GAME THEORY FOR ENGINEERS

How Strategic Thinking Shapes Code, Data, and Daily Decisions

Every day, you make dozens of strategic decisions — most of them without realizing you're playing a game.

NASH EQUILIBRIUM

Named after John Nash — subject of A Beautiful Mind

Your daily driving route is a Nash Equilibrium

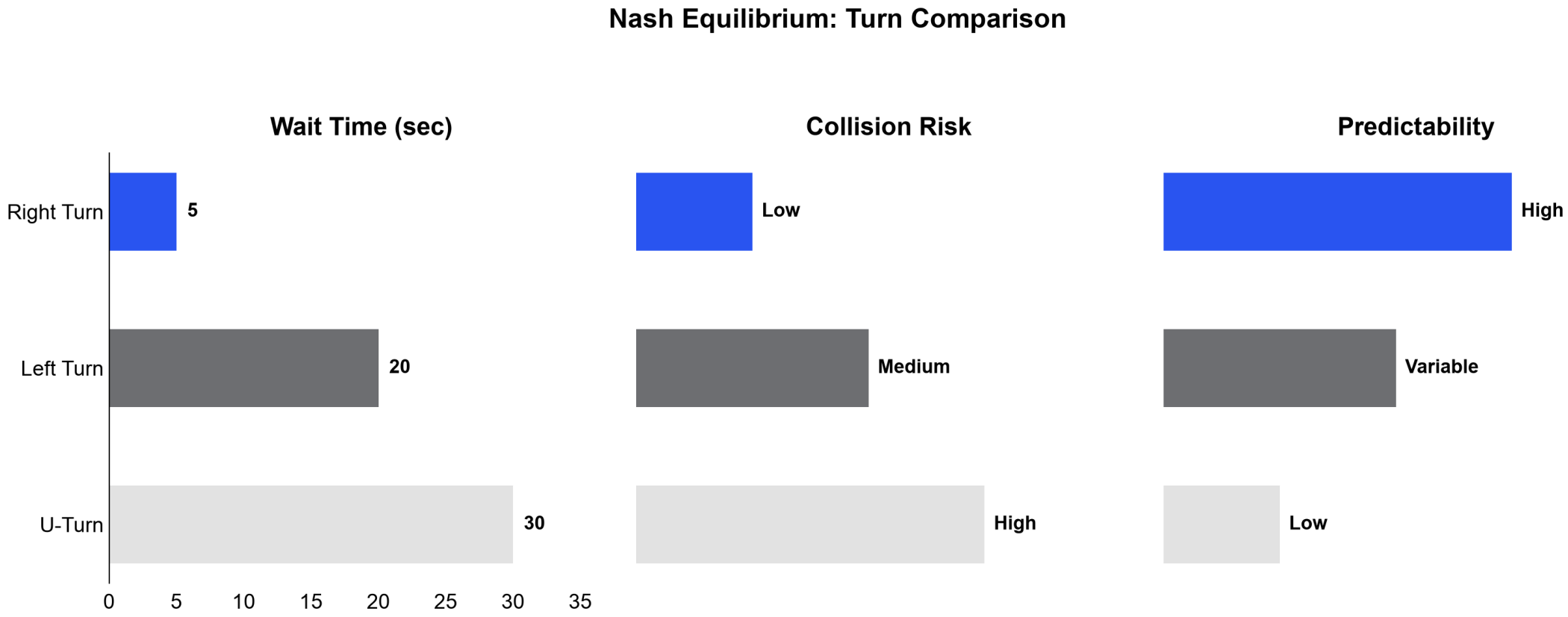
A stable state where no player can improve their outcome by changing only their own strategy. You've settled into a route you won't deviate from — because, given traffic patterns, nothing else consistently beats it.

Route Factor	Right Turn	Left Turn	U-Turn
Average wait time	~5 sec	~15–30 sec	~20–40 sec
Collision risk	Low	Medium	Higher
Predictability	High	Depends on traffic	Low

Also Applies To:

- Tool/framework adoption — teams settle on 'good enough' tools because switching only works if everyone switches
- Salary negotiations — market rates are a Nash Equilibrium across employers and candidates
- Cloud provider lock-in — each team's best response to what other teams chose

NASH EQUILIBRIUM — TURN TYPE COMPARISON



Right turns dominate across all three factors — wait time, collision risk, and predictability.

PRISONER'S DILEMMA

Two parties would both benefit from cooperating, but each has an individual incentive to defect — and when both defect, both end up worse off.

	B: Stay Silent	B: Confess
A: Stay Silent	(1, 1)	(10, 0)
A: Confess	(0, 10)	(5, 5)

(x, y) = Suspect A's sentence, Suspect B's sentence

At Work: Shared Datasets

Two analytics teams need enriched customer data. Each could build a clean, shared dataset (cooperate) — but building a quick team-specific query is faster (defect). Result: two divergent, poorly documented datasets with conflicting metrics and duplicate compute costs.

Also Applies To:

- Open source contributions — everyone benefits, but contributing takes individual effort
- Code review thoroughness — tempting to rubber-stamp, but quality drops when everyone does
- Documentation — every team benefits from good docs, no team wants to write them

DOMINANT STRATEGY

A strategy that produces the best outcome regardless of what anyone else does. If one exists, you should always play it.

Algorithm Performance

	100 rows	10K rows	1M rows
Nested Loop O(n²)	0.01s	10s	3+ hrs
Hash Map O(n)	0.01s	0.1s	1.2s

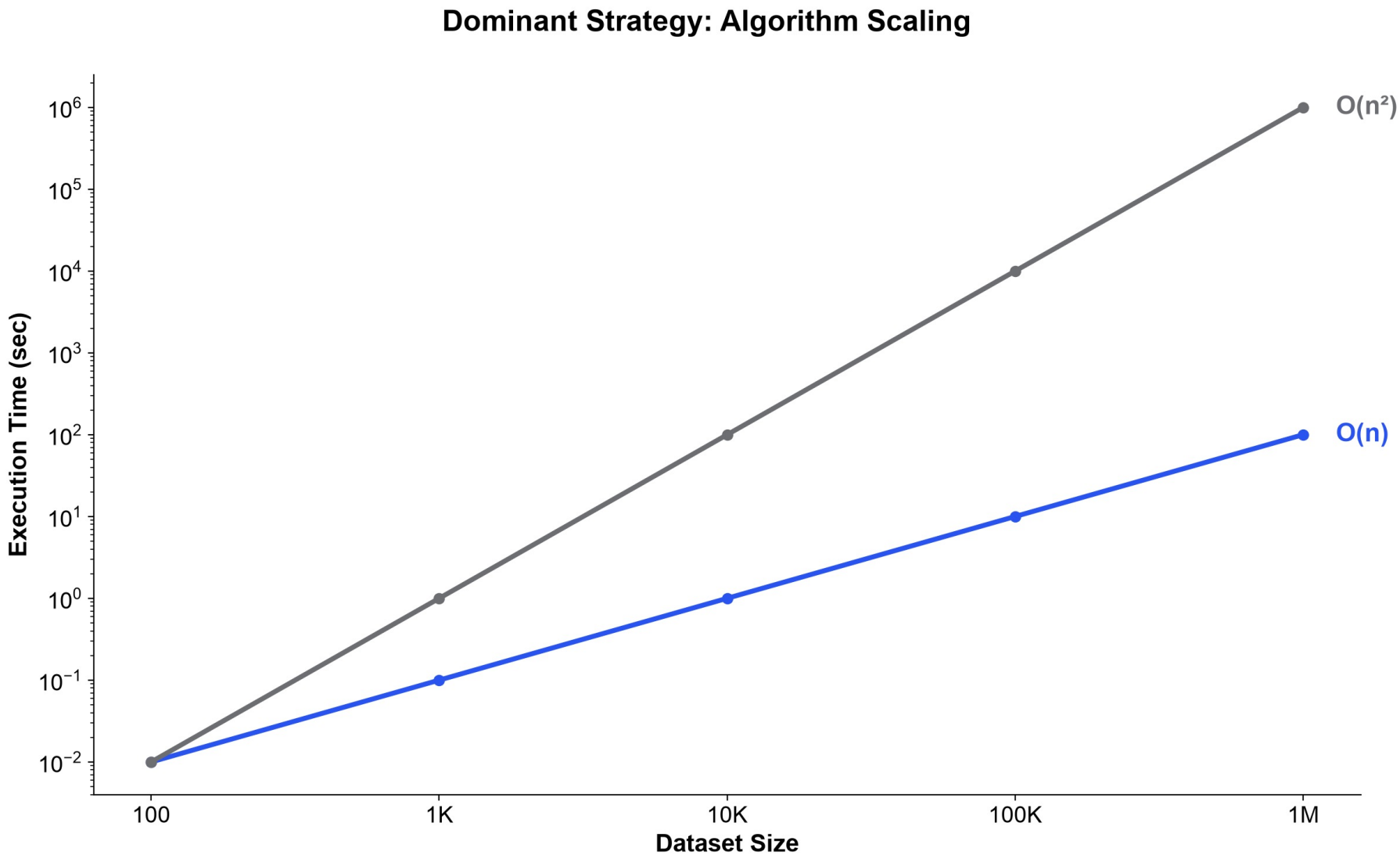
SQL Performance

	100 rows	10K rows	1M rows
IN (subquery)	Fast	Slower	Timeout
EXISTS	Fast	Fast	Fast

Also Applies To:

- Version control — always better than no version control, regardless of team size
- Input validation — protects you whether users are careful or careless
- Automated testing — catches regressions no matter how stable the codebase seems

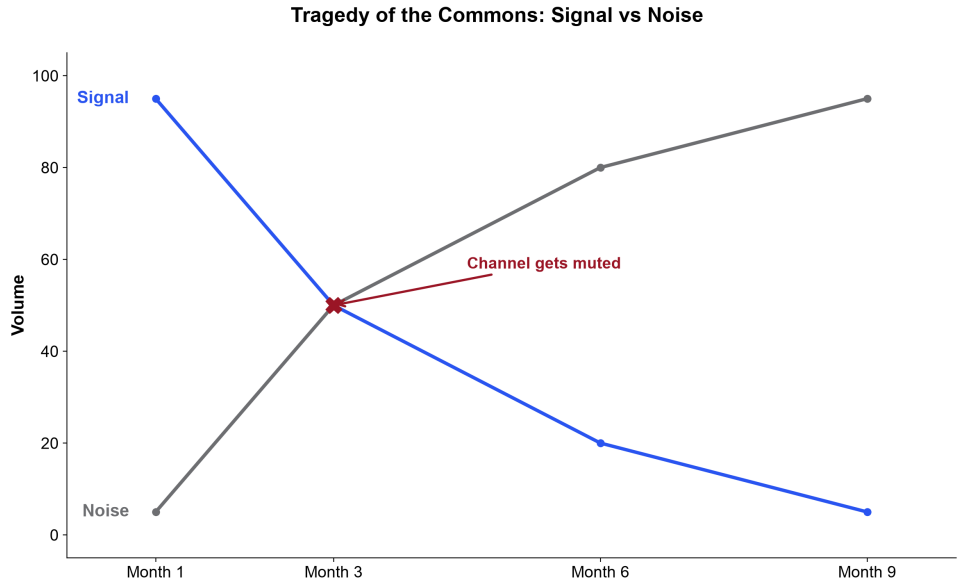
DOMINANT STRATEGY — SCALING



TRAGEDY OF THE COMMONS

When a shared resource has no ownership or usage limits, individuals acting in their own interest deplete it — making everyone worse off.

A team creates #general-engineering for important cross-team updates. It works great — for a while. Then memes, misrouted questions, and lunch threads pile up. No single message ruins the channel, but the cumulative effect does. Signal drowns in noise. The people who needed it most mute it.



Also Applies To:

- Shared database resources — everyone runs expensive queries, nobody owns capacity planning
- Technical debt — each shortcut is small, but accumulated debt slows the entire team
- Shared data environments — 'just add one more column' until the table has 200 columns
- Meeting culture — 'just one more meeting' until no one has time for deep work

MECHANISM DESIGN

"Reverse Game Theory" — Design rules so self-interest produces good outcomes

Don't rely on good intentions. Design the system so the easy path is the right path. Required PR approvals mean code gets reviewed without anyone needing to volunteer. CI pipelines that block on failure mean broken code never reaches production.

Without Mechanism Design	With Mechanism Design
"Please review my PR" (optional)	PR requires 1+ approvals to merge
"I ran the tests locally" (trust-based)	CI blocks merge on failure
"Don't push to main" (honor system)	Branch protection enforced
"Write tests" (guideline)	Coverage thresholds in pipeline

Also Applies To:

- Data contracts — define schemas so upstream changes can't silently break downstream
- Incentive structures in hiring — interview loops that reduce bias by structure, not willpower
- Infrastructure as Code — deployed state matches reviewed state by design
- Linting and formatting rules — removing style debates from code review entirely

MORE CONCEPTS WORTH KNOWING

Concept	What It Is	Quick Example
Iterated Games / Tit-for-Tat	When you play the same game repeatedly, cooperation becomes viable. Tit-for-tat — cooperate first, then mirror.	Code review reciprocity: thorough reviews earn thorough reviews back.
Schelling Points	When people coordinate without communicating, they converge on "obvious" choices.	Naming conventions — two engineers independently pick similar names if conventions exist.
Minimax	Choose the strategy that minimizes your worst-case outcome, rather than maximizing your best case.	Error handling and circuit breakers — you're capping how bad things can get.
Pareto Optimality	A state where you can't make one thing better without making something else worse.	Performance vs. readability — at some point, optimizing one degrades the other.
Zero-Sum vs Non-Zero-Sum	Zero-sum: one player's gain is another's loss. Non-zero-sum: both can win or lose.	Most engineering work is non-zero-sum. Competition for promotions can feel zero-sum.

YOU'RE ALREADY PLAYING

You're already playing these games every day — in how you write code, design systems, and navigate your commute. The difference is whether you're playing them intentionally.