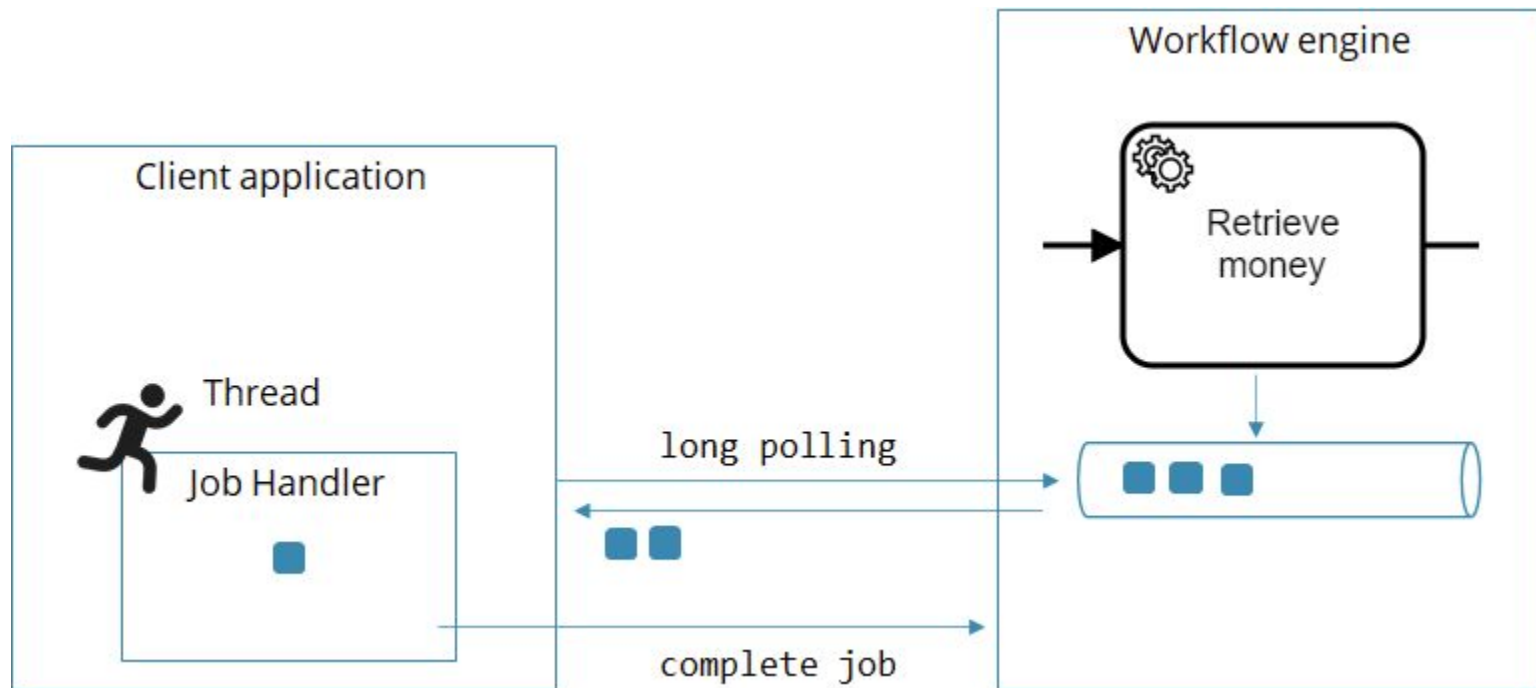# Job Workers

Long Polling vs Job Streaming

Long Polling:

- When execution **reaches a service task** with a given job type, the broker **creates a job**.
- The job is then **queued** in the broker's state until a worker of the same type activates it.
- If no worker is available, the job remains queued (it won't be lost).
- Workers use **long polling** to fetch jobs efficiently:
- A worker sends an **ActivateJobs** request: "Give me up to N jobs of type X, wait up to Y ms."
- If matching jobs are already queued ➜ the broker responds immediately.
- If not ➜ the request stays open until **a job arrives** or **the wait time expires**.
- **After activation**, the worker processes the job and **reports back** (completed / failed / retried), which updates or re-queues it.

Client application

Thread

Job Handler

Workflow engine

Retrieve money

long polling

complete job

Long Polling vs Job Streaming

Job Streaming:

- When execution **reaches a service task**, the broker **creates a job**.
- Workers **subscribe to a job type** and maintain **an open streaming connection**.
- The broker pushes jobs to **the gateway,** which **buffers** them.
- The gateway enforces **backpressure**: it only sends as many jobs as the worker can consume over the connection.
- Workers complete, fail, or retry jobs, freeing capacity in the gateway buffer, allowing more jobs to flow.
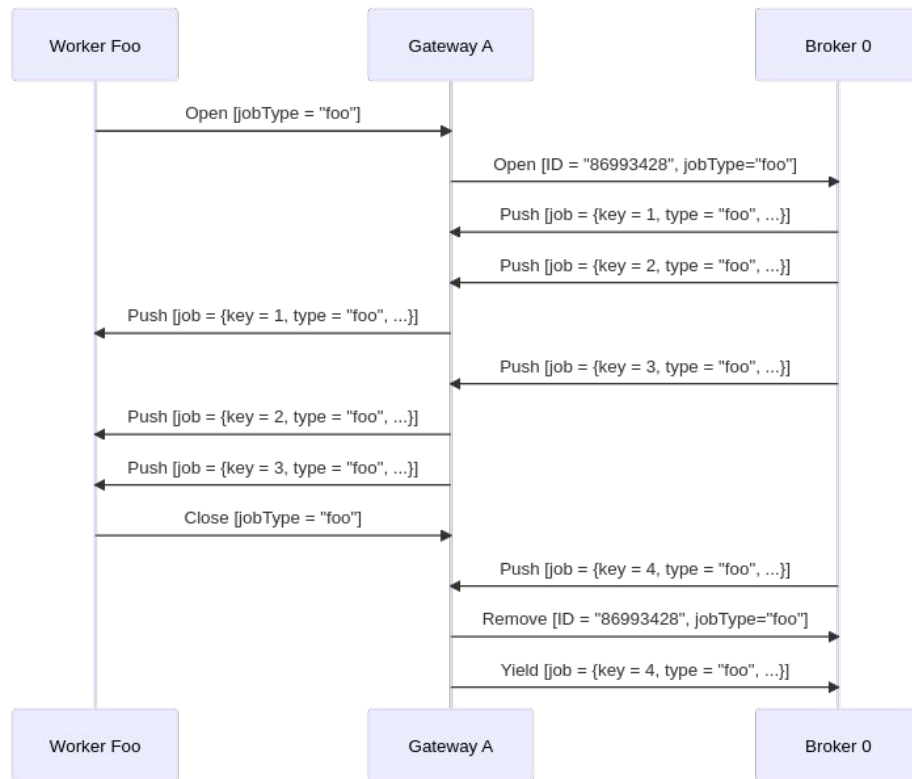
Blocking vs Non-blocking / Reactive code

```java
@ZeebeWorker(type = "rest")
public void blockingRestCall(final JobClient client, final ActivatedJob job) {
    counter.init();
    LOGGER.info("Invoke REST call...");
    String response = rest.getForObject( //
            PAYMENT_URL,
            String.class);
    LOGGER.info("...finished. Complete Job...");
    client.newCompleteCommand(job.getKey()).send()
            .join();
    counter.inc();
}
```

Non-blocking / Reactive code

```java
public class RestInvocationWorker {
    public void nonBlockingRestCall(final JobClient client, final ActivatedJob job) {
        counter.init();
        LOGGER.info("Invoke REST call...");
        Flux<String> paymentResponseFlux = WebClient.create()
                .get()
                .uri(PAYMENT_URL)
                .retrieve()
                .bodyToFlux(String.class);

        paymentResponseFlux.subscribe(
            response -> {
                LOGGER.info("...finished. Complete Job...");
                client.newCompleteCommand(job.getKey()).send()
                    .thenApply(jobResponse -> { counter.inc(); return jobResponse;})
                    .exceptionally(t -> {throw new RuntimeException("Could not complete job: " + t.getMessage(), t);});
            },
            exception -> {
                LOGGER.info("...REST invocation problem: " + exception.getMessage());
                client.newFailCommand(job.getKey())
                        .retries(1)
                        .errorMessage("Could not invoke REST API: " + exception.getMessage()).send()
                        .exceptionally(t -> {throw new RuntimeException("Could not fail job: " + t.getMessage(), t);});
            });
```

https://github.com/berndruecker/camunda-cloud-clients-parallel-job-execution/blob/main/java-worker/src/main/java/io/berndruecker/experiments/cloudclient/java/RestInvocationWorker.java