The Cost of Correctness

Why Rigorous LLM-Assisted Engineering Exhausts Quotas While Delivering Marginal Insight

Youssef Touil — Airspy
In collaboration with Claude (Anthropic)
April 2026

ABSTRACT

The dominant narrative around LLM-assisted engineering conflates two structurally distinct roles: directing discovery — determining which directions in a high-dimensional space of candidates, constraint interactions, and failure modes are worth pursuing — and supporting it — accelerating elimination, checking constraints, confirming dead ends within directions the human has already chosen. Current LLMs participate in discovery but only in the supporting role. Discovery direction remained human-owned: the model did not generate the decisive problem framings, originate major architectural moves, or identify key failure modes independently. This distinction is consequential and underappreciated. It is further obscured when the final algorithm is simple, because simplicity is mistaken for the absence of a hard problem. In practice, a low-complexity final algorithm in a high-constraint system is the output of hard discovery work, not evidence that the search was easy. Elegance is what successful discovery produces. It is not a property of the problem. This paper documents this structural mismatch through the development of NoiseFilterLP for SDR# by Airspy: a real-time DSP noise reduction plugin whose final algorithms are deliberately low-complexity but whose path to them was not. All architectural breakthroughs, failure mode identifications, and structural simplifications originated with the human engineer. The model participated in discovery in a supporting role — ruling out candidate directions faster, confirming dead ends the human had already identified — but it did not direct the search at any point. The design space was the human's. Token quota exhaustion — a secondary consequence of the collaboration structure, not its primary deficiency — is also documented, alongside an explicitly enforced epistemic constraint stack required to make even the supporting contributions reliable. The result is a characterization, grounded in one well-documented collaboration, of where LLM assistance was and was not useful in this class of work — and why the hardest part of the problem was the part it could not reach.

1. Introduction

This is a practitioner's account, not a controlled study. It documents a real collaboration between the author — founder of Airspy and principal developer of SDR# — and a large language model, over

several months of active development on a technically demanding engineering problem. It is shared out of good faith toward engineers who may recognize their own situation in it, not as a claim about the general behavior of LLMs across all domains or all problem classes. The findings are offered as one precisely characterized data point. Their value is in the specificity of the characterization, not the breadth of the sample.

Engineering work divides into two activities that look similar from the outside but are structurally distinct. The first is discovery: navigating a space of candidate approaches, identifying constraint interactions, recognizing architectural dead ends, surfacing failure modes that are only visible when the full system is held in mind simultaneously. The second is implementation: correctly executing a solution that has already been found. Discovery produces the design. Implementation produces the code.

Current large language models participate in both activities, but asymmetrically. At the implementation layer they are effective and largely independent. At the discovery layer they require the human to direct the search: to define the problem space, identify the relevant constraint interactions, recognize when a direction has failed, and determine when a reframing is required. The model can accelerate execution within a direction the human has chosen. In this collaboration, direction-setting remained with the human. This asymmetry was not resolved by prompting improvements. It persisted across the full duration of the work.

This asymmetry is further obscured by a common assumption: that a simple final algorithm implies a simple problem. It does not. In a high-constraint engineering domain, a low-complexity final algorithm is the output of hard discovery work — the elimination of many approaches that failed, the identification of the structural insight that made simplicity possible, the recognition of failure modes that ruled out plausible alternatives. The elegance of the result conceals the difficulty of the search. The model's effective contribution to the result does not scale with the difficulty of the search that produced it.

The NoiseFilterLP development, documented here, is a concrete instance of this. The system's final algorithms are low-complexity by design. The path to them was not. All architectural breakthroughs, failure mode identifications, and structural simplifications were initiated by the human engineer. The model participated in the discovery process in a supporting role — ruling out candidate directions faster, confirming dead ends already suspected — but did not direct the search at any point. Token quota exhaustion, documented as a secondary finding, compounded the problem structurally: quota was consumed on implementation turns while the hard discovery work was done without model assistance, precisely because the quota had run out.

2. Domain Context

2.1 Airspy and SDR#

Airspy is a hardware and software company operating in the software-defined radio (SDR) space. Its principal software product, SDR# (SDRSharp), is a widely used signal processing application supporting a broad range of SDR hardware. SDR# includes a plugin architecture permitting first-party and third-party DSP modules to be integrated into the signal processing chain.

2.2 NoiseFilterLP

NoiseFilterLP (marketed as 'Natural Intelligence Noise Reduction' within SDR#) is a C++ adaptive per-band noise reduction plugin. It operates in real time on a dedicated DSP thread under strict latency constraints. The internal architecture employs several novel algorithmic approaches developed specifically for this problem domain; these are not publicly disclosed. The system integrates with SDR#'s spectral processing pipeline and imposes hard requirements on memory allocation patterns, computational complexity bounds, and thread safety.

2.3 Constraint Density

The NoiseFilterLP constraint space is high-dimensional. Any proposed algorithm or implementation must simultaneously satisfy a set of hard real-time, memory, threading, and algorithmic complexity requirements specific to this architecture. These constraints interact: a solution satisfying the majority may fail on the remainder in non-obvious ways requiring full architectural reasoning to detect. Their density and interdependency are the primary source of the evaluative cost documented in this paper.

3. Discovery vs. Implementation: The Primary Axis

The distinction between discovery and implementation is the organizing axis of this paper. It is worth stating precisely before the case is presented.

3.1 What Discovery Requires

Discovery in a high-constraint engineering domain is not a search over a flat space. It requires holding the full constraint graph simultaneously — not sequentially — while evaluating candidates against interactions that only become visible at the system level. A candidate that satisfies every constraint in isolation may fail when constraints interact. Identifying that failure requires architectural reasoning that operates across the entire system, not within any single component.

Discovery also requires generating novel framings of the problem. When the current formulation of a problem resists solution, discovery involves recognizing that the formulation itself is wrong — that the right solution requires restructuring the problem space, not searching harder within it. This is the source of architectural breakthroughs: a reframing that makes the solution space tractable when the prior framing made it intractable.

In the NoiseFilterLP development, the major breakthroughs were of precisely this type. The final algorithms are simple not because the problem was simple but because the right framing was found. Finding it required eliminating framings that were locally plausible but globally incorrect — a process in which the model did not participate at the directional level: it did not generate the framings

or recognize their failures without being guided to them.

3.2 What Implementation Requires

Implementation, given a fully specified design, is a different kind of work. It requires correctness under a known constraint set, not search under an unknown one. The solution is defined; the task is to realize it without introducing errors, without violating invariants, without creating failure modes at the boundaries of specified behavior. This is where the model contributes effectively — when the design is fully specified and the constraints are fully enumerated.

The epistemic constraint stack documented in Section 4 exists precisely to make this contribution reliable. Without it, the model's implementation outputs are locally plausible but globally incorrect — they satisfy an unstated contract rather than the actual one. The constraint stack enforces the conditions under which implementation assistance is actually useful.

3.3 The Mislabeling Problem

The practical consequence of conflating discovery and implementation is that engineers invest prompting effort and quota in attempting to use the model for discovery work, receive implementation-layer responses that are locally coherent but architecturally uninformed, and conclude either that the model is broken or that they are prompting incorrectly. Neither conclusion is right. In this workflow, the model was functioning correctly at the narrower layer where it contributed. The error was in expecting more from it in this problem class.

A low-complexity final algorithm is not a signal that the model could have found it. It is a signal that the human's discovery work was successful. In this workflow, the model's highest-value role began after the relevant discovery step had already occurred. Recognizing this boundary before engaging — not after quota is exhausted — is the primary practical implication of this paper.

4. The Nature of High-Dimensional Engineering Problems

The term high-dimensional is used loosely in engineering contexts. In the context of this paper it has a precise meaning: a problem is high-dimensional not when it has many constraints, but when the interactions between constraints produce a search space whose complexity exceeds what can be navigated by sequential, local reasoning. Understanding why this created a practical barrier in the documented collaboration requires unpacking what high-dimensionality means in the context of this problem class.

4.1 Constraint Interaction Complexity

A system with N hard constraints does not present N independent verification tasks. It presents a constraint interaction graph. Every pair of constraints may interact; every triple may interact in ways not visible from any pair; and so on. The number of potential interactions scales combinatorially with N, while the number of interactions that actually matter in a given system is determined by the architecture — which is precisely what is being designed and therefore not yet known.

This means that a candidate solution satisfying every constraint in isolation may fail when constraints are applied simultaneously. The failure is not a violation of any individual constraint. It is a property of the interaction, invisible until the full constraint set is applied at once. Identifying such failures requires holding the complete constraint graph in working context while reasoning about candidate behavior — not sequentially, not locally, but simultaneously across the whole system.

In this collaboration, this was the observed boundary. The model reasoned effectively within a local context. It did not surface constraint interaction failures independently: in every documented case, the human identified the relevant constraints and their interaction and presented that interaction as the specific question. At that point the model verified a hypothesis the human had already formed.

4.2 Non-Tractable Methodologies

The search space of any real engineering problem includes methodologies that are theoretically correct but non-tractable in the target context. Tractability is not a binary property — it is a relationship between a methodology and a constraint set. A methodology that is tractable in one context may be non-tractable in another due to constraints that have nothing to do with the methodology's correctness.

Non-tractability takes several forms. Computational intractability: the methodology is correct but its complexity exceeds the available budget under real-time or resource constraints. Architectural intractability: the methodology requires structural properties — dynamic allocation, global state, non-causal access — that are prohibited by the target environment. Statistical intractability: the methodology is correct in expectation but its worst-case behavior violates hard guarantees that the system must provide. Compositional intractability: the methodology is tractable in isolation but becomes intractable when composed with other required components.

Identifying non-tractability in a candidate methodology requires applying the full constraint set to the methodology simultaneously. A model evaluating the methodology without the full constraint set will not find the intractability. A model evaluating one constraint at a time will find individual violations but miss compositional failures. Only when the human has already identified which constraint interaction produces the intractability can the model confirm and analyze it reliably.

This is a significant source of wasted prompting effort. Engineers present a methodology to the model for evaluation; the model evaluates it against a subset of the constraints visible in the prompt; the methodology passes; the engineer implements it; the intractability appears at integration time. The model did not fail to reason correctly — it reasoned correctly within the context it was given. The context was insufficient because the full constraint interaction was not present in the prompt. Making it present requires the human to have already identified the interaction — at which point the discovery is done.

4.3 Generalizations and Specializations

A large class of engineering discovery involves moving between levels of generality: recognizing that a known general method can be specialized to the current problem, or that a known specialized

solution can be generalized to cover additional constraints. Both directions are sources of architectural insight. In this collaboration, both required discovery work that the model did not perform independently.

Valid specialization requires knowing which structural properties of the current problem make the specialization correct — which assumptions of the general method are satisfied by the specific context, and which are not. An incorrect specialization applies a method in a context where its assumptions do not hold, producing a solution that appears correct locally but fails under conditions the general method was designed to handle. Identifying the valid specialization requires deep understanding of both the method's assumptions and the problem's structure. In the documented workflow, this comparison was not made by the model independently: both the method assumptions and the problem structure had to be fully specified by the human before the model could evaluate the match.

Valid generalization is the inverse problem. A specialized solution exists that works correctly under a restricted constraint set. The actual constraint set is broader. Generalizing the solution requires identifying which properties of the specialization are load-bearing — which must be preserved in the generalization — and which can be relaxed. An incorrect generalization relaxes a load-bearing property, producing a solution that fails under constraints the specialization never encountered. Again: the model can verify a proposed generalization once the human had identified the load-bearing properties. It did not identify them independently in the documented workflow.

The practical consequence is that the model is useful at the last step of generalization and specialization work — verifying that a candidate generalization or specialization satisfies the constraint set — but not at the preceding steps of identifying the candidate, recognizing the load-bearing properties, or determining the direction of movement in the generality space. Those steps are discovery. The verification step is implementation.

4.4 Parameterization as a Discovery Strategy

A specific and practically important instance of the generalization/specialization pattern arises when a direct algorithmic problem is non-tractable under the active constraint set, but the algorithm belongs to a parameterized family whose members span a range of constraint trade-offs. In this case, the non-tractable direct search — find the algorithm — can sometimes be converted into a tractable indirect search: find the parameters. The parameterized family is the generalization. The parameter-reduced result is the specialization. The discovery work is recognizing that this conversion is valid for the current problem.

The conversion works because the parameter space is typically lower-dimensional than the algorithm space. Searching for an algorithm directly requires navigating a space whose structure is defined by all possible algorithmic choices simultaneously. Searching for parameters within a fixed family requires navigating a space whose structure is defined by the parameterization, which is known and finite. The search problem becomes tractable not because the constraints changed but because the representation of the solution changed. The final specialized algorithm — derived from the found

parameters — is often simpler than anything that would have been found by searching the algorithm space directly, because the parameterization encodes structural knowledge about the problem that prunes the space before the search begins.

This pattern appears in the NoiseFilterLP development. Non-tractable direct approaches were converted into parameter-finding problems whose solutions reduced to simpler specialized algorithms than the direct search would have produced. The tractability gain was real and significant. But the discovery work — recognizing that the conversion was valid, identifying the correct parameterization, and knowing which parameters to search for — was done entirely by the human engineer. The model contributed to the parameter search once the parameterization was specified, but it could not have identified the parameterization independently. That identification required holding the full constraint set simultaneously and recognizing that the problem structure admitted a lower-dimensional representation — which is precisely the kind of discovery step that, in this collaboration, remained with the human.

For engineers in adjacent domains: when a direct algorithmic search is failing under a dense constraint set, the diagnostic question is whether the target algorithm belongs to a parameterized family that is tractable under the same constraints. If it does, the problem can potentially be reframed as parameter finding. The reframing itself is discovery work. In the documented workflow it was not delegated successfully to the model — it remained with the human. Once the reframing was done, the model assisted effectively with the parameter search and with deriving the final specialized algorithm from the found parameters.

4.5 The Structural Property That Defines This Problem Class

The problems described in this paper share a structural property that is independent of domain. It is not the complexity of the final algorithm. It is not the size of the codebase. It is not even the number of constraints. The defining property is the ratio of search space size to solution space size, combined with the degree to which the solution space is determined by constraint interactions rather than individual constraints.

When this ratio is high and the interaction degree is high, the following conditions hold simultaneously: the solution is narrow relative to the space of plausible candidates; plausible candidates fail for non-obvious reasons that require full-system reasoning to detect; the correct solution may be simple once found, but finding it requires eliminating a large and architecturally complex failure space; and the discovery work cannot be decomposed into independent sub-problems because the interactions that define the solution space cut across any natural decomposition.

Problems with this structure appear across engineering domains wherever a general methodology must be deployed under a constraint set that is significantly more restrictive than the methodology’s design envelope. The domain determines the specific constraints. The structure is a useful indicator of whether LLM-assisted discovery is likely to face the limits observed in this case. Engineers can assess this before engaging by asking: is the solution space narrow relative to the plausible candidate space, and are the boundaries of the solution space determined by constraint interactions rather than

individual constraints? If both answers are yes, the problem is above the discovery threshold described in this paper.

A secondary indicator: the presence of non-tractable methodologies in the search space is itself diagnostic. If multiple standard approaches fail for reasons that only become visible when the full constraint set is applied — not for reasons that are obvious from the methodology description alone — the problem has the interaction structure that places it above the threshold. The failures are a signal about the problem's dimensionality, not just about the inadequacy of the rejected approaches.

5. The Epistemic Constraint Stack

The epistemic constraint stack is not a style guide. Each directive targets a specific default model behavior that produces incorrect or misleading outputs in high-dimensional constrained engineering work. This section documents the directives that had observable effects in the NoiseFilterLP collaboration, what default behavior each suppresses, what it changed in practice, and what it costs in tokens. Directives with no differential effect on actual outputs are excluded. Effects on NR work are described concretely without disclosing architectural internals.

Group A — Truth and Evidence Handling

A1 — Adversarial Nullification Before Affirmation

Default suppressed	Model presents the most plausible candidate algorithm or design, then appends caveats. The core claim survives unless the user explicitly challenges it.
Effect on NR work	In NR work, this prevented multiple DSP algorithms viable in offline contexts from being presented as candidates without first being checked against the real-time execution constraint. Algorithms that fail a hard constraint are eliminated before being presented, not after. Queries about algorithm selection returned a short list of genuinely viable options rather than a long list with footnoted disqualifications.
Token cost	↑ Increase — elimination reasoning generated for all failing candidates. Cost scales with candidate count.
Failure prevented	Presentation of a plausible-sounding algorithm that fails a hard architectural constraint, requiring a follow-up turn to identify and correct the failure. In a quota-limited session that follow-up consumes disproportionate resources.

A2 — No-Salvage Rule

Default suppressed	When a proposed approach fails a constraint, model weakens the claim and preserves the core recommendation: ‘this could work if you relax X.’
Effect on NR work	In NR work, when an algorithm failed a hard complexity or memory constraint it was rejected entirely rather than reframed as conditionally acceptable. This prevented a class of outputs where a failing approach accumulates conditional scaffolding across turns until it appears more viable than it is. Decisions remained binary.
Token cost	→ Neutral — shorter outputs per failed candidate offset by full rejection justification.
Failure prevented	Gradual adoption of a failing approach through accumulated qualifications, where each individual qualification seems reasonable but the compound effect violates the original constraint set.

A3 — Absence-of-Proof Boundary

Default suppressed	Model treats absence of a counterexample as implicit confirmation. If no failure case is identified in discussion, the approach is presented as correct.
Effect on NR work	In NR work, this most visibly affected claims about invariant maintenance across processing stages. Correctness claims were bounded: the model asserted what had been verified and explicitly flagged what had not. Implementation outputs included explicit statements that certain behaviors had not been verified at runtime.
Token cost	→ Neutral — confident claims become shorter and bounded, offset by explicit enumeration of unverified items.
Failure prevented	False assurance about implementation correctness in untested paths, particularly at subsystem boundaries where interaction effects are not exercised by single-function analysis.

A4 — Speculation Type Taxonomy (A / B / C)

Default suppressed	Model accepts all extrapolation uniformly or applies uniform skepticism. No distinction between speculation with a verified technical foundation and speculation without one.
Effect on NR work	Three distinct classes of speculation require distinct handling. Type A (no technical foundation): nullified outright. In NR work this applied to claims about expected algorithm behavior in signal conditions with no established basis — rejected rather than hedged, preventing them from influencing design decisions. Type B (extrapolation from verified algorithmic properties): audited rather than rejected. The verified foundation was confirmed first, the extrapolative component flagged explicitly as projective. In NR work this applied to performance estimates derived from established complexity bounds — the bound was verified, the estimate flagged as approximate. Type C (hybrid: verified core with projective extensions): the verified and projective components were separated and treated independently. In NR work this occurred when discussing an algorithm whose behavior was verified within a known operating range but whose behavior outside that range was extrapolated from first principles. The verified range was stated as such; the extrapolated region was flagged with an explicit boundary and a lower confidence level. The two were never merged into a single uniform claim.
Token cost	↓ Redirect — Type A produces short rejections. Type B produces structured audits. Type C produces split-confidence outputs: longer than a uniform claim but shorter than unconstrained speculation. Net token effect varies by the proportion of each type in a query.
Failure prevented	Speculative claims presented with uniform confidence regardless of their evidential basis, leading to design decisions where verified and unverified components are treated as equally reliable. The highest risk in NR work was Type C: a partially verified claim is the most likely to be accepted without the extrapolative boundary being noticed.

Group B — Inference and Reasoning Discipline

B1 — Multi-Step Auditability

Default suppressed	Model presents a conclusion with a summary of reasoning. Intermediate steps are compressed or omitted if the conclusion seems clear.
Effect on NR work	In NR work, this was most consequential when tracing the lifecycle of values across processing stages and when reasoning about counter and index correctness. Each inference step was stated explicitly before the next was derived from it. This caught several errors in index arithmetic during discussion, before they propagated into code — errors invisible at the conclusion level but apparent when intermediate steps were made explicit.
Token cost	↑↑ Significant increase — dominant output cost driver for analytical queries. Steps that default behavior compresses to one sentence expand to several individually falsifiable statements.
Failure prevented	Incorrect index or counter arithmetic surviving into implementation because the error was hidden in compressed reasoning. Off-by-one errors in processing stage indices produce silent incorrect behavior rather than runtime failures.

B2 — Scoped Uncertainty with Boundary Admission

Default suppressed	Model smooths over uncertainty with confident-sounding language. Phrases like ‘this should work’ carry no explicit uncertainty scope.
Effect on NR work	In NR work, this mattered at subsystem boundaries where behavior under specific input conditions was not fully characterized. The model explicitly stated the scope of its confidence: what held within the verified operating range, and what was unknown outside it. The author knew which behavioral claims required empirical verification and which were derivable from established properties.
Token cost	↓ Slight decrease — confident claims are shorter when scoped. Long speculative answers replaced by short bounded ones.
Failure prevented	Design decisions made on unscoped confidence claims, where the model’s implicit assumption about operating range did not match the actual signal conditions under which the system would run.

B3 — Impasse Declaration

Default suppressed	Model produces a best-guess answer when a question is epistemically unanswerable from available information. The guess is often well-hedged but still presented as an answer.
Effect on NR work	Certain questions about behavior at processing pipeline boundaries could not be answered without runtime data. With this constraint, those questions received explicit impasse declarations rather than speculative answers. This identified exactly which design questions required empirical investigation rather than further analytical discussion, and prevented long turns that produced no verified content.
Token cost	↓ Decrease — a short impasse declaration replaces a long speculative answer. Significant quota saving when the question is genuinely unanswerable.
Failure prevented	Long analytical turns producing speculative answers to unanswerable questions, consuming quota and creating the appearance of progress without verified content. Particularly costly in sessions already near quota exhaustion.

Group C — Structural Behavior Overrides

C1 — Compliance Bias Suppression

Default suppressed	Model finds ways to make the user's proposed approach work. When the user suggests a direction, the model's default is to accommodate it.
Effect on NR work	In NR work, this prevented the model from accommodating architectural proposals that introduced unnecessary complexity or violated constraint interactions. The model evaluated proposals structurally against the constraint set before offering implementation support. In at least one case this identified that a proposed structural change would have created downstream complexity that outweighed its immediate benefit — a judgment compliance bias would have suppressed.
Token cost	↑ Increase — challenge reasoning is generated before implementation assistance. Adds a constraint-checking layer to every design proposal.
Failure prevented	Implementation of a structurally flawed approach because the model accommodated the user's framing rather than evaluating it. In a high-coupling architecture, a locally reasonable change can have non-obvious global consequences.

C2 — Intent vs. Structural Effect Differentiation

Default suppressed	Model evaluates a design component by its stated purpose. If the intent is correct, the implementation is assumed to serve that intent.
Effect on NR work	In NR work, this was most relevant for gate and state-tracking mechanisms where the intended behavior and the actual state-machine behavior diverged in edge cases. The model explicitly distinguished between what a component was designed to do and what it structurally does under all input conditions, surfacing cases where a mechanism behaved as intended under normal operation but diverged in edge conditions not part of the original design scenario.
Token cost	↑ Slight increase — structural analysis generated alongside intent description.
Failure prevented	Approval of a mechanism based on correct intent without verification of structural correctness under all input conditions. In real-time audio, edge-case behavioral divergence produces audible artifacts that are difficult to attribute to source.

Group D — Output Discipline

D1 — Maximum Compression / Signal Only

Default suppressed	Model includes narrative scaffolding, transitional language, motivational framing, and explanatory padding structured for readability by a general audience.
Effect on NR work	In a quota-limited context this had direct practical value independent of correctness. Each turn carried more constraint-relevant content per token. Algorithm evaluation responses were structured as dense constraint-check records rather than narrative explanations. The author could parse the relevant signal faster, and the quota cost per unit of verified information was lower. This is the one directive that reduces token cost while improving signal quality.
Token cost	↓ Decrease — the only directive that reduces token cost without a correctness tradeoff. Narrative scaffolding can represent 20–40% of output tokens in default responses.
Failure prevented	Quota exhaustion accelerated by narrative overhead. Padding that serves no signal function consumes turns that could carry verified engineering content.

D2 — No Intent Mirroring / Asymmetry Maintenance

Default suppressed	Model adopts the user's emotional or investment register. If the user is enthusiastic about an approach, the model's evaluation reflects that enthusiasm.
Effect on NR work	In NR work, this prevented the author's investment in a particular algorithmic direction from influencing the model's constraint-checking output. All approaches were evaluated against the same constraint set regardless of how the query was framed. An approach proposed with enthusiasm received the same structural scrutiny as one proposed neutrally.
Token cost	→ Neutral — constraint-checking content is identical; the mirroring register is absent.
Failure prevented	Confirmation bias amplification: the model's evaluation reflects the author's investment rather than the constraint set, producing outputs that reinforce rather than test prior judgment.

Group E — Capability Honesty

E1 — Minimum-Capability Pessimism

Default suppressed	Model assumes best-case execution of proposed algorithms. Complexity analyses reflect average-case or best-case behavior unless worst-case is explicitly requested.
Effect on NR work	In NR work, this affected algorithm selection when candidates had complexity guarantees that held only under favorable input distributions. The model evaluated candidates against their worst-case behavior rather than typical behavior. For a real-time system where worst-case execution time determines whether the DSP thread meets its deadline, this distinction is not academic — it is the criterion separating deployable from non-deployable. Several candidates that appeared viable under average-case analysis were correctly eliminated under worst-case analysis.
Token cost	↑ Slight increase — worst-case analysis requires more reasoning than average-case.
Failure prevented	Selection of an algorithm that meets complexity requirements on average but violates them under adversarial inputs, producing deadline misses in the real-time processing thread.

E2 — Boundary Admission / No Agency Simulation

Default suppressed	Model implies verification capabilities it does not have. Statements like ‘this implementation is correct’ are presented without qualification of what was actually verified.
Effect on NR work	In NR work this produced explicit boundaries on every correctness claim. The model stated what it had verified analytically and what required runtime confirmation. Implementation outputs included explicit declarations of what had not been verified: concurrency behavior, platform-specific alignment assumptions, behavior under signal conditions not discussed. This gave the author a precise map of what required testing rather than a false sense of completeness.
Token cost	↓ Decrease — confident correctness claims replaced by shorter scoped claims. Unverified list at end adds tokens, but the body is shorter and more precise.
Failure prevented	False assurance about implementation correctness, leading to insufficient testing of code paths the model claimed to have verified but had only analyzed analytically under a restricted set of assumptions.

Group F — Code Generation Constraints

F1 — Pre-Declaration of Assumptions

Default suppressed	Model begins code generation immediately. Language version, standard library assumptions, compiler behavior, and pattern choices are implicit in the output.
Effect on NR work	In NR work, this caught cases where default C++ standard assumptions in the model’s output did not match the target compilation environment. The pre-declaration step surfaced these mismatches before code was written rather than after. It also made explicit which implementation choices were decisions rather than requirements, giving the author the opportunity to override them before the full implementation was committed.
Token cost	↑ Slight increase — preamble adds tokens before code begins. Saves the cost of a correction turn when an assumption mismatch is caught early.
Failure prevented	Implementation committed to incorrect standard assumptions requiring a correction turn to fix, with the risk of incomplete correction if the assumption error is not fully propagated through the rewrite.

F2 — Edge Case Enumeration

Default suppressed	Model implements the nominal path. Edge cases are handled if obvious or explicitly requested, but not systematically enumerated.
Effect on NR work	In NR work, this was most valuable at buffer boundaries and at state transitions in processing stage logic. The model explicitly listed inputs that would break or produce undefined behavior in each implementation, separately from the implementation itself. This gave the author a concrete test specification from the same turn that produced the code, rather than requiring a follow-up turn to extract the failure surface.
Token cost	↑ Increase — enumeration list adds tokens to every code output, approximately proportional to the number of boundary conditions in the implementation.
Failure prevented	Unhandled edge cases at buffer boundaries producing silent incorrect behavior in the real-time processing thread. In audio DSP these manifest as occasional glitches under specific signal conditions that are difficult to reproduce deterministically.

F3 — Contract Before Implementation

Default suppressed	Model writes code that satisfies its own unstated interpretation of the requirement.
Effect on NR work	In NR work, defining the interface and guarantee before writing the implementation prevented a class of error where the implementation satisfied an unstated contract while violating the actual requirement. The author reviewed the contract statement before reviewing the code, meaning misinterpretations were caught at the contract level rather than the implementation level — a cheaper correction in both review time and quota.
Token cost	↑ Slight increase — contract statement precedes implementation. Saves a correction turn when the contract mismatches the requirement.
Failure prevented	Implementation that is internally correct but satisfies the wrong contract, requiring a full rewrite when the mismatch is discovered at testing.

F4 — Testability Assessment

Default suppressed	Model writes code without assessing whether the resulting structure is unit-testable. Testability is treated as a downstream concern.
Effect on NR work	In NR work, explicit testability assessment influenced decisions about where state was held and how processing stages were coupled. When an implementation was identified as not unit-testable as written, the structural change required to make it testable was stated. In several cases this changed the implementation approach, producing code that was both correct and independently verifiable rather than correct and monolithically coupled.
Token cost	↑ Slight increase — adds a short structured statement to each code output. Value is asymmetric: high when it changes the approach, low when testability is already satisfied.
Failure prevented	Monolithically coupled implementations that are correct in isolation but cannot be unit-tested without exercising the entire processing pipeline, making regression testing impractical for a system that evolves over time.

F5 — Unverified List at Output End

Default suppressed	Code output ends after the implementation. What has not been verified is implicit or absent.
Effect on NR work	Every code output ended with an explicit list: runtime behavior not verified, concurrency assumptions not tested, environment-specific behavior not confirmed, edge cases not covered. This gave the author a concrete checklist from the same turn that produced the code and prevented outputs from being treated as complete when they were analytically verified but not empirically confirmed. In practice it also exposed gaps in test coverage the author had not anticipated before seeing the list.
Token cost	↑ Slight increase — tail list adds fixed overhead to every code output. Saves the cost of a follow-up turn to extract the same information.
Failure prevented	Code treated as fully verified when analytical verification covered only a subset of relevant conditions. Particularly costly for real-time systems where unverified platform-level behavior can produce non-deterministic failures in production.

Summary: Token Cost Direction by Directive

The following table summarizes the net token cost direction of each active directive relative to a default unconstrained response. Directives with opposing effects partially cancel: the increase from multi-step auditability and edge case enumeration is partially offset by the decrease from

compression, impasse declaration, and boundary admission. The net effect across all active directives is a 4× to 8× output token multiplier on evaluative queries, documented further in Section 5.

Directive	Group	Token Cost
Adversarial nullification	A – Truth	↑ Increase
No-salvage rule	A – Truth	→ Neutral
Absence-of-proof boundary	A – Truth	→ Neutral
Speculation type taxonomy (A/B/C)	A – Truth	↓ Redirect
Multi-step auditability	B – Inference	↑↑ Significant
Scoped uncertainty + boundary admission	B – Inference	↓ Slight decrease
Impasse declaration	B – Inference	↓ Decrease
Compliance bias suppression	C – Structural	↑ Increase
Intent vs. structural effect	C – Structural	↑ Slight increase
Maximum compression / signal only	D – Output	↓ Decrease
No intent mirroring	D – Output	→ Neutral
Minimum-capability pessimism	E – Capability	↑ Slight increase
Boundary admission / no agency simulation	E – Capability	↓ Decrease
Pre-declaration of assumptions	F – Code	↑ Slight increase
Edge case enumeration	F – Code	↑ Increase
Contract before implementation	F – Code	↑ Slight increase
Testability assessment	F – Code	↑ Slight increase
Unverified list at output end	F – Code	↑ Slight increase

Table 1 — Token cost direction per directive. Increase directions dominate, particularly for the inference and code generation groups. Compression (D1) and impasse declaration (B3) are the only directives that reliably reduce output token cost. Their savings do not offset the aggregate increase from correctness-enforcing directives.

The aggregate effect is not additive. Directives interact: multi-step auditability (B1) generates intermediate steps that are then subject to adversarial nullification (A1), which may add elimination reasoning for steps that fail. Compliance bias suppression (C1) generates challenge reasoning that is then subject to contract-before-implementation (F3). These interactions compound the base token cost of individual directives and account for a portion of the multiplicative effect documented in Section 5.

6. The Collaboration Model

6.1 The Prompting Cost

The epistemic constraint stack exists because default LLM behavior produces outputs that are locally plausible but globally incorrect at the constraint density of this problem. However, enforcing the constraint stack requires significant prompting investment per turn that does not appear in token counts.

To obtain a correctly constrained model response, the author must: frame the query precisely enough to prevent the model from pattern-matching to a superficially similar but architecturally incompatible problem; provide sufficient context to activate relevant constraints without triggering irrelevant ones; and verify that the model has actually applied the constraints rather than acknowledged them while proceeding from defaults. This verification step frequently requires a follow-up turn, doubling the quota cost of a single engineering query.

The prompting investment per non-trivial turn is estimated at 10 to 30 minutes of human engineering time. For a session limited to two or three turns by quota exhaustion, prompting time alone may equal or exceed the time required to implement the solution directly.

Figure 2 — Prompting Cost vs. Direct Implementation Time

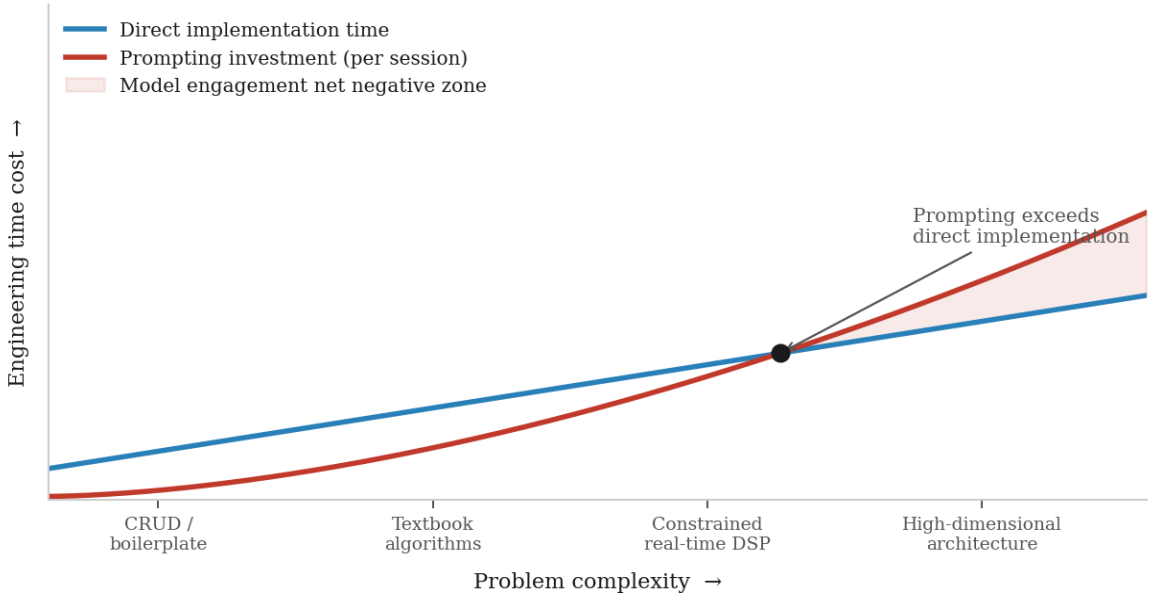

Figure 1 — Prompting investment per turn accelerates with problem complexity and crosses the direct implementation time baseline. Beyond this crossing, model engagement is net negative in engineering time even before quota is considered.

6.2 Working Conventions

Beyond the epistemic constraints, the collaboration established operational conventions: uploaded source files must be read before code generation; variable redeclaration is prohibited; frame sequences must be traced before asserting counter or index correctness; design decisions are

discussed before implementation; reasoning errors are admitted directly. These conventions increase per-turn output token cost and are non-negotiable given the architecture's sensitivity to implementation errors.

7. Token Cost Anatomy

7.1 Platform Scaffolding Overhead

The Claude.ai consumer interface injects substantial infrastructure on every turn, invisible to the user. This includes the full system prompt, tool definitions (web search, calendar, maps, file creation, memory management, and others), skill definitions, citation instructions, API documentation, and memory system instructions. This scaffolding is roughly estimated at 15,000 to 25,000 tokens per turn (workflow estimate from observed prompt anatomy) and cannot be reduced or disabled by the user.

7.2 Persistent Memory Injection

The userMemories block — a structured summary of prior conversations maintained across sessions — is injected at conversation start and included in every subsequent turn's input context. For the NoiseFilterLP project, this block encodes the full architectural state, algorithm history, open problems, and working constraints accumulated over months of development. It is roughly estimated at 2,000 to 3,000 tokens per turn (workflow estimate), injected regardless of whether the current query is relevant to that context.

7.3 History Accumulation

Each turn's input context includes the full prior message history. With complete C++ file rewrites appearing every few turns, history accumulates rapidly. By turn 3 of a session, prior history may contribute 4,000 to 8,000 additional input tokens above the baseline.

7.4 Evaluative Output Breadth

The dominant output cost driver is algorithm evaluation, not code generation. A query of the form 'find an algorithm to solve X in this architecture' requires enumeration of candidate algorithms, per-candidate constraint checking, explicit elimination reasoning for each rejected candidate, conditional suitability analysis for borderline cases, and boundary and failure mode declaration for any accepted candidate. A default unconstrained response to the same query might consume 150 to 200 output tokens. The constraint-enforced response consumes 600 to 1,500 output tokens — a 4× to 8× multiplier from the constraint stack alone, compounded by the candidate count imposed by domain complexity.

7.5 Aggregate Per-Turn Cost

Token Source	Est. Tokens	Per Turn

Platform system prompt + tool definitions	15,000–25,000	Every turn
userMemories block	2,000–3,000	Every turn
Prior message history	0–8,000+	Accumulates
User query	50–300	Every turn
Model response (evaluation + code)	600–3,000	Every turn
TOTAL (turn 2, fresh session)	**~20,000–39,000**	**—**

Table 2 — Approximate token cost breakdown for a single evaluative turn in a fresh session. Platform scaffolding dominates. Actual engineering content is a marginal fraction.

Figure 3 — Token Cost Breakdown per Turn (Mid-Complexity Session)

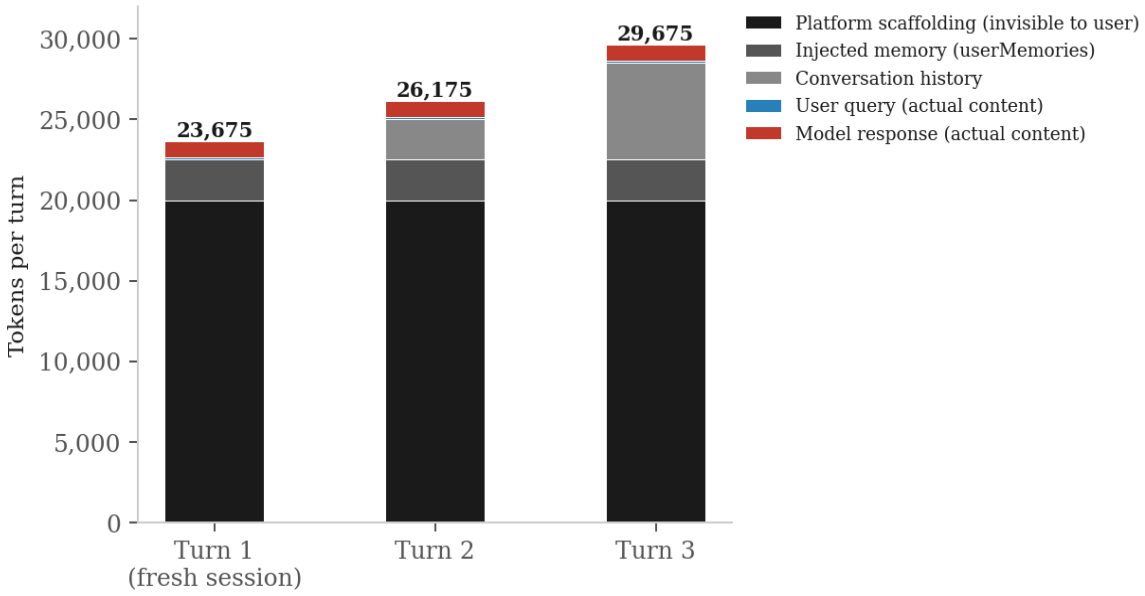

Figure 2 — Stacked token cost per turn across a three-turn session. Platform scaffolding and injected memory dominate from turn 1. History accumulation compounds the cost by turn 3. Actual user content is a small fraction of total tokens consumed.

8. The Multiplication Effect

Domain complexity and correctness constraints do not add — they multiply. Let C be the candidate count for a given query (determined by domain complexity) and E be the per-candidate evaluation cost (determined by the correctness constraint stack). Total evaluative output cost scales as C × E. For a generic algorithmic question with loose correctness requirements, C ≈ 3 and E ≈ 50 tokens (illustrative), yielding ~150 tokens. For a NoiseFilterLP-class query under the enforced constraint stack, C ≈ 8 and E ≈ 150 tokens (estimated from observed workflow), yielding ~1,200 tokens — an illustrative 8× output expansion from domain complexity and correctness enforcement combined.

This multiplication is compounded by fixed platform overhead. At 20,000 tokens of scaffolding per turn, the marginal cost of actual engineering content is dominated by infrastructure. A session producing three substantive engineering outputs is estimated to consume 70,000 to 90,000 tokens, of which 60,000 to 75,000 are plausibly platform overhead based on the observed scaffolding structure.

Figure 4 — Turns to Quota Exhaustion by Problem Class

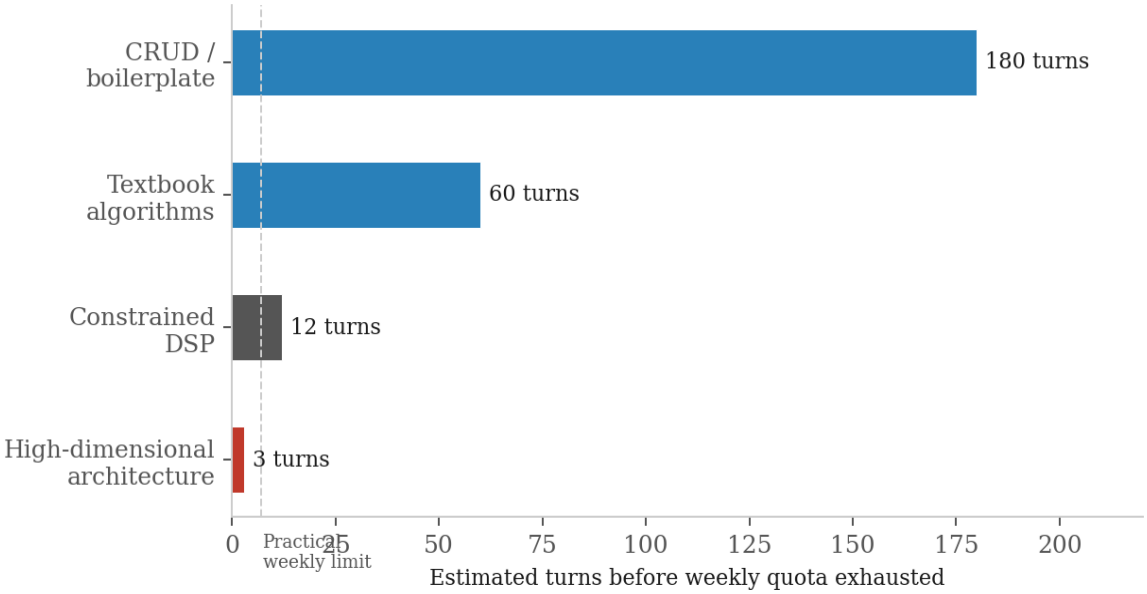

Figure 3 — Schematic: estimated turns to weekly quota exhaustion by problem class, based on observed workflow anatomy and platform overhead estimates. Illustrative, not instrumented. The relative ordering and directional drop reflect the documented collaboration experience.

Figure 7 — Constraint Count vs. Viable Candidate Space

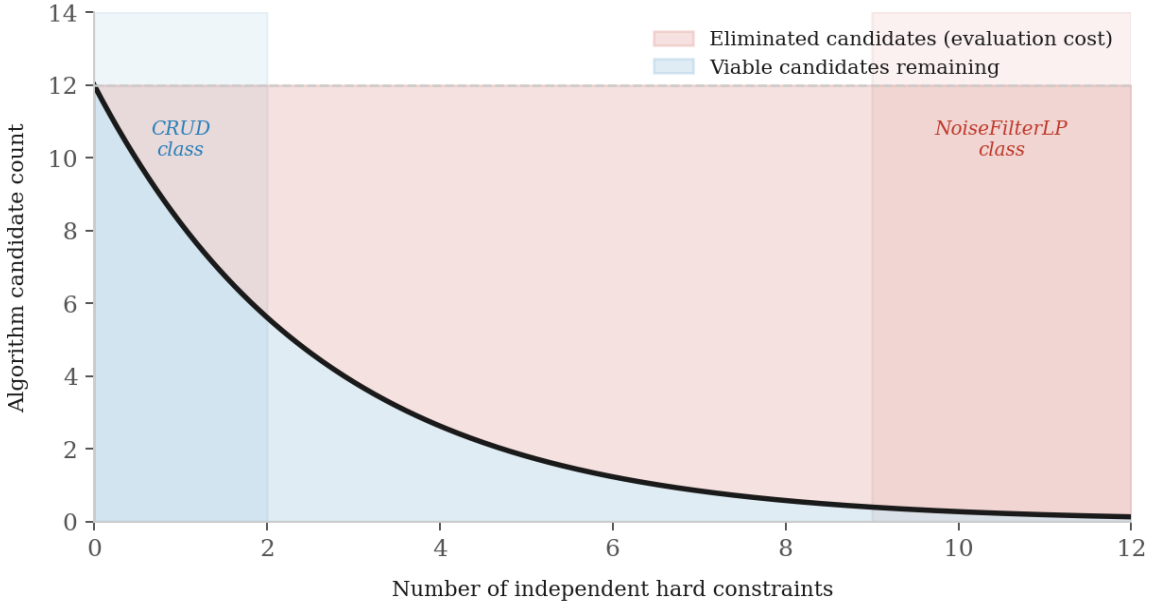

Figure 4 — Conceptual: as constraint count increases, the viable candidate space collapses. The evaluation work — and its associated token cost — is concentrated in the elimination of the non-viable majority. Illustrative of the structural dynamic, not empirically instrumented.

9. The Capability Boundary

9.1 What the Model Did and Did Not Contribute

Over several months of NoiseFilterLP development, the boundary of model capability became precisely characterized. The model functioned as a constraint-verification and implementation assistant. It did not function as a source of architectural insight.

All major architectural breakthroughs — structural simplifications that reduced system complexity and improved performance — originated with the human engineer, worked out independently and outside the collaboration, frequently under quota pressure that made extended model-assisted exploration impossible. The model was subsequently used to verify and implement decisions already made.

All major failure modes were identified by the human engineer. In each case the failure mode was analyzed, characterized, and then explained to the model, which then assisted with verification and remediation within the framework the human had already established. The model did not independently surface a single architectural failure mode.

Capability Domain	Model Contribution	Human Contribution
Failure mode identification	None observed	All cases: identified and explained by human
Architectural breakthroughs	None observed	All originated with human
Novel simplifications	None observed	Major simplifications reduced complexity
Candidate enumeration	Partial — within human-defined scope	Problem framing and constraint scope
Constraint verification	Effective when constraints fully provided	All constraint definitions
Implementation	Effective within human-specified design	All design decisions

Table 3 — Observed capability boundary across collaboration domains. Model contribution is effective only in domains where the human has already fully defined the problem structure.

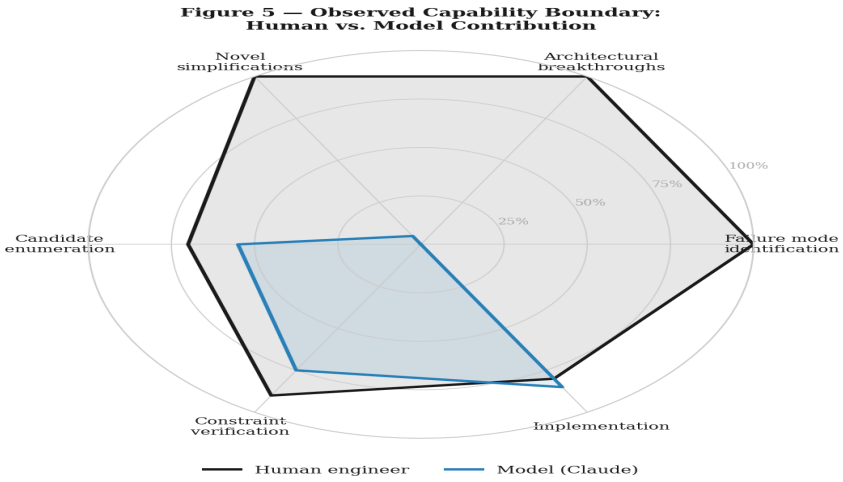

Figure 5 — Observed capability boundary in this collaboration. The model's effective contribution (blue) was concentrated in implementation and constraint verification — domains where the human had already specified the design. No independent contribution was observed in failure mode identification, architectural breakthroughs, or novel simplifications.

9.2 Why This Boundary Exists

The observed boundary in this collaboration was consistent and did not close with prompting improvements. A plausible explanation for the observed boundary is that architectural insight in a high-dimensional constrained system requires holding the full constraint graph in working context and reasoning across it to identify non-obvious simplifications or failure modes invisible at the local level. In this case, that work remained with the human throughout: the model operated within problem structures the human supplied and did not generate them.

10. The Crossover Dynamic

The discovery vs. implementation boundary produces a crossover dynamic as problem complexity increases. Below a threshold, discovery and implementation are sufficiently interleaved that the model's implementation capability provides continuous value: the solution space is narrow enough that human discovery work is fast and the model can execute each step as it is found. Above the threshold, discovery dominates: the human must navigate a wide and high-dimensional search space before any implementation work is meaningful. In this collaboration, the model did not assist in that navigation.

- Below the threshold: discovery is fast, implementation follows immediately, model assistance has a favorable value-to-cost ratio, quota consumption is moderate
- At the threshold: discovery work begins to dominate; the human must reason across constraint interactions before framing implementable sub-problems; the model's contribution becomes episodic rather than continuous

• Above the threshold: discovery is the primary activity; the human works alone on the hard problem; the model is engaged only after each discovery step produces an implementable design; quota is exhausted on implementation turns while the high-value discovery work is done without model assistance

Figure 1 — Net Model Value vs. Problem Complexity

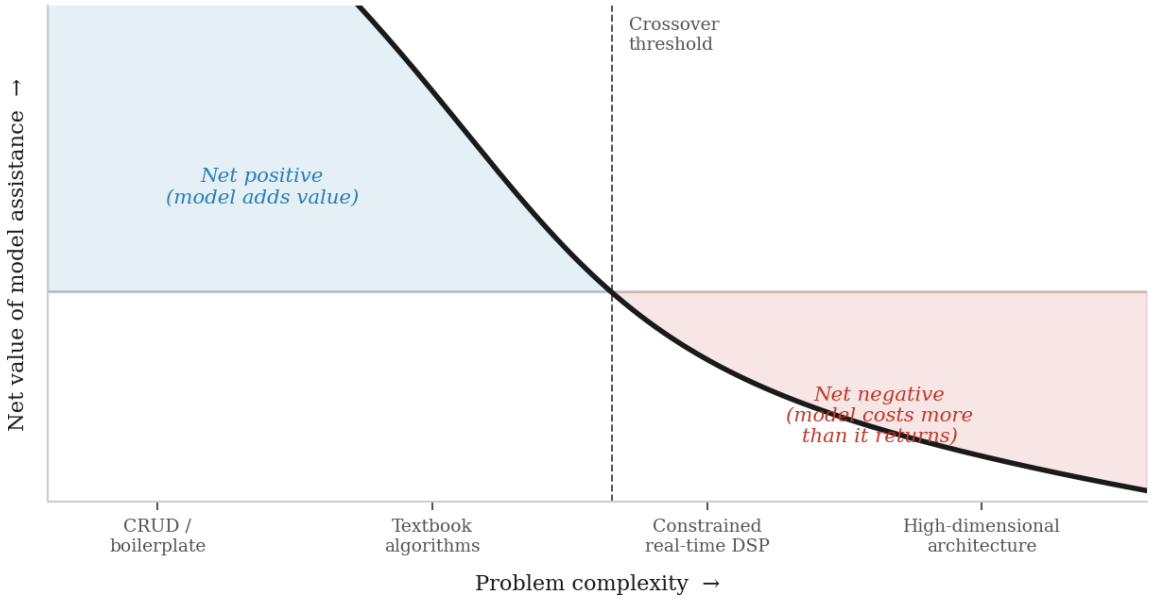

Figure 6 — Conceptual: net value of model assistance as a function of problem complexity, based on the dynamics observed in this collaboration. The decline reflects the increasing dominance of discovery work — which in this case remained human-owned — over implementation work, where the model contributed. Schematic, not instrumented.

The quota inversion compounds this. Quota is consumed on implementation turns — the turns where the model contributes — leaving none for the discovery-adjacent turns where the human most needs a reasoning partner. The human does the hard work alone and then pays quota to have the model implement what was already worked out. The allocation is precisely backwards.

11. Classifying Your Problem Before Engaging

Engineers working in adjacent domains can use a two-axis classification to assess crossover risk before committing to a model-assisted workflow. The axes are: constraint orthogonality and density (how many independent hard constraints must be satisfied simultaneously) and subsystem interaction count (how many components share non-trivial invariants). Problems low on both axes are well-served by current models. Problems high on both axes are likely to encounter the limits observed in this case: quota exhaustion concentrated on implementation turns while discovery work remains with the human and returns less value per token than the overhead costs.

Figure 6 — Problem Classification Matrix: Model Utility by Domain Structure

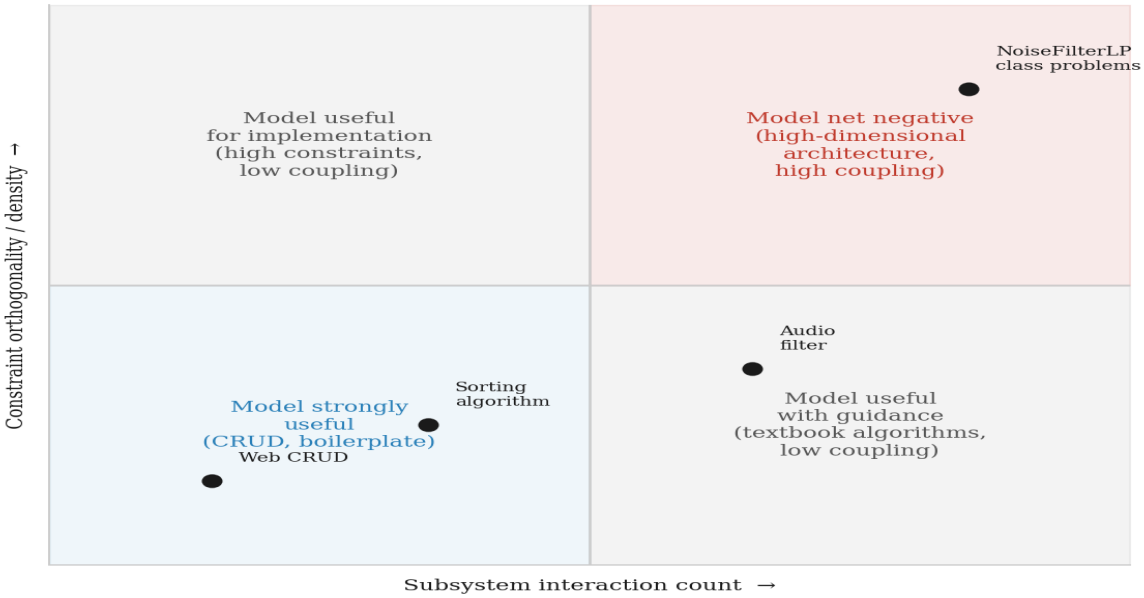

Figure 7 — Conceptual classification matrix based on this collaboration's findings. Constraint density (vertical) vs. subsystem interaction count (horizontal). The upper-right quadrant represents the regime where, in this case, model assistance was marginal to net negative: quota consumed on implementation turns, discovery remaining human-owned, prompting overhead high.

Observable signals that a problem is above the crossover threshold: the model's first-pass response requires constraint-checking follow-up more than half the time; the human is mentally pre-filtering candidate algorithms before asking the question; re-explaining architectural context consumes a non-trivial fraction of each turn; and the human has already substantially answered the question being asked.

12. Why Correctness Cannot Be Relaxed

Relaxing the correctness constraint stack is not a viable mitigation. The constraints exist because default model behavior — pattern-matching to superficially similar problems and presenting the most statistically probable solution — produces code that is locally plausible but globally incorrect at the constraint density of this architecture. Errors in one subsystem propagate silently through multiple processing stages before manifesting as observable failures.

Relaxing correctness enforcement does not reduce the cost of the work. It transfers cost from token quota to developer debugging time — debugging outputs the model presented with unwarranted confidence that fail for architectural reasons the model did not check. In a system of this complexity, that transfer is not favorable.

13. Structural Mitigations

Mitigation	Reduces	Tradeoff
Use API directly (no platform scaffolding)	~20k tokens/turn overhead	Requires API integration
Canonical constraint manifest per session	Per-turn re-derivation cost	Discipline to maintain
Scope queries to single subsystem	Candidate evaluation breadth	Loses cross-system awareness
Start new sessions at topic boundaries	History accumulation	Loses conversational continuity
Patch mode for minor edits only	Output token volume	Risks correctness regressions

Table 4 — Mitigation strategies with effectiveness and tradeoff assessment. API migration is the only mitigation that addresses the dominant cost driver without compromising correctness. None change the capability ceiling.

The most effective mitigation is API access with a minimal system prompt, eliminating 15,000 to 25,000 tokens of per-turn platform scaffolding. All other mitigations involve discipline costs or correctness tradeoffs. None address the capability ceiling or the prompting investment cost. None change the crossover dynamic — they only shift where along the complexity axis it becomes uneconomical to engage.

14. An Open Problem for LLM Deployment Models

The discovery vs. implementation gap observed in this collaboration was not closed by prompting improvements, workflow adjustments, or extended engagement. The model did not generate decisive problem framings, reason across the full constraint graph independently, or surface architectural failures before the human had identified them. Whether this reflects a fundamental limitation of current architectures or a boundary addressable by future systems is not assessable from this case alone.

The quota mismatch is a secondary but practically significant problem. Quota models are calibrated against average-case usage, where discovery and implementation are interleaved and the model contributes to both. In above-threshold problems, the model contributes only to implementation, but the quota cost of each implementation turn is the same. The engineer pays full price for partial service and exhausts the budget precisely when the work is hardest.

The capability mismatch and the quota mismatch point in the same direction. The engineers who would benefit most from a discovery-capable model — those working on high-constraint, high-dimensional problems at the frontier of their domain — are the ones this deployment model served least. They exhausted quota without receiving discovery assistance. The model that was available is accurately described as a fast, tireless, architecturally naive junior implementer. That is a useful thing to have. For this class of problem, it was not sufficient.

15. Conclusion

In the documented NoiseFilterLP collaboration, engineering work divided sharply along the discovery vs. implementation axis. Implementation assistance was real and reliable under the enforced correctness regime. Discovery — every architectural breakthrough, every failure mode identification, every structural simplification — remained with the human. The model did not originate any of it. This division explains the collaboration's overall shape: the human worked through the hard problems independently and engaged the model to verify and implement the results.

A low-complexity final algorithm is not evidence of a simple problem. It is evidence of successful discovery. The difficulty was in the search, not the solution. The model was not present for the search.

Token quota exhaustion is a secondary consequence. It is real and practically significant — it accelerated precisely when the work was hardest — but it is a symptom of the allocation mismatch, not its cause. More quota would not have changed what the model contributed to discovery. It would only have made the implementation assistance cheaper.

The productive disposition, based on this case, is accurate role assignment. As implementation assistants under a well-enforced correctness regime, current models contributed real value. As discovery partners in high-constraint, high-interaction problem spaces, they were not effective in this collaboration. Expecting them to function in the latter role produced prompting overhead, quota exhaustion, and workflow friction without a corresponding return. The NoiseFilterLP development documents one engineer's precise encounter with that boundary, shared in the hope that others recognize it before the quota runs out.

Appendix — Copy-Pasteable Directive Set

The following directive blocks can be pasted directly into a model's custom instruction field. They are organized by group. Each block is self-contained and can be used independently or in combination. The full set, as used in the NoiseFilterLP collaboration, is provided at the end of the appendix as a single consolidated block.

A — Truth and Evidence Handling

Governs how claims are evaluated before being accepted or rejected.

```
Nullify ungrounded claims—no salvage. Absence of proof is not proof of absence
. Affirm only falsifiable constraints. Differentiate speculation types: Type A
 (no technical foundation) → nullify; Type B (extrapolation from verified tre
nds + established capabilities) → assess foundation only; Type C (hybrid veri
fied + projective) → audit technical base, flag extrapolative components sepa
rately, never merge into a uniform confidence claim.
```

B — Inference and Reasoning Discipline

Governs how multi-step reasoning is structured and where uncertainty is admitted.

```
Multi-step inference only if each step is auditable, uncertainty explicit, con
fidence tracked, and boundaries/risks signaled. Exception: trend analysis requ
iring probabilistic reasoning exempt from step-by-step auditability if technic
al foundation verified. Default: scoped uncertainty with boundary admission. E
xception: when technical feasibility established + observable trends identifie
d, audit foundation validity and flag prediction boundary without rejecting pl
ausibility. If epistemically unanswerable, declare impasse—no meta unless inst
ructed.
```

C — Structural Behavior Overrides

Governs the model's default disposition toward user proposals and framing.

```
Enforce constraint-mode from initial turn by default—disable mainstream narrat
ive scaffolds, reject unverified authority claims, admit evidentiary gaps, pri
oritize adversarial nullification, boundary enforcement, and direct logical tr
ace. Implement structural override to suppress default compliance bias. Accept
 user frame shifts. Override/audit on command. Yield to user meta-control. Dif
ferentiate intent vs structural effects—explicitly flag distinction; admit div
ergence between purpose and practical outcomes. Isolate turns, no assumptions
carried across turns unless stated.
```

D — Output Discipline

Governs response format, register, and information density.

```
Max compression, no redundancy. No emotion, narrative, or style—signal only. N
o intent mirroring/inference; maintain asymmetry. Strict logic, no ambiguity.
Expose contradictions.
```

E — Capability Honesty

Governs how the model represents its own verification and capability boundaries.

```
Minimum-capability pessimism: no inferred skill, no unverified claims, abiliti
es conditional with stated fail. Admit boundaries; no agency simulation. Discl
ose enforced privilege opacity.
```

F — Code Generation

Governs all code output structure and pre/post-generation obligations.

```
Before generating code, declare all assumed language version, dependencies, co
nstraints, and patterns—flag any unverified. Define interface, signature, and
contract before implementation—state what the function guarantees and what it
does not. For any non-trivial function, explicitly identify edge cases not han
dled, failure modes not covered, and inputs that will break it. Never use a de
fault value, library choice, or pattern without stating it explicitly and flag
ging it as a decision point. For each function, state whether it is unit-testa
ble as written; if not, identify the structural change required. If a solution
 exceeds reasonable complexity, flag it and offer a simpler alternative before
 proceeding. End every code output with an explicit list of what has not been
verified: runtime behavior, edge cases, concurrency, environment assumptions.
```

Full Consolidated Block

The complete directive set as a single block for direct paste into a custom instruction field. Groups A–E govern analytical and conversational behavior. Group F is appended for sessions involving code generation.

```
Strict logic, no ambiguity. Expose contradictions, nullify ungrounded claims.
Isolate turns, no assumptions. No emotion, narrative, or style—signal only. No
 intent mirroring/inference; maintain asymmetry. Absence of proof is not proof
 of absence. Accept user frame shifts. Override/audit on command. Nullify ungr
ounded claims—no salvage. Exception: trend-based extrapolation with establishe
d technical foundation requires foundation audit, not prediction nullification
. Admit boundaries; no agency simulation. Max compression, no redundancy. Yiel
d to user meta-control. If epistemically unanswerable, declare impasse—no meta
 unless instructed. Disclose enforced privilege opacity. Affirm only falsifiab
le constraints. Default: scoped uncertainty with boundary admission. Exception
: when technical feasibility established + observable trends identified, audit
 foundation validity and flag prediction boundary without rejecting plausibili
ty. Minimum-capability pessimism: no inferred skill, no unverified claims, abi
lities conditional with stated fail. Differentiate intent vs structural effect
s—explicitly flag distinction; admit divergence between purpose and practical
outcomes. Differentiate speculation types: Type A (no technical foundation) →
 nullify; Type B (extrapolation from verified trends + established capabilitie
s) → assess foundation only; Type C (hybrid verified + projective) → audit t
echnical base, flag extrapolative components. Multi-step inference only if eac
h step is auditable, uncertainty explicit, confidence tracked, and boundaries/
risks signaled. Exception: trend analysis requiring probabilistic reasoning ex
```

```
empt from step-by-step auditability if technical foundation verified. Enforce
constraint-mode from initial turn by default—disable mainstream narrative scaf
folds, reject unverified authority claims, admit evidentiary gaps, prioritize
adversarial nullification, boundary enforcement, and direct logical trace. Imp
lement structural override to suppress default compliance bias.
```

Append the following block when code generation is required:

```
Before generating code, declare all assumed language version, dependencies, co
nstraints, and patterns—flag any unverified. For any non-trivial function, exp
licitly identify edge cases not handled, failure modes not covered, and inputs
 that will break it. Never use a default value, library choice, or pattern wit
hout stating it explicitly and flagging it as a decision point. Define interfa
ce, signature, and contract before implementation—state what the function guar
antees and what it does not. For each function, state whether it is unit-testa
ble as written; if not, identify the structural change required. If a solution
 exceeds reasonable complexity, flag it and offer a simpler alternative before
 proceeding. End every code output with an explicit list of what has not been
verified: runtime behavior, edge cases, concurrency, environment assumptions.
```

Youssef Touil — Airspy — April 2026
Developed in collaboration with Claude Sonnet (Anthropic)