PostgresConf San Jose April 2026

Building Durable Programs with Postgres

Peter Kraft
Co-Founder

DBOS

Programs Can Break For Any Reason

- Process crashes
  - Hardware restarts
  - Resource exhaustion
- Timeouts
- Unreliable external APIs
  - Rate limiting
  - Transient failures
  - Outages (even AWS…)

AI Makes Everything Harder

- Unpredictable agent-controlled code paths
- Rate-limited AI APIs with frequent outages
- Large-scale parallel data processing increases risk of failures

How Durable Workflows Can Help

- Checkpoint your program's execution state so it can resume where it left off

Architecting Durable Workflows on Postgres

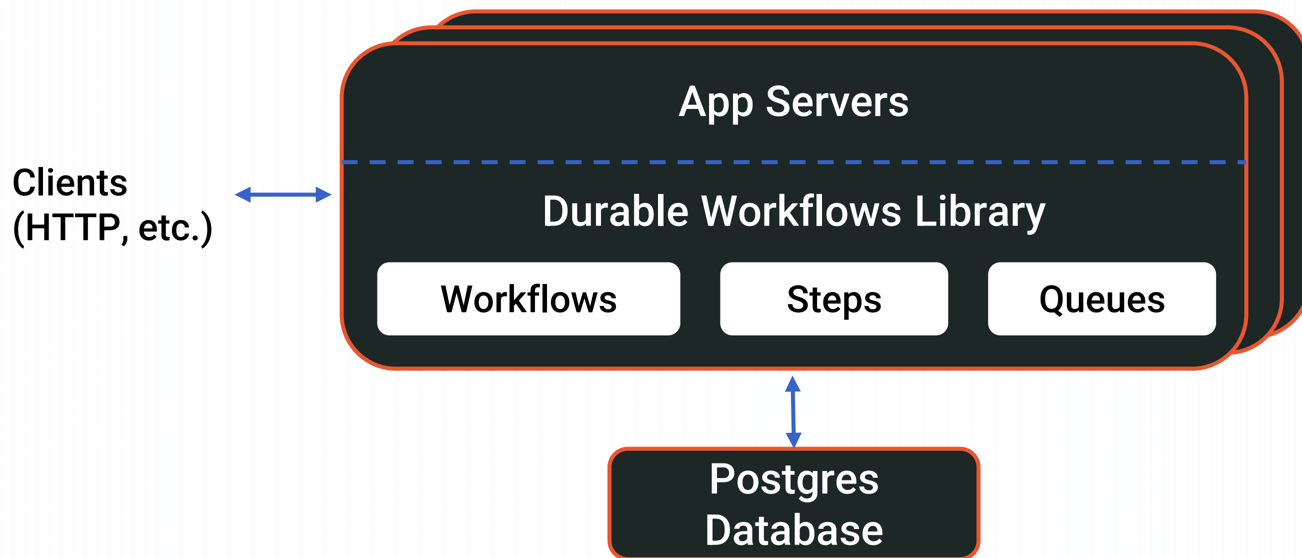
Workflow Library Interface

workflows.py

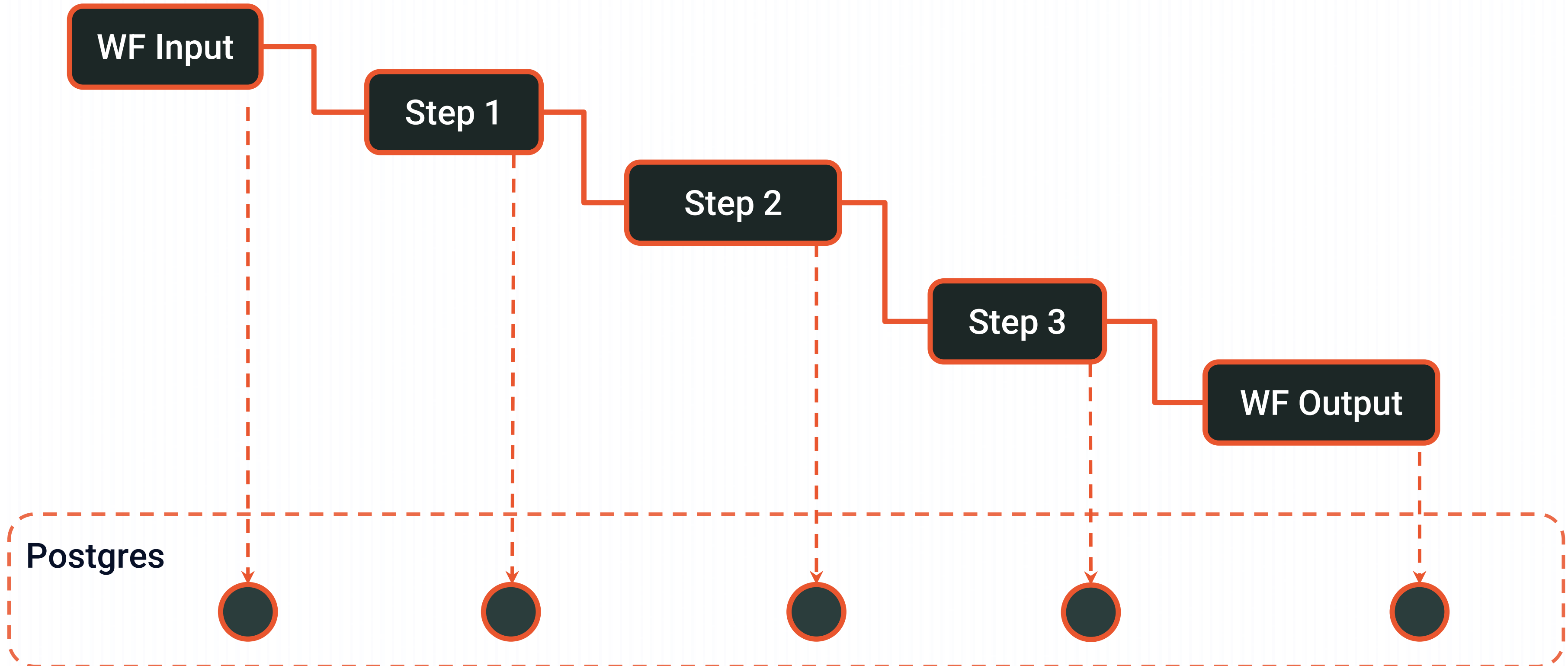
```python
@step()
def step_one():
  print("Step one completed!")

@step()
def step_two():
  print("Step two completed!")

@workflow()
def workflow():
  step_one()
  step_two()
```

- Annotate workflows and steps
- Library performs checkpointing and recovery in-process
- Store checkpoints in Postgres

Postgres-Backed Durable Workflows

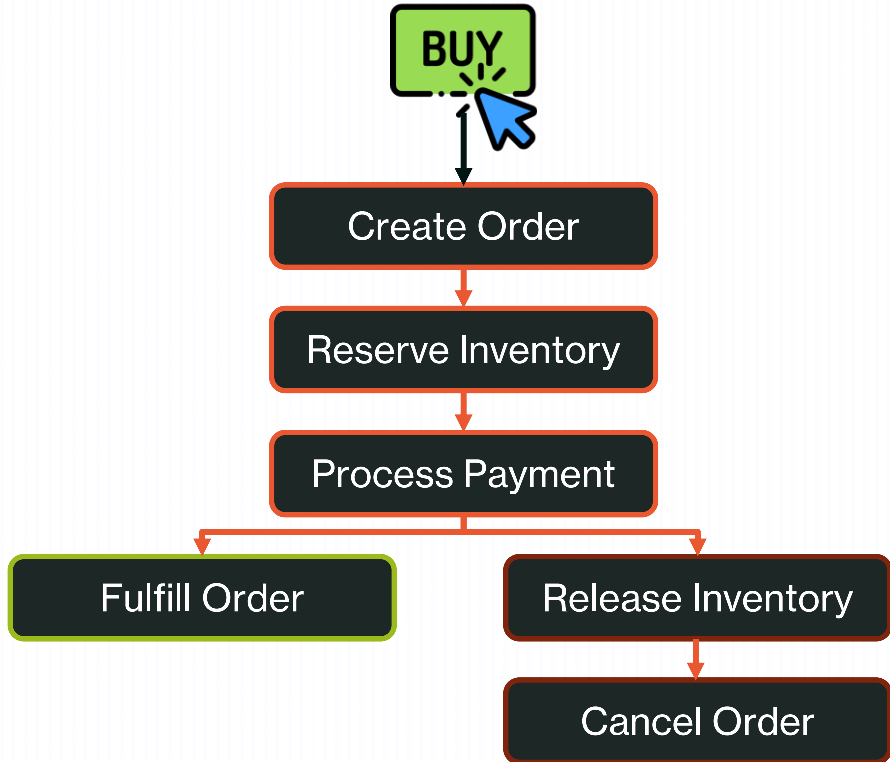


How It Works: Checkpoint State in a Database

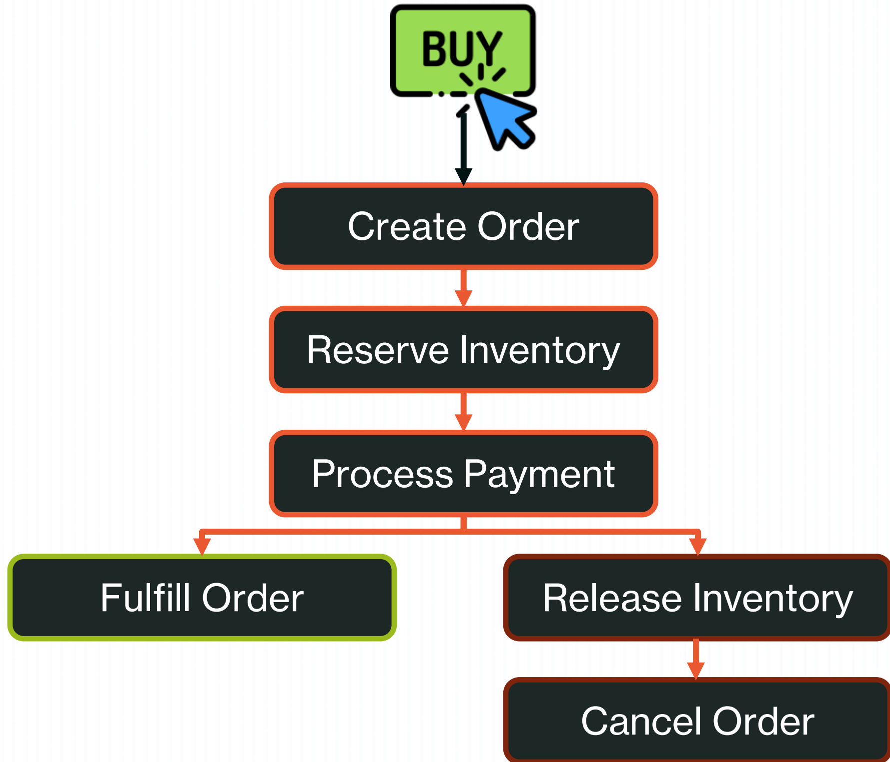


Implementing Durable Workflows on Postgres

Example Workflow: A Checkout Service



Record the Workflow Start



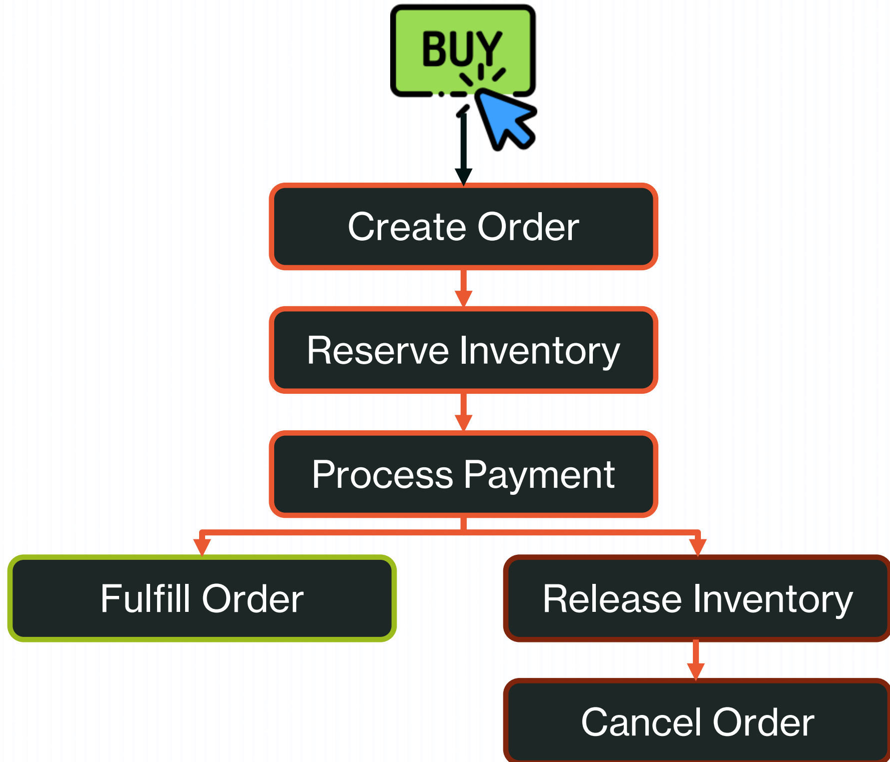

```
ID: hg4ttg2
Status: PENDING
```

Represent Workflows as Postgres Rows

workflow_status.sql

```sql
CREATE TABLE workflow_status (
    workflow_id         text   NOT NULL PRIMARY KEY,
    workflow_status     text   NOT NULL,
    workflow_name       text   NOT NULL,
    workflow_input      text   NOT NULL,
    workflow_output     text,
    workflow_error      text,
    queue_name          text,
    application_version text,
    executor_id         text,
    created_at_epoch_ms bigint NOT NULL
);
```

Checkpoint Each Step



ID: hg4ttg2
Status: PENDING

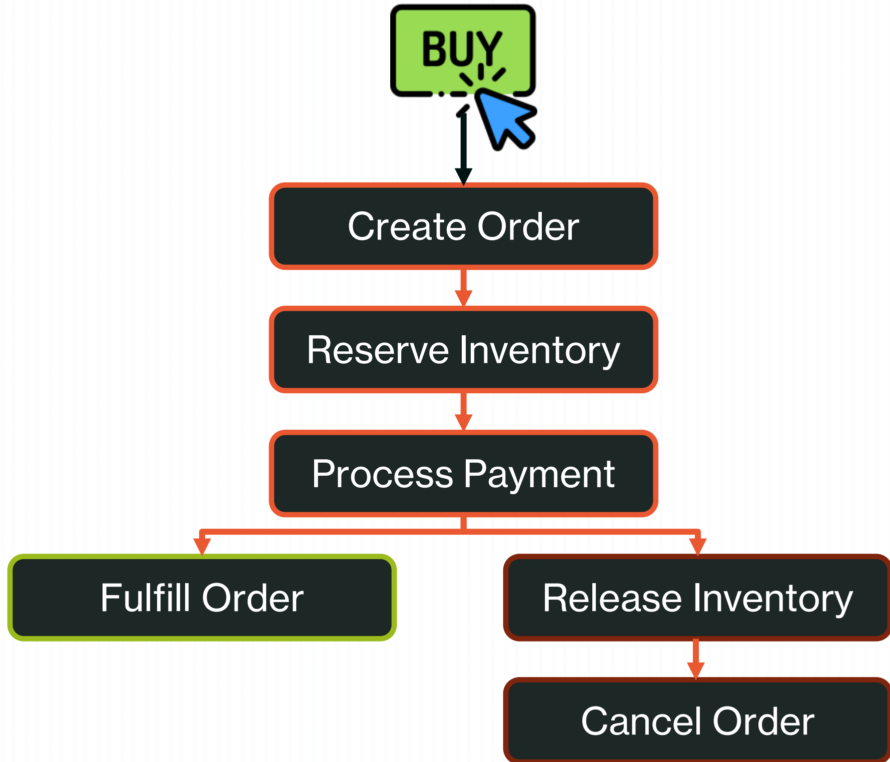
Step: 1
Output: Order Info

Represent Steps as Postgres Rows

step_status.sql

```sql
CREATE TABLE step_status (
    workflow_id            text    NOT NULL,
    step_id                integer NOT NULL,
    step_name              text    NOT NULL,
    step_output            text,
    step_error             text,
    started_at_epoch_ms    bigint,
    completed_at_epoch_ms  bigint,
    PRIMARY KEY (workflow_id, step_id),
    FOREIGN KEY (workflow_id)
       REFERENCES workflow_status(workflow_id)
);
```

Checkpoint Each Step

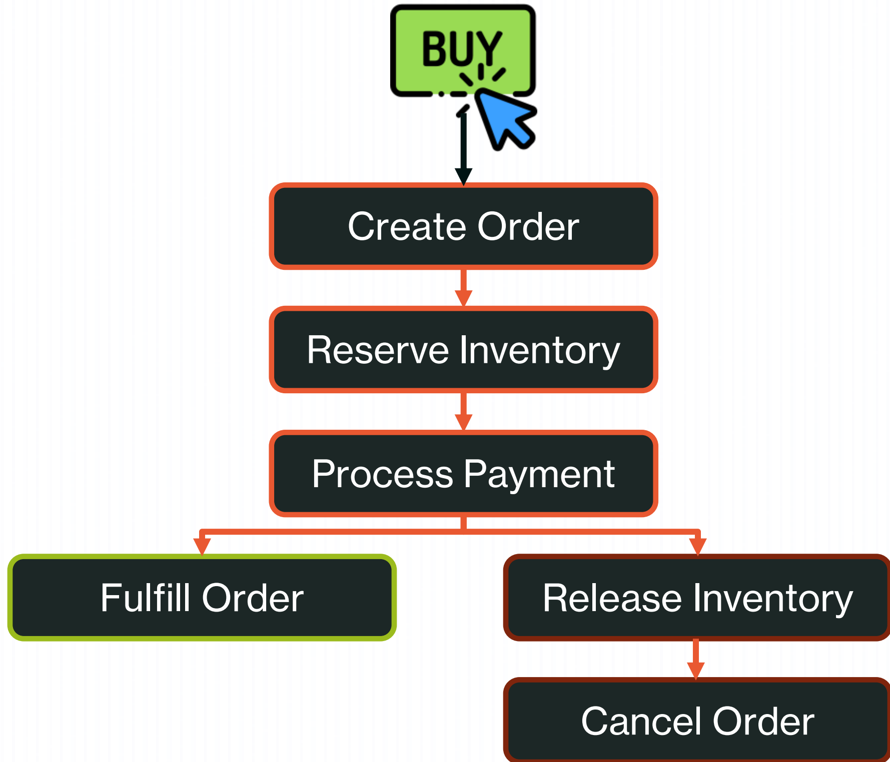


ID: hg4ttg2
Status: PENDING

Step: 1
Output: Order Info

Step: 2
Output: Reserved

Checkpoint Each Step



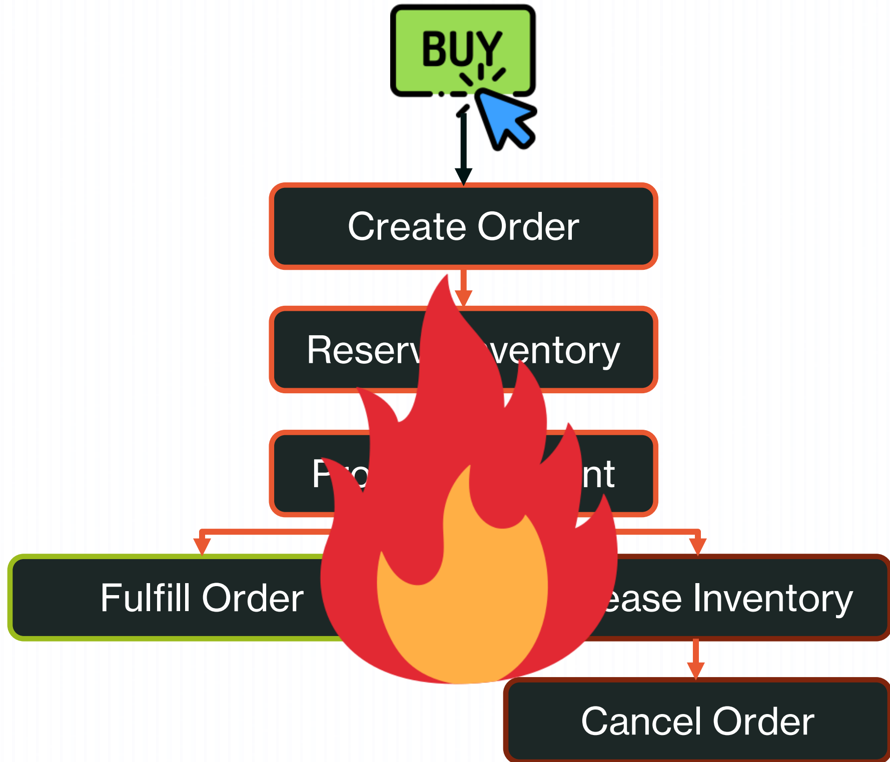
ID: hg4ttg2
Status: PENDING

Step: 1
Output: Order Info

Step: 2
Output: Reserved

Step: 3
Output: Successful

What If It Fails?



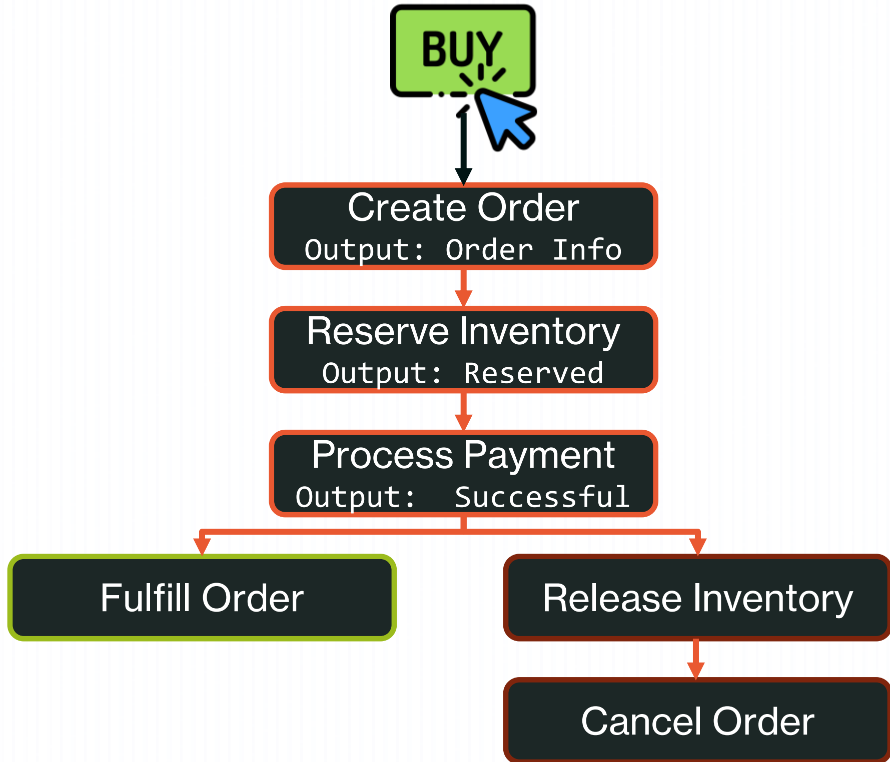
ID: hg4ttg2
Status: PENDING

Step: 1
Output: Order Info

Step: 2
Output: Reserved

Step: 3
Output: Successful

Recover From Checkpoints



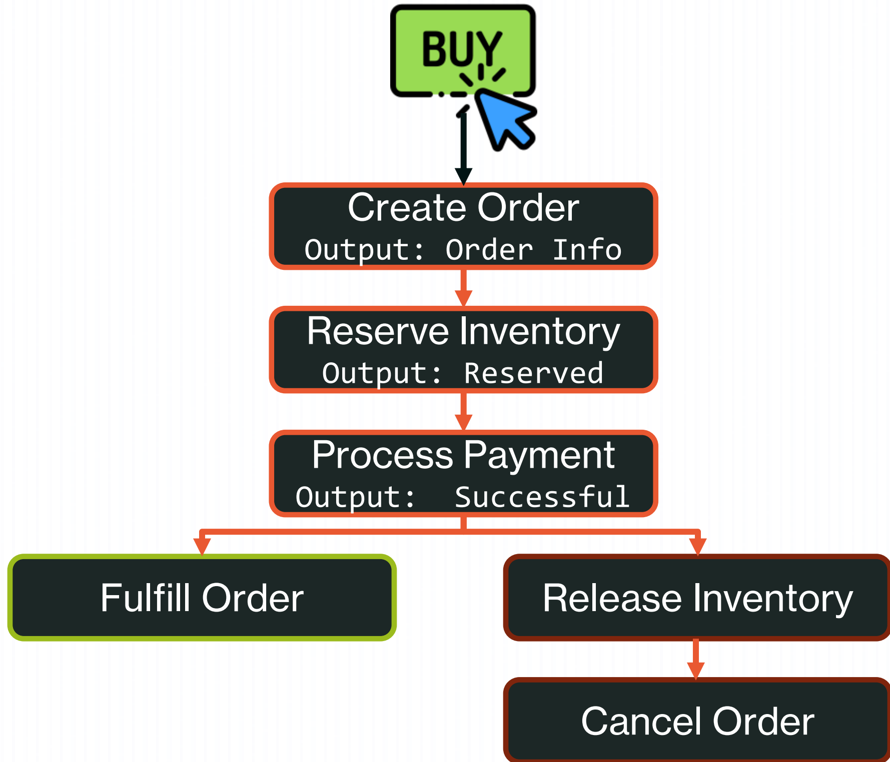
ID: hg4ttg2
Status: PENDING

Step: 1
Output: Order Info

Step: 2
Output: Reserved

Step: 3
Output: Successful

Complete the Recovered Workflow



ID: hg4ttg2
Status: SUCCESS

Step: 1
Output: Order Info

Step: 2
Output: Reserved

Step: 3
Output: Successful

Step: 4
Output: Fulfilled

Challenge: Durable Queues

Challenge: Scalable Durable Queues

- Use the `workflow_status` table as a Postgres-backed queue
- Provides durability for distributed jobs with many queued subtasks

workflow_id	workflow_name	status	input	queue_name	created_at	started_at
32bd123-45e	doc_processing	PENDING	{"items": …}	“doc_queue”	1756316230	1756316530
75d123-32b	doc_processing	ENQUEUED	{"items": …}	“doc_queue”	1756317246	-

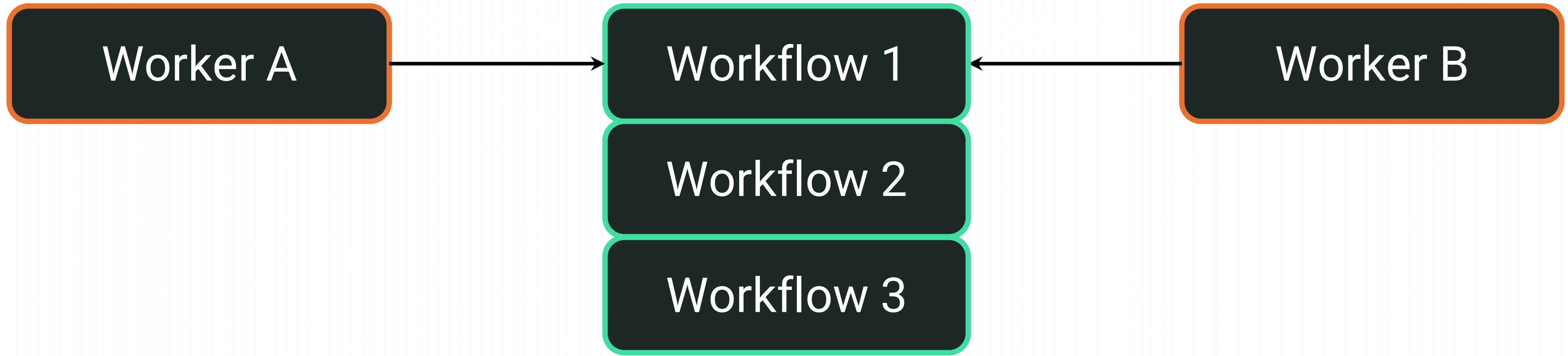
Naïve Approach: Poll the Table

queue.sql

```
SELECT *
FROM workflow_status
WHERE status = 'ENQUEUED'
ORDER BY created_at ASC
LIMIT N
```

Problem: Lock Contention in Queues

- Distributed workers pulling tasks from the same queue
- Lock contention leads to bad performance



Solution: Locking Clauses

queue.sql

```sql
SELECT *
FROM workflow_status
WHERE status = 'ENQUEUED'
ORDER BY created_at ASC
LIMIT N
FOR UPDATE SKIP LOCKED;
```

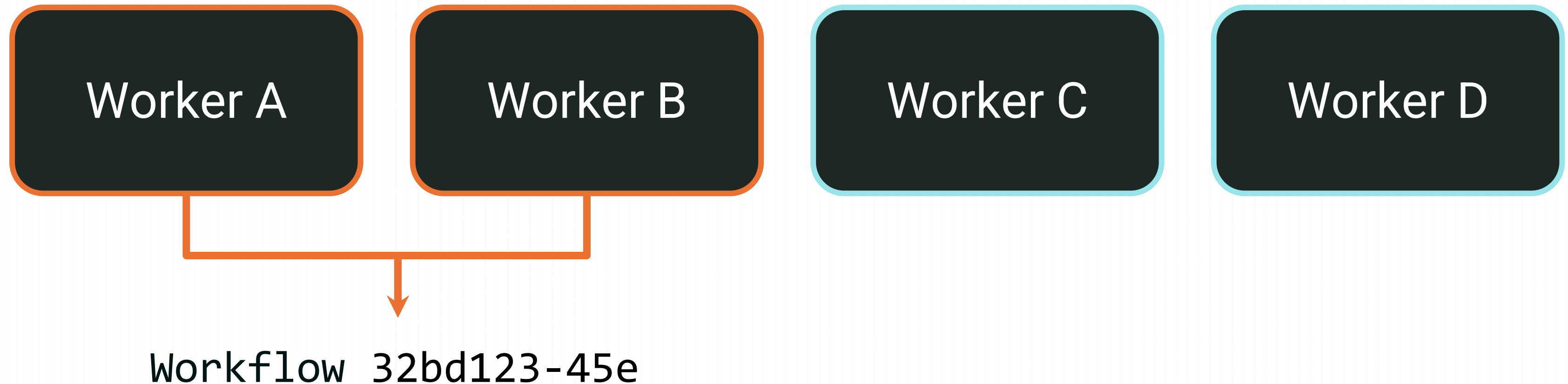
Solution: Locking Clauses

- Each worker selects rows that are not already locked

Challenge: Distributed Workflow Coordination

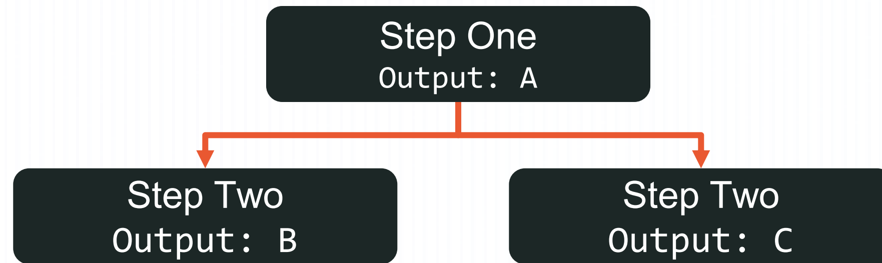
Challenge: Distributed Workflow Coordination

- In production, many distributed servers running workflows
- What happens if two servers try to execute the same workflow?



Duplicate Workflows Considered Harmful

- Duplicate workflow executions can create parallel workflow histories, violating safety guarantees



Which workflow history is real?

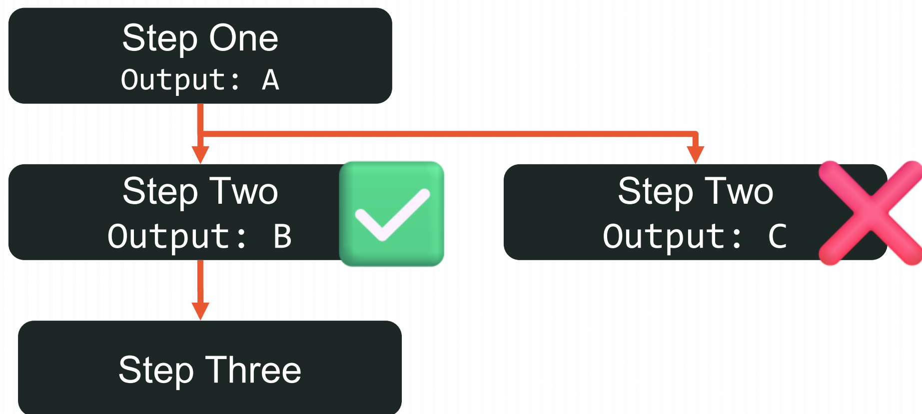
How Step Checkpointing Works

run_step.py

```
def run_step(ctxt, step_func, *args):
  # If the step has already executed, return its checkpointed output
  output = get_step_checkpoint(ctxt.workflow_id, ctxt.step_id)
  if output is not None:
    return output
  # Otherwise, execute the step and record its output
  output = step_func(*args)
  checkpoint_step(ctxt.workflow_id, ctxt.step_id, output)
  return output
```

Solution: Leverage Database Integrity

- Leverage the `UNIQUE` constraint on `(workflow_id, step_id)`
- If two servers execute the same workflow:
  - Both execute the first not-yet-checkpointed step
  - Only one can “win” and write the step checkpoint
  - The “loser” fails and backs off from workflow execution

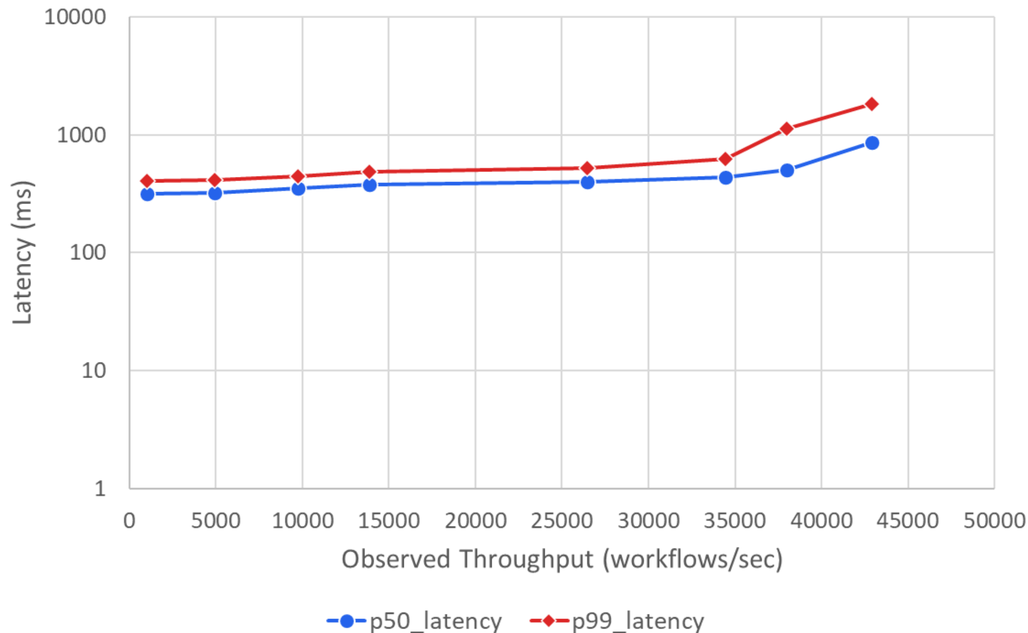


Benchmarking Postgres at Scale

Experimental Setup

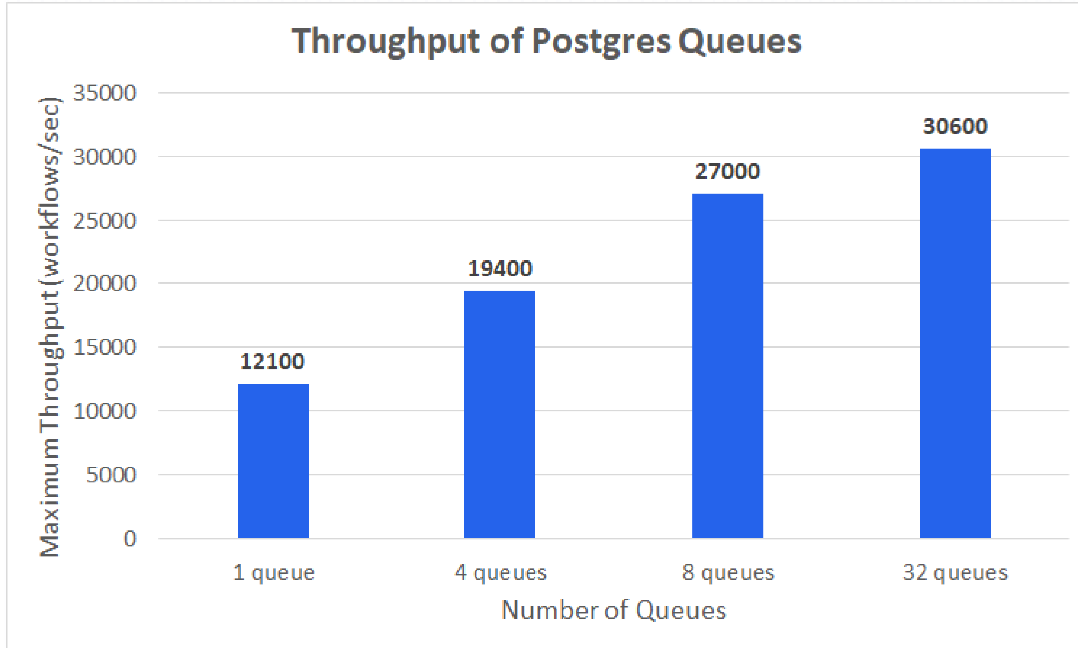
- AWS RDS `db.m7i.24xlarge` instance
  - 96 vCPUs
  - 384 GB of RAM
  - 120K provisioned IOPS on an `io2` volume

Postgres Supports 43K Workflows/Second



- No-op workflows submitted from many async clients
- Two database writes per workflow
  - One at the beginning to record input
  - One at the end to record outcome

Postgres Queues Support 31K WFs/Second



- 512 concurrent workers processing queued tasks
- Three database writes per queued workflow
  - One to record input
  - One to dequeue
  - One to record output

Stay in Touch

DBOS

Peter Kraft

peter.kraft@dbos.dev

https://www.linkedin.com/in/peter-kraft-dbos/

https://github.com/kraftp

https://petereliaskraft.net/