Teste de Resiliência do Celery: Uma Abordagem Experimental com Injeção de Falhas

Pedro Lima
Universidade Federal de Pernambuco
Recife, Brazil
pccql@cin.ufpe.br

Breno Miranda
Universidade Federal de Pernambuco
Recife, Brazil
bafm@cin.ufpe.br

RESUMO

A execução confiável de tarefas assíncronas é essencial para garantir a eficiência e a escalabilidade de sistemas distribuídos em larga escala. Nesse contexto, o Celery se destaca como uma das ferramentas mais utilizadas em aplicações Python para o processamento de tarefas em segundo plano, oferecendo uma abordagem prática e direta para a construção de sistemas robustos. No entanto, sistemas de filas de tarefas como o Celery frequentemente enfrentam gargalos de desempenho que só se manifestam sob cargas reais de trabalho. Diante disso, os testes de resiliência surgem como uma estratégia eficaz para antecipar possíveis falhas, por meio da injeção controlada de erros e da análise do comportamento resultante do sistema. Atualmente, o Celery não oferece suporte nativo para esse tipo de teste, o que dificulta a análise de seu comportamento em ambientes críticos. Nesse cenário, este trabalho propõe uma avaliação experimental da resiliência do Celery por meio da injeção de falhas controladas. Os resultados indicam que, com as configurações adequadas, a ferramenta é capaz de manter a continuidade do processamento e demonstrar um bom nível de resiliência mesmo sob alta demanda, evidenciando seu potencial como uma solução robusta e confiável para a execução de tarefas assíncronas.

PALAVRAS-CHAVE

Resiliência de Sistemas, Injeção de Falhas, Tarefas Assíncronas

1 Introdução

Em sistemas de grande escala, a execução de tarefas assíncronas e distribuídas é fundamental para garantir a eficiência e escalabilidade das operações, principalmente as que exigem o processamento de grandes volumes de dados ou a execução de tarefas críticas. Para garantir isso, muitos sistemas adotam ferramentas de fila de tarefas que provêm um bom balanceamento de carga e processamento contínuo sem sobrecarregar os recursos do sistema [12].

O Celery é uma das ferramentas open-source mais populares para a execução de tarefas distribuídas e assíncronas em Python, atualmente contando com mais de 25 mil estrelas no GitHub [4]. Se destaca por permitir a execução de tarefas em segundo plano de uma forma prática e escalável, sendo uma tecnologia amplamente utilizada em ambientes de produção para realizar tarefas críticas, como envio de e-mails em massa, processamento de dados em lotes e outras operações que exigem alta performance e confiabilidade. No entanto, apesar de sua robustez e flexibilidade, não possui ferramentas nativas para testes de resiliência ou para simular falhas em tempo real, o que dificulta a avaliação do seu comportamento quando exposto a condições adversas.

O objetivo deste trabalho é investigar a resiliência e o desempenho do Celery quando exposto a falhas controladas. Para alcançar esse objetivo, avaliamos o Celery em ambientes que simulam condições adversas, como falhas de rede, exaustão de memória e sobrecarga de CPU, com o intuito de observar o impacto dessas falhas na execução das tarefas e no comportamento da ferramenta. Durante a avaliação, monitoramos e analisamos métricas essenciais, como o tempo de execução das tarefas, taxa de processamento, taxa de sucesso, além de parâmetros relacionados à eficiência do sistema, como o número de tarefas em execução e pré-carregadas no worker.

Através da coleta dessas métricas, foi possível avaliar o desempenho do Celery sob diferentes cenários de falha, identificar gargalos no processamento das tarefas e testar a capacidade de recuperação do sistema em situações de sobrecarga ou falha. Nosso estudo fornece uma análise detalhada de como o Celery se comporta em um ambiente de produção simulado, oferecendo percepções sobre sua confiabilidade e eficiência em contextos de alta demanda e falhas inesperadas.

O restante deste trabalho está organizado da seguinte forma: Na Seção 2, apresentamos a fundamentação teórica; A metodologia adotada é descrita na Seção 3; Na Seção 4, apresentamos os resultados obtidos; Na Seção 5, discutimos as implicações e limitações; Finalmente, na Seção 6, concluímos o trabalho e discutimos as perspectivas futuras.

2 Fundamentação Teórica

2.1 Testes de resiliência

São testes que fazem parte de uma abordagem que visa avaliar a resiliência de sistemas introduzindo falhas intencionais em ambientes monitorados, avaliando as consequências disso no sistema em questão. Essa prática, também conhecida como Chaos Testing [11], surgiu da necessidade de entender como os sistemas reagem a falhas inesperadas e garantir que eles possam continuar operando de maneira confiável mesmo sob condições adversas e inesperadas. O princípio base dessa área é realizar experimentos em ambientes controlados, podendo ser até em ambientes de produção e pré-produção, e identificar pontos fracos, gargalos e efeitos cascatas indesejados para melhorar a robustez do sistema [6].

Alguns ganhos notáveis do uso dessa prática são:

- Identificar falhas antes que ocorram em produção.
- Melhorar a confiança na resiliência do sistema.
- Ajudar equipes a desenvolverem estratégias eficazes de recuperação do sistema.

A popularidade dessa prática veio à tona com a Netflix, que criou o Chaos Monkey para simular falhas em sua infraestrutura na nuvem, e a partir desse experimento muito original, o conceito evoluiu

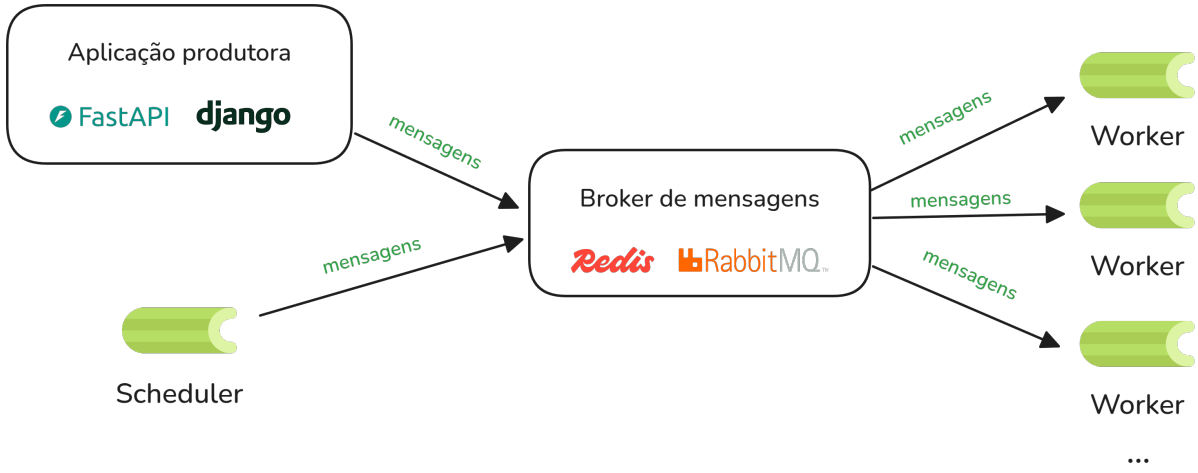


Figure 1: Funcionamento de um sistema com Celery

para um conjunto mais amplo de práticas que são conhecidas como Chaos Engineering [1].

2.2 Celery

Celery é um sistema open-source de fila de tarefas assíncronas que permite distribuir a execução de tarefas em múltiplos workers. Ele é amplamente utilizado para processar tarefas em background das mais diversas aplicações, melhorando a escalabilidade de sistemas e lidando com workloads intensivos de processamento de dados, por exemplo.

O funcionamento do Celery se baseia em três componentes principais conforme ilustrado na Figura 1:

- Aplicação produtora: Cria e envia as tarefas para a fila. Geralmente, é uma aplicação backend, como uma API, que utiliza o Celery para executar tarefas em segundo plano, sem bloquear o fluxo principal.
- Broker: O sistema de mensagens que armazena as tarefas criadas pela aplicação produtora ou pelo scheduler de tarefas, e as encaminha para os workers. Os mais usados são o Redis [10] e RabbitMQ [2].
- Workers: São os processos que consomem as tarefas armazenadas no broker e executam as funções definidas pela aplicação produtora.

É um sistema idealizado para lidar com grandes volumes de tarefas assíncronas de maneira eficiente, distribuindo a carga de processamento entre vários workers. Também suporta dois tipos de processamento de tarefas: tempo real e agendamento.

2.3 Como Falhas Podem Impactar o Celery

Sistemas de processamento assíncronos estão sujeitos a diferentes tipos de falhas inesperadas que podem impactar o comportamento e resultados da aplicação, por isso é importante testar a resiliência desse tipo de sistema diante de situações inesperadas do mundo real.

Alguns pontos críticos de impacto que um sistema com Celery pode sofrer são:

- Exaustão de recursos: Como qualquer sistema, ele depende dos recursos disponíveis em seu ambiente para operar de forma esperada, e em casos de limites ultrapassados, como uso excessivo de CPU, memória, disco ou tempo de execução, é comum ocorrer falhas ou interrupções no funcionamento dos workers e, consequentemente, no processamento de tarefas.
- Perda de mensagens: Sob condições adversas, falhas de workers podem resultar na perda de mensagens, o que é perigoso para aplicações críticas.
- Observabilidade: Por processar as tarefas de forma assíncrona, problemas de latência, perda de mensagens ou falhas intermitentes podem passar despercebidos sem uma observabilidade adequada, comprometendo a capacidade de diagnosticar e resolver problemas rapidamente.

3 Metodologia

3.1 Arquitetura da Avaliação

A arquitetura da avaliação, ilustrada na Figura 2 foi desenhada para simular e testar o comportamento de um sistema Celery experimental sob diferentes condições de carga e falhas. Para isso, é feito o uso do Docker Compose, que orquestra múltiplos containers Docker, onde cada serviço tem sua responsabilidade na avaliação:

(1) Celery Worker: É quem está processando as tarefas experimentais projetadas para simular diferentes tipos de uso de recursos em um sistema do mundo real.
(2) Celery Beat: Responsável pelo disparo das tarefas agendadas para o worker de forma periódica.
(3) Redis [10]: O broker de mensagens configurado para o Celery.
(4) Chaos Service: É uma API desenvolvida com FastAPI [9] que contém endpoints específicos de injeção de falhas no sistema Celery e seus componentes.
(5) Flower [7]: Ferramenta de monitoramento em tempo real do Celery, disponibilizando métricas como tempo de execução,

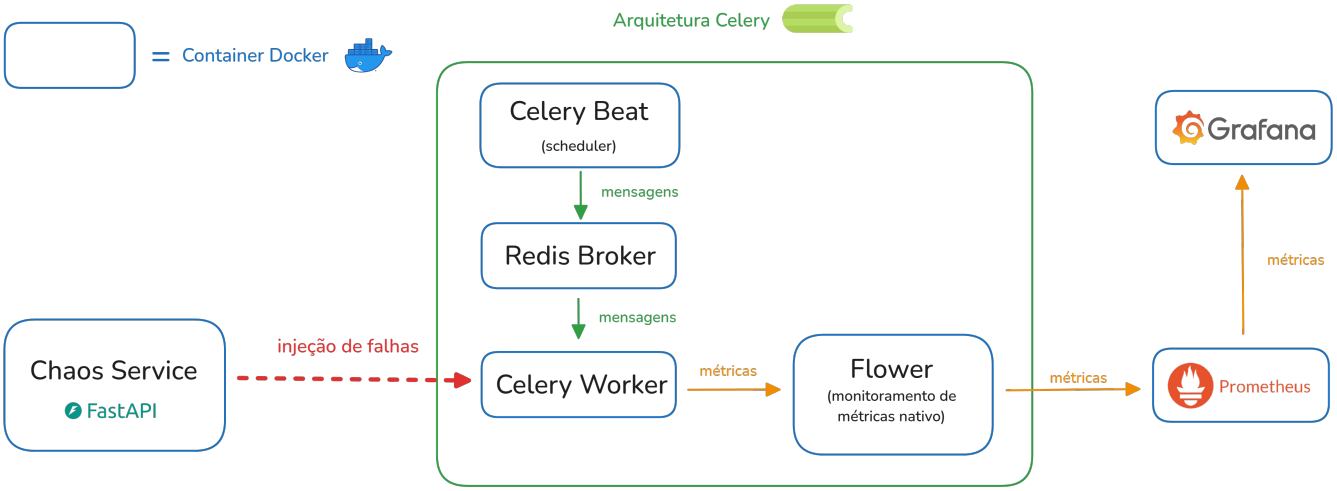


Figure 2: Fluxograma da arquitetura da avaliação

taxa de sucesso e falha, tempo médio de resposta, e estado dos workers.

(6) Prometheus [8]: Ferramenta que coleta e armazena métricas de desempenho do sistema, extraindo dados do serviço Flower.
(7) Grafana [5]: Utilizando os dados coletados pelo Prometheus, o Grafana é uma ferramenta usada para exibir um dashboard de observabilidade, proporcionando uma visão detalhada do comportamento do sistema.

A cada minuto, o serviço Celery Beat dispara uma tarefa pai executada pelo Celery Worker, que por sua vez adiciona todas as tarefas de simulação da carga realista do sistema na mesma fila de execução do worker utilizado.

3.2 Tarefas criadas

As tarefas que estão sendo executadas no worker do Celery foram projetadas para focarem no uso de diferentes recursos do sistema, simulando casos de uso do mundo real. Isso nos ajudará a entender como falhas dedicadas ao desgaste neste tipo de recurso afetará a realização das tarefas.

(1) Intensiva de Rede: Efetua uma requisição HTTP a uma API externa e retorna os dados extraídos. É semelhante a carga de recursos de uma consulta a uma API de pagamento para verificar a aprovação de transações em tempo real.
(2) Intensiva de Memória: Criação de uma matriz de números aleatórios de grande porte e são realizados cálculos pesados sobre essa matriz, simulando uma carga como o processamento de grandes volumes de dados.
(3) Intensiva de CPU: Realiza cálculos pesados para encontrar números primos até um limite definido, consumindo a CPU disponível para o serviço, simulando uma carga como a de efetuar cálculos complexos em simulações científicas.

3.3 Falhas Injetadas

As falhas introduzidas serão essenciais nesta avaliação experimental para avaliar o impacto de diferentes tipos de degradação de recursos no desempenho e na resiliência do sistema Celery utilizado. Vamos poder observar como o sistema lida com a recuperação e a continuidade da execução das tarefas durante a injeção destas falhas:

(1) Delay de rede no Redis: É criado um atraso na comunicação entre o broker Redis e o worker do Celery. A falha é implementada utilizando o comando `tc qdisc add`, que adiciona um delay de rede ao Redis.
(2) Exaustão de Memória: Tornamos escassos os recursos de memória livres para execução de tarefas em um worker através da ferramenta `stress` [13], que aloca constantemente memória, consumindo grande parte dos recursos disponíveis.
(3) Exaustão de CPU: Semelhante à falha de exaustão de memória, é executada a ferramenta `stress` dentro do container do worker, que irá realizar cálculos pesados e aumentar significativamente o uso da CPU disponível no serviço.

3.4 Métricas Coletadas

A coleta dessas métricas é essencial para analisar o desempenho e a resiliência do sistema porque irá nos permitir identificar gargalos e falhas no processamento de tarefas.

(1) Distribuição do Tempo de Execução da Tarefa: Mede o tempo de execução de cada tarefa, distribuindo-o para entender a variação e o comportamento da carga de trabalho de um worker.
(2) Taxa de Processamento de Tarefas: É o número de tarefas processadas por unidade de tempo, o que irá nos ajudar a avaliar a capacidade do sistema em processar tarefas.
(3) Tempo Médio de Execução da Tarefa no Worker: Calcula o tempo médio de execução das tarefas.

(4) Tempo de Pré-carregamento da Tarefa no Worker: Mede o tempo entre o momento em que uma tarefa é atribuída ao worker e o momento em que o worker começa a executá-la. Isso é importante para entender a latência no processamento das tarefas e detectar possíveis gargalos na fila.
(5) Taxa de Sucesso das Tarefas: A proporção de tarefas concluídas com sucesso em relação ao total de tarefas executadas.

Entre todas as métricas coletadas, a taxa de sucesso será considerada a métrica mais relevante para a análise de resiliência nesse estudo, pois reflete diretamente a capacidade do sistema de manter sua funcionalidade sob condições adversas. As demais métricas complementam essa análise ao indicar se o ambiente de execução realmente enfrentou situações adversas e como o sistema respondeu a elas.

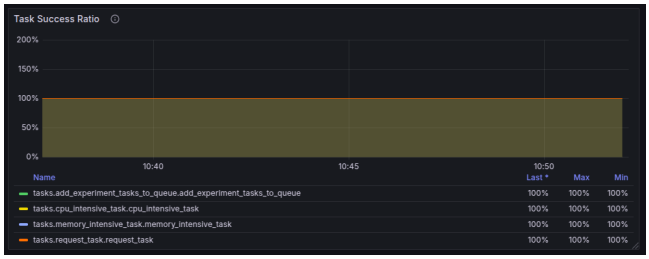
4 Resultados

4.1 Experimento baseline

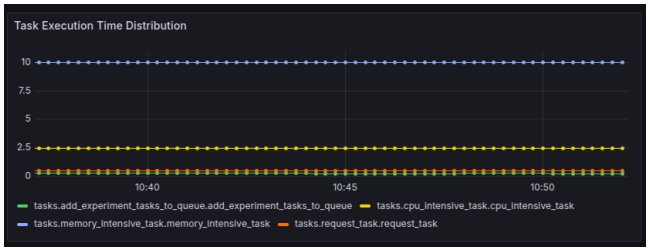
A cada 1 minuto, são geradas 20 tarefas de requisição HTTP, 2 tarefas intensivas de CPU e 2 tarefas intensivas de memória. Todas as tarefas são processadas por um único worker, que possui 1GB de memória disponível e possui nível 8 de concorrência de tarefas (baseado no número de núcleos da máquina usada). Essa configuração inicial serve como referência para entender o comportamento do sistema em termos de tempo de execução, taxa de processamento e uso de recursos, antes da introdução de falhas, e assim, temos uma visão clara de como o ambiente experimental configurado opera sem interferências externas, e o que ocorre após a injeção de um stress controlado.

Os resultados baseline são mostrados na Figura 3 e são discutidos a seguir:

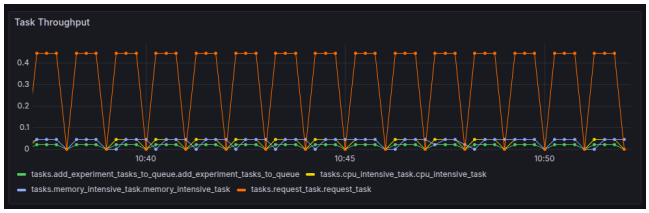
- **Taxa de Sucesso das Tarefas:** 100%
- **Distribuição do Tempo de Execução:** Bastante estável durante o experimento, com a tarefa de uso intensivo de memória se destacando como a mais demorada, seguido da tarefa de uso intensivo de CPU. Já a tarefa de requisição HTTP possui uma média de distribuição perto de 0.
- **Taxa de Processamento de Tarefas:** Resultado bem constante para o experimento, devemos lembrar que como são criadas mais tasks de requisições HTTP do que as outras, é natural que ela seja a com mais taxa de processamento por unidade de tempo, enquanto as outras ficam abaixo. As oscilações são explicadas pela frequência de criação de tasks, que é a cada 1 minuto.
- **Tempo Médio de Execução da Tarefa no Worker:** Semelhante ao resultado de tempo de execução, temos um gráfico relativamente constante demonstrando que as tarefas de requisição HTTP têm um tempo de execução muito rápido, enquanto as tarefas intensivas de CPU e memória têm tempos de execução mais altos, principalmente a tarefa de memória, como esperado devido à maior complexidade computacional exigida nela.
- **Tempo de Pré-carregamento da Tarefa no Worker:** O pré-carregamento das tarefas está eficiente, com tempos muito baixos (próximos de 0ms) para as tarefas de requisição HTTP e CPU, enquanto a tarefa de memória apresenta uma



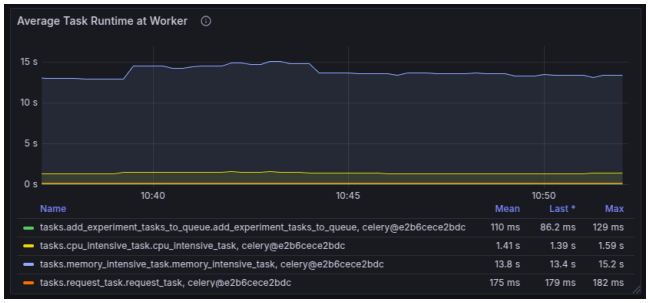
(a) Taxa de sucesso das tarefas



(b) Distribuição do tempo de execução



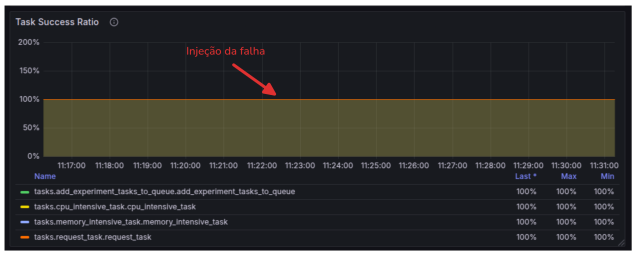
(c) Taxa de processamento das tarefas



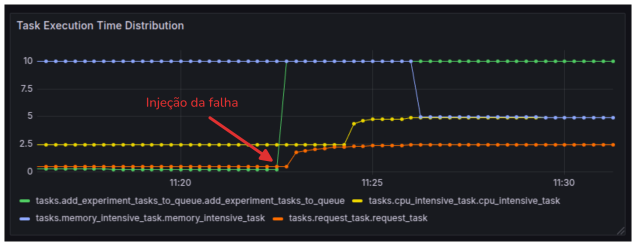
(d) Tempo médio de execução das tarefas

(e) Tempo de pré-carregamento das tarefas

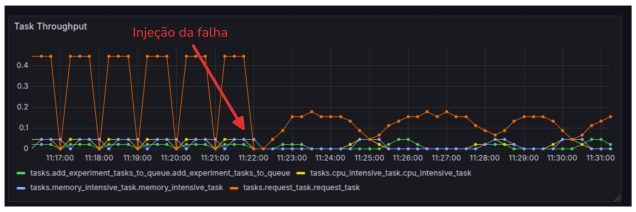
Figure 3: Resultados do experimento Baseline



(a) Delay de Rede - Taxa de sucesso das tarefas

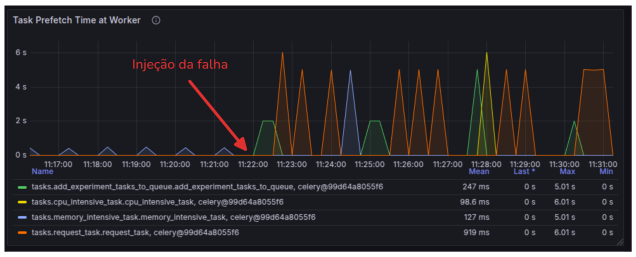


(b) Delay de Rede - Distribuição do tempo de execução



(c) Delay de Rede - Taxa de processamento das tarefas

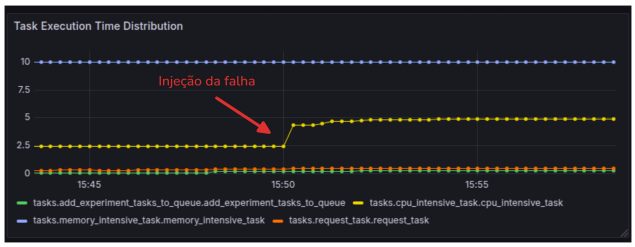
(d) Delay de Rede - Tempo médio de execução das tarefas



(e) Delay de Rede - Tempo de pré-carregamento das tarefas

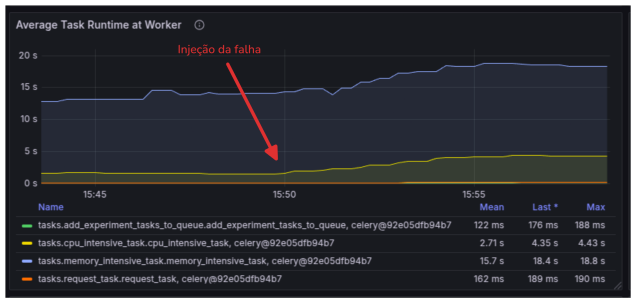
Figure 4: Resultados do experimento Delay de Rede

(a) Exaustão de CPU - Taxa de sucesso das tarefas



(b) Exaustão de CPU - Distribuição do tempo de execução

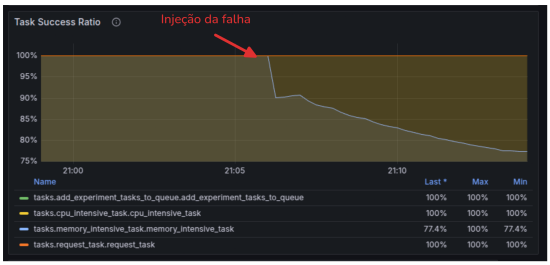
(c) Exaustão de CPU - Taxa de processamento das tarefas



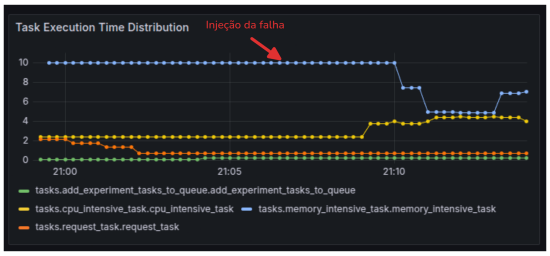
(d) Exaustão de CPU - Tempo médio de execução das tarefas

(e) Exaustão de CPU - Tempo de pré-carregamento das tarefas

Figure 5: Resultados do experimento Exaustão de CPU

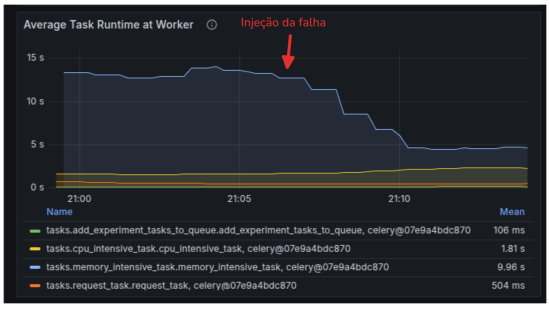


(a) Exaustão de Memória - Taxa de sucesso das tarefas



(b) Exaustão de Memória - Distribuição do tempo de execução

(c) Exaustão de Memória - Taxa de processamento das tarefas



(d) Exaustão de Memória - Tempo médio de execução das tarefas

(e) Exaustão de Memória - Tempo de pré-carregamento das tarefas

Figure 6: Resultados do experimento Exaustão de Memória

variação mais alta devido a seu tempo elevado de execução, mas ainda dentro de uma faixa razoável.

4.2 Impactos das falhas nas métricas

As falhas são injetadas individualmente após 15 minutos do início da execução da carga baseline, a fim de isolar seus efeitos no sistema. A seguir, são analisadas as consequências específicas de cada falha.

4.2.1 Delay de rede. Os resultados relacionados a Delay de Rede são mostrados na Figura 4 e são discutidos a seguir:

- **Taxa de Sucesso das Tarefas:** Se manteve em 100%.
- **Distribuição do Tempo de Execução:** É possível identificar um aumento repentino no tempo de execução das tarefas, principalmente na tarefa pai que adiciona as tarefas de simulação de carga na fila, que subiu de próximo a 0 para 10 segundos. Essa tarefa, por ser iniciada programaticamente pelo Celery Beat, necessita de uma latência boa entre as partes para um tempo de processamento curto. As tarefas de requisição HTTP e de uso da CPU também apresentam um aumento no tempo de execução, mas de forma menos aguda.
- **Taxa de Processamento de Tarefas:** Observamos uma diminuição significativa no throughput de todas as tarefas, especialmente na tarefa de requisição HTTP, e criação de um comportamento menos uniforme.
- **Tempo Médio de Execução da Tarefa no Worker:** Aumento abrupto no tempo de execução das tarefas, especialmente na tarefa pai de criação das tarefas que simulam a carga no ambiente.
- **Tempo de Pré-carregamento da Tarefa no Worker:** Aumento significativo no tempo de pré-carregamento, impactando o tempo de espera antes da execução. A injeção da falha tornou a sincronização e o gerenciamento das tarefas menos eficientes.

4.2.2 Exaustão de CPU. Os resultados relacionados a Exaustão de CPU são mostrados na Figura 5 e são discutidos a seguir:

- **Taxa de Sucesso das Tarefas:** Se manteve em 100%.
- **Distribuição do Tempo de Execução:** Aumento instantâneo na curva de distribuição de tempo apenas da tarefa de consumo de CPU, que se estabilizou conforme o tempo em um patamar um pouco mais elevado.
- **Taxa de Processamento de Tarefas:** Sem mudanças perceptíveis.
- **Tempo Médio de Execução da Tarefa no Worker:** Aumento no tempo de execução das tarefas com consumo de CPU e de memória, mas que se estabilizam com o tempo.
- **Tempo de Pré-carregamento da Tarefa no Worker:** Tarefa intensiva de memória apresentou mais picos de pré-carregamento.

4.2.3 Exaustão de memória. Os resultados relacionados a Exaustão de Memória são mostrados na Figura 6 e são discutidos a seguir:

- **Taxa de Sucesso das Tarefas:** Tarefa de consumo de memória tem uma queda imediata na taxa de sucesso devido a vários erros abruptos de encerramento do Worker, causado pelo uso de memória excedendo os recursos disponíveis no container.
- **Distribuição do Tempo de Execução:** Queda não imediata na distribuição de tempo na tarefa de memória devido a

diminuição da taxa de sucesso, e um aumento, também não imediato, na distribuição do tempo da tarefa de CPU.
- **Taxa de Processamento de Tarefas:** Taxa da tarefa intensiva de memória tem queda de perfomance imediata.
- **Tempo Médio de Execução da Tarefa no Worker:** Queda no tempo médio de execução da tarefa de memória devido a queda da taxa de sucesso, e aumento leve no de CPU.
- **Tempo de Pré-carregamento da Tarefa no Worker:** Tempo de pré-carregamento das tarefas de memória é zerado devido aos erros de terminação de worker.

5 Discussão

Acks Late. Nos resultados apresentados após a injeção da falha de exaustão de memória, foi possível observar que os impactos estavam diretamente relacionados aos erros de terminação de worker devido ao uso excessivo de memória (*Out of Memory*), e que as tarefas que estavam sendo executadas durante esses momentos foram perdidas. No entanto, a documentação oficial do Celery [3] recomenda ajustes de configuração específicos para evitar essa perda de tarefas, através da ativação das propriedades `task_acks_late` e `task_reject_on_worker_lost`. Com essas configurações ativadas, o worker só sinaliza que uma tarefa foi concluída após sua finalização, então, em caso de terminação abrupta, a tarefa segue marcada como pendente e não é perdida pelo processamento, sendo reinserida na fila. Após essa modificação, todas as métricas passaram a não ser mais afetadas pela falha de exaustão de memória, demonstrando um excelente novo resultado de avaliação da resiliência do Celery em relação a falha de memória.

Análise de impacto. Com a observabilidade criada no experimento através do Grafana, vemos que o sistema demonstrou boa resiliência quando configurado corretamente. No caso das falhas de exaustão de CPU e adição de delay de comunicação, os impactos observados foram principalmente no aumento do tempo de execução das tarefas, com uma redução no throughput e aumento do tempo de pré-carregamento, mas foram estabilizados ao longo do tempo e não trouxeram falhas críticas no sistema. A tarefa pai, responsável por adicionar as tarefas de simulação de carga à fila, foi a mais impactada, apresentando um aumento substancial no tempo de execução, indicando que a latência afetou diretamente o gerenciamento de tarefas. Apesar disso, o sistema se manteve estável, mesmo com pioras em várias métricas relacionadas à performance.

Após configuração das propriedades `task_acks_late` e `task_reject_on_worker_lost` para inibir a perda de tarefas durante falhas, os resultados mostraram significativa melhora de resiliência e estabilidade relacionados a exaustão de memória, que havia sido a falha injetada mais impactante originalmente. As tarefas que anteriormente foram perdidas devido a erros abruptos de encerramento do Worker pelos limites de memória, agora são reprocessadas sem comprometer o sistema, permitindo que o Celery se recupere sem perdas de dados e sem pioras perceptíveis no desempenho.

6 Conclusão e Trabalhos Futuros

Este trabalho teve como objetivo investigar a resiliência do Celery quando exposto a falhas controladas, especificamente falhas de rede, exaustão de memória e sobrecarga de CPU. Foi possível observar o impacto dessas falhas no comportamento do Celery por meio de diferentes experimentos e uso da observabilidade para acompanhamento dos resultados, através de métricas como tempo de execução das tarefas, taxa de processamento e taxa de sucesso.

A análise dos resultados mostrou que o Celery, sem configurações específicas, demonstrou boa resiliência, especialmente ao lidar com falhas de exaustão de CPU e latência de comunicação, que afetaram o tempo de execução das tarefas e o throughput sem comprometer o sucesso das tarefas, se mantendo estável ao longo do tempo. A implementação das configurações `task_acks_late` e `task_reject_on_worker_lost` trouxeram uma melhora significativa na resiliência diante de cenários com exaustão de memória, que era o caso mais impactante negativamente. Essas configurações permitiram que o Celery evitasse a perda de tarefas durante falhas, garantindo a continuidade do processamento e uma recuperação eficaz sem perda de dados.

Os resutados do nosso estudo demonstraram que o Celery pode ser uma ferramenta robusta e resiliente em ambientes de alta demanda e falhas controladas, com as configurações apropriadas para mitigar os efeitos de falhas inesperadas e manter a eficiência do sistema. Como grande parte das práticas adotadas de Chaos Testing em ambientes experimentais, existem limitações a partir do momento em que estamos executando simulações em um ambiente controlado e com recursos limitados, o que pode não refletir a complexidade e variabilidade de um sistema de produção real. Além disso, o número de workers foi limitado a um único worker em todos os testes, o que pode ter impactado a escalabilidade e a distribuição de carga do sistema. Futuros experimentos poderiam incluir diferentes configurações de workers e uma maior variedade de falhas para avaliar o comportamento do Celery em cenários mais complexos e diversificados.

DISPONIBILIDADE DE ARTEFATO

Os artefatos produzidos no contexto deste trabalho estão disponíveis em: https://github.com/pccql/celery_chaos.

REFERÊNCIAS

[1] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (2016), 35–41. https://doi.org/10.1109/MS.2016.60

[2] Broadcom. 2025. RabbitMQ Documentation. https://www.rabbitmq.com/docs Acesso em: 27 mar. 2025.

[3] Celery Project. 2023. Celery Documentation. https://docs.celeryq.dev/ Acesso em: 27 mar. 2025.

[4] Celery Project. 2025. Celery - Distributed Task Queue. https://github.com/celery/celery Acesso em: 27 mar. 2025.

[5] Grafana Labs. 2025. Grafana Documentation. https://grafana.com/docs/ Acesso em: 27 mar. 2025.

[6] Gremlin. 2025. What is Chaos Engineering? https://www.gremlin.com/chaos-engineering Acesso em: 27 mar. 2025.

[7] Mher Movsisyan. 2023. Flower: Celery monitoring tool. https://flower.readthedocs.io/ Acesso em: 27 mar. 2025.

[8] Prometheus Authors. 2025. Prometheus Documentation. https://prometheus.io/docs/ Acesso em: 27 mar. 2025.

[9] Sebastián Ramírez. 2025. FastAPI Documentation. https://fastapi.tiangolo.com/ Acesso em: 27 mar. 2025.

[10] Redis Project. 2025. Redis Documentation. https://redis.io/docs/ Acesso em: 27 mar. 2025.

[11] Casey Rosenthal and Nora Jones. 2020. *Chaos engineering: system resiliency in practice.* O'Reilly Media.

[12] Nagaraju Thallapally. 2025. Enhancing Distributed Systems with Message Queues: Architecture, Benefits, and Best Practices. *Journal of Electrical Systems* (3 2025). Issue Vol. 21 No. 1s (2025). https://journal.esrgroups.org/jes/article/view/8333

[13] Amos Waterland. 2008. stress - a workload generator for POSIX systems. https://github.com/resurrecting-open-source-projects/stress