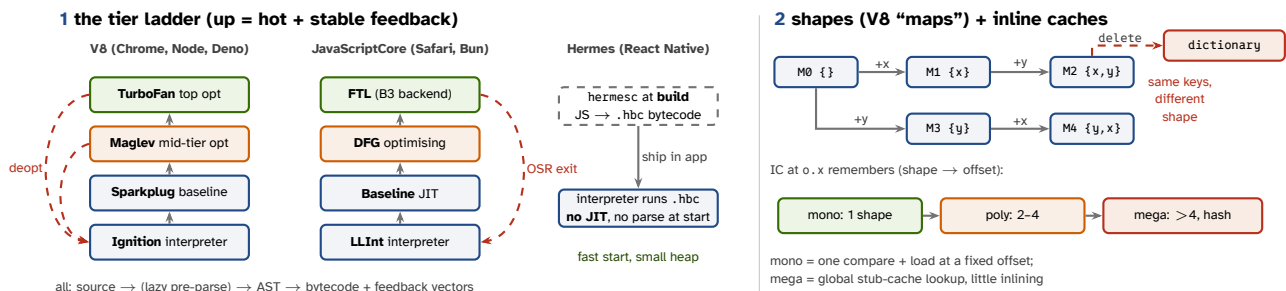


In one line: An engine parses to **bytecode**, **interprets** it while collecting **type feedback**, and JIT-compiles hot code in **tiers** that **speculate** on the shapes it saw — a failed guard **deoptimises** back to the interpreter. Memory is a **tracing, generational GC**: reachability, not reference counts.



How it works — the pipeline

- Lazy parsing:** inner functions only pre-parsed until first call. Each call site / property access records **type feedback** in a feedback vector.
- Tiers** trade compile time for speed: Sparkplug (Chrome 91) copies bytecode to machine code without optimising; Maglev (Chrome 117) is a fast SSA optimiser; TurboFan inlines, removes checks, unboxes numbers — *speculatively*. Long loops tier up mid-run via **OSR** (on-stack replacement).
- Deopt:** optimised code guards its assumptions (map check, “is Smi”, bounds); a failure rebuilds the interpreter frame and continues there; repeated deopts → no more optimising. Spot: node --trace-opt --trace-deopt (*wrong map, not a Smi*).
- Hermes:** best TTI and memory on phones, lower peak CPU than a JIT. An in-app JSC on iOS gets **no JIT** either (no writable+executable pages without Apple’s entitlement); Safari/WKWebView JIT in a separate process.
- Shapes:** same keys in the same **order** → one hidden class; a key added later = a transition; delete (non-last key) usually → **dictionary mode**. Field types generalise too (Smi → double → tagged).
- Elements kinds** (one-way lattice): PACKED_SMI → PACKED_DOUBLE → PACKED_ELEMENTS, each can turn HOLEY_*. new Array(n), writing past the end, -0/NaN into ints degrade it, for good. Smi = 31-bit int (pointer compression). Hot numeric data: typed arrays.

Example — shape-friendly vs not

```
class P { constructor(x, y) { this.x = x; this.y = y; } } // 1
const a = { x: 1, y: 2 }, b = { y: 2, x: 1 }; // 2 shapes
const getX = (o) => o.x; // IC here
getX(a); getX(b); // polymorphic now
a.z = 3; // new transition M2 -> +z
delete a.x; // dictionary: slow props
const arr = [1, 2, 3]; // PACKED_SMI
arr.push(1.5); // PACKED_DOUBLE (for good)
arr[10] = 0; // HOLEY_DOUBLE
```

Garbage collection

- Tracing:** live = reachable from roots (stack, globals, handles). Cycles die together — no weak needed, unlike **ARC** (deterministic, cycles leak). Price: no deinit timing, pauses, headroom.
- Generational** (V8 Orinoco): most objects die young. Young gen = **scavenger** (semi-space copy, parallel; survivors promoted); old gen = **mark-sweep-compact**, marking **incremental + concurrent** (write barriers), sweeping concurrent/lazy. Hermes: **Hades**, mostly concurrent old gen.

- JS leaks = unwanted reachability:** listeners/subscriptions never removed (RN: no useEffect cleanup), setInterval, unbounded Map caches, globals, **detached DOM** held by JS, closures — closures of one scope share one context, so a tiny callback can pin a huge sibling capture.
- Weak** (ES2015/2021): WeakMap/WeakSet — keys held weakly (ephemeron), not iterable: per-object side tables. WeakRef.deref() may be undefined; FinalizationRegistry callbacks may *never* run — cleanup hints, not logic.

Measuring

- Heap snapshot** (DevTools Memory / node --inspect / v8.writeHeapSnapshot()): shallow vs **retained** size, the **retainers** path (who keeps it alive) ≈ Xcode Memory Graph. Leak hunt: snapshot, repeat the action, snapshot, **Comparison** view; filter “Detached”.
- Allocation timeline / sampling:** who allocates, what survives. process.memoryUsage(): heapUsed vs rss vs external (Buffers). CPU: --cpu-prof, Performance panel ≈ Time Profiler; time with performance.now().

Interview traps

- ! Microbenchmarks lie: dead code eliminated, no warm-up (timing the interpreter), monomorphic in the bench but not in prod, GC in one run.
- ! “try/catch / arguments kill optimisation” — Crankshaft-era folklore; TurboFan handles them. Measure before believing.
- ! delete obj.k frees nothing and slows the object; set undefined or use a Map for dynamic keys.
- ! “JS can’t leak, it has a GC” — it leaks by reachability.

Remember

Same keys, same order, same types — and let go of what you subscribed to.

Likely questions

- What is a deopt? — a failed speculation guard; execution resumes in the interpreter.
- Why is Hermes fast without a JIT? — no parse/compile at startup, small heap, AOT bytecode.
- Mono vs mega? — fixed-offset load vs hashed lookup.
- Find a leak? — 3 snapshots, comparison view, follow retainers to the root.