
Batched Assignment in a Driverless-Taxi Fleet

Applying the State of the Art

SIT314: Software Architecture and Scalability for the Internet-of-Things

1 The Decision I Went Looking For

My project matches ride requests to driverless taxis. The dispatch worker pulls up to eight requests off the queue at once and then throws that grouping away, matching each request on its own to the nearest free car. First come, first served, no reference to the other seven requests in its hand.

That is exactly the arrangement DiDi Chuxing describe as the thing they replaced. Zhang et al. [1] call it Model 1 and note that it "was used at Didi Chuxing until the method of this paper was proposed". Their replacement optimises over a whole batch instead. They ran it live against two production models for a day on Beijing traffic, splitting drivers into three flows by phone number, and global success rate went from 79.8 percent on the old model to 84.4 percent on theirs.

Eom and Toriello [2] give the theory that says when this should and should not help. They study whether to match immediately or hold requests and solve over the pool, and report that on a Manhattan ride-sharing instance "the modified greedy and batching policies are both already within 91% of optimal for a sojourn period of 6, and achieve 95% or better with a sojourn period of 18", while "the straightforward greedy policy achieves less than 70% in all cases". They also say that under batching "agents wait in the system on average for half of the sojourn period", which is the cost side.

Those two together made this the thing to try. One paper says the mechanism works in production, the other says the size of the win depends on how thick the market is, so a null result would be a result rather than an embarrassment.

2 What Does Not Transfer, and Why

DiDi's Algorithm 1 is a hill climbing search. Seed every driver with the order it scores highest on, compute the mean success rate, then for each order walk the drivers not holding it and keep any swap that raises the mean. The objective is

$$E_i = 1 - \prod_j (1 - p_{ij})^{a_{ij}}$$

where p_{ij} is a learned probability that driver j accepts order i . The product is there because an order is broadcast to several drivers and any one of them may take it, so more drivers on an order raises the chance somebody says yes.

I transliterated that first, with p_{ij} replaced by a score derived from the pickup distance since there is no acceptance model in my system. It came out worse than the greedy matcher it was meant to beat. Over 500 batches of eight requests against 200 free cars it averaged 736 metres of pickup distance where greedy averaged 660, which is 11.5 percent worse, and the adapted version below averaged 657.

That is not a bug in their algorithm. A driverless car does not refuse a job, so p_{ij} is effectively 1 once the claim succeeds, the product collapses to a single term, and putting five cars on one request stops being insurance and becomes four cars sent to the wrong place.

So the adaptation is to keep the search and drop the broadcast. Each request gets at most one car and each car at most one request, and E_i becomes the score of whichever car holds request i .

Two things about my version are mine rather than theirs and I should say so. Algorithm 1 seeds by giving every driver its own best order, and mine seeds with the greedy answer, so the search can only improve on what the project already did. And Algorithm 1 reassigns one driver at a time, where mine does a two way swap when the car it wants is already held, which the one to one constraint makes necessary. Both versions are in the assignment library, listed in the code appendix of the project report, and both are measured below, because a claim that the adaptation was needed should have a number behind it.

3 Designing It In

The change is in the dispatch worker. A `MATCH` environment variable picks the policy, so one binary runs both arms and nothing else about the run differs.

```

1 if (MATCH === 'batch') {
2   var deadline = Date.now() + WINDOW_MS;
3   while (running && Date.now() < deadline && msgs.length < MAX_BATCH) {
4     var more = await get(Queue + '/queue/rides/receive?max=' + BATCH);
5     if (more.messages && more.messages.length) msgs = msgs.concat(more.messages);
6     else await sleep(5);
7   }
8   await handleBatch(msgs);
9 } else {
10  await Promise.all(msgs.map(m => handle(m)));
11 }

```

`handleBatch` asks the state service for candidates for every request in the batch, solves the assignment, then claims. A claim can still lose a race to another worker, so a request whose solved car is gone falls back down its own candidate list rather than being dropped.

The load harness already had the number I needed and was throwing it away. The dispatch worker publishes the pickup distance in metres on every match, and the harness subscribed to that topic and discarded the field. Recording it was six lines.

4 Offline, Where the Mechanism Does and Does Not Pay

Before touching the running system I measured the decision on its own. An offline bench builds batches of requests against a fleet with a set share of cars already busy, and runs both policies over identical inputs. That isolates the choice from queueing, claim races and the batch window.

The knob that matters is how many free cars there are per request in the batch.

Batch	Free cars	Cars per req	Greedy m	Batched m	Change	Batches won
8	800	100.0	322	321	−0.1%	5 of 300
8	10	1.3	4,607	4,224	−8.3%	235 of 300
32	800	25.0	325	323	−0.7%	86 of 300
32	10	0.3	5,505	1,838	−66.6%	300 of 300
128	800	6.3	338	332	−1.7%	291 of 300
128	10	0.1	5,427	833	−84.7%	300 of 300

In a thick market the two policies are the same thing. With 800 free cars and eight requests spread over the service area the nearest car to one request is almost never the nearest car to another, so there is nothing to trade and 295 of 300 batches came out identical. As the market thins the greedy order starts to matter, and by 128 requests against 10 free cars greedy is handing cars to whichever requests came first while the search gives each car to the request it is nearest. An 85 percent cut is the size of that mistake, and it is Eom and Toriello's finding on my own instance. Their claim is about sojourn time and mine is about supply against demand inside one batch, and both say greedy is only bad when the pool is small enough for the ordering to matter.

5 End to End, on the Running Platform

The offline result says the mechanism works. The question the project actually needs answered is whether it survives contact with queueing, claim races and a real fleet whose cars come and go.

Same five stage ramp as the final project report, 2,000 vehicles, a matched car held for 300 ms, everything else held still. The greedy arm was run twice and the figures below are the mean of the two, so a difference smaller than the spread between them is not a difference. The batched arm holds requests for up to 250 ms or until it has 64 of them, whichever comes first.

Offered/s	greedy			batched			
	mean m	p95 m	p95 ms	mean m	change	p95 m	p95 ms
200	238	500	29	237	−0.4%	497	283
800	1,189	6,412	34	861	−27.6%	3,528	112
1,600	968	4,260	40	777	−19.7%	3,029	111
3,000	1,044	4,868	94	796	−23.8%	3,065	111
800	1,074	5,154	36	820	−23.6%	3,326	113

One thing the batched arm does that the greedy arm does not, and it is worth putting a number on because it is not free. When a car the solver picked has already gone, the worker falls back down that request’s own candidate list rather than dropping the rider. That happened 683 times against 79,942 matches, so a little under one percent. The greedy arm records zero, because it takes the first car that claims and has nothing to fall back from. The fallback is also unco-ordinated inside a batch: it walks one request’s list without knowing the solver promised one of those cars to another request in the same batch, so a fallback can take a car a batchmate was going to claim. At one percent it does not move the means below, but it does mean the assignment that gets executed is not always exactly the one the hill climb computed, and at a higher claim conflict rate the two would diverge.

Across the four loaded stages the mean pickup distance falls from 1,068 m to 814 m, and the 95th percentile falls further, from 5,174 m to 3,237 m. The rider who was going to wait longest gains most, which is the half of the distribution I care about. Match latency goes the other way, from 51 ms to 112 ms at the 95th percentile.

The first stage is the control and it is the most useful row in the table. At 200 requests a second the fleet is thick, both policies pick the same cars, the pickup distance is identical to within a metre, and the batched arm pays the entire latency cost for nothing at all. That is the offline finding showing up unchanged in the live system.

5.1 The knob that actually moves it

I ran the batch window at 250, 500 and 1000 ms and it barely mattered, 814, 818 and 858 metres across the loaded stages. The window is not what ends a batch at these rates. At 800 requests a second sixty four requests arrive in about 80 ms, so on that arithmetic the size limit is reached long before the window runs out. I did not instrument the batch duration, so that is a calculation and not a measurement. The size limit is the real knob, and sweeping that gives a clean curve.

The rows below hold the window at 2000 ms so it never binds, and are labelled by the batch size the worker actually reached, which is not always the one I asked for. The receive call takes up to eight messages before the size limit is checked, so eight is the floor and asking for four gets eight. That is a bug in my loop and it turned into a free replicate, since the runs at 4 and at 8 are two runs of the same condition. They came out at 1,062 and 1,051 metres, which is eleven metres apart on a figure of about a thousand.

Mean batch	runs	mean m	vs greedy	p95 m	p95 ms
greedy, up to 8 at once	2	1,068	–	5,174	51
8	2	1,056	−1.1%	5,048	97
16	1	1,018	−4.8%	4,595	86
32	1	998	−6.6%	4,564	110
65	1	828	−22.5%	3,336	115
129	1	667	−37.6%	2,416	167

Mean pickup distance falls monotonically with batch size and the total matched barely moves, 80,183 for greedy against 79,847 at a batch of 128, a difference of 0.4 percent. So the batched policy is not buying shorter trips by refusing riders, it is sending the same number of cars to better places. The latency column is noisier than the distance column because a batch that fills early and one that waits out its window behave differently, and I would want more than one run per row before reading much into any single figure in it.

6 The Failure I Found by Sweeping That Knob

The first time I ran that sweep it broke.

The capture below is the harness output with the four latency columns cut out to fit the page. Nothing else is changed and the fifth stage, which is also zero, is left off the bottom.

size128 policy=batch window=2000ms maxBatch=128 fleet=2000 hold=300ms									
stage	asked	actual	sent	matched	unmatched	...	matched/s	mean m	p95 m
1	200/s	198.2	4000	3422	578	...	169.5	525	1634
2	800/s	791.5	16000	4	15996	...	0.2	5909	11277
3	1600/s	1584.6	32000	0	32000	...	0	0	0
4	3000/s	2971	90000	0	90000	...	0	0	0

At a batch of 128 the worker stopped matching anything at all from stage three onwards, and at 64 it fell over during the last stage. The cause is in my own code and not in the algorithm. `handleBatch` asked the state service for candidates one request at a time, in a loop, awaiting each one, so the wall clock cost of a batch grew in proportion to its size. The queue hides a received message for 20 seconds, and a batch that takes longer than that to settle has already lost its claim on the work before it finishes.

The logs show the collapse and not the mechanism, so I should be careful how far I push that. I never instrumented how long a batch took, the 20 second figure is the queue's setting rather than a measurement, and how the run failed to recover is inference. What is measured is that the serial version stopped matching at a batch of 128 and the parallel version does not.

The fix is two lines, issue the lookups together and settle the claims together.

```

1 var cands = await Promise.all(reqs.map(r =>
2   get(STATE + '/candidates?lat=' + r.lat + '&lng=' + r.lng + ...)
3   .then(c => c.candidates || []).catch(() => [])));
4 var picked = assign.hillClimb(reqs, cands);
5 await Promise.all(msgs.map((m, j) => settle(m, picked[j], cands[j])));

```

Every number in the tables above is from after that change. The algorithm was fine at every batch size I tried, before and after. What failed was a serial loop inside a service whose input size I had just made variable, and it failed in the way this unit spends six weeks describing, by pushing work past a timeout somewhere else in the system and turning a slowdown into a retry storm. That is worth more to me than the headline result.

7 What This Actually Says, and What It Does Not

Batching is worth having on this system and it is worth having for a reason the project only found by measuring. My fleet is thick at low load and thin at high load, and the thin case is where riders wait longest, so a policy that only helps in the thin case is helping exactly where it matters. The cost is a match latency that roughly doubles, which for a taxi is a fair trade against 250 metres less driving, and would not be for something where the answer has to come back fast.

Four things I would not claim from this.

My score is a distance. DiDi's whole first stage is a learned model of whether a human driver takes the job, and I deleted it because my cars cannot refuse. So I have implemented their optimiser and not their system, and the 84.4 percent success rate in their paper is not a number my result can be compared against.

Everything runs on one laptop with no network between any two tiers. Both policies issue their lookups concurrently, greedy in groups of up to eight and batched in groups of up to 64, so a real VPC hop between the worker and the state service would be amortised over more requests by the batched one. Every run in this report is local, so I have not measured that, and I would expect the trade to look better with a network in the path.

The hill climb is a local search and it can lose on one batch while winning on average. In the offline sweep at 32 requests against 40 free cars it was better overall by 10.8 percent and still worse than greedy on 43 batches out of 300. The greedy row of each live table is the mean of two runs and every batched row is one run, so a gap of a percent or two between two batched rows is inside the noise even where the trend across them is not.

And the whole thing sits downstream of the finding in the project report, that the binding constraint at 3,000 requests a second is the number of cars and not the matching service. Batching makes the cars I do have go to better places, it does not make more of them.

References

- [1] L. Zhang, T. Hu, Y. Min, G. Wu, J. Zhang, P. Feng, P. Gong and J. Ye, *A Taxi Order Dispatch Model based On Combinatorial Optimization*, KDD '17, Halifax NS, Canada, August 2017.
- [2] M. Eom and A. Toriello, *Batching and Greedy Policies: How Good Are They in Dynamic Matching?*, Manufacturing and Service Operations Management 28(2), 2026. I read the author manuscript dated 26 August 2025, not the journal version, so the figures I quote are the manuscript's.