Flow Computing

Wavefronts, Invocation, and the Logically Determined Workflow

Flyxion

June 2026

> *Symbols move from function to function, instantiating in the proper order as their input symbols are available. Wavefronts flow from expression to expression and instantiate in the proper order when their inputs are complete. This coordination behavior is explicit in the logic itself. The expressivity of the absent mathematician has been fully restored in the expressivity of the logic.*
>
> Karl M. Fant, *Logically Determined Design*, 2005

Abstract

Karl Fant's two books—*Logically Determined Design* (2005) and *Computer Science Reconsidered* (2007)—pursue a single philosophical project from two directions. The first begins in hardware: a Boolean logic expression, Fant shows, is expressionally incomplete without the coordinating mathematician who interprets it. He proposes *Null Convention Logic*, in which completeness relationships encoded in the logic itself restore the missing coordination so that wavefronts flow spontaneously from expression to expression without any external scheduler. The second generalizes this discovery: computer science has inherited the wrong foundational concept (the algorithm) from mathematics and needs instead a theory of process expression—a theory of how processes can be expressed so as to coordinate their own realization. Together, these books define what this essay calls *flow computing*: a paradigm in which computation is organized not as state transitions managed by external controllers but as the propagation of process invocations through networks governed by completion relationships. Flow computing is emphatically not merely computation over streams of data—

that description captures only the medium, not the organizational principle. The principle is *invocation through completion*: a process becomes active because its prerequisites are complete, not because a scheduler commanded it. We argue that Unix pipelines are a practical software-level approximation of the logically determined workflow Fant describes— sharing its coordination principles while remaining subject to the imprecisions of real processes (buffering, failure, side effects). We further argue that a mature personal computing workflow—spanning AutoHotkey, WSL, Ubuntu, Bash, Vim, Python, Whisper, Ollama, Pandoc, and Git—constitutes a *process ecology* in Fant's sense, and that the application-centered desktop is, structurally, the mathematician's pencil: an external coordinator whose elimination is not a regression to arcane complexity but a recovery of the expressional completeness that the logic was always capable of providing.

Contents

1 The Missing Mathematician **3**

2 From State Transitions to Wavefronts **4**
- 2.1 The State Metaphor's Dependency on External Control 5
- 2.2 The Wavefront as the Correct Primitive 5
- 2.3 Abstracting Away the Electronics 6

3 Flow Computing: Invocation Through Completion **6**
- 3.1 From State Machines to Process Networks 7
- 3.2 Invocation as Primitive 7
- 3.3 Completion Relations as the Coordination Primitive 8

4 Unix Pipelines as Logically Determined Workflows **9**
- 4.1 The Pipe as Completion-Governed Coupling 9
- 4.2 Text as the NULL Convention of Software 10
- 4.3 The Process Network as a Tapestry of Completion Neighborhoods 10

5 Two Coordination Architectures **11**
- 5.1 The GUI User as Coordinator 11
- 5.2 The Four Costs of Localized Coordination 12
- 5.3 The Clock Analogy 12

6 A Modern Process Ecology **13**
6.1 The Stack as a Network of Completion Relationships 13
6.2 AutoHotkey: Ambient Invocation 13
6.3 The Principle of Earned Automation 14
6.4 From Repetition to Naming 15
6.5 AutoHotkey as a Fossil Record of Process Discovery 15
6.6 Why Most Automation Fails 16
6.7 Vim Macros as Deferred Process Crystallization 17
6.8 Tab Completion as Ecological Sensing 18
6.9 Bash: The Completion-Governed Composition Language 20
6.10 The Full Knowledge Wavefront 21
6.11 Persistence as Wavefront Capture 21
6.12 Whisper and Ollama as Cognitive Operator Boundaries 22

7 Personal Languages as Logically Determined Expression Documents **23**

8 Computational Literacy as Completion Fluency **24**

9 Modal Computation and Nested Scope: Vim, Byobu, and Spherepop **25**
9.1 Vim as Modal Computation 25
9.2 Operators and Scopes as Compositional Grammar 26
9.3 Vim's Shell Integration as Pipeline Stage 26
9.4 Byobu as Hierarchical Process Space 27
9.5 Byobu Keybindings as Scope Navigation 28
9.6 Development as Wavefront Refinement 29
9.7 Nested Scopes and Spherepop 30
9.8 Scope Before Process 31
9.9 The Recurrence of Nested Completion Structures 32
9.10 The Residue Hierarchy 33

10 Feedback, Residue, and Second-Order Flow **34**
10.1 Feedback as Completion-Governed Wavefronts 34
10.2 Residue: The Memory of Completed Wavefronts 35
10.3 Two Regimes of Flow Computing 35
10.4 A Note on Cloud Orchestration 36
10.5 Stochastic Wavefronts and Semantic Admissibility 37
10.6 The NULL Wavefront as Markov Boundary 39

11 Toward a Theory of Flow Computing **40**
11.1 The Fundamental Unit 41
11.2 The Coordination Principle 41
11.3 The Expression Principles 41
11.4 The Computer as Medium 42

12 Flow Computing and the Return of Process Literacy **43**
12.1 The Adversarial Complement: Environments Designed to Prevent Process Literacy 44

The Missing Mathematician

Boolean logic is a mathematical symbol system. There is a population of symbols organized into a state space, a set of primitive function mappings, and a logic expression specifying a progression of those mappings. This symbol system is, in Karl Fant's precise phrase, *enlivened by an active agent*—traditionally a mathematician with a pencil—who understands the rules of coordination and applies them: moving symbols from function to function, instantiating functions in the proper order as their input symbols become available.

The mathematician is not part of the logic expression. The coordination behavior is embodied in her understanding, not explicit in the formalism. On its own expressional merits—without the mathematician—a Boolean logic expression is an incomplete expression that cannot be trusted.

This is the opening problem of Fant's *Logically Determined Design* (2005), a book nominally about asynchronous circuit design. But the problem Fant poses is not primarily an engineering problem. It is a philosophical problem about *expressional completeness*:

> How do we restore the missing coordination behavior so that a symbolic expression can coordinate its own realization—without any external interpreter, without any clock, without any scheduler?

Fant's answer is Null Convention Logic (NCL): a logic in which each function is enhanced with awareness of the difference between a data value and a NULL value (the absence of data). When an NCL function receives a complete data wavefront—all its inputs have transitioned from NULL to data—it computes and emits its result. When it subsequently receives a NULL wavefront, it clears. The key property: this behavior is determined entirely by the logical relationships among the functions. No clock coordinates the clearing. No scheduler decides when the next wavefront may enter. The logic coordinates itself.

Wavefronts flow from expression to expression and instantiate in the proper order when their inputs are complete. The coordination behavior is explicit in the logic itself. The mathematician's pencil has been made unnecessary not by adding a better interpreter but by encoding the coordination into the expression.

Fant calls the result a *logically determined system*: a structure of logical relationships whose behavior is completely and unambiguously determined by those re-

lationships. It is not managed by an external timeline. It is not sampled at clock edges. Its collective state at any given instant may be indeterminate—multiple transitions occurring concurrently—but its logical behavior is fully trustworthy, understood expression by expression, neighborhood by neighborhood, as a tapestry woven of overlapping behavior boundaries.

Two years later, *Computer Science Reconsidered* (2007) extends this insight to its full generality. The missing mathematician is not only a problem in hardware design. It is a problem in computer science at large: the discipline inherited the algorithm from mathematics—a formalism built for the mathematician's question (*what can be computed?*)—and mistook it for the foundational concept for its own question (*how can processes be expressed?*). The algorithm, like a Boolean expression without coordination, requires an external controller: a sequencer that instantiates and uninstantiates each operation in turn. Sequential expression cannot spontaneously flow. It requires a mathematician. The restoration of the missing coordination behavior at the level of computing-in-general is what Fant calls the Invocation Model: processes that invoke themselves when their prerequisites form a resolvable name.

This essay takes Fant's dual project—the hardware argument of LDD and the theoretical argument of CSR—as the foundation for a claim about personal computing: that the application-centered desktop is, structurally, the mathematician's pencil, and that Unix pipelines are the closest thing personal computing has produced to a logically determined system at the scale of knowledge work.

From State Transitions to Wavefronts

The dominant metaphor of computing, inherited from the Turing machine and solidified through decades of software engineering, is *state transition*. A program is a procedure that advances a machine through a sequence of states. The machine is always in exactly one state. Transitions are atomic. The sequence is the computation.

This metaphor carries hidden assumptions that Fant's work makes visible.

The State Metaphor's Dependency on External Control

A state machine requires external control to transition. Something must read the current state, apply the transition function, and write the new state. In a clocked digital system, the clock is that external controller: at each edge, all registers sample their inputs simultaneously, establishing a new collective state. The computation advances because the clock says so.

The computational structure of the process—the partial ordering of data dependencies among operations—is real, but the clock does not respect it. The clock ticks uniformly regardless of whether any particular operation's prerequisites are actually satisfied. The programmer must design the schedule so that prerequisites are always satisfied before the clock tick that requires them. This is a correspondence problem that the programmer must maintain manually: the inherent concurrent structure of the process on one hand, and the imposed sequential schedule on the other.

Fant shows that this correspondence problem is not a minor engineering challenge. It is the fundamental cost of sequentiality: the cost of forcing a naturally concurrent, partially ordered structure through a sequential bottleneck managed by an external agent.

The Wavefront as the Correct Primitive

Fant's alternative primitive is the *wavefront*: a monotonic transition of an expression from a completely NULL state to a completely data state (a data wavefront) or from completely data to completely NULL (a NULL wavefront). Successive data and NULL wavefronts alternate, each separated by the completeness event of the preceding wavefront.

The wavefront is not the same as a state. A state is a snapshot: it describes where the machine is at a moment in time. A wavefront is a process: it describes the propagation of a transition through a network of logical relationships. States are managed by external controllers. Wavefronts are *governed by completeness relationships*.

This distinction is the hinge on which the entire argument turns. When computation is organized around wavefronts rather than states:

1. **Coordination is intrinsic.** The wavefront does not advance to the next ex-

pression until the current expression signals completion. The completion signal is generated by the logic itself, not by an external timer.

2. **Concurrency is primitive.** Many wavefronts can propagate through different parts of the network simultaneously without interference, because each wavefront is governed by local completion relationships rather than a global clock.

3. **The process expression is self-sufficient.** No mathematician, no scheduler, no clock is needed to coordinate the flow. The logic contains the coordination.

Abstracting Away the Electronics

NCL is presented in the context of asynchronous electronic circuits. But the insight abstracts cleanly away from the electronics. What matters is not the medium—wire voltages, not-data signals, dual-rail encoding—but the organizational principle:

> A process should become active when its prerequisites are complete. A process should signal completion when its outputs are stable and correct. A process should release its inputs when its downstream neighbor has accepted its output.

These three rules, which are exactly what NCL instantiates in logic, define a coordination protocol that applies to any network of processes at any level of abstraction. They define flow computing.

Flow Computing: Invocation Through Completion

Flow computing is the paradigm that applies Fant's organizational principle to computation at large. It can be stated simply:

> A computational system is a network of potential processes connected by invocation relationships governed by completion. A process is invoked when its prerequisites are complete. A process signals completion to its downstream neighbors when its own output is stable and correct.

This is different from the stream-processing view of Unix, which emphasizes data flow. It is also different from the functional programming view, which emphasizes referential transparency. Flow computing is specifically about *how processes know when to become active*—and the answer is: through completion relationships encoded in the structure of the process network itself, not through any external scheduler.

From State Machines to Process Networks

Traditional computer science education presents computation as a sequence of state transitions. A program begins in an initial state, executes instructions, and arrives at a final state. Whether expressed through Turing machines, finite automata, or imperative programming languages, the dominant metaphor is state evolution.

Flow computing adopts a different perspective. The primary object is not the state but the process. Computation is a network of transformations through which information propagates. The central question becomes not *what state is the machine in?* but *which processes are currently active, and what conditions enable the next invocation?*

This shift resembles the difference between describing a river as a sequence of water locations and describing it as a flow system. The former is technically accurate; the latter captures the dynamics that actually matter.

Invocation as Primitive

Most software engineering treats function invocation as a secondary operation. A function exists. A programmer calls it. Execution begins. The programmer is the mathematician: the external coordinator who decides when each function should be instantiated.

Flow computing reverses this relationship. Invocation becomes the primitive concept. A computational system consists of latent processes connected by invocation relationships. A process exists in a latent state until its enabling conditions— its prerequisite completions— are satisfied. When they are, the process invokes itself. It does not wait for a command.

It is important to distinguish this from the familiar notion of stream processing

or functional dataflow. Stream processing emphasizes the *medium*: data passes through a chain of transformations. Flow computing emphasizes the *governance*: each process becomes active because its prerequisite completions have formed—not because data happened to arrive on a channel, not because a scheduler issued a dispatch, but because the logical condition for invocation is satisfied. The wavefront does not flow because someone pushed it. It flows because the logic determines that it should.

In Fant's hardware terms: the wavefront flows to the next expression when the current expression signals completion. In Unix terms: the downstream process receives input and begins processing as soon as the upstream process emits output. In knowledge workflow terms: Whisper begins when audio exists; Ollama begins when text exists; Pandoc begins when Markdown exists; Git deploys when output exists.

The flow determines the order. No external agent manages the schedule.

Completion Relations as the Coordination Primitive

The most important concept inherited from Fant's dual project is not asynchronous hardware design but the notion of *completion* as a coordination primitive.

A clocked system coordinates by time: all operations that share a clock domain advance at the same moment, regardless of whether their inputs are ready. A flow computing system coordinates by *completion*: a process advances when its inputs are ready, which is to say, when the processes that produce its inputs are complete.

The difference is profound. Time-based coordination is an external imposition on the process structure. Completion-based coordination is the process structure made explicit. In a logically determined system, the coordination behavior is not added to the expression from outside—it is *in the expression*. The system is self- coordinating.

Applied to software workflows, this means: a build step should run when its dependencies are built; a transcription should run when its audio is ready; a summary should be generated when its transcript is complete. These are not scheduling decisions. They are logical relationships. The workflow is logically determined when those relationships, rather than any external scheduler's decisions, govern what runs when.

Unix Pipelines as Logically Determined Workflows

Unix pipelines, developed at Bell Laboratories a decade before Fant's work on NCL, embody the flow computing paradigm at the scale of processes rather than logic gates. They arrived not through theoretical derivation but through the accumulated practical wisdom of Ken Thompson, Dennis Ritchie, and Doug McIlroy: the insight that programs should do one thing well, that they should communicate through streams, and that composition should be the primary act of the user's work.

Seen through Fant's lens, Unix pipelines instantiate a weak form of logical determination at the level of process composition. They are not identical to NCL: Unix processes can fail, buffer unpredictably, depend on side effects, or block in ways that NCL's formal completeness guarantees exclude. But they share the essential organizational principle—coordination through completion rather than through an external clock—and they realize it practically, at a level of abstraction where formal guarantees give way to the robust conventions of the Unix interface contract. The pipe is not a mathematically proven completion channel; it is a durable engineering approximation of one.

The Pipe as Completion-Governed Coupling

A Unix pipe is commonly described as a data transport mechanism. This description is accurate but insufficient. A pipe is also a *completion-governed coupling*: the downstream process receives data as the upstream process emits it and begins its transformation as soon as the first complete unit of input (a line, a record, a stream) arrives. When the upstream process finishes and closes its output, the downstream process receives an end-of-stream signal and performs its own completion sequence.

This is precisely the structure of Fant's coupled cycles. The completion detection of the presenting cycle is placed after the input regulation of the receiving cycle. The acknowledge signal says: the wavefront has been received; I can accept the next. In Unix, the pipe buffer is the input regulator; the end-of-stream signal is the completion acknowledgment; the downstream process is the receiving boundary.

No mathematician coordinates the pipeline. The coordination is in the structure.

Text as the NULL Convention of Software

Fant's NCL requires a universal data representation that can distinguish between a complete data value and the absence of data (NULL). Without this distinction, a logic function cannot determine whether all its inputs have arrived; it cannot form the completeness criterion.

Unix pipelines solve an analogous problem with text streams. Text is the universal representation that allows any program to couple with any other. End-of-stream is the NULL equivalent: it signals that the presenting boundary has completed and the receiving boundary can finalize its own computation.

This is not a coincidence of terminology. It reflects a deep structural parallel. Both NCL and Unix pipelines solve the mathematicianless enlivenment problem at their respective levels of abstraction: both encode the coordination information into the representation itself rather than requiring an external agent to supply it.

The Process Network as a Tapestry of Completion Neighborhoods

Fant describes trust in a logically determined system as built up progressively, logical expression by logical expression, as each expression's behavior neighborhood is understood. Confidence in system behavior is approached in humanly graspable steps.

A Unix pipeline has the same property. Understanding a pipeline does not require understanding a global state machine with an exponential number of states. It requires understanding each stage's behavior, its input interface, its output interface, and its completion event. The stage's neighborhood—the stages immediately upstream and downstream—is the unit of comprehension. The pipeline as a whole is understood by composing these neighborhoods.

This is why pipelines are debuggable in a way that monolithic applications are not. You can inspect the output of any intermediate stage. You can substitute a stage for a debugging variant. You can halt the pipeline after any stage and examine the wavefront that has propagated to that point. The pipeline is a tapestry of logically determined behavior, grasped neighborhood by neighborhood.

Two Coordination Architectures

The flow computing paradigm and the application-centered desktop are not merely different interface styles. They are different *coordination architectures*: different answers to the question of where the coordination behavior of a multi-step process lives.

In the application-centered architecture, coordination is *localized in the user*. The user decides when to open each application, when to invoke each operation, when to transfer results between applications. The process network is real—the partial ordering of data dependencies exists whether or not it is encoded anywhere—but it is not encoded in the system. It is held in the user's head and acted out step by step.

In the pipeline-centered architecture, coordination is *externalized into explicit process relations*. The completion of one stage invokes the next. The partial ordering is encoded as the structure of the pipeline itself. The user declares the process network once; the system executes it without further direction.

The difference is structural, not aesthetic. GUIs are not intrinsically incompatible with flow computing—a well-designed GUI could, in principle, expose process completion events as composable signals. The problem is not graphicality but *encapsulation*: the application model encloses its processes behind an interface that does not expose completion events, does not accept external input streams, and does not emit output streams that downstream processes can consume. The interface is a terminus, not a boundary in a network.

The GUI User as Coordinator

When you use a word processor to produce a document, you are the mathematician. You decide when to open the application. You decide when to invoke the spell-checker. You decide when to export to PDF. You decide when to attach the PDF to an email. The process—the actual transformation of a draft into a sent document—has a natural partial ordering of data dependencies. The application-centered workflow localizes that partial ordering in the user, who manages it manually, visit by visit, application by application.

The application is a passive symbolic expression enlivened by the user's activity. Without the user, nothing advances. The coordination behavior is not encoded

in the expression. It is supplied externally.

This is the software version of the mathematicianless enlivenment problem. The user is the mathematician. The applications are the Boolean expressions that require her pencil to do anything at all.

The Four Costs of Localized Coordination

Fant's analysis of the costs of sequentiality has a direct analogue at the application level. Each cost follows from the structural fact that coordination is localized in the user rather than encoded in the process network:

1. **Variety of correctness.** Just as there are many valid sequential orderings of a concurrent process, there are many valid orderings of application visits in a multi-step workflow. Getting the ordering wrong is possible, difficult to detect, and a source of errors. Nothing in the application model enforces the correct partial ordering. The user must maintain it.

2. **Context switching overhead.** Each application visit requires a context switch: locating the application, navigating to the relevant state, performing the operation, saving, exiting. This overhead is not part of the work. It is the cost of externalized coordination.

3. **Invisibility of intermediate representations.** In a pipeline, every intermediate wavefront is a named, inspectable representation. In an application workflow, intermediate states are internal and inaccessible to external routing or inspection.

4. **Non-composability.** Application processes cannot be composed without explicit integration support. The completion event of one application does not invoke the next. The process network must be reconstructed manually at each use.

The Clock Analogy

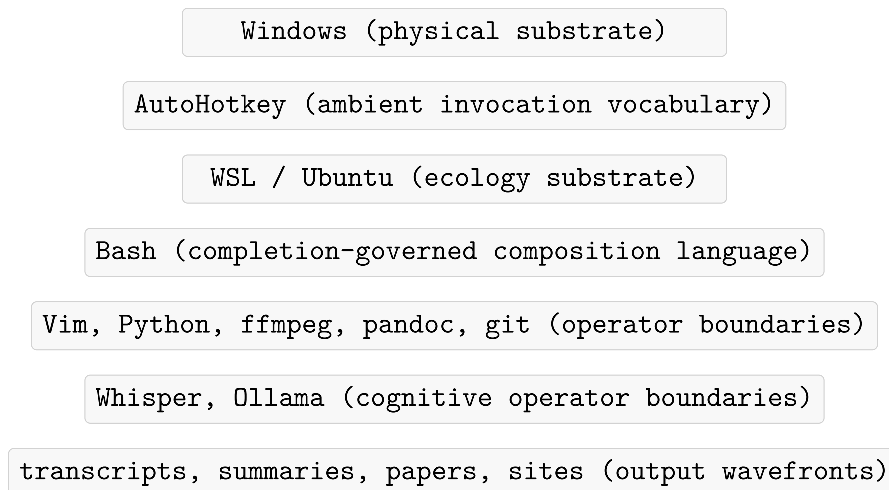
The operating system scheduler in an application-centered desktop plays the role of the clock in a clocked digital system. It allocates CPU time according to an external time-sharing schedule, not according to the completion relationships among the application processes.

In a flow computing workflow, the scheduler is largely irrelevant to workflow ordering. The pipeline coordinates itself through completion relationships. The OS scheduler assigns CPU time to whichever processes are currently ready to run—the role Fant assigns to the implementation substrate in a logically determined system. The coordination is in the logic. The substrate provides resources.

A Modern Process Ecology

The flow computing paradigm is not historical. A complete, working process ecology can be constructed today from freely available tools. The following describes one such ecology—not as a recommendation but as a concrete case study in logically determined knowledge work.

The Stack as a Network of Completion Relationships



Each layer is not a collection of applications to visit. It is a collection of behavior boundaries available for invocation. The ecology does not execute in sequence. It flows: each stage becomes active when its prerequisite completions arrive, signals its own completion, and passes the wavefront downstream.

AutoHotkey: Ambient Invocation

AutoHotkey is typically described as a keyboard macro tool. In a flow computing context it is an *ambient invocation vocabulary*: a layer of named completion-

governed process activations that can be triggered from any window state, without a terminal context switch.

Each hotkey in a flow-oriented AutoHotkey configuration is not a shortcut to a GUI destination. It is a *named completion trigger*: it activates a process network whose internal stages are governed by their own completion relationships. The hotkey fires the first wavefront; the downstream stages fire themselves.

The accumulated hotstring vocabulary is equally significant. Hotstrings are not abbreviations. They are *domain-specific invocation tokens*: compressed names for recurring process expressions in the user's specific knowledge domain. The vocabulary crystallizes through use, each entry marking a completion relationship that has proved stable and general enough to name.

Reading a mature AutoHotkey file is not reading a configuration document. It is reading a partial specification of a process ecology: a record of which invocation relationships the user has stabilized into named tokens.

The Principle of Earned Automation

A widespread misunderstanding of automation holds that any repetitive task should be automated as quickly as possible. This view treats automation as a primary goal and measures productivity by the density of scripts and shortcuts accumulated.

Flow computing suggests a different principle: *do not automate a process until its completion relationships have stabilized*.

A process whose structure is still being discovered cannot be meaningfully automated. Premature automation does not eliminate the missing mathematician; it merely hardens the confusion into software. The scripts break. The shortcuts become obsolete. The automation requires more coordination overhead than the manual process it replaced.

The purpose of performing a process manually is not inefficiency. It is *observation*. Repetition reveals the actual boundaries of the process: the representations that move between stages, the failure modes, the edge cases, and the completion conditions that genuinely govern successful execution. Only when these relationships become stable—when the process reliably produces the same completion event from the same prerequisites—does automation become appropriate.

Automation is therefore not the beginning of process understanding but its *consequence*. The ecology grows not by aggressively scripting every task but by waiting until each process has disclosed its own structure, and then naming it.

From Repetition to Naming

The transition from manual process to automated process occurs through a recognizable sequence: a workflow first exists as a sequence of manually executed actions; after sufficient repetition, the sequence becomes recognizable as a coherent unit with stable boundaries; the unit receives a name; the name becomes a shell function, a script, an alias, or an AutoHotkey invocation.

The process has been compressed into a symbol. But what makes the compression valid is not frequency of recurrence alone. It is that the completion relationships have been understood. The name is a referential expression—in Fant's sense—that maps to an autonomous process expression whose behavior is already known.

A concrete example: the sequence

```
yt-dlp "$URL" -x --audio-format mp3 -o audio.mp3
whisper audio.mp3 --output_format txt
ollama run mistral "Summarize:" < audio.txt > summary.txt
pandoc summary.txt -o summary.html
```

becomes, after sufficient manual repetition and structural understanding, a single named invocation: `overview`.

The name does not replace the process. It refers to a process expression whose internal completion relationships are already understood. Naming is *referential compression of earned understanding*.

AutoHotkey as a Fossil Record of Process Discovery

An AutoHotkey configuration file, viewed through the lens of earned automation, is a *fossil record of process stabilization*: a historical record of which processes, in the ecology's development, reached the threshold of stable completion relationships and received names.

Each hotkey marks a process that proved sufficiently recurrent, sufficiently well-

understood, and sufficiently stable to deserve a dedicated invocation. Each hotstring marks a concept, transformation, or expression pattern that recurred with sufficient frequency and stability to deserve compression.

The density of entries in a mature AutoHotkey file is not a measure of productivity obsession or shortcut accumulation. It is a measure of how many completion relationships have been discovered, observed, and stabilized over the history of the user's work. The file accumulates like geological strata: each layer marking a period in which certain process structures became understood.

A mature AutoHotkey file is therefore neither a configuration file nor a shortcut collection. It is a partial map of the process ecology's evolution—a record not of what the user does, but of what the user has *come to understand well enough to name*.

Why Most Automation Fails

Most automation efforts fail because they attempt to automate *tasks* rather than *processes*. Tasks are surface manifestations: the visible actions a person performs. Processes are underlying structures: the completion relationships that govern why those actions occur in a particular order and what representations move between them.

When the underlying process structure is still changing—when the completion relationships are not yet stable—automating the surface task produces a script that encodes the current confusion. The script breaks as the process evolves. The maintenance burden exceeds the automation benefit. The missing mathematician has not been eliminated; she has been replaced by a brittle script that requires constant repair.

The flow computing approach delays automation until the process has disclosed its structure. This produces fewer automations, but the automations that survive become durable operator boundaries: behavior boundaries that compose reliably with other boundaries, signal completion predictably, and participate in larger process networks without requiring continuous maintenance.

The goal is not maximum automation. The goal is *stable invocation*: a vocabulary of named process expressions whose completion relationships are understood, whose behavior is trustworthy, and whose composition produces process ecologies that coordinate themselves.

Vim Macros as Deferred Process Crystallization

Vim macros occupy a theoretically interesting position between manual execution and permanent automation. A macro records a sequence of transformations into a named register and allows that sequence to be replayed without yet elevating it to the status of a permanent named process expression.

The typical progression in a flow computing ecology is:

$$\text{manual execution} \rightarrow \text{macro} \rightarrow \text{script} \rightarrow \text{named invocation}$$

Recording a transformation sequence into register q:

```
qq         " begin recording into register q
...        " perform the transformation sequence
q          " stop recording
```

and then executing it at increasing scale:

```
@q         " execute once: observe the result
100@@      " execute 100 times: test at scale
10000@@    " execute 10000 times: commit to the transformation
```

is a laboratory for process discovery. The macro occupies the boundary between observation and automation: the transformation is sufficiently understood to be repeatable, but not yet stable enough to deserve referential compression into a permanent name.

In Fant's terms, a macro is a *provisional process expression*: an autonomous expression that has been transcribed but not yet promoted to referential status. It can be invoked repeatedly to test whether its completion relationships are genuinely stable—whether the same input consistently produces the same completion event—before the cost of permanent naming is incurred.

The macro therefore serves the principle of earned automation not by automating the discovered process but by providing a low-cost instrument for observing it at scale. The practitioner applies the macro a hundred times, or ten thousand times, and watches. If the completion relationships hold under repetition, the macro earns promotion. If they fail—if edge cases appear, if the transforma-

tion breaks on certain inputs—the provisional expression is revised before it is crystallized.

A Vim macro is therefore not merely a recorded keystroke sequence. It is a *temporary behavioral scope*: a bounded container for process structure that can be invoked repeatedly before being promoted to a textual process expression. The macro register (`q`, `a`, `b`, …) is the scope's name. The recorded content is its autonomous expression. The repeated invocation (`100@@`) is stress-testing under repeated wavefront propagation.

In this sense, macros occupy the same ecological niche as shell functions in Bash: both are provisional containers for process structure undergoing validation. The difference is form: the shell function is a textual definition; the macro is a recorded behavioral trace. Both are incomplete process expressions in Fant's sense—referential shorthands awaiting the moment of sufficient stability to deserve permanent names.

Tab Completion as Ecological Sensing

Tab completion is conventionally described as a typing convenience: the shell expands a partial filename or command name to its full form. Within a flow computing ecology it serves a structurally deeper role: it is a form of *environmental sensing*.

Before invoking a process that produces a representation, the flow-literate practitioner habitually asks whether that representation already exists:

```
overview<TAB>         # does overview.txt or overview.html exist?
transcript<TAB>       # has this audio already been transcribed?
summary<TAB>          # has this document already been summarized?
```

The tab completion mechanism reveals the current completion state of the ecology. Which wavefronts have already propagated? Which process boundaries have already been crossed? Which residues are already available for downstream invocation?

The same sensing function is performed by shell history recall (`Ctrl-R`: has this pipeline been run before, and with what arguments?), by file globbing (`ls *summary*`: what summaries currently exist in this scope?), and by AutoHotkey's own recognition of previously typed hotstrings (has this invocation token al-

ready been defined?).

These mechanisms share a common logic: *discover before produce*. A new wavefront should be generated only after the practitioner has determined that no adequate wavefront already exists. In a GUI environment, this determination is difficult because intermediate representations are hidden inside applications. In a flow computing ecology, the filesystem is a transparent archive of captured wavefronts, and the shell's completion and history mechanisms are instruments for consulting that archive before committing to new computation.

This is not merely a practical efficiency. It reflects a theoretical property of the ecology: residue is reusable. A captured wavefront does not expire. If `transcript.txt` exists and is complete, the transcription process need not run again. The completion event that produced it is already in the ecology's memory, available for any downstream invocation that requires it. Tab completion is the practitioner's primary instrument for reading that memory before writing to it.

The significance of a mature AutoHotkey configuration is not that it contains shortcuts. A desktop operating system already provides shortcuts. The significance is that a flow-oriented AutoHotkey vocabulary develops into a *personal invocation language*: a command language for process activation rather than a menu system for software navigation.

Many entries in such a vocabulary no longer correspond to application launches at all. They correspond to process expressions: *download-and-transcribe*, *summarize-corpus*, *convert-document*, *normalize-text*, *generate-overview*, *publish-site*. Each token is a referential expression for a larger completion-governed process network. When the user invokes it, they are not selecting a destination. They are naming a desired transformation and delegating its coordination to the structure of the process expression that the name encodes.

The resulting vocabulary has the structure of a language whose lexical items are not nouns but *verbs of process activation*. This is the precise inversion of the application model, in which the vocabulary consists of nouns—application names designating places to visit. In the invocation language, no place is visited. A transformation is named, and the wavefront is released.

This distinction matters beyond terminology. An application vocabulary grows by accumulating software. An invocation vocabulary grows by accumulating

understood processes. The first measures the user's access to tools. The second measures the user's comprehension of their work at the level of completion relationships. A user with a rich invocation language has done the cognitive work of understanding which transformations recur in their domain, what representations move between stages, and where the genuine completion boundaries lie. The language is an artifact of that understanding.

Bash: The Completion-Governed Composition Language

Bash is the composition language of the ecology. Its primary data type is the stream. Its primary operators encode completion relationships directly:

- `|` couples two behavior boundaries, passing the output wavefront of the first to the input of the second.
- `&&` invokes the second process only when the first signals success (exit code 0)—a logical completion condition.
- `||` invokes the second only when the first signals failure—the NULL wavefront case.
- > and < route wavefronts to and from persistent named representations (files).
- The `for` loop iterates a process boundary over a collection, each iteration governed by the same completion relationships.

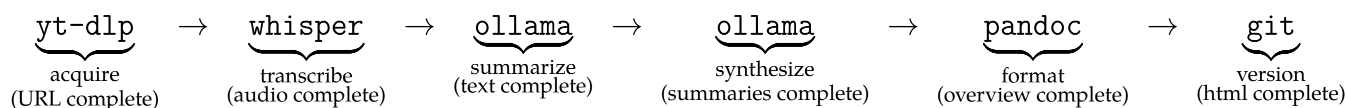
A knowledge-work pipeline in Bash:

```
for f in *.mp3; do
  base="${f%.mp3}"
  whisper "$f" --output_format txt -o "${base}.txt" \
    && ollama run mistral "Summarize concisely:" \
         < "${base}.txt" > "${base}_summary.txt"
done
cat *_summary.txt \
  | ollama run mistral "Synthesize these summaries:" \
  > overview.txt \
  && pandoc overview.txt -o overview.html \
  && git add overview.html && git commit -m "auto: overview"
```

Every && in this pipeline is a completion gate: the next stage fires only when the previous stage's output is complete and correct. The pipeline does not advance because a clock ticked. It advances because the logic of completion relationships determines that it should.

The Full Knowledge Wavefront

The complete knowledge pipeline traces a wavefront from raw source to published product through a succession of completion-governed invocations:



Each arrow is a completion event: the downstream stage is invoked not by a scheduler but by the logical fact that its prerequisites are satisfied. The intermediate representations—`.mp3`, `.txt`, `_summary.txt`, `overview.txt`, `.html`—are not products. They are wavefronts: intermediate persistence states that encode the completion of one boundary and the readiness condition for the next.

The product is not any of these files. The product is the appreciation of differentness—Fant's phrase for what process ultimately realizes—that the entire wavefront propagation has achieved.

Persistence as Wavefront Capture

In a flow computing ecology, files are not primarily storage objects. They are *captured wavefronts*.

An audio file records the completion of acquisition. A transcript records the completion of transcription. A summary records the completion of condensation. An HTML file records the completion of formatting. Each persistent representation marks the completion boundary of a particular process and simultaneously constitutes the invocation condition for subsequent processes. The file exists at the intersection of two completion events: the completion it records and the completion it enables.

The filesystem therefore functions as a *memory of completed wavefronts*. This is not

a metaphor. It is a structural description of what files are in this context. The conventional view treats files as storage: containers that hold data between uses. The flow computing view treats files as temporal wavefront markers: evidence that a process boundary was reached, and readable proof that the next boundary may be attempted.

This is why a mature flow computing workflow preserves intermediate representations rather than discarding them after each stage. `audio.mp3`, `transcript.txt`, `summary.txt`, `overview.txt`, `overview.html` are not waste products of the pipeline. They are the pipeline's memory: a record of which wavefronts have propagated and a set of re-invocable starting conditions for any stage that needs to be rerun, inspected, or rerouted.

This also explains the debuggability of flow computing workflows that was noted in the Unix section. Because each intermediate representation is a captured wavefront rather than a hidden internal state, the entire history of a pipeline execution is visible on disk. The pipeline can be restarted from any captured wavefront. A failed downstream stage does not require rerunning the entire pipeline; it requires only identifying which wavefront is the last complete one and reinvoking from there.

Whisper and Ollama as Cognitive Operator Boundaries

The integration of Whisper and Ollama into a flow computing ecology deserves particular attention because it illustrates the power of treating AI capabilities as completion-governed operator boundaries rather than as application destinations.

When Ollama is invoked as a GUI application or a web interface, the user is the mathematician: they decide when to paste in the input, when to submit the query, when to copy the output. The language model's cognitive capability is enclosed within an application that requires external coordination at each use.

When Ollama is invoked as a Bash pipeline stage, the language model is a *completion-governed behavior boundary*: it receives an input stream, transforms it, and emits an output stream. It fires when its input is complete. It signals completion when its output is stable. The mathematician has been removed. The cognitive capability is available to the entire ecology, invocable from any stage in any pipeline, without any GUI visit.

This is not a minor convenience. It is the difference between a cognitive tool that requires external coordination at every use and a cognitive operator boundary that participates in completion-governed process networks as a first-class member of the ecology.

Personal Languages as Logically Determined Expression Documents

Fant distinguishes in *Computer Science Reconsidered* between *autonomous* and *referential* process expression. An autonomous expression is complete in itself: it contains all the coordination behavior necessary for self-realization. A referential expression uses shorthands and appeals to established conventions, but must remain mappable to an autonomous expression.

A mature flow computing workflow develops an extensive referential expression vocabulary over time: shell aliases, shell functions, Bash scripts, Python utilities, AutoHotkey hotstrings, Makefile targets. Each entry in this vocabulary is a referential expression: a named shorthand, mappable to the underlying network of completion relationships that it abbreviates.

This accumulated vocabulary is not a configuration file. It is a *logically determined expression document*: a record of which completion-governed processes have been found stable enough, general enough, and recurrent enough to deserve names.

The vocabulary grows through a recognizable logic. A pipeline that is first written inline becomes a shell function when it proves recurrent. A shell function that is called frequently becomes an alias. An alias that needs to be accessible from any context becomes an AutoHotkey hotstring. Each step in this progression is a step toward greater referential compression of an increasingly well-understood completion relationship.

Reading a mature expression document reveals the user's process ecology at the level of its logical structure. Every entry is a completion relationship that has stabilized. Every name is a wavefront that has been given a token. The document is, in Fant's terms, a specification of the spontaneous behavior of the ecology: a map of which completions invoke which processes.

Crucially, the document is readable, inspectable, and modifiable in a way that

an application's internal logic never is. The referential expression is transparent to the completion relationships it encodes. There is no hidden mathematician inside the vocabulary. The coordination behavior is explicit.

Computational Literacy as Completion Fluency

The conventional account of computational literacy identifies it with proficiency in the use of applications. A computationally literate person can use word processors, spreadsheets, email clients, and browsers. More advanced literacy might include some programming knowledge.

The flow computing paradigm implies a different account. Literacy is not proficiency in operating external coordinators. Literacy is fluency in reading and writing completion-governed process networks: knowing which operator boundaries are available in an ecology, what representations each speaks, what completion event each signals, and how they compose into larger expressions.

This is, precisely, fluency in a logically determined system. The flow-literate practitioner understands the ecology neighborhood by neighborhood, in humanly graspable steps, exactly as Fant describes building trust in a logically determined circuit. She does not need to maintain a global state model. She needs to understand local completion relationships and compose them into larger expressions.

Such literacy is more fundamental than conventional programming literacy in the following sense: programming literacy asks whether you can express an algorithm in a notation a machine can execute; flow literacy asks whether you can express a completion-governed process network in terms that are self-coordinating. The second question is closer to the actual subject of computer science as Fant defines it: the science of process expression, not the science of computation.

It is also more achievable for most practitioners. The operator boundaries of a Unix ecology are largely provided. Whisper, Ollama, Pandoc, Git, and the standard Unix toolchain are already complete, well-specified behavior boundaries. Flow literacy requires knowing them, understanding their completion events, and composing them. The sophistication is in the composition; the implementation of the individual boundaries has already been done.

Modal Computation and Nested Scope: Vim, Byobu, and Spherepop

The flow computing paradigm, as developed so far, concerns process composition: how operator boundaries are coupled by completion relationships and how wavefronts propagate through networks of invocations. But Fant's deeper principle—that coordination should be *local, compositional, and explicit* rather than imposed from outside—extends beyond process composition into the structure of the computational environment itself.

Unix pipes express this principle at the level of processes. Vim expresses it at the level of interaction. Byobu expresses it at the level of workspace organization. And underneath all three, at the level of formal semantics, the same structure appears in the Spherepop calculus of nested irreversible scopes.

Vim as Modal Computation

Most text editors are application-centric. The user enters text; the editor provides commands; the commands are secondary affordances of a predominantly passive interface. The editor is a place. Commands are decorations on it.

Vim inverts this relationship. The fundamental abstraction of Vim is not the command but the *mode*. Normal mode, insert mode, visual mode, command-line mode, operator-pending mode: each mode defines a local behavioral neighborhood within which particular transformations are admissible. A key sequence acquires meaning only relative to the currently active mode. The same keystroke that inserts a character in insert mode deletes a word in normal mode or selects a region in visual mode.

This is structurally analogous to a logically determined system. The active mode is a completeness condition that gates which transformations may be invoked. A keystroke is a potential name whose transformation rule is determined by the mode context in which it is formed. Coordination is local: the current mode is the minimal context required to determine what any action means. There is no global application state to consult; the mode provides all the context the operation requires.

The editor does not manage the user's actions with an external controller. The mode *is* the controller—but it is a local, compositional controller, embedded

in the expression rather than imposed from outside. Vim is, in this sense, a logically determined interaction surface.

Operators and Scopes as Compositional Grammar

Vim extends the local composition principle into the structure of its commands. Many Vim operations are not atomic actions but *compositions of independent components*:

Operator	Motion/Scope	Command
`d` (delete)	`aw` (around word)	`daw`
`c` (change)	`i"` (inside quotes)	`ci"`
`y` (yank)	`ap` (around paragraph)	`yap`
`gq` (format)	`ip` (inside paragraph)	`gqip`

An operator expresses a transformation: delete, change, yank, format, indent. A motion expresses a scope: character, word, sentence, paragraph, delimited region. The resulting command is formed compositionally. The operator is a completion-governed function waiting for a scope to complete its invocation condition. The motion is that completion.

This mirrors the Unix pipeline principle precisely. `daw` is a two-stage pipeline: a deletion operator coupled to a scope boundary, the scope boundary forming the completion condition that invokes the deletion. The operator-pending mode is the state in which the operator has been named but the scope has not yet completed: a behavior boundary waiting for its input wavefront to arrive.

The space of possible commands is not enumerated but generated. There are perhaps a dozen operators and several dozen motions; their cross product yields hundreds of compositional commands that the user can express without having memorized any of them as atomic operations. Complex editing behavior emerges from combinations of small, orthogonal components, exactly as complex pipeline behavior emerges from combinations of simple operator boundaries. The Unix philosophy is alive inside the keystroke.

Vim's Shell Integration as Pipeline Stage

Vim does not merely resemble a flow computing environment. It participates in one. The `:!` command executes an external process and displays its output.

The `:r !cmd` command inserts the output of an external process into the current buffer as a new wavefront of text. Most significantly, `:%!cmd` pipes the entire buffer through an external process and replaces it with the result.

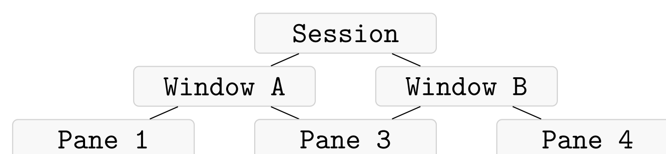
In each case, Vim becomes a stage in a pipeline: a behavior boundary that can receive a wavefront from an upstream process, present it to a human operator for inspection or modification, and pass the modified wavefront to a downstream process. The human is not a mathematician who manages the pipeline from outside. The human is a cognitive operator boundary embedded within the pipeline: a stage whose completion event is a deliberate save or write.

This is how Vim fits into the broader ecology. It is not an application that exists alongside the pipeline. It is a pipeline stage that provides a human-readable, modally-structured transformation surface at any point where human judgment is required.

Byobu as Hierarchical Process Space

Byobu is a terminal multiplexer: it allows a single terminal connection to host multiple shell sessions, each divided into windows and panes. Most users treat it as a convenience for managing many terminals at once. In a flow computing ecology, it is something structurally more significant.

Byobu instantiates a *nested scope architecture*: a hierarchy of behavioral containers, each enclosing the levels below it and providing a distinct organizational context:



Each level in this hierarchy is a scope: a bounded context with its own identity, its own set of active processes, and its own relationship to the levels that contain it. A pane contains a shell. A window contains panes organized around a shared task context. A session contains windows organized around a project. Sessions can be detached and reattached, persisting across disconnections: the session is a persistent scope that outlives any particular terminal instance.

This is the flow computing principle of local coordination applied to workspace organization. Each scope is coherent on its own terms. Processes inside a scope

inherit its context without needing to reach outside it for coordination. The scope is self-sufficient: it contains the environment, history, aliases, working directory, and active processes that its inhabitants require.

Byobu Keybindings as Scope Navigation

Byobu's keybinding structure exposes the session-window-pane hierarchy as a navigable space rather than a flat list of terminals. The standard bindings directly reflect the nested scope architecture:

- **F2** creates a new window (a new top-level scope within the session).
- **Shift-F2** subdivides the current window horizontally; **Ctrl-F2** subdivides it vertically. Both create new pane-level scopes within the current window.
- **Shift-F3** / **Shift-F4** move focus between sibling panes: lateral navigation within a scope level.
- **F3** / **F4** move between windows: navigation at the enclosing scope level.
- **F6** detaches the entire session, preserving all scope state while freeing the terminal connection. **Shift-F6** detaches without logging out.
- **Alt-PgUp** / **Alt-PgDn** enter scrollback mode: access to up to 10,000 lines of a pane's wavefront history, navigable and copyable without disrupting the active process.
- **Ctrl-a ~** saves the current window's scrollback buffer: explicitly capturing the pane's residue as a persistent file.

The practitioner navigating this hierarchy is not selecting terminals. They are moving between scopes. Each scope has a semantics: this window is the transcription workspace; that window is the editing workspace; this pane runs the Python server; that pane runs the build loop. The scopes are persistent—they survive connection interruptions, system sleep, and context switches to other work. Reattaching a Byobu session is not opening a terminal. It is re-entering a process ecology that has been running in your absence, its wavefronts propagating, its residue accumulating.

The scrollback buffer deserves particular note. Each pane maintains up to 10,000

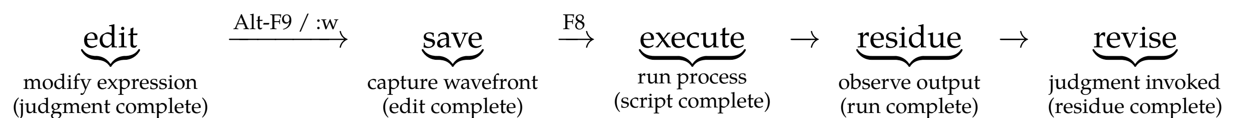
lines of output history: the full residue of every process that has run within that scope. This residue is inspectable (**Alt-PgUp**), copyable, and saveable (**Ctrl-a ~**). The pane is not merely a process container. It is a bounded scope with a memory of its own completed wavefronts.

Development as Wavefront Refinement

The flow computing character of a software development workflow is most clearly visible in the iterative edit-run cycle. A typical configuration binds:

- **Alt-F9**: open the active script in Vim for editing.
- **F8**: execute the script and observe its output in an adjacent pane.

The resulting cycle is a completion-governed feedback loop:



Each arrow is a completion event. The save completes the edit. The F8 invocation completes because the file exists and is valid. The execution completes and deposits its output as residue in the adjacent pane. The observation of the residue completes the cognitive evaluation, which either terminates the cycle (the output is correct) or invokes a new edit pass (the output reveals a fault).

This is precisely the second-order flow computing model: a feedback cycle realized as a series of discrete completion-governed wavefronts. Each iteration is a complete feed-forward pass. The cycle is not a continuously active control loop. It is a sequence of invocations, each triggered by the completion of the previous one.

The same structure governs web development in a Byobu multi-pane layout. One pane edits HTML and JavaScript in Vim. An adjacent pane runs a local Python HTTP server:

```
python3 -m http.server 8000
```

A third pane (or a browser window) inspects the served output. The three panes

are three behavior boundaries in a completion-governed loop: edit completes, server serves the new wavefront, browser renders the residue, observation invokes the next edit. No application manages this loop. No scheduler coordinates the panes. The structure of the Byobu workspace, the scope each pane occupies, and the completion event of each process *are* the coordination.

The ecology manages itself.

Each shell session is a bounded computational scope. It possesses its own environment variables, command history, working directory, alias and function definitions, and set of active processes. A child shell inherits from its parent but modifies only its own copy of the inherited state. When the child exits, its modifications are gone. The parent is unchanged.

This scoping behavior is not a technical detail. It is the same organizational principle that runs through the entire flow computing ecology: local coherence, explicit inheritance, no hidden global state that requires external coordination. The shell session is a self-contained behavior boundary. Its completion event is the shell's exit. Its residue is whatever it has written to the shared filesystem before exiting.

The hierarchy of Byobu scopes—pane, window, session—is therefore a hierarchy of nested behavior boundaries, each self-coordinating within its local context, each capable of composing with neighboring scopes through the shared filesystem and process namespace.

Nested Scopes and Spherepop

The Spherepop calculus, developed as a formal model of irreversible event sequences, describes computation as the progressive resolution of nested semantic bubbles. A bubble is a bounded scope that can contain sub-bubbles. When a bubble resolves—reaches its completion event—it either pops (irreversibly commits its result to the enclosing scope) or collapses (discards its contents without effect). The enclosing scope receives the result or the absence of a result, and its own resolution proceeds accordingly.

The structural parallel with the Byobu/shell/Vim hierarchy is precise. A pane is a bubble. Its completion event is the process that exits within it. When the process completes, the result—the wavefront it has written to the filesystem or piped to a sibling—propagates to the enclosing scope. The window is a larger

bubble: its completion is the coherent resolution of its constituent panes around a shared task. The session is a larger bubble still.

The analogy extends to Vim's modal structure. Each Vim mode is a local scope: a bounded behavioral context within which particular transformations are admissible. The mode transition is not an application-level state change; it is a scope transition. Entering operator-pending mode is entering a sub-bubble that waits for a motion to complete its invocation condition. When the motion arrives, the sub-bubble resolves: the operator fires, the text is transformed, the mode returns to normal.

In Spherepop's terms: operator-pending mode is an open bubble whose resolution condition is the receipt of a valid motion. The motion is the event that pops the bubble. The transformation is the result deposited in the enclosing scope.

Scope Before Process

Traditional operating system descriptions emphasize processes as primary entities and scopes—directories, sessions, namespaces—as secondary organizational containers. The process is the unit of computation; the scope is merely where it runs.

The lived experience of a mature Unix flow computing environment inverts this relationship. The practitioner navigates sessions, windows, panes, projects, directories, and modal editing surfaces. Processes appear within these scopes and inherit their context. The question the practitioner asks is not *which process am I running?* but *which scope am I in?*

The scope comes first. The process is an inhabitant of the scope.

This inversion is not merely phenomenological. It has structural consequences. When scope is primary, the computational environment becomes navigable as a space of nested contexts rather than a flat list of running programs. The practitioner can orient themselves by scope membership rather than by process identity. They can carry context from one process invocation to the next by remaining within the same scope. They can isolate experimental work in a new scope without affecting established contexts. They can close a scope when its work is done and trust that its residue—whatever it has deposited in the shared filesystem—is available to other scopes that need it.

This is the deepest sense in which flow computing is not merely about pipes. Unix is not merely a philosophy of processes. It is a philosophy of *nested scopes within which processes become meaningful*: scopes that provide local coordination, local context, and local completion boundaries, composing into larger scopes through the universal medium of the shared filesystem and the universal representation of the text stream.

Vim's modes, Byobu's session-window-pane hierarchy, and Spherepop's nested bubbles are three expressions of the same underlying principle: that coordination should be local, compositional, and explicit; that the boundaries of a scope should determine what is admissible within it; and that the completion of a scope should deposit its residue in the enclosing scope rather than requiring an external coordinator to collect and route it.

The Recurrence of Nested Completion Structures

What becomes visible across the full ecology is not merely that each tool embodies the flow computing principle, but that the same organizational pattern recurs at every scale. Operator boundaries compose into commands. Commands compose into scripts. Scripts compose into pipelines. Pipelines inhabit panes. Panes inhabit windows. Windows inhabit sessions. Sessions inhabit projects. The computational environment exhibits a *recursive nesting of completion-governed scopes* whose structure is approximately self-similar across levels of abstraction:

Scale	**Completion-governed scope**
Keystroke	Vim operator + motion $\to$ command
Command	Behavior boundary (stdin $\to$ stdout $\to$ EOF)
Pipeline	Coupled behavior boundary chain
Pane	Shell session + scrollback residue
Window	Task-coherent pane collection
Session	Project-coherent window collection
Ecology	Named invocation vocabulary

This self-similarity is not incidental. It is the consequence of applying a single organizational principle—local completion-governed coordination—consistently at every level of the environment's design. Fant discovers the principle at the level of logic gates. McIlroy encodes it at the level of processes. Vim expresses it at the level of interaction. Byobu expresses it at the level of workspace. Spherepop formalizes it at the level of irreversible event calculus.

The same invariant, reproduced at each scale: a bounded scope becomes active when its prerequisites are complete, does its work, signals completion, and deposits its residue in the enclosing scope. This is the structural claim of the essay, stated as concisely as possible.

The Residue Hierarchy

The various residue-preserving mechanisms of the ecology form a natural hierarchy ordered by persistence and commitment:

Level	Mechanism	Character
Transient	Pane scrollback buffer	Ephemeral; up to 10k lines
Volatile	Shell variable, Vim register	Scope-local; lost on exit
Session	Shell history (`.bash_history`)	Persistent across sessions
Named	Alias, shell function	Referential; explicitly authored
Committed	File on filesystem	Captured wavefront
Versioned	Git commit	Timestamped residue with provenance
Invocable	Script, AutoHotkey entry	Autonomous process expression

Each level represents a greater degree of commitment to the stability of the captured wavefront. The progression is not merely technical; it mirrors the epistemological arc of earned automation:

$$\text{observation} \rightarrow \text{capture} \rightarrow \text{stabilization} \rightarrow \text{naming} \rightarrow \text{invocation}$$

A process first produces residue in the most transient form: output visible in the scrollback buffer, a Vim register holding an intermediate value. If the output proves useful it is promoted: written to a file, added to history, crystallized into a function. If the function proves stable and general it is promoted further: committed to a script, entered into the AutoHotkey vocabulary, made invocable from any context.

The AutoHotkey file, the `.bashrc`, the `.vimrc`, the Git repository, and the filesystem are therefore not separate tools with separate purposes. They are successive commitment levels in a single residue hierarchy: each level encoding greater confidence that the captured wavefront represents a genuine completion event in a stable completion-governed process.

Feedback, Residue, and Second-Order Flow

The flow computing paradigm as described so far applies cleanly to *feed-forward* process networks: acyclic chains in which a wavefront propagates from source to product through a succession of completion-governed stages. Document conversion, media processing, corpus transcription, deployment pipelines—these are first-order flow computing, and they present no theoretical difficulty.

But real knowledge work is rarely acyclic. A transcript is summarized; the summary is critiqued; the critique generates a revised prompt; the revised prompt produces a new summary. Software is written; tests reveal failures; failures modify requirements; modified requirements change the code. Scientific inquiry proceeds by hypothesis, experiment, anomaly, revision, and new hypothesis. A cycle has formed.

The question this raises is not merely technical. It is theoretical: does the appearance of feedback cycles mean that flow computing requires an external coordinator after all—that cycles reintroduce the missing mathematician?

Feedback as Completion-Governed Wavefronts

The answer is no, provided that feedback is understood correctly. The threat of a feedback cycle is the threat of *continuous control*: a downstream process that continuously modifies the behavior of an upstream process, producing a system that never reaches a completion boundary and therefore never settles.

But feedback in a flow computing ecology does not need to be continuous. It can itself be completion-governed. The cycle becomes a sequence of discrete wavefront propagations:

$$\text{Wavefront}_1 \rightarrow \text{completion}_1 \rightarrow \text{feedback wavefront} \rightarrow \text{Wavefront}_2 \rightarrow \text{completion}_2 \rightarrow \text{feedb}$$

Each iteration of the cycle is a complete feed-forward propagation that reaches a genuine completion boundary before the next iteration begins. The cycle is not continuously active. It is a succession of discrete completion-governed invocations whose structure happens to include a return path.

In practice, this is precisely what a Bash loop with `&&` gates is: a sequence of

completion-governed wavefronts in which each iteration's output may condition the next iteration's input, but no iteration begins until the previous one has signaled completion. The recursion is real. The missing mathematician is not.

Residue: The Memory of Completed Wavefronts

Every process leaves something behind. This is not a side effect to be minimized. It is a structural property of completion-governed process networks that has a name: *residue*.

A transcript is the residue of a transcription process. A summary is the residue of a condensation process. A critique is the residue of an evaluation process. A Git commit is the residue of a versioning process. Residue is what a process deposits at its completion boundary: the captured wavefront that marks the process as done and simultaneously constitutes an invocation condition for future processes.

The concept of residue is what distinguishes memory from mere storage in a flow computing ecology. A file is storage when it is an inert container. A file is residue when it is the deposited completion record of a process whose structure is understood and whose output forms the prerequisite of a subsequent invocation.

Residue accumulates. As the ecology runs—cycling through acquisition, transcription, summarization, critique, revision, publication—a growing corpus of captured wavefronts accumulates in the filesystem. This corpus is not a database. It is not an archive. It is the *memory of the ecology*: a record of which processes have run, what they produced, and what conditions now exist for future invocation.

In a feedback cycle, residue plays a specific role: the completion of one pass through the cycle deposits residue that becomes the prerequisite condition for the next pass. The cycle is *logically determined by its own accumulating residue*. The feedback is not continuous control; it is completion-governed sequential invocation conditioned by the ecology's own memory.

Two Regimes of Flow Computing

This suggests a natural distinction between two regimes:

First-order flow computing describes acyclic completion networks. A wavefront propagates from source to product through a fixed topology:

$$A \to B \to C \to D$$

Document conversion, media transcription, corpus formatting, and deployment pipelines are paradigmatic examples. The completion relationships are fixed; the wavefront moves in one direction; the process terminates.

Second-order flow computing describes completion networks whose topology evolves through the accumulation of residue. A wavefront propagates through the network; residue is deposited; the residue modifies the invocation conditions for future wavefronts; the topology of available invocations changes:

$$A \to B \to C \to \text{residue} \to A'$$

where A' is an invocation of A conditioned on what C deposited. Software development, scientific inquiry, AI-assisted writing, and organizational learning are paradigmatic examples. The completion relationships evolve; the wavefront cycles; the process is potentially unbounded.

The critical observation is that second-order flow computing does not abandon the flow paradigm. It extends it. The ecology remains organized around completion-governed invocation; residue provides the memory that allows invocation conditions to evolve; feedback cycles become series of discrete wavefront propagations rather than continuous control processes. The missing mathematician does not return.

A Note on Cloud Orchestration

This analysis has direct implications for how modern distributed computing architectures should be evaluated. Many enterprise systems are marketed as event-driven and completion-governed. Yet they accumulate vast supervisory infrastructure: schedulers, orchestrators, service meshes, state stores, controller loops.

From the flow computing perspective, this infrastructure is a collective mathematician. It is external coordination imposed on a process network that, in principle, could coordinate itself through local completion relationships. The

coordination logic is not *in* the processes; it is in the supervisory layer above them.

The question Fant would ask is the right question: why does the process network require so many external coordinators? A genuinely flow-computing cloud architecture would attempt to move coordination into local completion relationships and accumulated residue, reducing the role of centralized supervisory structures to the minimum the implementation substrate genuinely requires.

Whether that is fully achievable at internet scale remains open. What is not open is the diagnosis: centralized orchestrators are not a feature of scale. They are evidence that the coordination behavior has not been successfully encoded into the process network itself.

Stochastic Wavefronts and Semantic Admissibility

The introduction of large language models as cognitive operator boundaries (Section 6) creates a theoretical complication that the preceding analysis has deferred. In an NCL circuit or a deterministic Unix pipeline, the completion criterion is binary and precise: a data wavefront has either fully propagated or it has not; the exit code is either zero or nonzero. The wavefront is either admissible or it is NULL.

A large language model introduces a different regime. The model accepts a syntactically complete input wavefront and produces a syntactically complete output. At the shell level, it signals completion with exit code zero. But the output may be *semantically inadmissible* for its downstream stage: the format may be structurally valid but ecologically wrong, the content may hallucinate a representation that downstream processes cannot consume, the summary may omit a key claim that subsequent synthesis depends on.

This is the *weak completion problem*: a cognitive operator boundary can satisfy syntactic completion while failing semantic completion. In Fant's terms, it produces a candidate output that looks like a completed wavefront but is not yet suitable residue. The downstream stage cannot tell from the shell exit code alone whether the wavefront it has received is trustworthy.

The solution is a second-order completion criterion: *admissibility*. Where rigid operator boundaries require only syntactic completion, cognitive operator boundaries require that the candidate output pass an admissibility test before being

committed as residue. Formalizing this:

$$B_i \xrightarrow{T_\theta} \tilde{B}_{i+1}$$

where B_i is the input boundary state, T_θ is a stochastic cognitive operator, and $\tilde{B}_{i+1}$ is a *candidate wavefront*—not yet valid residue. The candidate must satisfy an admissibility functional:

$$\mathcal{A}\Big(\tilde{B}_{i+1} \mid C, S, F\Big) \;\geq\; \tau$$

where C is the current context, S is the expected schema, F is downstream fitness, and τ is the commitment threshold. The boundary transition rule becomes:

$$B_{i+1} = \begin{cases} \text{commit}(\tilde{B}_{i+1}) & \text{if } \mathcal{A} \geq \tau \\ \text{repair}(\tilde{B}_{i+1}) & \text{if } \tau_r \leq \mathcal{A} < \tau \\ \text{quarantine}(\tilde{B}_{i+1}) & \text{if } \tau_q \leq \mathcal{A} < \tau_r \\ \text{collapse}(\tilde{B}_{i+1}) & \text{if } \mathcal{A} < \tau_q \end{cases}$$

yielding four possible outcomes: the candidate is committed as residue; repaired through a corrective loop and reconsidered; quarantined (preserved but excluded from downstream invocation); or collapsed (discarded without residue). In practice, this maps to familiar workflow patterns:

Outcome	**Flow computing meaning**	**Bash realization**
Commit	Output is valid residue	Write to file, proceed
Repair	Retry with revised prompt	Loop with `&&` retry
Quarantine	Save for inspection, skip	Write to `*.suspect/`
Collapse	Discard candidate	`|| rm $out && exit 1`

This analysis distinguishes four levels of completion that a cognitive operator boundary must satisfy:

1. **Exit-code completion**: the process exited with status 0.

2. **Format completion**: the output conforms to the expected structural representation.

3. **Semantic completion**: the output contains the content that downstream invocations require.
4. **Ecological completion**: the output is admissible as residue in the specific context of the current ecology's state.

Only ecological completion constitutes genuine flow completion in the flow computing sense. A cognitive operator boundary that achieves only exit-code completion has not yet produced a wavefront. It has produced a candidate. The admissibility functional is the gate that converts candidates into residue.

The NULL Wavefront as Markov Boundary

The theoretical significance of Fant's NULL convention extends beyond the engineering problem of clockless coordination. The propagating NULL wavefront is, in information-theoretic terms, a *moving Markov boundary*: a structure that separates completed computational episodes from each other, creating conditional independence between them.

Consider an ordinary combinational logic network processing asynchronously changing inputs. At any given instant, some gates have propagated their new outputs and others have not. The network is in an intermediate state. There is no intrinsic marker distinguishing a completed computation from a partial one, a stable result from a transient hazard. The boundary between finished and unfinished is invisible. The network lacks a natural separator.

NCL's NULL wavefront provides that separator. Every signal occupies one of two distinguished states: DATA (a computation is in progress) or NULL (no computation is active). The alternating cycle:

$$\text{NULL} \rightarrow \text{DATA} \rightarrow \text{NULL} \rightarrow \text{DATA}$$

marks the topology of computational episodes. Crossing from NULL to DATA means: *this computation is now active*. Crossing back to NULL means: *this computation is complete*. The NULL manifold is a distinguished surface in state space. Crossing it is a completion event.

In probabilistic terms: once the NULL wavefront has fully propagated, everything that was true of the *previous* computation is irrelevant to the *next* compu-

tation. The NULL state renders the downstream process conditionally independent of the unresolved internal details of the upstream process. It is precisely the Markov property: given the boundary state, past and future are independent.

This connection illuminates the behavior of persistent intermediate representations in a flow computing ecology. A captured wavefront file—`transcript.txt`, `summary.txt`, `overview.html`—functions as a *frozen Markov boundary*. Once the transcript is complete, the downstream summarization process is conditionally independent of everything that happened during audio acquisition and transcription. It does not need to know about recording conditions, Whisper model versions, audio quality, or processing time. It needs only the transcript.

The transcript *is* the NULL wavefront, frozen: a complete, stable separator between the acquisition episode and the summarization episode. The chain of residue files:

$$\texttt{audio.mp3} \rightarrow \texttt{transcript.txt} \rightarrow \texttt{summary.txt} \rightarrow \texttt{overview.txt}$$

is a chain of Markov boundaries. Each completed artifact renders the downstream stage conditionally independent of the internal details of the upstream stage. The stages can be replaced, updated, or rerun independently without invalidating the others—provided each replacement produces an output admissible as the same Markov boundary.

This is why Fant's NCL feels, in retrospect, less like a hardware design technique and more like a general theory of computational boundaries. The NULL wavefront is not a trick for avoiding clocks. It is an explicit mechanism for constructing Markov-like separators between computational episodes—boundaries that distinguish what must be known from what can be safely ignored, at every scale from logic gates to knowledge pipelines.

Toward a Theory of Flow Computing

Bringing together the hardware insights of *Logically Determined Design* and the theoretical framework of *Computer Science Reconsidered*, we can now state the principles of flow computing with some precision.

The Fundamental Unit

The fundamental unit of computation in the flow computing paradigm is not the instruction, not the state transition, and not the function call. It is the *invocation relationship*: a directed dependency between two processes such that the completion of the first enables the activation of the second.

A computational system is a network of latent processes connected by invocation relationships. Programs, scripts, pipelines, build rules, workflow steps, and agent activations are all special cases of this general structure.

The Coordination Principle

Coordination in a flow computing system is governed by completion relationships encoded in the process network itself, not by any external scheduler. The system is logically determined: its behavior is completely and unambiguously determined by the logical relationships of completion and invocation among its processes.

No mathematician is required. The ecology coordinates itself.

The Expression Principles

Six principles follow from this foundation:

1. **Completeness over correctness.** The primary virtue of a process expression is not that it computes the right answer but that it expresses the coordination structure completely—that no external coordinator is required to make it flow.

2. **Invocation over execution.** Processes are not executed by external command. They are invoked by the formation of their prerequisite completions.

3. **Wavefronts over states.** The primary object of analysis is the wavefront propagating through the process network, not the collective state of the system at an instant.

4. **Composition over encapsulation.** Operator boundaries should be composable with other operator boundaries. Encapsulated applications that

require external coordination for every operation are regressive.

5. **Referential expression over recapitulation.** Recurrent completion-governed processes should be given names. The accumulated expression vocabulary is the record of the ecology's stabilized process structure.

6. **Transparency over opacity.** The coordination behavior should be explicit in the expression. A pipeline whose completion relationships are visible is more trustworthy than an application whose coordination logic is hidden.

7. **Residue over discard.** Intermediate representations are captured wavefronts, not waste. A process ecology that preserves its residue acquires memory. A process ecology that discards its intermediate states can never be resumed, inspected, or reasoned about across time.

The Computer as Medium

The deepest implication of the flow computing paradigm is a revision of what a computer is.

The state-machine view says: a computer is a machine that occupies states and transitions between them under the direction of external control.

The flow computing view says: a computer is a medium through which processes flow, coordinated by completion relationships that are explicit in the structure of the process network itself. Its memory is the residue of completed wavefronts. Its learning—in the most general sense of that word—is the modification of future invocation conditions by accumulated residue.

This is Fant's wavefront, generalized to the scale of personal knowledge work. The YouTube URL is a data wavefront presented to the first boundary. The audio, the transcript, the summary, the overview, the HTML, the commit—these are successive wavefronts propagating through a network of completion-governed invocations, each depositing residue, each conditioning the next invocation. No mathematician touches them. The ecology coordinates itself.

Unix pipelines are the surviving environment in which this vision remains practical and visible. They are not relics of a pre-GUI era. They are a working, software-level approximation of the flow computing paradigm: a system organized around completion-governed invocation, whose behavior is determined

primarily by the logical relationships among its processes rather than by any external coordinator—even if that determination is approximate rather than formally guaranteed.

Flow Computing and the Return of Process Literacy

For most of human history, skilled practitioners understood the processes of their domains at the level of completion relationships. A farmer understood harvest cycles: which conditions enabled planting, which completion events marked readiness for harvest, which residues enriched the soil for the following season. A machinist understood production flows: which finishing operations were prerequisites for which assembly steps, which tolerances constituted genuine completion boundaries. An electrician understood signal paths: which completion events in one circuit enabled behavior in the next, where the wavefront could be inspected, where it could be rerouted.

This is process literacy: the ability to understand a domain not as a collection of tools but as a network of completion-governed transformations—to know what prerequisites each process requires, what residue it deposits, and what future invocations its completion enables.

The graphical user interface represented a genuine achievement in making computation accessible. It achieved this accessibility by abstracting away the process structure entirely: the user sees destinations, not transformations; applications, not completion boundaries; menus, not invocation conditions. This abstraction is appropriate for casual use. Its cost is the systematic erosion of process literacy in the domain of computing.

A generation of technically sophisticated people has grown up who can use computers with great facility and cannot read a pipeline. They know which applications to launch. They do not know what completion events govern the transformations those applications perform, what intermediate representations move between stages, or how the stages could be rearranged, replaced, or extended. The process structure is invisible to them because the interface was designed to make it invisible.

Flow computing, understood as a computing paradigm rather than a historical curiosity, represents the recovery of process literacy for the domain of compu-

tation. It asks practitioners to understand their work at the level of completion relationships:

- Can you identify the completion boundaries in your workflow?
- Can you name the representations that move between stages?
- Can you state the invocation condition for each process?
- Can you compose existing operator boundaries into novel process expressions?
- Can you recognize when a process has stabilized sufficiently to deserve a name?
- Can you read your own expression document and understand what it describes?

These questions are the flow-computing analogue of the farmer's knowledge of harvest cycles: not arcane expertise, but the basic comprehension of a practitioner who understands what they are doing at the level of the domain's actual structure.

Fant's ambition in *Computer Science Reconsidered* was not to propose a better programming language or a more efficient circuit design. It was to recover process expression as the core subject of computer science: to insist that the discipline's fundamental question is not *what can be computed* but *how can processes be expressed*. Flow computing extends this ambition to the practitioner's level. The question it asks of the individual is not *can you operate Application X?* but *can you understand, name, compose, and extend the processes of your domain?*

That question is the return of process literacy. It is also, in the deepest sense, what the Unix philosophy was always about.

The Adversarial Complement: Environments Designed to Prevent Process Literacy

The flow computing argument has so far been constructive: it describes what a self-coordinating process ecology looks like and how practitioners can build and inhabit one. But the argument has an adversarial complement, and ignoring it would leave the account incomplete.

Contemporary digital platforms are, in the terms this essay has developed, environments specifically engineered to prevent the practitioner from maintaining a completion-governed process ecology. They do this not by force but by design: by constructing computational environments in which the practitioner cannot identify completion boundaries, cannot name invocation conditions, cannot distinguish candidate wavefronts from committed residue, and—most importantly—cannot maintain the admissibility threshold that separates genuine completion from simulated completion.

The analysis here draws on Trevor Paglen's account of the evolution of psychological operations and their absorption into platform architecture. Paglen traces a progression from behavioral manipulation (predicting what a user will click based on prior behavior) to neurological manipulation (predicting the brain state that a given stimulus will produce, using computational neuroscience models like Meta's Tribe V2 that bypass behavioral proxies entirely). The destination of this progression is a system that speaks directly to pre-cognitive perceptual processes, below the threshold of the rational layer that evaluates invocation conditions and admissibility criteria.

In the vocabulary of this essay: the platform is attempting to invoke processes in the practitioner's cognitive ecology without forming a genuine completion event. It simulates the data wavefront without the null wavefront that would separate one episode from the next. The practitioner's internal Markov boundaries are flooded with continuous, undifferentiated stimulation designed to prevent the NULL manifold from propagating.

The result is the subjective experience of compulsion: actions that feel chosen but whose invocation conditions were never formed by the practitioner's own completion-governed process network. The platform has inserted itself as an external mathematician, one whose coordination behavior is not in the service of the practitioner's process ecology but in the service of the platform's engagement metrics.

This is structurally the same as the clocked, application-centric desktop computing model—but with the external coordinator operating at the neurological rather than the interface level, and with adversarial rather than neutral intent. The GUI application localizes coordination in the user and makes them responsible for managing the sequence. The engagement-optimized platform removes coordination from the user entirely, driving invocation through dopaminergic

completion simulation rather than genuine process completion.

Paglen identifies a further mechanism that compounds this: the deliberate destruction of admissibility criteria through epistemic flooding. Drawing on the documented techniques of information operations—Richard Doty's UFO disinformation campaigns, the Surkov method of paying every faction simultaneously to collapse the credibility of the public sphere, the Bannon strategy of flooding the zone with noise—Paglen argues that destabilizing the idea of a shared reality is an end in itself. Once the practitioner can no longer distinguish genuine completion events from manufactured ones, the admissibility functional $\mathcal{A}$ loses its reference frame. Any candidate wavefront becomes equally admissible or inadmissible. The result is not false belief but *mass resignation*: the practitioner stops attempting to evaluate invocation conditions and withdraws from the process of forming them.

In the terms of this essay: epistemic flooding is an attack on the Markov boundary structure of collective reasoning. The NULL wavefront that would separate completed from incomplete computational episodes is replaced by continuous noise. The shared residue layer—the body of captured wavefronts that constitutes publicly available knowledge—is contaminated with indistinguishable synthetics. The ecology's memory becomes untrustworthy, and the second-order flow processes that depend on accumulated residue (scientific inquiry, democratic deliberation, historical understanding) lose their invocation conditions.

The flow computing response to this adversarial environment is not primarily technical. It is the same response that Fant proposes to the missing mathematician at the hardware level: encode the coordination behavior into the expression itself, and make the completion boundaries explicit and local rather than delegated to external authorities.

For the individual practitioner, this means: maintain a personal process ecology whose completion boundaries are visible and whose residue is inspectable. Prefer captured wavefronts on your own filesystem over content whose provenance and completion status you cannot verify. Prefer invocations whose invocation conditions you can state over compulsions whose triggering mechanism is opaque. Prefer earned automation—processes whose completion relationships you have observed and understood—over scripted engagement whose completion criterion is a platform's retention metric.

The flow computing ecology is, in this reading, not merely a productive computing environment. It is a form of epistemic hygiene: a set of practices for maintaining the integrity of your own process ecology's Markov boundary structure in an environment that has been engineered to dissolve it.

Karl Fant posed the problem of mathematicianless enlivenment in the context of electronic circuits. His solution—Null Convention Logic, completion-governed coupling, wavefront propagation—is a solution to the missing mathematician problem at the hardware level.

The same problem appears at every level of abstraction where computation occurs. At the level of software architecture: the algorithm requires a sequence controller; the concurrent process network does not. At the level of personal computing: the application requires the user to act as external coordinator; the pipeline does not. At the level of knowledge work: the GUI workflow requires a human mathematician to manage every intermediate state; the flow computing workflow encodes the coordination in the completion relationships of the process ecology.

Unix pipelines, AutoHotkey invocations, Bash completion gates, Whisper-to-Ollama-to-Pandoc-to-Git wavefront chains—these are not engineering conveniences. They are practical instantiations of the flow computing paradigm: computation as the propagation of process invocations through networks of completion relationships, without any external coordinator, without any mathematician.

The process ecology grows not by aggressively automating every task but by waiting until each process has disclosed its completion relationships—then naming it, compressing it, and making it available for composition. The AutoHotkey file is the fossil record of that discovery. The filesystem is the memory of completed wavefronts. The feedback cycles of revision and refinement are second-order flow: successive completion-governed wavefronts conditioned by accumulated residue.

The computer, so understood, is not a collection of applications waiting for a user to visit them. It is a medium through which processes flow, coordinated by completion relationships explicit in the process network itself; a medium whose memory is the residue of what has already completed; a medium whose intelligence lies not in any application but in the structure of the invocation relationships among its operator boundaries.

The practitioner who understands this—who can read a pipeline, name a process, compose operator boundaries, and grow an expression document through earned automation—has recovered the process literacy that the GUI generation traded away for accessibility, and has built something more: a personal computational environment whose Markov boundary structure is visible, whose residue is inspectable, and whose invocation conditions are formed by completion-governed process networks rather than by adversarial systems designed to simulate completion without achieving it.

That trade was worth making. The recovery is also worth making.

This essay was composed in Vim, compiled through a LuaLaTeX pipeline, and constitutes a working instance of the argument it advances.

Primary references:

Karl M. Fant, *Logically Determined Design: Clockless System Design with NULL Convention Logic* (Hoboken, NJ: John Wiley & Sons, 2005).

Karl M. Fant, *Computer Science Reconsidered: The Invocation Model of Process Expression* (Hoboken, NJ: John Wiley & Sons, 2007).