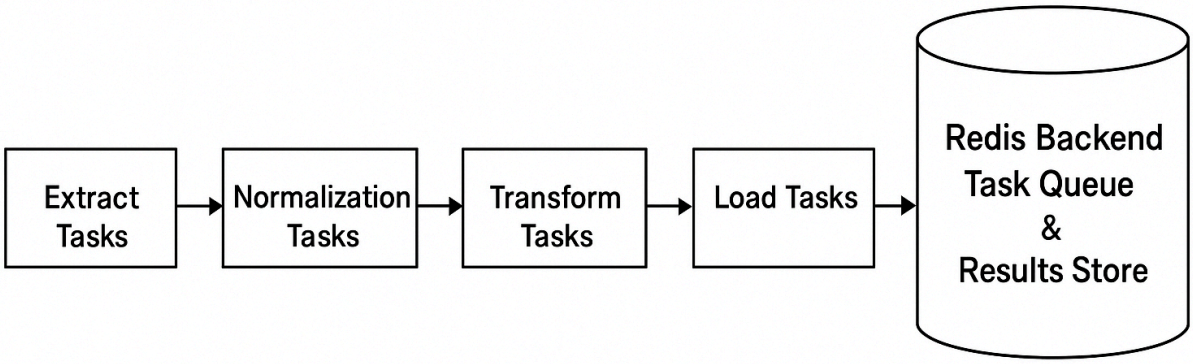


Building a Scalable Asynchronous ETL Workflow with Python, Celery, and Redis

Introduction:

PyxGen has started building out a framework to asynchronously process an ETL workflow. With the implementation of a SaaS model, there are multiple clients with multiple tasks to perform using core software services. Sequential processing would result in exponential time delays. Breaking the processing up into units of work or tasks eliminates dependencies and enables a more efficient workflow. This work started out in a Windows environment but was found to have challenging configuration and interplay issues. Switching to WSL- Ubuntu (unix platform as a windows subsystem) resolved the issues that were blocking the progress. The Pilot is now in test mode, undergoing testing of hundreds of tasks. PyxGen was also able to leverage VS Code to handle connections to the WSL instance, to utilize bash to activate sessions, start Celery worker instances and other bash tasks. Vim, a Unix text editor, was used to make

changes to existing python code. Pyxgen's existing suite of core ETL services are integrated with the new framework.

In today’s data-driven environment, small and medium-sized businesses face unique challenges when processing information from multiple sources. Traditional ETL (Extract, Transform, Load) pipelines often run sequentially, blocking main application workflows and slowing down data processing. To overcome these limitations, a design using an asynchronous ETL workflow incorporating Python, Celery, and Redis, enabled more scalable, reliable, and efficient processing.

Methods:

The primary goal was to build a system capable of ingesting, transforming, and loading data without bottlenecks. The key challenges were avoiding operations that block other processes, reliably tracking task completion and results, and maintaining flexibility to scale as the volume of data grows. Sequential ETL workflows simply could not meet these requirements. By introducing a task queue-based architecture, the workflow could be separated into discrete units of work, which Celery could execute asynchronously.

Python was the language of choice due to its versatility and the rich ecosystem of libraries. Celery served as the task manager, orchestrating the execution of ETL jobs across worker processes, while Redis functioned as both the broker and the result backend, handling message queuing and result storage. This stack is simple, robust, and widely supported, making it ideal for rapid development and reliable production use.

The architecture of the workflow revolves around three stages: extract, transform, and load. Each stage is implemented as a Celery task, which allows it to run independently and in parallel with other tasks. By defining modular, reusable functions, the workflow could scale horizontally by adding more worker processes as demand increases. For example, a simple addition task can be defined as a Celery task and executed asynchronously:

```
from celery import Celery

app = Celery(
```

```
 "etl_tasks",

 broker="redis://localhost:6380/0",

 backend="redis://localhost:6380/0"

)

app.autodiscover_tasks(["etl_tasks"])

# etl_tasks/tasks.py

from .celery import app

@app.task

def add(x, y):

 return x + y
```

Tasks are then queued and executed asynchronously using the .delay() method, with results retrieved via .get(timeout). This approach decouples task execution from the main program, allowing the system to continue processing other operations while waiting for tasks to complete.

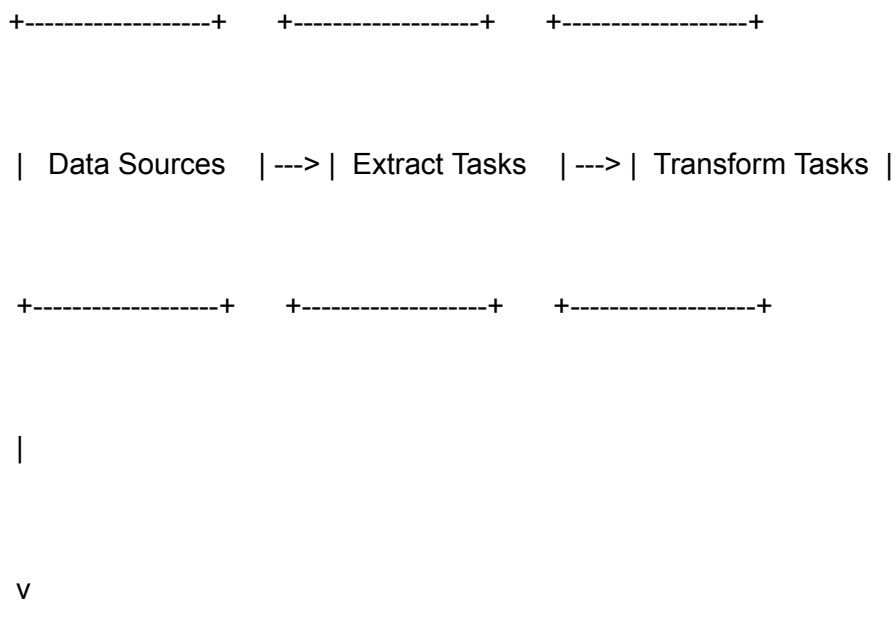
Insights:

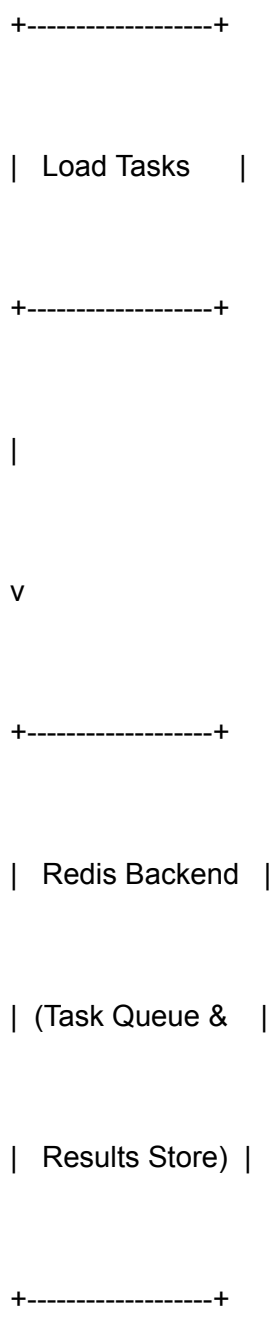
Through the process of building this workflow, several important insights emerged. First, verifying broker availability is crucial—without a running Redis server, Celery workers cannot connect, and task submission fails. Modular task design simplifies both asynchronous execution and debugging. Proper use of timeouts and retries ensures reliability in production environments, and tuning worker concurrency allows the system to efficiently utilize available resources. Monitoring tools such as logging or Flower can further improve visibility into task progress and system health.

Lessons Learned:

Implementing this workflow highlighted the value of simplicity and modularity. Small, independent tasks make troubleshooting straightforward and allow incremental testing without affecting the overall system. Additionally, asynchronous processing fundamentally changes how we think about workflow orchestration—tasks are no longer dependent on sequential completion, and the system can continue working even when certain tasks are delayed or fail. Investing time in understanding the interplay between the broker, Celery workers, and task definitions pays off in reliability and scalability.

Workflow Diagram (Text Representation):





This diagram illustrates how data flows from source to processing tasks and into the Redis-backed task queue. Each block represents a stage in the workflow, with Celery workers independently consuming tasks from the queue and writing results back to Redis.

Conclusion:

In conclusion, this architecture promises to be advantageous in designing an scalable SaaS workflow able to take on increasing numbers of clients and volumes of data in an efficient and satisfactory manner. In its implementation, PyxGen is gaining valuable insights and knowledge to help health organizations become more efficient, providing better care to their patients.