# SIMULACIÓN DEL MATCHMAKING DE CS2

*Simulación de Sistemas*

**INTEGRANTES:**

- Jhosep Marcel Changana Meza ... 2022-119062
- Hayner Joshua Rivero Flores ... 2022-119092
- Angel Eduardo Vivanco Laura ... 2022-119101
- Pedro Luis Aquino García ... 2021-119074

## Matchmaking CS2

```
#pip install numpy pandas plotly scipy joblib
```

```
import sys
#!{sys.executable} -m pip install numpy pandas plotly scipy joblib --break-system-packages -q
import numpy as np
import plotly.colors as pc
import pandas as pd
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots
from scipy.stats import norm
from scipy import stats
from dataclasses import dataclass, field
from typing import List, Optional, Tuple, Dict
from enum import Enum
import math
from collections import defaultdict
import random
from joblib import Parallel, delayed
```

---

## PARTE 1: SIMULACIÓN BASE DE MATCHMAKING

Parámetros basados en investigación real de CS2 (Nov 2025)

### 1.1 Configuración de parámetros

```
np.random.seed(42)
random.seed(42)

SKILL_MEAN = 10000
SKILL_STD = 6000
SKILL_MIN = 1000
SKILL_MAX = 35000

GLICKO_INITIAL_RATING = 1500
GLICKO_INITIAL_RD = 350
GLICKO_INITIAL_VOLATILITY = 0.06
GLICKO_TAU = 0.5
GLICKO_EPSILON = 0.000001
GLICKO_SCALE = 173.7178

MATCH_DURATION_MEAN = 35
MATCH_DURATION_STD = 8
OVERTIME_PROBABILITY = 0.12
OVERTIME_DURATION = 10

PEAK_PLAYERS = 1500000
VALLEY_PLAYERS = 625000
POPULATION_SCALE_FACTOR = 0.001

INITIAL_ELO_WINDOW = 500
WINDOW_EXPANSION_RATE = 50
MAX_ELO_WINDOW = 3000
MAX_WAIT_TIME_MINUTES = 15

AVG_MATCHES_PER_SESSION = 4
```

```
SESSION_FATIGUE_LAMBDA = 1/AVG_MATCHES_PER_SESSION
REST_TIME_MINUTES_MEAN = 5
REST_TIME_MINUTES_STD = 2

N_REPLICAS = 50
N_JOBS = -1
```

## 1.2 Configuración de clases

```
class PlayerState(Enum):
    BUSCANDO = "searching"
    JUGANDO = "playing"
    DESCANSANDO = "resting"
    DESCONECTADO = "offline"

@dataclass
class GlickoRating:
    rating: float = GLICKO_INITIAL_RATING
    rd: float = GLICKO_INITIAL_RD
    volatility: float = GLICKO_INITIAL_VOLATILITY

    def to_cs2_rating(self) -> float:
        normalized = (self.rating - GLICKO_INITIAL_RATING) / (3 * GLICKO_INITIAL_RD)
        cs2_rating = SKILL_MEAN + normalized * (3 * SKILL_STD)
        return np.clip(cs2_rating, SKILL_MIN, SKILL_MAX)

@dataclass
class Player:
    id: int
    hidden_skill: float
    glicko: GlickoRating
    state: PlayerState = PlayerState.BUSCANDO

    matches_played: int = 0
    total_wait_time: float = 0.0
    current_wait_time: float = 0.0
    match_end_time: Optional[float] = None
    rest_end_time: Optional[float] = None
    spawn_time: float = 0.0

    rating_history: List[float] = field(default_factory=list)
    rd_history: List[float] = field(default_factory=list)
    wait_times_history: List[float] = field(default_factory=list)

    @property
    def current_rating(self) -> float:
        return self.glicko.to_cs2_rating()

    def should_disconnect(self) -> bool:
        disconnect_prob = 1 - np.exp(-SESSION_FATIGUE_LAMBDA * self.matches_played)
        return np.random.random() < disconnect_prob

@dataclass
class Match:
    id: int
    players: List[Player]
    start_time: float
    duration: float
    avg_elo: float
    elo_range: float
    predicted_winner: Optional[str] = None
    actual_winner: Optional[str] = None
    win_probability_a: Optional[float] = None

    @property
    def end_time(self) -> float:
        return self.start_time + self.duration

    @property
    def prediction_correct(self) -> bool:
        if self.predicted_winner and self.actual_winner:
            return self.predicted_winner == self.actual_winner
        return False
```

## 1.3. Glicko-2

```
class Glicko2System:
    def __init__(self, tau: float = GLICKO_TAU):
        self.tau = tau

    @staticmethod
    def _g(phi: float) -> float:
        return 1 / math.sqrt(1 + 3 * phi**2 / math.pi**2)

    @staticmethod
    def _E(mu: float, mu_j: float, phi_j: float) -> float:
        return 1 / (1 + math.exp(-Glicko2System._g(phi_j) * (mu - mu_j)))

    def _compute_new_volatility(self, mu: float, phi: float, sigma: float, v: float, delta: float) -> float:
        alpha = math.log(sigma**2)

        def f(x):
            ex = math.exp(x)
            phi2 = phi**2
            return (ex * (delta**2 - phi2 - v - ex) / (2 * (phi2 + v + ex)**2) - (x - alpha) / self.tau**2)

        A = alpha
        if delta**2 > phi**2 + v:
            B = math.log(delta**2 - phi**2 - v)
        else:
            k = 1
            while f(alpha - k * self.tau) < 0:
                k += 1
            B = alpha - k * self.tau

        fA, fB = f(A), f(B)
        while abs(B - A) > GLICKO_EPSILON:
            C = A + (A - B) * fA / (fB - fA)
            fC = f(C)
            if fC * fB <= 0:
                A, fA = B, fB
            else:
                fA = fA / 2
            B, fB = C, fC

        return math.exp(A / 2)

    def update_rating(self, player_rating: GlickoRating, opponents_ratings: List[GlickoRating], outcomes: List[float]) -> G
        mu = (player_rating.rating - 1500) / GLICKO_SCALE
        phi = player_rating.rd / GLICKO_SCALE
        sigma = player_rating.volatility

        if not opponents_ratings:
            phi_star = math.sqrt(phi**2 + sigma**2)
            return GlickoRating(player_rating.rating, phi_star * GLICKO_SCALE, sigma)

        v_inv = delta_sum = 0
        for opp_rating, outcome in zip(opponents_ratings, outcomes):
            mu_j = (opp_rating.rating - 1500) / GLICKO_SCALE
            phi_j = opp_rating.rd / GLICKO_SCALE
            g_phi_j = self._g(phi_j)
            E_val = self._E(mu, mu_j, phi_j)
            v_inv += g_phi_j**2 * E_val * (1 - E_val)
            delta_sum += g_phi_j * (outcome - E_val)

        v = 1 / v_inv if v_inv > 0 else 1e10
        delta = v * delta_sum
        new_sigma = self._compute_new_volatility(mu, phi, sigma, v, delta)
        phi_star = math.sqrt(phi**2 + new_sigma**2)
        new_phi = 1 / math.sqrt(1/phi_star**2 + 1/v)
        new_mu = mu + new_phi**2 * delta_sum

        return GlickoRating(new_mu * GLICKO_SCALE + 1500, new_phi * GLICKO_SCALE, new_sigma)

    @staticmethod
    def predict_win_probability(team_a_ratings: List[float], team_b_ratings: List[float]) -> float:
        avg_a, avg_b = np.mean(team_a_ratings), np.mean(team_b_ratings)
        return 1 / (1 + 10**((avg_b - avg_a) / 400))
```

## 1.3. Gestor de Matchmaking

```
class PopulationManager:
    def __init__(self):
        self.next_player_id = 0
        self.active_players: List[Player] = []
        self.disconnected_players: List[Player] = []
```

```
        def get_target_population(self, hour_of_day: float) -> int:
            peak_hour = 21
            phase_shift = (hour_of_day - peak_hour) * (2 * np.pi / 24)
            avg_pop = (PEAK_PLAYERS + VALLEY_PLAYERS) / 2
            amplitude = (PEAK_PLAYERS - VALLEY_PLAYERS) / 2
            population = avg_pop + amplitude * np.cos(phase_shift)
            return int(population * POPULATION_SCALE_FACTOR)

        def spawn_new_players(self, target_count: int, current_time: float):
            current_count = len([p for p in self.active_players if p.state != PlayerState.DESCONECTADO])
            players_to_spawn = max(0, target_count - current_count)

            for _ in range(players_to_spawn):
                hidden_skill = np.clip(np.random.normal(SKILL_MEAN, SKILL_STD), SKILL_MIN, SKILL_MAX)
                cs2_rating = hidden_skill + np.random.normal(0, 200)
                cs2_rating = np.clip(cs2_rating, SKILL_MIN, SKILL_MAX)

                normalized = (cs2_rating - SKILL_MEAN) / (3 * SKILL_STD)
                glicko_rating = GLICKO_INITIAL_RATING + normalized * (3 * GLICKO_INITIAL_RD)

                player = Player(
                    id=self.next_player_id,
                    hidden_skill=hidden_skill,
                    glicko=GlickoRating(rating=glicko_rating),
                    spawn_time=current_time
                )
                player.rating_history.append(player.current_rating)
                player.rd_history.append(player.glicko.rd)
                self.active_players.append(player)
                self.next_player_id += 1

        def cleanup_disconnected(self):
            still_active = []
            for player in self.active_players:
                if player.state == PlayerState.DESCONECTADO:
                    self.disconnected_players.append(player)
                else:
                    still_active.append(player)
            self.active_players = still_active

    class MatchmakingEngine:
        def __init__(self):
            self.next_match_id = 0
            self.active_matches: List[Match] = []
            self.completed_matches: List[Match] = []
            self.glicko_system = Glicko2System()

        def get_elo_window(self, wait_time: float) -> float:
            return min(INITIAL_ELO_WINDOW + wait_time * WINDOW_EXPANSION_RATE, MAX_ELO_WINDOW)

        def find_matches(self, players: List[Player], current_time: float, match_size: int = 10) -> List[Match]:
            searching = [p for p in players if p.state == PlayerState.BUSCANDO]
            if len(searching) < match_size:
                return []

            searching.sort(key=lambda p: p.current_rating)
            new_matches = []
            used_players = set()
            candidates = sorted(searching, key=lambda p: p.current_wait_time, reverse=True)

            for anchor in candidates:
                if anchor.id in used_players:
                    continue
                if anchor.current_wait_time >= MAX_WAIT_TIME_MINUTES:
                    anchor.state = PlayerState.DESCONECTADO
                    continue

                window = self.get_elo_window(anchor.current_wait_time)
                compatible = [p for p in searching if p.id not in used_players and abs(p.current_rating - anchor.current_rating)

                if len(compatible) >= match_size:
                    compatible.sort(key=lambda p: abs(p.current_rating - anchor.current_rating))
                    match_players = compatible[:match_size]

                    base_duration = np.random.normal(MATCH_DURATION_MEAN, MATCH_DURATION_STD)
                    duration = max(20, base_duration + (OVERTIME_DURATION if np.random.random() < OVERTIME_PROBABILITY else 0))

                    ratings = [p.current_rating for p in match_players]
                    match = Match(
                        id=self.next_match_id,
                        players=match_players,
                        start_time=current_time,
```

```
                    duration=duration,
                    avg_elo=np.mean(ratings),
                    elo_range=max(ratings) - min(ratings)
                )

                for p in match_players:
                    p.state = PlayerState.JUGANDO
                    p.match_end_time = match.end_time
                    p.wait_times_history.append(p.current_wait_time)
                    p.total_wait_time += p.current_wait_time
                    p.current_wait_time = 0.0
                    used_players.add(p.id)

                new_matches.append(match)
                self.next_match_id += 1

        return new_matches

    def process_match_results(self, match: Match):
        team_a = match.players[:5]
        team_b = match.players[5:]

        avg_skill_a = np.mean([p.hidden_skill for p in team_a])
        avg_skill_b = np.mean([p.hidden_skill for p in team_b])
        prob_a_wins = 1 / (1 + 10**((avg_skill_b - avg_skill_a) / 400))
        team_a_wins = np.random.random() < prob_a_wins

        for player in team_a:
            outcome = 1.0 if team_a_wins else 0.0
            player.glicko = self.glicko_system.update_rating(player.glicko, [p.glicko for p in team_b], [outcome] * len(team
            player.rating_history.append(player.current_rating)
            player.rd_history.append(player.glicko.rd)

        for player in team_b:
            outcome = 0.0 if team_a_wins else 1.0
            player.glicko = self.glicko_system.update_rating(player.glicko, [p.glicko for p in team_a], [outcome] * len(team
            player.rating_history.append(player.current_rating)
            player.rd_history.append(player.glicko.rd)

    def update_matches(self, players: List[Player], current_time: float):
        still_active = []
        for match in self.active_matches:
            if current_time >= match.end_time:
                self.process_match_results(match)
                self.completed_matches.append(match)

                for player in match.players:
                    player.matches_played += 1
                    if player.should_disconnect():
                        player.state = PlayerState.DESCONECTADO
                    else:
                        rest_duration = max(1, np.random.normal(REST_TIME_MINUTES_MEAN, REST_TIME_MINUTES_STD))
                        player.state = PlayerState.DESCANSANDO
                        player.rest_end_time = current_time + rest_duration
            else:
                still_active.append(match)
        self.active_matches = still_active
```

## 1.4. Simulación de partidas

```
def run_single_replica(replica_id: int, simulation_hours: int = 24, time_step_minutes: int = 1, seed: int = None) -> Dict:
    if seed is not None:
        np.random.seed(seed + replica_id)
        random.seed(seed + replica_id)

    population_manager = PopulationManager()
    matchmaking_engine = MatchmakingEngine()

    total_steps = int(simulation_hours * 60 / time_step_minutes)
    metrics = {
        'time': [], 'active_population': [], 'target_population': [],
        'searching': [], 'playing': [], 'resting': [],
        'avg_wait_time': [], 'avg_elo_difference': []
    }

    for step in range(total_steps):
        current_time = step * time_step_minutes
        hour_of_day = (current_time / 60) % 24

        target_pop = population_manager.get_target_population(hour_of_day)
```

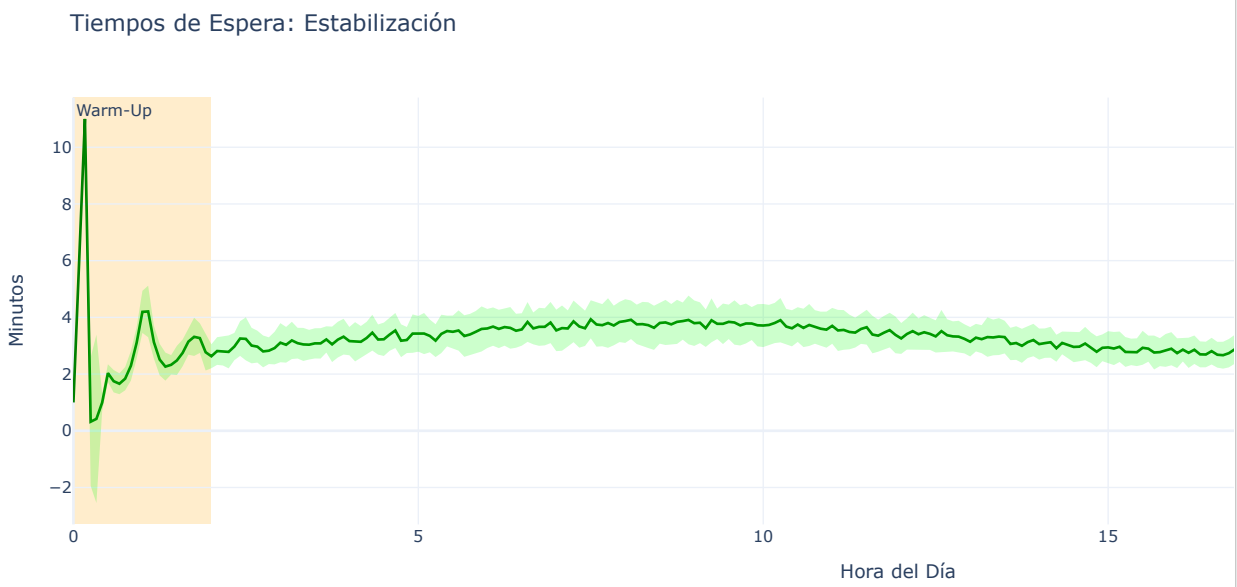
```
            population_manager.spawn_new_players(target_pop, current_time)

            for player in population_manager.active_players:
                if player.state == PlayerState.BUSCANDO:
                    player.current_wait_time += time_step_minutes
                elif player.state == PlayerState.DESCANSANDO and current_time >= player.rest_end_time:
                    player.state = PlayerState.BUSCANDO

            new_matches = matchmaking_engine.find_matches(population_manager.active_players, current_time)
            matchmaking_engine.active_matches.extend(new_matches)
            matchmaking_engine.update_matches(population_manager.active_players, current_time)
            population_manager.cleanup_disconnected()

            if step % 5 == 0:
                active = population_manager.active_players
                states = defaultdict(int)
                for p in active:
                    states[p.state] += 1

                searching_players = [p for p in active if p.state == PlayerState.BUSCANDO]
                avg_wait = np.mean([p.current_wait_time for p in searching_players]) if searching_players else 0
                recent_matches = matchmaking_engine.completed_matches[-100:]
                avg_elo_diff = np.mean([m.elo_range for m in recent_matches]) if recent_matches else 0

                metrics['time'].append(hour_of_day)
                metrics['active_population'].append(len(active))
                metrics['target_population'].append(target_pop)
                metrics['searching'].append(states[PlayerState.BUSCANDO])
                metrics['playing'].append(states[PlayerState.JUGANDO])
                metrics['resting'].append(states[PlayerState.DESCANSANDO])
                metrics['avg_wait_time'].append(avg_wait)
                metrics['avg_elo_difference'].append(avg_elo_diff)

        all_players = population_manager.active_players + population_manager.disconnected_players
        player_data = [{
            'player_id': p.id,
            'spawn_time': p.spawn_time,
            'initial_rating': p.rating_history[0],
            'final_rating': p.rating_history[-1],
            'matches_played': p.matches_played,
            'total_wait_time': p.total_wait_time,
            'avg_wait_time': p.total_wait_time / p.matches_played if p.matches_played > 0 else 0,
            'rating_change': p.rating_history[-1] - p.rating_history[0] if len(p.rating_history) > 1 else 0,
            'hidden_skill': p.hidden_skill
        } for p in all_players if p.matches_played > 0]

        match_data = [{
            'match_id': m.id,
            'start_time': m.start_time / 60,
            'duration': m.duration,
            'avg_elo': m.avg_elo,
            'elo_range': m.elo_range
        } for m in matchmaking_engine.completed_matches]

        return {
            'replica_id': replica_id,
            'metrics': pd.DataFrame(metrics),
            'players': pd.DataFrame(player_data),
            'matches': pd.DataFrame(match_data),
            'total_matches': len(matchmaking_engine.completed_matches),
            'total_players': len(all_players)
        }

    def run_parallel_simulation(n_replicas: int = N_REPLICAS, simulation_hours: int = 24, n_jobs: int = N_JOBS) -> Dict:
        print(f"🚀 Iniciando {n_replicas} réplicas en paralelo (jobs={n_jobs})...\n")

        results = Parallel(n_jobs=n_jobs, verbose=10)(
            delayed(run_single_replica)(i, simulation_hours, seed=42)
            for i in range(n_replicas)
        )

        all_metrics = pd.concat([r['metrics'].assign(replica=r['replica_id']) for r in results], ignore_index=True)
        all_players = pd.concat([r['players'].assign(replica=r['replica_id']) for r in results], ignore_index=True)
        all_matches = pd.concat([r['matches'].assign(replica=r['replica_id']) for r in results], ignore_index=True)

        aggregated_metrics = all_metrics.groupby('time').agg({
            'active_population': ['mean', 'std'],
            'target_population': 'mean',
            'searching': ['mean', 'std'],
            'playing': ['mean', 'std'],
            'resting': ['mean', 'std'],
            'avg_wait_time': ['mean', 'std'],
            'avg_elo_difference': ['mean', 'std']
```

```
    }).reset_index()

    aggregated_metrics.columns = ['_'.join(col).strip('_') for col in aggregated_metrics.columns.values]

    total_matches = sum(r['total_matches'] for r in results)
    total_players = sum(r['total_players'] for r in results)

    print(f"   - Réplicas: {n_replicas}")
    print(f"   - Total partidas: {total_matches:,}")
    print(f"   - Total jugadores: {total_players:,}")
    print(f"   - Promedio partidas/réplica: {total_matches/n_replicas:.0f}\n")

    return {
        'raw_results': results,
        'aggregated_metrics': aggregated_metrics,
        'all_players': all_players,
        'all_matches': all_matches,
        'summary': {
            'n_replicas': n_replicas,
            'total_matches': total_matches,
            'total_players': total_players,
            'avg_matches_per_replica': total_matches / n_replicas,
            'avg_players_per_replica': total_players / n_replicas
        }
    }

simulation_results = run_parallel_simulation(n_replicas=N_REPLICAS, simulation_hours=24)

agg = simulation_results['aggregated_metrics']
all_players = simulation_results['all_players']
all_matches = simulation_results['all_matches']
```

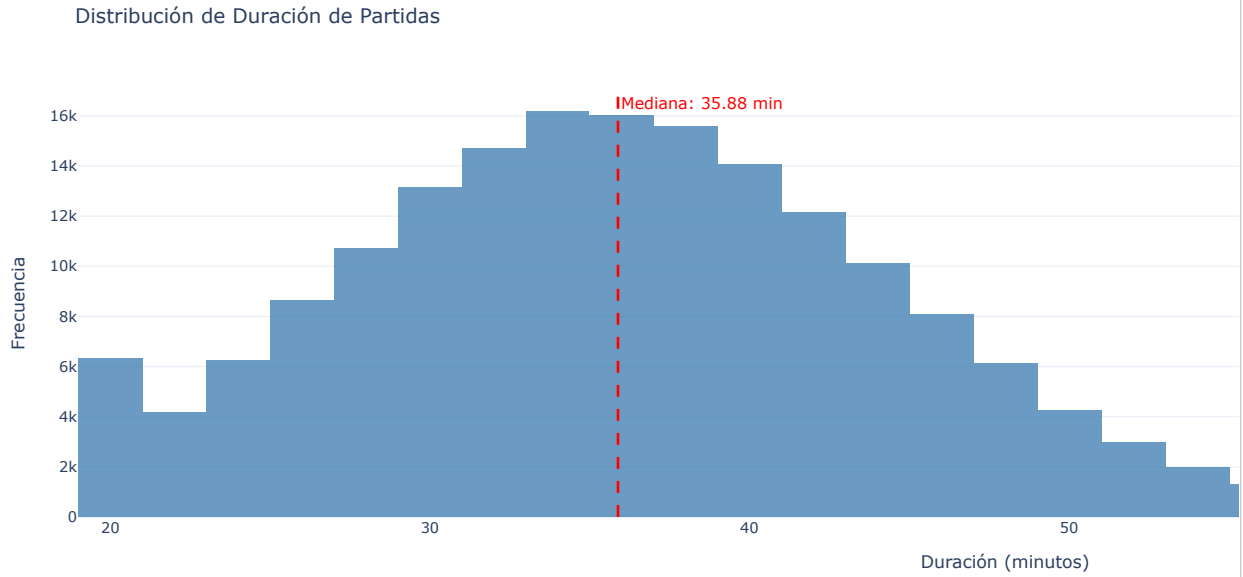
```
🚀 Iniciando 50 réplicas en paralelo (jobs=-1)...

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   1 tasks      | elapsed:  3.3min
[Parallel(n_jobs=-1)]: Done   4 tasks      | elapsed:  6.6min
[Parallel(n_jobs=-1)]: Done   9 tasks      | elapsed: 16.3min
[Parallel(n_jobs=-1)]: Done  14 tasks      | elapsed: 22.9min
[Parallel(n_jobs=-1)]: Done  21 tasks      | elapsed: 35.5min
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed: 45.4min
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed: 61.3min
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed: 74.1min
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed: 80.5min finished
   - Réplicas: 50
   - Total partidas: 174,383
   - Total jugadores: 759,057
   - Promedio partidas/réplica: 3488
```

```
warmup_end = 2
fig2 = go.Figure()

fig2.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['avg_wait_time_mean'],
    mode='lines',
    name='Tiempo Espera Promedio',
    line=dict(color='green', width=2)
))

fig2.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['avg_wait_time_mean'] + agg['avg_wait_time_std'],
    mode='lines',
    line=dict(width=0),
    showlegend=False,
    hoverinfo='skip'
))

fig2.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['avg_wait_time_mean'] - agg['avg_wait_time_std'],
    mode='lines',
    line=dict(width=0),
    fill='tonexty',
    fillcolor='rgba(0,255,0,0.2)',
    name='±1 Desv. Std.',
    hoverinfo='skip'
))

fig2.add_vrect(
    x0=0, x1=warmup_end,
```

```
    fillcolor="orange", opacity=0.2,
    layer="below", line_width=0,
    annotation_text="Warm-Up",
    annotation_position="top left"
)
fig2.update_layout(
    title='Tiempos de Espera: Estabilización',
    xaxis_title='Hora del Día',
    yaxis_title='Minutos',
    template='plotly_white',
    hovermode='x unified',
    height=500
)

fig2.show()
```

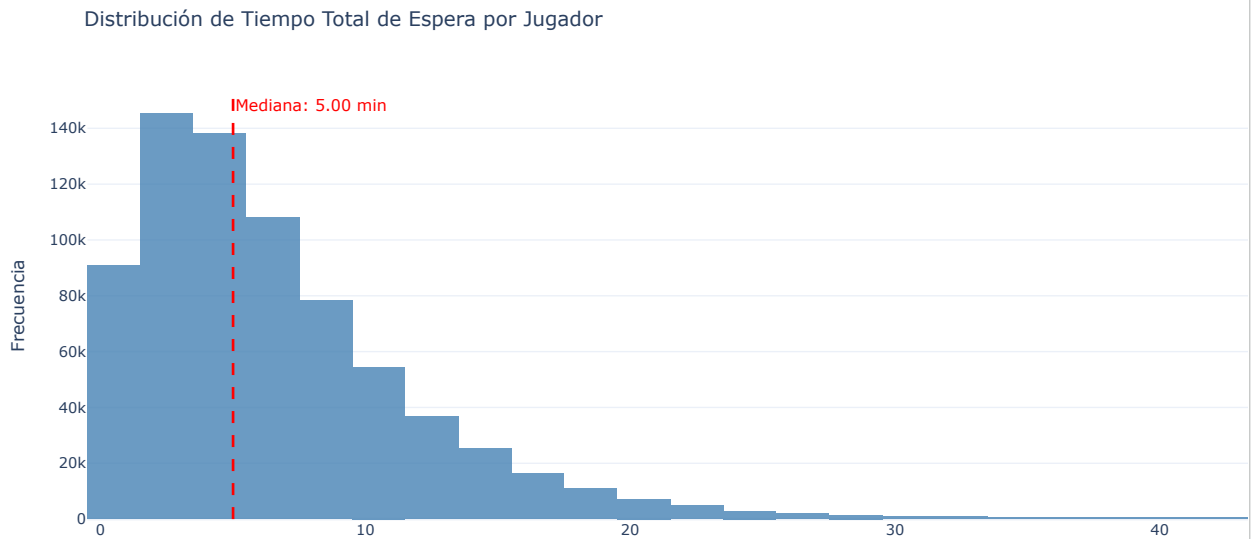


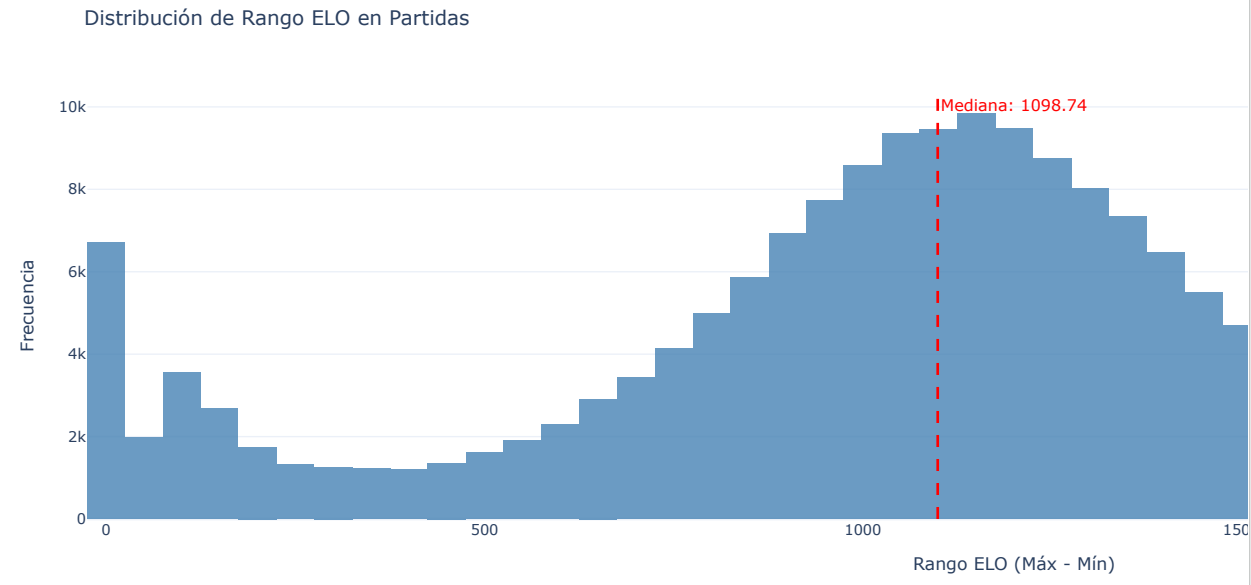

```
all_durations = all_matches['duration'].values
median_duration = np.median(all_durations)

fig_duration_dist = go.Figure()

fig_duration_dist.add_trace(go.Histogram(
    x=all_durations,
    nbinsx=50,
    marker_color='steelblue',
    opacity=0.8,
    hovertemplate='Duración: %{x:.2f} min<br>Frecuencia: %{y}<extra></extra>'
))

fig_duration_dist.add_vline(
    x=median_duration,
    line_dash="dash",
    line_color="red",
    line_width=2,
    annotation_text=f"Mediana: {median_duration:.2f} min",
    annotation_position="top right",
    annotation_font_size=12,
    annotation_font_color="red"
)

fig_duration_dist.update_layout(
    title='Distribución de Duración de Partidas',
    xaxis_title='Duración (minutos)',
    yaxis_title='Frecuencia',
    template='plotly_white',
    height=500,
    showlegend=False,
    font=dict(size=11)
)

fig_duration_dist.show()
```




```
total_wait_times = all_players['total_wait_time'].values
median_total_wait = np.median(total_wait_times)

fig_total_wait_dist = go.Figure()

fig_total_wait_dist.add_trace(go.Histogram(
    x=total_wait_times,
    nbinsx=50,
    marker_color='steelblue',
    opacity=0.8,
    hovertemplate='Tiempo Total: %{x:.2f} min<br>Frecuencia: %{y}<extra></extra>'
))

fig_total_wait_dist.add_vline(
    x=median_total_wait,
    line_dash="dash",
    line_color="red",
    line_width=2,
    annotation_text=f"Mediana: {median_total_wait:.2f} min",
    annotation_position="top right",
    annotation_font_size=12,
    annotation_font_color="red"
)

fig_total_wait_dist.update_layout(
    title='Distribución de Tiempo Total de Espera por Jugador',
    xaxis_title='Tiempo Total de Espera (minutos)',
    yaxis_title='Frecuencia',
    template='plotly_white',
    height=500,
    showlegend=False,
    font=dict(size=11)
)

fig_total_wait_dist.show()

elo_ranges = all_matches['elo_range'].values
median_elo_range = np.median(elo_ranges)

fig_elo_range_dist = go.Figure()

fig_elo_range_dist.add_trace(go.Histogram(
    x=elo_ranges,
    nbinsx=50,
    marker_color='steelblue',
    opacity=0.8,
    hovertemplate='Rango ELO: %{x:.2f}<br>Frecuencia: %{y}<extra></extra>'
))

fig_elo_range_dist.add_vline(
    x=median_elo_range,
    line_dash="dash",
    line_color="red",
    line_width=2,
```

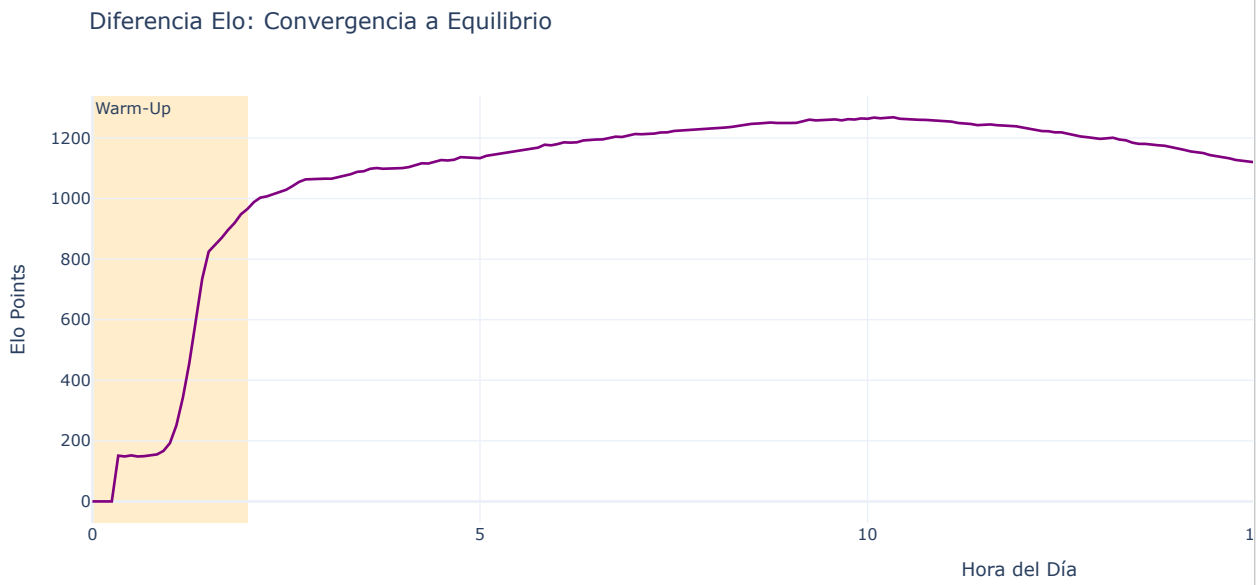
```
    annotation_text=f"Mediana: {median_elo_range:.2f}",
    annotation_position="top right",
    annotation_font_size=12,
    annotation_font_color="red"
)

fig_elo_range_dist.update_layout(
    title='Distribución de Rango ELO en Partidas',
    xaxis_title='Rango ELO (Máx - Mín)',
    yaxis_title='Frecuencia',
    template='plotly_white',
    height=500,
    showlegend=False,
    font=dict(size=11)
)

fig_elo_range_dist.show()
```



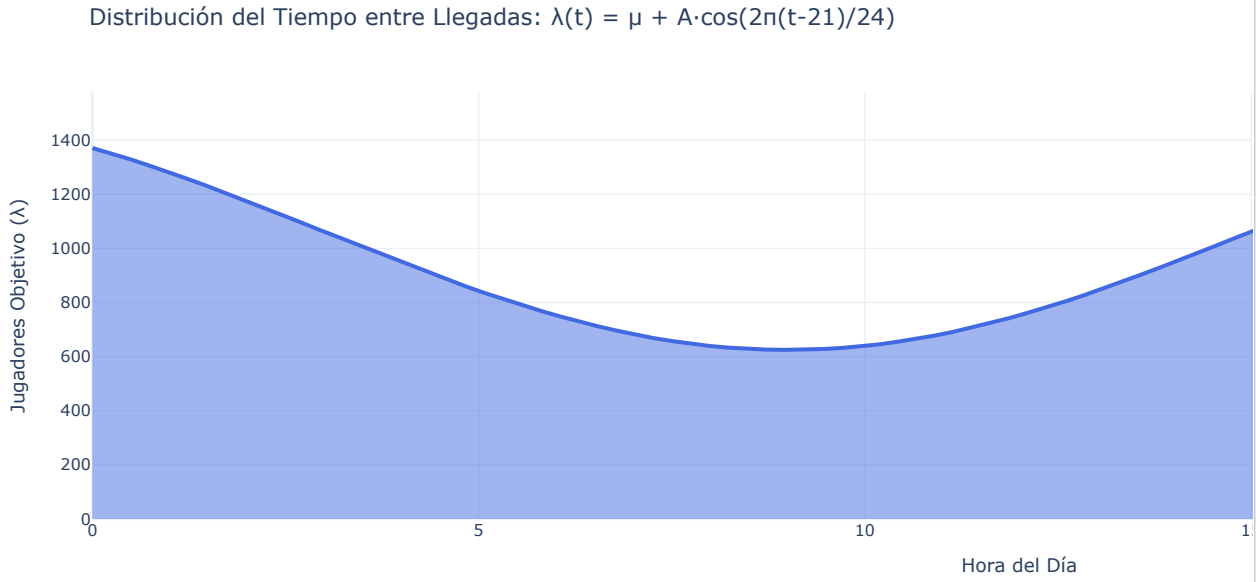

```
fig3 = go.Figure()

fig3.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['avg_elo_difference_mean'],
    mode='lines',
    name='Diferencia Elo',
    line=dict(color='purple', width=2)
))
```

```
fig3.add_vrect(
    x0=0, x1=warmup_end,
    fillcolor="orange", opacity=0.2,
    layer="below", line_width=0,
    annotation_text="Warm-Up",
    annotation_position="top left"
)

fig3.update_layout(
    title='Diferencia Elo: Convergencia a Equilibrio',
    xaxis_title='Hora del Día',
    yaxis_title='Elo Points',
    template='plotly_white',
    hovermode='x unified',
    height=500
)

fig3.show()
```

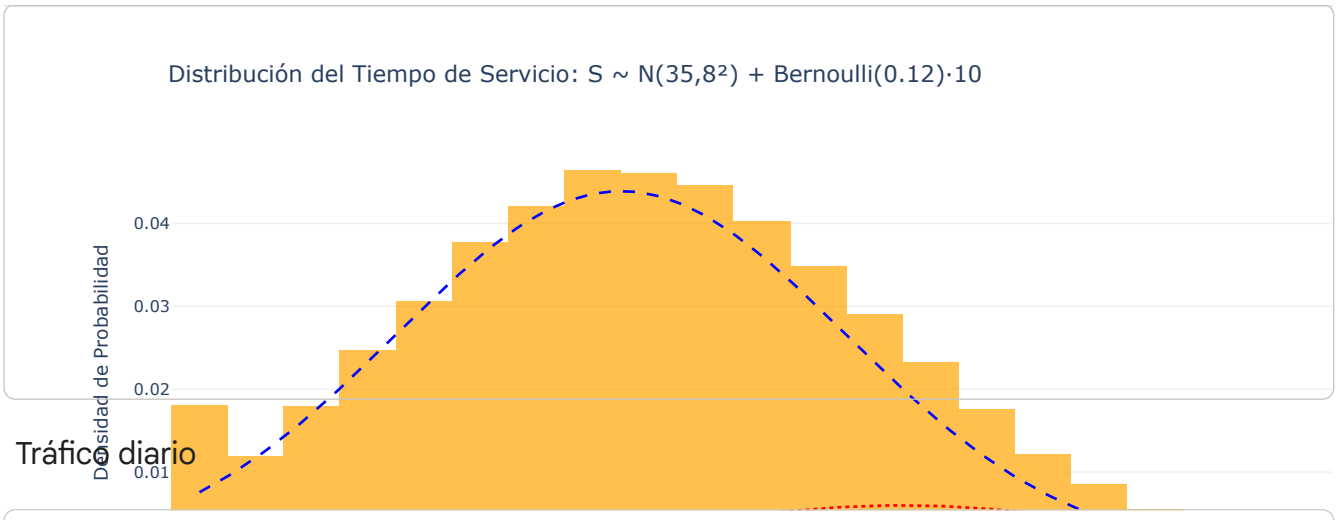



```
raw_metrics_list = [replica['metrics'] for replica in simulation_results['raw_results']]
num_replicas_actual = simulation_results['summary']['n_replicas']
colors = pc.qualitative.Dark24 + pc.qualitative.Light24
```

```
hours = np.linspace(0, 24, 100)
arrival_rates = []
pop_manager = PopulationManager()

for hour in hours:
    arrival_rates.append(pop_manager.get_target_population(hour))

fig_arrivals = go.Figure()
fig_arrivals.add_trace(go.Scatter(
    x=hours,
    y=arrival_rates,
    mode='lines',
    fill='tozeroy',
    line=dict(color='royalblue', width=3),
    name='Tasa de Llegadas'
))

fig_arrivals.add_vline(x=21, line_dash="dash", line_color="red",
                       annotation_text="Hora Pico (21:00)")

fig_arrivals.update_layout(
    title='Distribución del Tiempo entre Llegadas: λ(t) = μ + A·cos(2π(t-21)/24)',
    xaxis_title='Hora del Día',
    yaxis_title='Jugadores Objetivo (λ)',
    template='plotly_white',
    height=500
)
fig_arrivals.show()
```

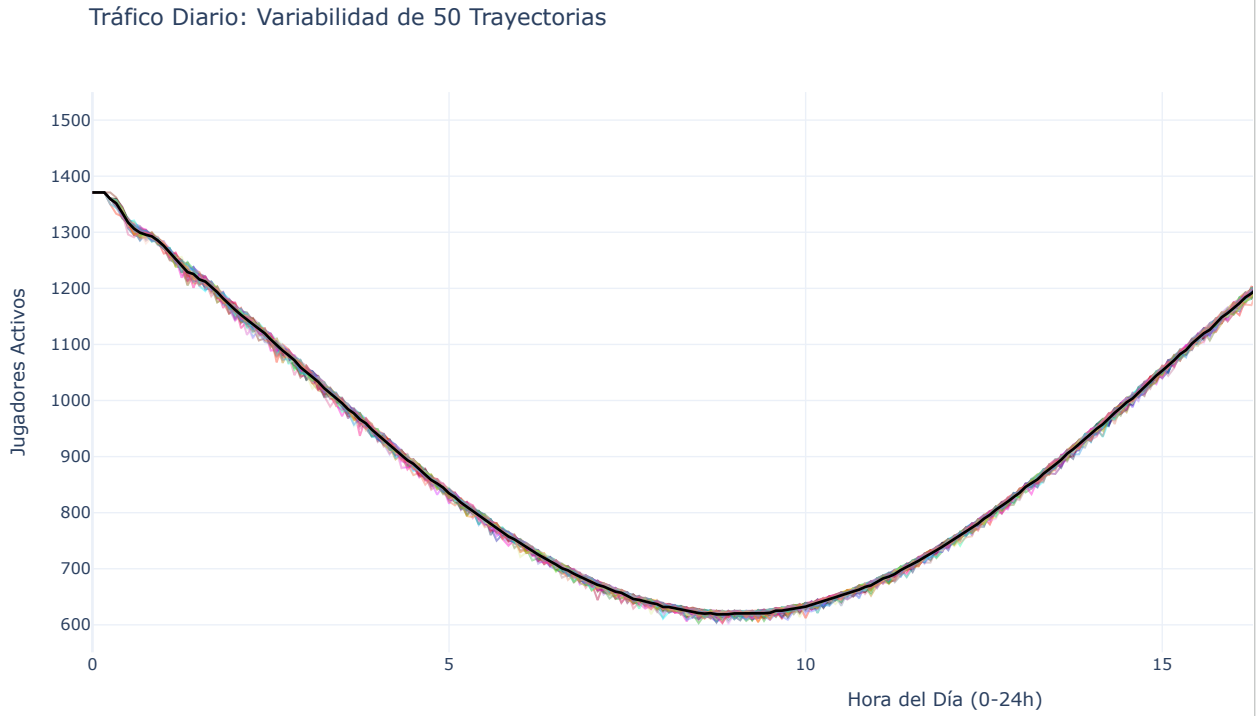



```
match_durations = all_matches['duration'].values

fig_service = go.Figure()

fig_service.add_trace(go.Histogram(
    x=match_durations,
    nbinsx=40,
    marker_color='orange',
    opacity=0.7,
    name='Observado',
    histnorm='probability density'
))

x_theory = np.linspace(20, 60, 200)
y_theory = norm.pdf(x_theory, MATCH_DURATION_MEAN, MATCH_DURATION_STD)

fig_service.add_trace(go.Scatter(
    x=x_theory,
    y=y_theory * 0.88,
    mode='lines',
    line=dict(color='blue', width=2, dash='dash'),
    name=f'Normal({MATCH_DURATION_MEAN}, {MATCH_DURATION_STD}²)'
))

y_theory_ot = norm.pdf(x_theory, MATCH_DURATION_MEAN + OVERTIME_DURATION,
                       MATCH_DURATION_STD)
fig_service.add_trace(go.Scatter(
    x=x_theory,
    y=y_theory_ot * 0.12,
    mode='lines',
    line=dict(color='red', width=2, dash='dot'),
    name=f'Normal({MATCH_DURATION_MEAN + OVERTIME_DURATION}, {MATCH_DURATION_STD}²) + Overtime'
))

fig_service.update_layout(
    title='Distribución del Tiempo de Servicio: S ~ N(35,8²) + Bernoulli(0.12)·10',
    xaxis_title='Duración de Partida (minutos)',
    yaxis_title='Densidad de Probabilidad',
    template='plotly_white',
    height=500
)
fig_service.show()
```



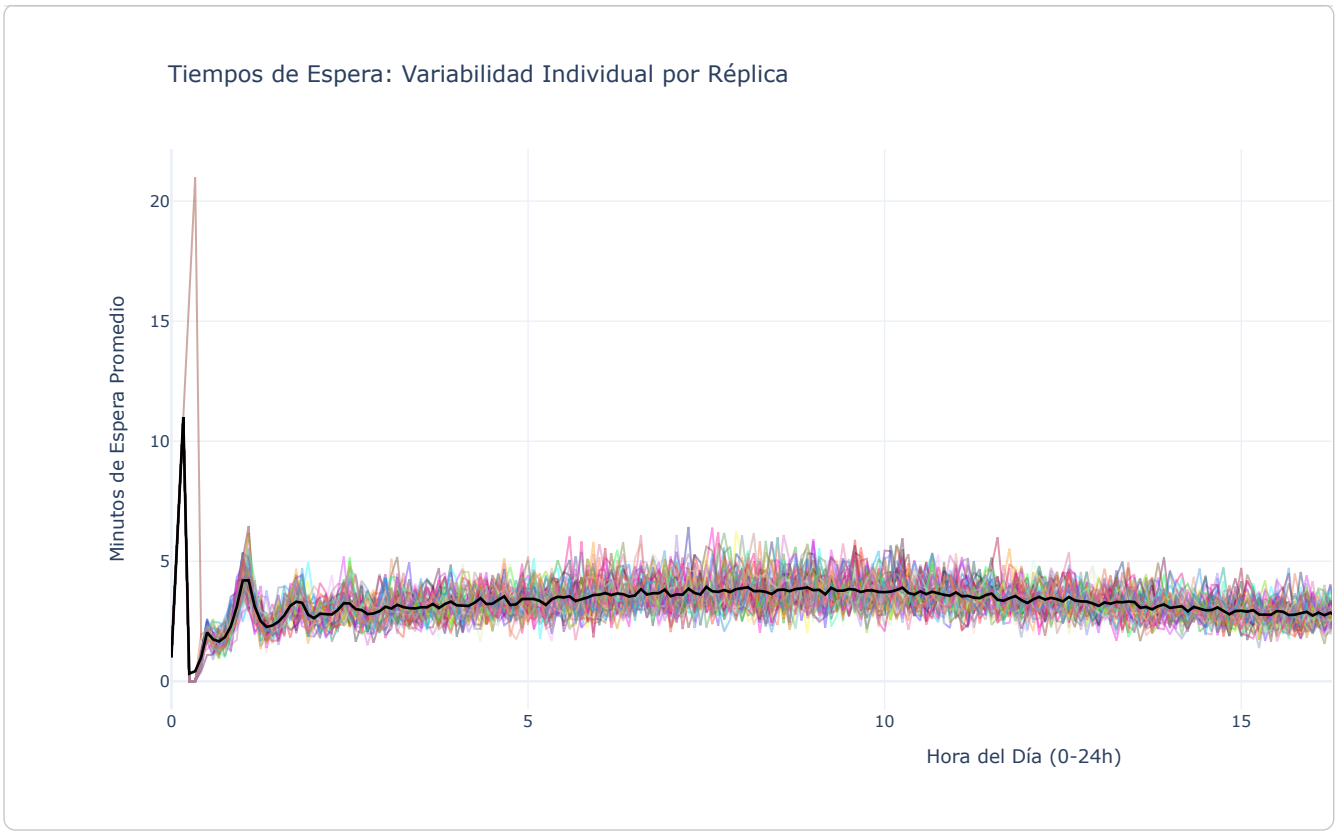

## Tráfico diario

```
col_name = 'active_population'
title = f"Tráfico Diario: Variabilidad de {num_replicas_actual} Trayectorias"
y_label = "Jugadores Activos"

fig_spag_1 = go.Figure()

for i, df in enumerate(raw_metrics_list):
    fig_spag_1.add_trace(go.Scatter(
        x=df['time'],
        y=df[col_name],
        mode='lines',
        line=dict(color=colors[i % len(colors)], width=1.5),
        opacity=0.4,
        name=f'Réplica {i+1}',
        showlegend=False,
        hoverinfo='skip'
    ))

df_mean = pd.concat(raw_metrics_list).groupby('time')[col_name].mean().reset_index()

fig_spag_1.add_trace(go.Scatter(
    x=df_mean['time'],
    y=df_mean[col_name],
    mode='lines',
    line=dict(color='black', width=2),
    name='Promedio General'
))

fig_spag_1.update_layout(
    title=title,
    xaxis_title="Hora del Día (0-24h)",
    yaxis_title=y_label,
    template='plotly_white',
    height=600,
    hovermode="x unified"
)

fig_spag_1.show()
```



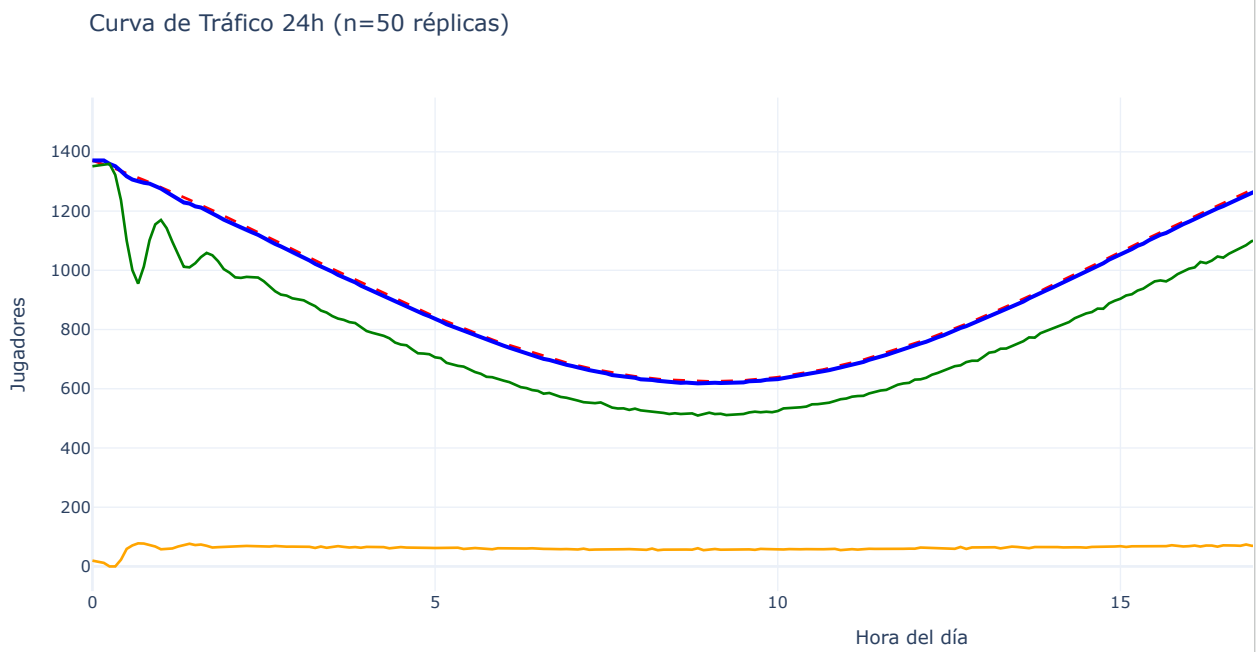

## Tiempos de espera

```
col_name = 'avg_wait_time'
title = f"Tiempos de Espera: Variabilidad Individual por Réplica"
y_label = "Minutos de Espera Promedio"

fig_spag_2 = go.Figure()

for i, df in enumerate(raw_metrics_list):
    fig_spag_2.add_trace(go.Scatter(
        x=df['time'],
        y=df[col_name],
        mode='lines',
        line=dict(color=colors[i % len(colors)], width=1.5),
        opacity=0.4,
        name=f'Réplica {i+1}',
        showlegend=False,
        hoverinfo='skip'
    ))

df_mean = pd.concat(raw_metrics_list).groupby('time')[col_name].mean().reset_index()

fig_spag_2.add_trace(go.Scatter(
    x=df_mean['time'],
    y=df_mean[col_name],
    mode='lines',
    line=dict(color='black', width=2),
    name='Promedio General'
))

fig_spag_2.update_layout(
    title=title,
    xaxis_title="Hora del Día (0-24h)",
    yaxis_title=y_label,
    template='plotly_white',
    height=600,
    hovermode="x unified"
)

fig_spag_2.show()
```



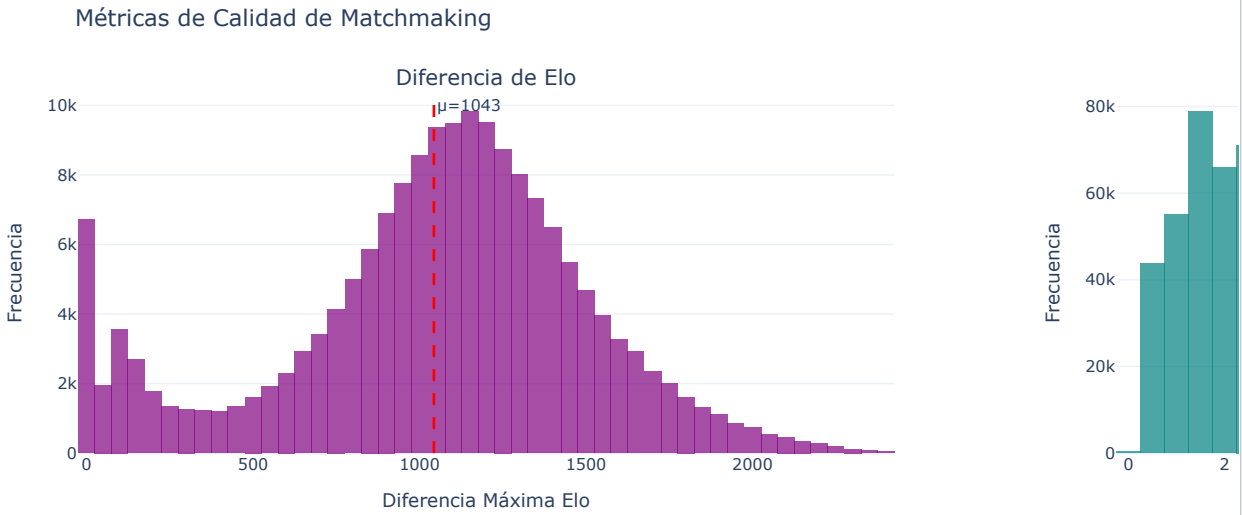

```
fig1 = go.Figure()

fig1.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['target_population_mean'],
    mode='lines',
    name='Población Objetivo',
    line=dict(color='red', dash='dash', width=2)
))

fig1.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['active_population_mean'] + agg['active_population_std'],
    mode='lines',
    line=dict(width=0),
    showlegend=False,
    hoverinfo='skip'
))

fig1.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['active_population_mean'] - agg['active_population_std'],
    mode='lines',
    line=dict(width=0),
    fillcolor='rgba(68, 138, 255, 0.2)',
    fill='tonexty',
    name='±1 STD',
    hoverinfo='skip'
))

fig1.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['active_population_mean'],
    mode='lines',
    name='Población Real (promedio)',
    line=dict(color='blue', width=3)
))

fig1.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['searching_mean'],
    mode='lines',
    name='Buscando',
    line=dict(color='orange', width=2)
))

fig1.add_trace(go.Scatter(
    x=agg['time'],
    y=agg['playing_mean'],
```

```
    mode='lines',
    name='Jugando',
    line=dict(color='green', width=2)
))

fig1.update_layout(
    title=f'Curva de Tráfico 24h (n={simulation_results["summary"]["n_replicas"]} réplicas)',
    xaxis_title='Hora del día',
    yaxis_title='Jugadores',
    template='plotly_white',
    height=550,
    hovermode='x unified'
)

fig1.show()
```




```
fig2 = make_subplots(
    rows=1, cols=2,
    subplot_titles=('Diferencia de Elo', 'Tiempo de Espera Promedio'),
    horizontal_spacing=0.12
)

fig2.add_trace(
    go.Histogram(
        x=all_matches['elo_range'],
        nbinsx=50,
        marker=dict(color='purple', opacity=0.7),
        name='Elo Range'
    ),
    row=1, col=1
)

mean_elo = all_matches['elo_range'].mean()
fig2.add_vline(
    x=mean_elo,
    line_dash="dash",
    line_color="red",
    annotation_text=f"μ={mean_elo:.0f}",
    row=1, col=1
)

fig2.add_trace(
    go.Histogram(
        x=all_players[all_players['matches_played'] >= 2]['avg_wait_time'],
        nbinsx=50,
        marker=dict(color='teal', opacity=0.7),
        name='Wait Time'
    ),
    row=1, col=2
)

mean_wait = all_players[all_players['matches_played'] >= 2]['avg_wait_time'].mean()
```

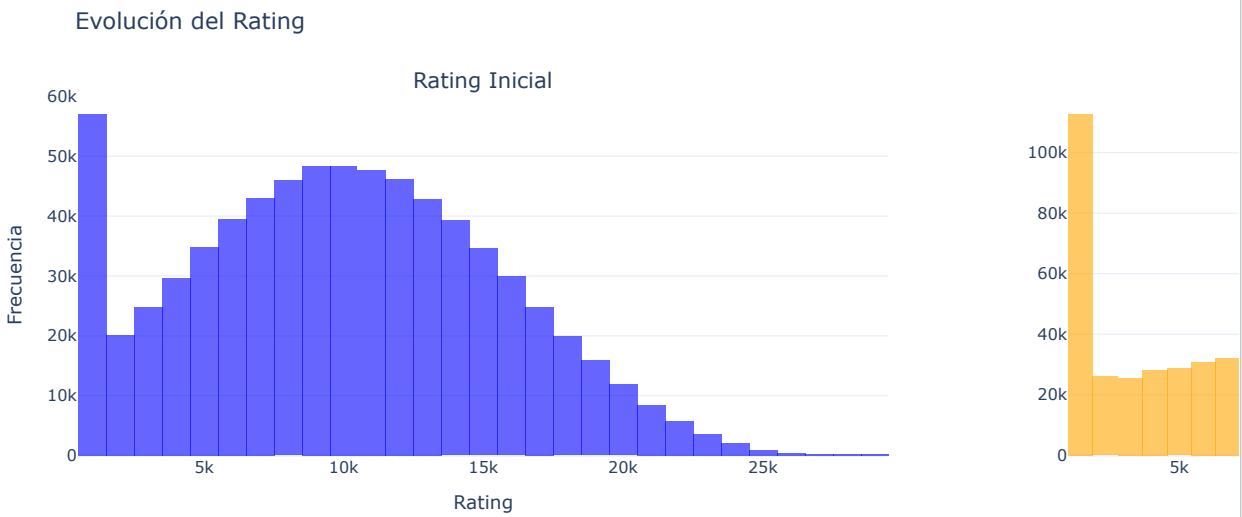
```
fig2.add_vline(
    x=mean_wait,
    line_dash="dash",
    line_color="red",
    annotation_text=f"μ={mean_wait:.2f}min",
    row=1, col=2
)

fig2.update_xaxes(title_text="Diferencia Máxima Elo", row=1, col=1)
fig2.update_xaxes(title_text="Minutos", row=1, col=2)
fig2.update_yaxes(title_text="Frecuencia", row=1, col=1)
fig2.update_yaxes(title_text="Frecuencia", row=1, col=2)

fig2.update_layout(
    title_text='Métricas de Calidad de Matchmaking',
    template='plotly_white',
    height=450,
    showlegend=False
)

fig2.show()

print(f"Elo Range: μ={mean_elo:.0f}, 90%ile={all_matches['elo_range'].quantile(0.9):.0f}")
print(f"Wait Time: μ={mean_wait:.2f}min, 90%ile={all_players[all_players['matches_played']>=2]['avg_wait_time'].quantile(0.
```




```
Elo Range: μ=1043, 90%ile=1584
Wait Time: μ=2.70min, 90%ile=5.00min
```

```
fig3 = make_subplots(
    rows=1, cols=2,
    subplot_titles=('Rating Inicial', 'Rating Final')
)

fig3.add_trace(
    go.Histogram(
        x=all_players['initial_rating'],
        nbinsx=40,
        marker=dict(color='blue', opacity=0.6),
        name='Inicial'
    ),
    row=1, col=1
)

fig3.add_trace(
    go.Histogram(
        x=all_players['final_rating'],
        nbinsx=40,
        marker=dict(color='orange', opacity=0.6),
        name='Final'
    ),
    row=1, col=2
)

fig3.update_xaxes(title_text="Rating", row=1, col=1)
fig3.update_xaxes(title_text="Rating", row=1, col=2)
fig3.update_yaxes(title_text="Frecuencia", row=1, col=1)
```

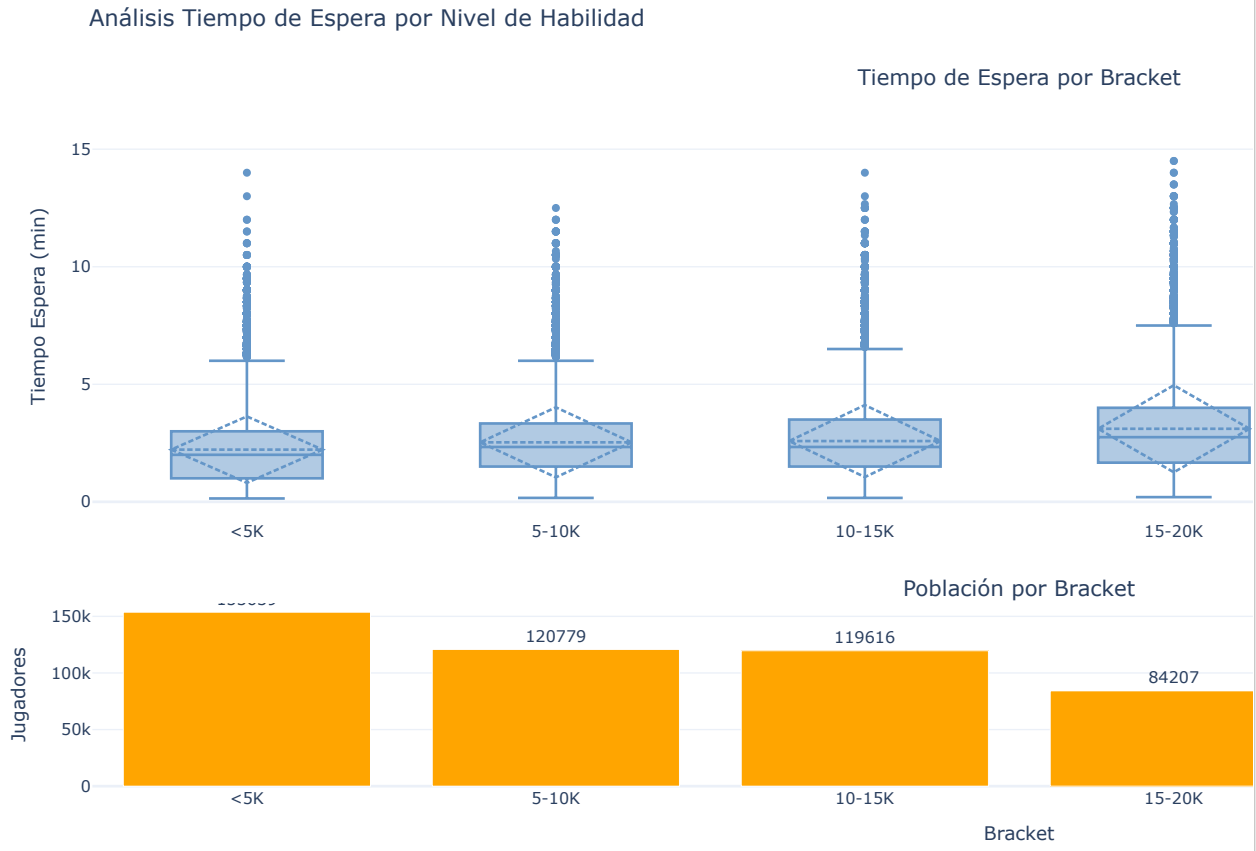
```
fig3.update_layout(
    title_text='Evolución del Rating',
    template='plotly_white',
    height=450
)

fig3.show()

print(f"Cambio promedio: {all_players['rating_change'].mean():+.1f}")
print(f"STD cambio: {all_players['rating_change'].std():.1f}")
```




```
Cambio promedio: +415.6
STD cambio: 4197.3
```

```
df_filtered = all_players[all_players['matches_played'] >= 2].copy()
df_filtered['rating_bracket'] = pd.cut(
    df_filtered['final_rating'],
    bins=[0, 5000, 10000, 15000, 20000, 25000, 40000],
    labels=['<5K', '5-10K', '10-15K', '15-20K', '20-25K', '>25K']
)

fig4 = make_subplots(
    rows=2, cols=1,
    row_heights=[0.7, 0.3],
    subplot_titles=('Tiempo de Espera por Bracket', 'Población por Bracket'),
    vertical_spacing=0.12
)

for bracket in df_filtered['rating_bracket'].cat.categories:
    data = df_filtered[df_filtered['rating_bracket'] == bracket]['avg_wait_time']
    fig4.add_trace(
        go.Box(
            y=data,
            name=str(bracket),
            boxmean='sd',
            marker_color='rgb(100,150,200)'
        ),
        row=1, col=1
    )

bracket_counts = df_filtered['rating_bracket'].value_counts().sort_index()
fig4.add_trace(
    go.Bar(
        x=bracket_counts.index.astype(str),
        y=bracket_counts.values,
        marker_color='orange',
        text=bracket_counts.values,
        textposition='outside',
        showlegend=False
    ),
    row=2, col=1
)

fig4.update_yaxes(title_text="Tiempo Espera (min)", row=1, col=1)
fig4.update_yaxes(title_text="Jugadores", row=2, col=1)
fig4.update_xaxes(title_text="Bracket", row=2, col=1)
```

```
fig4.update_layout(
    template='plotly_white',
    height=700,
    showlegend=False,
    title_text='Análisis Tiempo de Espera por Nivel de Habilidad'
)

fig4.show()

print("\n📊 Tiempo de espera por bracket:")
for bracket in df_filtered['rating_bracket'].cat.categories:
    subset = df_filtered[df_filtered['rating_bracket'] == bracket]
    if len(subset) > 0:
        print(f"  {bracket}: μ={subset['avg_wait_time'].mean():.2f}min, n={len(subset)}")
```



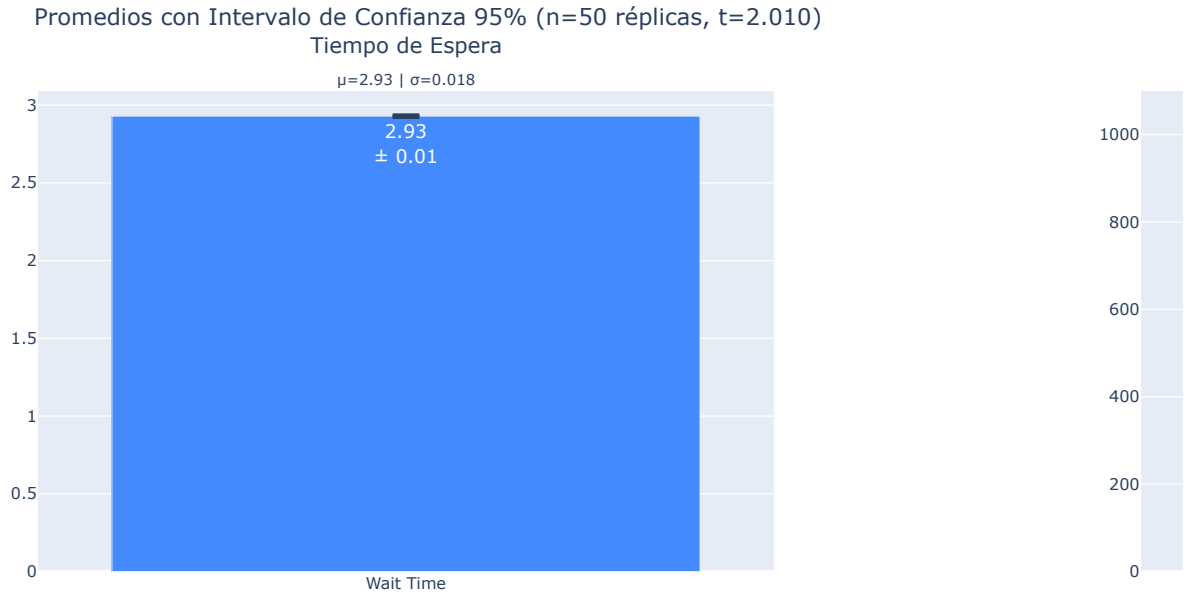

```
📊 Tiempo de espera por bracket:
  <5K: μ=2.22min, n=153639
  5-10K: μ=2.53min, n=120779
  10-15K: μ=2.59min, n=119616
  15-20K: μ=3.11min, n=84207
  20-25K: μ=3.99min, n=38518
  >25K: μ=4.99min, n=9523
```

## Intervalo de confianza

```
replica_wait_times = all_players.groupby('replica')['avg_wait_time'].mean()
replica_elo_ranges = all_matches.groupby('replica')['elo_range'].mean()

r = len(replica_wait_times)
alpha = 0.05
t_crit = stats.t.ppf(1 - alpha/2, r - 1)

# Tiempo de Espera
wait_mean = replica_wait_times.mean()
wait_std = replica_wait_times.std(ddof=1)
wait_se = wait_std / np.sqrt(r)
wait_margin = t_crit * wait_se
wait_ci = [wait_mean - wait_margin, wait_mean + wait_margin]

# Balance ELO
elo_mean = replica_elo_ranges.mean()
elo_std = replica_elo_ranges.std(ddof=1)
```

```
elo_se = elo_std / np.sqrt(r)
elo_margin = t_crit * elo_se
elo_ci = [elo_mean - elo_margin, elo_mean + elo_margin]

fig = make_subplots(
    rows=1, cols=2,
    subplot_titles=(
        f"Tiempo de Espera<br><sub>μ={wait_mean:.2f} | σ={wait_std:.3f}</sub>",
        f"Balance ELO<br><sub>μ={elo_mean:.0f} | σ={elo_std:.2f}</sub>"
    ),
    horizontal_spacing=0.2
)

fig.add_trace(go.Bar(
    x=['Wait Time'],
    y=[wait_mean],
    error_y=dict(
        type='data',
        array=[wait_margin],
        visible=True,
        thickness=3,
        width=10
    ),
    marker_color='#448aff',
    text=[f'{wait_mean:.2f}<br>± {wait_margin:.2f}'],
    textposition='auto',
    textfont=dict(size=14, color='white'),
    name='Espera',
    hovertemplate=(
        '<b>Tiempo de Espera</b><br>' +
        'Media: %{y:.3f} min<br>' +
        f'Desv. Std: {wait_std:.3f} min<br>' +
        f'Error Std: {wait_se:.3f} min<br>' +
        f'Margen: ±{wait_margin:.3f} min<br>' +
        f'IC 95%: [{wait_ci[0]:.3f}, {wait_ci[1]:.3f}]<br>' +
        '<extra></extra>'
    )
), row=1, col=1)

fig.add_trace(go.Bar(
    x=['ELO Range'],
    y=[elo_mean],
    error_y=dict(
        type='data',
        array=[elo_margin],
        visible=True,
        color='white',
        thickness=3,
        width=10
    ),
    marker_color='#ff8a44',
    text=[f'{elo_mean:.0f}<br>± {elo_margin:.1f}'],
    textposition='auto',
    textfont=dict(size=14, color='white'),
    name='ELO',
    hovertemplate=(
        '<b>Balance ELO</b><br>' +
        'Media: %{y:.1f}<br>' +
        f'Desv. Std: {elo_std:.2f}<br>' +
        f'Error Std: {elo_se:.2f}<br>' +
        f'Margen: ±{elo_margin:.2f}<br>' +
        f'IC 95%: [{elo_ci[0]:.1f}, {elo_ci[1]:.1f}]<br>' +
        '<extra></extra>'
    )
), row=1, col=2)

fig.update_layout(
    title_text=f'Promedios con Intervalo de Confianza 95% (n={r} réplicas, t={t_crit:.3f})',
    showlegend=False,
    height=500,
    margin=dict(t=100, b=40),
    font=dict(size=12)
)

fig.show()

# Reporte Completo
print("="*70)
print(f"ANÁLISIS DE {r} RÉPLICAS - INTERVALO DE CONFIANZA 95%")
print("="*70)
print(f"t-crítico (df={r-1}): {t_crit:.4f}")
```

```
print("\n📊 TIEMPO DE ESPERA PROMEDIO")
print("-"*70)
print(f"  Media (μ):              {wait_mean:.4f} min")
print(f"  Desviación Estándar (σ): {wait_std:.4f} min")
print(f"  Error Estándar (SE):     {wait_se:.4f} min")
print(f"  Margen de Error:         ±{wait_margin:.4f} min")
print(f"  IC 95%:                  [{wait_ci[0]:.4f}, {wait_ci[1]:.4f}] min")
print(f"  Coef. Variación (CV):    {(wait_std/wait_mean)*100:.2f}%")

print("\n⚖️ BALANCE ELO (RANGO PROMEDIO)")
print("-"*70)
print(f"  Media (μ):              {elo_mean:.2f} pts")
print(f"  Desviación Estándar (σ): {elo_std:.2f} pts")
print(f"  Error Estándar (SE):     {elo_se:.2f} pts")
print(f"  Margen de Error:         ±{elo_margin:.2f} pts")
print(f"  IC 95%:                  [{elo_ci[0]:.2f}, {elo_ci[1]:.2f}] pts")
print(f"  Coef. Variación (CV):    {(elo_std/elo_mean)*100:.2f}%")

print("\n" + "="*70)
print("✅ INTERPRETACIÓN")
print("="*70)
print(f"Con 95% de confianza, el tiempo de espera verdadero está entre")
print(f"{wait_ci[0]:.3f} y {wait_ci[1]:.3f} minutos.")
print(f"\nCon 95% de confianza, el balance ELO verdadero está entre")
print(f"{elo_ci[0]:.1f} y {elo_ci[1]:.1f} puntos.")
```




```
======================================================================
ANÁLISIS DE 50 RÉPLICAS - INTERVALO DE CONFIANZA 95%
======================================================================
t-crítico (df=49): 2.0096

📊 TIEMPO DE ESPERA PROMEDIO
----------------------------------------------------------------------
  Media (μ):              2.9293 min
  Desviación Estándar (σ): 0.0177 min
  Error Estándar (SE):     0.0025 min
  Margen de Error:         ±0.0050 min
  IC 95%:                  [2.9243, 2.9344] min
  Coef. Variación (CV):    0.61%

⚖️ BALANCE ELO (RANGO PROMEDIO)
----------------------------------------------------------------------
  Media (μ):              1042.75 pts
  Desviación Estándar (σ): 4.71 pts
  Error Estándar (SE):     0.67 pts
  Margen de Error:         ±1.34 pts
  IC 95%:                  [1041.41, 1044.09] pts
  Coef. Variación (CV):    0.45%

======================================================================
✅ INTERPRETACIÓN
======================================================================
Con 95% de confianza, el tiempo de espera verdadero está entre
2.924 y 2.934 minutos.

Con 95% de confianza, el balance ELO verdadero está entre
1041.4 y 1044.1 puntos.
```

# Parte 2: Análisis y casos de estudio

## 2.1. Motor general de experimentos

```
def run_experiment_simulation(duration_hours: float = 2, team_size: int = 5, glicko_tau: float = 0.5,
                              num_players: int = 100, custom_players: Optional[List[Player]] = None,
                              track_predictions: bool = False, seed: int = 42) -> Dict:

    np.random.seed(seed)
    random.seed(seed)

    glicko_sys = Glicko2System(tau=glicko_tau)
    players = custom_players if custom_players else []
    next_id = max([p.id for p in players]) + 1 if players else 0

    if not custom_players:
        for i in range(num_players):
            hidden = np.clip(np.random.normal(1500, 300), 800, 2200)
            visible = hidden + np.random.normal(0, 100)
            p = Player(id=next_id + i, hidden_skill=hidden, glicko=GlickoRating(rating=visible))
            p.rating_history.append(p.glicko.rating)
            p.rd_history.append(p.glicko.rd)
            players.append(p)

    matches = []
    total_minutes = int(duration_hours * 60)

    for minute in range(total_minutes):
        for p in players:
            if p.state == PlayerState.BUSCANDO:
                p.current_wait_time += 1

        available = [p for p in players if p.state == PlayerState.BUSCANDO]
        while len(available) >= team_size * 2:
            available.sort(key=lambda p: p.glicko.rating)
            selected = available[:team_size * 2]
            team_a, team_b = selected[:team_size], selected[team_size:]

            visible_a = [p.glicko.rating for p in team_a]
            visible_b = [p.glicko.rating for p in team_b]
            win_prob_a = glicko_sys.predict_win_probability(visible_a, visible_b)
            predicted = "team_a" if win_prob_a > 0.5 else "team_b"

            skill_a = np.mean([p.hidden_skill for p in team_a])
            skill_b = np.mean([p.hidden_skill for p in team_b])
            actual = "team_a" if np.random.random() < (1 / (1 + 10**((skill_b - skill_a) / 400))) else "team_b"

            team_a_won = (actual == "team_a")
            for p in team_a:
                p.glicko = glicko_sys.update_rating(p.glicko, [x.glicko for x in team_b], [1.0 if team_a_won else 0.0] * te
                p.rating_history.append(p.glicko.rating)
                p.rd_history.append(p.glicko.rd)
                p.matches_played += 1
                p.total_wait_time += p.current_wait_time
                p.wait_times_history.append(p.current_wait_time)
                p.current_wait_time = 0
                p.state = PlayerState.JUGANDO

            for p in team_b:
                p.glicko = glicko_sys.update_rating(p.glicko, [x.glicko for x in team_a], [0.0 if team_a_won else 1.0] * te
                p.rating_history.append(p.glicko.rating)
                p.rd_history.append(p.glicko.rd)
                p.matches_played += 1
                p.total_wait_time += p.current_wait_time
                p.wait_times_history.append(p.current_wait_time)
                p.current_wait_time = 0
                p.state = PlayerState.JUGANDO

            match = Match(
                id=len(matches),
                players=team_a + team_b,
                start_time=minute,
                duration=35,
                avg_elo=np.mean(visible_a + visible_b),
                elo_range=max(visible_a + visible_b) - min(visible_a + visible_b),
                predicted_winner=predicted,
                actual_winner=actual,
                win_probability_a=win_prob_a
```

```
                )
                matches.append(match)
                available = [p for p in available if p not in selected]

            for p in players:
                if p.state == PlayerState.JUGANDO and np.random.random() < 0.03:
                    p.state = PlayerState.BUSCANDO

        return {'players': players, 'matches': matches}
```

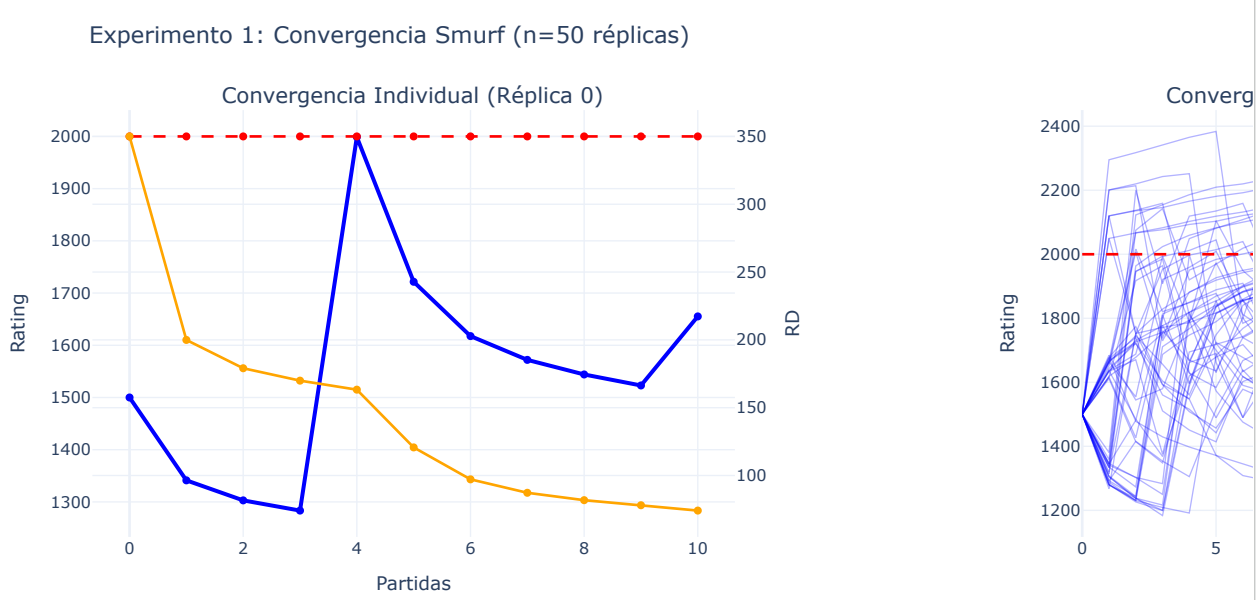
## 2.2. Experimento 1: Convergencia Smurf

```
def run_smurf_experiment(n_replicas: int = 10, duration_hours: float = 10):
    results = []

    for replica in range(n_replicas):
        normal_players = [
            Player(id=i, hidden_skill=np.clip(np.random.normal(1500, 300), 800, 2200), glicko=GlickoRating())
            for i in range(50)
        ]
        for p in normal_players:
            p.rating_history.append(p.glicko.rating)
            p.rd_history.append(p.glicko.rd)

        smurf = Player(id=999, hidden_skill=2000, glicko=GlickoRating(rating=1500))
        smurf.rating_history.append(1500)
        smurf.rd_history.append(350)

        result = run_experiment_simulation(
            duration_hours=duration_hours,
            team_size=5,
            custom_players=normal_players + [smurf],
            seed=42 + replica
        )

        smurf_updated = [p for p in result['players'] if p.id == 999][0]
        results.append({
            'replica': replica,
            'matches_played': smurf_updated.matches_played,
            'initial_rating': smurf_updated.rating_history[0],
            'final_rating': smurf_updated.rating_history[-1],
            'initial_rd': smurf_updated.rd_history[0],
            'final_rd': smurf_updated.rd_history[-1],
            'rating_history': smurf_updated.rating_history,
            'rd_history': smurf_updated.rd_history,
            'hidden_skill': smurf_updated.hidden_skill
        })

    return results

smurf_results = run_smurf_experiment(n_replicas=N_REPLICAS)

avg_matches = np.mean([r['matches_played'] for r in smurf_results])
avg_final_rating = np.mean([r['final_rating'] for r in smurf_results])
avg_final_rd = np.mean([r['final_rd'] for r in smurf_results])

print(f"Promedio de partidas: {avg_matches:.1f}")
print(f"Rating inicial: 1500 → Final: {avg_final_rating:.0f}")
print(f"RD inicial: 350 → Final: {avg_final_rd:.0f}")
```

```
Promedio de partidas: 16.3
Rating inicial: 1500 → Final: 1814
RD inicial: 350 → Final: 60
```

```
fig_exp1 = make_subplots(
    rows=1, cols=2,
    specs=[[{"secondary_y": True}, {"secondary_y": False}]],
    subplot_titles=('Convergencia Individual (Réplica 0)', 'Convergencia Agregada (Todas las Réplicas)')
)

replica_0 = smurf_results[0]
matches_range = list(range(len(replica_0['rating_history'])))

fig_exp1.add_trace(
    go.Scatter(
        x=matches_range,
        y=replica_0['rating_history'],
        name="Visible Rating",
        line=dict(color="blue", width=3)
```

```
    ),
    row=1, col=1,
    secondary_y=False
)

fig_exp1.add_trace(
    go.Scatter(
        x=matches_range,
        y=[replica_0['hidden_skill']] * len(matches_range),
        name="Hidden Skill",
        line=dict(color="red", dash="dash", width=2)
    ),
    row=1, col=1,
    secondary_y=False
)

fig_exp1.add_trace(
    go.Scatter(
        x=matches_range,
        y=replica_0['rd_history'],
        name="RD",
        line=dict(color="orange", width=2)
    ),
    row=1, col=1,
    secondary_y=True
)

max_length = max(len(r['rating_history']) for r in smurf_results)
for replica in smurf_results:
    fig_exp1.add_trace(
        go.Scatter(
            x=list(range(len(replica['rating_history']))),
            y=replica['rating_history'],
            mode='lines',
            line=dict(color='blue', width=1),
            opacity=0.3,
            showlegend=False,
            hovertemplate='Réplica %{text}<br>Rating: %{y:.0f}<extra></extra>',
            text=[replica['replica']] * len(replica['rating_history'])
        ),
        row=1, col=2
    )

fig_exp1.add_trace(
    go.Scatter(
        x=[0, max_length],
        y=[2000, 2000],
        mode='lines',
        line=dict(color='red', dash='dash', width=2),
        name='Target (2000)',
        showlegend=True
    ),
    row=1, col=2
)

fig_exp1.update_xaxes(title_text="Partidas", row=1, col=1)
fig_exp1.update_xaxes(title_text="Partidas", row=1, col=2)
fig_exp1.update_yaxes(title_text="Rating", row=1, col=1, secondary_y=False)
fig_exp1.update_yaxes(title_text="RD", row=1, col=1, secondary_y=True)
fig_exp1.update_yaxes(title_text="Rating", row=1, col=2)

fig_exp1.update_layout(
    title=f"Experimento 1: Convergencia Smurf (n={len(smurf_results)} réplicas)",
    template="plotly_white",
    height=500,
    hovermode='closest'
)

fig_exp1.show()
```



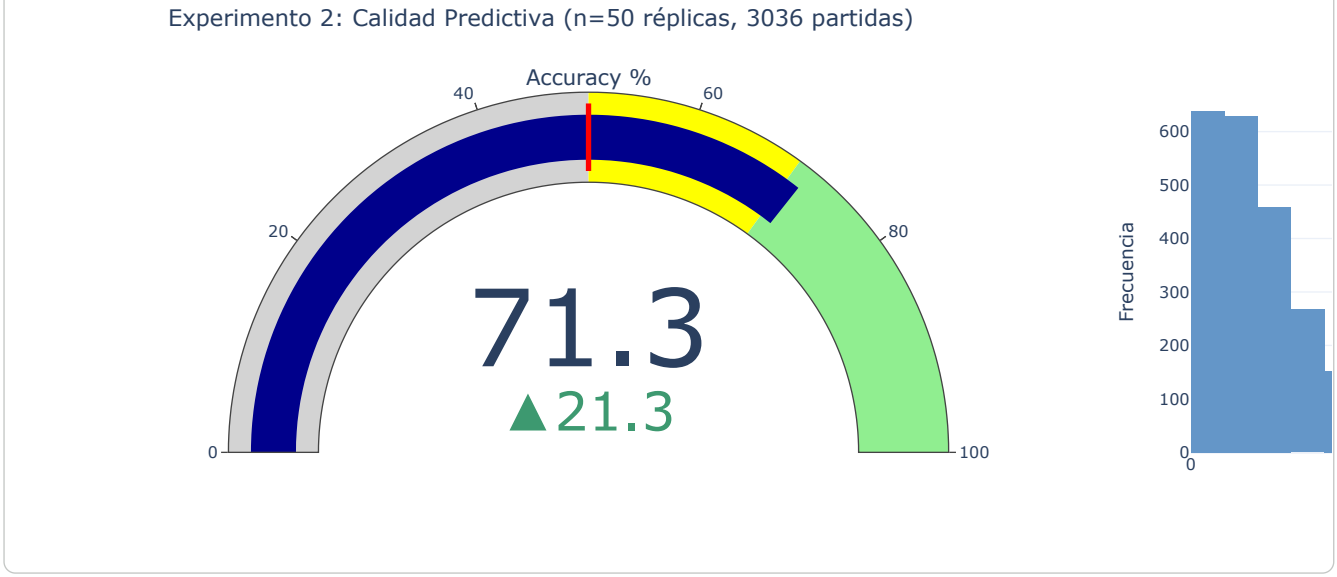

## 2.3. Experimento 2: Calidad Predictiva de Glicko-2

```
def run_prediction_experiment(n_replicas: int = 10, duration_hours: float = 3, num_players: int = 100):
    all_results = []

    for replica in range(n_replicas):
        result = run_experiment_simulation(
            duration_hours=duration_hours,
            num_players=num_players,
            track_predictions=True,
            seed=42 + replica
        )

        matches = result['matches']
        predictions_correct = sum(1 for m in matches if m.prediction_correct)
        accuracy = predictions_correct / len(matches) if matches else 0
        brier_scores = [(m.win_probability_a - (1.0 if m.actual_winner == "team_a" else 0.0))**2 for m in matches]
        brier = np.mean(brier_scores)

        all_results.append({
            'replica': replica,
            'total_matches': len(matches),
            'accuracy': accuracy,
            'brier_score': brier,
            'matches': matches
        })

    return all_results

prediction_results = run_prediction_experiment(n_replicas=N_REPLICAS)

avg_accuracy = np.mean([r['accuracy'] for r in prediction_results])
avg_brier = np.mean([r['brier_score'] for r in prediction_results])
total_matches = sum(r['total_matches'] for r in prediction_results)

print(f"Total partidas: {total_matches}")
print(f"Accuracy promedio: {avg_accuracy*100:.2f}%")
print(f"Brier Score promedio: {avg_brier:.4f}")
```

```
Total partidas: 3036
Accuracy promedio: 71.35%
Brier Score promedio: 0.2339
```

```
all_probs = []
for r in prediction_results:
    all_probs.extend([m.win_probability_a for m in r['matches']])

fig_exp2 = make_subplots(
    rows=1, cols=2,
    specs=[[{"type": "indicator"}, {"type": "histogram"}]],
    subplot_titles=("Accuracy %", "Distribución de Probabilidades Predichas")
)
```

```
fig_exp2.add_trace(
    go.Indicator(
        mode="gauge+number+delta",
        value=avg_accuracy * 100,
        #title={"text": "Accuracy (%)"},
        delta={"reference": 50},
        gauge={
            "axis": {"range": [0, 100]},
            "bar": {"color": "darkblue"},
            "steps": [
                {"range": [0, 50], "color": "lightgray"},
                {"range": [50, 70], "color": "yellow"},
                {"range": [70, 100], "color": "lightgreen"}
            ],
            "threshold": {
                "line": {"color": "red", "width": 4},
                "thickness": 0.75,
                "value": 50
            }
        }
    ),
    row=1, col=1
)

fig_exp2.add_trace(
    go.Histogram(
        x=all_probs,
        nbinsx=25,
        marker_color="rgb(100,150,200)",
        name="Probabilidades"
    ),
    row=1, col=2
)

fig_exp2.update_xaxes(title_text="P(Team A gana)", row=1, col=2)
fig_exp2.update_yaxes(title_text="Frecuencia", row=1, col=2)

fig_exp2.update_layout(
    title=f"Experimento 2: Calidad Predictiva (n={len(prediction_results)} réplicas, {total_matches} partidas)",
    template="plotly_white",
    height=450,
    showlegend=False
)

fig_exp2.show()
```




## 2.4. Experimento 3: Sensibilidad de tamaños de equipo

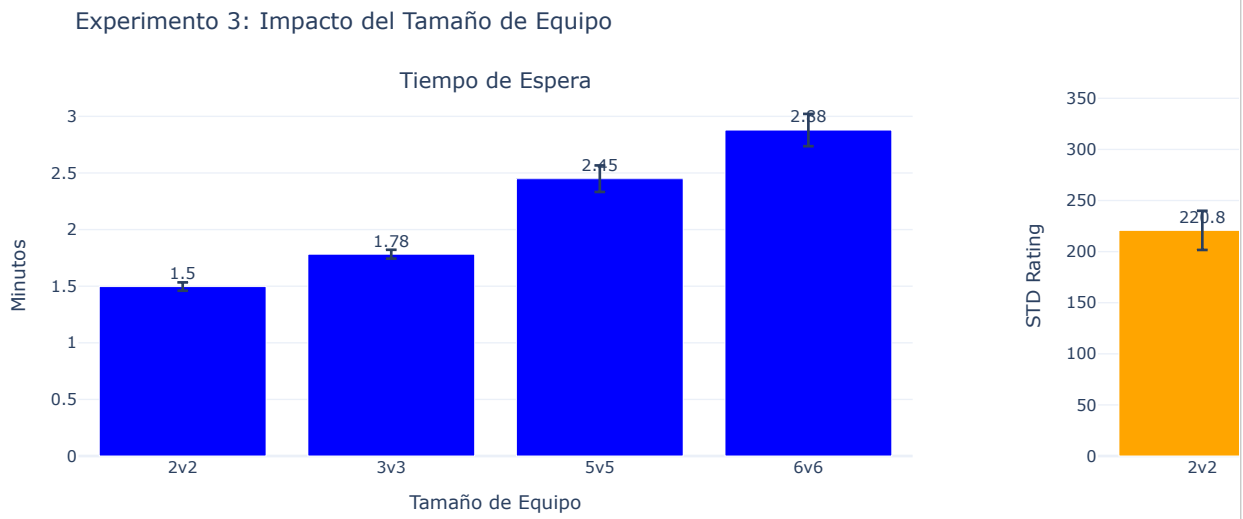
```
def run_team_size_experiment(team_sizes: List[int], n_replicas: int = 5, duration_hours: float = 2):
    results_by_size = []

    for size in team_sizes:
        size_results = []
```

```
        for replica in range(n_replicas):
            result = run_experiment_simulation(
                duration_hours=duration_hours,
                team_size=size,
                num_players=80,
                seed=42 + replica
            )

            players = [p for p in result['players'] if p.matches_played > 0]
            matches = result['matches']

            avg_wait = np.mean([p.total_wait_time / p.matches_played for p in players]) if players else 0
            fairness = np.mean([np.std([p.glicko.rating for p in m.players]) for m in matches]) if matches else 0

            size_results.append({
                'replica': replica,
                'team_size': size,
                'matches': len(matches),
                'avg_wait': avg_wait,
                'fairness_std': fairness
            })

        avg_matches = np.mean([r['matches'] for r in size_results])
        avg_wait_time = np.mean([r['avg_wait'] for r in size_results])
        avg_fairness = np.mean([r['fairness_std'] for r in size_results])
        std_wait_time = np.std([r['avg_wait'] for r in size_results])
        std_fairness = np.std([r['fairness_std'] for r in size_results])

        results_by_size.append({
            'team_size': size,
            'avg_matches': avg_matches,
            'avg_wait': avg_wait_time,
            'std_wait': std_wait_time,
            'avg_fairness': avg_fairness,
            'std_fairness': std_fairness
        })

        print(f"{size}v{size}: {avg_matches:.0f} partidas, wait={avg_wait_time:.2f}min, fairness={avg_fairness:.1f}")

    return pd.DataFrame(results_by_size)

team_sizes = [2, 3, 5, 6]
df_size = run_team_size_experiment(team_sizes, n_replicas=5)
```

```
2v2: 90 partidas, wait=1.50min, fairness=220.8
3v3: 59 partidas, wait=1.78min, fairness=293.3
5v5: 34 partidas, wait=2.45min, fairness=282.5
6v6: 28 partidas, wait=2.88min, fairness=291.3
```

```
fig_exp3 = make_subplots(
    rows=1, cols=2,
    subplot_titles=("Tiempo de Espera", "Fairness (STD Rating)")
)

labels = [f"{s}v{s}" for s in df_size['team_size']]

fig_exp3.add_trace(
    go.Bar(
        x=labels,
        y=df_size['avg_wait'],
        error_y=dict(type='data', array=df_size['std_wait']),
        marker_color='blue',
        text=df_size['avg_wait'].round(2),
        textposition='outside',
        name='Wait Time'
    ),
    row=1, col=1
)

fig_exp3.add_trace(
    go.Bar(
        x=labels,
        y=df_size['avg_fairness'],
        error_y=dict(type='data', array=df_size['std_fairness']),
        marker_color='orange',
        text=df_size['avg_fairness'].round(1),
        textposition='outside',
        name='Fairness'
    ),
    row=1, col=2
)
```

```
fig_exp3.update_yaxes(title_text="Minutos", row=1, col=1)
fig_exp3.update_yaxes(title_text="STD Rating", row=1, col=2)
fig_exp3.update_xaxes(title_text="Tamaño de Equipo", row=1, col=1)
fig_exp3.update_xaxes(title_text="Tamaño de Equipo", row=1, col=2)

fig_exp3.update_layout(
    title="Experimento 3: Impacto del Tamaño de Equipo",
    template="plotly_white",
    height=450,
    showlegend=False
)

fig_exp3.show()
```



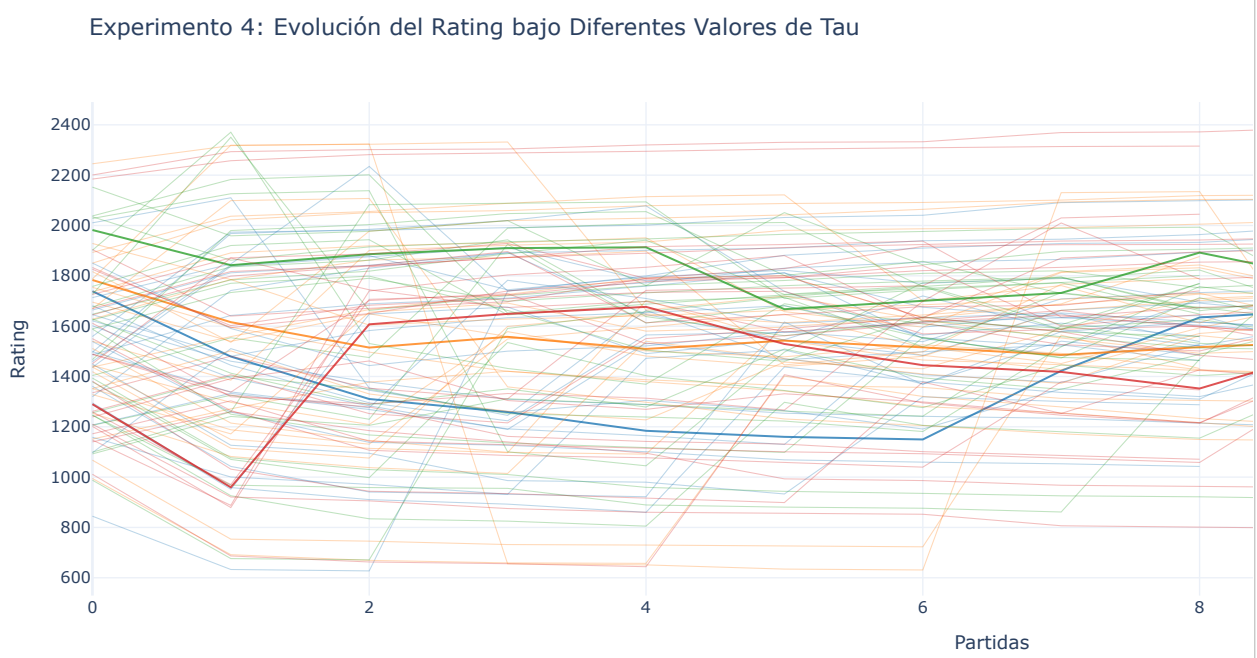

## 2.5. Experimento 4: Sensibilidad de Tau

```
def run_tau_experiment(tau_values: List[float], n_replicas: int = 5, duration_hours: float = 3):
    tracked_by_tau = {}

    for tau_idx, tau in enumerate(tau_values):
        tau_players = []

        for replica in range(n_replicas):
            unique_seed = 42 + (tau_idx * 100) + replica

            result = run_experiment_simulation(
                duration_hours=duration_hours,
                glicko_tau=tau,
                num_players=50,
                seed=unique_seed
            )

            top5 = sorted(result['players'], key=lambda p: p.matches_played, reverse=True)[:5]
            tau_players.extend(top5)

        tracked_by_tau[tau] = tau_players

        avg_matches = np.mean([p.matches_played for p in tau_players])
        avg_volatility = np.mean([np.std(np.diff(p.rating_history)) for p in tau_players if len(p.rating_history) > 1])

        print(f"τ={tau}: {avg_matches:.1f} partidas, volatilidad={avg_volatility:.2f}")

    return tracked_by_tau

tau_values = [0.3, 0.5, 0.8, 1.2]
tracked_by_tau = run_tau_experiment(tau_values, n_replicas=5)
```

```
τ=0.3: 9.1 partidas, volatilidad=173.98
τ=0.5: 9.6 partidas, volatilidad=186.47
τ=0.8: 9.4 partidas, volatilidad=197.04
τ=1.2: 9.5 partidas, volatilidad=145.55
```

```
fig_exp4 = go.Figure()
colors = ['rgb(31,119,180)', 'rgb(255,127,14)', 'rgb(44,160,44)', 'rgb(214,39,40)']

for idx, tau in enumerate(tau_values):
```

```
for idx, tau in enumerate(tau_values):
    for p_idx, player in enumerate(tracked_by_tau[tau]):
        fig_exp4.add_trace(
            go.Scatter(
                x=list(range(len(player.rating_history))),
                y=player.rating_history,
                mode='lines',
                line=dict(color=colors[idx], width=1.5 if p_idx == 0 else 0.8),
                opacity=0.8 if p_idx == 0 else 0.3,
                name=f"τ={tau}" if p_idx == 0 else None,
                legendgroup=f"tau_{tau}",
                showlegend=(p_idx == 0),
                hovertemplate=f'τ={tau}<br>Partida: %{{x}}<br>Rating: %{{y:.0f}}<extra></extra>'
            )
        )

fig_exp4.update_layout(
    title="Experimento 4: Evolución del Rating bajo Diferentes Valores de Tau",
    xaxis_title="Partidas",
    yaxis_title="Rating",
    template="plotly_white",
    height=550,
    hovermode='closest'
)

fig_exp4.show()

print("\n📊 Volatilidad promedio por Tau:")
for tau in tau_values:
    vols = [np.std(np.diff(p.rating_history)) for p in tracked_by_tau[tau] if len(p.rating_history) > 1]
    print(f"  τ={tau}: {np.mean(vols):.2f} ± {np.std(vols):.2f}")
```

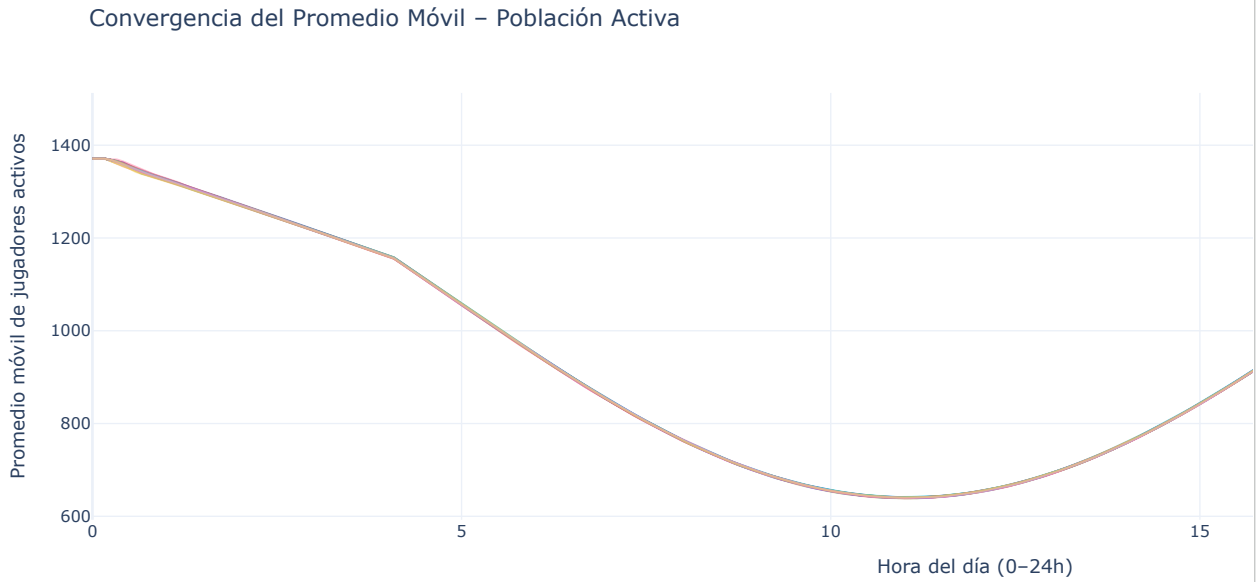



```
📊 Volatilidad promedio por Tau:
  τ=0.3: 173.98 ± 73.13
  τ=0.5: 186.47 ± 142.71
  τ=0.8: 197.04 ± 86.64
  τ=1.2: 145.55 ± 84.19
```

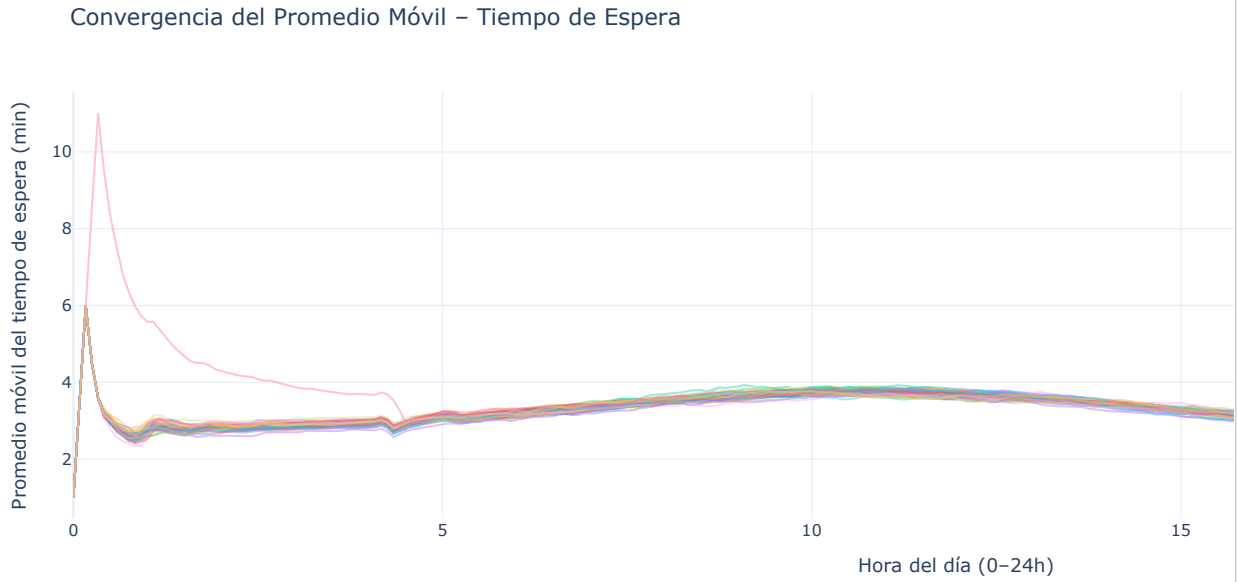
## ET

```
raw_metrics_list = [replica['metrics'] for replica in simulation_results['raw_results']]
num_replicas_actual = simulation_results['summary']['n_replicas']


window = 50

fig_conv_1 = go.Figure()

for i, df in enumerate(raw_metrics_list):
    df_sorted = df.sort_values("time")
    df_sorted["moving_avg"] = df_sorted["active_population"].rolling(window,min_periods=1).mean()
```

```
    df_sorted["moving_avg"] = df_sorted["active_population"].rolling(window,min_periods=1).mean()

    fig_conv_1.add_trace(go.Scatter(
        x=df_sorted["time"],
        y=df_sorted["moving_avg"],
        mode='lines',
        line=dict(width=1.5),
        opacity=0.4,
        name=f"Réplica {i+1}"
    ))

fig_conv_1.update_layout(
    title="Convergencia del Promedio Móvil – Población Activa",
    xaxis_title="Hora del día (0–24h)",
    yaxis_title="Promedio móvil de jugadores activos",
    template="plotly_white",
    height=500
)

fig_conv_1.show()
```




```
fig_conv_2 = go.Figure()

for i, df in enumerate(raw_metrics_list):
    df_sorted = df.sort_values("time")
    df_sorted["moving_avg"] = df_sorted["avg_wait_time"].rolling(window,min_periods=1).mean()

    fig_conv_2.add_trace(go.Scatter(
        x=df_sorted["time"],
        y=df_sorted["moving_avg"],
        mode='lines',
        line=dict(width=1.5),
        opacity=0.4,
        name=f"Réplica {i+1}"
    ))

fig_conv_2.update_layout(
    title="Convergencia del Promedio Móvil – Tiempo de Espera",
    xaxis_title="Hora del día (0–24h)",
    yaxis_title="Promedio móvil del tiempo de espera (min)",
    template="plotly_white",
    height=500
)

fig_conv_2.show()
```




```
colors = pc.qualitative.Dark24 + pc.qualitative.Light24


replica_summary = []

for r in simulation_results["raw_results"]:
    replica_id = r["replica_id"]
    metrics_df = r["metrics"]
    players_df = r["players"]
    matches_df = r["matches"]

    avg_active = metrics_df["active_population"].mean()
    avg_wait = metrics_df["avg_wait_time"].mean()
    avg_elo_diff = metrics_df["avg_elo_difference"].mean()
    avg_searching = metrics_df["searching"].mean()
    avg_playing = metrics_df["playing"].mean()
    avg_resting = metrics_df["resting"].mean()

    total_matches = r["total_matches"]
    total_players = r["total_players"]

    replica_summary.append({
        "Réplica": replica_id + 1,
        "Partidas Totales": total_matches,
        "Jugadores Totales": total_players,

        "Promedio Población Activa": round(avg_active, 2),
        "Promedio Tiempo Espera": round(avg_wait, 2),
        "Promedio Diferencia Elo": round(avg_elo_diff, 2),

        "Prom. Buscando": round(avg_searching, 2),
        "Prom. Jugando": round(avg_playing, 2),
        "Prom. Descansando": round(avg_resting, 2)
    })

df_replica_summary = pd.DataFrame(replica_summary)

print("TABLA DE ESTADÍSTICAS POR RÉPLICA")
df_replica_summary
```

| | Réplica | Partidas Totales | Jugadores Totales | Promedio Población Activa | Promedio Tiempo Espera | Promedio Diferencia Elo | Prom. Buscando | Prom. Jugando | Prom. Descansando |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 3492 | 15220 | 1052.27 | 3.09 | 1054.96 | 66.04 | 903.30 | 82.93 |
| **1** | 2 | 3498 | 15127 | 1052.48 | 3.03 | 1049.60 | 64.80 | 904.38 | 83.31 |
| **2** | 3 | 3498 | 15222 | 1052.37 | 3.10 | 1058.92 | 64.68 | 904.13 | 83.56 |
| **3** | 4 | 3487 | 15114 | 1052.64 | 3.04 | 1062.12 | 66.30 | 902.85 | 83.50 |
| **4** | 5 | 3480 | 15134 | 1052.61 | 3.10 | 1047.10 | 64.75 | 904.93 | 82.93 |
| **5** | 6 | 3489 | 15107 | 1053.05 | 3.09 | 1057.97 | 65.44 | 904.38 | 83.23 |
| **6** | 7 | 3482 | 15164 | 1052.04 | 3.09 | 1046.73 | 65.38 | 903.16 | 83.50 |
| **7** | 8 | 3469 | 15015 | 1052.70 | 3.10 | 1054.29 | 65.21 | 904.06 | 83.42 |
| **8** | 9 | 3483 | 15207 | 1052.76 | 3.08 | 1065.80 | 64.75 | 904.65 | 83.36 |
| **9** | 10 | 3478 | 15096 | 1052.59 | 3.05 | 1055.88 | 63.52 | 906.35 | 82.72 |
| **10** | 11 | 3486 | 15094 | 1052.35 | 3.05 | 1061.94 | 65.04 | 904.31 | 83.00 |
| **11** | 12 | 3495 | 15223 | 1052.59 | 3.08 | 1053.10 | 66.11 | 902.43 | 84.05 |
| **12** | 13 | 3470 | 15142 | 1052.59 | 3.12 | 1061.89 | 65.61 | 904.48 | 82.50 |
| **13** | 14 | 3500 | 15267 | 1052.01 | 3.05 | 1061.52 | 65.54 | 903.30 | 83.18 |
| **14** | 15 | 3491 | 15184 | 1052.55 | 3.07 | 1046.39 | 65.06 | 904.51 | 82.97 |
| **15** | 16 | 3494 | 15114 | 1052.71 | 3.03 | 1060.05 | 63.94 | 905.42 | 83.35 |
| **16** | 17 | 3490 | 15218 | 1051.87 | 3.23 | 1065.26 | 65.08 | 903.82 | 82.97 |
| **17** | 18 | 3486 | 15139 | 1053.05 | 3.09 | 1054.24 | 64.02 | 906.32 | 82.71 |
| **18** | 19 | 3502 | 15231 | 1052.53 | 3.08 | 1047.40 | 64.93 | 903.92 | 83.68 |