

Cloudflare — Systems Engineer, Hyperdrive

45-minute hiring manager interview · prep sheet

What this is. A hiring manager round is not a softball. At Cloudflare it is run *early* in the loop — earlier than most companies — and it is where the manager decides whether to spend four more engineers' time on you. The single highest-leverage thing you can do is know what Hyperdrive actually is and be able to talk about it like someone who has thought about the problem.

The thesis: 45 minutes with the hiring manager is about **architectural judgement**, not algorithms. They want to hear you justify decisions you made in the past, and they want evidence you understand the problem their team owns. Everything in Section 2 is free points — it is public documentation that most candidates will not have read.

The shape of the round

Minute	What should be happening
0–5	Intros. He describes the team. Take notes — you will reuse his words later.
5–20	Your background. A tight narrative, not a resume recital.
20–35	Deep dive on one project. This is the round. Expect “why did you do it that way?” three levels down.
35–42	Team fit, what you want next, logistics.
42–45	Your questions. Have three, ask two.

Cloudflare's loop after this: a technical phone screen (45–60 min, CoderPad, one medium DSA problem or a practical task like a rate limiter), then a virtual onsite of three to five rounds — deep-dive coding, system design, behavioural, and often a project retrospective. Some teams substitute a take-home. Sections 4 and 5 of this sheet cover those.

1. Know Hyperdrive cold

Hyperdrive makes an existing regional Postgres or MySQL database feel global to Cloudflare Workers. You point it at a database you already have — RDS, Neon, PlanetScale, CockroachDB — change the connection string, and queries from anywhere on Cloudflare's 300+ locations get faster. No new database, no rewrite.

The problem it solves

A Worker is stateless and short-lived. A traditional database connection is expensive to create and expensive to hold. Those two facts are in direct conflict:

- Establishing a Postgres connection takes **~7 round trips** — TCP handshake, then TLS negotiation, then authentication. At 50–200ms per round trip to a database on another continent, you can burn a second before you have sent a single query.
- Postgres allocates a **process (and non-trivial memory) per connection**. A few hundred connections is already a real cost. Globally distributed traffic would exhaust `max_connections` almost immediately.
- So the connection is both too slow to create *and* too expensive to keep. That is the whole problem in one sentence — and it is a good sentence to be able to say out loud.

The three mechanisms

Mechanism	What it does
1. Edge connection setup	The driver's handshake happens between the Worker and Hyperdrive <i>in the same Cloudflare location</i> , not across an ocean. The expensive multi-round-trip part becomes local.
2. Connection pooling	Hyperdrive keeps warm pools near the origin database , not near the user. A cache-missing query costs one round trip, not seven. It also caps total connections to the origin so the database is never overwhelmed.
3. Query caching	Hyperdrive parses the query , decides whether it mutates state, and caches the results of non-mutating ones for a configurable <code>max_age</code> .

Reported effect: **17–25x** faster for cached queries, **6–8x** for uncached queries and writes. The second number is the interesting one — it is what turns “you cannot query a central database from the edge” into something people actually do.

The asymmetry is the insight. Connection setup is moved *close to the Worker*; the connection pool is kept *close to the database*. Those are opposite directions, and each is right for what it optimises. If you can explain why in your own words, you sound like someone who belongs on this team.

The sharp edges — bring these up yourself

1. **Cache invalidation is not automatic.** Hyperdrive does *not* invalidate cached reads when your application writes. If you need read-after-write consistency you have to bypass or scope the cache. This is a deliberate trade-off, and noticing it unprompted is the strongest single signal you can give.
2. **Transactions pin a connection.** You cannot multiplex a pool across an open transaction — session state (temp tables, SET, prepared statements, advisory locks) belongs to one backend. This is the classic session/transaction/statement pooling trade-off from PgBouncer, and it is why transaction-level pooling breaks named prepared statements.
3. **Classifying a query requires parsing the wire protocol.** Deciding read vs write means understanding Postgres' simple and extended query protocols, not just regexing for SELECT. A SELECT can call a volatile function that writes. That is a genuinely hard systems problem and a great thing to be curious about out loud.
4. **Placement.** If a Worker makes several *sequential* queries per request, each one pays the round trip; Smart Placement can move the Worker next to the database (20–30ms per query becomes 1–3ms). If it makes exactly one query, placement buys nothing — the total path is the same either way. Knowing when it does *not* help is the more interesting half.
5. **Multi-tenancy.** One customer's pool must not starve another's, and credentials to customer databases have to be held securely at the edge.

Vocabulary to have loaded

- **Pooling modes:** session (one client = one backend for the whole connection), transaction (returned to the pool at COMMIT), statement (returned after every statement, no transactions allowed).
- **Postgres wire protocol:** startup message, auth (SCRAM-SHA-256), simple query vs extended query (Parse / Bind / Execute), ReadyForQuery.
- **Anycast:** the same IP announced from every Cloudflare location via BGP; the network routes you to the nearest one. This is why “the edge” is close to everyone.
- **TLS 1.3** cuts the handshake to one round trip; 0-RTT resumption to zero, at the cost of replay exposure for non-idempotent requests.

2. What he will ask you

Cloudflare's hiring managers optimise for practical engineering judgement. The questions are ordinary; the follow-ups are where it is decided.

Near-certain

- **“Walk me through the most technically complex thing you have built.”** Pick one with real constraints. Lead with the constraint, not the tech.
- **“Why did you choose X over Y?”** Asked recursively. Have a real answer at three levels of depth, and be willing to say “that was the wrong call, here is what I would do now.”
- **“Tell me about a production incident you owned.”** Detection, diagnosis, mitigation, the permanent fix, and what you changed so the class of bug could not recur. Cloudflare publishes unusually candid post-mortems — they respect this genre.
- **“What is the hardest bug you have debugged?”** Best answers involve a system you did not write and a hypothesis you disproved.
- **“Why Cloudflare? Why this team?”** See below — do not wing this one.

Likely, for a systems role specifically

- Performance work you have done: **how did you measure it**, what was the baseline, what did you learn was actually slow? (Answers that skip measurement read as guessing.)
- A time you had to reason about something across a network boundary — latency, partial failure, retries, timeouts.
- How you decide between fixing something properly and shipping the workaround.
- How you have handled being the person who has to say no to a design.

“Why Cloudflare, why Hyperdrive?”

A good answer is specific and technical. Something in the shape of: the interesting part of Hyperdrive is that it is not a new database — it is a compatibility layer that lets people keep the database they already have while changing the physics of accessing it. That is a systems problem (connection lifecycle, protocol parsing, cache semantics) rather than a product problem, and the constraint of being invisible to existing drivers and ORMs makes it harder and more interesting.

Then connect it to something you have actually done: any work on connection management, caching with correctness constraints, protocol handling, or latency

measurement is directly relevant. If you have none, say what is adjacent and be honest — Cloudflare's stated values are principled, curious and transparent, and manufactured enthusiasm reads badly against all three.

Three questions to ask him

- “Hyperdrive does not invalidate cached reads on write — how do you think about giving people read-after-write without giving up the cache hit rate?”
- “Where does the pool actually live as the origin database moves, or when a customer has read replicas in several regions?”
- “What breaks first as Hyperdrive scales — is it connection count at the origin, memory at the edge, or protocol coverage for the long tail of drivers?”

All three come out of the public documentation. All three are real engineering questions with no marketing answer, which is exactly why they land.

3. Your story, prepared

The 90-second opener

He will open with some version of “tell me about yourself.” Do not narrate your resume top to bottom. Use this shape:

1. **One sentence of frame.** What kind of engineer you are and what you are drawn to. (“I work on backend systems where the interesting constraint is latency or correctness under failure.”)
2. **Two or three beats of trajectory**, each one a problem and an outcome, not a job title and a date range.
3. **One sentence about why you are in this conversation.** Land on Hyperdrive specifically.

The deep-dive project — pick it now

Choose one project you can defend three levels deep. Before the call, write down:

- The **constraint** that made it hard. Not “we built a service” but “we had to do it in under 50ms at the 99th percentile” or “we could not take the write path down.”
- The **alternative you rejected** and why. Every good design has one.
- **What you measured**, and the number before and after.
- **What went wrong**, and what you would do differently. Volunteering this is a strength signal, not a weakness one.
- **What you owned** versus what the team owned. Be precise and generous.

The trap in this round is answering at the level of “we used Kafka.” A hiring manager for a systems team is listening for whether you know *why* the thing you used behaves the way it does. Always be ready to go one layer below whatever you just said.

4. The coding rounds

From Cloudflare-tagged frequency data, one pattern dominates: **implement a data structure with an exact API and awkward edge cases**. That is not a coincidence for an infrastructure company — ring buffers, caches and counters are what their code is made of.

Problem	Freq	Why it fits
Design Circular Queue	100	Ring buffer. #1 all-time.
3Sum / 4Sum	94	Two pointers, dedup edge cases.
Design a Stack With Increment	84	Lazy propagation trick.
Reaching Points (Hard)	84	Math / work backwards.
Design Hit Counter	75	Sliding window over a stream.
Number of Islands	70	Flood fill.
LRU Cache	64	Plus concurrency follow-ups.
Game of Life	56	In-place grid simulation.

Design Circular Queue — type this once

```
class MyCircularQueue:
    def __init__(self, k: int):
        self.q = [0] * k
        self.cap = k
        self.head = 0          # index of the front element
        self.size = 0          # count, so full/empty are unambiguous

    def enqueue(self, value: int) -> bool:
        if self.size == self.cap:
            return False
        self.q[(self.head + self.size) % self.cap] = value
        self.size += 1
        return True

    def dequeue(self) -> bool:
        if self.size == 0:
            return False
        self.head = (self.head + 1) % self.cap
        self.size -= 1
        return True

    def Front(self):
        return -1 if self.size == 0 else self.q[self.head]

    def Rear(self):
        if self.size == 0:
            return -1
        return self.q[(self.head + self.size - 1) % self.cap]

    def isEmpty(self): return self.size == 0
```

```
def isFull(self): return self.size == self.cap
```

Keep a size counter. The alternative — head and tail pointers only — cannot distinguish full from empty without wasting a slot or adding a flag. Say that trade-off out loud; it is the entire point of the question, and it is exactly the decision you make writing a real packet ring buffer.

Design a Stack With Increment — the lazy trick

The naive increment is $O(k)$. The intended answer is $O(1)$ by deferring the work to pop.

```
class CustomStack:
    def __init__(self, maxSize: int):
        self.stack = []
        self.cap = maxSize
        self.pending = [0] * maxSize # pending[i] applies to stack[i]
                                     # and everything below it

    def push(self, x: int) -> None:
        if len(self.stack) < self.cap:
            self.stack.append(x)

    def increment(self, k: int, val: int) -> None:
        i = min(k, len(self.stack)) - 1
        if i >= 0:
            self.pending[i] += val # one write, not k writes

    def pop(self) -> int:
        if not self.stack:
            return -1
        i = len(self.stack) - 1
        if i > 0:
            self.pending[i - 1] += self.pending[i] # push it down
        val = self.stack.pop() + self.pending[i]
        self.pending[i] = 0
        return val
```

This is the same idea as a lazy segment tree, and the same idea as deferring cache invalidation until read. Naming that connection is worth more than the code.

LRU Cache — and the follow-up that actually matters

```
from collections import OrderedDict
```

```
class LRUCache:
    def __init__(self, capacity: int):
        self.cap = capacity
        self.d = OrderedDict()

    def get(self, key: int) -> int:
        if key not in self.d:
            return -1
        self.d.move_to_end(key) # most recently used = last
        return self.d[key]

    def put(self, key: int, value: int) -> None:
```

```

if key in self.d:
    self.d.move_to_end(key)
self.d[key] = value
if len(self.d) > self.cap:
    self.d.popitem(last=False) # evict least recently used

```

Offer the hand-rolled doubly-linked-list version if they want it — a dict from key to node, plus head and tail sentinels so no unlink needs a null check. But expect the real question to be the follow-up:

- **“Make it thread-safe.”** A single global lock serialises every get, and get is a *write* to the recency order, so a plain read-write lock does not help. Options: shard the cache by key hash into N independently locked segments; or approximate recency (CLOCK / second-chance) so reads only set a bit.
- **“What if it does not fit in memory?”** Bounded by bytes rather than entries, admission control so one huge object cannot evict everything (TinyLFU), and a note that LRU is pathological for scans — one sequential pass evicts your whole working set. That is why real caches use LRU-K, ARC or segmented LRU.
- **“Now make it distributed.”** Consistent hashing with virtual nodes; the interesting failure is that eviction and invalidation are per-node, so you get divergence. Straight to the Hyperdrive problem, which is a good place to end up.

3Sum — get the dedup right

```

def threeSum(nums):
    nums.sort()
    res, n = [], len(nums)
    for i in range(n - 2):
        if nums[i] > 0: # sorted: no triple can sum to 0
            break
        if i and nums[i] == nums[i - 1]: # skip duplicate anchors
            continue
        lo, hi = i + 1, n - 1
        while lo < hi:
            s = nums[i] + nums[lo] + nums[hi]
            if s < 0:
                lo += 1
            elif s > 0:
                hi -= 1
            else:
                res.append([nums[i], nums[lo], nums[hi]])
                lo += 1
                while lo < hi and nums[lo] == nums[lo - 1]:
                    lo += 1 # skip duplicate seconds
                hi -= 1
    return res

```

$O(n^2)$ time, $O(1)$ extra space beyond the sort. The whole difficulty is the two dedup steps; write them deliberately and say why each is there.

Also worth a glance

- **Design Hit Counter** — deque of timestamps, then bucket it into a fixed ring of 300 slots when they ask about high traffic. Same shape as Circular Queue.
- **Game of Life** — in place, by encoding the next state in a second bit ($state \mid = next \ll 1$) and shifting at the end. The follow-up is always “what if the board does not fit in memory” → process it in row-strips, keeping only three rows live.
- **Reaching Points** — work backwards from the target with modulo instead of forwards with recursion. Forwards times out; that is the entire question.

5. System design

The Cloudflare-specific rule: candidates report being asked to design **without leaning on managed cloud services**. “I would use S3 and SQS” is close to a failing answer here — Cloudflare *is* the infrastructure. Reach for primitives and things you would run yourself: Kafka, ClickHouse, RocksDB, consistent hashing, gossip, Raft.

Prompts reported at Cloudflare

- A **globally distributed key-value store** (i.e. Workers KV).
- A **DDoS mitigation system**.
- A **rate limiter inside an HTTP proxy**, at edge scale.
- A **log collection pipeline** spanning 300+ edge data centres.
- A **connection pooler** — that is Hyperdrive. For this team specifically, assume it is on the table.

If you get “design Hyperdrive”

1. **Clarify:** which protocols? read-heavy or write-heavy? is stale data acceptable, and for how long? how many origin databases, and how many tenants?
2. **Data path:** Worker → nearest edge PoP (terminate the driver handshake here) → regional pool near the origin → database. Say why the pool is near the origin and the handshake is near the user.
3. **Query classification:** parse the protocol, not the string. Reads are cacheable, writes are not, and anything you cannot classify confidently must be treated as a write. Failing safe is the correct default and worth stating.
4. **Cache:** key on (database, normalised query, parameters). TTL-based, because invalidating on write would require understanding which rows every write touches — that is the honest reason the real product does not do it.
5. **Pool management:** cap connections per origin; queue when saturated; evict idle connections; pin a connection for the duration of a transaction and do not return it to the pool until COMMIT or ROLLBACK.
6. **Failure modes:** origin down (fail fast, do not queue forever), pool region down (re-route, cold pool), poison query, one tenant saturating shared capacity.
7. **Observability:** cache hit rate, p50/p99 by cached/uncached, connections in use per origin, queue depth. Then say which of those you would page on.

Numbers to have in your head

Operation	Order of magnitude
L1 / main memory reference	1ns / 100ns
SSD random read	~100µs
Same-datacentre round trip	~0.5ms
Cross-country round trip (US)	~30–70ms
Cross-continent round trip	~100–200ms
TCP + TLS + auth to Postgres	~7 round trips
TLS 1.3 full handshake / 0-RTT	1 round trip / 0

Fundamentals they may probe

- **Anycast and BGP** — how one IP is served from 300 locations, and what happens to a TCP connection when routing shifts mid-flow.
- **HTTP/2 vs HTTP/3** — head-of-line blocking at the TCP layer is why QUIC exists; QUIC also folds the transport and crypto handshakes together.
- **Consistent hashing** — and why you need virtual nodes for balance.
- **Consistency models** — be able to distinguish eventual, read-your-writes, and linearizable, and say which one a cache in front of a database actually gives you (it is the weak one, and that is the trade).
- **Backpressure** — what your system does when it is over capacity. “Queue it” is a wrong answer without a bound.

The last five minutes

Do

- **Lead with constraints.** Every good answer starts with what made the problem hard, not what technology you picked.
- **Quantify.** p99 latency, request rate, data volume, the before and after. Systems people trust numbers and distrust adjectives.
- **Volunteer a mistake.** One well-told “we got this wrong and here is what I learned” does more for you than three clean successes.
- **Go one layer deeper than asked,** once. Not on everything — but pick the moment and show the floor is further down than you have shown so far.

Don't

- Don't say “we just used *[managed service]*” without being able to explain what it does for you. At Cloudflare that is the answer they are testing for.
- Don't claim ownership that was not yours. Managers verify, and generosity about credit reads as seniority.
- Don't bluff on Hyperdrive internals. “I have read the public docs, I have not worked on connection pooling directly, here is the closest thing I have done” is a strong answer. Inventing detail is fatal in a systems role.
- Don't leave without asking your questions. Not asking reads as not caring which team you land on.

If you only do one thing: be able to explain, in your own words, why the connection setup moves to the user and the connection pool stays near the database. That single answer signals more than any amount of resume.