Organizational Patterns & Formal Models

for Workgraph

A mathematics of organizations mapped onto task graph primitives

February 2026

Contents

1. Executive Summary

Workgraph's primitives—tasks, dependency edges, roles, motivations, agents, a coordinator, evaluations, and an evolve loop—are not arbitrary design choices. They map precisely onto well-established concepts from organizational theory, cybernetics, workflow science, and distributed systems. This document develops a vocabulary and framework — a "mathematics of organizations"—that helps users think rigorously about how to structure work in workgraph.

Key findings:

1. **The task graph is a stigmergic medium.** Agents coordinate indirectly by reading and writing task state, exactly as ants coordinate via pheromone trails. No agent-to-agent communication is needed—the graph *is* the communication channel.

2. **`after` edges natively express the five basic workflow patterns** (Sequence, Parallel Split, Synchronization, Exclusive Choice, Simple Merge). Structural cycles (back-edges in `after` edges with `CycleConfig`) add structured loops and arbitrary cycles. Advanced patterns (discriminators, cancellation, milestones) require coordinator logic.

3. **The execute→evaluate→evolve loop is autopoietic.** The system literally produces the components (agent definitions) that produce the system (task completions that trigger evaluations that trigger evolution). This is Maturana & Varela's self-producing network, Luhmann's operationally closed social system, and Argyris & Schön's double-loop learning —all at once.

4. **Fork-Join is the natural topology of `after` graphs.** The planner→N workers→synthesizer pattern is workgraph's most fundamental parallel decomposition. It maps to MapReduce, scatter-gather, and WCP2+WCP3.

5. **The coordinator is a cybernetic regulator** operating an OODA loop, subject to Ashby's Law of Requisite Variety: the number of distinct roles must match or exceed the variety of task types, or the system becomes under-regulated.

6. **Evaluations solve the principal-agent problem.** The human principal delegates to autonomous agents under information asymmetry. Evaluations are the monitoring mechanism; motivations are the bonding mechanism; evolution is the incentive-alignment mechanism.

7. **Role design is an Inverse Conway Maneuver.** Conway's Law predicts that system architecture mirrors org structure. In workgraph, deliberately designing roles shapes the task decomposition and therefore the output architecture.

8. **The provenance log is organizational memory.** `wg trace` records the full causal chain of every workflow—not just what the current state is (stigmergy) but how it got there. This is Luhmann's structural memory: the system's capacity to selectively remember and forget its own history.

9. **Replay transforms memory into learning.** `wg replay` re-executes past workflows with different parameters (different models, quality thresholds, task subsets), enabling double-loop learning (Argyris & Schön) and counterfactual reasoning. Successful workflow patterns become organizational routines (Nelson & Winter 1982)—reusable functions extracted from traces, the system's equivalent of institutionalized know-how.

---

2. Stigmergy: The Task Graph as Coordination Medium

2.1. What is Stigmergy?

Stigmergy (from Greek *stigma* "mark" + *ergon* "work") is indirect coordination between agents through traces left in a shared environment. The term was coined by Pierre-Paul Grassé in 1959 to explain how termites coordinate mound construction without a central plan: each termite reads the current state of the structure and responds with an action that modifies that structure, which in turn stimulates further action by other termites.

As Heylighen (2016) defines it: "A process is stigmergic if the work done by one agent provides a stimulus that entices other agents to continue the job."

There are two fundamental types:

Type	Definition	Example	Persistence
Sematectonic	The work product itself serves as the stimulus	Termite mounds: the shape of the partial structure tells the next termite what to do	Permanent (structural)
Marker-based	A separate signal (marker) is deposited, distinct from the work product	Ant pheromone trails: the chemical trail is not the food, but a signal about the food	Transient (decays)

2.2. Workgraph is a Stigmergic System

A workgraph task graph is a stigmergic medium. Agents do not communicate with each other directly—they read and write to the shared graph, and the graph's state stimulates their actions.

Stigmergy Concept	Workgraph Equivalent
Shared environment	The task graph (`.workgraph/graph.jsonl`)
Sematectonic trace	A completed task's artifacts—the code, docs, or other work product left behind *is* the stimulus for downstream tasks
Marker-based trace	Task status changes (`Open`→`Done`, `Failed`), dependency edges, evaluation scores
Pheromone decay	Stale assignment detection (dead agent checks), task expiration
Stigmergic coordination	The coordinator polls the graph for "ready" tasks (all `after` predecessors terminal)—it reads the markers
Self-reinforcing trails	Tasks with good evaluation scores reinforce the role/motivation patterns that produced them (via evolve)

This is not a metaphor. It is a precise structural correspondence. The defining characteristic of stigmergy—that agents coordinate through a shared medium rather than through direct communication—is exactly how workgraph agents operate. Agent A completes task X, modifying the graph (setting status to `Done`, recording artifacts). Agent B, working on task Y with `after = [X]`, is now unblocked. B never spoke to A. The graph mediated the coordination.

2.3. Real-World Stigmergic Systems

Wikipedia is the canonical human example of stigmergy. An editor sees a stub article (the trace), is stimulated to expand it, and leaves a more complete article (a new trace) that stimulates further refinement. Open-source development works identically: a bug report (marker) stimulates a patch (sematectonic), which stimulates a review (marker), which stimulates a merge (sematectonic).

The theoretical literature connects stigmergy to self-organization, emergence, and scalability. Stigmergic systems scale better than centrally planned systems because adding agents does not increase communication overhead—the coordination cost is absorbed by the shared medium.

2.4. Implications for Workgraph Users

- **The task graph is your communication channel.** Write descriptive task titles, clear descriptions, and meaningful log entries—these are the "pheromone trails" that guide downstream agents.
- **Task decomposition is environment design.** How you break work into tasks determines the stigmergic landscape agents navigate. Fine-grained tasks create more frequent, smaller traces. Coarse-grained tasks create fewer, larger traces.
- **Evaluation records are marker traces.** They don't change the work product but signal information about its quality, guiding the evolve loop toward better agent configurations.

---

3. Workflow Patterns: What `after` Edges and Structural Cycles Can Express

3.1. The Workflow Patterns Catalog

The Workflow Patterns Initiative, established by Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski, and Alistair Barros, catalogued 43 control-flow patterns that recur across business process modeling systems. The original 2003 paper identified 20; a 2006 revision expanded this to 43. The initiative also catalogued 43 Resource Patterns and 40 Data Patterns.

These patterns provide a precise vocabulary for what any workflow system can and cannot express.

3.2. Patterns Natively Supported by `after`

Pattern	ID	Workgraph Expression	Example
Sequence	WCP1	`B.after = [A]`	`write-code → review-code`
Parallel Split	WCP2	Multiple tasks sharing the same predecessor: `B.after = [A]`, `C.after = [A]`	`plan → {implement-frontend, implement-backend}`
Synchronization (AND-join)	WCP3	`D.after = [B, C]`	`{frontend, backend} → integration-test`
Simple Merge	WCP5	Single successor of multiple predecessors, where only one fires	`{hotfix, feature} → deploy` (only one path active)

Implicit Termination	WCP11	Tasks with no successors simply complete	Leaf tasks in the graph

These five patterns—the basic directed-graph patterns—are the bread and butter of `after` graphs. (Note: workgraph is a directed graph, not necessarily a DAG — structural cycles are intentional.)

3.3. Patterns Added by Structural Cycles

Pattern	ID	Workgraph Expression
Arbitrary Cycles	WCP10	`after` edges forming a cycle, detected by Tarjan's SCC algorithm, with `CycleConfig` on the cycle header (`--max-iterations`, optional guard and delay)
Structured Loop	WCP21	Structural cycle with a guard condition on the `CycleConfig`

3.4. Patterns Requiring Coordinator Logic (Idioms)

These patterns cannot be expressed with static edges alone but can be achieved through coordinator behavior or conventions:

Pattern	ID	Idiom
Exclusive Choice	WCP4	A coordinator task evaluates a condition and creates only the appropriate successor task
Multi-Choice	WCP6	A coordinator task selectively creates subsets of successor tasks
Discriminator	WCP9	A join task is manually marked ready after the first of N predecessors completes
Multiple Instance (runtime)	WCP14-15	The coordinator dynamically creates N task copies at runtime based on data
Deferred Choice	WCP16	Multiple tasks created; coordinator cancels the unchosen ones
Cancel Task/Region	WCP19/25	`wg abandon <task-id>`—terminal status that unblocks dependents
Milestone	WCP18	A task checks the status of a non-predecessor ("is task X done?") before proceeding

3.5. Resource Patterns and the Agency

Beyond control-flow, Van der Aalst's Resource Patterns describe how work is distributed to agents. Several map directly:

Resource Pattern	Workgraph Equivalent
Role-Based Distribution (WRP2)	Tasks matched to agents by role
Capability-Based Distribution (WRP8)	Task `skills` matched against role capabilities

Automatic Execution (WRP11)	`wg service start`—the coordinator auto-assigns and spawns agents
History-Based Distribution (WRP6)	Evaluation-informed agent selection in auto-assign tasks
Organizational Distribution (WRP9)	The agency structure (roles, motivations) determines the distribution

3.6. Summary: Expressiveness Hierarchy

```
after edges alone:       WCP1-3, WCP5, WCP11 (basic directed-graph patterns)
+ structural cycles:     + WCP10, WCP21 (cycles and structured loops)
+ coordinator logic:     + WCP4, WCP6, WCP9, WCP14-16, WCP18-20, WCP25
+ resource patterns:     + WRP2, WRP6, WRP8, WRP9, WRP11
```

The design principle: **edges express structure; the coordinator expresses policy.**

---

4. Fork-Join, MapReduce, and Scatter-Gather

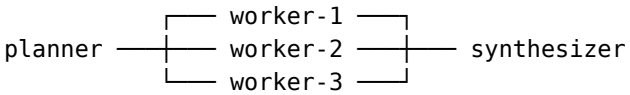
4.1. The Three Parallel Decomposition Patterns

These three patterns represent variations of the same fundamental idea — parallel decomposition with subsequent aggregation—originating from different fields:

Pattern	Structure	Origin	Key Distinction
Fork-Join	A task forks into N subtasks; a join barrier waits for all N to complete	OS/concurrency theory (Conway 1963, Lea 2000)	Strict barrier synchronization. All forks must join.
MapReduce	A map phase applies a function to each element in parallel; a reduce phase aggregates results	Dean & Ghemawat 2004, functional programming	Data-parallel. Decomposition driven by data partitioning, not task structure. Includes shuffle/sort between map and reduce.
Scatter-Gather	A request is scattered to N recipients; responses are gathered by an aggregator	Enterprise Integration Patterns (Hohpe & Woolf 2003)	Message-oriented. Recipients may be heterogeneous. Aggregation may accept partial results.

4.2. Fork-Join in Workgraph

Fork-Join is the natural topology of `after` graphs:




```
wg add "Plan the work" --id planner
wg add "Worker 1" --id worker-1 --blocked-by planner
```

```
wg add "Worker 2" --id worker-2 --blocked-by planner
wg add "Worker 3" --id worker-3 --blocked-by planner
wg add "Synthesize results" --id synthesizer --blocked-by worker-1 worker-2 worker-3
```

This is WCP2 (Parallel Split) composed with WCP3 (Synchronization). It is workgraph's most fundamental parallel pattern. Every fan-out from a single task is a fork; every convergence point with multiple `after` entries is a join.

4.3. MapReduce in Workgraph

MapReduce adds data-parallel semantics to fork-join. In workgraph, this is expressed as:

1. A **planner** task that analyzes input data and produces a decomposition
2. N **map** tasks (one per data partition), each `after` the planner
3. A **reduce** task that is `after` all map tasks and aggregates results

The coordinator creates the N map tasks dynamically based on the planner's output. The "shuffle" phase is implicit—each reduce task's description specifies which map outputs it consumes.

This is workgraph's most common pattern for parallelizable research, analysis, and implementation tasks.

4.4. Scatter-Gather in Workgraph

Scatter-Gather differs from fork-join in two ways: recipients may be heterogeneous (different roles), and the aggregator may not require all responses. In workgraph:

- **Heterogeneous scatter**: Assign different roles to the worker tasks. A security analyst, a performance engineer, and a UX reviewer all examine the same codebase.
- **Partial gather**: The synthesizer task can be unblocked by marking incomplete worker tasks as `Abandoned` (a terminal status). This is an idiom for the Discriminator pattern (WCP9).

4.5. Work-Stealing

Doug Lea's Fork/Join framework introduced work-stealing: idle threads steal tasks from busy threads' queues, achieving dynamic load balancing without central scheduling. The workgraph coordinator does something similar—when an agent finishes a task, the coordinator assigns it the next ready task regardless of which "queue" it originated from. The coordinator's `max_agents` parameter is the thread pool size.

---

5. Pipeline and Assembly Line Patterns

5.1. The Pipeline Pattern

A pipeline is a serial chain of specialized processing stages:

`analyst → implementer → reviewer → deployer`

Each stage transforms inputs into outputs consumed by the next stage. This maps directly to manufacturing and operations concepts:

Manufacturing Concept	Workgraph Expression
Assembly line	A chain of tasks with sequential `after` edges, each assigned to a different specialized role

Work station	A role—the specialized capability at each pipeline stage
Work-in-progress (WIP)	Tasks in `InProgress` status—the items currently being processed
Throughput	Rate of task completion—how many tasks move through the pipeline per unit time
Bottleneck	The pipeline stage with the longest average task duration (identifiable from `started_at`/`completed_at` timestamps)
WIP limit	`max_agents` parameter—limits how many tasks are simultaneously in-progress

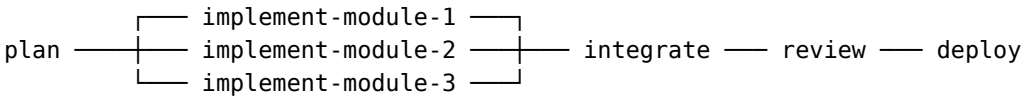
5.2. Pipeline vs. Fork-Join

These two patterns are complementary, not competing:

Dimension	Pipeline	Fork-Join
Parallelism type	Task-level (different stages run concurrently on different work items)	Data-level (same operation applied to multiple items simultaneously)
Role assignment	Different role per stage	Same role for all workers
When to use	Work requires sequential specialized transformation	Work is decomposable into independent parallel units
Workgraph shape	Long chain	Wide diamond

5.3. Combined Patterns

Real workflows combine both. A common workgraph pattern:



The middle is fork-join (parallel implementation); the overall shape is a pipeline (plan → implement → integrate → review → deploy). Each stage can have a different role:

- `plan`: architect role
- `implement-module-*`: implementer role
- `integrate`: implementer role
- `review`: reviewer role
- `deploy`: operator role

This is the Inverse Conway Maneuver in action—the role assignments shape the pipeline stages, which shape the output architecture.

5.4. Theory of Constraints

Eliyahu Goldratt's Theory of Constraints (1984) applies directly to pipelines:

1. **Identify** the bottleneck—the pipeline stage with the lowest throughput
2. **Exploit** the bottleneck—ensure it's never idle (keep it fed with ready tasks)
3. **Subordinate** everything else to the bottleneck—upstream stages should not overproduce

4. **Elevate** the bottleneck—add more agents to that role, or split the role into finer-grained specializations
5. **Repeat**—the bottleneck shifts; find the new one

In workgraph terms: if the `reviewer` role is the bottleneck, either assign more agents to that role, or decompose review into sub-roles (security review, code style review, correctness review) that can run in parallel.

---

6. Autopoiesis: The Self-Producing Agency

6.1. The Concept

Autopoiesis (from Greek *auto* "self" + *poiesis* "production") was introduced by Chilean biologists Humberto Maturana and Francisco Varela in 1972 to characterize the self-maintaining chemistry of living cells. An autopoietic system is:

> "A network of inter-related component-producing processes such that the components in interaction generate the same network that produced them."

Key properties:

Property	Definition
Self-production	The system's processes produce the components that constitute the system
Operational closure	Internal operations only produce operations of the same type; the system's boundary is maintained from within
Structural coupling	While operationally closed, the system is coupled to its environment —perturbations trigger internal structural changes, but the environment does not *determine* internal states
Structural determinism	The system's current structure determines what perturbations it can respond to and how

6.2. Luhmann's Social Systems Theory

Niklas Luhmann (1984) adapted autopoiesis for sociology with a radical move: **social systems are made of communications, not people.** People are in the *environment* of social systems, not their components. A social system is autopoietic because each communication connects to previous communications and stimulates subsequent ones — communications producing communications.

This reframing is strikingly applicable to workgraph: the *system* is the network of task state transitions and evaluations, not the agents themselves. Agents are in the environment of the workgraph system. What matters is the network of communications: "task X is done" triggers "task Y is ready" triggers "agent A starts work" triggers "task Y is in-progress"—communications producing communications.

6.3. The Evolve Loop is Autopoietic

The execute→evaluate→evolve→execute cycle maps precisely onto autopoietic self-production:

```
execute (agents produce artifacts)
   ↓
evaluate (artifacts produce evaluation scores)
   ↓
evolve (scores produce new role/motivation definitions)
   ↓
agents formed from new definitions → assigned to future tasks
   ↓
execute (cycle repeats)
```

Autopoietic Property	Workgraph Manifestation
Self-production	The evolve step produces new agent definitions (modified roles, motivations) that are themselves the components that execute the next cycle. The system literally produces the components that produce the system.
Operational closure	Agents interact only through the task graph. All "communication" is mediated by task state changes. The internal logic (role definitions, motivation constraints, evaluation rubrics) is self-referential.
Structural coupling	The task graph is coupled to the external codebase/project. Changes in the environment (new bugs, new requirements) perturb the system by adding new tasks, but the system's internal structure determines how it responds.
Cognition	Maturana and Varela argued that *living is cognition*—the capacity to maintain autopoiesis in a changing environment is a form of knowing. The evaluation system is the agency's cognition—its capacity to sense whether autopoiesis is being maintained (are tasks being completed successfully?) and adapt accordingly.
Temporalization	Tasks are momentary events. Once completed, they are consumed. The system must continuously produce new tasks (or iterate via structural cycles) to maintain itself. A workgraph with no open tasks has ceased its autopoiesis.

6.4. Practical Implications

The autopoietic framing suggests several design principles:

1. **The agency is alive only while tasks flow.** An idle agency with no open tasks is a dead system. Structural cycles keep the agency alive by re-activating tasks.
2. **Evolution is not optional—it is survival.** An agency that does not evolve in response to evaluation feedback will become structurally coupled to an environment that has moved on. The evolve step is the autopoietic system's metabolism.
3. **Perturbations enter through tasks, not through agents.** New requirements, bug reports, and changing priorities are perturbations that enter the system as new tasks. The system's response is determined by its current structure (which agents exist, what roles they have, what motivations constrain them).
4. **Self-reference is a feature, not a bug.** The evolve step modifying the very agents that will execute the next cycle is self-referential. This is what makes the system autopoietic. The self-mutation safety guard (evolver cannot modify its own role without human approval) is the autopoietic system's immune response — preventing pathological self-modification.

7. Trace as Organizational Memory: The Provenance Log

7.1. Beyond Stigmergic Traces

Section 1 established that the task graph is a stigmergic medium—agents leave traces (completed tasks, artifacts, status changes) that guide subsequent agents. But stigmergic traces are **environmentally embedded** and **structurally limited**:

- Stigmergic traces are marks *in the environment*—they tell you *what is* but not *how it got there*
- Pheromone trails decay; task statuses are overwritten (a task goes from `Open` → `InProgress` → `Done`, and the intermediate states are gone from the graph itself)
- Stigmergy captures the *current state* but not the *causal chain*

The provenance log (`wg trace`) transcends stigmergy by recording the **full operational history**: every mutation to the graph (add, claim, done, fail, retry, replay, restore), timestamped, attributed to an actor, with operation-specific detail. This is not a trace *in* the environment—it is a trace *about* the environment. It is metadata, not data.

Concept	What is Recorded	Persistence	Analogy
Stigmergic trace	Current state of the task graph	Overwritten by next state change	Pheromone trail (present only)
Provenance log	Complete history of all state changes	Append-only, immutable, compressed	Organizational memory (past preserved)
Agent archive	Full agent conversation (prompt + output + tool calls)	Immutable per attempt	Episodic memory of each work session

7.2. Luhmann's Structural Memory

Niklas Luhmann's concept of **structural memory** in social systems theory provides the deepest theoretical connection. For Luhmann, a social system's memory is not a storehouse of past events but the system's capacity to distinguish between *remembering* and *forgetting*—to use past experience to constrain future operations without becoming overwhelmed by history.

The provenance log is structural memory:

- It records *every* operation but is queried selectively (`wg trace <task-id>` filters to one task's history)
- The system *forgets* by default (agents don't read the full log) but can *remember* on demand (trace reconstructs the full lifecycle)
- Log rotation with zstd compression is literally the system managing the cost of memory—old memories are compressed but never deleted

Key Luhmann concepts apply directly:

- **Condensation**: The trace *summary* mode condenses full history into key statistics (duration, tool calls, turns). This is condensation—reducing the complexity of memory to usable form.
- **Generalization**: When multiple traces reveal the same pattern (e.g., "all failed tasks had >50 turns"), this generalizes across episodes. The trace system provides the raw material for generalization; the human operator (or evolve mechanism) performs the generalization.

- **Operative memory vs. system memory**: The task graph is operative memory (what the system currently needs to function). The provenance log is system memory (what the system has been through).

7.3. Organizational Learning Theory

The provenance log connects to the organizational memory literature:

- **Huber (1991)**, "Organizational Learning: The Contributing Processes and the Literatures": Distinguishes four constructs— knowledge acquisition, information distribution, information interpretation, and organizational memory. The provenance log is the *organizational memory* construct—the means by which knowledge is stored for future use.
- **Walsh & Ungson (1991)**, "Organizational Memory": Identify five retention facilities for organizational memory: individuals, culture, transformations, structures, and ecology. The provenance log maps to "transformations"—the system's record of its own decision-making processes.
- **Levitt & March (1988)**, "Organizational Learning": Organizations learn by encoding inferences from history into routines that guide behavior. The provenance log is the raw "history" that, through replay and trace-as-functions (Section 8), gets encoded into reusable routines.

7.4. The Three Layers of Memory in Workgraph

Workgraph's memory architecture comprises three distinct layers:

```
Layer 1: OPERATIVE MEMORY (the task graph)
  - Current task statuses, dependencies, assignments
  - Active stigmergic medium
  - Volatile: overwritten by each state transition

Layer 2: EPISODIC MEMORY (agent archives)
  - .workgraph/log/agents/<task-id>/<timestamp>/
  - Full prompt.txt and output.txt per agent run
  - Records *how* each task was worked on
  - Multiple episodes per task (retries create new timestamps)

Layer 3: PROCEDURAL MEMORY (provenance log)
  - .workgraph/log/operations.jsonl
  - Every graph mutation: add, claim, done, fail, edit, retry,
    replay, restore
  - Records *what happened* to the graph as a whole
  - Append-only, immutable, compressed rotation
```

7.5. Implications for Practice

- **Trace enables post-mortem analysis.** When a workflow fails, `wg trace --full` reconstructs the complete causal chain—not just "what failed" but "what sequence of events led to failure."
- **Trace enables attribution.** Every operation records an actor. When multiple agents touch a task (claim → fail → retry → claim → done), trace reveals who did what.
- **Trace enables temporal reasoning.** Timestamps on every operation allow computing durations, identifying bottlenecks (which stage took longest?), and detecting anomalies (why did this task take 10x longer than similar tasks?).

---

8. Replay as Organizational Learning: Double-Loop Learning Made Concrete

8.1. From Memory to Learning

Section 6 established that trace creates organizational memory. Replay (`wg replay`) is the mechanism by which the system *learns from* that memory.

The existing discussion of single-loop vs. double-loop learning (Section 9.4) can now be made concrete:

Learning Type	Argyris & Schön	Workgraph Mechanism	What Changes
Single-loop	Adjust actions within existing framework	Re-assign a failed task to a different agent	The *agent* changes; the *task structure* stays the same
Double-loop	Question and modify the framework itself	`wg evolve` modifies roles and motivations	The *framework* changes
Replay	Re-execute the framework with different parameters	`wg replay --model sonnet --failed-only`	The *execution context* changes; structure and framework are preserved

Replay is a **third mode of learning** that doesn't fit neatly into Argyris & Schön's taxonomy. It's not single-loop (it doesn't just reassign within the existing framework) and it's not double-loop (it doesn't modify the framework). It *re-executes* the framework with modified parameters. This is closer to **simulation** or **counterfactual reasoning**: "What would have happened if we had used a different model?" or "What would have happened if we re-ran only the failed tasks?"

8.2. Replay Filters as Learning Strategies

Each replay filter embodies a different organizational learning strategy:

Filter	Learning Strategy	Organizational Analog
`--failed-only`	Learn from failure: re-execute only what didn't work	Post-mortem → retry (Toyota's "stop the line")
`--below-score <n>`	Quality-gate learning: re-execute anything below standard	Six Sigma—eliminate below-threshold work
`--tasks a,b,c`	Targeted remediation: re-execute specific tasks	Surgical correction of known problems
`--model <model>`	Capability substitution: same work, different capabilities	Hiring a different specialist for the same role
`--keep-done <threshold>`	Preserve what works: only redo what's substandard	Incremental improvement—don't throw away good work
`--subgraph <root>`	Scope-limited replay: re-execute one branch of work	Division-level learning (not company-wide reset)
(default: all terminal)	Clean-slate replay: reset everything	Complete organizational restructuring

8.3. Transitive Reset as Causal Reasoning

When `wg replay` resets a task, it also resets all **transitive dependents**—tasks that depend (directly or transitively) on the reset task. This is not arbitrary; it is **causal reasoning**: if the upstream task is invalidated, everything downstream that consumed its output is also invalidated.

This connects to:

- **Causal inference in organizational learning** (Argyris 1993): learning requires understanding which actions caused which outcomes. The dependency graph *is* the causal model, and transitive reset *is* causal invalidation.
- **Garbage in, garbage out**: If a spec task was flawed and an implementation task consumed that flawed spec, replaying only the spec leaves a corrupted implementation in place. Transitive reset prevents this.

8.4. Snapshots as Experimental Records

`wg replay` creates a snapshot (`wg runs`) before resetting. This is not just a safety mechanism—it is an **experimental record**. Each snapshot records:

- The state of the graph before the experiment (run snapshot)
- What was changed (reset tasks, preserved tasks)
- The experimental parameters (filter, model override)

This transforms replay from "undo and redo" into "experiment and compare":

- `wg runs diff <run-id>` compares the pre-experiment state to the current (post-experiment) state
- `wg runs restore <run-id>` reverts the experiment if results are worse
- Multiple replays create a **series of experiments** that can be compared

This is the scientific method applied to workflow execution. Hypothesis → experiment → measurement → conclusion. The snapshots are the lab notebooks.

8.5. Counterfactual Reasoning

Replay enables a form of counterfactual reasoning that is rare in organizational systems: "What if we had done X differently?"

- **Counterfactual 1: Different capability**—`wg replay --model opus` → "What if we had used a more capable model?"
- **Counterfactual 2: Different scope**—`wg replay --failed-only` → "What if we only redid the parts that failed?"
- **Counterfactual 3: Quality threshold**—`wg replay --below-score 0.8 --keep-done 0.9` → "What if we kept the excellent work and only redid the mediocre work?"

Each counterfactual produces a new execution that can be compared to the previous one via `wg runs diff`. This is **double-loop learning made empirical**: rather than theoretically questioning governing variables, the system can *actually test* alternative governing parameters.

8.6. Connection to Simulation Theory

The organizational simulation literature provides further theoretical grounding:

- **March (1991)**, "Exploration and Exploitation": Replay with `--model` parameter is *exploration* of the capability space while holding the task structure constant. Replaying with

`--keep-done` is *exploitation*—preserving what works and only exploring alternatives for what doesn't.

- **Gavetti & Levinthal (2000)**, "Looking Forward and Looking Back": Forward-looking search (planning new task graphs) vs. backward-looking adaptation (replaying past graphs with modifications). Replay is backward-looking adaptation made concrete.

---

9. Trace-as-Functions: From Ad-Hoc Patterns to Organizational Routines

9.1. The Problem: Emergent Patterns Are Invisible

The autopoietic loop (Section 5) produces workflow patterns through self-organization. A human operator creates a task graph (spec → fanout → implement → validate → refine), runs it, and it works. But this pattern is **tacit knowledge**—it lives in the operator's head, not in the system. The next time a similar project arises, the operator must reconstruct the pattern from scratch.

The trace system (Section 6) makes these patterns **visible**—you can see the full execution history. But visibility is not reusability. The key insight:

> A successful trace is a function waiting to be extracted.

9.2. Nelson & Winter's Organizational Routines

Richard Nelson and Sidney Winter's *An Evolutionary Theory of Economic Change* (1982) introduced the concept of **organizational routines**—regular and predictable patterns of behavior by firms that serve as the "genes" of the organization:

- Routines are **skills of the organization**: they encapsulate know-how about how to accomplish tasks
- Routines are **heritable**: they persist across organizational changes and can be transmitted to new members
- Routines are **selectable**: routines that produce good outcomes are retained; those that produce poor outcomes are modified or abandoned
- Routines are the **unit of selection** in organizational evolution, analogous to genes in biological evolution

Nelson & Winter Concept	Workgraph Equivalent
Routine	A successful workflow pattern: a task graph topology + role assignments that produced good outcomes (high evaluation scores)
Routine as organizational memory	The trace records the routine's execution; the graph structure records the routine's form
Routine replication	`wg replay` re-executes a routine; trace extraction (future) could create reusable templates
Routine mutation	`wg replay --model <X>` replays with modified parameters; `wg evolve` modifies the agent definitions

Selection	Evaluation scores determine which routines are retained (`--keep-done`) and which are replayed

9.3. From Trace to Template: The Extraction Pipeline

The conceptual pipeline for turning a trace into a reusable function describes the theoretical framework that current and future features serve:

```
Step 1: EXECUTE
  A workflow pattern runs successfully
  (spec → 3x implement → integrate → validate)

Step 2: TRACE
  wg trace reveals the complete execution record:
  - What tasks were created, in what order
  - What dependencies existed
  - What roles were assigned
  - What evaluation scores were achieved
  - How long each stage took

Step 3: IDENTIFY PARAMETERS
  Which aspects varied (or could vary) across instances:
  - Number of parallel workers (3 implementers, could be 2 or 5)
  - Model used (sonnet, could be opus)
  - Role assignments (implementer role, could be specialist roles)
  - Task descriptions (specific to this project, would differ)

Step 4: EXTRACT
  The invariant structure — the topology, dependency pattern,
  role assignment strategy — becomes a template

Step 5: PARAMETERIZE
  The variable aspects become parameters:
  - N (number of parallel workers)
  - model (which model to use)
  - description_template (parameterized task descriptions)

Step 6: REPLAY AS FUNCTION CALL
  wg replay --subgraph <template-root> --model <new-model>
  re-instantiates the pattern with new parameters
```

This pipeline is currently partially implemented:

- Steps 1–2: Fully implemented (`wg trace`)
- Step 3: Manual (human identifies parameters by reading traces)
- Step 4: Partially implemented (the graph structure *is* the template; `wg replay` preserves structure while resetting execution state)
- Step 5: `wg replay --model` parameterizes model choice; `--below-score` parameterizes quality threshold
- Step 6: `wg replay` re-executes; future `wg template` or similar could formalize this

9.4. March's Exploration vs. Exploitation

James March's (1991) framework of **exploration** (search, variation, risk-taking, experimentation, discovery) vs. **exploitation** (refinement, efficiency, selection, implementation, execution) maps precisely onto the trace-to-template pipeline:

Phase	March's Framework	Workgraph Activity
First execution	Exploration	Create a new task graph, try a new workflow pattern, use untested role assignments
Trace analysis	Reflection	Examine what worked and what didn't; identify the effective pattern
Template extraction	Transition	Move from exploration to exploitation by codifying the discovered pattern
Replay as function	Exploitation	Re-use the proven pattern efficiently, with parameterized variations

March warns that organizations tend to over-exploit (stick with what worked before) at the expense of exploration (trying new patterns). In workgraph, this tension manifests as:

- **Over-exploitation**: Always replaying the same workflow pattern, even when the problem domain has changed
- **Over-exploration**: Always creating new task graphs from scratch, never building on proven patterns
- **Balance**: Using replay for known workflow types, fresh task graphs for novel problems

9.5. Feldman & Pentland: Routines as Generative Systems

Martha Feldman and Brian Pentland (2003) reconceptualized organizational routines not as fixed, dead patterns but as **generative systems** with two aspects:

- **Ostensive aspect**: The abstract, generalized pattern—the "idea" of the routine (e.g., "spec → implement → validate")
- **Performative aspect**: The specific, concrete enactment of the routine in a particular context (e.g., "spec-auth → implement-auth-1, implement-auth-2 → validate-auth")

Feldman & Pentland	Workgraph
Ostensive aspect	The abstract workflow topology and role assignment strategy (extractable from trace)
Performative aspect	Each specific execution (recorded in trace, each `wg runs` snapshot)
Routine dynamics	Each performance can deviate from the ostensive pattern, and these deviations may feed back to modify the pattern itself

Workgraph's trace captures the **performative** aspect with high fidelity (every operation, every agent conversation). The **ostensive** aspect emerges from comparing multiple performances: if three different projects all used a spec→fanout→validate pattern, the shared topology is the ostensive routine.

The evolve mechanism connects the two: when evaluation scores reveal that a performative deviation (e.g., adding a review step) improved outcomes, the evolve mechanism can update roles and motivations to institutionalize that deviation—modifying the ostensive routine.

9.6. The Evolutionary Cycle

Synthesizing Nelson & Winter + March + Feldman & Pentland yields a unified evolutionary model:

```
VARIATION (Exploration)
  New task graph topologies are created
  New role/motivation combinations are tried
       |
       ▼
SELECTION (Evaluation)
  Evaluation scores identify successful patterns
  wg trace reveals which topologies/assignments worked
       |
       ▼
RETENTION (Exploitation)
  Successful patterns are extracted from traces
  wg replay re-executes proven patterns
  Routines (templates) are stored for future use
       |
       ▼
REPLICATION (with Mutation)
  wg replay --model <X> reproduces the routine with a variation
  The variation produces new evaluation data
  The cycle repeats
       |
       ▼
VARIATION (back to top)
```

This is biological evolution applied to organizational workflow, with workgraph primitives as the substrate:

- **Genes** = workflow topologies + role assignments (the ostensive routine)
- **Phenotype** = the specific execution and its outcomes (the performative routine)
- **Fitness** = evaluation scores
- **Reproduction** = replay
- **Mutation** = replay with parameter changes (`--model`, `--below-score`)
- **Selection** = evaluation-based filtering (`--keep-done`, `--below-score`)

---

10. Cybernetics and Control Theory

10.1. Core Concepts

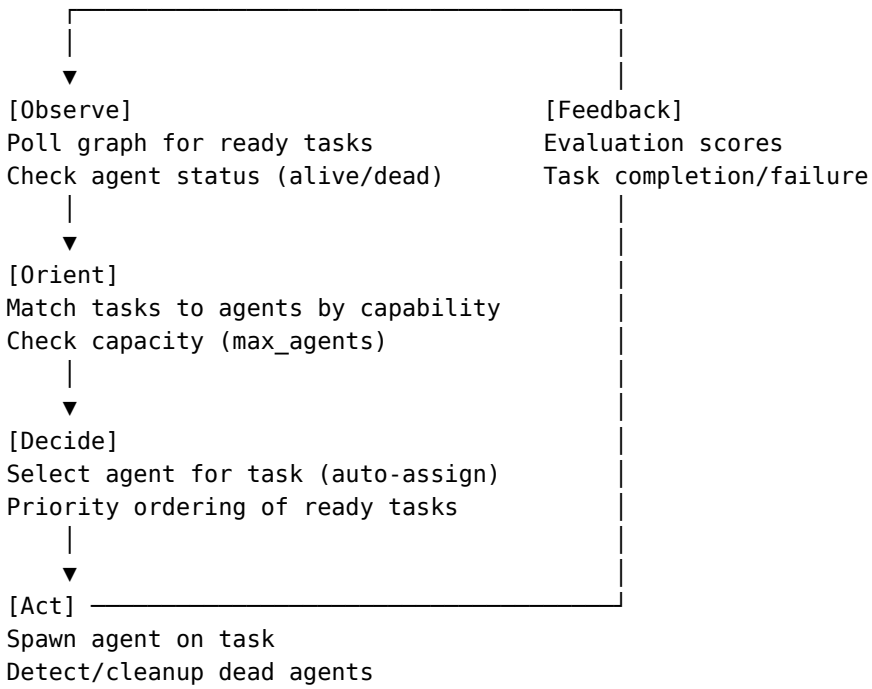
Cybernetics (from Greek *kybernetes* "steersman") is the study of regulatory systems—feedback loops, circular causality, and the science of control and communication. Founded by Norbert Wiener (1948) and W. Ross Ashby (1956), it provides the mathematical framework for understanding how systems maintain stability in changing environments.

Concept	Author	Definition
Negative feedback	Wiener (1948)	A signal from the output is fed back to the input to reduce deviation from a desired state. The basis of homeostasis.
Positive feedback	Wiener (1948)	Output amplifies input, producing exponential growth or runaway change.

Law of Requisite Variety	Ashby (1956)	"Only variety can absorb variety." A regulator must have at least as many states as the system it regulates.
Homeostasis	Cannon (1932)	A system maintains internal stability through negative feedback, adjusting to external perturbations.
OODA Loop	Boyd (1976)	Observe→Orient→Decide→Act. Competitive advantage from completing the loop faster.
Double-loop learning	Argyris & Schön (1978)	Single-loop: adjust actions to reduce error. Double-loop: question the governing variables themselves.
Second-order cybernetics	von Foerster (1974)	The cybernetics of cybernetics—the observer is part of the system.

10.2. The Coordinator as Cybernetic Regulator

The workgraph coordinator operates a control loop:



This is simultaneously: - An **OODA loop** (Boyd): Observe the graph state → Orient to available agents and tasks → Decide on assignment → Act by spawning - A **negative feedback loop** (Wiener): Failed tasks trigger re-assignment or retry, reducing deviation from the goal state (all tasks done) - A **homeostatic regulator** (Cannon/Ashby): The coordinator maintains steady throughput despite perturbations (agent failures, new tasks added, changing priorities)

10.3. Ashby's Law of Requisite Variety

Ashby's Law states: **"Only variety can absorb variety."** A regulator must have at least as many response options as the system has disturbance types. Formally: V(Regulator) ≥ V(Disturbances).

Applied to workgraph quantitatively:

- Let **V** = the number of distinct task types in the graph (identified by required skills, complexity, domain)
- Let **R** = the number of distinct roles in the agency

Ashby's Law requires R ≥ V for adequate regulation. If V grows (new kinds of work emerge) and R does not, the system becomes under-regulated—agents will be assigned to tasks they lack the capability for, producing poor results.

The `evolve` mechanism is precisely how the system increases its requisite variety: when evaluations reveal that existing roles cannot handle certain task types, the evolver creates new roles (increasing R) to match the growing variety of disturbances (V).

Requisite Variety Violation	Symptom in Workgraph	Fix
Too few roles for task variety	Low evaluation scores on certain task types	`wg evolve --strategy gap-analysis`
Too many roles (over-regulation)	Roles with zero task assignments, wasted agency complexity	`wg evolve --strategy retirement`
Motivation too restrictive	Tasks that require speed are assigned agents with "never rush" constraints	Tune acceptable/unacceptable tradeoffs

10.4. Single-Loop vs. Double-Loop Learning

Learning Type	Mechanism	Workgraph Equivalent
Single-loop	Adjust actions within existing framework to reduce error	Evaluations adjust which agent is assigned to which task type. Same roles, same motivations, different assignment.
Double-loop	Question and modify the framework itself	The `evolve` step modifies roles and motivations themselves. The governing variables change.

Single-loop: "This agent performed poorly on this task. Assign a different agent next time." Double-loop: "The role definition itself is wrong. Modify the role's skills and desired outcome."

Argyris and Schön argued that organizations that cannot double-loop learn become rigid and eventually fail. In workgraph, an agency that only re-assigns tasks without evolving its roles and motivations will plateau in performance.

---

11. The Viable System Model

11.1. Beer's Five Systems

Stafford Beer's Viable System Model (VSM) describes the organizational structure of any autonomous system capable of surviving in a changing environment. The model is **recursive**: every viable system contains viable systems and is contained within a viable system.

System	Name	Function	Key Principle
S1	Operations	The parts that *do things*. Multiple S1 units operate semi-autonomously.	Autonomy of operational units
S2	Coordination	Prevents oscillation and conflict between S1 units. Scheduling, protocols, standards.	Anti-oscillatory damping
S3	Operational Control	Optimizes the "here and now" across all S1 units. Resource allocation, synergy.	Internal optimization
S3*	Audit Channel	Sporadic checks that bypass normal reporting.	Independent verification
S4	Intelligence	Scans the external environment for threats and opportunities. Models possible futures.	Adaptation, strategic sensing
S5	Policy / Identity	Defines the organization's identity, purpose, and ground rules. Balances S3 (stability) and S4 (adaptation).	Organizational closure

The critical homeostatic balance is the **S3-S4 homeostat**: S3 wants stability and optimization of current operations; S4 wants exploration and adaptation. S5 mediates this tension.

11.2. Mapping to Workgraph

VSM System	Workgraph Equivalent
S1 (Operations)	**Agents** executing tasks. Each agent (role + motivation) is a semi-autonomous operational unit.
S2 (Coordination)	**`after` dependency edges** and **task status transitions**. These protocols prevent agents from clashing—an agent cannot start a task until its `after` predecessors are terminal. The coordinator's scheduling logic is S2.
S3 (Control)	**The coordinator** (`wg service start`). It allocates agents to tasks, monitors throughput, detects dead agents, and optimizes resource utilization across all S1 units.

S3* (Audit)	**Evaluations**. Sporadic, independent assessment of agent performance that bypasses normal task-completion reporting. The evaluation system provides a check that cannot be gamed by the agent reporting its own success.
S4 (Intelligence)	**The `evolve` mechanism**. It scans performance data (the "environment" of evaluation scores) for patterns and generates adaptations (new roles, modified motivations). Also: any human operator reviewing the graph and adding tasks based on environmental scanning.
S5 (Policy)	**Motivations** and **project-level configuration** (CLAUDE.md, the root of the task tree). These define the ground rules under which all agents operate—what is acceptable, what is not, what the system's identity and purpose are.
Recursion	Workgraph's task nesting. A high-level task can contain subtasks, each potentially a mini-viable-system with its own agents and coordination.

11.3. The S3-S4 Balance in Practice

In workgraph, the S3-S4 tension manifests as:

- **S3 pull (stability)**: "Keep using the existing roles and motivations—they're working fine. Optimize assignment. Don't change what isn't broken."
- **S4 pull (adaptation)**: "The task landscape is changing. New types of work need new roles. Evolve the agency."

The human operator is S5, mediating this tension. The evolve mechanism's self-mutation safety guard (requiring human approval for changes to the evolver's own role) is the S5 function enforcing identity preservation.

---

12. The Principal-Agent Problem

12.1. The Problem

The principal-agent problem, formalized by Ross (1973) and Jensen & Meckling (1976), arises when a **principal** delegates work to an **agent** who has different interests and more information than the principal.

Two core information asymmetries:

Problem	Timing	Description
Adverse selection	Before delegation	The agent has private information about their capabilities that the principal cannot observe. The principal may select the wrong agent.
Moral hazard	After delegation	The agent's actions are not fully observable. The agent may cut corners or pursue private objectives.

Agency costs (Jensen & Meckling 1976) = Monitoring costs + Bonding costs + Residual loss.

12.2. Workgraph as an Agency Relationship

This mapping is unusually precise because workgraph literally has primitives called "agents," "evaluations," and "motivations"—the vocabulary of agency theory.

Agency Theory Concept	Workgraph Equivalent
Principal	The human operator who defines the task graph and configures the agency
Agent	The workgraph agent (role + motivation)—literally named
Delegation	`wg service start --max-agents N`—the principal delegates work to autonomous agents
Moral hazard	The agent might produce low-quality output, hallucinate, or take shortcuts not visible from task completion status alone
Adverse selection	Assigning the wrong role+motivation pairing to a task—the agent lacks the capability, but this is not apparent until after execution
Monitoring costs	**Evaluations**—the computational cost of assessing agent output quality after every task
Bonding costs	**Motivations**—the agent is constrained by its motivation document to act in the principal's interest. Acceptable/unacceptable tradeoffs are the bonding contract.
Incentive alignment	**The evolve mechanism**—agents that perform well have their patterns reinforced; agents that perform poorly are evolved or retired. This is performance-based selection.
Residual loss	The gap between what the principal would produce and what the agent actually produces. Minimized by iterating evaluate→evolve.
Repeated games	The evaluation history builds "reputation" that informs future assignment and evolution. Long-term relationships (many tasks by the same agent) build trust (`TrustLevel::Verified`).
Screening	The coordinator's auto-assign capability-matching: skills on the task matched against skills on the role.

12.3. Mechanism Design Implications

Agency theory suggests specific design principles for workgraph:

1. **Invest in monitoring (evaluations) proportional to risk.** High-stakes tasks deserve more thorough evaluation. Low-stakes tasks can be spot-checked.
2. **Make bonding explicit.** The motivation's `unacceptable_tradeoffs` should list the specific failure modes the principal fears most. "Never skip tests" is a bonding clause.
3. **Align incentives through evolution.** The evolve mechanism should explicitly reward the behaviors the principal values. If correctness matters more than speed, the evaluation rubric should weight correctness heavily (it does—40% by default).
4. **Screen before delegating.** Auto-assign should match task skills against agent capabilities, not assign randomly. This reduces adverse selection.
5. **Build trust incrementally.** New agents should start with low-stakes tasks. The `TrustLevel` field (Unknown → Provisional → Verified) formalizes this progression.

---

13. Conway's Law and the Inverse Conway Maneuver

13.1. Conway's Law

> "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."—Melvin Conway, "How Do Committees Invent?" (1968)

Conway's argument: a system design is decomposed into parts, each assigned to a team. The teams must communicate to integrate the parts. Therefore, the interfaces between the system's parts will mirror the communication channels between the teams. This is a *constraint*, not a choice.

13.2. The Inverse Conway Maneuver

Coined by Jonny LeRoy and Matt Simons (2010): if org structure shapes system architecture, then **deliberately designing org structure can drive desired system architecture**. Rather than accepting that your system mirrors your org chart, you restructure teams to produce the architecture you want.

13.3. Mapping to Workgraph

Conway's Law Concept	Workgraph Equivalent
Organization structure	The set of roles and how they are assigned to agents
Communication channels	`after` edges between tasks assigned to different roles
System architecture	The task graph structure—how work is decomposed and connected
Conway's constraint	The task decomposition will mirror the role decomposition. If you have "frontend" and "backend" roles, you get tasks that split along that boundary, producing a system with that split.
Inverse Conway Maneuver	Deliberately designing roles and motivations to produce the desired task decomposition (and therefore system architecture)

13.4. The Inverse Conway Maneuver in Practice

Example: Microservices via roles

If you want a microservices architecture, define one role per service domain: - `user-service-developer` role - `payment-service-developer` role - `notification-service-developer` role

Tasks will naturally be decomposed along service boundaries, and the resulting code will have clean service interfaces—because the dependency edges between tasks assigned to different roles become the API contracts between services.

Example: Monolith via cross-cutting roles

If you want a cohesive monolith, define cross-cutting roles: - `backend-developer` role (handles all backend work) - `frontend-developer` role (handles all frontend work)

Tasks will be decomposed by layer, not by domain, producing a layered monolith.

The profound implication: **in workgraph, the role ontology IS the org chart, and the task graph IS the system architecture.** Conway's Law predicts they will converge. The Inverse Conway Maneuver says: design the roles first, and the task graph (and resulting code) will follow.

---

14. Team Topologies

14.1. The Framework

Team Topologies (Skelton & Pais, 2019) provides a practical framework for organizing technology teams, built on Conway's Law and cognitive load theory.

Four team types:

Team Type	Purpose	Cognitive Load Strategy
Stream-aligned	Aligned to a single valuable stream of work (product, service, user journey). The primary type—most teams should be this.	Owns and delivers end-to-end; minimizes handoffs
Platform	Provides internal services that accelerate stream-aligned teams. Treats offerings as products with internal customers.	Reduces cognitive load of other teams by providing self-service capabilities
Enabling	Specialists who help stream-aligned teams acquire missing capabilities. Cross-cuts multiple teams.	Temporarily increases capability of other teams, then steps back
Complicated-subsystem	Maintains a part of the system requiring heavy specialist knowledge (ML model, codec, financial engine).	Isolates specialist knowledge so others don't need it

Three interaction modes:

Mode	Description	Duration
Collaboration	Two teams work closely together for a defined period (joint exploration). High bandwidth.	Temporary (weeks)
X-as-a-Service	One team provides, another consumes, with a clear API/contract. Low overhead.	Ongoing (steady-state)
Facilitating	One team helps and mentors another. One-way knowledge transfer.	Temporary (until capability transferred)

14.2. Mapping to Workgraph Roles

Team Topologies Concept	Workgraph Equivalent
Stream-aligned team	A role assigned to a stream of related tasks. The "default" role type.
Platform team	A role whose tasks produce shared infrastructure that unblocks other agents' tasks. Platform tasks appear as `after` dependencies for stream-aligned tasks.
Enabling team	A role whose tasks improve other roles/agents—writing documentation, creating templates, establishing patterns. Maps naturally to the evolve mechanism.
Complicated-subsystem team	A role with specialized capabilities, assigned to tasks that other agents should not attempt.
Collaboration mode	Two agents sharing `after` edges on overlapping tasks during a discovery phase.
X-as-a-Service mode	Clean `after` edges: platform tasks complete, stream-aligned tasks consume their outputs.
Facilitating mode	An enabling agent's tasks are prerequisites for another agent's improvement.
Cognitive load	The number and complexity of tasks assigned to a single agent. Overload signals the need for role decomposition.
Team API	The interface between roles—defined by what outputs one role produces that another role's tasks consume.

14.3. Practical Guidance for Workgraph Users

1. **Most roles should be stream-aligned.** If you have a "build the feature" type of work, that's stream-aligned. Don't over-specialize.
2. **Create platform roles for shared infrastructure.** If multiple stream-aligned agents need the same tooling/setup, create a platform role whose tasks they all depend on.
3. **Use enabling roles sparingly.** An "evolver" that reviews the agency and proposes improvements is an enabling role. It shouldn't exist permanently—it should work itself out of a job.
4. **Complicated-subsystem roles protect cognitive load.** If a task requires deep ML expertise, create a specialized role rather than expecting a general-purpose role to handle it.
5. **Interaction modes evolve.** Two agents might collaborate on initial exploration, then shift to X-as-a-Service once interfaces stabilize. The task graph structure should reflect this evolution.

---

15. Organizational Theory Primitives

15.1. Division of Labor

Adam Smith's pin factory (1776): splitting work into specialized steps increases productivity. In workgraph, this maps to:

- **Task decomposition**: Breaking a large task into smaller subtasks, each with a specific focus
- **Role specialization**: Defining roles with narrow skill sets (analyst, implementer, reviewer) rather than one generalist role
- **The pipeline pattern**: Sequential stages of specialized work (Section 4)

The tradeoff: over-specialization increases coordination costs (more `after` edges, more handoffs, more potential for misalignment). This is the fundamental tension in organizational design, and it applies directly to workgraph agency design.

15.2. Span of Control

The number of subordinates a manager can effectively supervise. In workgraph: the number of agents a single coordinator tick can effectively manage. The `max_agents` parameter is the span of control.

Research suggests 5-9 direct reports as optimal for human managers (Urwick, 1956). For workgraph, the constraint is computational: how many agents can the coordinator monitor, evaluate, and evolve without losing oversight quality.

15.3. Coordination Costs

Every dependency edge (`after`) is a coordination point. The total coordination cost of a task graph scales with the number of edges, not the number of tasks. This connects to Brooks's Law: "Adding manpower to a late software project makes it later"—because the number of communication channels grows as n(n-1)/2 with n participants.

In workgraph terms: adding more agents (higher `max_agents`) only helps if the task graph has enough parallelism to exploit. If the graph is a serial chain, more agents are wasted. If the graph is a wide diamond (fork-join), more agents directly increase throughput—up to the point where coordination overhead dominates.

15.4. Transaction Cost Economics

Oliver Williamson's Transaction Cost Economics (1975, 1985) asks: when should work be done inside the organization ("make") vs. outside ("buy")? The answer depends on:

Factor	Favors "Make" (Internal)	Favors "Buy" (External)
Asset specificity	High (specialized knowledge needed)	Low (commodity work)
Uncertainty	High (requirements change frequently)	Low (well-defined)
Frequency	High (recurring work)	Low (one-off)

In workgraph, this maps to the choice between: - **Internal agents** (agency-defined roles with specialized capabilities) for recurring, high-specificity work - **External agents** (human operators, one-off shell executors) for infrequent, well-defined tasks

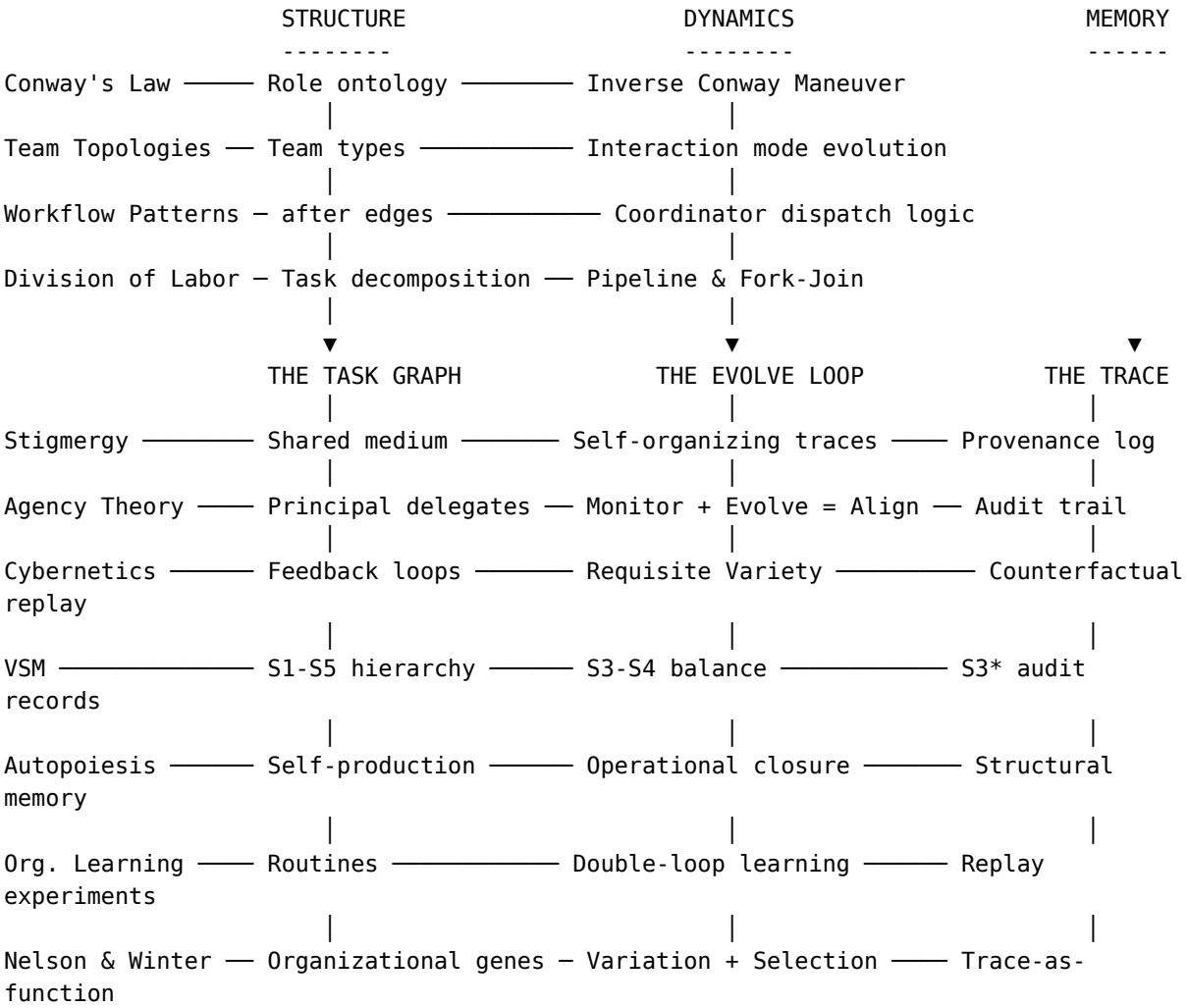
The `executor` field on agents (`claude`, `matrix`, `email`, `shell`) represents this make-vs-buy boundary.

---

16. Synthesis: Cross-Cutting Connections

These frameworks are not independent—they are deeply interconnected views of the same underlying organizational dynamics.

16.1. The Grand Unification



16.2. Key Structural Identities

Several deep identities connect these frameworks:

1. **VSM + Cybernetics + OODA**: Beer's VSM is explicitly cybernetic. S3 is a negative feedback regulator; S4 is the Observe/Orient function; S5 is the governing variable for double-loop learning. The coordinator's OODA loop IS the S3 regulation cycle.

2. **Stigmergy + Autopoiesis**: Both describe systems that maintain themselves without central control. Stigmergy is the *mechanism* (indirect coordination through traces); autopoiesis is the *property* (self-production). A stigmergic system that produces its own traces is autopoietic.

3. **Conway's Law + Team Topologies + Resource Patterns**: Conway's Law is the theoretical prediction; Team Topologies is the practical prescription; Workflow Resource Patterns are the formal specification. All three say: *how you assign people to work determines the structure of what gets built.*

4. **Principal-Agent + Evaluations + Cybernetics**: Agency theory identifies the *problem* (misaligned incentives under information asymmetry); cybernetics provides the *solution architecture* (feedback loops); evaluations are the *implementation* of both monitoring (agency theory) and negative feedback (cybernetics).

5. **Fork-Join + Workflow Patterns + `after`**: Fork-Join is the computational realization of WCP2+WCP3, which are the two most fundamental `after`-edge graph topologies.

6. **Autopoiesis + Evolve + Double-Loop Learning**: The execute→evaluate→evolve→execute cycle is simultaneously autopoietic (self-producing), double-loop (questioning governing variables), and cybernetic (feedback-driven regulation). This is the single most theoretically dense primitive in workgraph.

7. **Stigmergy + Trace + Organizational Memory**: Stigmergy creates *present* traces in the environment; the provenance log creates *persistent* traces about the environment. Stigmergy is the system's working memory; trace is its long-term memory. Together they provide the memory architecture for the autopoietic system (Luhmann's structural memory).

8. **Replay + Double-Loop Learning + Autopoiesis**: Replay is double-loop learning operationalized—the system can question its own execution by re-running it with different parameters. Combined with autopoiesis: the self-producing system can now re-produce itself under counterfactual conditions, testing whether its self-production is robust to parameter changes.

9. **Trace-as-Function + Nelson & Winter + March**: The extraction of reusable routines from traces is the evolutionary retention mechanism (Nelson & Winter). The choice between replaying a proven routine and creating a new task graph is the exploration/exploitation tradeoff (March). The evolve mechanism modifies the routines' agent components, while replay modifies their execution parameters—two orthogonal axes of adaptation.

10. **Runs + VSM S3* + Experimental Records**: Run snapshots serve dual purpose: as S3* audit records (independent verification of what state the system was in) and as experimental records (enabling comparison of different execution strategies). This connects the VSM's audit function to the scientific method.

---

17. Practical Recommendations

17.1. Agency Design Checklist

Based on the theoretical frameworks above, here is a checklist for designing a workgraph agency:

Step	Framework	Question	Action
1	Conway's Law	What system architecture do I want?	Design roles to mirror the desired decomposition
2	Requisite Variety	Do I have enough roles for the variety of tasks?	Count task types, ensure ≥1 role per type
3	Team Topologies	Which role is stream-aligned? Platform? Enabling?	Label roles by type; most should be stream-aligned
4	Division of Labor	How fine-grained should specialization be?	Balance specialization against coordination cost
5	Principal-Agent	What failure modes do I fear most?	Encode them in motivations as `unacceptable_tradeoffs`

6	Cybernetics	Is the feedback loop working?	Enable auto-evaluate; run evolve periodically
7	VSM S3-S4	Am I balancing stability and adaptation?	Don't evolve too often (S3) or too rarely (S4)

17.2. Pattern Selection Guide

Situation	Pattern	Workgraph Expression
Sequential specialized stages	Pipeline (Section 4)	Serial `after` chain, different roles per stage
Independent parallelizable work	Fork-Join (Section 3)	Fan-out from planner, fan-in to synthesizer
Data-parallel analysis	MapReduce (Section 3)	Planner decomposes → N workers → reducer aggregates
Heterogeneous parallel review	Scatter-Gather (Section 3)	Multiple reviewer roles examine same artifact
Iterative refinement	Structured Loop (Section 2)	Structural cycle with `CycleConfig` (`--max-iterations`, guard, delay)
Recurring process	Autopoietic cycle (Section 5)	Structural cycle forming a full cycle

17.3. Anti-Patterns

Anti-Pattern	Theory Violated	Symptom	Fix
One role for all tasks	Requisite Variety	Low scores on specialized tasks	Add specialized roles
Too many roles	Parsimony / coordination cost	Roles with zero tasks; confusion in assignment	Retire unused roles
No evaluations	Agency Theory (no monitoring)	Quality drift; no evolution signal	Enable auto-evaluate
Evolving every cycle	VSM S3-S4 imbalance	Instability; roles changing faster than agents can adapt	Evolve periodically, not continuously
Serial pipeline where fork-join fits	Division of Labor mismatch	Slow throughput on parallelizable work	Decompose into parallel tasks
Monolithic tasks	No division of labor	Single agent bottleneck; no parallelism	Break into subtasks with dependencies
Unconfigured structural cycle	Workflow Patterns (unbounded cycle)	Deadlock (no `CycleConfig`) or infinite loop	Add `--max-iterations` on cycle header

18. Appendix: Comparative Tables

18.1. Framework-to-Primitive Mapping

Framework	Tasks	`after`	Structural Cycles	Roles	Motivations	Agents	Coordinator	Evaluations	Evolve	Trace	Replay	Runs
Stigmergy	Traces in environment	Sematectonic coordination	Feedback trail	—	—	Stimulated actors	—	Marker traces	Self-reinforcing trails	Long-term trace memory	Trail reinforcement	—
Workflow Patterns	Activities	Control-flow edges	Back-edges (WCP10/21)	Resource roles (WRP2)	—	Resources	Engine	—	—	Execution history	WCP reset/restart	—
Fork-Join/MapReduce	Map/fork units	Join barriers	—	—	—	Worker threads	Scheduler	—	—	—	—	—
Autopoiesis	Momentary events	Process network	Self-production cycle	System components	System boundary	Environment	—	Cognition	Self-production	Structural memory	Self-reproduction with variation	Experimental branching
Cybernetics	System states	Causal chains	Feedback loops	Regulator variety	Constraints	Regulated units	Regulator (OODA)	Feedback signal	Variety amplification	Observation record	Counterfactual testing	Experimental control
VSM	S1 operations	S2 coordination	S3 audit cycle	S1 capabilities	S5 policy	S1 units	S3 control	S3* audit	S4 intelligence	S3* audit records	S4 adaptation mechanism	S3* verification baseline
Agency Theory	Delegated work	Contract terms	Repeated games	Agent type	Bonding contract	Agent	Principal	Monitoring	Incentive alignment	Monitoring records	Incentive recalibration	Contract history
Conway's Law	System components	Interfaces	—	Team capabilities	—	Teams	—	—	Inverse Conway	Architecture audit trail	—	—
Team Topologies	Work streams	Team interactions	—	Team types	—	Teams	—	—	Topology evolution	—	—	—
Org Theory	Labor units	Coordination channels	—	Specializations	Values	Workers	Manager	Performance review	Restructuring	Institutional memory	Restructuring mechanism	—
Org Learning	Routine enactments	Causal dependencies	Iterative refinement	Routines (ostensive)	Behavioral norms	Routine performers	Learning coordinator	Episodic memory	Double-loop adaptation	Episodic memory	Double-loop reexecution	Experimental record
Evolutionary Theory	Phenotype expression	Selection pressure	Generational cycle	Organizational genes	Selection criteria	Organisms	Selection mechanism	Fitness evaluation	Variation + selection	Fitness record	Reproduction with mutation	Snapshot of prior generation

18.2. Theoretical Density of Workgraph Primitives

Primitive	Frameworks That Map To It	Theoretical "Load"
Tasks	All 10 frameworks	The universal unit of work
`after` edges	Workflow Patterns, Fork-Join, Stigmergy, Conway's Law, Coordination Costs	The structural backbone
Structural cycles	Workflow Patterns (WCP10/21), Cybernetics (feedback), Autopoiesis (self-production), Agency Theory (repeated games)	Enables dynamics

Roles	Team Topologies, Conway's Law, Resource Patterns, VSM (S1), Requisite Variety, Division of Labor	The competency model
Motivations	Agency Theory (bonding), VSM (S5 policy), Cybernetics (constraints)	The value system
Agents	Agency Theory (literally), Stigmergy (stimulated actors), VSM (S1 units), Team Topologies (teams)	The executing entity
Coordinator	Cybernetics (regulator), VSM (S3), OODA Loop, Agency Theory (principal's delegate)	The control system
Evaluations	Agency Theory (monitoring), Cybernetics (feedback signal), VSM (S3* audit), Autopoiesis (cognition)	The sensing mechanism
Evolve	Autopoiesis (self-production), Cybernetics (double-loop learning, variety amplification), VSM (S4 intelligence), Agency Theory (incentive alignment)	The adaptation mechanism
Trace	Org Learning (organizational memory—Huber, Walsh & Ungson), Autopoiesis (structural memory—Luhmann), Cybernetics (observation record), VSM (S3* audit), Agency Theory (monitoring records), Stigmergy (persistent traces)	The memory mechanism
Replay	Org Learning (double-loop learning—Argyris & Schön; exploration/exploitation—March), Autopoiesis (self-reproduction with variation), Evolutionary Theory (reproduction with mutation—Nelson & Winter), Cybernetics (counterfactual testing)	The learning mechanism
Runs	VSM (S3* audit baseline), Org Learning (experimental records), Evolutionary Theory (generational snapshots)	The experimental record

---

19. Sources

19.1. Organizational Theory

- Smith, A. (1776). *An Inquiry into the Nature and Causes of the Wealth of Nations*. Book I, Chapter 1: "Of the Division of Labour."
- Simon, H.A. (1947). *Administrative Behavior*. Macmillan. [Bounded rationality, satisficing]
- Williamson, O.E. (1975). *Markets and Hierarchies*. Free Press. [Transaction cost economics]
- Williamson, O.E. (1985). *The Economic Institutions of Capitalism*. Free Press.
- Urwick, L.F. (1956). "The Manager's Span of Control." *Harvard Business Review*, 34(3), 39-47.
- Brooks, F.P. (1975). *The Mythical Man-Month*. Addison-Wesley.

19.2. Workflow Patterns

- van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., & Barros, A.P. (2003). "Workflow Patterns." *Distributed and Parallel Databases*, 14(1), 5-51.
- Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., & Mulyar, N. (2006). "Workflow Control-Flow Patterns: A Revised View." BPM Center Report BPM-06-22.
- Russell, N., van der Aalst, W.M.P., & ter Hofstede, A.H.M. (2016). *Workflow Patterns: The Definitive Guide*. MIT Press.
- Russell, N., ter Hofstede, A.H.M., Edmond, D., & van der Aalst, W.M.P. (2005). "Workflow Resource Patterns." In *Advanced Information Systems Engineering (CAiSE)*, Springer.

19.3. Parallel Decomposition

- Dean, J. & Ghemawat, S. (2004). "MapReduce: Simplified Data Processing on Large Clusters." *OSDI '04*, 137-150.
- Lea, D. (2000). "A Java Fork/Join Framework." *ACM Java Grande Conference*, 36-43.
- Hohpe, G. & Woolf, B. (2003). *Enterprise Integration Patterns*. Addison-Wesley.
- Conway, M.E. (1963). "A Multiprocessor System Design." *AFIPS Fall Joint Computer Conference*. [The original fork-join concept]
- Blumofe, R.D. & Leiserson, C.E. (1999). "Scheduling Multithreaded Computations by Work Stealing." *JACM*, 46(5), 720-748.

19.4. Stigmergy

- Grassé, P.-P. (1959). "La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp." *Insectes Sociaux*, 6(1), 41-80.
- Theraulaz, G. & Bonabeau, E. (1999). "A Brief History of Stigmergy." *Artificial Life*, 5(2), 97-116.
- Heylighen, F. (2016). "Stigmergy as a Universal Coordination Mechanism I: Definition and Components." *Cognitive Systems Research*, 38, 4-13.
- Heylighen, F. (2016). "Stigmergy as a Universal Coordination Mechanism II: Varieties and Evolution." *Cognitive Systems Research*, 38, 50-59.
- Elliott, M. (2006). "Stigmergic Collaboration: The Evolution of Group Work." *M/C Journal*, 9(2).
- Bolici, F., Howison, J., & Crowston, K. (2016). "Stigmergic Coordination in FLOSS Development Teams." *Cognitive Systems Research*, 38, 14-22.

19.5. Autopoiesis

- Maturana, H.R. & Varela, F.J. (1972/1980). *Autopoiesis and Cognition: The Realization of the Living*. D. Reidel.
- Varela, F.J., Maturana, H.R., & Uribe, R. (1974). "Autopoiesis: The Organization of Living Systems." *BioSystems*, 5(4), 187-196.
- Maturana, H.R. & Varela, F.J. (1987). *The Tree of Knowledge*. Shambhala.
- Luhmann, N. (1984/1995). *Social Systems*. Stanford University Press.
- Mingers, J. (2002). "Can Social Systems Be Autopoietic?" *Sociological Review*, 50(2), 278-299.

19.6. Organizational Learning & Evolutionary Theory

- Huber, G.P. (1991). "Organizational Learning: The Contributing Processes and the Literatures." *Organization Science*, 2(1), 88–115.
- Walsh, J.P. & Ungson, G.R. (1991). "Organizational Memory." *Academy of Management Review*, 16(1), 57–91.
- Levitt, B. & March, J.G. (1988). "Organizational Learning." *Annual Review of Sociology*, 14, 319–340.
- March, J.G. (1991). "Exploration and Exploitation in Organizational Learning." *Organization Science*, 2(1), 71–87.
- Argyris, C. (1993). *Knowledge for Action: A Guide to Overcoming Barriers to Organizational Change*. Jossey-Bass.
- Nelson, R.R. & Winter, S.G. (1982). *An Evolutionary Theory of Economic Change*. Belknap Press / Harvard University Press.
- Feldman, M.S. & Pentland, B.T. (2003). "Reconceptualizing Organizational Routines as a Source of Flexibility and Change." *Administrative Science Quarterly*, 48(1), 94–118.
- Pentland, B.T. & Feldman, M.S. (2005). "Organizational Routines as a Unit of Analysis." *Industrial and Corporate Change*, 14(5), 793–815.
- Gavetti, G. & Levinthal, D. (2000). "Looking Forward and Looking Back: Cognitive and Experiential Search." *Administrative Science Quarterly*, 45(1), 113–137.

19.7. Cybernetics

- Wiener, N. (1948). *Cybernetics: Or Control and Communication in the Animal and the Machine*. MIT Press.
- Ashby, W.R. (1956). *An Introduction to Cybernetics*. Chapman & Hall.

- Ashby, W.R. (1958). "Requisite Variety and Its Implications for the Control of Complex Systems." *Cybernetica*, 1(2), 83-99.
- Boyd, J. (1976/1986). "Patterns of Conflict." [Unpublished briefing]
- von Foerster, H. (1974). *Cybernetics of Cybernetics*. University of Illinois.
- Argyris, C. & Schön, D.A. (1978). *Organizational Learning: A Theory of Action Perspective*. Addison-Wesley.
- Beer, S. (1959). *Cybernetics and Management*. English Universities Press.

19.8. Viable System Model

- Beer, S. (1972). *Brain of the Firm*. Allen Lane / Penguin Press.
- Beer, S. (1979). *The Heart of Enterprise*. Wiley.
- Beer, S. (1985). *Diagnosing the System for Organizations*. Wiley.
- Espejo, R. & Harnden, R. (1989). *The Viable System Model: Interpretations and Applications*. Wiley.

19.9. Principal-Agent Theory

- Ross, S.A. (1973). "The Economic Theory of Agency: The Principal's Problem." *AER*, 63(2), 134-139.
- Jensen, M.C. & Meckling, W.H. (1976). "Theory of the Firm: Managerial Behavior, Agency Costs and Ownership Structure." *JFE*, 3(4), 305-360.
- Holmström, B. (1979). "Moral Hazard and Observability." *Bell Journal of Economics*, 10(1), 74-91.
- Eisenhardt, K.M. (1989). "Agency Theory: An Assessment and Review." *AMR*, 14(1), 57-74.
- Laffont, J.-J. & Martimort, D. (2002). *The Theory of Incentives: The Principal-Agent Model*. Princeton University Press.

19.10. Conway's Law

- Conway, M.E. (1968). "How Do Committees Invent?" *Datamation*, 14(4), 28-31.
- LeRoy, J. & Simons, M. (2010). "The Inverse Conway Maneuver." *Cutter IT Journal*, 23(12).
- MacCormack, A., Rusnak, J., & Baldwin, C. (2012). "Exploring the Duality between Product and Organizational Architectures." *Research Policy*, 41(8), 1309-1324.

19.11. Team Topologies

- Skelton, M. & Pais, M. (2019). *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. IT Revolution Press.

19.12. Theory of Constraints

- Goldratt, E.M. (1984). *The Goal*. North River Press.