Autotuning Geo-distributed systems

A Contextual Bandit for Dynamic Data Movement in Detock

Rares Andrei Popa[1]

Supervisor(s): Asterios Katsifodimos[1]**, Oto Mráz**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Rares Andrei Popa
Final project course: CSE3000 Research Project
Thesis committee: Asterios Katsifodimos, Oto Mráz, Burcu Ozkan

Abstract

Geo-distributed databases must serve transactions close to where their data is accessed to keep latency low, which is why systems like Detock assign each data item a ”home” region and route its transactions there. But Detock leaves the placement policy open: where should each key live, and when should it move? Reactive heuristics fill this gap poorly. Deciding one key at a time, they split co-accessed records across regions and turn otherwise single-home transactions into expensive multi-home ones. Migrations that fire too often or at the wrong time also force concurrent transactions reading a stale home to restart.

This paper presents an adaptive placement agent for Detock. It groups co-accessed keys into communities using the Leiden algorithm, then uses a contextual bandit to decide when and where to migrate each community. The bandit learns from the locality and restart signals it observes at runtime, with no prior knowledge of the workload. We evaluate the agent on a two-region, follow-the-sun deployment of the Product–Parts–Supplier benchmark. Under matched migration budgets, the agent tracks the shifting hotspot as well as a DynaMast-style baseline while issuing about 29% fewer home-movements and causing roughly $4.3\times$ fewer transaction restarts. These results show that a lightweight, group-aware and cost-aware learned policy is a practical way to drive home-movement in a deterministic geo-distributed database.

1 Introduction

Applications increasingly rely on geo-distributed databases for availability and low latency worldwide, replicating and partitioning data across regions so requests can be served close to the user. But physically distant locations introduce significant network delays [13], making it hard to maintain strict consistency for transactions spanning partitions in different regions without high latency [22; 16; 7].

Recent systems tackle this cross-region synchronization problem. Detock [15] in particular combines a specialized concurrency-control protocol with deterministic deadlock resolution to maintain strict serializability without destroying performance under high contention, assigning each data item a “home” region and routing transactions there to minimize cross-region communication. Its weakness: a transaction whose accessed keys are homed in different regions becomes multi-home and must pay cross-region communication cost.

One natural way to reduce multi-home transactions is to dynamically move keys to better home regions as the workload changes. Systems like DynaMast [1] and PolyBase [17] take this approach, but both target different architectures than Detock. DynaMast is designed for fully replicated systems on a fast local network, where remastering is a lightweight, metadata-only operation. PolyBase is geo-distributed but falls back to two-phase commit (2PC) when a transaction touches rows whose hottest regions differ [17].

Detock’s situation is different. Its deterministic deadlock resolution protocol handles multi-home transactions without 2PC and without deadlock-related aborts, retaining at least 76% [15] of its throughput even at the highest contention levels. This means aggressive migration is less urgent than in other systems.

However, migration is not free. Home-movement transactions are themselves multi-home transactions that must coordinate across regions over wide-area network delays. Concurrent transactions that read a stale home from the *Home Directory* will abort and restart after a migration completes [15]. Triggering many migrations under high contention can therefore hurt throughput rather than improve it. Like approaches that identify and group co-accessed keys to reduce multi-home transactions [5], we move these groups dynamically, but only when the long-term benefit outweighs the short-term migration cost.

This paper addresses this by proposing an adaptive key placement framework for Detock. The framework uses a contextual bandit algorithm [9; 10] combined with the Leiden community detection algorithm [23] to learn and adapt key placement decisions in real time, without requiring any prior knowledge of the workload.

The rest of the paper is organized as follows. Section 2 gives background on Detock, existing remastering policies, the Leiden community detection algorithm, and contextual bandits; Section 3 identifies the opportunities for an effective home-movement policy; Section 4 details the placement agent; Section 5 evaluates it against static and reactive baselines; Sections 6 and 7 discuss reproducibility and the approach’s limitations; and Section 8 concludes.

2 Background

This section covers the background for our approach: Detock’s architecture and home-movement mechanism (Section 2.1), the existing remastering policies we use as baselines (Section 2.2), the Leiden community detection algorithm (Section 2.3), and contextual bandits (Section 2.4).

2.1 Detock

Detock is a geo-partitioned database that uses deterministic transaction scheduling to minimize cross-region coordination [15]. Each data item is assigned to exactly one home region, the only region whose local log orders operations on that item. When a transaction arrives, the receiving region becomes its coordinator: it looks up each accessed key’s home in a distributed index, the Home Directory, and forwards the transaction to the relevant home regions, each of which inserts it into its local log and asynchronously replicates it. Because every region eventually receives the complete set of batched requests, all regions construct the same global order and reach the same state deterministically [22].

A transaction accessing keys in a single home region is *single-home* (SH; Figure 1) and executes entirely within that region’s local log with no cross-region communication. A transaction spanning multiple home regions is *multi-home*

(MH); it is inserted into each participating region's local log, and execution must wait until it has received the relevant log entries from all of them.

A single-home transaction whose home region differs from the coordinating region is *foreign single-home* (FSH). It remains single-home in the protocol, but the coordinator must forward it over the wide-area network and wait for the result, so it pays a cross-region round trip comparable to that of an MH transaction.

Detock handles the resulting ordering challenges through a graph-based deadlock resolution protocol. This protocol allows each region to independently construct the same dependency graph and execute transactions locally, without relying on a global ordering service. As a result, Detock can process MH transactions without the round-trip overhead of two-phase commit or a centralized sequencer [15].

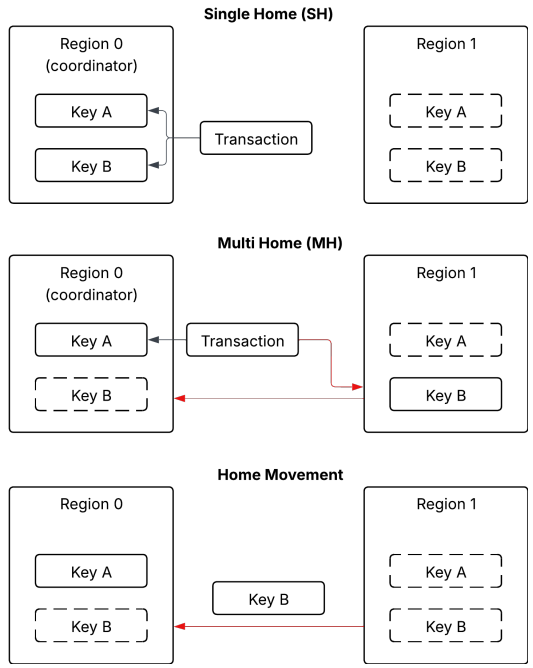

Figure 1: Solid borders indicate the home (master) copy of a key; dashed borders are replicas. *Top:* a single-home (SH) transaction (both keys homed in Region 0) executes locally. *Middle:* a multi-home (MH) transaction (Key A in Region 0, Key B in Region 1) forces the coordinator to forward to Region 1 and wait. *Bottom:* a home-movement migrates Key B from Region 1 to Region 0; concurrent transactions reading the stale home abort and restart until the new assignment propagates.

Detock also supports home-movement transactions, which reassign a key's home region at runtime (Figure 1, bottom) [15]. A home-movement transaction is always treated as an MH transaction, since it must coordinate between the old and new home regions.

The risk of home-movement lies in execution-time validation. A transaction embeds each key's home as looked up at creation; at execution Detock checks that the recorded home still matches the current one, and if a migration completed in between, the check fails and the transaction aborts and restarts. Any transaction that looked up a key before the migration but executes after it will abort, and this window persists until the new home propagates to all coordinators. Triggering home-movement under high contention can therefore cause a burst of aborts that temporarily hurts throughput [15]. Detock provides the mechanism but no policy for when or where to move keys; supplying one is the focus of this work (Section 4).

2.2 Existing Remastering Policies

Detock supplies the home-movement mechanism but not a policy, and prior replicated and geo-distributed systems have proposed different strategies for deciding what to remaster and when. We adopt two of them as baselines.

SLOG [16] takes a reactive, per-key approach. It tracks where each key is accessed from and, once a key has been requested from the same remote region enough times in a row, remasters that key to the requesting region. The decision is made independently for every key, using only that key's own recent access history.

DynaMast [1] takes a score-based approach. It was designed for fully replicated systems on a fast local network, where a copy of every record already exists in each site, so changing a record's master is a lightweight metadata operation. DynaMast scores each region by how well it would serve a record's recent accesses and remasters the record to the best-scoring region, aiming to keep subsequent transactions single-master.

Our two reactive baselines, *slog3* and *dynamast_composite*, are concrete adaptations of these strategies to Detock; we give their exact decision rules where they are used (Sections 3 and 5).

2.3 Leiden Community Detection

The Leiden algorithm [23] is a community detection algorithm. It partitions a graph into groups of densely interconnected nodes by optimizing a quality metric called modularity, as illustrated in Figure 2. Modularity measures how much more densely connected nodes within a community are compared to what would be expected by chance [14]. One practical advantage is that the number of communities does not need to be specified in advance.

Leiden was introduced as a direct improvement over Louvain [3], which can produce badly connected or even disconnected communities: Traag et al. [23] found up to 25% badly connected and 16% disconnected. Leiden adds a refinement phase that guarantees every community is connected, and is also up to 20× faster [23]. Unlike Girvan–Newman [6], whose $O(m^2n)$ cost is impractical for repeated use, Leiden inherits the near-linear empirical scaling in the number of edges on sparse graphs that Blondel et al. showed for Louvain's local-moving and aggregation phases [3]; Leiden's refinement step does not change this complexity. This keeps each invocation cheap enough to rerun every decision round. At the scales we consider, repeated invocation cost rather than graph size is the binding constraint.

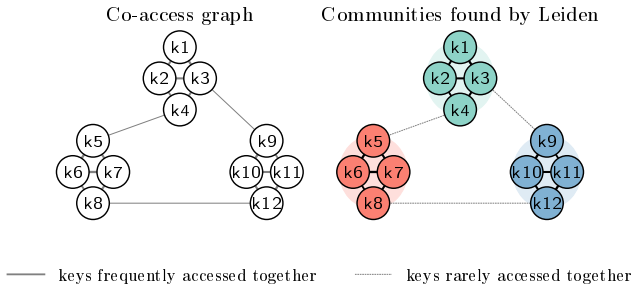

Figure 2: An illustrative co-access graph, where nodes represent keys and edges represent co-accessing transactions. *Left:* the raw graph exhibits three densely connected groups linked by a small number of weak cross-group edges. *Right:* the Leiden algorithm correctly recovers these three groups as communities, which form the migration units used by the placement framework (Section 4).

2.4 Contextual Bandits

Reinforcement learning (RL) is a framework in which an agent learns to act by interacting with an environment, taking actions, receiving rewards, and updating its policy to maximize long-term cumulative reward [20]. Full RL requires modeling how actions affect future states, which is hard to estimate in dynamic database environments; systems like Bao sidestep this by using contextual-bandit-style learning for query optimization instead of full RL [11].

A contextual bandit [9; 10] drops this long-term dependency: at each decision point the agent observes a context vector, selects an action, and receives an immediate reward, with no notion of future state. This makes bandits easier to train and more stable in non-stationary environments where the workload shifts. We apply the bandit to per-community placement in Section 4.4.

3 Opportunities

Detock provides a home-movement mechanism but leaves the placement policy to the operator. This section identifies two opportunities for building an effective placement policy, each supported by a motivating experiment.

3.1 Motivating Workload

We build on the PPS (Product–Parts–Supplier) benchmark [12], reusing its existing Detock implementation, rather than a generic key–value workload (YCSB) or a warehouse workload (TPC-C) because it cleanly exhibits two properties common in geo-distributed OLTP that make placement non-trivial. First, PPS has a schema-level notion of co-access: each product is composed of several parts read together, so co-accessed keys form stable groups that recur across transactions, as in many real OLTP workloads [5; 19]. This differs both from a key–value store, where keys are independent, and from TPC-C [24], whose `NEW-ORDER` transactions draw a random set of items per order, so its recurring locality is the warehouse partition rather than fixed groups of keys. Second, regional demand should not be static: access locality in large geo-distributed deployments shifts over time and across regions, motivating systems that migrate data to follow it [2]. Since the original benchmark uses a fixed per-region affinity, we extend it with a time-varying affinity that rotates the dominant region over a single run, as in a follow-the-sun deployment. Because a good placement must keep co-accessed records together *and* follow them as locality moves, PPS lets us isolate this effect cleanly.

In our motivating runs all transactions are single-home, so every locality miss is in principle correctable by remastering. Each region's clients target a hot product set with a sharp regional affinity that rotates over the run, shifting where each client directs its accesses from its local region to a remote one and back, as in a follow-the-sun deployment (we give the exact parameters in Section 5.1). When the dominant region moves away from where a product is homed, under a fixed placement every transaction on it becomes a cross-region request paying a full wide-area round-trip, correctable only by remastering the product's keys to the new region. Since a product's parts are accessed together, each product forms a co-accessed group, so the workload tests two requirements at once: keep co-accessed records together, and adapt to periodic shifts in dominance.

We compare three configurations, drawing the two reactive policies from the approaches described in Section 2.2. *Disabled* keeps the initial placement static with no migration. *Slog3* is the SLOG heuristic [16]: it remasters a key to the requesting region after three consecutive remote accesses from that region. *Dynamast_composite* is our adaptation of DynaMast [1] to Detock; each key is scored by its dominant-region fraction plus a load-balance bonus and a co-access bonus, and remastered when the score crosses a fixed threshold.

3.2 Opportunity 1: Placement Must Adapt to Workload Shifts

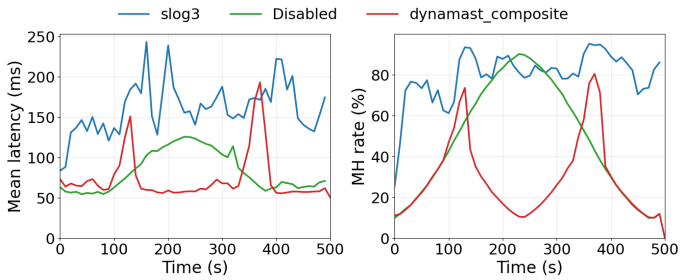

Figure 3: Mean committed-transaction latency (left) and multi-home rate (right) for the three configurations. *Disabled* is fast when the workload aligns with the initial placement and slow otherwise. *Slog3* stays elevated throughout, its per-key migrations splitting co-accessed products and keeping the MH rate above 50%. *Dynamast_composite* returns to the low-latency regime between rotations but pays a large transient at each transition (latency ≈200 ms, MH rate ≈85%).

Figure 3 shows committed-transaction latency and the MH rate over time. The static baseline drops to ≈60 ms when the workload aligns with the initial placement and climbs to ≈140 ms when it does not. The latency climb comes from foreign single-home transactions (Section 2.1), which the figure reports alongside MH because their latency is compara-

ble. Because the baseline never migrates, it cannot correct these and can never adapt when the workload shifts. *Slog3* sits at 150–210 ms throughout with its MH rate above 50% and never recovers, even when the placement is stable, so its per-key remastering is actually worse than leaving the placement alone (we explain why in Opportunity 2, Section 3.3). *Dynamast_composite* does find placements that return to the local-access regime, but pays a transient at each transition while it migrates the co-accessed group (latency ≈200 ms, MH rate ≈85%). In short, the workload keeps shifting, so a good policy must move keys to keep up with it. But because every migration has a cost, it should move keys only when a shift actually makes it worthwhile.

3.3 Opportunity 2: Co-Accessed Keys Must Move Together

A transaction becomes multi-home as soon as any two of its keys have different home regions, and an MH transaction pays the full cross-region ordering cost regardless of how many keys are local [15]. This is why *slog3*'s per-key remastering backfires: once it sees three consecutive remote accesses on one key it remasters that key alone, but the product's other parts stay behind, so the product is split across regions and every transaction on it becomes multi-home. Moving only a subset of a co-accessed group therefore does not help; the transaction stays MH. The multi-home rate in Figure 3 (right) shows this directly: although the workload is single-home by construction, *slog3* keeps the MH rate above 50% for most of the run, with peaks above 70%. An effective policy must therefore identify groups of keys that are consistently accessed together and migrate them as a unit.

3.4 Approach

These two observations motivate our framework's two components. For Opportunity 1, a contextual bandit learns placement decisions from observed locality and restart costs; for Opportunity 2, the Leiden algorithm clusters a co-access graph so decisions are made per group rather than per key.

4 Implementation

This section describes the integration of the agent into the Detock codebase. The agent runs as a controller on a single node, collecting access events from the whole system, and is invoked once every configurable evaluation interval to process all events accumulated since the previous round.

The pipeline has five steps: (1) collecting access events, (2) building a co-access graph over frequently accessed keys, (3) applying community detection to identify clusters, (4) using a contextual bandit to select an action per cluster, and (5) computing the reward.

4.1 Event Collection

Each server emits an access event for every key it touches, recording the key, the originating region, the key's current home region, a transaction identifier, and an origin type that lets the controller discard accesses generated by remaster transactions themselves so migrations do not pollute the statistics. Events are batched and forwarded to the controller over the internal messaging channel and processed asynchronously (Figure 4).

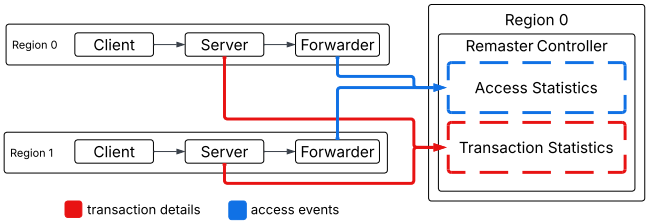

Figure 4: Telemetry collection for the bandit agent. Two reporting paths converge at the *Remaster Controller*. The **Forwarder** is the only component observing individual key accesses with their master metadata, so it emits *access events* that build the per-key statistics used as bandit context. The **Server** relays a *complete transaction record* per root transaction (the only true end-to-end view across retries, including total latency and the remaster-induced restart count), which feeds the reward computation. Both paths batch and flush every 100 ms.

The controller maintains per-key statistics: a sliding window of recent accesses, per-region access counts within a short *trend* window and a longer *recent* window, the current home region, and a bounded set of co-access partners. The recent window gives a stable view of access patterns while the trend window captures shifts that may signal a workload change. Co-access partners are recorded during batch processing: within each transaction every pair of distinct keys is considered co-accessed, incrementing both keys' partner sets.

Separately, for each transaction the system reports a complete end-to-end record to the controller: the root transaction id, the keys accessed, whether it committed, the total time from first submission to commit, and the number of remaster-induced restarts the transaction incurred along the way. These reports are accumulated per key for reward computation (Section 4.5), letting the agent attribute remaster-induced restarts back to the move that caused them.

4.2 Graph construction

At each round the agent builds a co-access graph $G = (V, E)$ from the tracked keys. A key is *hot* if it has been accessed at least T times (a configurable threshold); only hot keys become nodes, keeping the graph small and focused on keys frequent enough for placement to matter.

An edge (k_i, k_j) is added between two hot keys if they appear together in the same transaction at least once. The weight of the edge is defined as:

$$w(k_i, k_j) = \frac{\text{co-occurrences}(k_i, k_j)}{\min(\text{count}(k_i),\ \text{count}(k_j))} \tag{1}$$

This measures how consistently k_i and k_j are accessed together relative to the less frequent of the two: a weight near 1 means they almost always co-occur, near 0 that they rarely do. Using the minimum keeps small keys that always co-occur with a larger group at weight 1, so they are not lost. Edges below a minimum weight τ are discarded to avoid connecting keys that co-occur only by chance.

4.3 Leiden Clustering

The co-access graph is partitioned into communities using the Leiden algorithm [23], applied via the igraph C library [21]. Each community is a group of keys that are more densely co-accessed with each other than with the rest of the graph, and serves as the unit of migration: rather than deciding placement key by key, the agent makes a single placement decision per community, ensuring that co-accessed keys are always moved together.

The clustering is rebuilt each round. Following igraph's API convention [21], we pass the resolution parameter divided by $2m$ (the graph's total edge weight) to recover the standard modularity formula. This keeps community granularity stable as window traffic varies.

To keep the bandit's per-round work bounded, we rank communities by their recent access volume and pass only the smallest prefix whose cumulative traffic covers a fraction ρ of the window's total to the *action-selection* step; the rest are left in place.

4.4 Agent Decision

For each candidate community, the agent selects one of $R+1$ actions: move the community to any of the R regions, or leave it in its current placement (the *stay* action).

The *region* and *stay* actions are not interchangeable, since a community is generally split across regions at decision time. A region action consolidates the whole community at one region, issuing a home-movement for every member not already there; stay leaves each member at its current, possibly heterogeneous, master and issues nothing.

The agent uses a small neural network with parameters θ (two hidden layers of 64 and 32 ReLU units, trained with Adam [8] at learning rate 0.01). It maps the feature vector $\phi(c, a)$ built for a given (community, action) pair to a scalar score $Q_\theta(c, a)$; Algorithm 1 uses both symbols when it selects an action. The network's design and tuning were not a focus of this work, and the architecture is intentionally small to keep prediction cheap, since it runs on every candidate pair every round.

The highest-scoring action is selected under an ε-greedy policy that picks a uniformly random action with probability ε. Two rules refine the greedy choice: a move is preferred over staying only if its score exceeds the stay score by a hysteresis margin of 0.2 (suppressing marginal migrations), and a move in which no member key needs to relocate is reduced to stay.

ε decays linearly from 0.2 to 0.01 over the first 120 rounds, exploring aggressively early when predictions are unreliable and exploiting once trained. It never reaches zero: because the workload changes over time, the agent must keep occasionally exploring actions that currently look suboptimal so it can adapt.

A community is evaluated only if it has at least `min_samples` recent accesses and none of its keys has a migration in flight or a reward pending. On a non-stay action, the controller issues one home-movement per member whose home differs from the target region; these enter Detock as standard MH transactions and are tracked until they commit or abort.

Algorithm 1 One evaluation round of the placement agent

Require: recent access events and latency reports
▷ Phase 1: telemetry
update per-key statistics: recent/trend windows, co-access partners
▷ Phase 2: clustering
$G \leftarrow$ co-access graph over hot keys
$\mathcal{C} \leftarrow \textsc{Leiden}(G)$
$\mathcal{S} \leftarrow$ top communities by recent traffic covering fraction ρ
▷ Phase 3: per-community decision
for all $c \in \mathcal{S}$ **do**
  if c lacks samples **or** has a pending migration/reward **then**
    continue
  end if
  $a^\star \leftarrow \textsc{SelectAction}(c)$
  if $a^\star \neq$ STAY **then**
    issue home-movement for each member not homed at $a^\star$
  end if
  record $\phi(c, a^\star)$ and schedule its delayed reward
end for

procedure $\textsc{SelectAction}(c)$
  for all $a \in \{0, \ldots, R-1, \text{STAY}\}$ **do**
    $q_a \leftarrow Q_\theta(c, a)$
  end for
  if $\text{rand}() < \varepsilon$ **then**
    return uniformly random action ▷ exploration
  end if
  $a^\star \leftarrow \arg\max_a q_a$
  if $a^\star \neq$ STAY **and** $q_{a^\star} - q_{\text{STAY}} < \text{margin}$ **then**
    $a^\star \leftarrow$ STAY ▷ hysteresis
  end if
  if $a^\star \neq$ STAY **and** no member of c must relocate **then**
    $a^\star \leftarrow$ STAY
  end if
  return $a^\star$
end procedure

Algorithm 1 summarizes one evaluation round, from event collection through the per-community decision; the delayed reward update that closes the loop is described in Section 4.5.

The feature vector for a given action (target region) and community contains six values, each signaling either the benefit of the action (how many MH transactions it converts to SH) or its cost (aborts and migration overhead):

- **Recent access fraction:** accesses to the community originating from the target region over the recent window; high values mean a move converts many transactions to SH.
- **Trend delta:** the same fraction in the short trend window minus the longer recent window; positive values mean accesses are shifting toward the target region.
- **Migration fraction:** fraction of member keys that would have to move, each an extra MH home-movement

and potential abort source.

- **Traffic share:** fraction of all accesses in the previous interval that targeted this community, weighing the disruption of moving a high-traffic community.
- **Community density:** intra-community edge weight (Equation 1) normalized by the number of possible pairs; denser communities are more self-contained, so moving them more reliably converts MH to SH.
- **Stay flag:** a binary flag marking the stay action; without it, staying and moving-to-the-current-region can yield identical features.

The network is trained fully online and on-policy, with no offline dataset or pre-training. Each decision yields one training example, the chosen pair's feature vector and its delayed reward, on which the network takes an Adam step minimizing squared error between predicted and observed reward. The agent thus acts from round 0 with a random network and high exploration, learning from the (context, reward) stream it generates.

A replay buffer of capacity C stores past (feature, reward) pairs. After the gradient step on the current observation, the agent samples a batch of b entries uniformly with replacement and steps on each. This prevents overfitting to the most recent signal: training only on the current window would bias the weights toward whatever action that window rewards, forgetting other situations.

4.5 Reward calculation

The reward is delayed. At decision time the agent records the affected keys' cumulative access and restart statistics as a baseline, then waits until at least `reward_min_post_samples` new latency reports have arrived and at least `reward_window_rounds` rounds have elapsed since the move committed; if neither holds within `reward_timeout_rounds`, the pending update is dropped.

The reward credits the improvement in local service relative to a counterfactual, penalized by the restarts the decision induced:

$$r = 2 \cdot \max(-1,\ (\text{local\_frac} - \text{cf\_frac}) - \text{restart\_frac}) \quad (2)$$

where:

- **local_frac**: the fraction of post-decision accesses to keys in the community that were served by the chosen master region. A migration that successfully concentrates traffic at its target pushes this toward 1.
- **cf_frac**: a counterfactual reference value for local_frac. For a *move*, it is the fraction that *would have been local* if the community had stayed at its previous master, i.e., what the agent's locality would have been had it not acted. For a *stay*, it is the fraction the best *alternative* region would have served, i.e., the locality we forwent by not migrating. The reward thus measures the actual gain over the agent's best alternative in either direction.
- **restart_frac**: the number of remaster-induced restarts suffered by transactions touching the community after the decision, divided by the community's post-decision transaction count, and capped at 1. This is the agent's share of blame for restarts caused by its own home movements.

Both move and stay decisions generate a reward and training update: a move is rewarded only when it raises the local fraction above the previous master's, and a stay only when the current placement beats its strongest alternative.

Detock revalidates every key's home at execution time [15]: a migration committing while concurrent transactions hold a stale home (Section 2.1) forces them to restart. These restarts enter the reward through its penalty term, teaching the agent that migrations are not free and must justify their cost through improved locality.

5 Experimental Setup and Results

This section evaluates the bandit-based placement agent against the baselines from Section 3: Section 5.1 describes the deployment and automation, Section 5.2 the results.

5.1 Deployment and Experiment Automation

We conduct all experiments on a dedicated four-node cluster at TU Delft, each node with dual AMD EPYC 7H12 processors (256 hardware threads), 503 GiB of RAM, and 10 Gigabit Ethernet. We simulate a two-region deployment across the four nodes, injecting a one-way delay of 65 ms (3 ms jitter) on cross-region links via `tc netem` for an inter-region round-trip of ≈130 ms. Clients and servers run as Docker containers.

The cluster configuration mirrors the setup used for the motivating experiments in Section 3: two regions, two partitions per region, with the PPS schema [12] loaded with 250 products, 500 parts, and 250 suppliers. Each region holds a complete copy of the data, split across two partitions with one server container per partition (and a single replica per partition), giving four server containers in total. We run 1600 benchmark clients per region (3200 total), with each region's clients colocated on one of the physical nodes. We evaluate all four policy presets on the same configuration in a single 1600 s trial each, under the same workload.

The workload is the same follow-the-sun pattern described in Section 3.1, restricted so that every transaction is single-home and single-partition. This choice keeps the measurements interpretable: because the workload never generates a multi-home transaction on its own, any multi-home transaction that appears during a run is induced by the placement policy itself: either a co-access group that a policy has split across regions, or a transaction caught mid-migration.

The restart counts we report are scoped to a single cause: restarts that Detock tags as remaster home-validation failures, where a transaction's recorded home no longer matches the current home at execution time (Section 2.1). Detock's deterministic execution does not abort for concurrency-control reasons [15]; other aborts (a two-phase *OrderProduct* failing its second-phase validation, or transient distributed failures) still occur but are deliberately excluded. We therefore plot the cost attributable to each placement policy, not the system's total abort rate.

Time-varying affinity is enabled: each client's preferred region rotates smoothly, advancing by one region every 240 s. The affinity distribution is sharply peaked, so the center region receives roughly ten times the weight of its neighbor, and a NURand skew [24] within each region concentrates accesses on a small subset of products to create realistic contention. The fast rotation forces any adaptive policy to detect the shift and react in time.

To make the comparison fair, the three active policies are given comparable migration budgets, so that differences in migration volume reflect how often each policy *decides* to move keys rather than an artificial throttle. The bandit evaluates every 2 s with a per-round cap of 75 migrations; *slog3* and *dynamast_composite* evaluate every 0.5 s with a per-round cap of 17. This works out to roughly 34–38 permitted migrations per second for each policy (37.5/s for the bandit, 34/s for the other two).

Four policies are compared:

- **Disabled.** No remastering. Keys stay at their initial home for the entire run, serving as the static baseline.
- **Slog3.** The SLOG-style heuristic [16]: a key is remastered to the requesting region after three consecutive remote accesses from the same region. A local access resets the counter, and a remote access from a different region restarts the streak at that new region. Decisions are made per key with no awareness of co-access groups.
- **Dynamast_composite.** Our adaptation of DynaMast [1] for Detock. Each key is scored by its dominant-region access fraction, plus a load-balance bonus (scaling up to 0.10, and only when the candidate region is less loaded than the current master) and a flat strongest-partner co-access bonus of 0.10. The key is remastered when the composite score exceeds a fixed threshold of 0.70. Like *slog3*, decisions are made per key, but the co-access bonus gives it partial awareness of related keys.
- **Bandit.** The contextual bandit agent of Section 4: it clusters co-accessed keys with Leiden and makes one placement decision per community, under ε-greedy exploration (ε decaying linearly from 0.2 to 0.01 over the first 120 rounds). At a 2 s evaluation interval, the exploration phase spans the first 240 s of the run.

5.2 Results

Bandit Behavior

Figure 5 isolates the bandit against the static baseline. During the exploration phase (left of the dashed line) ε is still high, so the agent takes random actions up to 20% of the time. Once ε has decayed to 0.01 the learned policy dominates: each rotation of the affinity center is detected and the affected communities are moved to the new dominant region. The narrow latency spike at each boundary is the cost of that move: the home-movements are themselves MH, and concurrent transactions reading a stale home abort and restart (Section 2.1). It is short-lived because communities settle within one or two rounds.

We deliberately do not report a single aggregate "latency reduction" against the static baseline: the gap is dominated by how long each misaligned phase lasts, set by our rotation interval (240 s), so any single percentage would describe that choice rather than the policy. The meaningful quantity is time-resolved: the bandit returns to the local-access regime shortly after each shift and stays there, whereas the static baseline spends roughly half of every cycle misaligned.

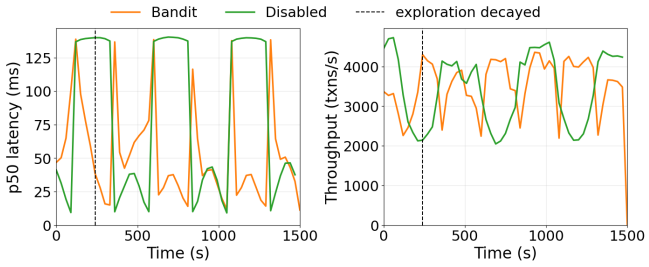

Figure 5: Median latency (left) and throughput (right) for the bandit and the static baseline; the dashed line marks full ε decay (240 s). *Disabled* square-waves between the aligned (≈10 ms) and misaligned (≈140 ms) regimes as the affinity center rotates; *bandit* holds a low band after exploration, broken only by brief spikes at each rotation.

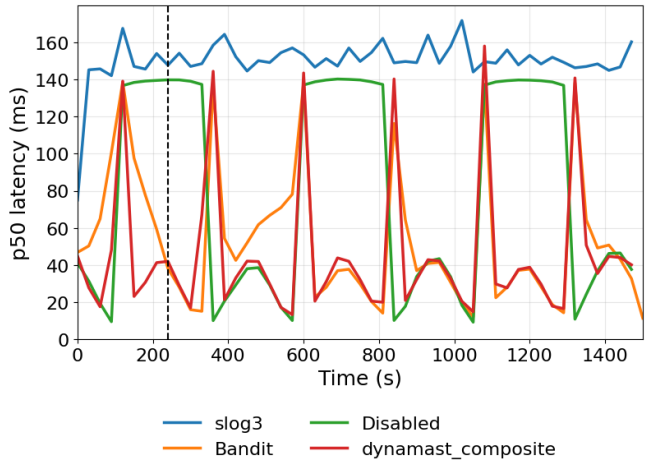

Figure 6: Median latency over time for all four policies. *Slog3* stays elevated and noisy for the whole run; *disabled* square-waves with the rotation; *dynamast_composite* and *bandit* both track the shifts, spiking briefly at each rotation and recovering to the low-latency regime. The dashed line marks the end of the bandit's ε-decay exploration phase (240s) and applies only to bandit's curve.

All-Policy Comparison

Figure 6 places all four policies on the same axes. The clearest outcome is that *slog3* never recovers: its latency stays elevated and noisy across the entire run, never returning to the local-access regime even when the affinity center is stable. This is the per-key splitting effect described in Section 3.3: an MH transaction pays the full cross-region cost regardless of how many of its keys are local [15], so as the affinity center rotates continuously, the heuristic keeps splitting and re-splitting co-accessed groups and never converges.

Dynamast_composite and *bandit* both recover after each shift, since both react to the same signal, a change in the dom-

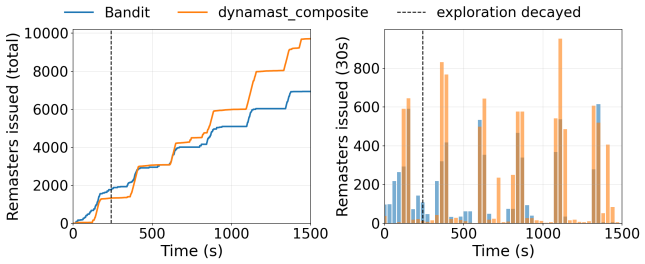

Figure 7: Cumulative home-movements (left) and per-window counts (right) for *bandit* and *dynamast_composite* under comparable per-second budgets; the dashed line marks the end of exploration (240 s) and applies only to the bandit. The bandit issues fewer total migrations, concentrated in bursts at the rotation boundaries; *dynamast_composite* also migrates between rotations.

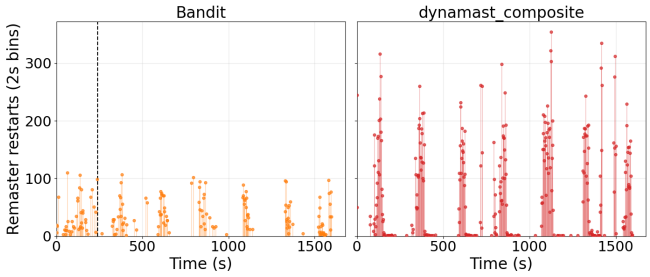

Figure 8: Remaster-induced restarts in 2 s bins for *bandit* (left) and *dynamast_composite* (right): 6 184 versus 26 844 over the run; the dashed line marks the end of the bandit's exploration phase (240 s). The bandit's restarts stay in short bursts at the rotation boundaries (peaks ≈110/bin) with quiet stretches between; *dynamast_composite* reaches taller peaks (≈354/bin) and also restarts during stable periods.

inant access region, so their latency curves are close. The difference is not in latency but in how much work each does to achieve it, which the next two figures quantify.

Migration Volume

Figure 7 counts the home-movements each policy issues. Because both were granted comparable per-second budgets (Section 5.1), the gap reflects their decisions, not a cap: the bandit issues 6 935 migrations against 9 711 for *dynamast_composite*, about 29% fewer. The per-window panel shows why: after exploration ends, the bandit's activity collapses into sharp bursts at the rotation boundaries and falls to near zero between them, because the hysteresis margin makes it prefer staying once a community is well placed (Section 4.4).

Dynamast_composite keeps migrating between rotations too. It scores each key over a sliding window and remasters whenever the best region differs from the current master and the composite score clears the fixed 0.70 threshold. It skips keys already best-placed, but applies no hysteresis to the move itself, so during a stable phase a transient burst of remote accesses can briefly push another region over the threshold, triggering a move a longer-horizon view would not justify.

Transaction Restarts

Figure 8 shows the downstream cost. A home-movement that commits while concurrent transactions still hold the old home forces them to abort and restart at execution time [15]; the restart counts we report are scoped to this mechanism (Section 5.1). The totals diverge sharply: *dynamast_composite* forces 26 844 restarts against the bandit's 6 184, roughly 4.3× more, while issuing only about 40% more home-movements (9 711 versus 6 935, Figure 7), so each of its migrations is, on average, far more disruptive.

The per-bin view explains the gap. At each rotation boundary *dynamast_composite*'s peaks reach about three times the bandit's (≈354 versus ≈110 per bin), because its per-key decisions push many keys over the threshold at once, widening the window in which stale homes can be read. It also keeps restarting transactions during stable periods, caused by the extra remasters it issues even when the placement is stable (Figure 7). The bandit's restarts stay concentrated at the transitions, because the restart-penalty term in its reward (Section 4.5) teaches it not to migrate an already well-placed community.

6 Responsible Research

This research does not involve any personal data, user studies, or sensitive information. All workloads used in the experiments were generated synthetically by the benchmarking framework, with no connection to real users or production systems. As a result, the work raises no concerns related to privacy, confidentiality, or the protection of individuals.

Several measures were taken to support the reproducibility of the results. The experimental setup, including the hardware configuration, the workload parameters, and the metrics collected, is described in Section 5. The deployment is fully automated using Docker, and all components of the system, the modified Detock server, the client, and the experiment-running scripts will be made available in a public repository [1], together with the configuration files used in this work and instructions for rebuilding and rerunning each experiment. The orchestration script fixes the random seed per trial, so within each trial all four policies execute against the same transaction stream and any difference between them reflects the policies, not the input. We ran each experiment several times to confirm that the measurements are stable and do not vary substantially between runs; the figures report a single representative run rather than an average, since the run-to-run variation we observed was small relative to the differences between the policies. Even with these measures, exact numerical results may differ slightly between machines or runs; this is expected in distributed-systems research and does not affect the comparisons.

Generative AI tools were used during this project as an assistive aid, but not as a substitute for the reasoning, design decisions, or analysis presented in this thesis. Specifically, AI was used to navigate parts of the Detock codebase more

[1] https://github.com/delftdata/Detock/tree/rl-dynamic-data-movement-rpopa

quickly during the limited project timeframe, to look up syntax for C++ and Python, and to help generate quick scripts for analyzing logs and producing exploratory plots. The design of the placement framework, the choice of algorithms, the feature engineering, the reward formulation, the experimental setup, and the interpretation of the results are our own work.

7 Discussion

The experiments in Section 5 show that a group-aware, learning-based placement agent can keep a rotating workload close to its local-access latency while issuing far fewer migrations and causing far fewer transaction restarts than a per-key heuristic. This section reflects on the scope of that result: how the approach extends beyond the two-region setup we evaluated (Section 7.1), and two situations the current design handles imperfectly, the cold start of an untrained agent (Section 7.2) and the timing of migrations around traffic peaks (Section 7.3).

7.1 Generalizing Beyond Two Regions

Our evaluation uses two regions, but the agent is not specialized to two. At each decision it already chooses among $R+1$ actions (consolidate a community at any of the R regions, or leave it in place; Section 4.4), so adding regions only widens the action set and requires no change to the co-access graph, the clustering, or the reward.

What the current design does not capture is that, in deployments spanning more than two regions, not all cross-region links are equally expensive. Round-trip latency between nearby regions can be an order of magnitude smaller than between distant ones [4], yet our reward (Section 4.5) treats every cross-region link as equally costly. A migration between nearby regions saves little latency but still triggers the stale-home restart burst of (Section 5); the same move across a distant region can save a full wide-area round-trip. One way to extend the policy is to make it distance-aware: add the round-trip time between a community's current home and each candidate region as a feature, and weight the reward by the latency a move actually saves. We leave a multi-region evaluation with realistic inter-region latencies to future work.

7.2 Warm-Starting the Agent

The agent acts from the first round with a randomly initialized network and high exploration (Section 4.4). This is the riskiest moment in the run: scores are arbitrary, ε-greedy adds further random moves, and early migrations no trained policy would make pay the stale-home restart cost of Section 5. Hysteresis and ε-decay dampen but do not remove this.

Because the reward depends only on the (context, reward) pair and not on which policy produced the decision, the agent could be warm-started: run a cheap baseline for a short bootstrap, log its (decision, reward) pairs into the replay buffer (Section 4.4), and train the value network offline before the bandit takes over. Exploration remains necessary afterwards, since the bootstrap log only records the action the baseline took in each context, leaving the network uninformed about alternatives such as staying when the baseline moved. A lower starting ε and shorter decay schedule would likely suffice in this case, since the network begins from an informed prior. We leave its implementation and evaluation to future work.

7.3 Timing Migrations Around Traffic Peaks

A sudden hotspot, such as a flash sale concentrating most of the workload on one group of keys, remasters poorly if triggered mid-burst: invalidating the group's home while many concurrent transactions are in flight forces all of them to restart at once (Section 2.1). The agent has the ingredients to avoid this without lookahead: a community's traffic share is an input feature (Feature 4, Section 4.4) and the restart fraction enters the reward (Section 4.5), so a move at peak traffic earns a low reward and the network can learn a reactive threshold above which staying beats moving. Once the surge passes, the same per-round comparison favors moving again.

Where that threshold lands depends on the reward weighting, which an operator can tune. The counterfactual term also penalizes *staying* when a community is misplaced (Section 4.5), so during a misplaced hotspot both actions are costly. Under the current symmetric weighting in Equation 2 (locality and restart terms both scale to magnitude two, with the restart fraction capped at one), the move can still win, so the agent absorbs the restart storm once rather than stay misplaced. An operator who instead prefers to ride out a hotspot at degraded locality can shift this balance by giving the restart term its own, larger coefficient in Equation 2, so a severe restart burst outweighs the locality gain and the agent holds off until the surge passes. We leave exploring this tuning to future work.

8 Conclusions and Future Work

Detock [15] reduces cross-region coordination by giving each record a home region, but provides only the mechanism to move homes—not a policy for when and where. We showed that the obvious reactive policies fit this problem poorly: moving keys one at a time, as a SLOG-style heuristic [16] does, splits co-accessed records across regions and turns single-home transactions into multi-home ones, so on a shifting-hotspot workload it never settles.

This paper presented an adaptive agent that groups co-accessed keys with Leiden [23] and learns, via a contextual bandit, when and where to migrate each group from the locality and restart signals it observes at runtime. On a two-region follow-the-sun workload, the agent tracks the shifting hotspot and returns to the local-access regime shortly after each shift, matching a DynaMast-style baseline [1] but far more cheaply: under matched migration budgets it issued about 29% fewer home-movements (6 935 vs 9 711) and caused roughly 4.3× fewer transaction restarts (6 184 vs 26 844).

The gap between the two policies comes from two design choices the experiments isolate: migrating whole co-access groups rather than individual keys, and a reward that charges each migration for the restarts it induces. Together they let the agent act only when a shift genuinely warrants it. The broader takeaway: restart storms can be avoided by making the policy group- and cost-aware, without trading adaptivity away for caution.

8.1 Future Work

Several directions follow from the limitations discussed in Section 7. First, the reward is currently distance-agnostic; weighting it by round-trip time instead would let the same agent handle setups with more than two regions, where some links are much cheaper than others: it would still migrate across expensive links but leave cheap ones alone (Section 7.1). Second, the agent acts from a randomly initialized network; warm-starting it on experience gathered under a cheaper policy would avoid the destructive early migrations of the cold-start phase (Section 7.2). Third, tuning the relative weight of the restart penalty would give finer control over how the agent behaves around extreme traffic peaks (Section 7.3).

A further direction concerns the clustering itself. The agent currently rebuilds the co-access communities from scratch at every decision round (Section 4.3). As the number of tracked keys grows, this repeated cost becomes the binding constraint on how often the agent can afford to run. Incremental, or *dynamic*, community detection (which updates an existing partition in response to the few edges that changed rather than recomputing it) would remove this bottleneck; recent work on dynamic Leiden clustering [18] is a promising starting point. Finally, our evaluation deliberately used a clean single-home workload so that every residual multi-home transaction could be attributed to the policy. A broader study with natively multi-home transactions, more partitions, and real-world access traces would test how the approach holds up under workloads that remastering cannot fully resolve.

References

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. Dynamast: Adaptive dynamic mastering for replicated systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1381–1392. IEEE, 2020.

[2] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: Managing datastore locality at scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 445–460, 2018.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[4] CloudPing.co. CloudPing: AWS inter-region latency monitoring. https://www.cloudping.co. Accessed: 2026-06-20.

[5] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[6] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[7] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. Caerus: Low-latency distributed transactions for geo-replicated systems. *Proceedings of the VLDB Endowment*, 17(3):469–482, 2023.

[8] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[9] Tor Lattimore and Csaba Szepesvári. *Bandit Algorithms*. Cambridge University Press, 2020.

[10] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 661–670. ACM, 2010.

[11] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, pages 1275–1288. ACM, 2021.

[12] Eduard-Alex Mihai. Benchmarking geo-distributed databases: Evaluating performance using the product-parts-supplier workload. Bachelor's thesis, Delft University of Technology, 2025.

[13] Oto Mraz, Kyriakos Psarakis, George Christodoulou, Paris Carbone, and Asterios Katsifodimos. The missing dimensions in geo-distributed database evaluation. *arXiv preprint arXiv:2605.30156*, 2026.

[14] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, 2004.

[15] Cuong D. T. Nguyen, Johann K. Miller, and Daniel J. Abadi. Detock: High performance multi-region transactions at scale. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

[16] Kun Ren, Dennis Li, and Daniel J. Abadi. SLOG: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11):1747–1761, 2019.

[17] Chaoyi Ruan, Yingqiang Zhang, Juncheng Zhang, C. Li, Xiaosong Ma, Hao Chen, Jie Zhou, Feifei Li, and Xinjun Yang. Polybase: Adapting to data affinity changes in geo-replicated database via row-level consensus-group affiliation re-assignment. *Proceedings of the VLDB Endowment*, 18(3):702–714, 2024.

[18] Subhajit Sahu. A starting point for dynamic community detection with leiden algorithm. *arXiv preprint arXiv:2405.11658*, 2024.

[19] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016.

[20] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[21] The igraph development team. igraph: The network analysis package (C library), version 0.10.15. https://github.com/igraph/igraph, 2024. Release `0.10.15`, https://github.com/igraph/igraph/releases/tag/0.10.15.

[22] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[23] V. A. Traag, L. Waltman, and N. J. van Eck. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1):5233, 2019.

[24] Transaction Processing Performance Council. TPC Benchmark C, standard specification. http://www.tpc.org/tpcc/, 2010.

A Use of Generative AI

In compliance with the TU Delft guidelines on the use of generative AI in BSc end projects, this section discloses the use of large language models (LLMs) during the preparation of this thesis. I used Claude Sonnet 4.6 (medium and high reasoning effort) and Claude Opus 4.7 as assistive tools during the writing and development process, in the following ways:

- **Code understanding and navigation.** The LLMs were used to help navigate and understand parts of the Detock codebase during implementation, as noted in Section 6.
- **Clarity and sentence structure.** Passages I had drafted were rewritten for clarity, with overly dense or too long sentences split into clearer ones.
- **Grammar and language correction.** The LLMs were used to identify and fix grammatical and writing errors in my own drafted text.
- **Consistency checks.** The LLMs were used to verify that references to figures and tables in the text remained accurate after edits and reorganization.
- **Latex syntax.** The LLMs were used to look up and troubleshoot Latex syntax (e.g., formatting, package usage, table/figure layout).
- **Bibliography formatting.** The LLMs were used to help generate `.bib` entries from sources I provided; all entries were subsequently verified against the original publications for venue, year, and page accuracy before inclusion.

In all cases, the LLMs were used as a fast, low-cost first pass to surface easily identifiable issues, not as a source of content, ideas, or analysis. Every suggestion was reviewed and reworked by me; I did not accept output without independently re-checking and rewriting it. The research design, the algorithms and architecture described in Sections 4 and 5, the experimental setup, the analysis of results, and all interpretive claims are my own work. I take full responsibility for the accuracy, originality, and integrity of the content of this thesis.