m$: la **capacità rimanente**, espressa in termine di **posti disponibili**, dei veicoli **attualmente impegnati in servizio di ride-sharing**.
- $L_{richieste}$: l'**elenco** delle **richieste in attesa di matching** nel centro ride-sharing.

11.2 Descrizione degli eventi

Nel **sistema migliorativo**, gli eventi che modificano lo stato del sistema sono:

1. L'**arrivo di una richiesta** di corsa per una specifica **tipologia** di veicolo per il servizio **tradizionale**.
2. Il **termine di una corsa** che rende un veicolo nuovamente **disponibile** per il servizio **tradizionale**.
3. L'**uscita dal sistema** di un utente che ha annullato la propria **richiesta di servizio**.
4. L'**arrivo di una richiesta** di corsa che prevede l'utilizzo del servizio di **ride-sharing**.
5. Il **matching** tra una richiesta e un veicolo con posti disponibili per il servizio di **ride-sharing**.
6. Il **completamento di una corsa** che prevede l'utilizzo del servizio di **ride-sharing**.

12 Modello delle specifiche

12.1 Matrice di routing

Il modello migliorativo è caratterizzato da due probabilità, $\mathbf{P_{Simple}}$ e $\mathbf{P_{RideSharing}}$, che rappresentano rispettivamente la probabilità di ricevere una richiesta destinata al servizio tradizionale e la probabilità di ricevere una richiesta destinata al nuovo servizio di ride-sharing. Considerando l'ipotesi in cui quest'ultimo servizio sia stato appena introdotto nell'applicazione, è ragionevole stimare che, almeno nel periodo iniziale, soltanto una ridotta percentuale di utenti (circa il 30%) scelga di sperimentarlo. Analogamente al modello base, inoltre, il modello migliorativo prevede anch'esso una probabilità di abbandono dalla coda, indicata con $\mathbf{P_{Exit}}$. In aggiunta, viene introdotta una nuova probabilità, denotata con $\mathbf{P_{Feedback}}$, che rappresenta la possibilità che una richiesta di *ride-sharing* non possa essere soddisfatta. In questo caso, la richiesta non viene scartata, ma viene reindirizzata al servizio tradizionale, garantendo così che l'utente non perda completamente l'opportunità di ottenere un veicolo.

La tabella di routing risultante è la seguente:

	Esterno	Centro small	Centro medium	Centro large	Centro ride-sharing
Esterno	0	$p_{\text{Simple}}\, p_1$	$p_{\text{Simple}}\, p_2$	$p_{\text{Simple}}\, p_3$	$p_{\text{RideSharing}}$
Centro small	1	0	0	0	0
Centro medium	1	0	0	0	0
Centro large	1	0	0	0	0
Centro ride-sharing	$1 - p_{\text{Feedback}}$	$p_{\text{Feedback}}\, p_1$	$p_{\text{Feedback}}\, p_2$	$p_{\text{Feedback}}\, p_3$	0

12.2 Equazioni di traffico

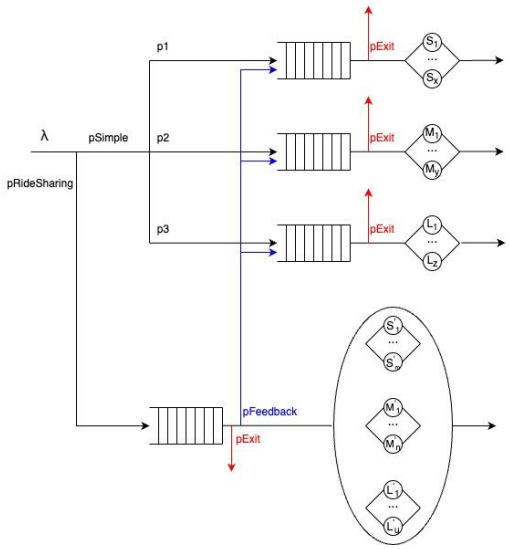


Figura 22: Modello concettuale del servizio di ride-hailing con aggiunta del servizio di ride-sharing

Le equazioni di traffico del sistema sono:

$$\begin{cases} \lambda_1 = p_1\ p_{Simple}\ (1 - p_{Exit})\ \lambda + p_{RideSharing}\ p_{Feedback}\ (1 - p_{Exit})\ p_1\ \lambda \\ \lambda_2 = p_2\ p_{Simple}\ (1 - p_{Exit})\ \lambda + p_{RideSharing}\ p_{Feedback}\ (1 - p_{Exit})\ p_2\ \lambda \\ \lambda_3 = p_3\ p_{Simple}\ (1 - p_{Exit})\ \lambda + p_{RideSharing}\ p_{Feedback}\ (1 - p_{Exit})\ p_3\ \lambda \\ \lambda_4 = p_{RideSharing}\ (1 - p_{Exit})\ (1 - p_{Feedback})\ \lambda \end{cases}$$

12.3 Modellazione dei centri di servizio

- I centri di servizio "classici" sono modellati come code **M/N/k**, caratterizzati da arrivi di tipo **Poisson**, un numero **finito** di server e da una **coda di attesa a capacità infinita**, alimentata da un pool di utenti. Nel sistema sono presenti tre centri di questo tipo, ciascuno dedicato a una specifica tipologia di veicolo: **small**, **medium** e **large**. Il numero di server in ciascun centro è determinato in funzione della categoria di veicolo che esso gestisce.

- Il centro di servizio "ride-sharing" è modellato come coda **M/N/k**[1], caratterizzato da arrivi di tipo **Poisson**, un numero **finito** di server e da una **coda di attesa a capacità infinita**, alimentata da un pool di utenti. A differenza dei sistemi M/N/k classici, i server di questo centro non sono tutti identici: per ciascuna tipologia di veicolo (**small**, **medium**, **large**) è previsto un numero specifico di server all'interno dello stesso centro.

12.4 Scelta delle distribuzioni

In assenza di dati sufficienti per un'analisi statistica precisa, le scelte adottate per modellare i tempi di servizio nei diversi centri mirano a rappresentare in modo realistico il comportamento dei processi.

- **Distribuzione Gaussiana Troncata – Tempi di Servizio**

  Per modellare i tempi di servizio è stata adottata una distribuzione **Gaussiana troncata**, considerata adatta a descrivere un processo con forte concentrazione dei valori attorno alla media. Questa distribuzione permette, infatti, di catturare l'andamento centrato dei tempi di servizio, pur lasciando spazio a variazioni legate a corse più brevi o più lunghe. La troncatura inferiore evita la generazione di tempi irrealisticamente bassi, mentre quella superiore pone un limite massimo coerente con le tempistiche reali del sistema.

- **Distribuzione Esponenziale – Tempi di Arrivo**

  Per modellare gli istanti di arrivo delle richieste si è scelta la distribuzione **esponenziale**, ampiamente impiegata in ambito modellistico per rappresentare processi di arrivo casuali. La scelta è giustificata dalla proprietà di assenza di memoria tipica di questa distribuzione: la probabilità di un prossimo arrivo non dipende dal tempo trascorso dall'ultimo evento. Questo riflette adeguatamente l'elevata aleatorietà del flusso di richieste e l'indipendenza tra arrivi successivi.

[1]Si precisa che la notazione M/N/k è utilizzata *per convenzione*: il modello considera esplicitamente le differenze tra i server, quindi il comportamento simulato riflette la realtà del centro e non è limitato dalla semplificazione della notazione.

13 Modello computazionale

Il modello computazionale mantiene la struttura del modello di base, integrando estensioni e adattamenti per rappresentare fedelmente le specificità del nuovo sistema in esame.

13.1 Eventi

La classe `MsqEvent` viene estesa nel modello ride-sharing come segue:

```
public class MsqEvent {
    double t;
    int x;
    int postiRichiesti;
    int capacitaRimanente;
    int numRichiesteServite;
    int capacita;
    double svc;
}
```

Gli attributi principali della classe sono:

- **t**: istante temporale in cui l'evento si verifica;
- **x**: variabile di stato che indica se l'evento è attivo o meno;
- **postiRichiesti**: numero di posti richiesti dall'utente associato all'evento;
- **numRichiesteServite**: numero di richieste aggregate su un singolo server;
- **capacita**: capacità totale del server;
- **capacitaRimanente**: posti ancora disponibili sul server;
- **svc**: tempo di servizio cumulato erogato dal server fino a quel momento.

13.2 RideSharingSystem

La logica di simulazione del modello ride-sharing mantiene la stessa struttura di base implementata da `SimpleSystem`, con le opportune estensioni per rappresentare le specificità del nuovo servizio.

13.3 RideSharingMultiServerNode

La classe `RideSharingMultiServerNode` incapsula la logica operativa del centro ride-sharing. Nel modello sono presenti due varianti di questa classe:

- `RideSharingMultiServerNodeSimple`, che replica il comportamento del servizio tradizionale;
- `RideSharingMultiServerNode`, che implementa la logica del servizio ride-sharing.

Per motivi di chiarezza e sintesi, nella presente documentazione viene riportata esclusivamente la versione dedicata al servizio ride-sharing, poiché la logica del servizio tradizionale è già stata descritta nel modello base.

Il metodo `processNextEvent()` nella versione ride-sharing è quindi modificato come illustrato di seguito:

```
{
  public int processNextEvent(double t) {
        int e = peekNextEventType();
        clock.current = t;

        if (e == ARRIVAL) {
            numberJobInSystem++;

            MsqEvent arr = new MsqEvent();
            arr.t = distrs.getNextArrivalTimeRideSharing(rng, clock.
                ↪ current);
            arr.x = 1;
            arr.postiRichiesti = getNumPosti();
            event.set(ARRIVAL, arr);

            rng.selectStream(6);
            double p = rng.random();
            if (p < P_EXIT) {
                numberJobInSystem--;
                return -1;
            }

            pendingArrivals.add(arr);

            if (Double.isInfinite(nextMatchTime)) {
                nextMatchTime = clock.current + TIME_WINDOW;
```

```
            }

        } else {
            MsqEvent sEvent = event.get(e);
            int numRichiesteServite = sEvent.numRichiesteServite;
            numberJobInSystem -= numRichiesteServite;
            sum[e].served += numRichiesteServite;
            sum[e].service += event.get(e).svc;

            sEvent.x = 0;
            sEvent.capacitaRimanente = sEvent.capacita;
            sEvent.numRichiesteServite = 0;
            sEvent.postiRichiesti = 0;

            return e;
        }

        if (clock.current >= nextMatchTime) {
            while (true) {
                int matched = findOne();
                if (matched == 0) {
                    if (!pendingArrivals.isEmpty()) {
                        numberJobInSystem --;
                        MsqEvent toFb = pendingArrivals.getFirst();
                        generateFeedback(toFb);
                        pendingArrivals.removeFirst();
                        continue;
                    }
                    break;
                }
            }
            nextMatchTime = Double.POSITIVE_INFINITY;
        }

        return -1;
    }
}
```

Nel caso di elaborazione di arrivi nel servizio ride-sharing, il sistema valuta la possibilità di assegnare una richiesta a un server disponibile. L'algoritmo di matching è stato esteso rispetto al modello tradizionale per tener conto di due aspetti specifici del ride-sharing:

- **Capacità residua**: il server può accogliere più richieste contempora-

neamente fino a esaurire la propria capacità;

- **Match con server già attivo**: una richiesta può essere assegnata a un server già in servizio con una certa probabilità `P_MATCH_BUSY`, rispettando comunque il vincolo sulla capacità residua.

Per realizzare questa logica, si utilizza una politica di *selection in order*, implementata dal metodo `findOne()`. In sintesi, l'algoritmo procede come segue:

1. Si seleziona il primo evento di arrivo in attesa (`firstReq`).
2. Si scansionano prima i server attivi (`x = 1`) per verificare se possono accogliere la richiesta senza superare la propria capacità residua e rispettando la probabilità di match con server occupato.
3. Se un server attivo idoneo viene trovato, la richiesta viene assegnata a quel server.
4. In caso contrario, si cercano server inattivi (`x = 0`) in grado di ospitare la richiesta; se viene individuato un server libero, la richiesta viene assegnata e si tenta di aggregare ulteriori richieste in coda fino al completamento della capacità disponibile.

```
{
  public int findOne() {
        if (pendingArrivals.isEmpty()) return 0;

        MsqEvent firstReq = pendingArrivals.getFirst();

        int bestActive = -1;
        double bestCapActive = -1;

        rng.selectStream(7);
        for (int i = 1; i <= RIDESERVERS; i++) {
            if (event.get(i).x == 1
                    && event.get(i).capacitaRimanente >= firstReq.
                        ↪ postiRichiesti
                    && rng.random() < P_MATCH_BUSY
                    && event.get(i).capacitaRimanente > bestCapActive)
                        ↪  {
                bestCapActive = event.get(i).capacitaRimanente;
                bestActive = i;
```

```
            }
        }

        if (bestActive != -1) {
            assignToServer(bestActive, firstReq);
            pendingArrivals.removeFirst();
            return 1;
        }

        int bestIdle = -1;
        double bestCapIdle = -1;
        rng.selectStream(8);
        for (int i = 1; i <= RIDESERVERS; i++) {
            if (event.get(i).x == 0
                    && event.get(i).capacitaRimanente >= firstReq.
                        ↪ postiRichiesti
                    && event.get(i).capacitaRimanente > bestCapIdle) {
                bestCapIdle = event.get(i).capacitaRimanente;
                bestIdle = i;
            }
        }
        if (bestIdle == -1) return 0;

        event.get(bestIdle).x = 1;
        int totalMatched = 0;
        Iterator<MsqEvent> it = pendingArrivals.iterator();
        while (it.hasNext()) {
            MsqEvent req = it.next();
            if (req.postiRichiesti <= event.get(bestIdle).
                ↪ capacitaRimanente) {
                assignToServer(bestIdle, req);
                it.remove();
                totalMatched++;
                if (event.get(bestIdle).capacitaRimanente == 0) break;
            }
        }
        return totalMatched;
    }
}
```

13.4 getNextArrivalTimeRideSharing() e getServiceTimeRideSharing()

Per generare i tempi di arrivo e di servizio, il modello sfrutta la classe `Rngs` con un approccio multi-stream, garantendo l'indipendenza tra le sequenze di numeri casuali utilizzate dalle diverse componenti del sistema.

Si mostra come esempio il codice di `getNextArrivalTimeRideSharing()`:

```
{
  public double getNextArrivalTimeRideSharing(Rngs r, double sarrival
      ↪ ) {
        r.selectStream(2);
        double lambda = config.getDouble("simulation","lambdaride") *
            ↪  config.getDouble("simulation","pride");
        sarrival += exponential(1/lambda, r);
        return sarrival;
    }
}
```

Mentre `getServiceTimeRideSharing()` ha la seguente implementazione:

```
{
  public double getServiceTimeRideSharing(Rngs r) {
        r.selectStream(4);
        double esi = config.getDouble("simulation","esi");
        double alpha, beta;
        double a = 2;
        double b = 40;

        alpha = cdfNormal(esi, 4, a);
        beta = cdfNormal(esi, 4, b);

        double u = uniform(alpha, beta, r);
        return idfNormal(esi, 4, u);
    }
}
```

14 Design degli esperimenti

14.1 Intervalli di confidenza

All'interno della trattazione sono stati utilizzati gli intervalli di confidenza. Un intervallo di confidenza è un intervallo calcolato a partire dai dati campionari, che ha una certa probabilità di contenere il valore reale della media della popolazione. In altre parole, fornisce una stima dell'incertezza associata alla media campionaria, indicando un intervallo plausibile entro cui ci si può ragionevolmente aspettare che la media reale del sistema si trovi.

Per questo studio sono stati scelti intervalli con un *livello di confidenza* pari al **95%**. Gli intervalli di confidenza sono stati calcolati utilizzando la seguente formula:

$$Intervallo\ di\ Confidenza = t^* \left(\frac{s}{\sqrt{n-1}} \right)$$

dove

- **s** è la deviazione standard campionaria;
- **n** è la dimensione del campione;
- $\mathbf{t}^*$ è il valore critico della distribuzione t di *Student* (nel codice, il calcolo è stato effettuato utilizzando la funzione `idfStudent()` fornita dalla classe `Rvms`).

14.2 Primo Obiettivo

Il primo obiettivo è gestire lo stesso carico di lavoro con l'introduzione del servizio di ride-sharing, individuando il numero ottimale di veicoli da assegnare a ciascuna tipologia, in modo da ottenere una distribuzione del carico tra i centri di servizio quanto più omogenea possibile. Ogni nuova configurazione è stata impostata cercando di mitigare gli eventuali colli di bottiglia individuati nella configurazione precedente.

14.2.1 Analisi del transitorio

Prima di procedere con l'esecuzione degli esperimenti, è fondamentale analizzare la fase transitoria del sistema per verificare ***se*** e ***quando*** esso raggiunge uno stato stazionario. A tal fine, si esegue una simulazione a orizzonte finito (*finite-horizon*), osservando il sistema per un tempo limitato. Per ottenere stime affidabili delle statistiche transitorie, la simulazione viene replicata più volte a partire dallo stesso stato iniziale. In ciascuna replica, i generatori di numeri pseudo-casuali sono inizializzati utilizzando lo **stato finale** degli stream `rng` della replica precedente. Questo garantisce che le sequenze casuali siano indipendenti e non si sovrappongano tra le repliche, permettendo di catturare correttamente la variabilità naturale del sistema e di valutare con precisione il comportamento transitorio e la convergenza verso lo stato stazionario.

Analogamente al modello base, il tempo medio di risposta $E[T_s]$ è stato scelto come indicatore della convergenza transitoria. Nella prima configurazione, considerando che solo una quota limitata di utenti opta per il servizio di ride-sharing, sono stati destinati a questo servizio **15** veicoli, suddivisi tra le tre categorie (small, medium, large) secondo la proporzione **9-1-4**, in modo da rispecchiare la distribuzione attesa delle richieste. I principali risultati ottenuti per questa configurazione sono riportati di seguito:

14.2.1.1 Configurazione 9-1-4

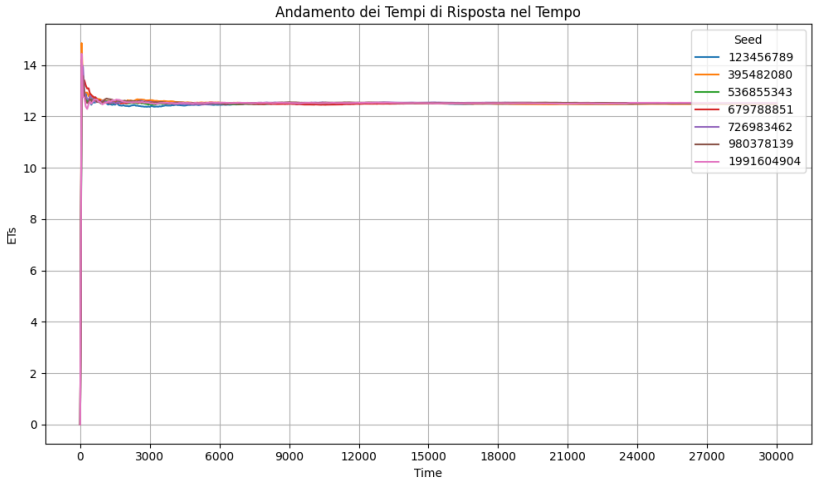
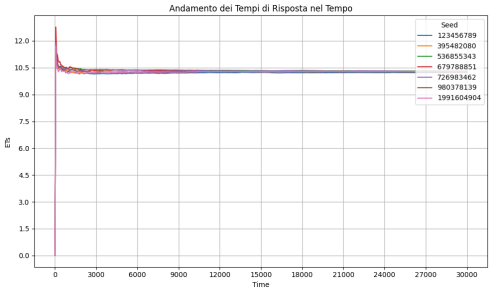
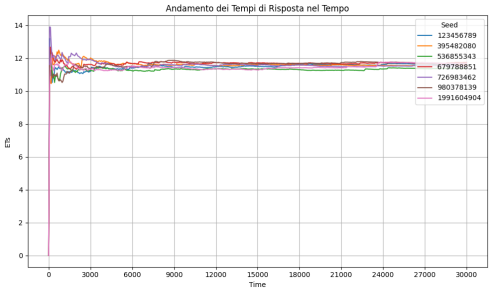


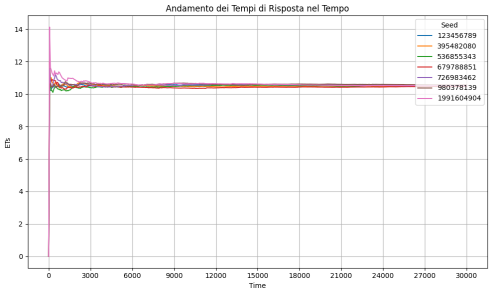
Figura 23: Risultati globali



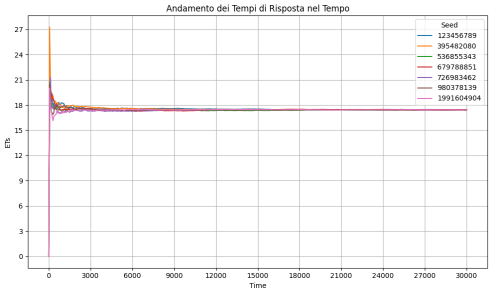
(a) Centro small



(b) Centro medium



(c) Centro large



(d) Centro ride-sharing

14.2.1.2 Configurazione 8-1-3

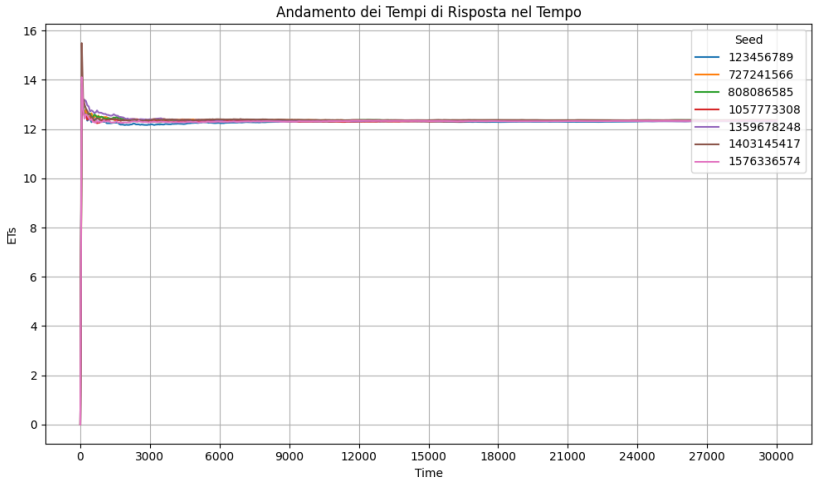
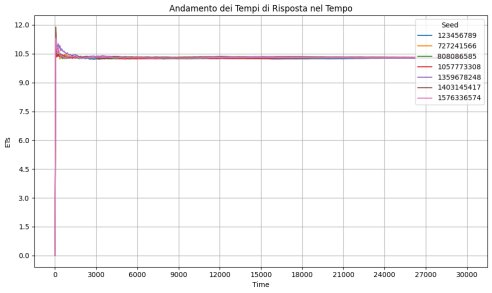
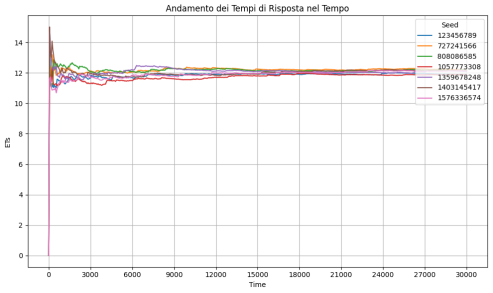


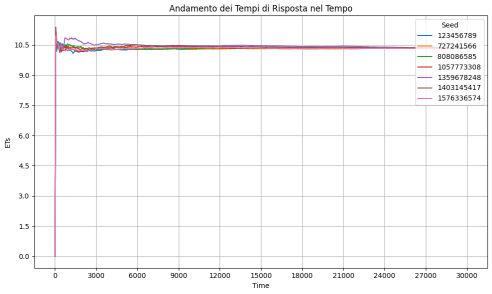
Figura 25: Risultati globali



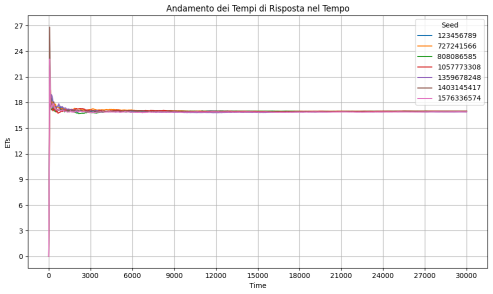
(a) Centro small



(b) Centro medium



(c) Centro large



(d) Centro ride-sharing

14.2.1.3 Configurazione 6-0-3

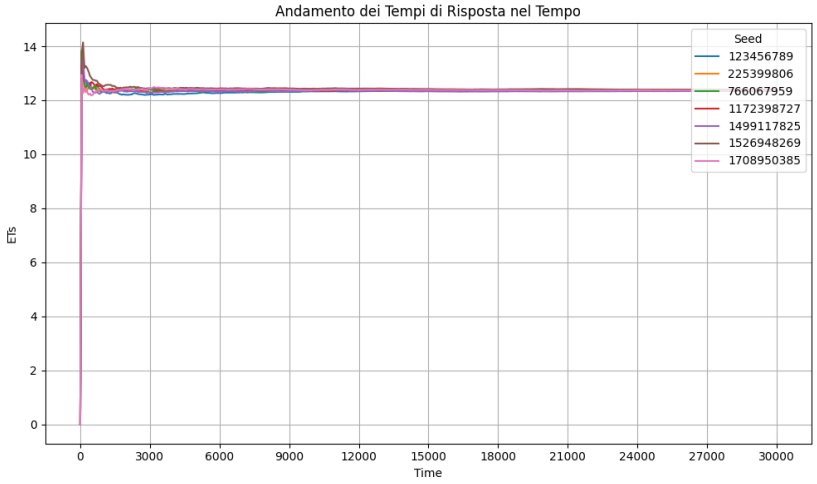
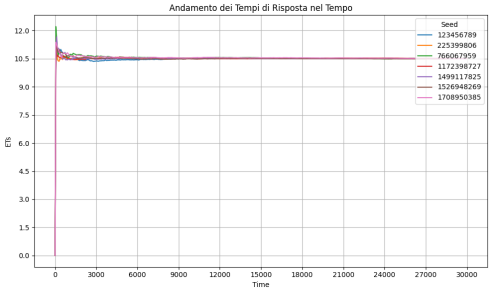
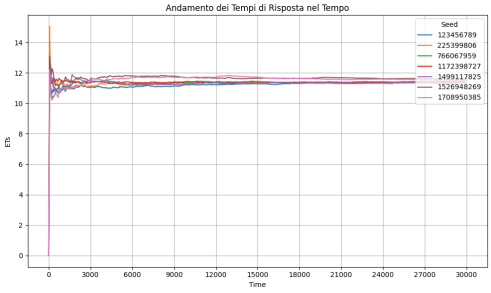


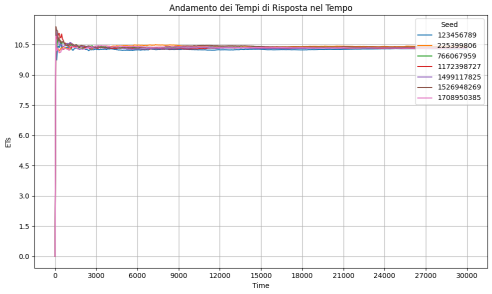
Figura 27: Risultati globali



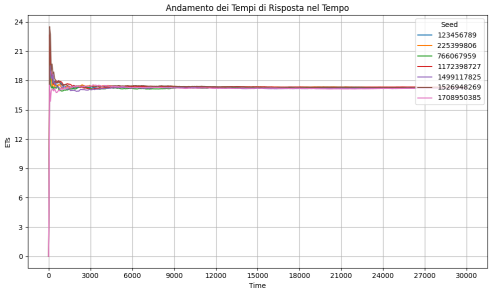
(a) Centro small



(b) Centro medium



(c) Centro large



(d) Centro ride-sharing

14.2.2 Simulazione ad orizzonte infinito

Lo scopo di una simulazione a orizzonte infinito è stimare le caratteristiche del sistema in regime stazionario. Questo approccio è stato adottato per analizzare il comportamento asintotico del sistema, cioè il periodo in cui le metriche di interesse diventano indipendenti dalle condizioni iniziali. Considerando che Bolt è un'applicazione già ampiamente consolidata e adottata dagli utenti e che lo studio si concentra su una singola fascia operativa a partire da uno stato iniziale non vuoto, la simulazione a orizzonte infinito risulta particolarmente appropriata poiché consente di ottenere stime delle prestazioni operative prive del bias introdotto dalla fase transitoria.

Per stimare le statistiche di interesse, si è applicato il metodo delle ***Batch Means***. I parametri scelti sono stati $\mathbf{b = 4096}$ (dimensione di ciascun batch) e $\mathbf{k = 512}$ (numero complessivo di batch). Tali valori non sono stati fissati a priori: inizialmente si era optato per $b = 256$ e $k = 64$, ma entrambi i parametri sono stati aumentati progressivamente fino a garantire che la *lag-1 autocorrelazione* tra le medie di batch risultasse inferiore a **0.2**, in accordo con quanto raccomandato in [4]. In questo modo si è raggiunto un compromesso soddisfacente perché da un lato le medie di batch risultano sufficientemente indipendenti; dall'altro, il numero di batch rimane abbastanza ampio da permettere una stima affidabile della varianza.

È noto che, all'inizio della simulazione, le caratteristiche del sistema differiscono da quelle in regime stazionario, ad esempio quando il sistema viene avviato vuoto. È quindi necessario eseguire la simulazione per un certo periodo di tempo prima che il sistema raggiunga lo stato stazionario. Le osservazioni raccolte durante questo periodo iniziale potrebbero compromettere la precisione delle stime degli indici di prestazione, specialmente se il periodo di transizione è particolarmente lungo. Questo problema è noto come *initialization bias* o *start-up problem* [5]. Per mitigare tale problema, si è deciso di scartare le osservazioni corrispondenti ai primi $r \cdot b \cdot k$ job processati. Dopo alcune prove, il parametro $\mathbf{r}$ è stato fissato a **0.2** sia per il modello iniziale sia per quello migliorato. Non esiste una regola universale per determinare il valore ottimale di r; risulta, tuttavia, importante evitare valori troppo elevati, poiché questi ridurrebbero eccessivamente la quantità di dati disponibili e potrebbero aumentare l'autocorrelazione tra le osservazioni.

Eseguendo la simulazione, per le medesime configurazioni valutate nell'analisi del transitorio, si ottengono i risultati:

14.2.2.1 Configurazione 9-1-4

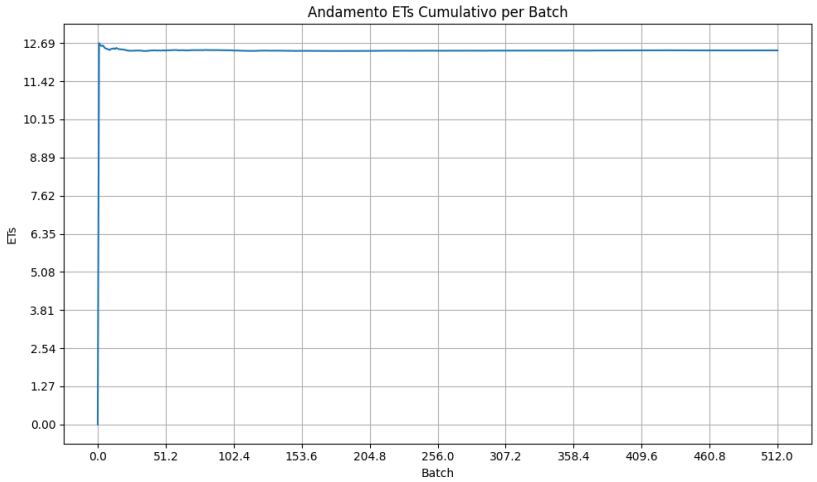
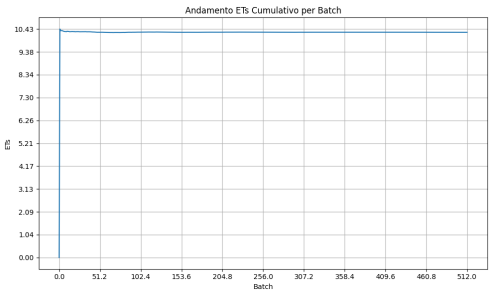
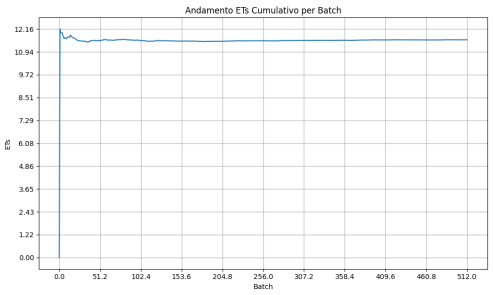


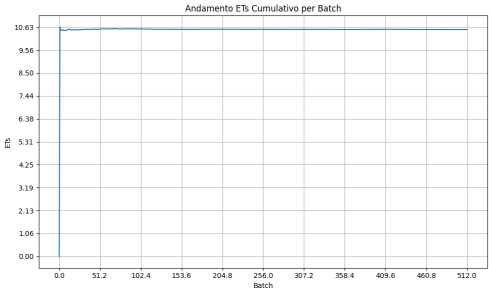
Figura 29: Risultati globali



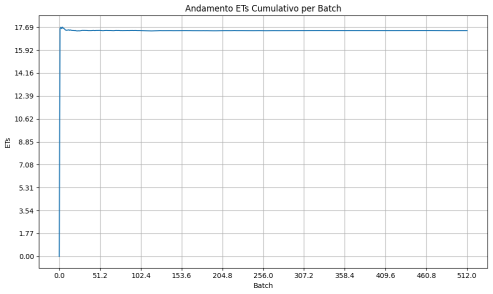
(a) Centro small



(b) Centro medium



(c) Centro large



(d) Centro ride-sharing

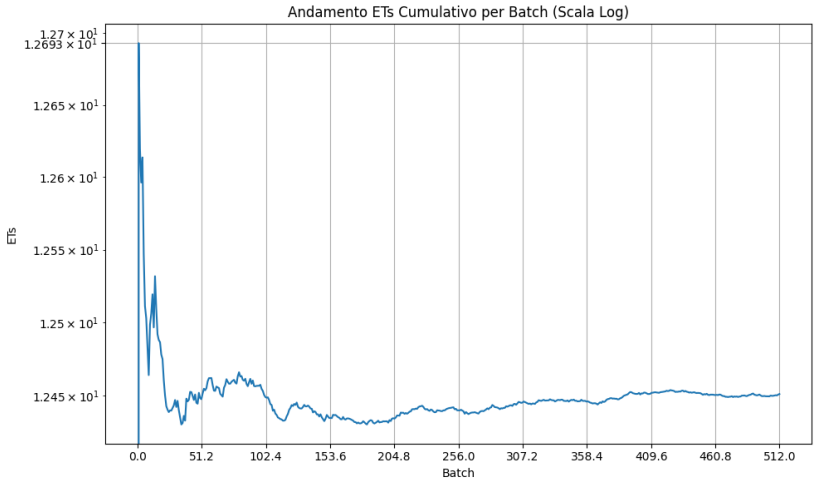
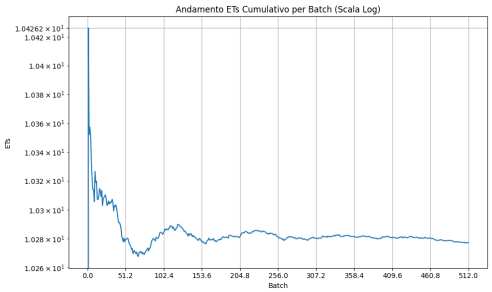
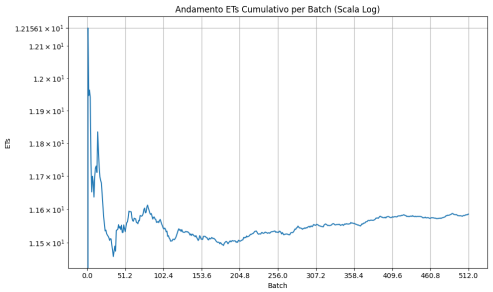


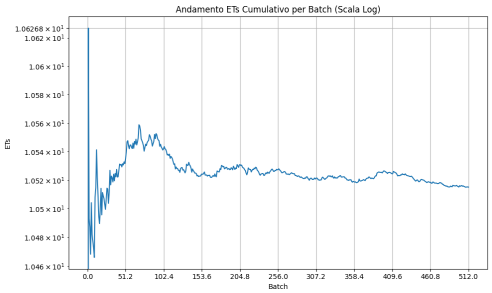
Figura 31: Risultati globali scala log.



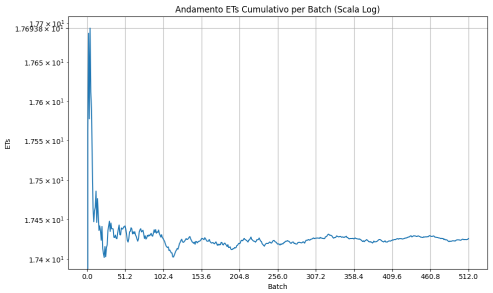
(a) Centro small scala log.



(b) Centro medium scala log.



(c) Centro large scala log.



(d) Centro ride-sharing scala log.

Centro	$E[N_S]$	$E[T_S]$	$E[N_Q]$	$E[T_Q]$	ρ	$E[S]$
small	16.6470 ± 0.0386	10.2772 ± 0.0091	0.0901 ± 0.0050	0.0550 ± 0.0030	0.6899 ± 0.0015	10.2219 ± 0.0081
medium	3.5605 ± 0.0289	11.5859 ± 0.0489	0.3087 ± 0.0130	0.9914 ± 0.0388	0.6503 ± 0.0037	10.5945 ± 0.0208
large	8.5072 ± 0.0326	10.5150 ± 0.0170	0.2389 ± 0.0102	0.2930 ± 0.0120	0.6890 ± 0.0021	10.2221 ± 0.0110
ride-sharing	19.4743 ± 0.0449	17.4257 ± 0.0230	3.1625 ± 0.0092	2.8297 ± 0.0060	0.4046 ± 0.0137	14.5960 ± 0.022

14.2.2.2 Configurazione 8-1-3

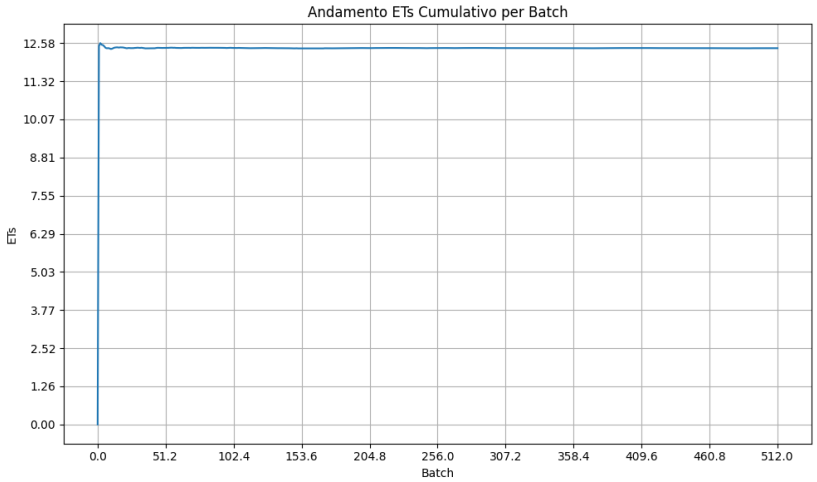
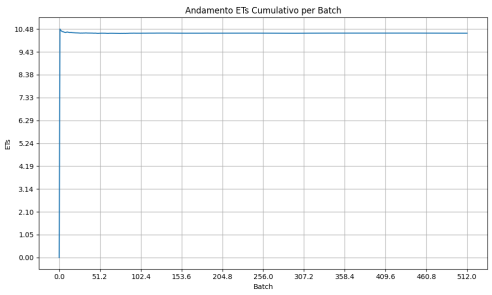
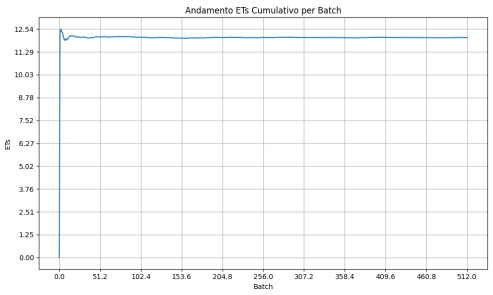


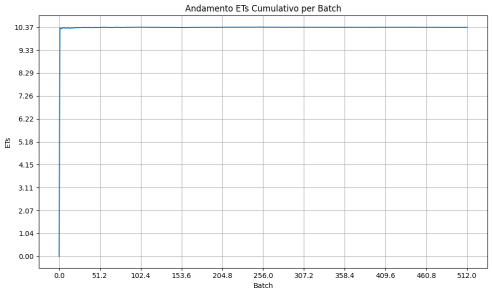
Figura 33: Risultati globali



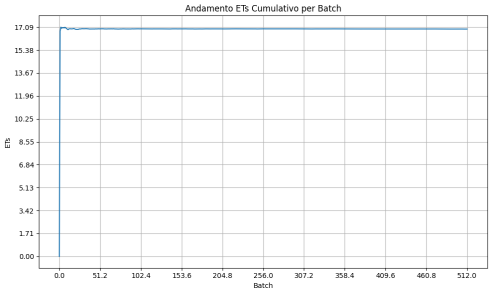
(a) Centro small



(b) Centro medium



(c) Centro large



(d) Centro ride-sharing

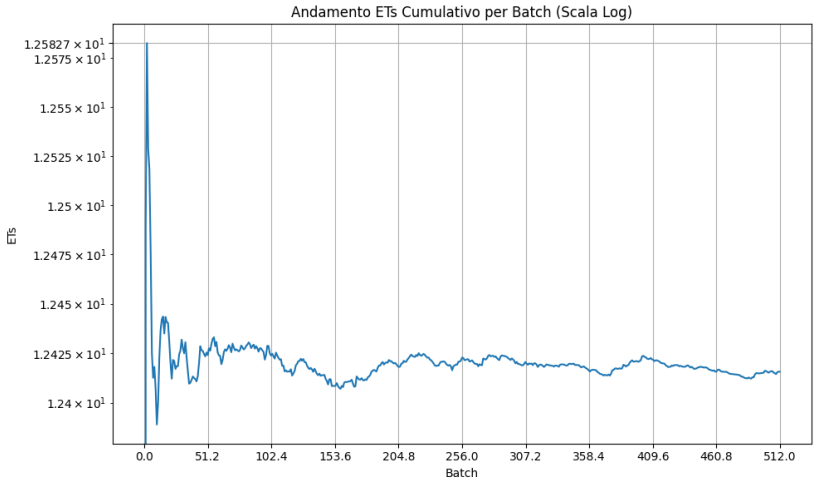
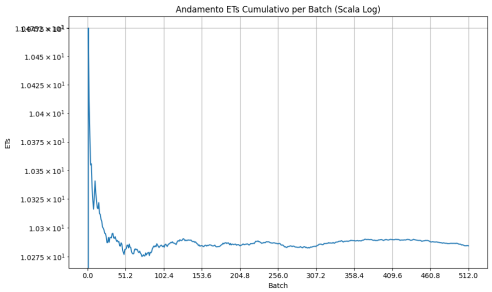
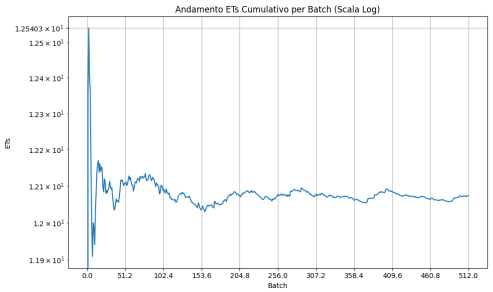


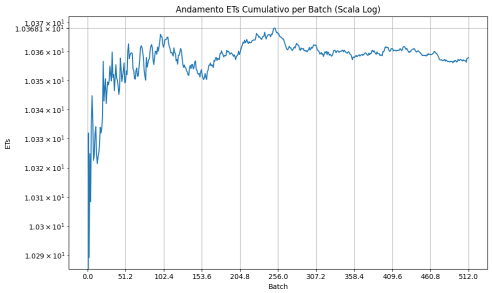
Figura 35: Risultati globali scala log.



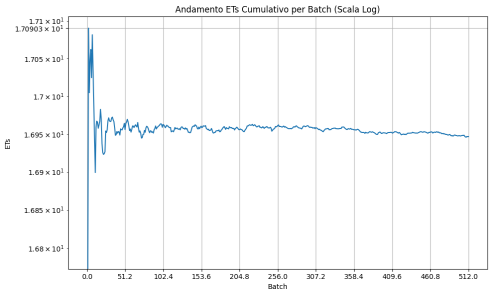
(a) Centro small scala log.



(b) Centro medium scala log.



(c) Centro large scala log.



(d) Centro ride-sharing scala log.

Centro	$E[N_S]$	$E[T_S]$	$E[N_Q]$	$E[T_Q]$	ρ	$E[S]$
small	16.7449 ± 0.0403	10.2843 ± 0.0087	0.0541 ± 0.0037	0.0330 ± 0.0022	0.6676 ± 0.0015	10.2513 ± 0.0079
medium	3.9146 ± 0.0332	12.0738 ± 0.0595	0.4118 ± 0.0176	1.2548 ± 0.0492	0.7005 ± 0.0038	10.8190 ± 0.0231
large	8.3796 ± 0.0302	10.3579 ± 0.0149	0.1086 ± 0.0059	0.1329 ± 0.0070	0.6362 ± 0.0020	10.2250 ± 0.0115
ride-sharing	18.5128 ± 0.0424	16.9465 ± 0.0224	3.1624 ± 0.0091	2.8948 ± 0.0062	0.5024 ± 0.0158	14.0517 ± 0.0212

14.2.2.3 Configurazione 6-0-3

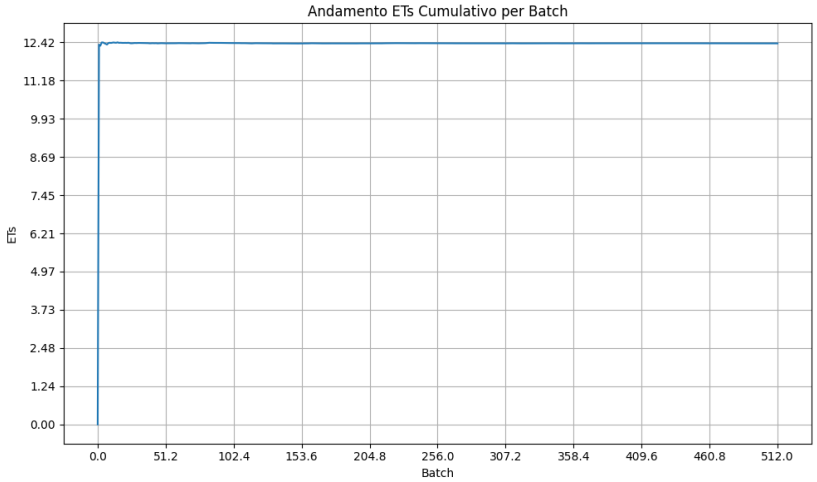
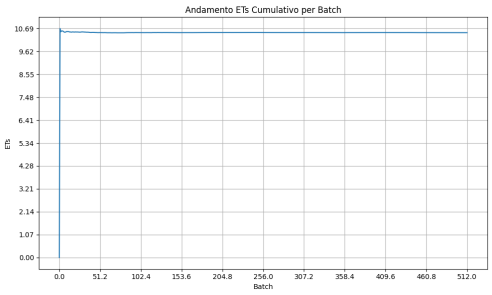
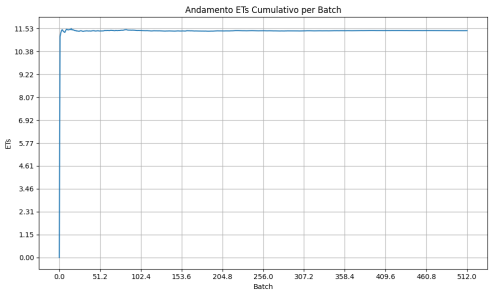


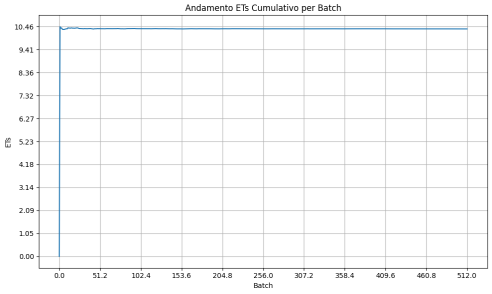
Figura 37: Risultati globali



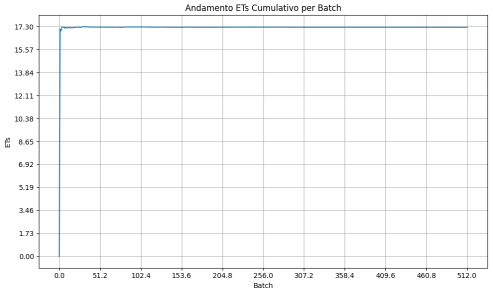
(a) Centro small



(b) Centro medium



(c) Centro large



(d) Centro ride-sharing

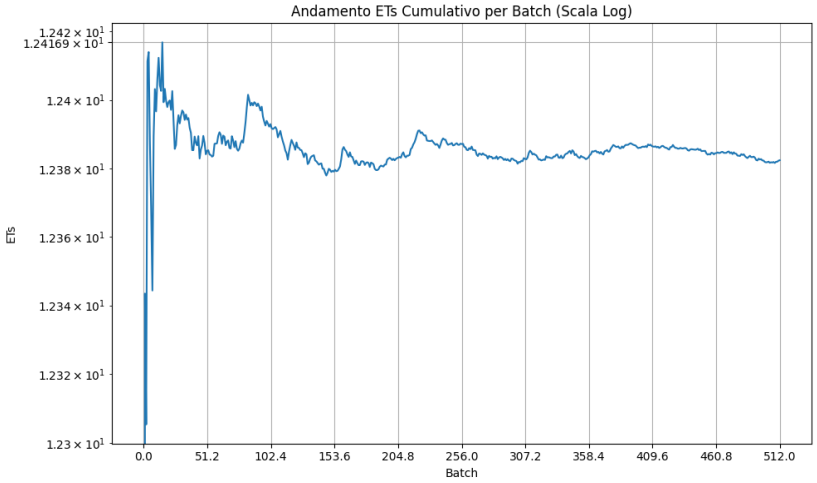
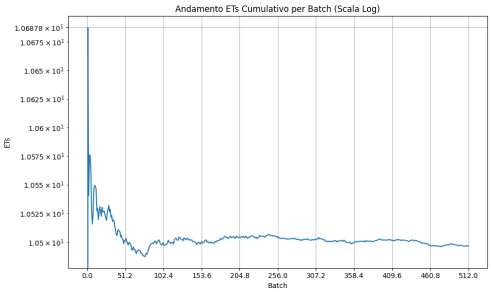
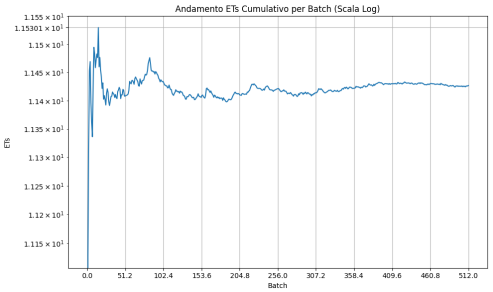


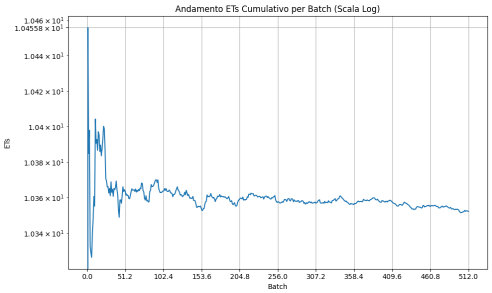
Figura 39: Risultati globali scala log.



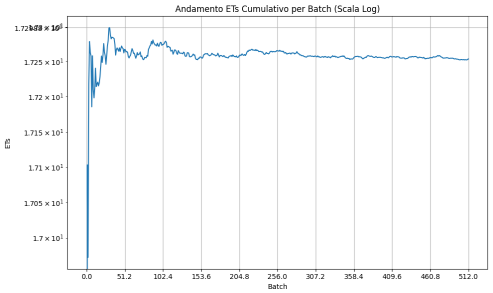
(a) Centro small scala log.



(b) Centro medium scala log.



(c) Centro large scala log.



(d) Centro ride-sharing scala log.

Centro	$E[N_S]$	$E[T_S]$	$E[N_Q]$	$E[T_Q]$	ρ	$E[S]$
small	17.5944 ± 0.0439	10.4966 ± 0.0110	0.0319 ± 0.0026	0.0189 ± 0.0015	0.6505 ± 0.0016	10.4777 ± 0.0107
medium	3.8318 ± 0.0259	11.4265 ± 0.0355	0.1504 ± 0.0072	0.4427 ± 0.0199	0.6136 ± 0.0035	10.9839 ± 0.0236
large	8.3738 ± 0.0283	10.3522 ± 0.0145	0.1097 ± 0.0054	0.1346 ± 0.0065	0.6357 ± 0.0019	10.2176 ± 0.0118
ride-sharing	17.8217 ± 0.0373	17.2540 ± 0.0227	3.1621 ± 0.0091	3.0614 ± 0.0072	0.6135 ± 0.0187	14.1926 ± 0.0215

14.3 Secondo Obiettivo

Il secondo obiettivo dello studio consiste nel valutare l'effettiva possibilità di **ridurre** il **numero di veicoli** necessari per gestire il carico di lavoro iniziale grazie all'introduzione del servizio di ride-sharing. Dall'analisi dei risultati è emerso che la configurazione minima in grado di riportare l'utilizzo dei vari centri ai livelli precedenti all'introduzione del ride-sharing è la seguente:

- **(23, 5, 11)**: veicoli allocati per il servizio tradizionale per tipologia (small, medium, large);
- **(5, 0, 4)**: veicoli allocati per il servizio di ride-sharing per tipologia (small, medium, large).

Risulta quindi possibile ridurre il numero di veicoli small da **33** a **28**, il numero di veicoli medium da **6** a **5**, e il numero di veicoli large da **16** a **15**, ottenendo una riduzione complessiva del numero di veicoli pari al $\mathbf{12.\overline{72}}\%$.

14.3.1 Analisi del transitorio

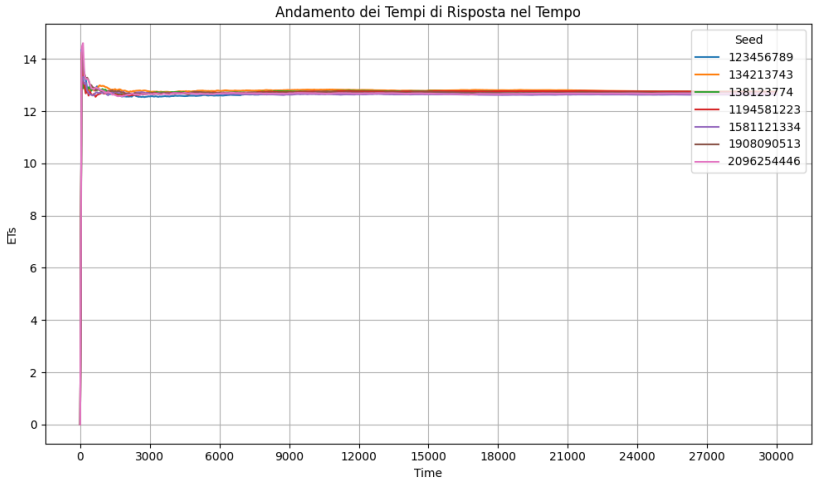
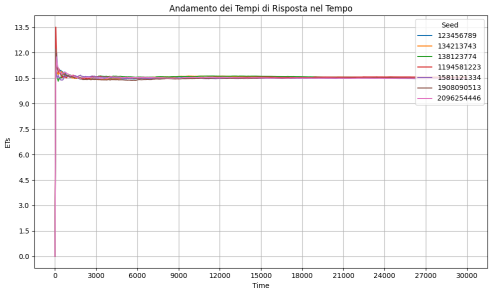
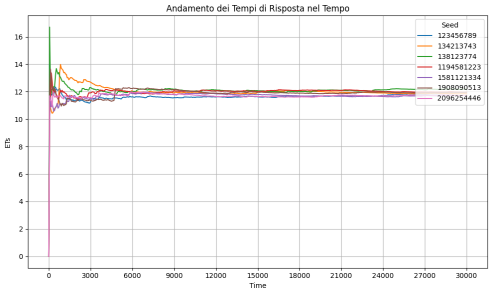


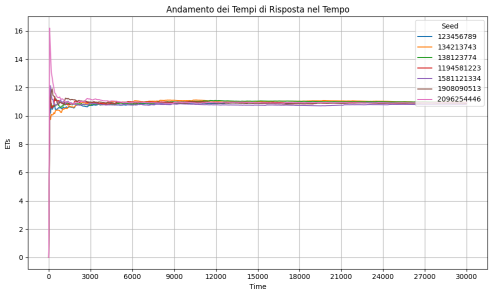
Figura 41: Risultati globali



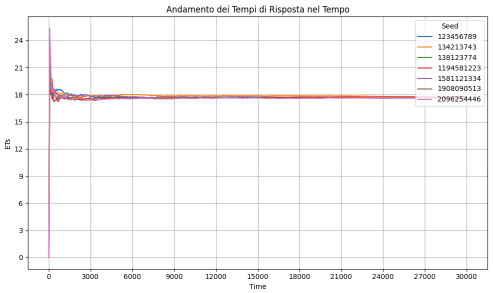
(a) Centro small



(b) Centro medium



(c) Centro large



(d) Centro ride-sharing

14.4 Studio Orizzonte Infinito

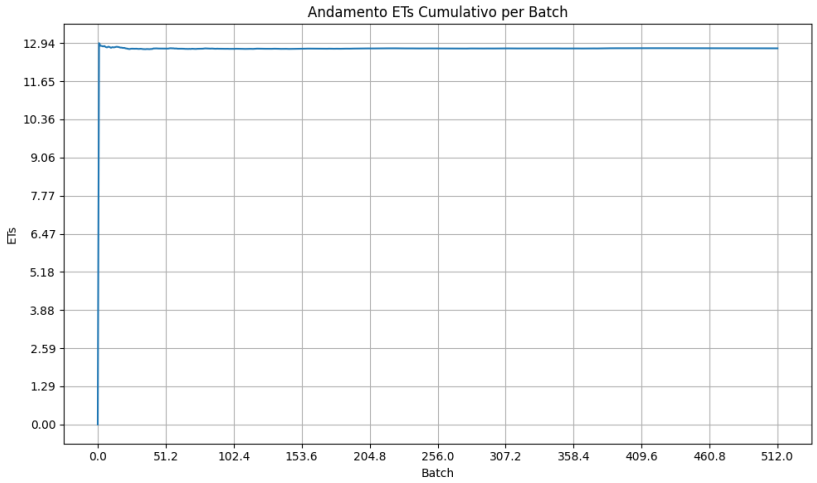
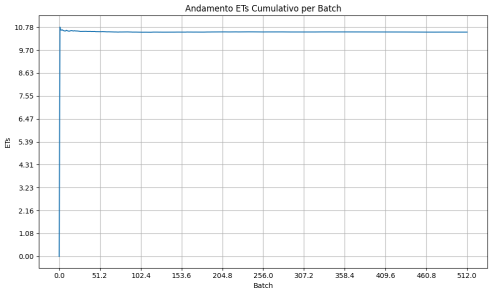
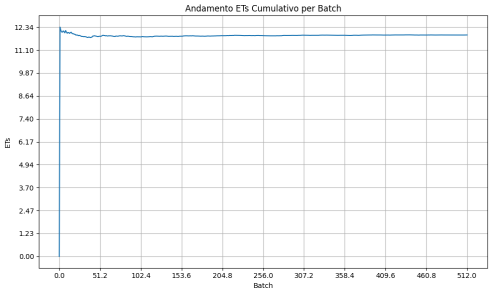


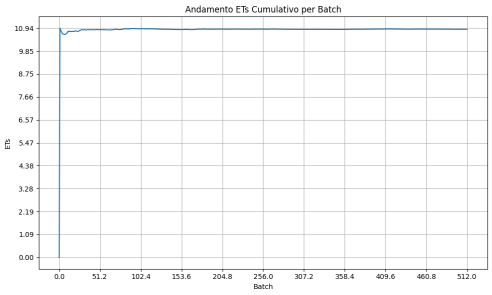
Figura 43: Risultati globali



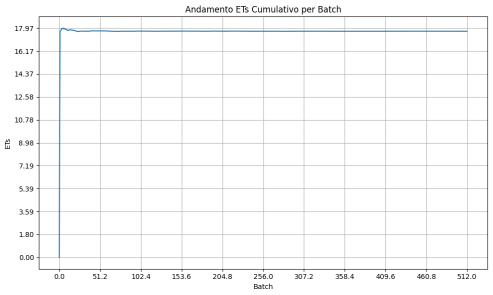
(a) Centro small



(b) Centro medium



(c) Centro large



(d) Centro ride-sharing

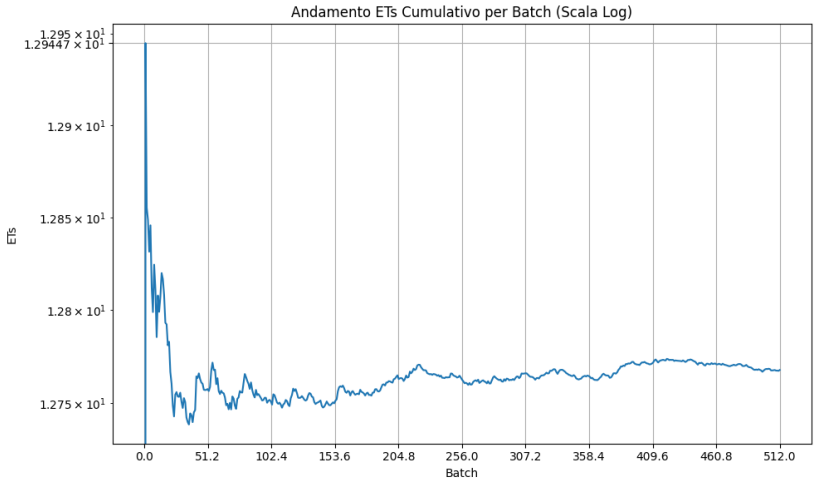
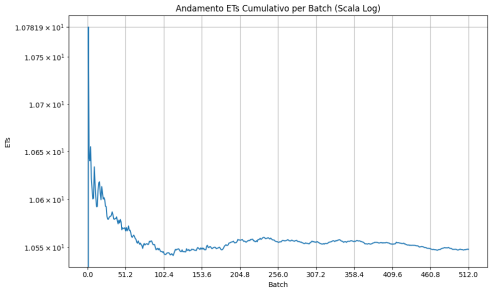
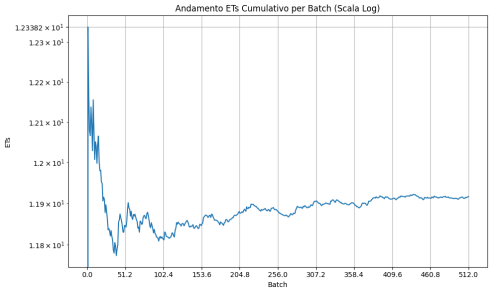


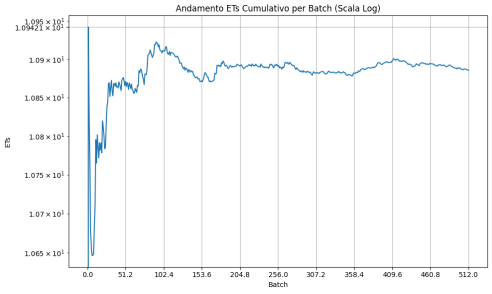
Figura 45: Risultati globali scala log.



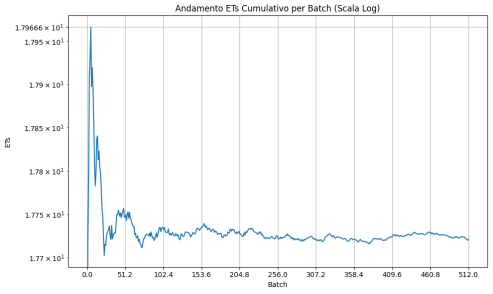
(a) Centro small scala log.



(b) Centro medium scala log.



(c) Centro large scala log.



(d) Centro ride-sharing scala log.

Centro	$E[N_S]$	$E[T_S]$	$E[N_Q]$	$E[T_Q]$	ρ	$E(S)$
small	17.5098 ± 0.0477	10.5477 ± 0.0129	0.2478 ± 0.0114	0.1484 ± 0.0067	0.7505 ± 0.0018	10.3993 ± 0.0095
medium	3.7946 ± 0.0321	11.9168 ± 0.0579	0.3762 ± 0.0161	1.1669 ± 0.0463	0.6837 ± 0.0038	10.7499 ± 0.0221
large	8.8103 ± 0.0455	10.8858 ± 0.0324	0.5451 ± 0.0230	0.6677 ± 0.0268	0.7514 ± 0.0024	10.2181 ± 0.0120
ride-sharing	18.8983 ± 0.0402	17.7211 ± 0.0258	3.1623 ± 0.0089	2.9651 ± 0.0064	0.6011 ± 0.0182	14.7559 ± 0.0241

14.5 Autocorrelazione

Di seguito sono riportati i valori dell'autocorrelazione delle statistiche della **configurazione 6-0-3** forniti dal simulatore, che utilizza la classe di libreria `Acs.java`:

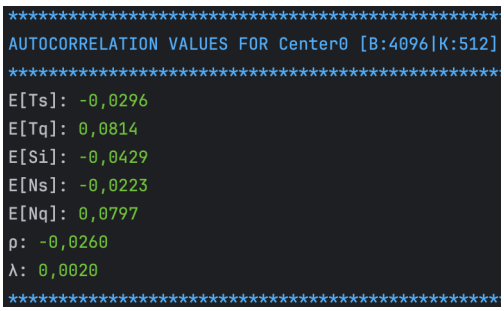


Figura 47: Valori di autocorrelazione del centro small

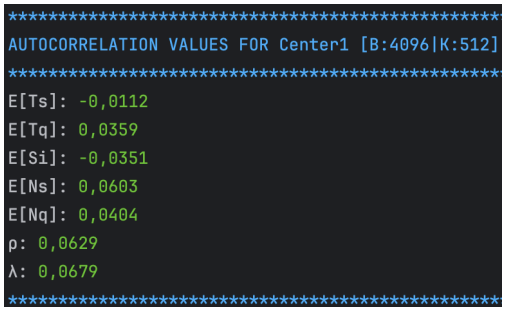


Figura 48: Valori di autocorrelazione del centro medium

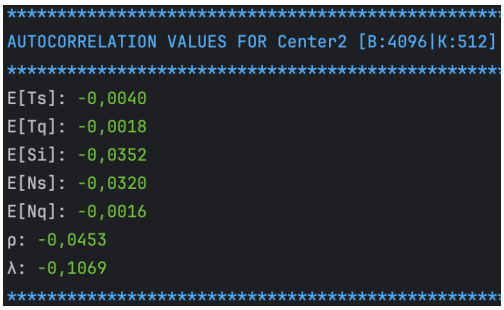


Figura 49: Valori di autocorrelazione del centro large

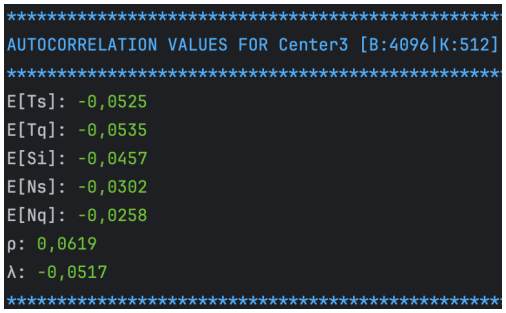


Figura 50: Valori di autocorrelazione del centro ride-sharing

15 Verifica

La verifica del modello migliorativo può essere condotta distinguendo le due principali componenti del sistema:

1. **Centri di servizio classici**: disabilitando la generazione del feedback per le richieste non soddisfatte dall'algoritmo di matching del ride-sharing, il comportamento dei centri semplici rimane identico a quello del modello base. Infatti, senza il feedback, i flussi di arrivo rimangono separati: uno diretto ai centri semplici e l'altro al centro ride-sharing. Pertanto, non sono necessari ulteriori controlli rispetto a quelli già eseguiti nella verifica del modello semplice, a cui si rimanda per i dettagli.

2. **Centro Ride Sharing**: per questa componente, la struttura dei server, con capacità differenziate e la possibilità di gestire contemporaneamente più richieste, impedisce l'applicazione dei tradizionali approcci analitici di verifica. Di conseguenza, l'unica strada percorribile consiste nell'effettuare controlli di consistenza interna sui risultati della simulazione, finalizzati a confermare che la logica implementata rispetti le ipotesi del modello. I principali controlli includono:

   - Controllo della coerenza delle statistiche raccolte;
   - Controllo specifico sull'interazione tra le due componenti.

15.1 Controllo della coerenza delle statistiche raccolte

Sui risultati degli esperimenti riportati nel paragrafo precedente sono stati effettuati i seguenti controlli di consistenza:

$$E(T_S) = E(T_Q) + E(S)$$
$$\text{Legge di Little}^2\text{: } E(N_S) = \lambda E(T_S); E(N_Q) = \lambda E(T_Q)$$

15.2 Controllo specifico sull'interazione tra le due componenti

Si è verificato che le richieste non soddisfatte dal centro ride-sharing vengano effettivamente reindirizzate ai centri di servizio classici e che questo trasferimento comporti un aumento conforme del flusso dei centri di servizio classici.

[2]Tenendo conto degli intervalli di confidenza

Tale controllo garantisce la corretta integrazione tra i due sistemi dei risultati della simulazione.

Utilizzazione dei centri senza feedback:

Centro	ρ
small	0.6123 ± 0.0014
medium	0.4687 ± 0.0027
large	0.6362 ± 0.0019
ride-sharing	0.6272 ± 0.0217

Si osservi che l'utilizzazione del centro large è assimilabile al sistema con feedback poiché le richieste di ride-sharing possono essere effettuate per un numero di posti massimo di quattro persone.

16 Validazione

La fase di validazione ha l'obiettivo di verificare che il modello computazionale sia coerente e rappresentativo del sistema reale che si intende simulare. In genere, questa fase prevede il confronto dei risultati del modello con dati empirici o con misurazioni del sistema reale, al fine di accettarne la correttezza e la capacità predittiva.

Nel nostro caso, tuttavia, non è stato possibile effettuare una validazione tradizionale, in quanto non sono disponibili dati osservati dettagliati sul servizio di ride-hailing dell'oggetto di studio. La mancanza di informazioni reali ci impedisce di confrontare direttamente le prestazioni simulate con il sistema reale.

17 Conclusioni

Le simulazioni evidenziano che il **modello migliorativo** consente di soddisfare le stesse richieste di servizio del **modello tradizionale** utilizzando un numero inferiore di **veicoli**. L'analisi dell'ultima configurazione mostra infatti come sia possibile contenere il parco veicoli complessivo senza compromettere la qualità del servizio percepita dagli utenti.Questo risultato rappresenta un beneficio concreto in termini di **sostenibilità ambientale** e di supporto a politiche di **mobilità condivisa**, in linea con la **strategia green** di **Bolt**.

Riferimenti bibliografici

[1] https://eur-lex.europa.eu/legal-content/IT/TXT/HTML/?uri=CELEX:32019R0631

[2] https://www.consilium.europa.eu/it/policies/fit-for-55

[3] https://etrr.springeropen.com/articles/10.1186/s12544-025-00708-x

[4] Jerry Banks, John S Carson II, Barry L Nelson, and David M Nicol *Discrete-event system simulation fourth edition*

[5] https://dl.acm.org/doi/10.1145/1096166.1096174