



Chandan Gowda

chandan@hackculture.in

Assessment Motor Policy & Claims Engine Stack JavaScript · Node.js Completed Jul 4, 2026



Recommendation

Hire

Confidence 82%

Reviewer summary

- A strong, ships-it-clean candidate who works with AI **deliberately** rather than blindly. Planned before building, drove the solution with a tight test loop, and committed in reviewable increments. The one failing hidden case — the No Claim Bonus is discounted off the add-on covers as well as the own-damage premium — plus a thin patch of input validation are the main gaps; both were acknowledged, unprompted, during the interview. Comfortably clears the **senior backend bar** for this role.
- ✓ Spec-driven: reviewed and revised the AI plan before writing any code, and pushed back on one over-engineered suggestion.
 - ✓ Disciplined test loop: wrote his own edge-case tests ahead of the hidden suite and committed 7 times incrementally.
 - ⚠ Applied the NCB discount to the whole premium, so every policy with an add-on is under-billed; validation on IDV and claim amounts is inconsistent.
 - ✓ Interviews well: explained trade-offs in his own words and spotted the add-on discounting bug without a hint.

Scorecard

Coding Test	84
hidden + sample suite	
Code Quality	72
review + conventions	
Interview	76
role + codebase Q&A	
AI Fluency	88
prompting & workflow	

✓ Positive

6 signals

- ✓ **Plans before building.** Opened in plan mode, spent 6m mapping the pricing pipeline and the policy record before the first edit.
- ✓ **Owns the AI output.** Rejected a suggested premium cache as premature and made 34% of edits by hand, not blind acceptance.
- ✓ **Test-first instinct.** Authored 9 of his own test cases (incl. a claim equal to the IDV, and the 50% NCB cap) before running the sample suite.
- ✓ **Reviewable git history.** 7 small, well-messaged commits, one logical change each, not a single dump at the end.
- ✓ **Reads the codebase.** 68% of prompts referenced specific files or line ranges rather than pasting whole files.
- ✓ **Rate card as data.** Kept base rates, zone loadings and add-on prices in one lookup table instead of scattering them through the pricing path.

⚠ Concerns

4 signals

- ⚠ **Discount applied to the wrong base.** One hidden test fails: the NCB is taken off the add-on covers as well as the own-damage premium, so every policy with an add-on is under-billed.
- ⚠ **Inconsistent error handling.** Unknown-policy paths sometimes throw, sometimes return null, with no single error contract.
- ⚠ **Light on input validation.** Negative or non-integer IDVs and claim amounts are not rejected at the service boundary.
- ⚠ **Doc lookups skew shallow.** Referenced Node docs once; leaned on the model for API recall rather than verifying edge behavior.

79% match

 against the job-description requirements

Service / module API design Clean module boundaries, sensible method surface	88%
Domain modeling Policy record and rate card cleanly separated	82%
Error handling & resilience Gaps in error contract and validation	58%
Money & rounding correctness Right rounding steps, discount taken off the wrong base	62%
Testing discipline Wrote own tests, TDD-leaning loop	84%
AI-assisted workflow Deliberate, plan-first, reviews output	90%
Communication Clear, structured, honest about gaps	80%

Strengths & development areas

✓ Strengths

Deliberate AI workflow

Plans in spec mode, questions suggestions, and hand-edits where it matters, exactly the senior behavior the role wants.

Test-driven by habit

Reached for tests before the grader forced it; his own suite caught two bugs he then fixed.

Treats pricing rules as data

Rate card in one table, pricing logic reading from it — a change of tariff is a data edit, not a rewrite.

Self-aware

Named his own weak spots (the discount base, validation) before we asked. Low ego, high signal.

— Development areas

Reads the spec, not the invoice

Implemented the formula in the order written without asking what each line is charging for, which is how the add-on discount slipped through.

Defensive coding

Boundary validation and a single error contract are missing, so this would need review before prod.

Verification depth

Trusts model recall over docs for edge cases; occasionally ships an unverified assumption.

92%

Feature completion. Hidden suite 6 / 7, sample 5 / 5.
One hidden failure: the No Claim Bonus is discounted off the add-on covers too.

Feature completion by requirement

issuePolicy & policy record	100%
premium calculation	85%
add-on covers	70%
claims & IDV limits	95%
NCB renewal rules	100%
input validation	60%

Signals & metrics

Time to first edit read statement + explored files	6m 04s
Total active time	38m 12s / 45m
Commits incremental, one change each	7
Manual vs AI edits surgical hand edits	34% / 66%
Own tests authored ahead of the hidden suite	9
Sample suite runs fixed forward	4
Token usage well under the hard limit	142k / 200k

Test results sample (visible) + hidden (graded)

✓	issuePolicy then premiumFor prices a fresh car off IDV, rate and zone	sample	4ms
✓	zone B applies no loading and bikes use the bike base rate	sample	3ms
✓	every policy-scoped method throws UNKNOWN_POLICY for an unknown id	sample	3ms
✓	renew walks up the NCB slabs, returns the new value and caps at 50	sample	4ms
✓	a claim leaves this year untouched and costs the NCB at the next renewal	sample	3ms
✓	issuePolicy rejects invalid idv, vehicleType and zone without creating a policy	hidden	5ms
✓	issuing the same policyId twice throws DUPLICATE_POLICY and keeps the original	hidden	4ms
✓	addAddOn is idempotent and rejects unknown add-on codes	hidden	4ms
✓	the claim year resets at renewal — an old claim cannot punish a later year	hidden	5ms
✓	fileClaim validates the amount and records nothing when it exceeds the IDV	hidden	4ms
✓	highRiskReport filters strictly above the threshold and sorts by amount desc then id	hidden	6ms
✗	add-ons are charged in full — the NCB discount never applies to them	hidden	5ms
↳ Car at IDV 6,00,000 in zone A, NCB 20, with ZERO_DEP attached: returned 16800, expected 17040. The 20% bonus was taken off the add-on price as well as the own-damage premium.			

How they got there plan → build → test

Step 1 Planning strong

Entered plan mode and spent ~6 min before touching code. Sketched the policy record (policyId → { idv, rating factors, addOns, ncb, claims }) and the order of the pricing steps, then trimmed the plan when the model over-scoped it.

Plan mode: used Spec-driven Questioned AI once Time to understand: 6m 04s

Step 2 Build solid

TDD-leaning: wrote failing tests first for the core paths. 34% of edits were manual surgical changes; the rest were file-scoped AI edits referencing src/policy.js. Referenced Node docs once. No CLAUDE.md / skill files used.

Strategy: TDD-leaning Manual edits: 34% File-scoped prompts: 68% Docs referenced: 1
Incremental commits: 7

Step 3 Test

solid

Ran the sample suite 4 times, fixing forward each time rather than guessing. Sample went green at 24m, which is exactly why the add-on case went unnoticed — nothing visible exercised a discounted policy with a cover attached.

Sample runs: 4 Iterative, not thrashing Hidden fail: discount base Fixed forward

Code quality

Folder structure	90%
Naming conventions	86%
Readability	82%
Error handling	55%
Tests authored	84%

Authenticity check faking to pass tests?

Low risk

No sign of hard-coding to pass tests. The rate card is read from a table rather than special-cased per vehicle, variable names are meaningful, and the solution structure matches the reasoning shown in the prompt transcript. The code was built, not curve-fit to the visible cases.

Code review highlights

ISSUE

NCB discounted off the add-on covers in `premiumFor()`

The bonus is applied to the running total after the add-on prices have been added, so a customer with ZERO_DEP gets 20% off their add-on too. Discount the own-damage premium first, then add the covers at full price. This is the failing hidden test, and it under-bills every policy with a cover attached.

ISSUE

Inconsistent unknown-policy contract

`premiumFor()` throws UNKNOWN_POLICY, but `getNcb()` returns null for the same case. Pick one contract (throw a typed error, or Result-style) and apply it everywhere.

NIT

Boundary validation missing

IDV and claim amounts are trusted. Negative or fractional inputs mutate the policy silently. Validate at the public method boundary.

PRAISE

Clean domain error type

Introduced a `PolicyError` with a machine-readable code, a good instinct for a callable service API.

PRAISE

Rate card kept as data

Base rates, zone loadings and add-on prices live in one lookup table, so a tariff revision is a data change rather than a rewrite of the pricing path.

src

policy.js

+128

errors.js

+22

tests

policy.own.test.js

+64

README.md

src/policy.js

javascript

```
1 import { PolicyError } from './errors.js';
2
3 // One record per insured vehicle, keyed by policy id.
4 export class PolicyEngine {
5   constructor() {
6     this.policies = new Map();
7   }
8
9   getNcb(policyId) {
10     const p = this.policies.get(policyId);
11     return p ? p.ncb : null;
12   }
13
14   premiumFor(policyId) {
15     const p = this.policies.get(policyId);
16     if (!p) throw new PolicyError('UNKNOWN_POLICY');
17     const base = Math.round(p.idv * BASE_RATE[p.vehicleType]);
18     const loaded = Math.round(base * (1 + ZONE_LOADING[p.zone]));
19
20     let total = loaded;
21     for (const code of p.addOns) total += ADD_ONS[code];
22     return total - Math.round(total * p.ncb / 100);
23   }
24
25   fileClaim(policyId, amount) {
26     const p = this.policies.get(policyId);
27     if (!p) throw new PolicyError('UNKNOWN_POLICY');
28     if (amount > p.idv) throw new PolicyError('CLAIM_EXCEEDS_IDV');
```

▲ Returns null for an unknown policy, but premiumFor() throws for the same case — two contracts for one condition.

▲ The NCB is taken off 'total', which already includes the add-on prices. It should discount 'loaded' only, then add the covers at full price. This is the failing hidden test.

```
29     p.claims.push(amount);
30     p.claimsThisYear += 1;
```

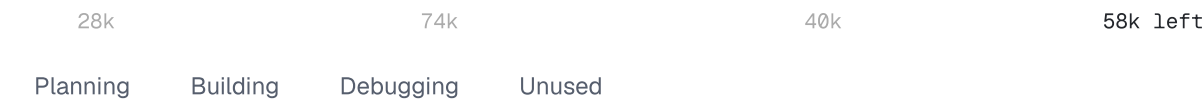
▲ No check that amount is a positive integer, so a negative claim would be recorded and would still cost the customer their bonus.



Workflow strategies detected

- Plan-first
- TDD loop
- File-scoped edits
- Reviews AI output
- Incremental commits
- Questions suggestions

Token usage 142k of 200k budget



Interaction stats

Total prompts	23
Avg prompt length	41 words
Prompts referencing files	68%
Plan vs build vs debug	22% / 48% / 30%
Manual edits after AI	34%
Suggestions rejected	2

User-prompt transcript phase-tagged · file references highlighted

00:41 **PLAN**

3.1k tokens

Before coding, outline a plan for a motor policy engine: the policy record, where the rate card lives, and the order of the pricing steps. Don't write code yet.

05:12 **PLAN**

2.4k tokens

Skip the premium cache for now, it's premature. Just the in-memory Map of policies we discussed.

07:38 **BUILD**

6.8k tokens

In `src/policy.js` implement `issuePolicy` and `premiumFor` per the plan. Read every rate from the lookup tables, don't inline the numbers.

12:02 **BUILD**

7.2k tokens

Add `fileClaim` and `renew` in `src/policy.js:24-60`. Throw a typed `PolicyError` with a code, don't return null.

18:44 **TEST**

5.0k tokens

Write tests for a claim equal to the IDV and for the NCB stopping at the 50% slab. I want these failing first.

26:19 **DEBUG**

4.6k tokens

`highRiskReport` is including policies that never claimed. Fix the filter in that method only.

33:50 **DEBUG**

5.3k tokens

Rounding check: should the NCB come off before or after the add-ons are added? Reading the statement again. (thinking out loud, low on time)



Answered 5 / 6 Set Backend · Senior (Group B of 3) Duration 13m 40s

Mixed MCQ, free-text, and code questions from a role/JD library plus dynamic questions generated from the candidate's own code. Push-to-talk answers were transcribed, with no AI TTS.

FREE TEXT Q1

✓ 8 / 10

Walk me through the internal data structure you chose to hold a policy — its rating factors, its add-ons, its NCB slab and its claim history. What would you change if this engine had to hold ten million live policies?

CANDIDATE ANSWER

A Map from policy id to a flat record: idv and the two rating factors, a Set of add-on codes, the current ncb percent, a lifetime claims array and a counter for claims since the last renewal. Flat because every pricing read touches almost all of it, and a Set because add-ons are membership, not a list — that is what makes attaching the same cover twice a no-op for free. At ten million policies none of this stays in process memory; the record goes to a row keyed by policy id, and I would keep the pricing function pure so it can be called from anywhere with a record passed in.

✓ **Hirelie.** Justified each structure by the operation it serves rather than by habit, and the Set choice was load-bearing for the idempotency requirement. Volunteered the pure-function boundary unprompted.

MCQ Q2

✓ Correct

A claim larger than the vehicle's IDV must be rejected, and the policy must be left exactly as it was. Which approach most reliably guarantees that invariant in a method that touches several fields on the policy record?

- ☒ Validate the amount against the IDV first, and only mutate once every check has passed
- ☐ Append the claim immediately, then remove it again if the amount turns out to exceed the IDV
- ☐ Wrap the whole method in a try/catch block and swallow any error that is thrown
- ☐ Record the claim anyway and let the downstream billing job filter out the invalid ones

✓ **Hirelie.** Picked validate-then-mutate and noted that the compensating-undo option is where partial writes come from once a method grows a third field.

Finance says the NCB line on customer invoices is too generous. Work through what this premiumFor() charges for a car at IDV 6,00,000 in zone A, NCB 20, with ZERO_DEP attached — versus what it should charge.

↳ Codebase-derived question (dynamic) — this is his own submitted method.

CANDIDATE ANSWER

Base is 18000, loaded is 19800, then I add 1200 for ZERO_DEP to get 21000 and take 20% off all of it, so 16800. It should be 17040 — the bonus is a no-claim discount on my own-damage cover, it was never meant to discount a cover the customer bought separately. So the discount has to come off loaded, and the add-ons get added after. I am giving away 240 rupees on every policy with a cover on it.

```
premiumFor(policyId) {  
  const p = this.policies.get(policyId);  
  const base = Math.round(p.idv * BASE_RATE[p.vehicleType]);  
  const loaded = Math.round(base * (1 + ZONE_LOADING[p.zone]));  
  let total = loaded;  
  for (const code of p.addOns) total += ADD_ONS[code];  
  return total - Math.round(total * p.ncb / 100);  
}
```

⚠ **Hirelie.** Diagnosed his own bug correctly and unprompted, with the right domain reasoning for why the two bases differ. Marked down only because he shipped it: the sample suite never priced a discounted policy with a cover attached, and he did not think to construct that case himself.

This engine sits behind an HTTP claims API. A customer taps "submit claim" on a flaky connection and the app retries, so two identical requests arrive at almost the same moment. What goes wrong, and how do you make claim submission idempotent?

CANDIDATE ANSWER

Today it books the claim twice, and worse, both count toward the claim year, so the customer loses a bonus they should only have lost once. The fix is a client-generated claim reference sent with the request and stored unique on the claim row — the second insert loses and I return the first claim instead of creating a new one. Across instances that uniqueness has to live in the database, not in process memory, or two pods each think they are first.

✓ **Hirelie.** Went straight to the domain consequence rather than the generic "duplicate row" answer, and correctly located the uniqueness constraint in the store rather than the app.

MCQ Q5

✓ Correct

In highRiskReport you collect the matching policies and then sort them by claimed amount, then by policy id. For n matching policies with a standard comparison sort, what is the complexity of just that sorting step?

- ☐ $O(n)$, linear in the number of policies
- ☒ $O(n \log n)$, a typical comparison sort
- ☐ $O(n^2)$, quadratic
- ☐ $O(1)$, constant time

✓ **Hirelie.** Correct, and noted the filter ahead of it is the linear part so the sort dominates only once the threshold is loose.

FREE TEXT Q6

— Skipped

Pricing and claims rules change often. Tell me about a time you built something in a domain whose rules kept moving, and how you decided what to make configurable versus hard-coded.

No answer, skipped for time.

⚠ **Hirelie.** Skipped for time. Behavioral, low weight for this technical round.

Interview transcript

- 00:00
Assessment started
Opened the Motor Policy & Claims Engine problem.
- 00:41
Planning
Entered plan mode; mapped the policy record and the pricing order before any edit.
- 06:04
First edit
Began implementing issuePolicy / premiumFor from the reviewed plan.
- 18:44
Own tests
Wrote boundary tests (claim equal to IDV, the 50% slab cap) ahead of the hidden suite.
- 24:10
Sample suite green
All 5 sample tests passing after 4 fix-forward runs.
- 33:50
Rounding re-read
Questioned whether the NCB comes off before or after the add-ons; ran low on time.
- 38:12
Submitted code
11 of 12 tests passing; feature completion 92%.
- 39:05
AI interview
Hirelie ran a 6-question round on role + codebase.
- 52:45
Final processing
Code analysis, tests report, prompt transcript, and interview merged into this report.