From Messages to Worlds

Transformational Communication, Semantic Version Control,
and the Collaborative Evolution of Shared State

Flyxion

Independent Researcher

April 2026

Abstract. This essay proposes a shift in how we understand communication between agents—individual or artificial—from the transmission of messages to the proposal and evaluation of transformations on shared or implicit states. Beginning with the observation that version control platforms such as GitHub produce a qualitatively different kind of communication than email or chat, we develop a theoretical account of why: commits function as structured morphisms acting on an evolving shared object, and merge operations implement constraint closure over competing local deformations. We extend this analysis to large language model dialogue, where prompts function as minimal semantic perturbations of an implicit latent state, and iterative conversation traces trajectories through a semantic manifold. These two regimes—the explicit and structured, the implicit and fluid—are shown to be instances of a single underlying framework: transformational communication, in which meaning is enacted through admissible deformation of a shared world rather than inferred from free-form text. We conclude by gesturing toward a fully realized system in which semantic states, semantic transformations, and constraint enforcement are all made explicit, and in which knowledge emerges through constrained composition rather than assertion.

Introduction: Communication Beyond Messages

The standard picture of communication is essentially hydraulic. A sender encodes meaning into a message, transmits it through a channel, and a receiver decodes it, recovering—in ideal conditions—something close to the original meaning. Noise, ambiguity, and misinterpretation are treated as engineering problems, failures of the channel or the codec rather than symptoms of something deeper. This picture underlies most of the design philosophy behind email, chat, and even formal correspondence: the goal is to get the message through.

But there is a puzzle lurking in the everyday experience of collaborative work. Developers often report that communication on GitHub—through issues, pull requests, and code review—feels more precise, more actionable, and less prone to fundamental misunderstanding than the same conversations conducted over email. This is puzzling because GitHub is, on the surface, more constrained. You can only comment on specific lines of specific files; your proposed changes must be expressed as diffs; your assertions are evaluated not by rhetorical force but by whether they survive integration into a shared codebase. The system is more demanding, not less. And yet something about this constraint appears to generate clarity rather than friction.

This essay argues that the puzzle dissolves once we recognize that GitHub and email are not doing the same thing. Email is doing something like classical message-passing: participants produce assertions, elaborate positions, and attempt to coordinate by accumulating a shared record of statements. GitHub is doing something categorically different, which we will call *transformational communication*: meaning is conveyed not through statements about the world but through structured changes to a shared object. The repository is the world; the commit is the act; the merge is the judgment. Communication becomes inseparable from transformation.

This reframing has consequences that extend well beyond software development. Once the distinction is clear, it becomes visible in other domains. Large language model interaction turns out to have a strikingly similar structure—prompts as perturbations, conversations as trajectories, hallucination as constraint failure—but in an underconstrained and implicit register. We propose that both phenomena are instances of a more general framework, and

that understanding this framework is important not only for building better collaborative tools, but for developing a richer theory of what communication is and what it is for.

From Email to GitHub: The Failure of Unstructured Discourse

Email is an extraordinarily flexible medium, and this flexibility is also its deepest limitation. A single email thread may simultaneously carry: a proposed decision, an objection to that decision, a correction to a factual claim made in the objection, a social gesture of accommodation, a tacit withdrawal of the original proposal, and an implicit negotiation about who has authority to decide. All of this is entangled in a linear stream of natural language, and the participants must maintain, entirely in their own minds, a model of the current state of affairs. There is no external representation of this state. There is no object that one can point to and say: *this is where we are*.

The result is that email threads about substantive decisions are typically very difficult to reconstruct after the fact, even for participants. The state of a project—who has agreed to what, what has been decided, what remains open—must be inferred from the archaeology of a message archive. This is not merely inconvenient. It is a structural feature of the medium: email externalizes assertions but internalizes state. Participants must reconstruct context continuously, from fragments, and this reconstruction is inherently lossy and divergent.

GitHub externalizes the state. The repository is a shared object that all participants can inspect, modify, and reason about. When a developer opens an issue, the issue is not a message expressing a concern; it is a structured object representing a tracked deficiency in the shared artifact. When they submit a pull request, they are not proposing a course of action in the abstract; they are constructing a concrete candidate transformation of the shared state and submitting it for evaluation. Comments are not free-floating assertions but are anchored to specific locations within the structure—a particular line of a particular file in a particular version—so that their referent is unambiguous.

The transition from email to GitHub is therefore not a tooling improvement. It is a shift in the underlying ontology of communication. Email treats communication as the production of statements; GitHub treats it as the production of transformations. The repository is not

the backdrop against which communication occurs; it is the medium of communication itself. To say something in GitHub is to change something—or to propose a change, or to evaluate a proposed change. There is no way to speak without acting on the shared world.

This has immediate practical consequences. Ambiguity in email arises partly because it is never fully clear what the object of a statement is. In GitHub, the object is always specified: this comment is about this line of this file. This does not eliminate misunderstanding, but it localizes it. Disagreements can be isolated to specific sites in the shared structure, rather than ramifying through an entire interpretive context. The system is, in a precise sense, better at pointing.

Commits as Primitive Transformations

The commit is the fundamental unit of GitHub communication, and it deserves more theoretical attention than it typically receives. In most practical discourse, a commit is described as a snapshot, a saved state, a point in the project's history. This description is accurate but misleading. It treats the commit as a noun—a thing that exists—rather than as a verb, an act that was performed. The snapshot characterization obscures the relational structure: a commit is not merely a state, but a transition from one state to another. It is, in the language of category theory, a morphism.

The developer norm of "one logical change per commit" is typically justified pragmatically: it makes history easier to read, bisection easier to perform, rollback more surgical. But this justification, while correct, misses the deeper logic. The norm is an attempt to approximate *semantic minimality*. A commit that contains exactly one logical change is one that cannot be decomposed into two independent transformations without losing something essential. It is a primitive in the algebra of change—an element that cannot be further factored.

Current version control systems can only approximately enforce this norm, because they operate at the level of textual representations rather than semantic objects. A commit is, mechanically, a set of line-level diffs—insertions and deletions in files of bytes. This is a representation of a transformation, not the transformation itself. And the mismatch between the textual level and the semantic level is pervasive. Two very different sequences

of textual edits may correspond to the same semantic change—say, renaming a function across a codebase. A single textual insertion of one line may constitute an enormous semantic transformation—adding an invariant that propagates through the entire reasoning about a system. The text is a projection of the semantic object, and like all projections, it loses information.

The developer who crafts atomic commits is compensating, manually, for this mismatch. They are performing a semantic decomposition in their own reasoning that the system cannot perform for them, and then encoding the result of that decomposition in the textual medium that the system does understand. This is an enormous amount of invisible cognitive labor, and the fact that it is routinely performed, and that its results are legible to other developers, suggests that there is a real structure being tracked—a structure that the current systems approximate but do not directly represent.

The Limits of Text: Syntax versus Semantics

The difficulty runs deeper than a gap between coarse and fine granularity. It is a gap between two different kinds of space. Textual diffs operate in what we might call the representation space: the space of syntactic structures, encodings, sequences of characters. Meaning, however, lives in a different space—a semantic space whose geometry is not determined by the syntax of the encoding but by the inferential and functional relationships among the entities being described.

The relationship between these spaces is not isometric. The map from semantic space to representation space is many-to-one: many different texts have the same meaning. This is why refactoring—restructuring code without changing its behavior—is possible and useful. It is also why the same mathematical proof can be written in many equivalent notations, why natural language admits paraphrase, why two programs in different languages can implement the same algorithm. Semantic equivalence is a much coarser equivalence relation than textual identity.

But the map also has regions of extreme sensitivity: small textual changes can correspond to large semantic changes. A single character determines whether a conditional is negated. A one-word change in a legal document may alter its entire meaning. A subtle type error

can produce behavior that diverges catastrophically from the intended semantics. In these regions, the projection is wildly distorting: the geometry of the semantic object is not locally preserved by its textual representation.

The research literature on program analysis and version control has recognized these difficulties and produced partial responses. Abstract syntax tree diffing attempts to operate at a structural level above raw text, tracking changes to syntactic units—functions, declarations, blocks—rather than lines. Semantic merge tools attempt to resolve conflicts not by aligning text but by reasoning about the behavioral equivalence of competing changes. Formal verification systems attempt to specify and check semantic properties explicitly, creating a separate domain in which meaning can be reasoned about directly.

These efforts are valuable, but they remain confined to syntactic structure. An abstract syntax tree is still a representation, just a more structured one. It tracks the shape of the code rather than its behavior, its type structure, its specification, or its intended function. A change that preserves the AST may break the program's behavior; a change that restructures the AST may leave the behavior entirely intact. Moving from line diffs to tree diffs is progress, but it does not reach semantic space. It reaches a richer representation space, and the gap remains.

Toward Semantic Commits

Given this analysis, we can now state a positive proposal. A *semantic commit* is a minimal admissible transformation in semantic space—a change that is primitive (not decomposable into independent sub-changes), admissible (consistent with the constraints on the shared object), and complete (expressed directly in terms of semantic relationships rather than their textual encoding).

In this framing, the repository is not a collection of files. It is an evolving structured object, a mathematical entity with a type, constraints, and a history of transformations. The semantic commits are composable morphisms acting on this object: they can be sequenced, and the composition of two commits is itself a commit if it is admissible. This gives the history of a repository the structure of a category, or perhaps a groupoid, where objects are states of the shared artifact and morphisms are semantic transformations

between them.

Merge, in this framework, acquires a precise meaning. Two commits starting from the same state are mergeable if and only if they can be composed without violating the constraints on the shared object. This is not a question about textual overlap—the familiar notion of a merge conflict—but about semantic compatibility. Two changes conflict semantically if and only if the state produced by their composition fails to satisfy some constraint that both branches individually satisfy. Merge is thus a gluing operation: given two locally admissible deformations, determine whether their combination remains globally admissible.

This is, of course, a much harder problem than textual merging. Semantic admissibility requires a formal specification of what the shared object is supposed to be, and reasoning about that specification under composition. For software systems, this is roughly the problem of compositional verification, which remains an active research frontier. But the difficulty of the problem does not undermine the clarity of the concept. Semantic commits are what developers are already attempting to produce, and semantic merge is what code review is already attempting to evaluate. The proposal is to take these implicit norms seriously as a theoretical framework.

Merge, Conflict, and Constraint Closure

The operation of merge, reconsidered semantically, reveals a general pattern that deserves a name: *constraint closure*. A shared object is characterized by a set of constraints—invariants, specifications, typing rules, behavioral properties—that must hold at every admissible state. A transformation is admissible if it maps admissible states to admissible states. And the closure operation is the process by which a candidate state is evaluated against the constraint set and either accepted, rejected, or repaired.

In a version control system, this process is partially automated and partially social. The automated part is whatever continuous integration and testing infrastructure the project maintains: the build must pass, the tests must pass, the linter must not complain. The social part is code review: the maintainers must be convinced that the proposed transformation is semantically appropriate, that it addresses the stated problem, that it does not introduce

new problems, and that it fits with the intended trajectory of the project. A rejected pull request is a finding that the proposed transformation is not admissible under the community's constraint set, even if it passes all automated checks. A reverted commit is a post-hoc finding that a previously accepted transformation was not actually admissible.

What is striking is that this process defines, operationally, what counts as true within the system. The repository's current state is not the set of all proposed changes; it is the closure of all proposed changes under the constraint set. Truth is what survives iteration. This is not a relativistic notion—the constraints are real and often formally specified—but it is an operational one. The system does not have privileged access to some external standard of correctness; it has a community of agents applying a set of constraints to a sequence of proposals, and the result of that process, accumulated over time, is what the system knows.

This perspective reframes the status of branches, conflicts, and reverts. A branch is a local exploration of a semantic neighborhood: a sequence of transformations that have not yet been evaluated for global admissibility. A conflict is not a problem to be fixed but information: it signals that two local explorations have made incompatible assumptions about the shared object, and that the constraint closure process must adjudicate between them. A revert is a correction: the constraint set was applied more carefully, and a previously accepted state was found to be inadmissible after all. Each of these operations reveals something about the geometry of the semantic space being navigated.

GitHub as a Proto-Epistemic System

We can now state a stronger claim. GitHub is not merely a development tool. It is a primitive *epistemic institution*: a system in which knowledge is produced and validated through a specific social and technical process. What distinguishes it from other epistemic institutions is its particular form of discipline: claims must be expressed as transformations of a shared object, and they are evaluated by being applied to that object under constraint enforcement.

This discipline has a number of important features. First, it forces explicitness. A claim that cannot be expressed as a transformation of the shared object—a vague concern, a

general feeling that something is wrong—cannot enter the system. To participate in GitHub's epistemic process, you must be able to point to something specific and propose a specific change. Second, it makes disagreement precise. Conflicts are not about who has better reasons; they are about which transformations are jointly admissible. This does not eliminate judgment, but it locates it. Third, it creates a record. The history of the repository contains not only the current state of knowledge but the full trajectory by which that state was reached—including the proposals that were rejected, the conflicts that were resolved, and the branches that were abandoned.

The history is therefore epistemically richer than the current state alone. It records not only what was accepted but which proposed realities failed to stabilize. The rejected pull request is not merely a failure; it is a documented exploration of a region of semantic space that turned out to be inadmissible under the system's constraints. This is how science is supposed to work—through publication of negative results—but it is rarely achieved in practice. GitHub achieves it structurally.

There is a sense in which GitHub instantiates a form of distributed Bayesian reasoning, without representing it explicitly. Each proposed transformation is a hypothesis about the shared object; the constraint closure process is the evaluation of that hypothesis against the accumulated evidence of the specification; the merge is the acceptance of the hypothesis; the history is the posterior. The analogy is imperfect in many ways, but the structural parallel is genuine. GitHub is a system in which belief, represented as a state of a shared object, is updated through structured proposals evaluated under constraint.

LLM Dialogue as Implicit Semantic Version Control

We now turn to a different kind of system, which at first appears to have little in common with what we have described. A conversation with a large language model is fluid, informal, unconstrained. There is no repository, no explicit state, no formal constraint set. The model produces text in response to text, and the conversation accumulates as a linear sequence without external anchoring. And yet, on closer examination, the structural parallel is striking.

Consider a prompt. It is not, semantically, a message transmitted to a receiver who decodes

it. It is a perturbation applied to the model's implicit state, which we can understand as a position in a high-dimensional semantic manifold—the space of possible continuations, weighted by the model's learned distributions. The model's response is not a decoded message but the output of a deformation of this implicit state under the perturbation. The conversation as a whole traces a trajectory through the semantic manifold, with each prompt acting as a local displacement and each response defining the new position from which the next displacement begins.

This is semantic version control in an entirely implicit register. The state is the model's current context window and the distribution it induces over possible next tokens—not a repository, but a position in latent space. The transformation is the prompt—not a commit, but a deformation. The constraint set is the model's training-induced preferences and the user's accumulating expectations—not a formal specification, but a set of implicit pressures that shape which trajectories are acceptable. And the evaluation of admissibility is not explicit merge review but the user's judgment about whether the response is useful, coherent, and on-target.

The failures specific to LLM interaction become visible in this framework. Hallucination is constraint failure: the model generates a response that is locally fluent but violates the constraint of factual accuracy that the user implicitly imposed. Drift is trajectory instability: over a long conversation, the model's position in semantic space slowly diverges from the intended attractor, and the responses become less coherent with respect to the user's goals. Prompt sensitivity is the analog of the distorting map between text and semantics: small changes in phrasing can produce large changes in the trajectory, because the model's implicit constraint set is not explicitly specified and has complex local geometry.

Unlike GitHub, the LLM system lacks explicit state, explicit transformations, and explicit constraint enforcement. All three are present implicitly, in the model's internal dynamics and in the user's expectations, but none of them are made available as objects that can be inspected, versioned, or formally evaluated. This is precisely why the system is underconstrained: without explicit state, there is nothing to anchor the trajectory; without explicit transformations, there is no way to ensure minimality or admissibility; without explicit constraint enforcement, there is no mechanism for detecting when the trajectory has left the region of acceptability.

Prompts as Semantic Commits

The analogy between commits and prompts can be made precise enough to be productive. A good prompt—one that reliably produces useful output and advances the conversation toward a coherent goal—has exactly the properties we attributed to a semantic commit: it is minimal (it introduces exactly one semantic perturbation), admissible (it applies to the current implicit state without violating the conversation's established constraints), and complete (it fully specifies the intended transformation without relying on the model to infer multiple unstated intentions simultaneously).

A poor prompt typically fails in one of two ways. It may be semantically overdetermined: it introduces multiple simultaneous perturbations, asking the model to make several independent changes at once, and the result is a response that addresses all of them partially and none of them well. This is the prompt equivalent of a commit that bundles multiple logical changes, violating semantic minimality. Or it may be semantically underdetermined: it does not adequately specify the intended transformation, leaving the model to explore a region of semantic space that may not intersect with the user's goals. This is the equivalent of a commit whose diff is valid but whose semantic intent is unclear.

The practice of iterative prompting—breaking a complex request into a sequence of simpler prompts, building toward a complex output through incremental steps—is best understood as manual semantic decomposition. The user performs, explicitly, the work that a semantic version control system would perform automatically: they identify the primitive transformations that compose to produce the intended result, and they apply them in sequence, evaluating the intermediate states for admissibility before proceeding. This is cognitively demanding, and the demand falls entirely on the user because the system provides no scaffolding for it.

The user in a sophisticated LLM interaction is therefore playing the role of both developer and version control system. They are proposing transformations (prompts), evaluating intermediate states (responses), deciding which trajectory to continue (conversation flow), and enforcing admissibility constraints (redirecting when the model drifts or hallucinates). The model's role is to apply transformations fluently; the user's role is to ensure that the

right transformations are applied in the right order. When this collaboration works well, it produces outputs of remarkable sophistication. When it fails, it typically fails because the user does not have adequate tools for representing the implicit state and the trajectory, and therefore cannot make reliable judgments about admissibility.

Toward a General Theory of Transformational Communication

Having traced the parallel between GitHub and LLM interaction at some length, we are in a position to propose a general framework. Let us define *transformational communication* as any system of interaction with the following structure: there exists a shared or implicit state; agents interact by proposing candidate transformations of that state; proposed transformations are evaluated against a constraint set; admissible transformations update the state; inadmissible transformations are rejected or repaired; and the history of accepted transformations constitutes the system's knowledge.

This framework generalizes the two cases we have examined. In GitHub, the shared state is explicit (the repository), the transformations are partially explicit (commits), the constraint set is partially formal (CI, tests) and partially social (code review), and the history is fully explicit and immutable. In LLM dialogue, the state is implicit (the model's context and latent position), the transformations are informal (prompts), the constraint set is entirely implicit (training priors and user expectations), and the history is explicit only as a sequence of text rather than as a versioned object.

The framework also subsumes other cases. A mathematical proof is a sequence of transformations of an implicit state (the set of established propositions), each step proposing a new derivation (a transformation) that must be admissible under the rules of inference (the constraint set). A legal argument is a sequence of proposed reframings of the relevant facts and precedents, evaluated against the constraint set of the applicable law. A scientific experiment is a proposed transformation of the community's shared model of the world, evaluated against the constraint set of experimental reproducibility and theoretical coherence. In each case, the structure is the same: transformation, constraint, closure.

What distinguishes these regimes from each other is the degree of explicitness of the state, the transformations, and the constraints. Mathematical proof is highly explicit

in all three; legal argument is somewhat explicit; scientific practice is variable; LLM dialogue is almost entirely implicit. GitHub sits in a middle position: explicit enough to provide substantial scaffolding, but not explicit enough to enforce semantic admissibility automatically. The gradient from implicit to explicit is, roughly, the gradient from informal to formal, and the efficiency gains that formal systems provide—precision, reproducibility, verifiability—are precisely the efficiency gains that transformational communication provides over message-passing communication.

Limitations and Incompleteness of Current Systems

The framework we have proposed is an idealization, and the existing systems we have discussed are only partial realizations of it. It is important to be clear about the limitations, because they reveal where the significant work remains.

GitHub's limitations are primarily representational. The system operates on textual representations and enforces only a minimal constraint set—build success, test passage, and whatever social norms the community maintains. It permits the acceptance of semantically invalid states, either because the constraint set is incomplete (the tests do not cover the relevant behavior) or because the social evaluation misses a subtle implication of the proposed transformation. The system also lacks any explicit representation of semantic state: the repository's current meaning must be inferred by reading the code, and this inference is performed by human readers rather than by the system itself. There is no object within GitHub that represents what the repository means, as distinct from what it contains.

LLMs' limitations are more fundamental. They lack explicit structure at every level. The state is not represented as an object; it is distributed across the model's activations and the conversation history in a way that is not directly accessible or manipulable. The transformations are not composed from primitives; each prompt is a global perturbation of the implicit state, and the model has no mechanism for ensuring that this perturbation is minimal or decomposable. The constraint set is not enforced explicitly; it is expressed only in the model's tendency to produce certain kinds of outputs, and there is no mechanism for detecting constraint violation before a response is generated. These are not engineering

oversights but reflect fundamental choices in the architecture of current language models: they are trained to predict text, not to maintain and update explicit semantic states.

Both limitations point in the same direction: toward systems with richer explicit representations of semantic state, richer explicit representations of semantic transformation, and stronger formal mechanisms of constraint enforcement. This is a research agenda, not a product specification, and it is a genuinely difficult one. But the framework makes clear what such systems would be trying to achieve, and why.

Historical Roots: Git, the Patch Lineage, and the Accidental Fixing of Email

---

Understanding why GitHub feels like a solution to email's failures requires a brief historical digression. Git was designed by Linus Torvalds in 2005 not to replace email but to coordinate Linux kernel development without a central authority. Yet its prehistory is deeply entangled with email. Before Git, large collaborative projects were coordinated through email patches—literally, diffs sent over mailing lists, where a message was a change, a thread was a sequence of revisions, and acceptance meant applying the patch to one's local tree. Git formalized and scaled this practice. GitHub then layered a social interface on top of Git, introducing pull requests, issue tracking, and inline review as structured containers for the discourse that email had previously carried in free-form.

The result is that GitHub did not so much replace email as constrain and externalize what email was already attempting to do. Email threads about software changes were implicitly trying to coordinate transformations of a shared object; they merely lacked the machinery to make that object explicit and the transformations composable. GitHub provided that machinery. The shift was not one of intention but of structure: the same communicative impulse that drove email-based patch coordination found a more adequate medium.

This lineage is relevant because it situates the present analysis within a longer arc. Critics of email such as Cal Newport and Jason Fried correctly identified the problem—that email conflates task tracking, discussion, and state updates into a single linear stream—and proposed structured tools as the remedy. But their account stops at task management; it does not reach the deeper claim that the unit of communication should be a minimal state

transformation rather than a narrative message. The patch-theoretic lineage of kernel development comes much closer. Kernel developers still think of their discipline in exactly these terms: a contribution is a delta, a thread is a composition of deltas, and acceptance is integration under constraint. GitHub made this discipline accessible and visible to a vastly wider community.

From an academic standpoint, researchers in software engineering have recognized the relevant phenomena under the name of semantic version control. Tools such as GumTree and SemanticMerge attempt to operate at the level of abstract syntax trees rather than raw text, tracking moves, renames, and refactorings as first-class operations rather than line-level coincidences. The research literature explicitly distinguishes textual conflicts, syntactic conflicts, and semantic conflicts—acknowledging that a clean textual merge can still produce incorrect behavior, because the objects of interest are not files but meanings. These efforts provide partial implementations of the ideal described in this essay, though they remain confined to syntactic structure rather than full semantic equivalence. The present argument can therefore be read as a theoretical synthesis of this engineering literature: a statement of what semantic version control would have to be if it were carried to its logical conclusion.

Recency as Address: The Temporal Structure of Collaborative Attention

---

A key feature of transformational communication systems that has not yet been analyzed is their addressing mechanism. In message-passing systems such as email, addressing is explicit and symbolic: the sender chooses a recipient, composes a subject, and the message is routed by identity. The relevance of the message to its recipient is a matter of human judgment, exercised after the fact. In transformational communication systems, by contrast, addressing is delegated to structure and, crucially, to recency.

GitHub does not simply expose all changes uniformly. Its interface organizes visibility around recent activity: recent commits appear prominently, open pull requests are sorted by update time, notification feeds are ordered by temporal proximity. The effective address of a communication is therefore not a person but a location in a dynamically

evolving state weighted by how recently it was modified. To communicate with another developer through GitHub is not to send them a message; it is to attach a transformation to a shared object in a way that is temporally visible within their active attention field. One can, in principle, communicate with anyone in the world instantly by proposing a change to their project or opening an issue on their repository—not because the channel is fast, but because the reference is unambiguous and the proposal is immediately legible within the structure they have built.

This recency-weighted visibility is not incidental. It functions as a probabilistic proxy for relevance. Recent changes are assumed to be more important than older ones because active projects are in motion: the questions being asked, the constraints being enforced, and the goals being pursued are all concentrated in the region of recent activity. Proposals that arrive within this region are more likely to be evaluated promptly; proposals that fall outside it—raising issues on dormant repositories, commenting on old commits—are less likely to receive attention, not because the system routes them away, but because they fall outside the recency-weighted attention field.

The same structure governs large language model interaction, though in a more extreme form. The context window of a language model is entirely dominated by recency: each token attends primarily to nearby tokens, recent exchanges shape the current response more strongly than earlier ones, and the beginning of a long conversation exerts progressively less influence as new content accumulates. The model does not have an address book; it has a context window, and relevance is determined by position within that window. A prompt issued early in a conversation is, in a precise sense, farther away than a prompt issued moments ago. Recency is address.

This substitution of temporal proximity for symbolic identity has significant practical consequences. Systems organized around recency are responsive and self-updating: they automatically focus attention on what is currently active without requiring the user to manage routing manually. But they also introduce characteristic failure modes. Information that is not recently active becomes effectively invisible, regardless of its importance. Long-lived issues on GitHub may simply stop appearing in active feeds. Important constraints established early in a conversation may be overridden by more recent context. In both cases, the system's recency bias produces a form of structural forgetting: older structure is not deleted, but it ceases to exert influence proportional to its semantic weight.

The Notification Index and the Temporal Cost of Semantic Integration

Given the recency-weighted structure of collaborative attention, it becomes possible to define a measurable quantity that characterizes the dynamics of semantic proposal and acceptance. Call this the *notification index* of a proposed transformation: the elapsed time between the moment a change is proposed and the moment it is accepted, integrated, or definitively rejected. In the GitHub setting, this corresponds to the time between submitting a pull request and receiving a merge or close event. In the LLM setting, it corresponds to the number of conversational turns required for the system to stabilize on an interpretation consistent with the user's intent. In both cases, the notification index measures how long a semantic proposal remains in an unresolved state.

The notification index is not a simple scalar. It is a composite quantity reflecting several distinct sources of delay. The first is attention latency: the time before the proposal enters the active consideration of any evaluating agent. In GitHub, this corresponds to the period before a maintainer notices the pull request. In LLM interaction, attention latency is nearly zero—the context window processes all inputs immediately—but an analogous quantity exists in the time before the model produces a response aligned with the user's actual intent rather than a plausible but misaligned continuation.

The second component is processing latency: the time required to render the proposal into an interpretable form. A pull request that modifies a large number of files, or that introduces a novel architectural pattern, requires more time to understand than a single-line fix to an isolated function. A prompt that attaches a lengthy document, or that requests the generation of a complex multi-part artifact, introduces processing overhead that delays effective evaluation. When a user requests a data export from a platform, the notification index is dominated entirely by processing latency: the system must gather, compile, and package a large quantity of structured information, and the user's role is to grant permission for this process and await a deferred notification. This converts a synchronous interaction into an asynchronous one governed by event completion rather than attentional recency—the user is effectively giving permission to be emailed later, outside the active field of real-time exchange.

The third component is semantic integration latency: the time required to determine

whether the proposal is admissible under the system's constraint set and, if so, to incorporate it into the shared state without producing incompatibilities. This is the component most directly tied to the theoretical framework developed in this essay. A well-formed, minimal semantic transformation should have low integration latency: it introduces no ambiguity, conflicts with no existing constraint, and requires no negotiation. A poorly formed or semantically entangled proposal may require extensive revision, discussion, or redesign before it reaches a form that can be cleanly integrated.

It is important to resist the temptation to treat the notification index as a pure measure of semantic quality. A long delay may indicate that a proposal is semantically distant from the current admissible state, but it may equally indicate that the evaluating agent is simply occupied, or that the proposal fell out of the recency-weighted attention field before being evaluated, or that operational overhead dominates for reasons entirely independent of the proposal's content. The notification index is therefore a composite observable that mixes semantic, attentional, and operational signals. Disentangling these components requires additional contextual information, and the composite nature of the measure is itself informative about the architecture of the system being studied.

The Extended Stack: Synthesis, Entropy, and Rapport

The preceding analysis of transformational communication can be extended by recognizing three additional layers that govern whether proposed transformations can be successfully integrated into a shared state: theoretical synthesis, entropy budget, and rapport. These layers do not replace the earlier framework but complete it, transforming a descriptive account of interaction into a dynamical theory of semantic stabilization.

Theoretical synthesis concerns the capacity of a system to reconcile multiple transformations into a single coherent world-state. A version control system, or any collaborative environment, does not merely check whether individual changes are locally valid; it must determine whether they can coexist globally. Two commits may each pass all local constraints and yet remain mutually incompatible when composed. In conventional systems, such incompatibility appears as a merge conflict. In a more general setting, it manifests as a failure to construct a consistent global section from locally admissible

patches—what in sheaf-theoretic terms would be called a failure of the gluing condition. Theoretical synthesis is the process by which such failures are addressed: not simply by rejecting one of the conflicting proposals, but by attempting to reinterpret, refactor, or lift them into a higher-order structure in which compatibility can be restored. This is the layer at which systems evolve their representational capacity, expanding the space of admissible configurations rather than merely selecting among existing ones.

The second layer, entropy budget, introduces a resource constraint on this process. Every proposed transformation carries a certain degree of semantic entropy: ambiguity, degrees of freedom, and unresolved structure that must be resolved before the transformation can be cleanly incorporated. Systems cannot absorb arbitrarily high entropy without destabilizing. In practice, this limitation appears as cognitive load in human collaborators, computational limits in automated systems, or combinatorial explosion in formal reconstruction. A minimal semantic change is therefore not only one that isolates a single conceptual effect, but one that does so with minimal entropy. Large, underspecified, or highly entangled proposals consume disproportionate portions of the system's entropy budget, increasing both processing latency and the likelihood of rejection or deferral. Conversely, well-structured transformations conserve entropy, allowing the system to maintain coherence while continuing to evolve. The discipline of semantic minimality is not merely aesthetic; it is a strategy for operating within the entropy constraints of the collaborative medium.

The third layer, rapport, governs how entropy is interpreted and managed across agents. Rapport can be understood as a shared compression scheme: a set of aligned priors, conventions, and expectations that allow participants to encode and decode semantic transformations efficiently. In high-rapport settings, a transformation can be specified with relatively little explicit detail because much of its structure is implicitly understood. The effective entropy of the proposal is therefore lower than its raw description would suggest. In low-rapport settings, even simple transformations must be elaborated extensively to avoid misinterpretation, increasing both entropy load and integration cost. Rapport acts as a multiplier on the entropy budget, determining how much complexity can be handled without loss of coherence. It is not merely a social amenity but a structural parameter of the system, directly influencing the feasibility of semantic integration.

These three layers interact in nontrivial ways. A system with high rapport can process higher-entropy proposals because shared context enables efficient compression. Strong

theoretical synthesis can reduce effective entropy by reorganizing disparate transformations into a unified structure. Conversely, low rapport and weak synthesis capabilities force the system to operate under a much tighter entropy budget, leading to increased latency, frequent conflicts, and fragmentation of the shared state. The observable dynamics of collaboration—such as the time required to merge a change, the frequency of conflicts, or the stability of a conversation over many turns—are projections of this deeper interplay. Reconsidered in this light, the notification index acquires a fuller decomposition: delay reflects not only attention and processing overhead, but also the entropy of the proposal relative to system capacity, the rapport available to compress it, and the synthesis effort required to situate it within the evolving shared structure.

Nested Clipboards and the Compositionality of Interaction

The framework developed in this essay has focused on large-scale communicative events: commits, pull requests, prompts, and merges. But the same compositional structure is present at a much finer grain, in the microinteractions through which users construct, store, and reapply transformations in the course of everyday work. In particular, the mechanisms of nested clipboards, command history, and terminal multiplexers reveal an underlying layer of semantic macro composition that deserves explicit attention.

A standard clipboard already functions as a primitive macro: it allows a user to extract a fragment of structure and reinsert it elsewhere. Yet most interfaces treat the clipboard as a single-slot buffer, discarding most of its generative potential. Once clipboards become nested—when multiple fragments can be stored simultaneously in an organized hierarchy—they begin to resemble a stack of parameterized transformations. Each clipboard entry is no longer merely a piece of text but a reusable deformation that can be invoked within different contexts. The act of copying becomes an act of abstraction, and pasting becomes an act of reapplication. The user is no longer merely moving content; they are composing and deploying stored operations.

This perspective aligns closely with the behavior of shell environments. In systems such as Bash, the command history is not simply a log of past inputs; it is a repository of executable transformations, each of which can be recalled, modified, and reissued. A command

retrieved from history can be adjusted and applied in a new context, functioning as a template for a new operation. The history is therefore a lightweight version of a commit log: a record of past transformations that can be replayed, composed, or refined. Terminal multiplexers such as `byobu` and `tmux` add a further dimension by maintaining multiple concurrent sessions, each with its own local history and state. Switching between windows is not merely a change of view but a traversal across parallel strands of transformation sequences, each of which can be resumed, forked, or recombined. The user navigates not just space but a structured history of operations.

The equivalence between nested clipboards, command history, and multiplexed sessions becomes clear when viewed through the lens of semantic versioning. Each stored item—whether a clipboard fragment, a history entry, or a suspended session—is a delta: a transformation that has been previously applied or is available for future application. The user implicitly manages a small personal repository of such transformations, curating the ones that are worth keeping and discarding those that are not. When a developer maintains a set of frequently reused snippets, or a writer builds up a library of prose templates, or a researcher stores a sequence of analytical commands, they are constructing a personal vocabulary of semantic operations.

What distinguishes these local mechanisms from full semantic version control is the absence of explicit constraint enforcement. Clipboard macros and shell histories allow arbitrary recombination but provide no guarantee that the resulting state is admissible under any global consistency condition. They are expressive but weakly regulated. Nonetheless, they reveal something important: users naturally and persistently gravitate toward reusable, composable transformations even in the absence of formal support. This behavior is not an accident of interface design but reflects a genuine structural feature of the tasks being performed. Meaningful work in complex domains proceeds through the accumulation, refinement, and recomposition of operations—not through the sequential creation of content from nothing.

Within the framework of this essay, nested clipboards and command histories can be interpreted as mechanisms for managing semantic entropy at the level of individual interaction. By storing and reusing previously validated transformations, the user avoids reconstructing complex operations from scratch, thereby conserving entropy budget and reducing integration cost. In collaborative settings, shared snippet libraries, project-specific aliases, and documented command workflows function as distributed clipboards:

compressed representations of transformations that any participant can expand and apply with minimal ambiguity. This is rapport materialized at the level of tooling—a shared vocabulary of operations that reduces the effective entropy of each individual contribution.

The integration of such mechanisms into a fully realized semantic version control system suggests a natural direction for future development. Rather than treating commits, prompts, and interactions as isolated events, such a system would expose a hierarchy of reusable transformations, allowing users to compose higher-level operations from previously validated components. The boundary between editing, scripting, and version control would dissolve, replaced by a unified interface for constructing and deploying semantic macros at multiple levels of granularity. In this vision, the everyday act of copying a fragment or recalling a command is not a convenience feature but a primitive instance of a deeper principle: that meaningful interaction proceeds through the structured reuse and recombination of semantic operations, and that the systems best suited to human collaboration are those that make this structure visible, composable, and conserving of entropy.

Conclusion: From Messages to Worlds

We began with a puzzle: why does GitHub feel more precise than email? We end with a general answer. GitHub and email are not doing the same thing. Email is message-passing communication: participants produce assertions about the world, accumulate a shared record of those assertions, and coordinate by trying to maintain consistent models of the current situation in their individual minds. GitHub is transformational communication: participants interact by proposing, evaluating, and accepting or rejecting structured changes to a shared object, and the current state of the shared object is the externalized, verifiable product of that process.

The difference is not one of degree but of kind. Message-passing communication places the burden of world-modeling on the participants: each must maintain their own representation of the current situation and update it as messages arrive. Transformational communication externalizes this burden: the shared object is the world, and its current state is always available for inspection. The cost of this externalization is that communi-

cation becomes more demanding: you cannot say something without changing something, or proposing to change something. But this cost is also a benefit: it forces precision, eliminates a class of ambiguities, and makes the record of reasoning transparent and inspectable.

We have seen that LLM interaction already has the structure of transformational communication, but in an underconstrained and implicit register. The trajectory-through-semantic-space picture is not a metaphor; it is a description of what is happening in any coherent extended conversation with a language model. The user is navigating a semantic manifold, applying perturbations (prompts), evaluating admissibility (response quality), and maintaining the trajectory in a coherent region (through iterative refinement). What the system lacks is the scaffolding that would make this navigation explicit and supported.

A fully realized system of transformational communication would operate directly on semantic structures. Its states would be explicit representations of meaning, not textual encodings of meaning. Its transformations would be semantic morphisms, verifiably minimal and composable. Its constraint enforcement would be formal and automatic, rather than social and post-hoc. In such a system, meaning would not be inferred from text but enacted through the structured evolution of a shared reality. Communication would be the collaborative maintenance and development of a world.

This is, admittedly, a distant prospect. But it is not a speculative one. The tools we already have—version control, formal verification, type systems, language models—are all partial realizations of pieces of this picture. What is needed is the theoretical framework that makes the picture coherent, so that progress can be made consciously rather than accidentally. This essay is an attempt to contribute to that framework. The central thesis is simple: communication is evolving from the transmission of messages to the collaborative transformation of shared worlds, and the systems we build to support this evolution should be designed with that destination in mind.

Note on TARTAN. The framework developed here has a direct connection to the TARTAN programme. GitHub, interpreted through the lens of this essay, is a primitive TARTAN-like system: truth is operationally defined as whatever survives merge under consistency

constraints, the history encodes not just accepted states but the full record of proposed and rejected transformations, and the constraint closure process is a formal analog of the iterative tightening of admissibility conditions that TARTAN employs in its more general setting. The connection is not merely analogical. A more fully realized system of transformational communication would need precisely the kind of explicit semantic state representation and compositional constraint enforcement that TARTAN formalizes. The line of thought is surprisingly fertile.

References

[1] Linus Torvalds. *Git: Fast Version Control System*. 2005. https://git-scm.com

[2] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2nd Edition, 2014.

[3] Jean-Rémy Falleri et al. *Fine-grained and Accurate Source Code Differencing*. Proceedings of ASE, 2014.

[4] Tom Mens. *A State-of-the-Art Survey on Software Merging*. IEEE Transactions on Software Engineering, 28(5), 2002.

[5] James J. Hunt and M. Douglas McIlroy. *An Algorithm for Differential File Comparison*. Bell Laboratories, 1976.

[6] Peter Naur. *Programming as Theory Building*. Microprocessing and Microprogramming, 15(5), 1985.

[7] Karl Fogel. *Producing Open Source Software*. O'Reilly Media, 2005.

[8] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[9] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[10] Masaki Kashiwara and Pierre Schapira. *Sheaves on Manifolds*. Springer, 1990.

[11] David I. Spivak. *Category Theory for the Sciences*. MIT Press, 2014.

[12] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] Ashish Vaswani et al. *Attention Is All You Need*. NeurIPS, 2017.

[14] Tom B. Brown et al. *Language Models are Few-Shot Learners*. NeurIPS, 2020.

[15] Karl Friston. *The Free-Energy Principle: A Unified Brain Theory?* Nature Reviews Neuroscience, 2010.

[16] Claude E. Shannon. *A Mathematical Theory of Communication*. Bell System Technical Journal, 1948.

[17] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley, 2006.

[18] Douglas Engelbart. *Augmenting Human Intellect: A Conceptual Framework*. 1962.

[19] Niklas Luhmann. *Social Systems*. Stanford University Press, 1995.

[20] Bruno Latour. *Reassembling the Social*. Oxford University Press, 2005.