**Ideia da Aplicação Simples com Python:**

1. **API de Submissão de Tarefas (Produtor):**
   1. Framework: **Flask** ou **FastAPI** (Python).
   2. Função: Recebe requisições para executar uma tarefa (ex: processar um arquivo de texto, redimensionar uma imagem simples, gerar um relatório).
   3. Ação: Publica a tarefa em uma fila de mensagens.
2. **Trabalhadores (Workers/Consumidores):**
   1. Scripts Python.
   2. Framework: **Celery** (uma biblioteca Python poderosa para tarefas distribuídas que se integra bem com message brokers).
   3. Função: Escutam a fila de mensagens, pegam uma tarefa, processam-na (ex: a lógica de contar palavras no texto, redimensionar a imagem usando Pillow, etc.) e, opcionalmente, armazenam o resultado.
3. **Fila de Mensagens (Message Broker):**
   1. **Redis** ou **RabbitMQ**. O Celery se integra facilmente com ambos. Redis é frequentemente mais simples de configurar para cenários básicos. Pode ser um container Docker ou um serviço gerenciado na nuvem.
4. **Backend de Resultados (Opcional):**
   1. **Redis** ou um banco de dados simples (como SQLite para um exemplo muito básico, ou PostgreSQL/MySQL para algo mais robusto) para armazenar o status ou o resultado das tarefas. O Celery também pode gerenciar isso.

**Projeto OpenSource Utilizado (Componentes):**

* **Flask/FastAPI:** Para a API web.
* **Celery:** Para a gestão das tarefas distribuídas e dos workers.
* **Redis/RabbitMQ:** Como message broker (e backend de resultados para Celery).
* **Docker:** Para containerizar a API e os workers, facilitando a implantação.

## **Slides da Apresentação (PPT) - Versão Python Simplificada**

### **Slide 1: Arquitetura Distribuída com Python**

**(Título do Slide: Sistema Distribuído de Processamento de Tarefas com Python)**

**Diagrama da Arquitetura:**

*(Idealmente, aqui entraria um diagrama visual. Imagine o seguinte):*

* **Cliente (HTTP Requester)** envia uma tarefa para a **API de Submissão (Flask/FastAPI)**.
* A **API de Submissão** publica a tarefa na **Fila de Mensagens (Redis/RabbitMQ via Celery)**.
* Múltiplos **Workers Python (Celery)** consomem tarefas da fila.
* Cada **Worker** processa a tarefa.
* (Opcional) O resultado/status da tarefa é armazenado no **Backend de Resultados (Redis/BD via Celery)**.
* A API poderia ter um endpoint para consultar o status/resultado da tarefa.

**Premissas de Sistemas/Programação Distribuídos Atendidas:**

* **Compartilhamento de Recursos:** A fila de mensagens é um recurso compartilhado. Os workers compartilham a carga de processamento.
* **Abertura (Openness):** Fácil adicionar novos tipos de tarefas (novas funções Celery) ou mais instâncias de workers. APIs bem definidas na API de submissão.
* **Concorrência:** Múltiplas tarefas são processadas simultaneamente por diferentes workers ou por múltiplos processos/threads dentro de um worker (configurável no Celery).
* **Escalabilidade:**
  + **Workers:** Pode-se aumentar ou diminuir o número de instâncias de workers Celery independentemente da API.
  + **API:** A API de submissão também pode ser escalada (múltiplas instâncias).
* **Tolerância a Falhas:**
  + **Workers:** Se um worker falhar enquanto processa uma tarefa, o Celery pode ser configurado para reenfileirar a tarefa para outro worker (dependendo da natureza da tarefa e da configuração).
  + **Fila de Mensagens:** Brokers como RabbitMQ ou Redis (com persistência) podem garantir que as mensagens não sejam perdidas se um worker ou a API caírem temporariamente.
* **Transparência (parcial):**
  + **De Acesso e Localização:** O cliente que submete a tarefa não precisa saber qual worker a processará ou onde ele está localizado. A API e a fila abstraem isso.
  + **De Falha (parcial):** O sistema pode tentar reprocessar tarefas, mascarando falhas transitórias de workers.

### **Slide 2: Escolha da Nuvem e Fundamentação**

**(Título do Slide: Plataforma de Nuvem: Google Cloud Platform (GCP) - Alternativa)**

**Nuvem Escolhida: Google Cloud Platform (GCP)**

*(Para variar e mostrar outra opção viável, vamos usar GCP. AWS ou Azure seriam igualmente adequadas).*

**Fundamentação da Escolha:**

1. **Excelentes Ferramentas para Python e Containers:**
   1. **Google Kubernetes Engine (GKE):** Orquestração de containers Docker de ponta para a API e os workers Celery.
   2. **Cloud Run:** Opção serverless para rodar containers (API e/ou workers) sem gerenciar servidores, pagando apenas pelo uso. Ideal para simplicidade e escalabilidade.
   3. **Compute Engine (VMs):** Para maior controle, se necessário, ou para rodar o message broker se não usar um serviço gerenciado.
   4. **Artifact Registry:** Para armazenar as imagens Docker.
2. **Serviços Gerenciados para Mensageria e Dados:**
   1. **Cloud Pub/Sub:** Serviço de mensageria global, escalável e durável, uma ótima alternativa para RabbitMQ/Redis como broker para Celery (requer adaptador, mas Celery suporta).
   2. **Memorystore for Redis:** Serviço Redis totalmente gerenciado, ideal como message broker para Celery ou como backend de resultados.
   3. **Cloud SQL:** Para bancos de dados relacionais gerenciados (PostgreSQL, MySQL) se um backend de resultados mais robusto for necessário.
3. **Foco em Desenvolvedores e IA/ML:**
   1. GCP tem uma forte reputação de ferramentas amigáveis ao desenvolvedor e integrações fáceis, especialmente para cargas de trabalho Python.
   2. Embora não seja o foco aqui, a proximidade com ferramentas de Big Data e ML do GCP pode ser um benefício futuro.
4. **Escalabilidade Global e Custo-Benefício:**
   1. Infraestrutura global que permite implantar aplicações próximas aos usuários.
   2. Modelos de precificação flexíveis, incluindo faturamento por segundo para VMs e opções serverless que podem ser muito econômicas.
5. **Monitoramento e Logging Integrados:**
   1. **Cloud Monitoring e Cloud Logging:** Ferramentas poderosas para observabilidade da API, workers e outros componentes da infraestrutura.

### **Slide 3: Benefícios da Solução Implementada**

**(Título do Slide: Benefícios do Processador de Tarefas Distribuído em Python na GCP)**

1. **Desacoplamento e Resiliência:**
   1. A API de submissão e os workers operam de forma independente. Uma falha em um worker não afeta a capacidade da API de aceitar novas tarefas. A fila de mensagens atua como um buffer.
2. **Escalabilidade Flexível e Eficiente:**
   1. Workers Celery podem ser escalados horizontalmente (adicionando mais instâncias via GKE ou Cloud Run) para lidar com aumentos na carga de tarefas, sem impactar a API.
   2. A API também pode ser escalada independentemente.
3. **Processamento Assíncrono Melhorando a Experiência do Usuário:**
   1. A API pode responder rapidamente ao cliente após enfileirar a tarefa, sem esperar que o processamento (potencialmente demorado) termine. O cliente pode ser notificado depois ou consultar o status.
4. **Simplicidade de Desenvolvimento com Python e Celery:**
   1. Python é conhecido por sua simplicidade e produtividade.
   2. Celery abstrai muitas complexidades da comunicação com message brokers e da gestão de tarefas distribuídas.
5. **Custo-Efetividade na Nuvem (GCP):**
   1. Usar serviços como Cloud Run ou GKE com autoscaling permite pagar apenas pelos recursos consumidos.
   2. Serviços gerenciados (Pub/Sub, Memorystore) reduzem a carga operacional.
6. **Manutenção e Evolução Facilitadas:**
   1. Código da API e lógica dos workers são separados, facilitando a atualização ou correção de bugs em cada parte sem afetar a outra.
   2. Novas funcionalidades de processamento podem ser adicionadas como novas tarefas Celery.
7. **Robustez com Filas de Mensagens:**
   1. Garante que as tarefas não sejam perdidas mesmo que os workers estejam temporariamente indisponíveis ou sobrecarregados. As tarefas permanecem na fila até serem processadas.

**Para a "Avaliação ao Vivo":**

Este sistema é conceitualmente mais simples de implementar. Você precisaria:

1. Desenvolver a API Flask/FastAPI (poucos endpoints).
2. Definir as tarefas Celery e a lógica dos workers.
3. Configurar o Celery para usar Redis (ou RabbitMQ).
4. Criar Dockerfiles para a API e para os workers Celery.
5. Implantar no GCP:
   1. Configurar uma instância de Memorystore for Redis.
   2. Implantar os containers no Cloud Run ou GKE.
   3. (Opcional) Configurar um balanceador de carga ou API Gateway do GCP.