🧱Mable Pipeline Library – Architecture Report

✂️Data Structures Used and Reasoning

`Pipeline[T]`

- Core struct that holds:
- Stages (Map, Filter, Reduce, etc.)
- Worker count
- Batch size
- Metrics channel
- **Reason:** Provides a type-safe, extensible container for building data pipelines generically using Go's type parameters.

`StageMetrics`

- Struct storing metrics for each stage:

```
type StageMetrics struct {
    StageName   string
    BatchID     int
    InputCount  int
    OutputCount int
    Duration    time.Duration
    WorkerID    int
    Timestamp   time.Time
}
```

- **Reason:** Enables observability and performance tracking per stage, crucial for debugging and optimization.

Channels

- `batchChan`, `resultChan`, and `metricsChan` are used for safe concurrent communication between goroutines.
- **Reason:** Channels are Go's built-in synchronization primitive—ideal for worker pipelines.

✒️Algorithm Used and Reasoning

Batch Processing Algorithm

1. Input slice is split into chunks of configurable `batchSize`.
2. Each batch is sent to a worker goroutine.
3. Each worker executes all stages sequentially.

4. Results are collected and merged.

Worker Scheduling

- Uses a **bounded worker pool** pattern.
- A fixed number of goroutines (set by `WithWorkers()`) concurrently process batches.
- Synchronization is managed via `sync.WaitGroup`.

CPU Utilization Optimization

- The pipeline dynamically leverages **available CPU cores** by creating workers up to the number of logical CPUs.
- Each worker processes batches independently, ensuring **maximum CPU utilization** while maintaining memory efficiency.
- Batch size can be tuned based on CPU performance to achieve optimal throughput.

Reasoning:
This pattern maximizes CPU utilization while avoiding excessive goroutine creation.

Metrics Collection Algorithm

- Each worker sends timing data to a central `metricsChan`.
- A background goroutine (`emitMetrics`) consumes metrics asynchronously and stores them in ClickHouse.

🛏️Interfaces Design and Reasoning

`Stage` Interface

Each processing step implements:

```
type Stage[T any] interface {
    Name() string
    Process([]T) ([]T, StageMetrics)
}
```

- **Reason:** Enables uniform handling of all stages—Map, Filter, If, Reduce—without hardcoding logic.
- Encourages extension: new stages can be added without modifying existing code.

Pipeline Fluent API

Example:

```
p := New[int]().
    WithWorkers(5).
    WithBatchSize(100).
    Map(...).
    Filter(...).
    Collect(...)
```

- **Reason:** Fluent chaining makes the pipeline declarative and readable.

🪁Areas for Improvement & Potential Optimizations

Area	Issue	Suggested Optimization
Batch Splitting	Current split is serial	Use a concurrent batcher for very large inputs
Metrics Collection	Metrics sent for each batch synchronously	Buffer or aggregate before DB write
Memory Efficiency	Each stage allocates new slices	Reuse slices via sync.Pool
Dynamic Scaling	Worker count is static	Implement auto-scaling worker pool based on system load
Error Handling	Limited propagation between stages	Add error channels and structured error types
Serialization	JSON marshalling for ClickHouse writes	Replace with binary protocol for lower overhead

---

🌡️ Summary

The Mable Pipeline Library is a **modular, concurrent, and observable data pipeline framework** built using Go's generics, channels, and goroutines.
It dynamically uses **CPU cores and batch-based parallelism** to achieve **maximum utilization and throughput**, while maintaining scalability and low latency.
It balances **simplicity**, **performance**, and **extensibility**—suitable for both event stream processing and batch analytics workloads.

📊**Core Strengths** - Type-safe generic design - High concurrency via worker pools - Integrated observability - Fluent, chainable API - Dynamic CPU-aware batching system

⚙️ **Next Steps** - Optimize memory use with `sync.Pool` - Add error-aware pipeline stages - Introduce dynamic scaling and batch-level parallel reductions