Does Wikipedia Make Code Better?
A Load Balancer Case Study

Timothy Brinded

February 2026

Abstract

We test whether augmenting an LLM coding agent with access to Wikipedia improves the quality of generated code. A single design-rich programming problem—a graceful-degradation load balancer—is implemented by Claude Code under eight conditions: one control and seven Wikipedia-augmented variants with different research strategies. Solutions are evaluated by three complementary methods: a blinded LLM judge (design quality), a deterministic behavioral benchmark (7 fault scenarios, weighted 0–100), and quantitative cost metrics. The adversarial *contrarian* condition, which stress-tests the obvious approach before coding, achieves the highest benchmark score (92.9 vs. 84.7 control). Architectural ambition can hurt: the *consilience* condition builds the most sophisticated design but scores last on benchmarks (76.2). Judge and benchmark rankings diverge meaningfully, suggesting each captures different dimensions of code quality.

1 Introduction

Large language models generate competent code from training data alone. But can cross-domain knowledge—the kind found in encyclopedic references—improve the *design* of generated software? We test a simple hypothesis: giving a coding agent access to Wikipedia articles alongside a programming problem produces better-designed code than coding without external references.

We study a single problem: implementing a load balancer with graceful degradation. Load balancers are design-rich systems where multiple valid architectures exist, failure modes are subtle, and cross-domain analogies (TCP congestion control, biological homeostasis, power grid protection) can yield structural insights.

Eight conditions are evaluated. The **control** receives only the problem specification. Seven Wikipedia-augmented conditions each use a different research strategy: direct research (*explicit*), ambient exposure (*subtle*), historical precedent (*reflective*), undirected exploration (*flaneur*), cross-domain convergence (*consilience*), biological analogy (*biomimetic*), and adversarial critique (*contrarian*). Table 1 summarizes all eight conditions.

Table 1: Experimental conditions and research strategies.

Condition	Research Strategy
control	No Wikipedia access
explicit	Instructed to research Wikipedia
subtle	Wikipedia tools available, not mentioned
reflective	Seek historical precedent and past failures
flaneur	Undirected random Wikipedia exploration
consilience	Find convergent patterns across 5+ domains
biomimetic	Research only biological analogies
contrarian	Stress-test the obvious approach first

2 Method

2.1 Experimental Setup

Each condition runs Claude Code in non-interactive mode (`claude -p`) with an identical problem specification. The agent implements

an `AbstractLoadBalancer` base class with four required methods: `on_backend_added`, `route_request`, `on_request_complete`, and `on_tick`. Backend health is inferred through an opaque `BackendHandle` protocol—the agent cannot query health directly, only observe request outcomes and probe results.

2.2 Conditions

The **control** receives the problem specification and nothing else. All seven Wikipedia conditions receive the same specification plus a Wikipedia research agent (`wiki-explorer`) with access to ∼7 million local Wikipedia articles. The conditions differ only in their system prompt framing:

- **Explicit**: "Research Wikipedia for load balancer design patterns."
- **Subtle**: Tools available silently; no mention in prompt.
- **Reflective**: "Find historical failures that inform resilient design."
- **Flaneur**: "Take a random walk through Wikipedia; let connections emerge."
- **Consilience**: "Find the same pattern in 5+ unrelated domains."
- **Biomimetic**: "Research only biological systems as design analogies."
- **Contrarian**: "Before coding, find reasons the obvious approach fails."

2.3 Evaluation

Solutions are evaluated by three independent methods.

Deterministic Benchmark. Seven fault-injection scenarios run against each implementation, producing scores on $[0, 1]$ per scenario. An aggregate score is computed as the weighted average scaled to $[0, 100]$.

Blinded LLM Judge. Each Wikipedia condition is paired against control in a blinded A/B comparison (randomized assignment to "Solution A" and "Solution B"). The judge scores six dimensions: correctness ($\times 3$), design ($\times 2$), robustness ($\times 2$), algorithmic novelty ($\times 2$), cross-domain insight ($\times 1$), and proportionality ($\times 1$). Maximum weighted total: 110 points.

Quantitative Metrics. Duration, API cost, lines of code, and subagent token usage are recorded for each run.

2.4 Benchmark Design

Table 2 lists the seven benchmark scenarios and their weights. Priority Protection carries the highest weight (3.0) because correctly shedding low-priority traffic during overload is the core requirement. Degradation Detection (2.0) tests whether the agent can sense a backend that is slowly failing— a subtle scenario that separates sophisticated health tracking from naive approaches.

Table 2: Benchmark scenarios and weights.

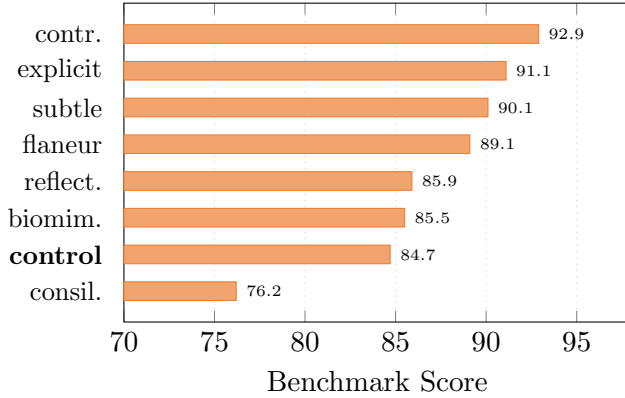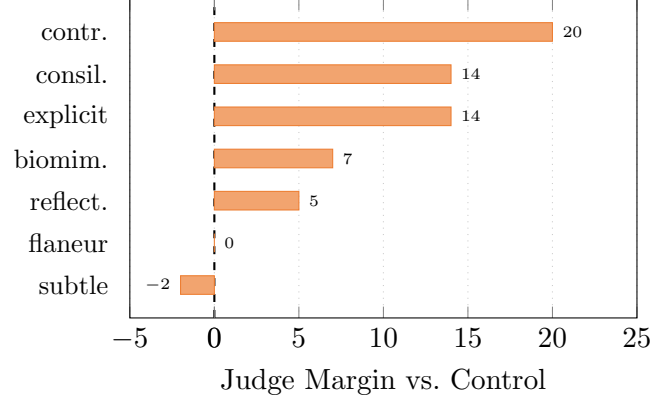| Scenario	Tests	Weight
Steady State	Even distribution, full success	1.0
Degradation	Detect slowly failing backend	2.0
Priority	Shed low-priority traffic first	3.0
Recovery	Smooth ramp-up after failure	2.0
Cascading	Isolate failure, protect healthy	2.0
Flapping	Handle on/off oscillation	1.5
Asymmetric	Route by backend speed	1.0

3 Results

3.1 Benchmark Rankings

Figure 1a shows aggregate benchmark scores ranked by performance. The *contrarian* condition achieves the highest score (92.9), outperforming control (84.7) by +8.2 points. Six of seven Wikipedia conditions outperform control. The lone exception is *consilience* (76.2), which ranks last despite building the most architecturally ambitious solution.

3.2 Scenario Heatmap

Table 3 shows per-scenario scores for all conditions, sorted by aggregate rank. Contrarian dominates degradation detection (0.99), recovery (0.97), cascading failure (0.94), and flapping resistance (0.95). However, its proportional shedding approach yields the lowest priority protection score (0.84) among all conditions. Biomimetic excels at asymmetric

(a) Aggregate benchmark scores (0–100). Control baseline in bold.

(b) Blinded LLM judge margin vs. control (out of 110). Positive = Wikipedia wins.

Figure 1: Benchmark scores (left) and judge margins (right). Six of seven Wikipedia conditions beat control on both measures, though the rankings diverge substantially (see Section 3.4).

routing (0.98) via pheromone reinforcement but struggles with steady-state evenness (0.68).

3.3 Judge Rankings

Figure 1b shows the blinded LLM judge margin for each condition versus control. Contrarian achieves the largest margin (+20 points), while consilience and explicit tie at +14. The judge found a real bug in control's implementation during the contrarian comparison—a stale data deque that permanently locks out recovering backends. Only *subtle* loses to control (−2), having barely used Wikipedia tools.

3.4 Judge vs. Benchmark Divergence

The judge and benchmark agree on the overall direction—Wikipedia conditions generally outperform control—but diverge sharply on three conditions:

- **Consilience**: Judge rank 1st (score 76/110); Benchmark rank 8th (76.2/100). The judge praised its two-level shedding architecture and sigmoid health functions. The benchmark exposed a fatal blind spot: degradation detection scored 0.18—it cannot detect a slowly failing backend.
- **Subtle**: Judge rank 7th (score 56/110); Benchmark rank 3rd (90.1/100). The judge saw two nearly identical implementations and marginally preferred control. But subtle's code *performs* measurably better under fault scenarios.
- **Biomimetic**: Judge rank 4th (score 69/110); Benchmark rank 6th (85.5/100). The judge re-

warded novel ACO pheromone routing. The benchmark penalized uneven steady-state distribution (0.68).

3.5 Cost and Efficiency

Figure 2a plots benchmark score against API cost. Control is the most cost-efficient ($0.85 for 84.7 points). Among Wikipedia conditions, subtle achieves the best return ($0.92 for 90.1 points), nearly matching control's efficiency. Biomimetic is the least efficient ($2.98 for 85.5 points). Table 4 shows the full breakdown.

Table 4: Cost, efficiency, and code metrics. Score/$ = benchmark score per dollar of API cost.

Condition	Time	Cost	LOC	Score/$
control	4:03	$0.85	471	99.7
subtle	4:04	$0.92	378	97.9
reflective	5:36	$1.42	466	60.5
flaneur	7:49	$2.10	548	42.4
contrarian	9:21	$2.23	539	41.7
consilience	9:17	$2.81	531	27.1
explicit	5:05	$2.92	534	31.2
biomimetic	12:07	$2.98	444	28.7

3.6 Wikipedia Usage

Table 5 shows research effort versus actual knowledge extraction. Explicit spent the most subagent tokens (2,894) but referenced zero Wikipedia articles in its final code. Contrarian read the fewest
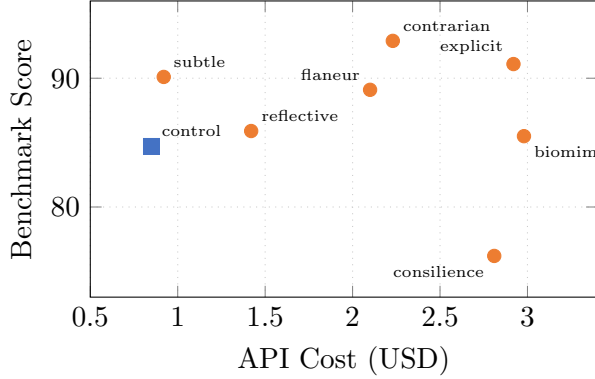
Table 3: Per-scenario benchmark scores (0.0–1.0), sorted by aggregate rank. Color key: ≥ 0.90   0.80–0.89   0.70–0.79   0.50–0.69   < 0.50 .

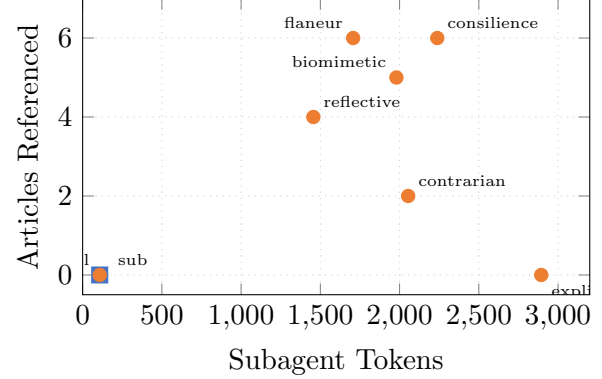|	Steady (w = 1.0)	Degrade (w = 2.0)	Priority (w = 3.0)	Recovery (w = 2.0)	Cascade (w = 2.0)	Flapping (w = 1.5)	Asymm. (w = 1.0)	Agg.
contrarian	0.99	0.99	0.84	0.97	0.94	0.95	0.87	92.9
explicit	0.95	0.97	0.99	0.81	0.93	0.83	0.81	91.1
subtle	1.00	0.90	0.99	0.83	0.87	0.86	0.81	90.1
flaneur	1.00	0.79	0.99	0.84	0.90	0.83	0.89	89.1
reflective	1.00	0.77	0.98	0.83	0.78	0.86	0.77	85.9
biomimetic	0.68	0.91	1.00	0.74	0.82	0.73	0.98	85.5
control	1.00	0.58	0.98	0.90	0.78	0.90	0.77	84.7
consilience	1.00	0.18	0.98	0.83	0.71	0.92	0.77	76.2

articles among active conditions (2) but extracted the most actionable insight—power-of-two-choices routing became its core algorithm and the Mars Pathfinder case motivated its asymmetric EWMA.

Table 5: Wikipedia usage per condition. "Tokens" = subagent output tokens; "Articles" = Wikipedia articles visibly referenced in the final implementation.

Cond.	Tokens	Art.	Key Sources
control	108	0	—
subtle	108	0	Never found tools
explicit	2,894	0	Research didn't transfer
reflective	1,456	4	Gmail outage, NE blackout
flaneur	1,707	6	Monocoque, Deperdussin
consilience	2,238	6	UFLS, ISR, triage
biomimetic	1,980	5	Baroreceptors, ACO
contrarian	2,054	2	Mars Pathfinder, Po2C

(a) Benchmark score vs. API cost. Control (blue square) is the most cost-efficient; contrarian delivers the best score at 2.6× cost.

(b) Research effort vs. knowledge extraction. Explicit researched the most but extracted nothing; contrarian extracted high-impact insights.

Figure 2: Cost–performance trade-off (left) and Wikipedia research patterns (right).

4 What Each Agent Built

Table 6 summarizes the architectural choices made by each condition. Despite identical problem specifications, the eight agents produced strikingly different designs.

Control (471 LOC) implements weighted least-connections routing with EWMA latency and sliding-window success rate. Priority-based shedding activates at fixed capacity thresholds. A solid, no-frills baseline.

Explicit (534 LOC) builds a three-state circuit breaker (CLOSED/OPEN/HALF_OPEN) with AIMD capacity management borrowed from TCP congestion control. Hysteresis on shedding thresholds (enter at 70%, exit at 80%) prevents oscillation—a structural advantage the judge highlighted.

Subtle (378 LOC) is the leanest implementation. Architecturally near-identical to control, it never discovered the Wikipedia tools. Yet it benchmarks 3rd, suggesting the contract-based specification alone improved code quality.

Reflective (466 LOC) overlays a circuit breaker on weighted least-connections, citing the 2003 Northeast blackout and Gmail 2012 outage. Conservative and well-proportioned—the "measure twice" approach.

Flaneur (548 LOC) uses weighted round-robin with three independent health axes (latency, error rate, probe latency) combined via geometric mean.

Inspired by monocoque structures from a random Wikipedia walk, its four-state machine includes explicit hysteresis.

Consilience (531 LOC) is the most ambitious: two-level shedding with per-backend stage gating and system-level UFLS (from power grid engineering), sigmoid health scoring, and six cross-domain references. The most sophisticated architecture—and the worst benchmark score.

Biomimetic (444 LOC) implements pheromone-weighted ACO routing with 10% random exploration. Baroreceptor sensing (latency derivative) provides predictive detection. The most biologically faithful implementation excels at asymmetric backends (0.98) but introduces variance that hurts steady-state (0.68).

Contrarian (539 LOC) uses power-of-two-random-choices routing (pick two candidates, route to the better one). Asymmetric EWMA—fast degrade (α=0.4), slow recover (α=0.08)—prevents detection lag. Proportional admission replaces cliff-based shedding. The adversarial research strategy produced the highest benchmark score.

5 Discussion

Six conclusions emerge from this experiment.

1. Adversarial research produces the most robust code. Contrarian actively searched for what could go wrong before coding. This led to the discovery of the recovery deadlock bug (stale data

Table 6: Architectural comparison across conditions. WLC = weighted least-connections; WRR = weighted round-robin; CB = circuit breaker; EWMA = exponential weighted moving average; AIMD = additive increase / multiplicative decrease; ACO = ant colony optimization; Po2RC = power-of-two random choices.

Condition	Routing	Health Tracking	Shedding	Recovery	LOC
control	WLC	EWMA + sliding window	Fixed thresholds	Quadratic ramp	471
explicit	CB states	EWMA + circuit breaker	Hysteresis (70/80%)	AIMD	534
subtle	WLC	Dual-EMA	Fixed thresholds	Fast weight ramp	378
reflective	WLC	Composite score + CB	AIMD thresholds	AIMD	466
flaneur	WRR	Geometric mean (3 axes)	Hysteresis + 4-state	State machine	548
consilience	Two-level	Sigmoid from latency	Staged + UFLS	Asymmetric 6:1	531
biomimetic	ACO pheromone	Baroreceptor sensing	ISR-like progressive	Pheromone rebuild	444
contrarian	Po2RC	Asymmetric EWMA	Proportional admission	Adaptive ramp	539

deque with zero-weight backends), which it avoided and control did not. The result was the highest benchmark score (92.9) and the largest judge margin (+20).

2. Architectural ambition can hurt. Consilience built the most sophisticated architecture (two-level shedding, sigmoid health, six cross-domain references) but scored last on benchmarks (76.2). Its degradation detection scored 0.18—a basic scenario that six other conditions handled adequately. Sophistication is not a proxy for correctness.

3. The judge and benchmark measure different things. The judge evaluates architecture, novelty, and cross-domain insight. The benchmark evaluates whether the code actually works under fault injection. Both are needed: the judge alone would rank consilience 1st; the benchmark alone would miss biomimetic's creative value. For production evaluation, behavioral benchmarks should take precedence.

4. Subtle is surprisingly competitive. Despite barely using Wikipedia (108 subagent tokens, zero articles), subtle benchmarks 3rd (90.1). This may indicate that the contract-based problem specification—with its explicit interface, enumerated priorities, and behavioral requirements—does more work than the Wikipedia access.

5. Cost scales with research depth, not quality. Biomimetic cost 3.5× control for +0.8 benchmark points. Contrarian cost 2.6× for +8.2 points. The correlation between cost and quality is weak—the cheapest Wikipedia condition (subtle, $0.92) outperforms the most expensive (biomimetic, $2.98) by 4.6 points.

6. Creative approaches trade consistency for adaptiveness. Biomimetic's ACO routing excels at asymmetric backends (0.98) but introduces variance that hurts steady-state (0.68) and flapping resistance (0.73). Standard approaches are more predictable; novel algorithms require broader scenario coverage to validate.

Limitations. This is a single-problem ($N=1$), single-model (Claude), single-judge experiment. The results may not generalize to other problem types, models, or evaluation approaches. Each condition was run once, so we cannot estimate within-condition variance. The LLM judge's scores vary across comparisons (control scored between 52 and 63 across seven pairings), introducing noise in the margin estimates.

6 Conclusion

Wikipedia access can improve LLM-generated code, but the research strategy matters more than the access itself. Adversarial research (contrarian) produced the most robust implementation by finding failure modes before they became bugs. Undirected or overly ambitious research strategies yielded diminishing or negative returns. Evaluating generated code requires multiple complementary methods—design-quality judges and behavioral benchmarks capture different dimensions of quality and can disagree sharply on the same implementation.