Dangers of multi-stage governance.**

Dangers of multi-stage governance.

Multi-stage governance, i.e., a setting in which a manufacturing of a product is governed by a hierarchical chain of command, bears dangers of instability and eventually drives the system to collapse.

This can be illustrated by a simple system of ordinary differential equations.

1. **Single stage control.**
   Assume manufacturing of a product **x** (in software development, let **x** be a number of features delivered within a sprint) is controlled by a manager, who can make a decision on the speed of delivery:

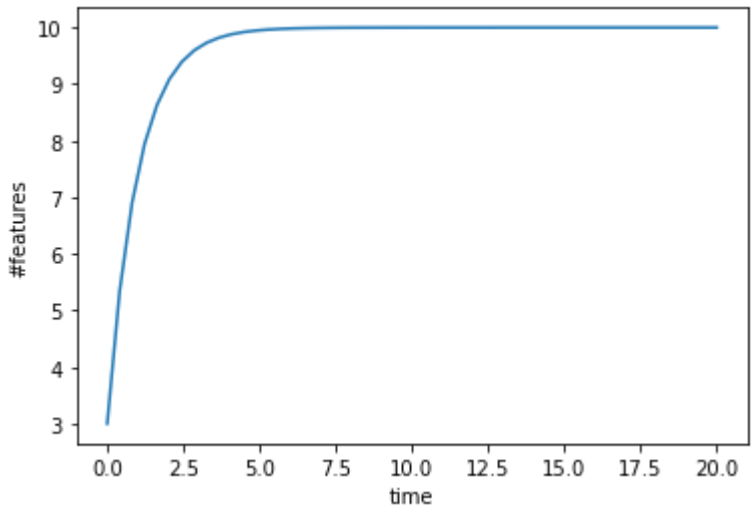
```
x′(t)  = - k(x(t) - 10),  (let k=1)
x(0) = 3
```

Here, 10 is the number of features to be delivered within a sprint. We underutilize resources if we deliver less than 10 features. On the other hand, if we deliver more, we compromise in quality.

Regardless of the initial conditions, we achieve a stable delivery with time.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def model(x,t):
    dxdt = -(x - 10)
    return dxdt

# initial condition
x0 = 3

# time points
t = np.linspace(0,20)

# solve ODE
y = odeint(model,x0,t)

# plot
plt.plot(t,y)
plt.xlabel('time')
```

```
plt.ylabel('#features')
plt.show()
```



2. **Two-stage control.**
   Let us introduce a middle man **y** who controls the speed of delivery:

```
x' = y
```

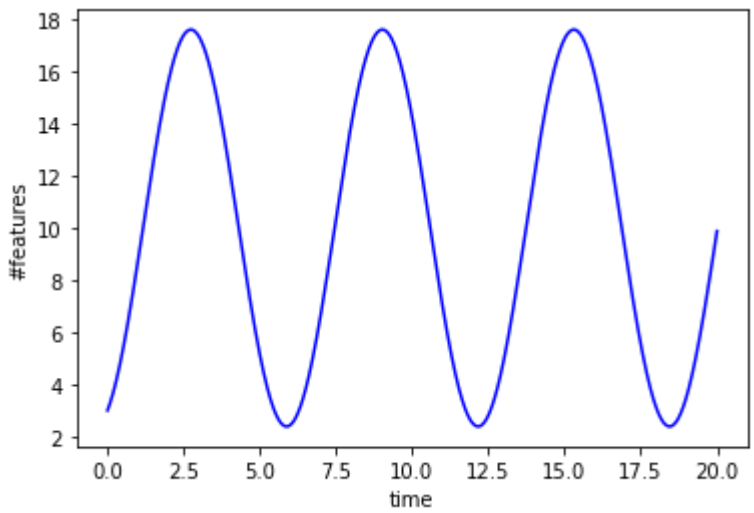
whereas the **y** is governed by a manager of higher rank, who would put pressure on **y** if the desired number of features is not achieved, or ask to slow down the delivery if the number is higher:

```
y' = -k(x-10)
```

Code:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint


# function that returns dz/dt
def model(u, t):
    x = u[0]
    y = u[1]
    dxdt = y
    dydt = -(x - 10)
    dudt = [dxdt, dydt]
    return dudt


# initial condition
u0 = [3, 3]
```

```
# number of time points
n = 1000

# time points
t = np.linspace(0, 20, n)

# store solution
x = np.empty_like(t)
y = np.empty_like(t)
# record initial conditions
x[0] = u0[0]
y[0] = u0[1]

# solve ODE
for i in range(1, n):
    # span for next time step
    tspan = [t[i - 1], t[i]]
    # solve for next step
    u = odeint(model, u0, tspan)
    # store solution for plotting
    x[i] = u[1][0]
    y[i] = u[1][1]
    # next initial condition
    u0 = u[1]

# plot results
plt.plot(t, x, 'b-', label='x(t)')
plt.ylabel('#features')
plt.xlabel('time')
plt.show()
```



On average we still produce 10 features, but the production is oscillating and is not stable.

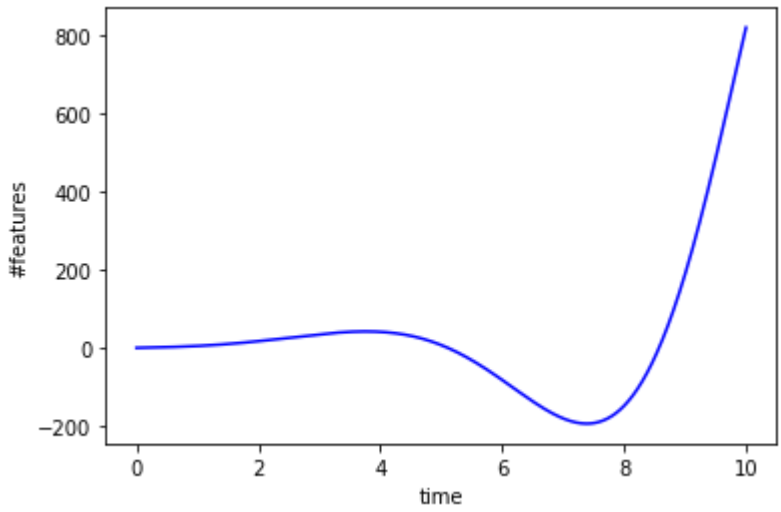
3. **Multi-stage control.**
   Let us extend the previous system with yet another player:

```
x' = y
y' = z
z' = -k(x - 10)
```

Here, **y** controls the delivery speed, **y** is controlled by **z** and **z** is controlled by higher management, which takes into account the feedback from the system.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint


# function that returns dz/dt
def model(u, t):
    x = u[0]
    y = u[1]
    z = u[2]
    dxdt = y
    dydt = z
    dzdt = -(x - 10)
    dudt = [dxdt, dydt, dzdt]
    return dudt

# initial condition
u0 = [1, 3, 1]

# number of time points
n = 1000

# time points
t = np.linspace(0,10,n)

# store solution
x = np.empty_like(t)
y = np.empty_like(t)
z = np.empty_like(t)
# record initial conditions
x[0] = u0[0]
y[0] = u0[1]
z[0] = u0[2]
```

```
# solve ODE
for i in range(1,n):
    # span for next time step
    tspan = [t[i-1],t[i]]
    # solve for next step
    u = odeint(model, u0, tspan)
    # store solution for plotting
    x[i] = u[1][0]
    y[i] = u[1][1]
    z[i] = u[1][2]
    # next initial condition
    u0 = u[1]

# plot results
plt.plot(t,x,'b-',label='x(t)')
plt.ylabel('values')
plt.xlabel('time')
plt.show()
```



The system gets out of order.

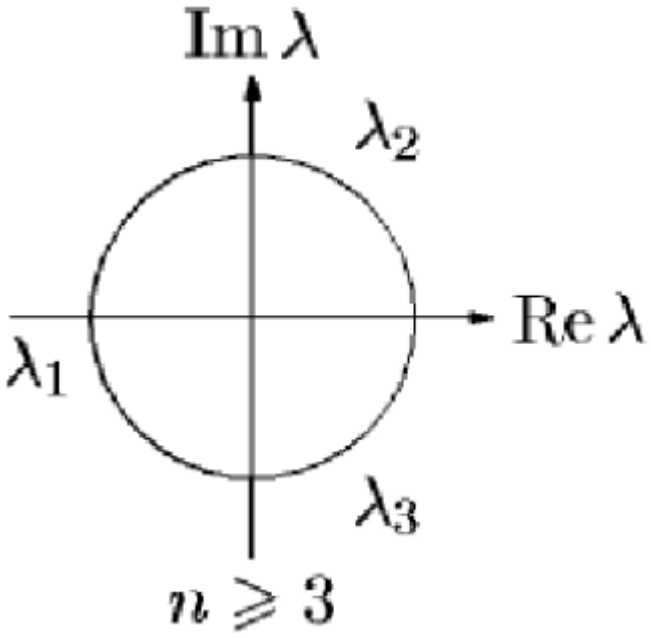
If we increase the number of control stages, the solution would remain unstable.

In general, the systems of equations can be rewritten in the form:

```
x^(n) = -k(x-N),
```

so some of the complex roots of the characteristic equation

```
\lambda^(n) = -k
```



would end up in the right half-plane.

TL;DR

Whenever you find yourself in a project following multi-stage governance governance, raise an alarm. This model would draw the project to collapse.

Why? Because the roots of the characteristic equation \lambda^(n) = -k lie in the right half-plane :)

Literature: В. И. Арнольд ««Жесткие» и «мягкие» математические модели», ISBN: 978-5-4439-2008-5.