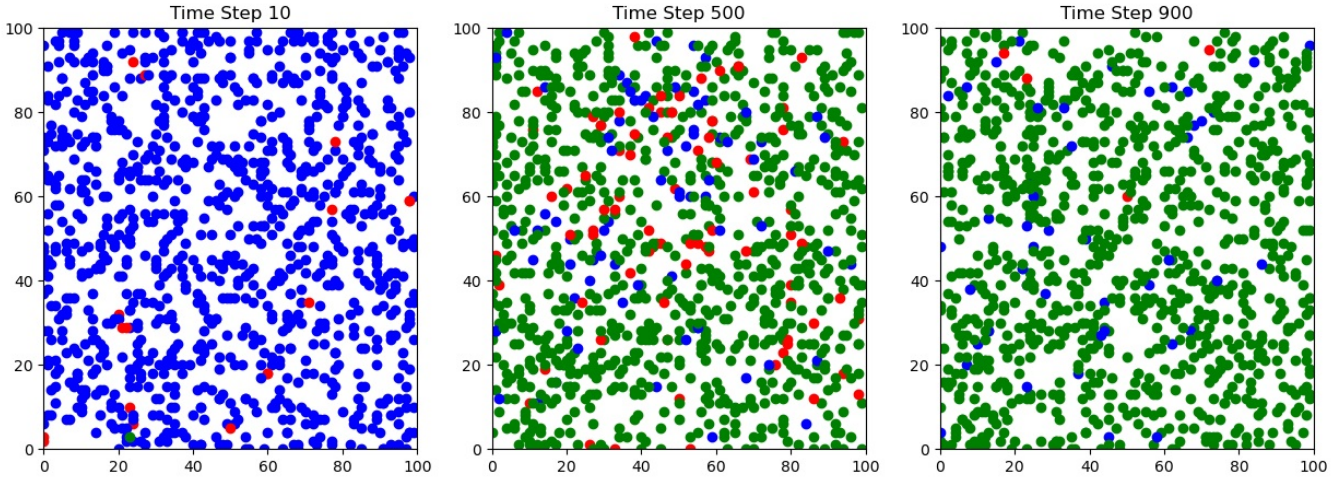
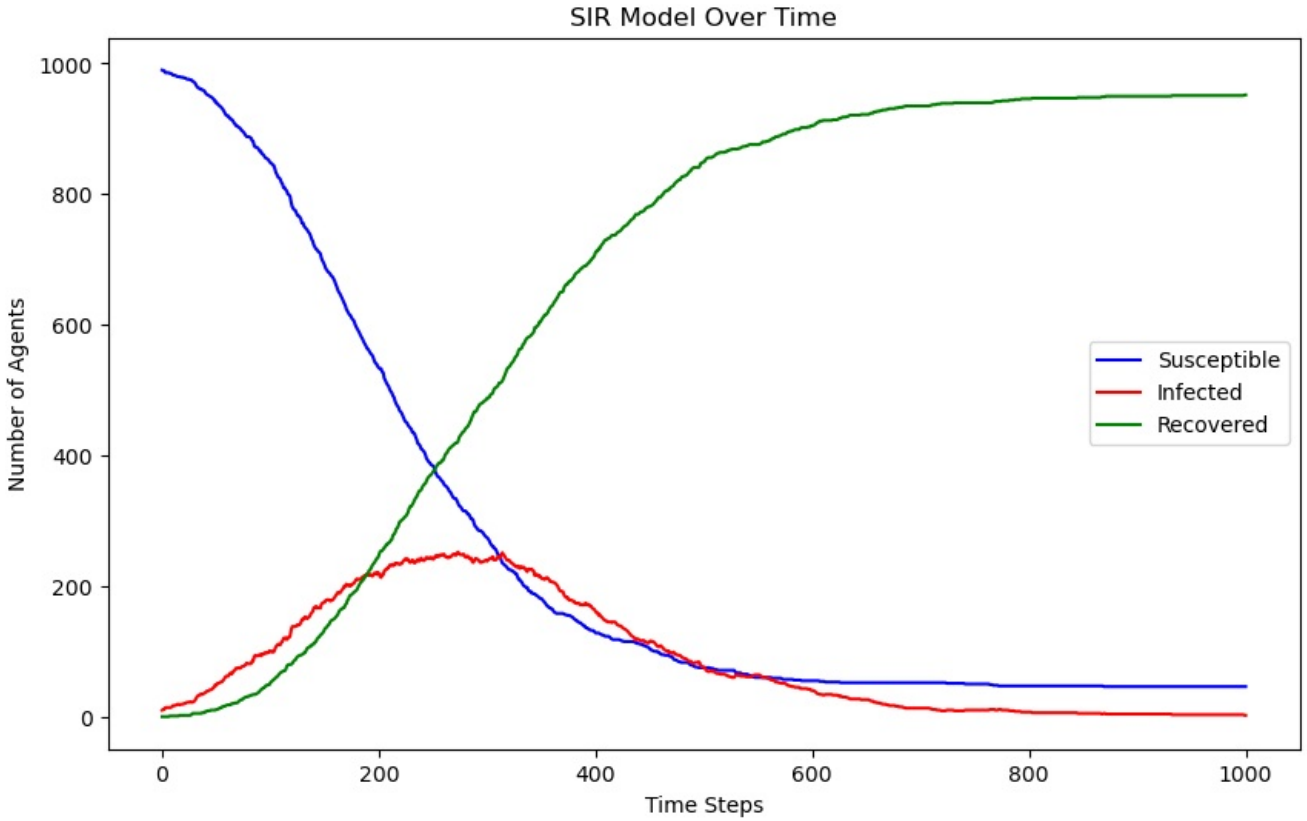
In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from numba import njit

@njit
def initialize_agents(lattice_size, num_agents, initial_infection_rate):
    num_infected = int(num_agents * initial_infection_rate)
    agents = np.zeros((num_agents, 3), dtype=np.int32)

    # Create a shuffled array of indices
    indices = np.arange(num_agents)
    np.random.shuffle(indices)

    for i in range(num_agents):
        agents[i, 0] = np.random.randint(0, lattice_size)
        agents[i, 1] = np.random.randint(0, lattice_size)
        # Infect a random 1% of the agents, based on the indices shuffled
        agents[i, 2] = 1 if i in indices[:num_infected] else 0

    return agents

@njit
def move_agents(agents, lattice_size, diffusion_rate):
    for i in range(agents.shape[0]):
        if np.random.random() < diffusion_rate:
            move_x = np.random.randint(-1, 2)
            move_y = np.random.randint(-1, 2)
            agents[i, 0] = (agents[i, 0] + move_x) % lattice_size
            agents[i, 1] = (agents[i, 1] + move_y) % lattice_size

@njit
def infect_agents(agents, beta, lattice_size):
    for i in range(agents.shape[0]):
        if agents[i, 2] == 1:
            for j in range(agents.shape[0]):
                if agents[j, 2] == 0 and agents[i, 0] == agents[j, 0] and agents[i, 1] == agents[j, 1]:
                    if np.random.random() < beta:
                        agents[j, 2] = 1

@njit
def update_recovery(agents, gamma):
    for i in range(agents.shape[0]):
        if agents[i, 2] == 1 and np.random.random() < gamma:
            agents[i, 2] = 2

@njit
def update_agents(agents, lattice_size, diffusion_rate, beta, gamma):
    move_agents(agents, lattice_size, diffusion_rate)
    infect_agents(agents, beta, lattice_size)
    update_recovery(agents, gamma)
```

In [2]:
```python
# Simulation params
lattice_size = 100
num_agents = 1000
beta = 0.6      # Infection rate
gamma = 0.01    # Recovery rate
diffusion_rate = 0.8  # Diffusion rate, probability to move
total_time_steps = 1000
initial_infection_rate = 0.01

# Initialize agents
agents = initialize_agents(lattice_size, num_agents, initial_infection_rate)

# Lists to track the number of S, I, R
susceptible_count = []
infected_count = []
recovered_count = []

# Simulation and Plotting
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
time_steps_to_plot = [10, 500, 900]
cmap = ListedColormap(['blue', 'red', 'green'])

for step in range(total_time_steps + 1):
    # Count S, I, R
    state_count = np.bincount(agents[:, 2], minlength=3)
    susceptible_count.append(state_count[0])
    infected_count.append(state_count[1])
    recovered_count.append(state_count[2])

    if step in time_steps_to_plot:
        ax = axes[time_steps_to_plot.index(step)]
        ax.set_title(f"Time Step {step}")
        ax.set_xlim(0, lattice_size)
        ax.set_ylim(0, lattice_size)
```

```
        for agent in agents:
            color = cmap(agent[2])
            ax.scatter(agent[0], agent[1], color=color)
    update_agents(agents, lattice_size, diffusion_rate, beta, gamma)

plt.show()

# Plot S, I, R over time
plt.figure(figsize=(10, 6))
plt.plot(susceptible_count, label='Susceptible', color='blue')
plt.plot(infected_count, label='Infected', color='red')
plt.plot(recovered_count, label='Recovered', color='green')
plt.title('SIR Model Over Time')
plt.xlabel('Time Steps')
plt.ylabel('Number of Agents')
plt.legend()
plt.show()
```





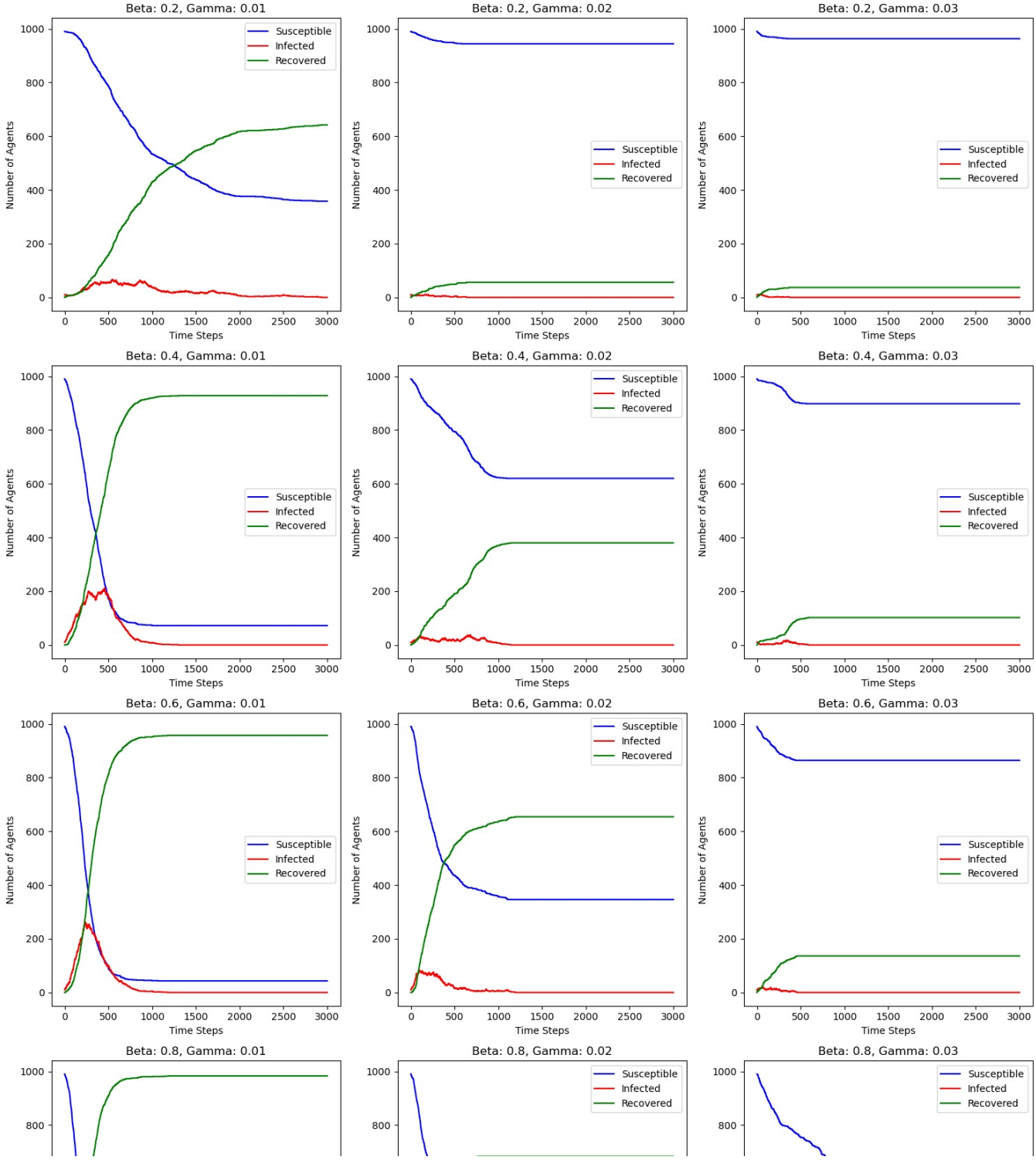
In [5]:

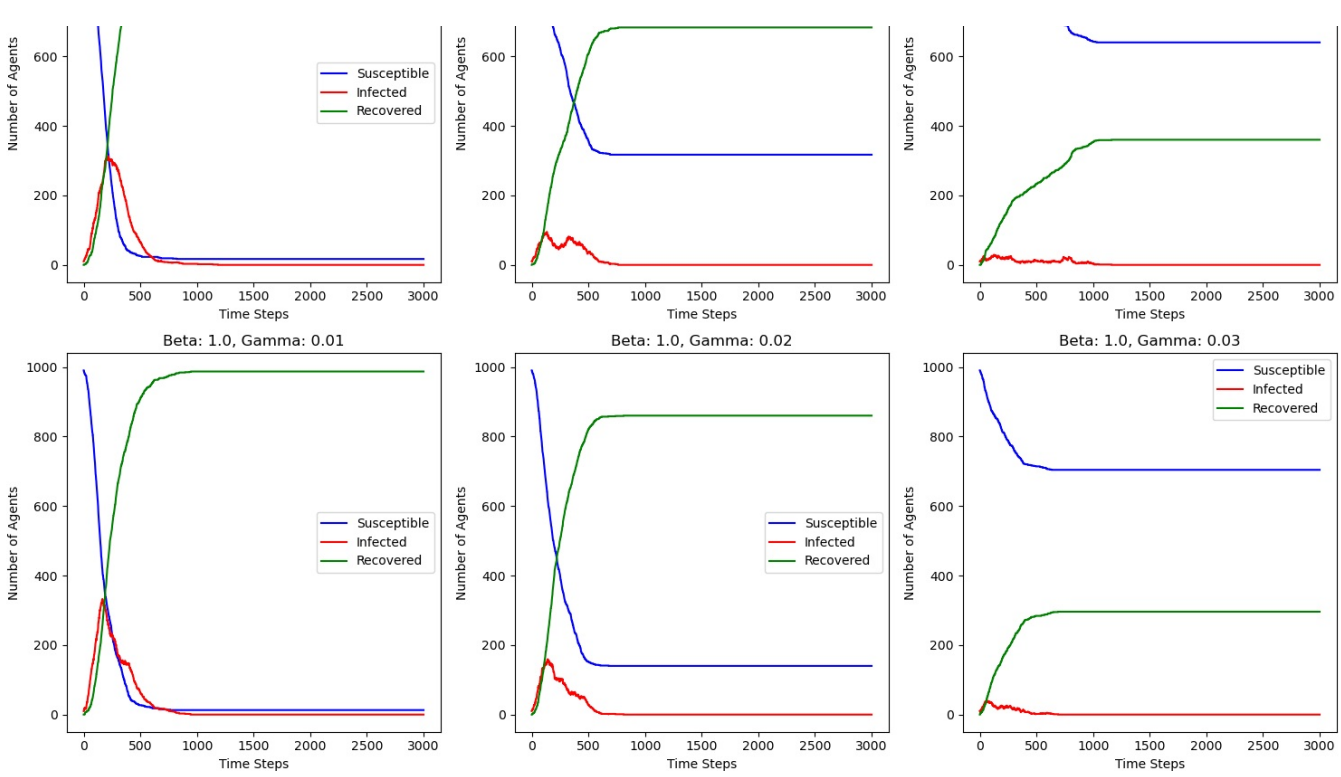
```
@njit
def run_simulation(beta, gamma, total_time_steps):
    # Initialize agents
    agents = initialize_agents(lattice_size, num_agents, initial_infection_rate)

    # Lists to track the number of S, I, and R
    susceptible_count = []
    infected_count = []
    recovered_count = []

    # Run Simulation & Count SIR States
    for step in range(total_time_steps + 1):
        state_count = np.bincount(agents[:, 2], minlength=3)
        susceptible_count.append(state_count[0])
        infected_count.append(state_count[1])
```

```
        recovered_count.append(state_count[2])
        update_agents(agents, lattice_size, diffusion_rate, beta, gamma)

    return susceptible_count, infected_count, recovered_count
```

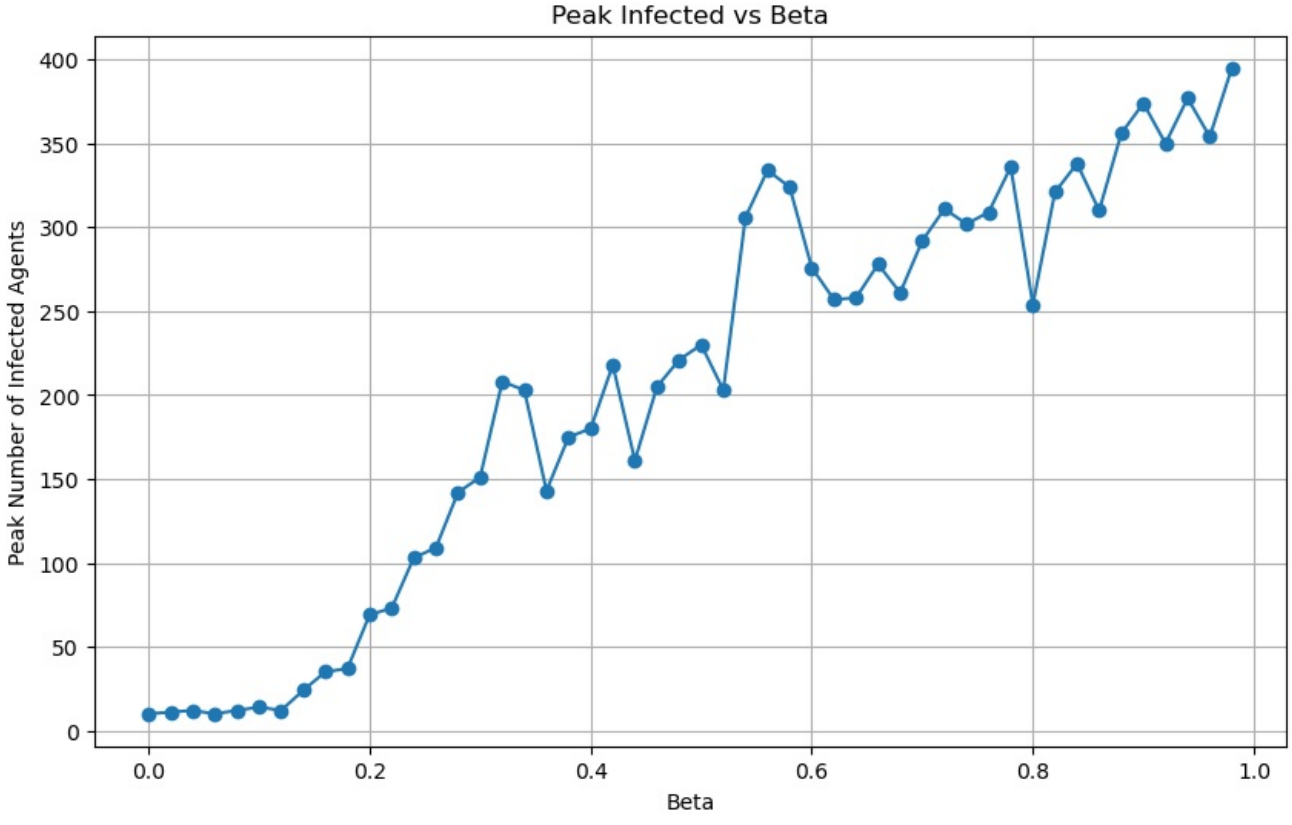
In [9]:

```
# Varying beta, gamma values
beta_values = [0.2, 0.4, 0.6, 0.8, 1.0]
gamma_values = [0.01, 0.02, 0.03]

plt.figure(figsize=(15, 25))

for i, beta in enumerate(beta_values):
    for j, gamma in enumerate(gamma_values):
        susceptible, infected, recovered = run_simulation(beta, gamma, total_time_steps=3000)

        # Plot SIR evolution beta-gamma combination
        plt.subplot(len(beta_values), len(gamma_values), i * len(gamma_values) + j + 1)
        plt.plot(susceptible, label='Susceptible', color='blue')
        plt.plot(infected, label='Infected', color='red')
        plt.plot(recovered, label='Recovered', color='green')
        plt.title(f'Beta: {beta}, Gamma: {gamma}')
        plt.xlabel('Time Steps')
        plt.ylabel('Number of Agents')
        plt.legend()

plt.tight_layout()
plt.show()
```



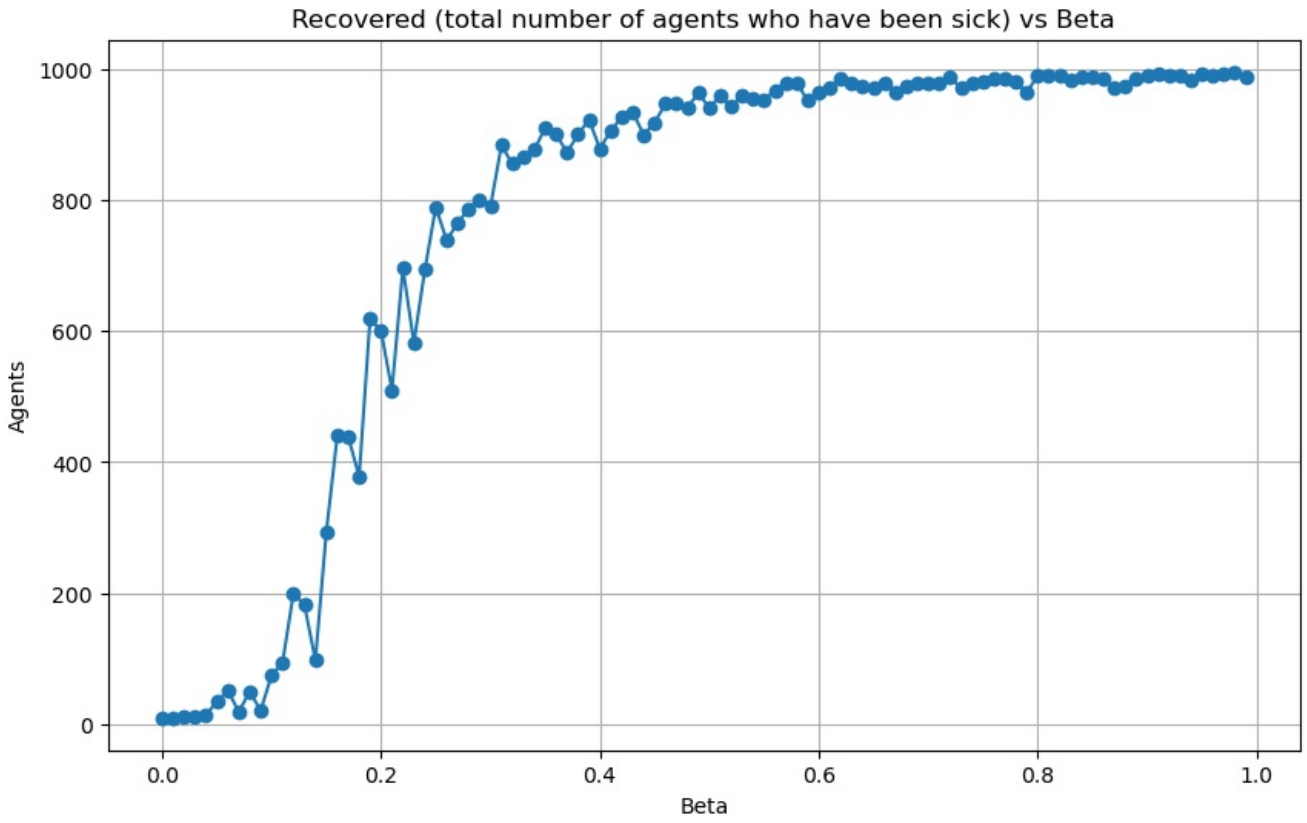


In [7]:
```
gamma = 0.01
beta_range = np.arange(0, 1, 0.02)

peak_infected_counts = []

for beta in beta_range:
    susceptible, infected, recovered = run_simulation(beta, gamma, total_time_steps=5000)
    peak_infected = max(infected)  # Find the peak number of infected agents
    peak_infected_counts.append(peak_infected)

plt.figure(figsize=(10, 6))
plt.plot(beta_range, peak_infected_counts, marker='o', linestyle='-')
plt.title('Peak Infected vs Beta')
plt.xlabel('Beta')
plt.ylabel('Peak Number of Infected Agents')
plt.grid(True)
plt.show()
```



In [8]:
```
gamma = 0.01
beta_range = np.arange(0, 1, 0.01)

recovered_counts = []
```

```
for beta in beta_range:
    susceptible, infected, recovered = run_simulation(beta, gamma, total_time_steps=5000)
    recovered = max(recovered)
    recovered_counts.append(recovered)

plt.figure(figsize=(10, 6))
plt.plot(beta_range, recovered_counts, marker='o', linestyle='-')
plt.title('Recovered (total number of agents who have been sick) vs Beta')
plt.xlabel('Beta')
plt.ylabel('Agents')
plt.grid(True)
plt.show()
```



In [ ]:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from numba import njit

@njit
def initialize_agents(lattice_size, num_agents, initial_infection_rate):
    num_infected = int(num_agents * initial_infection_rate)
    agents = np.zeros((num_agents, 3), dtype=np.int32)

    # Create a shuffled array of indices
    indices = np.arange(num_agents)
    np.random.shuffle(indices)

    for i in range(num_agents):
        agents[i, 0] = np.random.randint(0, lattice_size)
        agents[i, 1] = np.random.randint(0, lattice_size)
        # Infect a random 1% of the agents, based on the indices shuffled
        agents[i, 2] = 1 if i in indices[:num_infected] else 0

    return agents

@njit
def move_agents(agents, lattice_size, diffusion_rate):
    for i in range(agents.shape[0]):
        if np.random.random() < diffusion_rate:
            move_x = np.random.randint(-1, 2)
            move_y = np.random.randint(-1, 2)
            agents[i, 0] = (agents[i, 0] + move_x) % lattice_size
            agents[i, 1] = (agents[i, 1] + move_y) % lattice_size

@njit
def infect_agents(agents, beta, lattice_size):
    for i in range(agents.shape[0]):
        if agents[i, 2] == 1:
            for j in range(agents.shape[0]):
                if agents[j, 2] == 0 and agents[i, 0] == agents[j, 0] and agents[i, 1] == agents[j, 1]:
                    if np.random.random() < beta:
                        agents[j, 2] = 1

@njit
def update_recovery(agents, gamma):
    for i in range(agents.shape[0]):
        if agents[i, 2] == 1 and np.random.random() < gamma:
            agents[i, 2] = 2

@njit
def update_agents(agents, lattice_size, diffusion_rate, beta, gamma):
    move_agents(agents, lattice_size, diffusion_rate)
    infect_agents(agents, beta, lattice_size)
    update_recovery(agents, gamma)
```

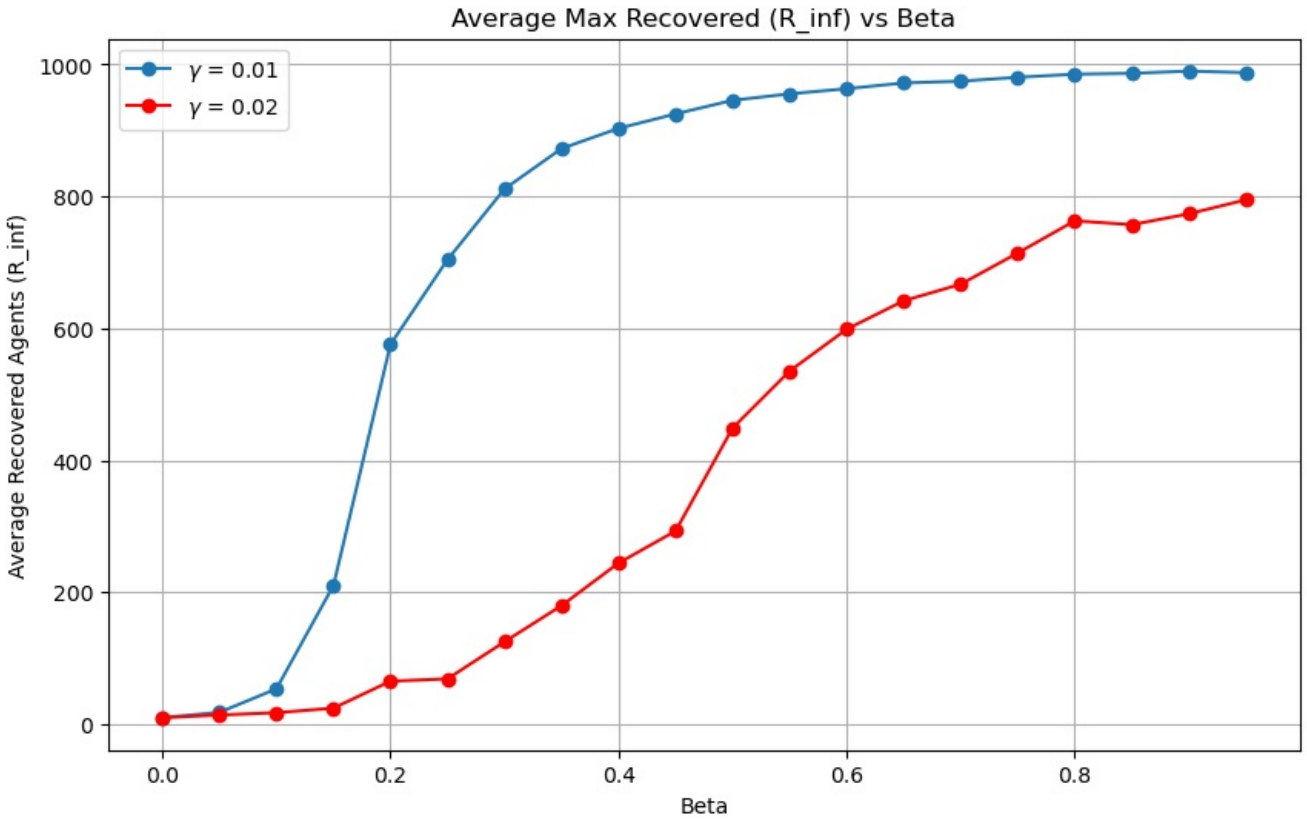
In [2]:
```
# Simulation params
lattice_size = 100
num_agents = 1000
#beta = 0.6      # Infection rate
gamma = 0.01    # Recovery rate
diffusion_rate = 0.8  # Diffusion rate, probability to move
total_time_steps = 5000
initial_infection_rate = 0.01
```

In [3]:
```
@njit
def run_simulation(beta, gamma):
    # Initialize agents
    agents = initialize_agents(lattice_size, num_agents, initial_infection_rate)

    susceptible_count = []
    infected_count = []
    recovered_count = []

    # Run simulation and count SIR
    for step in range(total_time_steps + 1):
        state_count = np.bincount(agents[:, 2], minlength=3)
        susceptible_count.append(state_count[0])
        infected_count.append(state_count[1])
        recovered_count.append(state_count[2])
        update_agents(agents, lattice_size, diffusion_rate, beta, gamma)

    return susceptible_count, infected_count, recovered_count
```

R_inf vs β plot

In [4]:
```
beta_range = np.arange(0, 1, 0.05)
num_sim = 10
```

```
R_inf_gamma_001 = []
R_inf_gamma_002 = []

gamma = 0.01
for beta in beta_range:
    total_max_recovered = 0
    for _ in range(num_sim):
        susceptible, infected, recovered = run_simulation(beta, gamma)
        total_max_recovered += max(recovered)
    average_max_recovered = total_max_recovered / num_sim
    R_inf_gamma_001.append(average_max_recovered)

gamma = 0.02
for beta in beta_range:
    total_max_recovered = 0
    for _ in range(num_sim):
        susceptible, infected, recovered = run_simulation(beta, gamma)
        total_max_recovered += max(recovered)
    average_max_recovered = total_max_recovered / num_sim
    R_inf_gamma_002.append(average_max_recovered)

plt.figure(figsize=(10, 6))
plt.plot(beta_range, R_inf_gamma_001, marker='o', linestyle='-', label='$\gamma$ = 0.01')
plt.plot(beta_range, R_inf_gamma_002, marker='o', linestyle='-', color='red', label='$\gamma$ = 0.02')
plt.title('Average Max Recovered (R_inf) vs Beta')
plt.xlabel('Beta')
plt.ylabel('Average Recovered Agents (R_inf)')
plt.legend()
plt.grid(True)
plt.show()
```

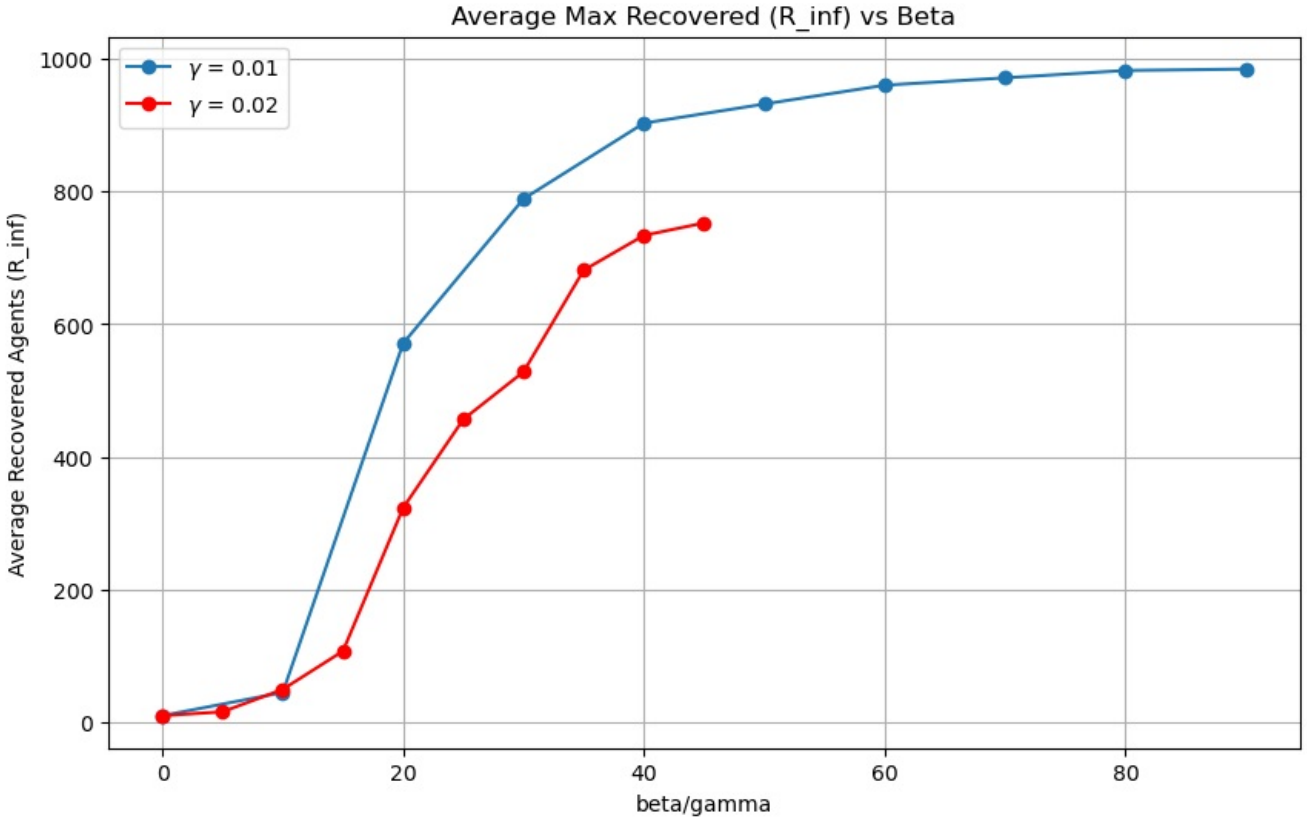


R_inf vs β/γ plot

In [5]:

```
beta_range = np.arange(0, 1, 0.1)
num_sim = 10

R_inf_gamma_001 = []
R_inf_gamma_002 = []

gamma = 0.01
for beta in beta_range:
    total_max_recovered = 0
    for _ in range(num_sim):
        susceptible, infected, recovered = run_simulation(beta, gamma)
        total_max_recovered += max(recovered)
    average_max_recovered = total_max_recovered / num_sim
    R_inf_gamma_001.append(average_max_recovered)

gamma = 0.02
for beta in beta_range:
    total_max_recovered = 0
```

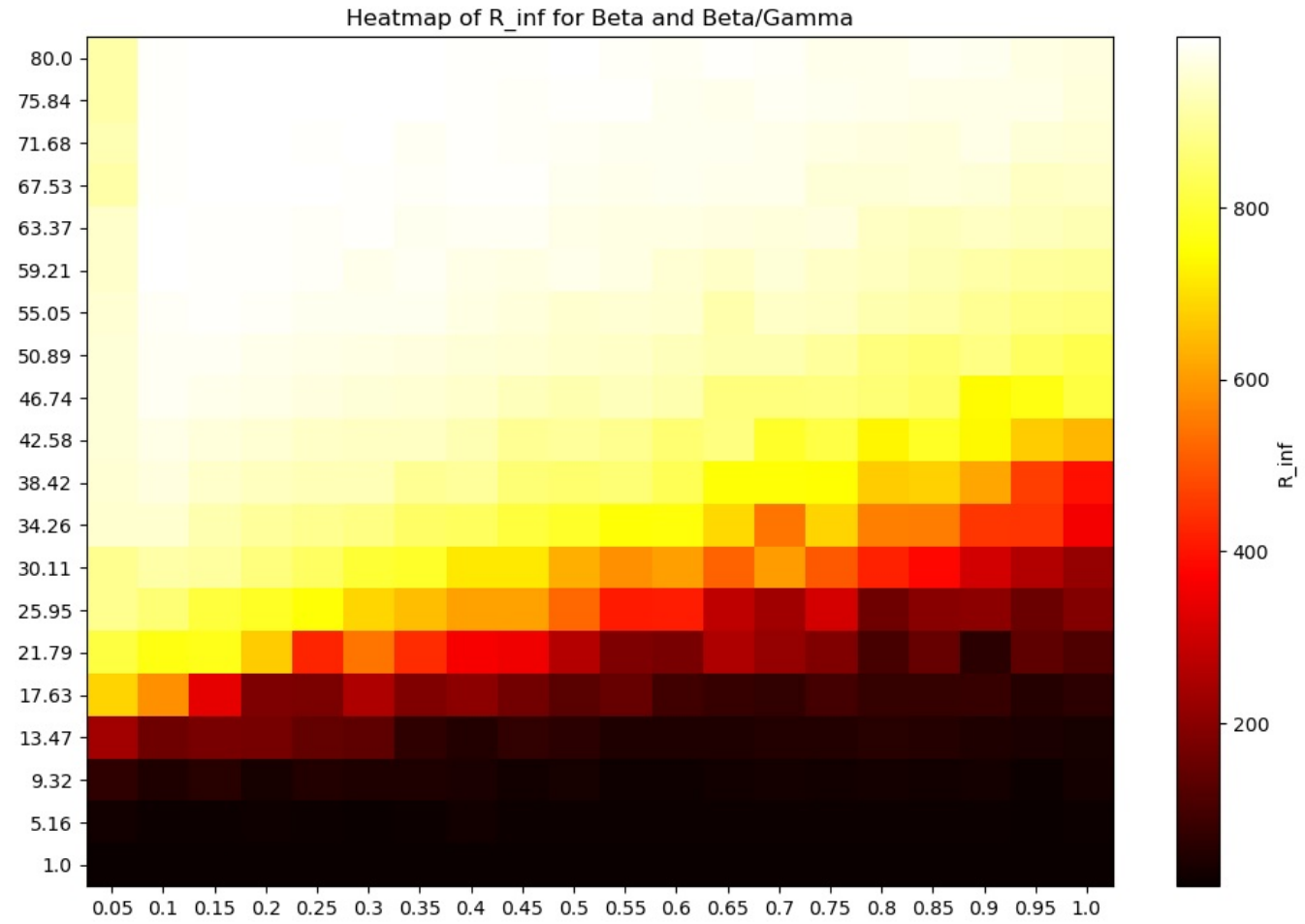
```
    for _ in range(num_sim):
        susceptible, infected, recovered = run_simulation(beta, gamma)
        total_max_recovered += max(recovered)
    average_max_recovered = total_max_recovered / num_sim
    R_inf_gamma_002.append(average_max_recovered)

plt.figure(figsize=(10, 6))
plt.plot(beta_range/0.01, R_inf_gamma_001, marker='o', linestyle='-', label='$\gamma$ = 0.01')
plt.plot(beta_range/0.02, R_inf_gamma_002, marker='o', linestyle='-', color='red', label='$\gamma$ = 0.02')
plt.title('Average Max Recovered (R_inf) vs Beta')
plt.xlabel('beta/gamma')
plt.ylabel('Average Recovered Agents (R_inf)')
plt.legend()
plt.grid(True)
plt.show()
```



Phase diagram

In [6]:

```
# beta and y ranges
beta_range = np.linspace(0.05, 1, 20)
y_range = np.linspace(1, 80, 20)[::-1]
num_sim = 3

R_inf_grid = np.zeros((len(beta_range), len(y_range)))

# Simulation for each combination of beta and y (correlates with gamma)
for i, beta in enumerate(beta_range):
    for j, y in enumerate(y_range):
        gamma = beta / y
        R_inf = 0
        for _ in range(num_sim):
            _, _, recovered = run_simulation(beta, gamma)
            R_inf += max(recovered)

        R_inf_grid[j, i] = R_inf/num_sim

plt.figure(figsize=(12, 8))
plt.imshow(R_inf_grid, cmap='hot', interpolation='nearest', aspect='auto')
plt.colorbar(label='R_inf')

plt.yticks(np.arange(len(beta_range)), np.round(y_range, 2))
plt.xticks(np.arange(len(y_range)), np.round(beta_range, 2))

plt.title('Heatmap of R_inf for Beta and Beta/Gamma')
plt.show()
```



In [ ]:

Processing math: 100%

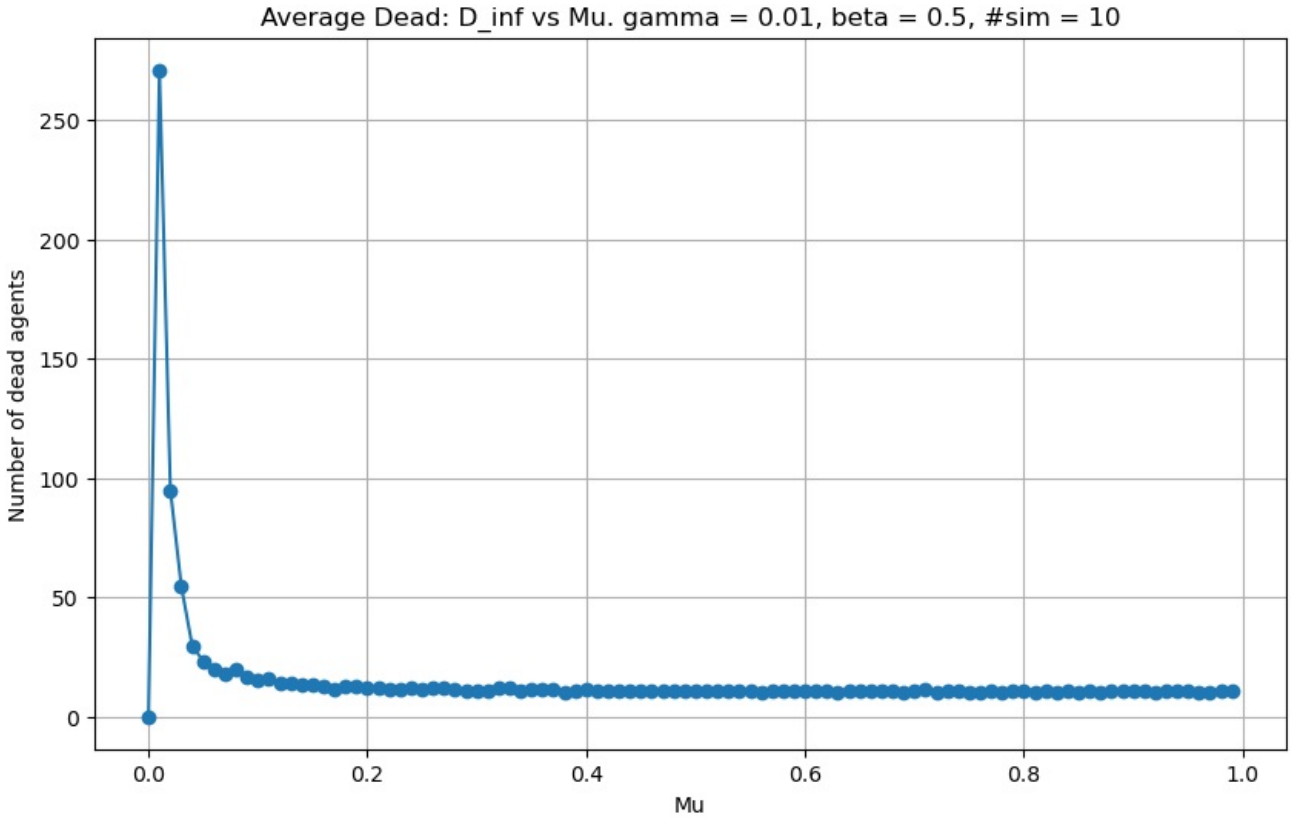
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from numba import njit

@njit
def initialize_agents(lattice_size, num_agents, initial_infection_rate):
    num_infected = int(num_agents * initial_infection_rate)
    agents = np.zeros((num_agents, 3), dtype=np.int32)

    # Create a shuffled array of indices
    indices = np.arange(num_agents)
    np.random.shuffle(indices)

    for i in range(num_agents):
        agents[i, 0] = np.random.randint(0, lattice_size)
        agents[i, 1] = np.random.randint(0, lattice_size)
        # Infect a random 1% of the agents, based on the indices shuffled
        agents[i, 2] = 1 if i in indices[:num_infected] else 0

    return agents

@njit
def move_agents(agents, lattice_size, diffusion_rate):
    for i in range(agents.shape[0]):
        if np.random.random() < diffusion_rate:
            move_x = np.random.randint(-1, 2)
            move_y = np.random.randint(-1, 2)
            agents[i, 0] = (agents[i, 0] + move_x) % lattice_size
            agents[i, 1] = (agents[i, 1] + move_y) % lattice_size

@njit
def infect_agents(agents, beta, lattice_size):
    for i in range(agents.shape[0]):
        if agents[i, 2] == 1:
            for j in range(agents.shape[0]):
                if agents[j, 2] == 0 and agents[i, 0] == agents[j, 0] and agents[i, 1] == agents[j, 1]:
                    if np.random.random() < beta:
                        agents[j, 2] = 1

@njit
def update_recovery(agents, gamma):
    for i in range(agents.shape[0]):
        if agents[i, 2] == 1 and np.random.random() < gamma:
            agents[i, 2] = 2

@njit
def update_death(agents, mu):
    for i in range(agents.shape[0]):
        # Check if agent infected
        if agents[i, 2] == 1 and np.random.random() < mu:
            agents[i, 2] = 3  # state 3 = dead


@njit
def update_agents(agents, lattice_size, diffusion_rate, beta, gamma, mu):
    move_agents(agents, lattice_size, diffusion_rate)
    infect_agents(agents, beta, lattice_size)
    update_recovery(agents, gamma)
    update_death(agents, mu)
```

In [2]:
```
# Simulation params
lattice_size = 100
num_agents = 1000
diffusion_rate = 0.8  # Diffusion rate, probability to move
total_time_steps = 2000
initial_infection_rate = 0.01
```

In [3]:
```
@njit
def run_simulation(beta, gamma, mu):
    # Initialize agents
    agents = initialize_agents(lattice_size, num_agents, initial_infection_rate)

    # Lists to track the number of S, I, R
    susceptible_count = []
    infected_count = []
    recovered_count = []
    dead_count = []

    # Run Simulation & Count SIR States
    for step in range(total_time_steps + 1):
        state_count = np.bincount(agents[:, 2], minlength=4)
        susceptible_count.append(state_count[0])
        infected_count.append(state_count[1])
        recovered_count.append(state_count[2])
        dead_count.append(state_count[3])
```

```
        update_agents(agents, lattice_size, diffusion_rate, beta, gamma, mu)

    return susceptible_count, infected_count, recovered_count, dead_count
```

In [4]:
```
beta = 0.6      # Infection rate
gamma = 0.01    # Recovery rate
mu_range = np.arange(0, 1, 0.01)
num_sim = 10

number_dead = []

for mu in mu_range:
    total_max_dead = 0
    for _ in range(num_sim):
        susceptible, infected, recovered, dead  = run_simulation(beta, gamma, mu)
        total_max_dead += max(dead)
    total_max_dead = total_max_dead / num_sim
    number_dead.append(total_max_dead)


plt.figure(figsize=(10, 6))
plt.plot(mu_range, number_dead, marker='o', linestyle='-')
plt.title('Average Dead: D_inf vs Mu. gamma = 0.01, beta = 0.5, #sim = 10')
plt.xlabel('Mu')
plt.ylabel('Number of dead agents')
plt.grid(True)
plt.show()
```

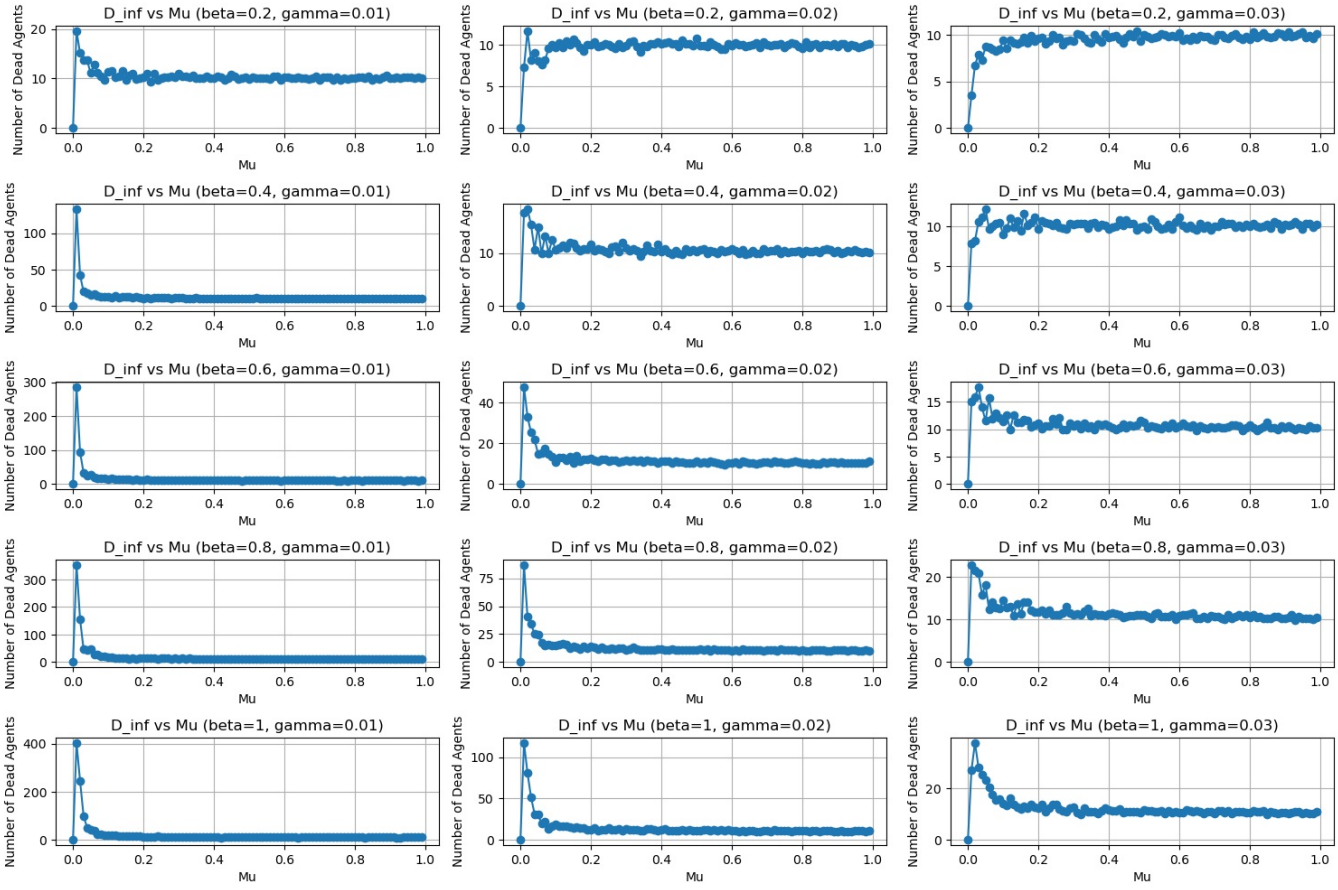


If mu very high, sick people die quick and therefore no more sick people remain.

In [7]:
```
beta_values = [0.2, 0.4, 0.6, 0.8, 1]
gamma_values = [0.01, 0.02, 0.03]
mu_range = np.arange(0, 1, 0.01)
num_sim = 10

plt.figure(figsize=(15, 10))

for i, beta in enumerate(beta_values):
    for j, gamma in enumerate(gamma_values):
        number_dead = []

        for mu in mu_range:
            total_max_dead = 0
            for _ in range(num_sim):
                susceptible, infected, recovered, dead = run_simulation(beta, gamma, mu)
                total_max_dead += max(dead)
            total_max_dead = total_max_dead / num_sim
            number_dead.append(total_max_dead)

        # Find max D_inf and corresponding mu
        max_dead = max(number_dead)
        max_mu = mu_range[number_dead.index(max_dead)]
```

```
        # Print max D_inf and corresponding mu
        print(f"Max D_inf: {max_dead} at mu: {max_mu} for beta: {beta} and gamma: {gamma}")

        # Plot
        plt.subplot(len(beta_values), len(gamma_values), i * len(gamma_values) + j + 1)
        plt.plot(mu_range, number_dead, marker='o', linestyle='-')
        plt.title(f'D_inf vs Mu (beta={beta}, gamma={gamma})')
        plt.xlabel('Mu')
        plt.ylabel('Number of Dead Agents')
        plt.grid(True)

plt.tight_layout()
plt.show()
```

```
Max D_inf: 19.6 at mu: 0.01 for beta: 0.2 and gamma: 0.01
Max D_inf: 11.7 at mu: 0.02 for beta: 0.2 and gamma: 0.02
Max D_inf: 10.4 at mu: 0.48 for beta: 0.2 and gamma: 0.03
Max D_inf: 133.4 at mu: 0.01 for beta: 0.4 and gamma: 0.01
Max D_inf: 18.3 at mu: 0.02 for beta: 0.4 and gamma: 0.02
Max D_inf: 12.2 at mu: 0.05 for beta: 0.4 and gamma: 0.03
Max D_inf: 286.5 at mu: 0.01 for beta: 0.6 and gamma: 0.01
Max D_inf: 47.9 at mu: 0.01 for beta: 0.6 and gamma: 0.02
Max D_inf: 17.7 at mu: 0.03 for beta: 0.6 and gamma: 0.03
Max D_inf: 353.1 at mu: 0.01 for beta: 0.8 and gamma: 0.01
Max D_inf: 87.1 at mu: 0.01 for beta: 0.8 and gamma: 0.02
Max D_inf: 22.8 at mu: 0.01 for beta: 0.8 and gamma: 0.03
Max D_inf: 404.0 at mu: 0.01 for beta: 1 and gamma: 0.01
Max D_inf: 117.2 at mu: 0.01 for beta: 1 and gamma: 0.02
Max D_inf: 37.6 at mu: 0.02 for beta: 1 and gamma: 0.03
```



In [ ]:

In [8]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from numba import njit

@njit
def initialize_agents(lattice_size, num_agents, initial_infection_rate):
    num_infected = int(num_agents * initial_infection_rate)
    agents = np.zeros((num_agents, 3), dtype=np.int32)

    # Create a shuffled array of indices
    indices = np.arange(num_agents)
    np.random.shuffle(indices)

    for i in range(num_agents):
        agents[i, 0] = np.random.randint(0, lattice_size)
        agents[i, 1] = np.random.randint(0, lattice_size)
        # Infect a random 1% of the agents, based on the indices shuffled
        agents[i, 2] = 1 if i in indices[:num_infected] else 0

    return agents

@njit
def move_agents(agents, lattice_size, diffusion_rate):
    for i in range(agents.shape[0]):
        if np.random.random() < diffusion_rate:
            move_x = np.random.randint(-1, 2)
            move_y = np.random.randint(-1, 2)
            agents[i, 0] = (agents[i, 0] + move_x) % lattice_size
            agents[i, 1] = (agents[i, 1] + move_y) % lattice_size

@njit
def infect_agents(agents, beta, lattice_size):
    for i in range(agents.shape[0]):
        if agents[i, 2] == 1:
            for j in range(agents.shape[0]):
                if agents[j, 2] == 0 and agents[i, 0] == agents[j, 0] and agents[i, 1] == agents[j, 1]:
                    if np.random.random() < beta:
                        agents[j, 2] = 1

@njit
def update_recovery(agents, gamma):
    for i in range(agents.shape[0]):
        if agents[i, 2] == 1 and np.random.random() < gamma:
            agents[i, 2] = 2

@njit
def update_susceptibility(agents, alpha):
    for i in range(agents.shape[0]):
        if agents[i, 2] == 2 and np.random.random() < alpha:
            agents[i, 2] = 0  # Recovered agent becomes susceptible again

@njit
def update_agents(agents, lattice_size, diffusion_rate, beta, gamma, alpha):
    move_agents(agents, lattice_size, diffusion_rate)
    infect_agents(agents, beta, lattice_size)
    update_recovery(agents, gamma)
    update_susceptibility(agents, alpha)
```

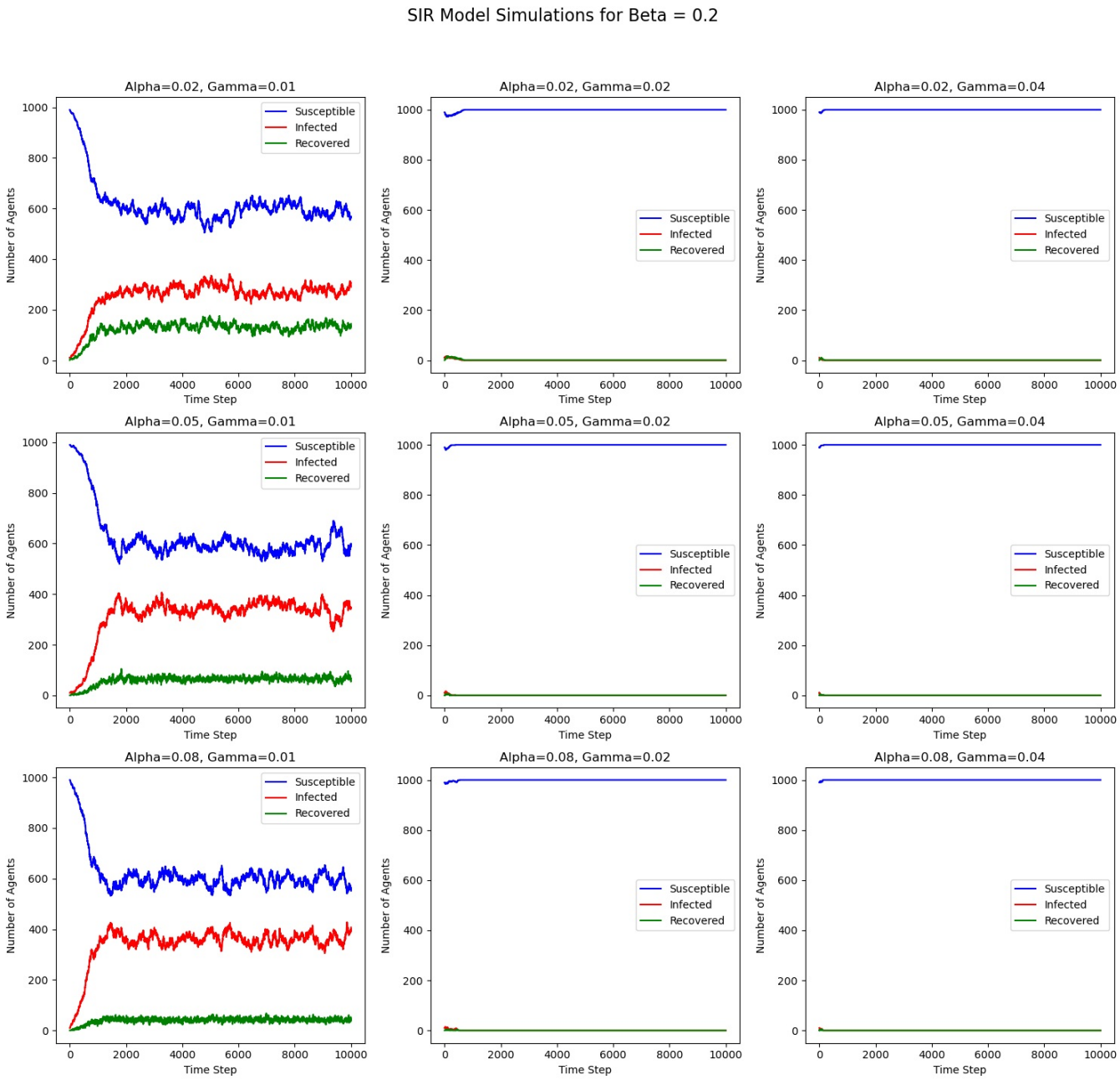
In [9]:
```
# Simulation params
lattice_size = 100
num_agents = 1000
diffusion_rate = 0.8  # Diffusion rate, probability to move
total_time_steps = 10000
```

In [10]:
```
@njit
def run_simulation(beta, gamma, alpha, initial_infection_rate):
    # Initialize agents
    agents = initialize_agents(lattice_size, num_agents, initial_infection_rate)

    # Lists to track S, I, R
    susceptible_count = []
    infected_count = []
    recovered_count = []

    # Run sim, count SIR
    for step in range(total_time_steps + 1):
        state_count = np.bincount(agents[:, 2], minlength=3)
        susceptible_count.append(state_count[0])
        infected_count.append(state_count[1])
        recovered_count.append(state_count[2])
        update_agents(agents, lattice_size, diffusion_rate, beta, gamma, alpha)

    return susceptible_count, infected_count, recovered_count
```

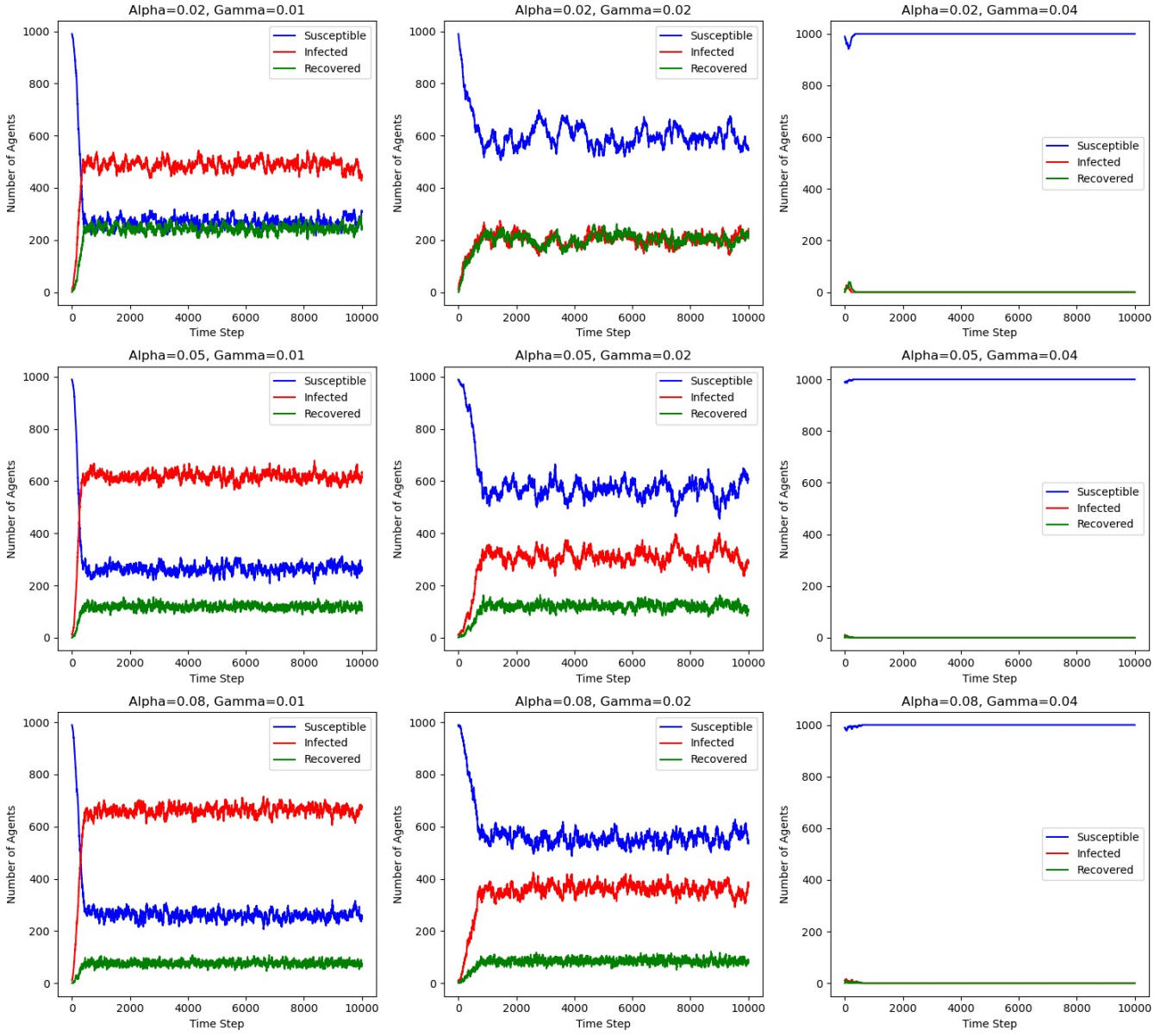
Varying alpha, beta and gamma

Varying alpha, beta and gamma

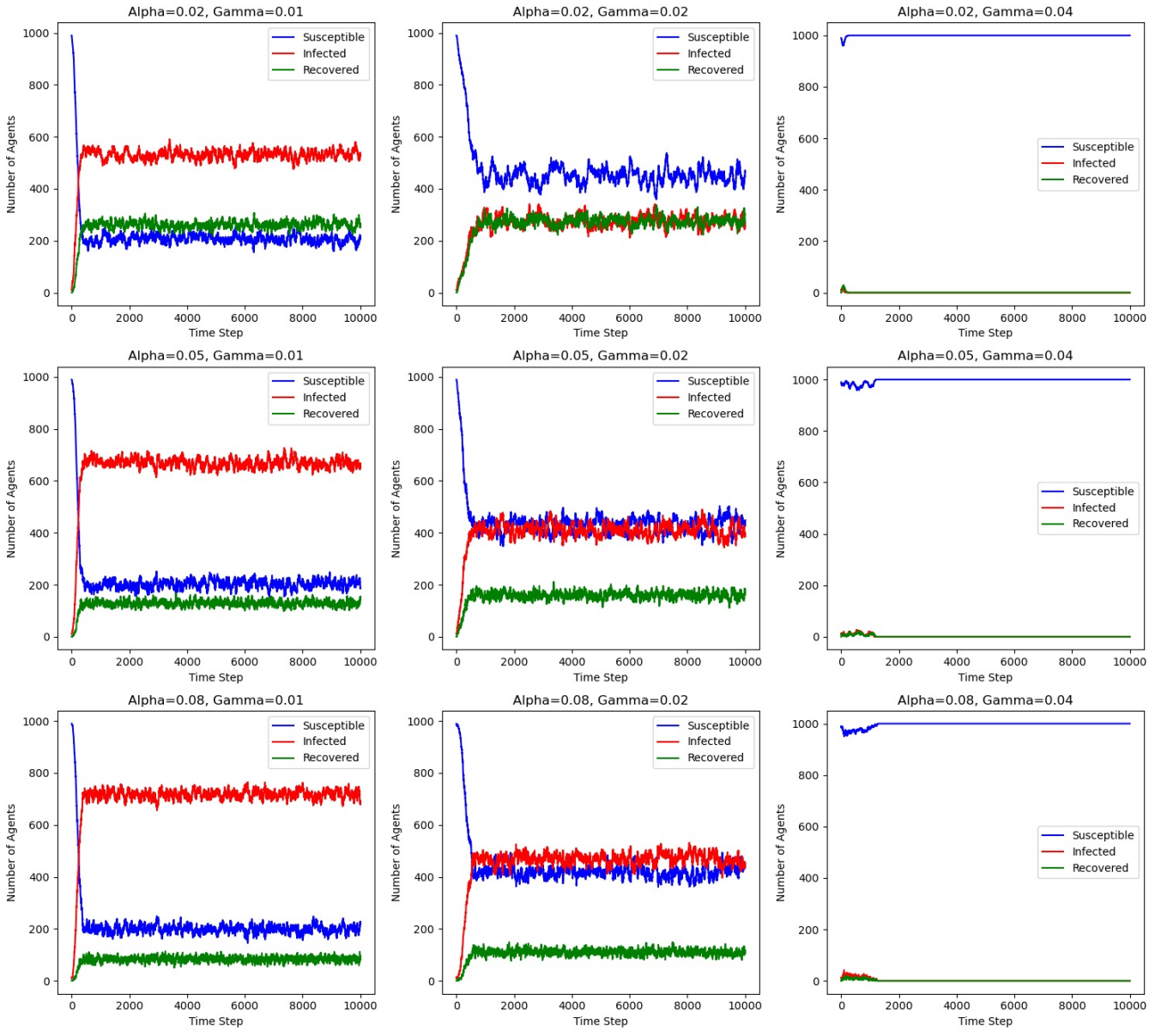
In [11]:
```
initial_infection_rate = 0.01
alphas = [0.02, 0.05, 0.08]
betas = [0.2, 0.5, 0.7, 0.9]
gammas = [0.01, 0.02, 0.04]

# ONE plot for each beta, containing 9 subplots for varying beta and alpha
for beta in betas:
    fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(15, 15))
    fig.suptitle(f'SIR Model Simulations for Beta = {beta}', fontsize=16)

    for i, alpha in enumerate(alphas):
        for j, gamma in enumerate(gammas):
            susceptible, infected, recovered = run_simulation(beta, gamma, alpha, initial_infection_rate)

            ax = axes[i, j]
            ax.plot(susceptible, label='Susceptible', color='blue')
            ax.plot(infected, label='Infected', color='red')
            ax.plot(recovered, label='Recovered', color='green')
            ax.set_title(f'Alpha={alpha}, Gamma={gamma}')
            ax.set_xlabel('Time Step')
            ax.set_ylabel('Number of Agents')
            ax.legend()

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()
```

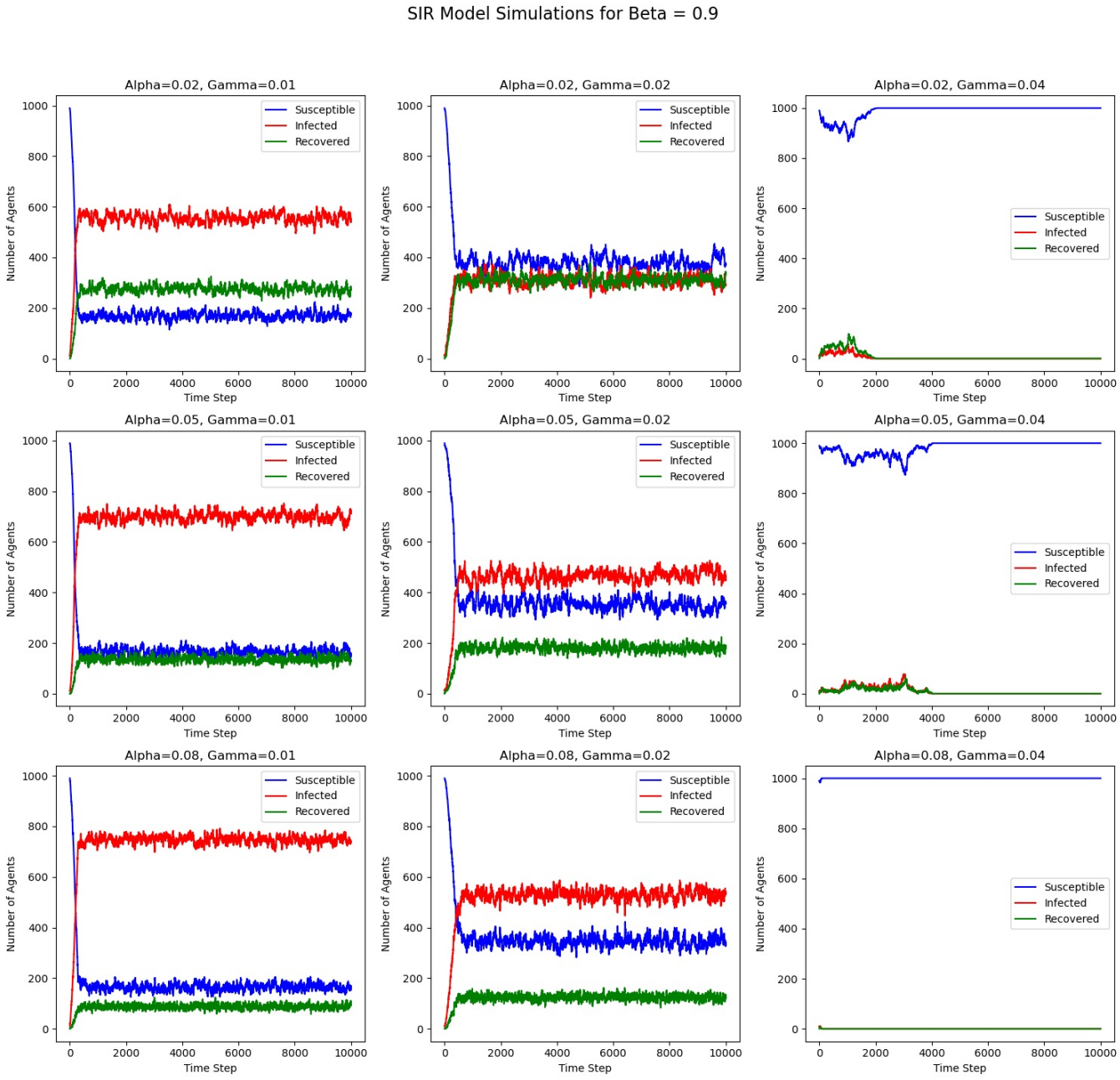


SIR Model Simulations for Beta = 0.5



SIR Model Simulations for Beta = 0.7





In [ ]:

In [ ]:

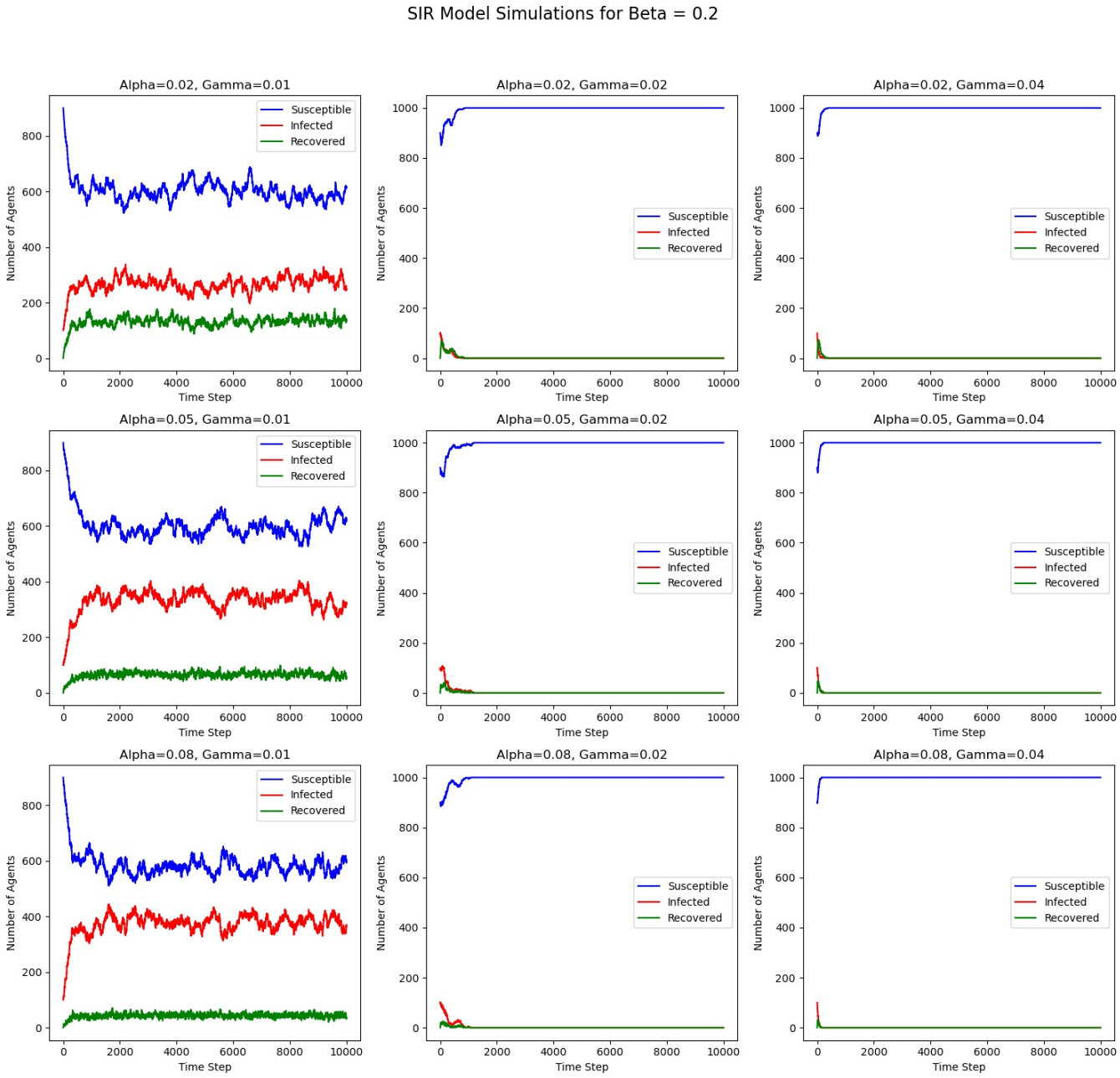
In [ ]:

In [ ]:

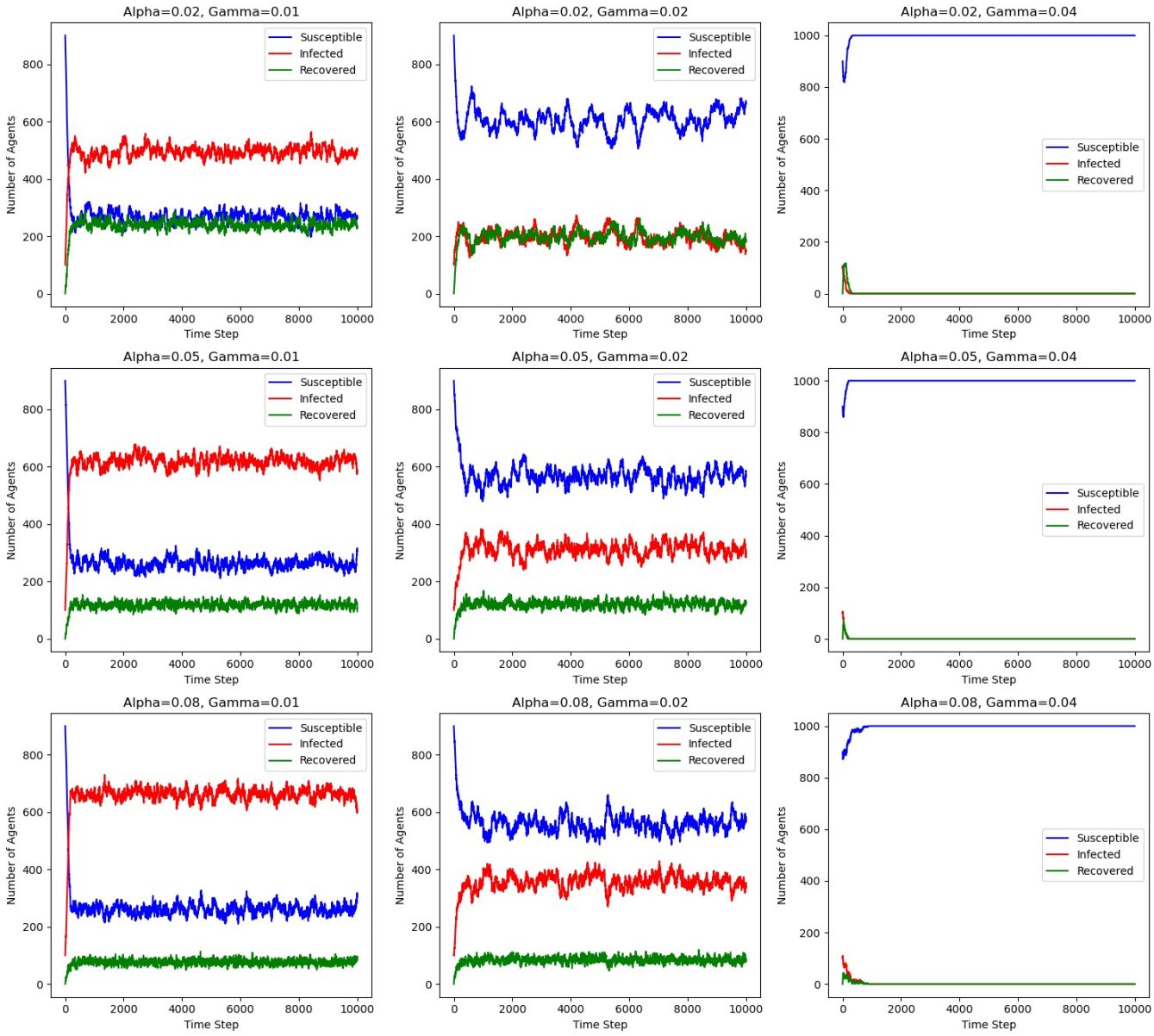
10% initially infected

In [12]:
```
initial_infection_rate = 0.1
alphas = [0.02, 0.05, 0.08]
betas = [0.2, 0.5, 0.7, 0.9]
gammas = [0.01, 0.02, 0.04]

# ONE plot for each beta, containing 9 subplots for varying beta and alpha
for beta in betas:
    fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(15, 15))
    fig.suptitle(f'SIR Model Simulations for Beta = {beta}', fontsize=16)

    for i, alpha in enumerate(alphas):
        for j, gamma in enumerate(gammas):
            susceptible, infected, recovered = run_simulation(beta, gamma, alpha, initial_infection_rate)
```

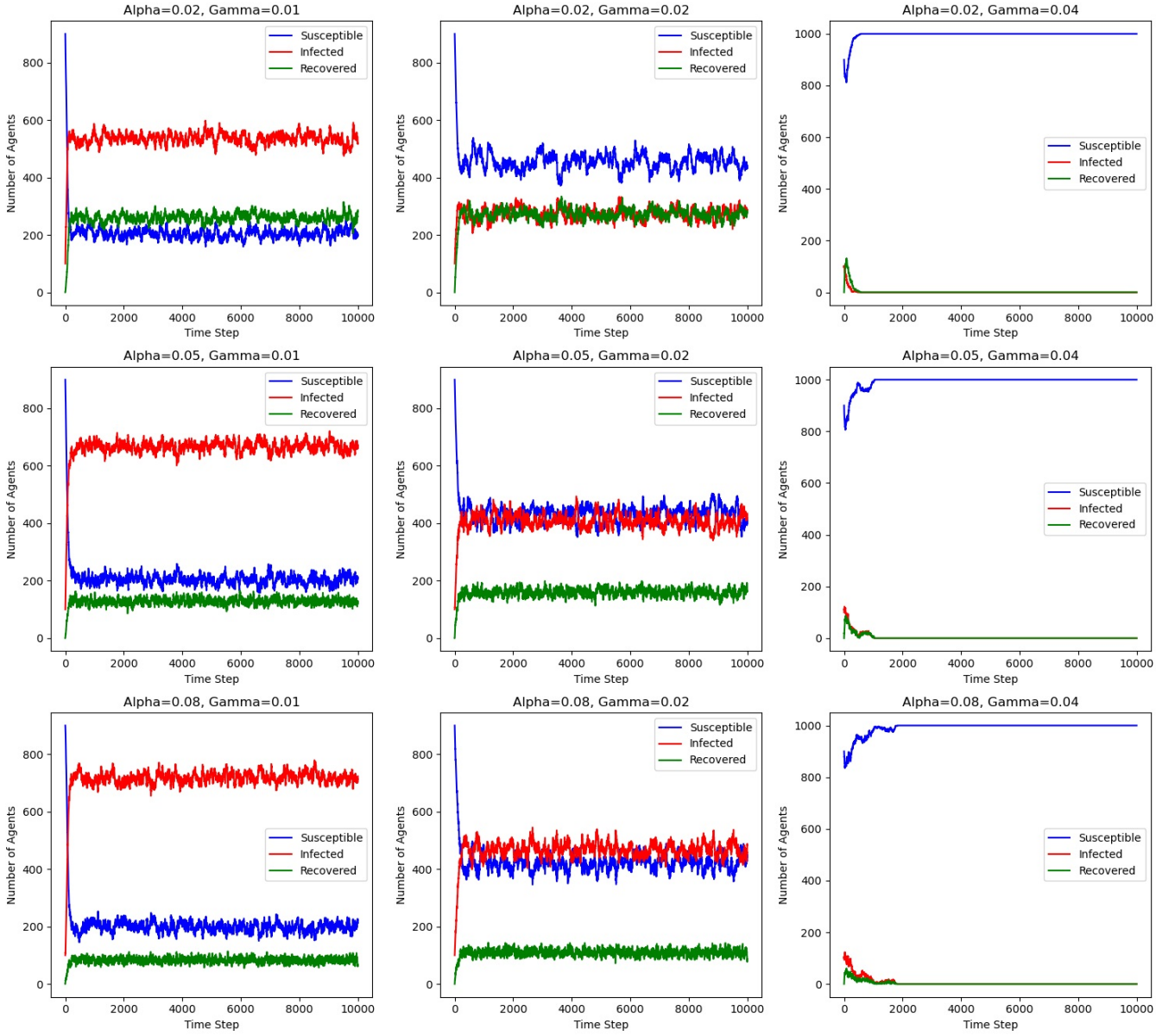
```
            ax = axes[i, j]
            ax.plot(susceptible, label='Susceptible', color='blue')
            ax.plot(infected, label='Infected', color='red')
            ax.plot(recovered, label='Recovered', color='green')
            ax.set_title(f'Alpha={alpha}, Gamma={gamma}')
            ax.set_xlabel('Time Step')
            ax.set_ylabel('Number of Agents')
            ax.legend()

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()
```

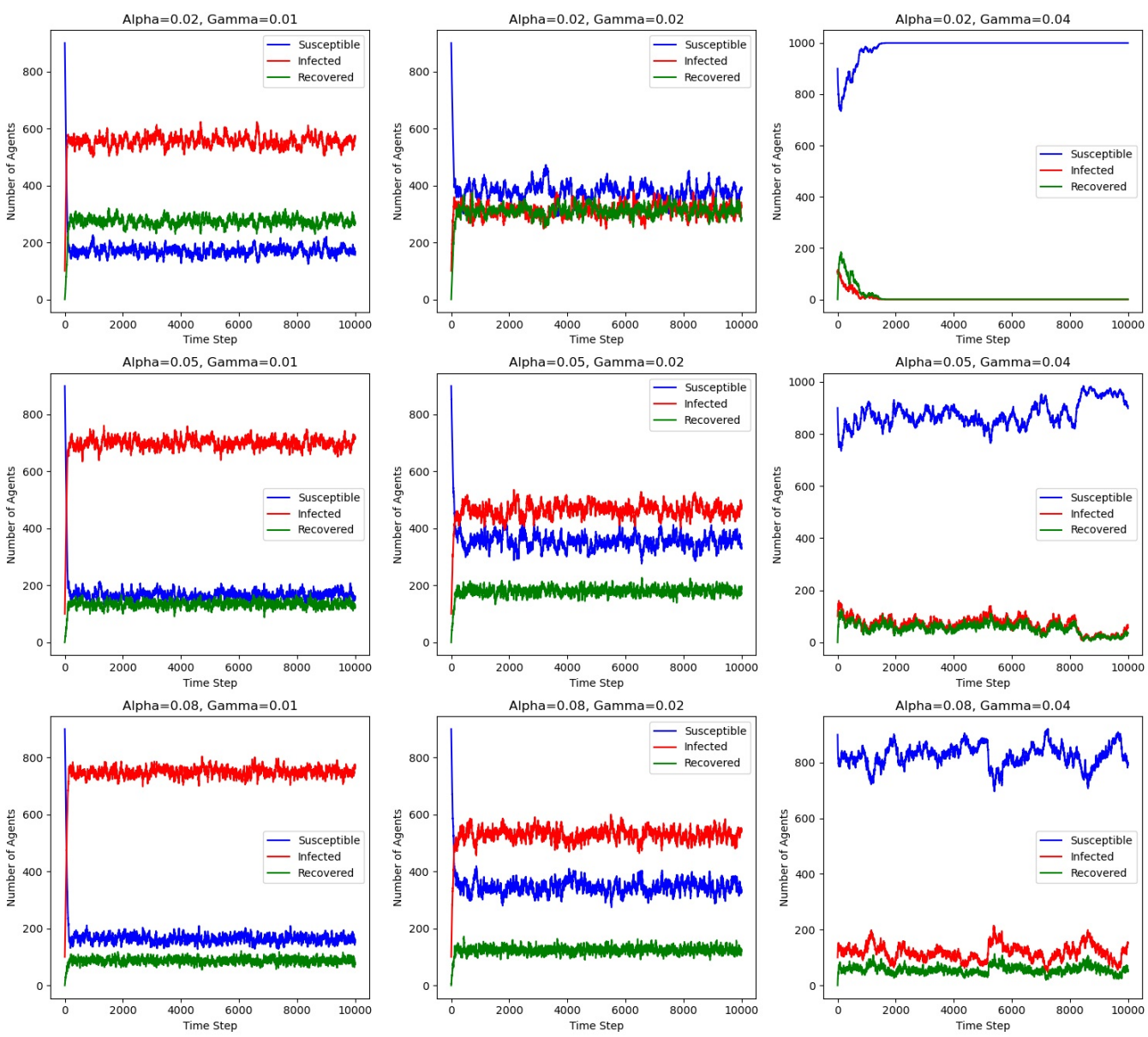


SIR Model Simulations for Beta = 0.5



SIR Model Simulations for Beta = 0.7





In [ ]: