The Agentic Pyramid

Mapping a Hierarchy of Concepts onto a Hierarchy of Agents

Gene Leybzon

June 2025

Abstract

As agentic systems move from demos to production, the dominant design - a single capable model handed every tool and asked to reason across every timescale at once runs into a wall. Strategic deliberation and motor-level control have almost nothing in common except that, in a flat design, they share one context window, one objective, and one failure surface. This paper argues for an alternative organizing principle: **the Agentic Pyramid**, in which a hierarchy of *concepts* (levels of abstraction) is mapped onto a hierarchy of *agents*, each agent a specialist that reasons and acts only in the vocabulary of its own level. Lower levels are concrete, fast, and atomic; higher levels are abstract, slow, and strategic. We define what a *level* actually contains - its semantic graph, skill library, decision memory, knowledge assets, world model, guardrails, and interface contracts - and we work the pattern through four domains: a physical-AI factory, autonomous driving, a quantitative trading desk, and an autonomous software-engineering system. We then address the practical concerns that production deployments demand: human-in-the-loop oversight at each altitude, dynamic reconfiguration, per-level evaluation methodology, and cost/latency optimization. The pattern is not new in spirit; it is the rediscovery of a structure that control engineering, planning, and reinforcement learning each arrived at independently.

1. The Problem with Flat Agent Designs

The first instinct when building an agent is to give one model everything: every tool, every document, every instruction, and a goal. This works until it doesn't, and the ways it stops working are instructive.

Context bloat. A single agent responsible for both "achieve 94% on-time delivery this quarter" and "close the gripper to 12 N" must hold both kinds of information in the same context. The strategic facts (demand forecasts, contracts, capacity) and the tactical facts (joint limits, friction coefficients, frame timings) crowd each other out. Relevance collapses.

No separation of timescales. Strategy changes over days; control loops close in milliseconds. A flat agent has no natural place to put the difference. It either over-deliberates on fast decisions or under-deliberates on slow ones.

Brittle coordination. The popular fix - spawn many peer agents and let them talk - replaces hierarchy with a mesh. Mesh coordination has no privileged direction. Every agent can in principle message every other, and the emergent behavior becomes hard to predict, test, or bound.

One failure surface. When reasoning at all altitudes lives in one place, a failure at any altitude is a failure of the whole. There is no containment.

The throughline is that **abstraction levels are real, and flat designs pretend they aren't.** A factory genuinely has a level at which "weld this seam" is a primitive and a different level at which "weld this seam" is an irrelevant detail subsumed inside "hit the Q3 output target." Good architecture names those levels and gives each one its own agent.

2. The Core Idea

> A hierarchy of concepts maps one-to-one onto a hierarchy of agents. Each agent is a specialist that reasons and acts only in the concepts of its level. It treats the level below as its action space and the level above as its source of objectives.

Concretely:

- **Concepts** are levels of abstraction - atomic skills, composed tasks, orchestration, strategy.
- Each level has a **vocabulary**: the entities, relations, and verbs that are first-class at that level and invisible at others.
- Each level is staffed by one or more **agents** whose entire job is to operate fluently in that vocabulary.
- Agents communicate **vertically** through narrow contracts, not horizontally through open chat.

The shape is a pyramid because the layers narrow as they rise: many atomic skills at the base, few strategic goals at the apex. It is also a pyramid because each upper layer *rests on* and is *supported by* the competence of the layer below.

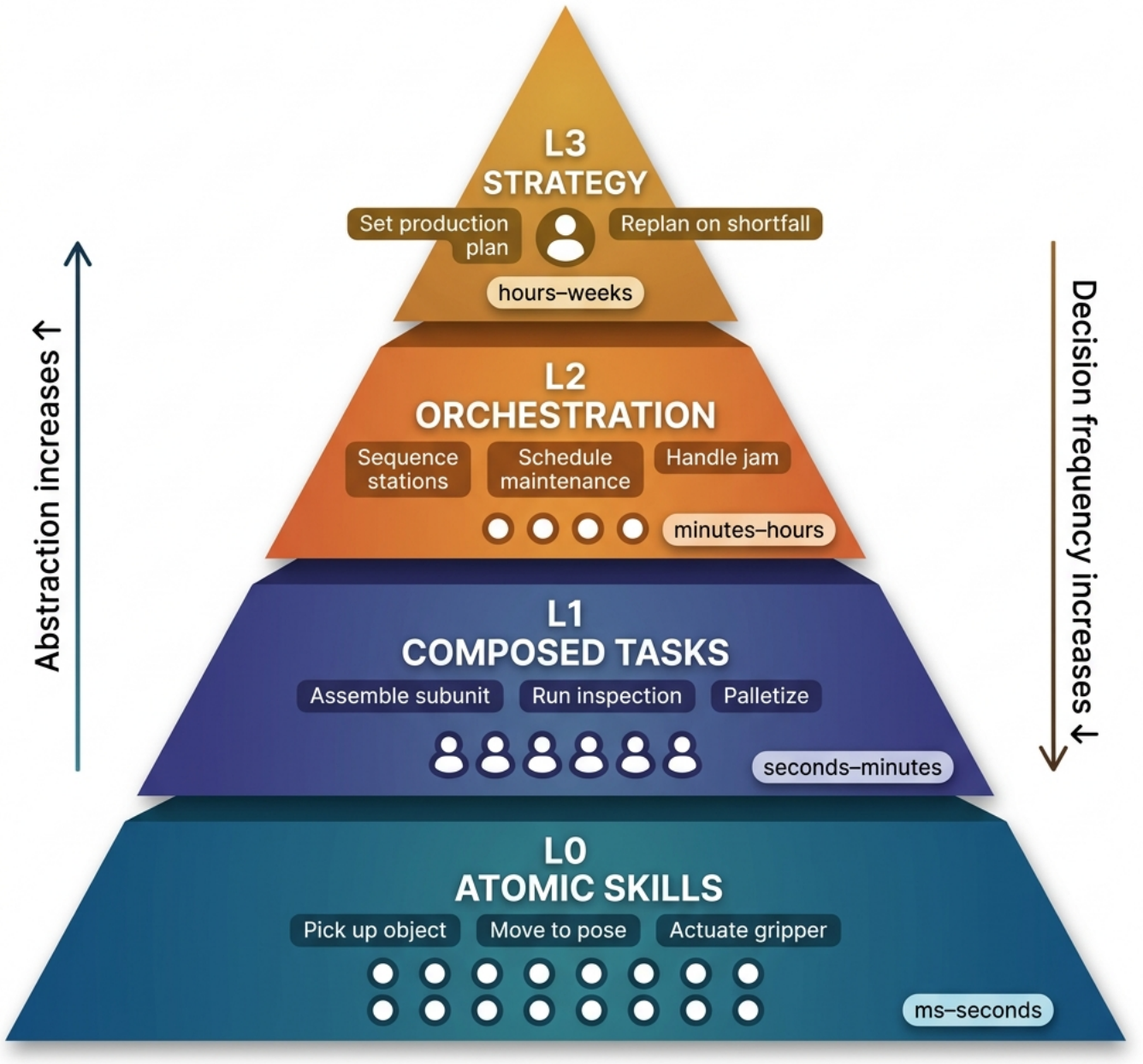

Figure 1. *The Agentic Pyramid. Four tiers - L0 Atomic Skills (ms–seconds), L1 Composed Tasks (seconds–minutes), L2 Orchestration (minutes–hours), L3 Strategy (hours–weeks) - narrow from many concrete agents at the base to few abstract agents at the apex. Abstraction increases upward; decision frequency increases downward.*

3. Intellectual Lineage: Why a Pyramid

The Agentic Pyramid is not a novel invention so much as a convergence. Four independent traditions arrived at hierarchical decomposition, and an engineer who knows them will design better pyramids. (A related insight from the prompting literature is that complex reasoning benefits from explicit decomposition into subproblems - Least-to-Most prompting achieves this within a single model by chaining subquestions (Zhou et al., 2023), while the pyramid achieves it across multiple models by chaining agents.)

Hierarchical Task Networks (HTN). Classical planning systems such as SHOP2 decompose a compound task into subtasks recursively, using *methods*, until only primitive operators remain (Erol et al., 1994; Nau et al., 2003). The idea traces to early work on planning across abstraction spaces (Sacerdoti, 1974). The decomposition tree *is* a concept hierarchy: compound tasks at the top, primitives at the leaves. An LLM agent that breaks "fulfill the order" into "pick, pack, ship" and then expands "pick"

into motor primitives is executing an HTN by another name.

Hierarchical and feudal reinforcement learning. The *options* framework formalizes temporally extended actions - closed-loop policies with an initiation condition and a termination condition - so that a higher policy can select an option and let it run for many low-level steps (Sutton et al., 1999). Feudal RL goes further: a *manager* sets goals in an abstract space and a *worker* is rewarded for achieving them, with the manager blind to low-level detail and the worker blind to the manager's ultimate objective (Dayan & Hinton, 1993; Vezhnevets et al., 2017). This is precisely the upward/downward information firewall the pyramid enforces. (For a modern synthesis of the field, see the survey by Pateria et al., 2021.)

Subsumption architecture. Brooks's layered robot control stacks competence layers, each able to connect sensing to action, with higher layers modulating or overriding lower ones (Brooks, 1986). A complementary lens from human-factors engineering is Rasmussen's skill–rule–knowledge taxonomy, which likewise separates fast sensorimotor behavior from slow knowledge-based reasoning (Rasmussen, 1983). The lesson the pyramid borrows is that **layers should be independently functional and loosely coupled**, communicating through thin, well-defined channels rather than a shared global state.

The ISA-95 automation pyramid. Industrial control has standardized this shape for decades (IEC 62264 / ANSI-ISA-95). Level 0 is the physical process; Level 1 is sensing and actuation; Level 2 is supervisory control (SCADA/HMI); Level 3 is manufacturing operations management (scheduling, dispatch, MES); Level 4 is enterprise planning (ERP). Each level has its own data, its own time horizon, and a defined interface to its neighbors. The Agentic Pyramid is, in one reading, **the ISA-95 pyramid with a reasoning agent installed at each level** - replacing fixed control logic with goal-directed deliberation while keeping the layered contract structure that already makes factories tractable.

The takeaway: we already build complex physical and computational systems as layered hierarchies because flat ones are unmanageable. Agentic AI is rediscovering the same necessity.

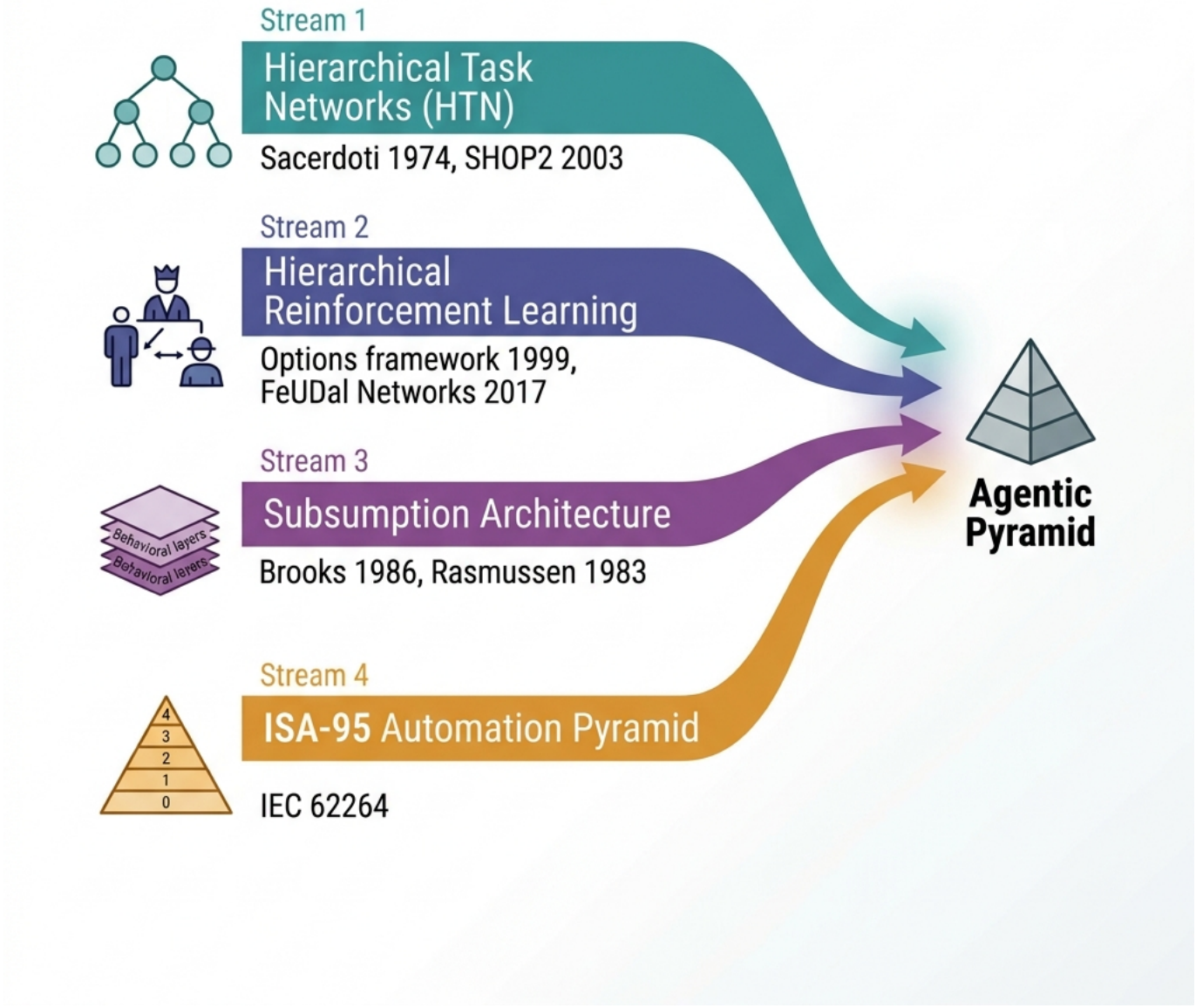

Figure 2. *Intellectual lineage. Four independent traditions - Hierarchical Task Networks, Hierarchical Reinforcement Learning, Subsumption Architecture, and the ISA-95 Automation Pyramid - converge on the same structural insight that the Agentic Pyramid codifies.*

4. Anatomy of a Level: What a Layer *Contains*

This is the heart of the architecture. An agentic level is not just "an agent with a system prompt." It is a self-contained competence package. Defining a level means specifying eight kinds of contents. The same eight slots appear at every altitude; only their granularity changes.

#	Content	What it holds	How it shifts as you go *up*
1	**Concept vocabulary (semantic graph)**	The ontology of the level - entities, relations, predicates - ideally represented as an explicit semantic/knowledge graph the agent can query and update.	Vocabulary becomes more abstract: "joint", "pose" → "subassembly", "station" → "line", "order" → "throughput", "margin".
2	**Skill library**	The callable competencies the level can invoke: the interface exposed by the level below, plus this level's own composed procedures. Parametric templates, code examples, reusable plans.	Skills become coarser and longer-running: a millisecond actuation becomes a multi-minute task becomes a multi-hour campaign.
3	**World / cost model**	Predictive and evaluative models at the level's timescale - physics and kinematics low down; queueing, scheduling, and throughput models mid; demand, cost, and risk models high.	The model's state variables aggregate; its prediction horizon lengthens.
4	**Decision memory**	Episodic record of prior decisions and outcomes - a case library of what was tried, what worked, what failed, and why. Provenance for every consequential choice.	Memory entries become rarer but heavier: fewer strategic decisions, each with larger consequences and longer feedback loops.
5	**Knowledge assets**	The documents, specifications, SOPs, manuals, instructions, and reference data the level needs. The "read-only" corpus the agent grounds itself in.	Documents shift from technical (datasheets, tolerances) to procedural (work instructions) to strategic (policies, contracts, objectives).
6	**Guardrails**	Hard constraints, safety envelopes, invariants, validation and verification rules, and explicit prohibitions. What the level must never do, and the checks that enforce it.	Guardrails shift from physical safety (collision, force limits) to operational (quality gates) to strategic (compliance, budget, risk limits).
7	**Interface contract**	The level's public surface: the skills it **exposes upward** (with preconditions, postconditions, and cost/latency estimates) and the primitives it **consumes downward**. The contract is the real, durable product of a level.	The exposed skills become higher-value and lower-frequency; the contract carries more semantic weight per call.
8	**Objective binding**	How goals arrive from above and how results report upward. The translation of an upper-level intent into this level's target, and the compression of this level's outcome into an upper-level abstraction.	Near the top, objectives originate from humans or the business rather than a higher agent.

Table 1. *The eight contents of an agentic level. The same eight slots appear at every altitude; only their granularity changes.*

A useful mental test for whether you have drawn level boundaries correctly: **each of the eight slots should be populated with materially different content at each level.** If two adjacent levels share the same vocabulary, the same guardrails, and the same world model, they are one level wearing two hats.

To make the interface contract concrete rather than abstract, here is what a representative L1 contract entry looks like as a typed specification:

```
Skill:          assemble(unit_id: UnitID, program: ProgramID) → AssemblyResult
Preconditions:  unit_id ∈ active_inventory
                station.status == IDLE
                fixtures.last_calibration < 24h ago
Postconditions: unit.status ∈ {ASSEMBLED, DEFECT}
                station.status == IDLE
                quality_record created for unit_id
Cost estimate:  $0.12–$0.18 per invocation
Latency:        45s ± 15s (p95: 62s)
Escalation:     if station.status == FAULTED → escalate(L2, FaultReport)
                if unit.defect_count > 2     → escalate(L2, QualityAlert)
```

This is the durable artifact. The agent behind the contract can be swapped - from a rule-based controller to a learned policy to an LLM - without changing a single line of the specification. The level above never knows or cares *how* the assembly happens, only that the contract is honored.

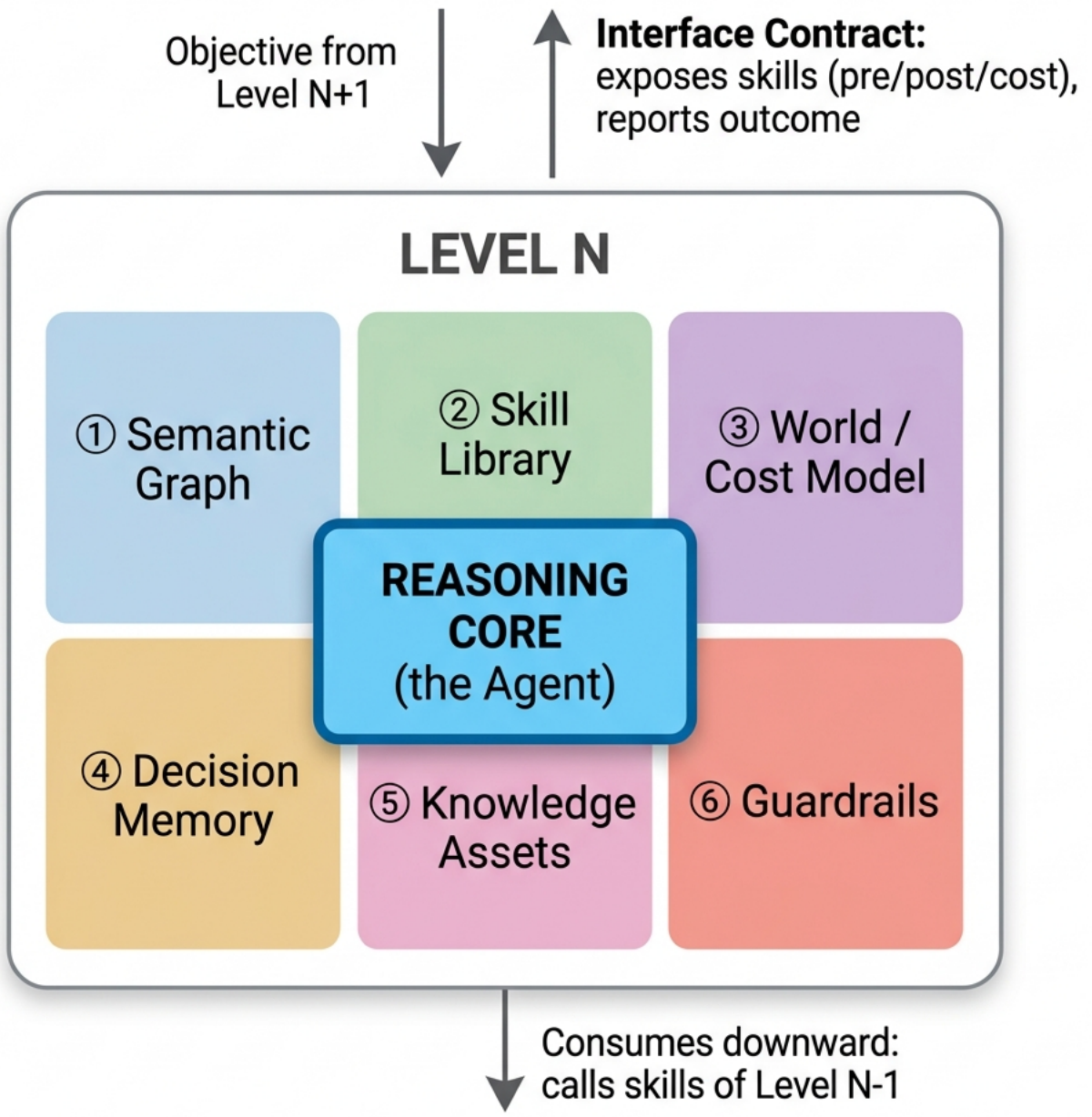

Figure 3. *Anatomy of a level. Each level is a self-contained competence package: six content tiles - Semantic Graph, Skill Library, World/Cost Model, Decision Memory, Knowledge Assets, Guardrails - surround the Reasoning Core (the agent). Objectives arrive from above via the Objective Binding; the level exposes its capabilities upward through the Interface Contract and consumes skills from the level below.*

5. Information Flow: Commands Down, Abstractions Up

The pyramid runs on two opposing streams.

Decomposition flows down. A goal at level N is translated into one or more subgoals at level N–1, which are themselves decomposed, until the stream reaches atomic skills that act on the world. "Hit the throughput target" becomes "balance the line to 30 units/hour," becomes "run station 4 on the high-speed program," becomes "move to pose, close gripper, weld."

Abstraction flows up. Raw, high-volume detail at the bottom is compressed at each level into the smaller, semantically richer state that the level above can reason about. Thousands of sensor readings become "station 4 nominal." A day of station states becomes "line OEE 87%." A quarter of line metrics becomes "we are 6% under plan."

This dual flow is also the architecture's **context firewall.** Detail does not leak upward; the strategy agent never sees gripper torque, because by the time information reaches it, torque has been abstracted away into "station healthy / degraded / down." This is what keeps each agent's context window populated with only level-appropriate information - the single most important practical benefit when the agents are LLMs with finite context.

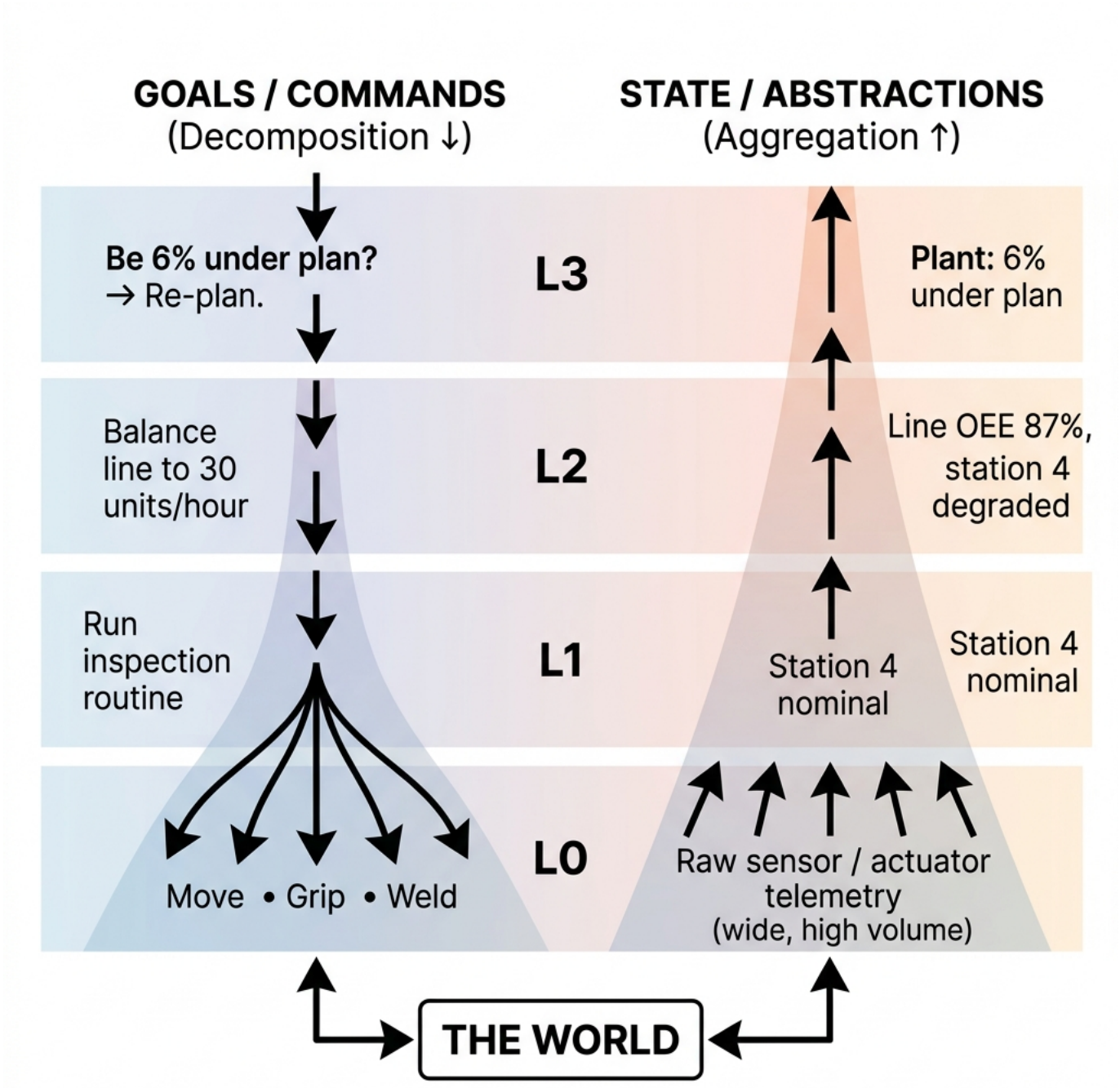

Figure 4. *Bidirectional information flow. Goals and commands decompose downward (left channel), fanning from one strategic objective into many atomic sub-commands. State and abstractions aggregate upward (right channel), compressing high-volume sensor telemetry into concise summaries. The funnel shapes emphasize the information volume asymmetry: narrow summaries at the top, wide data streams at the bottom.*

6. Worked Example - The Physical-AI Factory

This is the anchor example. We give all eight contents at each of four levels. The L3→L0 grounding problem - translating an abstract instruction into feasible low-level skills - is exactly what systems like SayCan address by pairing a language model's high-level knowledge with a library of pretrained robot skills (Ahn et al., 2022).

6.1 L0 - Atomic Skills

Vocabulary: poses, frames, joints, forces, grasp points, image regions. *Skills:* `move_to_pose`, `actuate_gripper(force)`, `run_weld_bead(path)`, `capture_image`. *World model:* kinematics, collision geometry, contact/friction physics. *Memory:* recent grasp successes/slips per object class. *Knowledge:* robot datasheets, joint limits, tool specs, calibration data. *Guardrails:* force ceilings, collision-avoidance envelopes, e-stop conditions, keep-out zones. *Contract (up):* exposes verbs like `pick(object_id) → grasped|failed` with success rates and cycle times. *Objective binding:* receives a single primitive command, returns success/failure plus telemetry. **Timescale: milliseconds to seconds.**

6.2 L1 - Composed Tasks

Vocabulary: parts, subassemblies, fixtures, programs, inspection results. *Skills:* `assemble_subunit`, `load_cnc`, `run_inspection_routine`, `palletize` - each a plan over L0 primitives. *World model:* task-level state machines, fixture occupancy, part-flow logic. *Memory:* which assembly sequences yield fewest reworks; failure modes per part. *Knowledge:* work instructions, assembly drawings, quality criteria, tolerances. *Guardrails:* sequence interlocks (don't machine an unclamped part), quality gates, tool-wear limits. *Contract (up):* exposes `assemble(unit) → ok|defect(reason)` with yield and takt time. *Objective binding:* receives "produce N of unit X," reports completion and quality. **Timescale: seconds to minutes.**

6.3 L2 - Cell / Line Orchestration

Vocabulary: stations, lines, buffers, jobs, queues, maintenance windows. *Skills:* `balance_line`, `sequence_stations`, `handle_jam`, `schedule_maintenance` - each composing L1 tasks across resources. *World model:* queueing and discrete-event models; bottleneck and buffer dynamics. *Memory:* historical bottleneck patterns, recovery playbooks for known stoppages. *Knowledge:* routings, station capabilities matrix, shift schedules, SLA targets. *Guardrails:* WIP caps, safety-stock floors, throughput-vs-quality tradeoff limits, labor rules. *Contract (up):* exposes `run_line(plan) → OEE, output, exceptions`. *Objective binding:* receives output and mix targets, reports OEE and exception summaries. **Timescale: minutes to hours.**

6.4 L3 - Strategic Factory Goals

Vocabulary: throughput targets, product mix, energy budget, demand, cost, capacity. *Skills:* `set_production_plan`, `reallocate_capacity`, `trade_energy_for_speed`, `replan_for_shortfall`. *World model:* demand forecasts, cost/margin models, capacity and energy-price models. *Memory:* outcomes of prior strategic plans; how the plant responded to past shocks. *Knowledge:* contracts, demand forecasts, sustainability policy, financial targets. *Guardrails:* budget ceilings, emissions caps, contractual delivery commitments, compliance. *Contract (up):* reports to human leadership; receives business objectives. *Objective binding:* objectives originate from humans/business; reports plan attainment and risk. **Timescale: hours to weeks.**

Level	Layer	Representative skills	Timescale
L3	**Strategy**	throughput targets · energy↔speed · replan on shortfall	hours–weeks
L2	**Orchestration**	balance line · sequence stations · handle jam · maintenance	minutes–hours
L1	**Composed Task**	assemble subunit · load CNC · run inspection · palletize	seconds–minutes
L0	**Atomic Skill**	move to pose · actuate gripper · weld bead · capture image	ms–seconds

Table 2. *Factory pyramid summary. Representative skills at each level with their characteristic timescales.*

6.5 Escalation in Action: A Walkthrough

To see the pyramid's escalation mechanism in motion, consider a concrete failure scenario that propagates from L0 to L3, showing the typed messages at each boundary and how the vocabulary shifts at every level.

L0 → L1. A robot arm at station 4 attempts `pick(part_0847)` and the gripper slips. L0 retries with adjusted force; the slip recurs. L0 reports upward through its contract: `pick(part_0847) → FAILED {reason: GRASP_SLIP, retries: 2, gripper_force: [12N, 15N]}`. Note the vocabulary: forces, retries, part IDs - all L0 concepts.

L1 receives and handles. The L1 composed-task agent - currently executing `assemble(unit_042)` - sees that its `pick` subtask has failed. L1 consults its decision memory: this part geometry has a 4% slip rate; the standard recovery is to re-seat the part in the fixture and retry with the alternate grasp point. L1 issues `re_seat(part_0847, fixture_B)` followed by `pick(part_0847, grasp_alt)`. This also fails. L1 has exhausted its recovery playbook. It escalates to L2, but in *L1's* vocabulary: `assemble(unit_042) → BLOCKED {reason: PART_UNPICKABLE, station: 4, estimated_delay: 8min}`. The gripper force details are gone - compressed away by the context firewall. L2 does not need them.

L2 receives and reroutes. The L2 line orchestrator sees station 4 blocked. It consults its world model (the queueing/bottleneck model) and identifies that station 7 has compatible tooling and is currently idle. L2 issues `reroute(unit_042, station_7)` and adjusts the line balance. But its model also shows that the buffer between stations 3 and 7 is at capacity - rerouting will stall the upstream flow. L2 can absorb a 4% throughput dip within its contract, so it reports normally. But if the blockage persists (a second unit fails the same way), L2's cumulative output forecast drops below its contracted target. L2 escalates to L3:
`run_line(plan_Q3_week12) → AT_RISK {output: −6%, cause: STATION_4_DEGRADED, options: [EXTEND_SHIFT, ACCEPT_SHORTFALL, EMERGENCY_MAINTEN`
The part IDs and station rerouting details are gone; L3 sees only throughput impact and decision options.

L3 decides. The strategy agent evaluates the three options against its cost model and contractual delivery commitments. Extending the shift costs $4,200 and recovers the 6%; accepting the shortfall risks a penalty clause worth $18,000; emergency maintenance takes station 4 offline for 2 hours but resolves the root cause. L3 selects

`emergency_maintenance(station_4)` and issues a revised plan to L2. The entire escalation - from a 12-newton gripper slip to a strategic maintenance decision - completed in under 90 seconds, with each level reasoning only in its own vocabulary and passing only decision-relevant information upward.

7. More Worked Examples

The same skeleton - the same eight contents, the same down-commands/up-abstractions flow - re-skins cleanly across domains. Three treatments follow, chosen for contrast.

7.1 Autonomous Driving (embodied, hard real-time)

Level	Vocabulary	Representative skills	Guardrails	Horizon
L3 Mission / fleet	trips, ETAs, fleet utilization	accept ride, rebalance fleet	service-area limits, regulatory geofences	minutes–hours
L2 Route	roads, intersections, traffic	plan route, choose corridor	legal route constraints, toll/HOV rules	minutes
L1 Maneuver	lanes, gaps, merges	change lane, merge, overtake, yield	right-of-way rules, safe-gap minimums	seconds
L0 Control	steering, throttle, brake	actuate steering/brake, hold lane	actuator limits, stability envelope, collision avoidance	milliseconds

Table 3. *Autonomous driving pyramid.*

The instructive contrast with the factory is **latency**. The lowest loops cannot afford a round trip to a large model. In driving, L0 (and often L1) are classical or learned controllers with hard real-time guarantees; the *agentic* deliberation lives at L2–L3. A correct pyramid lets you place a deliberative LLM agent only where its latency is affordable, and hand the fast loops to specialized controllers underneath the same contract.

7.2 Quantitative Trading Desk (fully abstract, no physics)

Level	Vocabulary	Representative skills	Guardrails	Horizon
L3 Mandate / risk	strategies, risk budget, drawdown	set risk budget, allocate to strategies	VaR limits, leverage caps, mandate compliance	days–quarters
L2 Portfolio	positions, exposures, hedges	rebalance, hedge, size positions	exposure limits, concentration rules	hours–days
L1 Position / order	orders, fills, schedules	manage a position, schedule a parent order	participation caps, no self-cross	minutes–hours
L0 Execution	child orders, venues, ticks	route child order, post/take liquidity	price collars, fat-finger checks, kill switch	ms–seconds

Table 4. *Quantitative trading desk pyramid.*

This pyramid has **no physical layer at all**, which makes the "concepts per level" claim especially crisp: an order is a primitive at L1 and an irrelevant detail at L3, where the only objects are strategies and risk. The world models differ sharply by level - microstructure and market-impact models at L0, factor and covariance models at L3.

7.3 Autonomous Software Engineering (familiar to the reader)

The software-engineering domain is the one most readers interact with daily, so we give it the full eight-content treatment at each level, matching the depth of the factory example in §6.

L0 - Atomic Edits

Vocabulary: functions, lines, symbols, AST nodes, tokens, diffs, diagnostics, lint errors. *Skills:* `edit_function`, `insert_lines`, `delete_block`, `rename_symbol`, `run_test(file)`, `run_linter`, `read_file`, `search_codebase` - each completing in seconds. *World model:* the syntax tree, type-checker state, symbol table, and dependency graph of the current file or module; a model of which edits break which tests. *Memory:* recent edit–test cycles - which edits caused which regressions, which refactorings succeeded on the first try, which patterns required multiple attempts. *Knowledge:* language references, API docs, style guides, linter rule definitions, framework documentation. *Guardrails:* the build must pass after every edit; no secret or credential may appear in source; diffs must be minimal and scoped to intent; no edits outside the authorized file set. *Contract (up):* exposes verbs like `apply_edit(spec) → success|failure(diagnostics)` with cycle-time and test-pass-rate metadata. *Objective binding:* receives a single, precisely scoped edit instruction ("add null check to `parse_input`"), returns the diff and test results. **Timescale: seconds to low minutes.**

L1 - Feature Implementation

Vocabulary: features, user stories, test suites, pull requests, branches, acceptance criteria, code-review comments. *Skills:* `implement_feature(spec)`, `write_tests(coverage_target)`, `open_pull_request`, `address_review_comments`, `run_integration_tests` - each a plan composed of many

L0 edits. *World model:* the feature-level dependency graph (which files must change, in what order); test-coverage maps; estimated effort per subtask; a model of likely review feedback based on project conventions. *Memory:* prior feature implementations - which decomposition strategies led to clean PRs, which produced excessive review rounds, common pitfalls per area of the codebase. *Knowledge:* work-item descriptions, acceptance criteria, design docs for the feature's subsystem, team coding conventions, test-writing guidelines. *Guardrails:* test coverage must not decrease; the PR must be atomic (one logical change); no public-API changes without design approval; CI pipeline must be green before marking the PR ready. *Contract (up):* exposes `deliver_feature(story) → PR_url | blocked(reason)` with estimated completion time and review-readiness status. *Objective binding:* receives a feature specification from L2, reports completion status, blockers, and test results. **Timescale: minutes to hours.**

L2 - Architectural Orchestration

Vocabulary: modules, services, APIs, contracts, migrations, dependency graphs, tech debt, design patterns. *Skills:* `design_module(spec)`, `define_api_contract`, `plan_migration(old → new)`, `decompose_epic_to_features`, `sequence_feature_work`, `resolve_cross-cutting_concern` - each composing L1 feature deliveries across module boundaries. *World model:* the system architecture graph (services, their interfaces, ownership, coupling); a cost model for migrations and refactors (estimated effort, risk of regressions, blast radius); tech-debt heat maps identifying high-maintenance areas. *Memory:* architectural decision records (ADRs) - past design choices, their rationale, and observed consequences; migration playbooks that succeeded or failed; coupling patterns that caused repeated integration pain. *Knowledge:* system architecture docs, API specifications, infrastructure runbooks, security and compliance policies, performance budgets, dependency-management policies. *Guardrails:* architectural invariants (no circular dependencies, services must own their data, public APIs require versioning); security policies (authentication on all external endpoints, no plaintext secrets); performance gates (latency budgets per service). *Contract (up):* exposes `execute_plan(epic) → delivered | partially_delivered(remaining, blockers)` with architectural-impact assessment. *Objective binding:* receives an epic or milestone from L3, decomposes it into features and sequencing constraints, reports progress and architectural risk. **Timescale: hours to days.**

L3 - Product Strategy

Vocabulary: roadmap items, releases, milestones, users, markets, KPIs, OKRs, competitive positioning, technical investment. *Skills:* `prioritize_roadmap(constraints)`, `define_milestone(scope, deadline)`, `allocate_engineering_budget`, `evaluate_build_vs_buy`, `replan_for_scope_change` - each a strategic deliberation that sets the agenda for L2. *World model:* product-market models (user growth, churn, feature adoption curves); engineering capacity and velocity forecasts; competitive landscape; risk/reward models for technical investments (e.g., paying down tech debt vs. shipping features). *Memory:* outcomes of prior roadmap decisions - which bets paid off, which features underperformed expectations, how accurate past effort estimates proved to be; post-mortems on missed deadlines and their root causes. *Knowledge:* product strategy documents, market research, customer feedback syntheses, investor and leadership directives, compliance and regulatory requirements, partnership agreements. *Guardrails:* budget ceilings, headcount constraints, compliance and legal requirements (GDPR, SOC 2, accessibility standards), scope limits to prevent feature creep, mandatory security and privacy reviews for user-facing changes. *Contract (up):* reports to human leadership; exposes `roadmap_status → progress, risks, recommendations`. *Objective binding:* objectives originate from human stakeholders and the business; the agent translates them into engineering milestones and reports plan attainment, risk, and strategic tradeoffs. **Timescale: days to weeks.**

The software-engineering pyramid makes an important structural point: the skill libraries at each level have fundamentally different *types of knowledge*. L0's library is concrete tool invocations and code patterns. L1's is feature-delivery playbooks and testing strategies. L2's is architectural patterns, migration recipes, and ADR templates. L3's is product strategy frameworks and prioritization rubrics. An ever-growing, executable skill library of exactly this kind is the central mechanism in Voyager (Wang et al., 2023), and modern coding agents like SWE-Agent (Yang et al., 2024) are beginning to operate across multiple levels of this hierarchy, though typically without explicit level boundaries. Multi-agent frameworks that assign role-specialized agents to a software pipeline, such as MetaGPT, are an early instantiation of this layering (Hong et al., 2024).

Other domains fit the same mold. Site reliability: restart pod → remediate service → manage deployment → uphold reliability/cost SLOs. Customer operations: answer a query → resolve a ticket → manage the queue → optimize CSAT-versus-cost strategy. In every case, the four levels differ in vocabulary, timescale, world model, and guardrails - the signature of a correctly drawn pyramid.

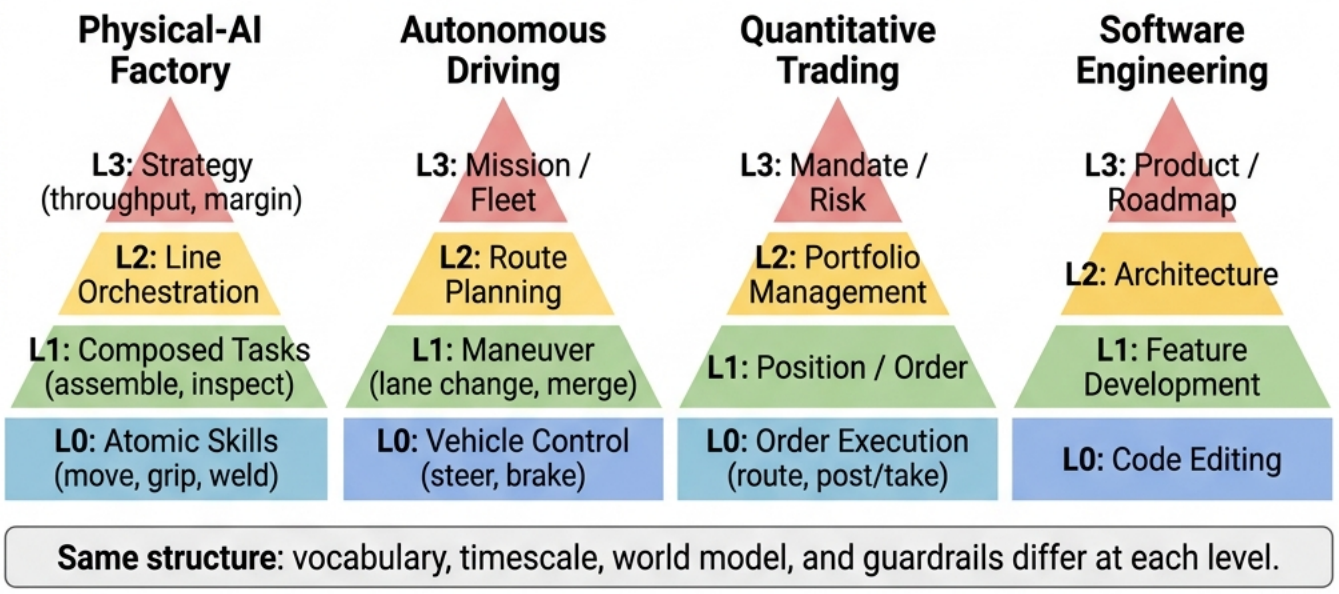

Figure 5. *The same pyramid skeleton re-skinned across four domains. The structural invariant - vocabulary, timescale, world model, and guardrails differ at each level - holds regardless of whether the domain involves atoms, bits, dollars, or code.*

8. Design Principles and Patterns

8.1 Core Principles

The interface contract is the product. Levels come and go; you may swap an L0 of hand-written controllers for an L0 of learned policies. What endures is the contract - the set of skills a level exposes with their preconditions, postconditions, and cost estimates. Design contracts first; design agents second.

Temporal abstraction is non-negotiable. Each level should close its loop at its own tempo. An upper level issues a goal and does not re-decide every tick; it re-decides when the lower level reports completion, failure, or a threshold breach. This is the options framework in practice and the main defense against thrash.

Memory and state are level-local. Each level keeps its own decision memory and semantic graph. Do not pool them into one store that every agent reads - that re-

creates the flat design's context bloat through the back door. Cross-level information moves only through the contract.

Replan at the right altitude. When something goes wrong, the question is *which level owns the recovery.* A jammed station is an L2 concern; a failed grasp is an L0/L1 concern; a structural demand shift is L3. Misrouting recovery upward turns small problems into strategic ones.

Escalate, don't broadcast. A level handles what it can within its contract and **escalates** only what exceeds its authority or competence - upward, to its parent, not sideways to peers. Escalation is a typed event ("cannot meet target, options are X/Y/Z"), not free-form chatter.

8.2 Human-in-the-Loop and Trust

The Agentic Pyramid does not replace humans; it gives them predictable places to stand. A flat agent forces the human to supervise everything or nothing - there is no middle ground, because there is no structure onto which to hang graduated oversight. A layered pyramid, by contrast, lets the human choose *an altitude* at which to engage, and lets the system earn autonomy one level at a time.

Override at every altitude. At each level, the human can observe, approve, correct, or take over - and the semantics of "override" change with the altitude:

- **L0 (Atomic Skills).** Override is physical: an e-stop, a joystick, a manual code edit. The human replaces the controller. At this level, override is rare in production but essential during commissioning and failure recovery.
- **L1 (Composed Tasks).** Override is procedural: the human approves, rejects, or modifies a plan before it executes. In a software-engineering pyramid, this is the pull-request review; in a factory, it is the operator confirming a rework sequence.
- **L2 (Orchestration).** Override is managerial: the human adjusts scheduling, reorders priorities, or vetoes a resource allocation. The system proposes; the human disposes.
- **L3 (Strategy).** Override is directorial: the human sets or changes objectives, budgets, and constraints. This is less "override" and more "governance" - the human is the source of objectives at the top of the pyramid, the origin of the downward command stream.

Observability and explainability per level. Each level should expose its reasoning in terms the human can audit *at that level's vocabulary.* L0 shows sensor traces and actuator commands; L1 shows task plans and step outcomes; L2 shows schedules, resource maps, and exception logs; L3 shows strategic tradeoff analyses and scenario comparisons. Forcing a human to read L0 telemetry to understand an L3 decision is a design failure - it means the upward abstraction pipeline is not doing its job. Explainability is therefore not a single feature bolted onto the system; it is a *by-product of the architecture.* Because each level already compresses its state into a level-appropriate summary for the level above, the same summary serves the human observer.

Trust calibration and deployment order. A practical deployment strategy follows the levels bottom-up: grant autonomy first where decisions are fast, narrow, and cheaply reversible (L0), and last where decisions are slow, broad, and consequential (L3). This mirrors how trust is built in human organizations - you promote after demonstrated competence at the current level, not before. A useful heuristic: **autonomy should be inversely proportional to blast radius.** L0 failures break a single action; L3 failures can break a quarter's plan. Design the system so that each level can run in one of three modes - *autonomous*, *human-on-the-loop* (human monitors and can intervene), or *human-in-the-loop* (human approves every decision) - and slide the dial per level as confidence grows.

Progressive autonomy. Over time, the system's decision memory accumulates evidence about per-level reliability. Use this evidence mechanically: if L1's contract-compliance rate exceeds a threshold over a window, propose to shift L1 from human-in-the-loop to human-on-the-loop. If the rate drops, shift back. This closed-loop trust calibration - a kind of meta-level control - is what turns a static architecture into one that learns how much rope to give itself, always subject to human governance at L3 (Bai et al., 2022; Huang et al., 2023).

8.3 Dynamic Reconfiguration

A deployed pyramid is not a monument; it is a running system that must adapt to changing workloads, failures, and complexity without a full redesign.

Runtime level addition and removal. Not every task requires all four levels. A simple, single-step request - "fix this lint warning" - should not be forced through a strategy layer. The system should be able to *instantiate only the levels it needs* for a given task and tear them down when the task completes. This is the agentic analogue of serverless scaling: pay for hierarchy only when hierarchy is useful. Conversely, a task that begins simple may grow complex enough to warrant an additional level. If an L1 agent discovers that the feature it is implementing requires cross-module changes, it escalates to an L2 that was not previously active. The key enabler is the interface contract: because each level's exposed surface is defined independently of whether the level above or below currently exists, levels can be plugged in or unplugged without breaking the architecture.

Horizontal scaling within a level. A single level may be staffed by multiple agents working in parallel - several L0 controllers running different tools, several L1 agents implementing separate features. Horizontal scaling within a level is straightforward precisely because agents at the same level share a common contract with the level above. The orchestrating level fans out goals and collects results. This is the natural locus for parallelism in the pyramid, and it does not violate the "communicate vertically, not horizontally" principle: the parallel agents share a coordinator, not a peer mesh.

Conflict resolution in parallel execution. Horizontal scaling raises an immediate question: what happens when two parallel agents at the same level interfere with each other? The answer is that **the orchestrating level above owns conflict resolution**, not the parallel agents themselves - this is precisely why the pyramid insists on vertical coordination rather than peer negotiation. A concrete example from the software-engineering domain makes the mechanism clear.

Suppose an L2 architectural orchestrator decomposes an epic into three features and fans them out to three parallel L1 agents: Agent-A implements a new authentication flow, Agent-B refactors the database access layer, and Agent-C adds a caching tier. All three features touch `UserService`, a shared dependency. Without coordination, their concurrent edits will produce merge conflicts, duplicated interface changes, or - worse - semantically incompatible modifications that each pass their own tests but fail when integrated.

The L2 orchestrator prevents this through three mechanisms that operate at *its* vocabulary (modules, APIs, dependency graphs), not at L1's vocabulary (functions, diffs, tests):

1. **Dependency-graph partitioning at dispatch time.** Before fanning out the three goals, L2 consults its semantic graph - the system architecture map - and identifies that all three features touch `UserService`. It has three options, chosen by policy: (a) *serialize* the conflicting features, scheduling Agent-B's refactor first because the others depend on the new interface; (b) *define an interface contract* for the shared module up front, freezing `UserService`'s public API so that each L1 agent codes against a stable target; or (c) *partition the module*, splitting `UserService` into `UserAuthService` and `UserDataService` before dispatch so that Agent-A and Agent-B operate on disjoint surfaces. The choice among these strategies is itself a decision recorded in L2's decision memory for future reuse.

2. **Resource locking and arbitration during execution.** Even with careful partitioning, runtime conflicts can emerge - Agent-A discovers it needs to add a method to an interface that Agent-B is also modifying. The L1 agent cannot resolve this; it lacks the architectural context to know whether its proposed change is compatible with Agent-B's work. Instead, it escalates to L2: "I need to modify `UserRepository.findByToken()`; is this surface locked?" L2 checks its coordination state, sees Agent-B's active scope, and either grants the modification (if compatible), queues it (if Agent-B is nearly done), or mediates by defining a merged interface change and issuing updated specifications to both agents. This arbitration is a normal part of L2's `sequence_feature_work` skill, not an exceptional recovery path.

3. **Integration verification before upward reporting.** When all three L1 agents report completion, L2 does not simply forward three "done" signals to L3. It runs an integration step at its own level: merging the three branches, running cross-feature tests, and checking for semantic conflicts (e.g., Agent-A assumed synchronous database calls that Agent-B's refactor made asynchronous). If integration fails, L2 owns the remediation - it may ask one L1 agent to rebase, or it may adjust the architectural plan and re-dispatch. Only after integration passes does L2 report `execute_plan(epic) → delivered` upward, compressing the parallel execution into a single contract-compliant result.

The pattern generalizes beyond software. In the factory pyramid, an L2 line orchestrator resolves resource conflicts between parallel L1 station tasks - two tasks that both need the same fixture, or that compete for buffer space. In trading, an L2 portfolio manager prevents two parallel L1 position agents from both selling the same holding. In every case, the principle is the same: **parallel agents at level N are blind to each other; conflict detection and resolution live at level N+1, where the shared resources are visible in the semantic graph.** This is another expression of the context firewall: each L1 agent sees only its own task, which keeps its context clean, while L2 sees the full resource map, which is exactly the information needed to arbitrate.

Graceful degradation when a level fails. If an L2 orchestrator becomes unresponsive, the system should not halt. Graceful degradation means that the level above (L3) detects the failure via its contract-monitoring guardrail, and either falls back to a simpler plan (e.g., issuing L1 tasks directly using a pre-approved playbook) or escalates to the human. Meanwhile, L1 and L0 continue executing any in-flight tasks autonomously. Because each level is independently functional - borrowing the subsumption-architecture lesson (Brooks, 1986) - partial failures do not propagate unconditionally.

Adaptive complexity: collapsing levels for simple tasks. The pyramid's overhead is justified only when abstraction levels are genuinely distinct. An adaptive system detects when a task's complexity does not span all levels and *collapses* intermediate layers. A strategy agent that receives a trivial single-step goal can skip orchestration and issue the goal directly to L1 or even L0, provided the contract supports it. This "short-circuit" path keeps the architecture honest: the pyramid is a ceiling, not a floor.

State lifecycle during level transitions. When a level is bypassed or torn down, its two stateful contents - the **decision memory** and the **semantic graph** - require explicit lifecycle management. The architecture defines three states for a level's persistent stores:

- *Dormant.* When a level is not instantiated (either never activated or torn down after use), its decision memory and semantic graph are persisted to durable storage but are not actively maintained. No new entries are written; no graph mutations occur. The stores are frozen at the state they held when the level was last active. This is the common case for an L2 orchestrator that is not needed for a simple task.
- *Active.* The level is instantiated and its agent reads from and writes to both stores in real time. This is normal operation.
- *Rehydrated.* When a previously dormant level is re-instantiated - for example, an L2 orchestrator spun up because a task that began as a simple L1 feature grew into a cross-module epic - its decision memory and semantic graph are loaded from durable storage. The rehydrated agent picks up where it left off, but must reconcile its potentially stale semantic graph against the current state of the levels that remained active during its absence. This reconciliation step is a defined phase of the level's startup contract: the rehydrated level queries the levels below for a current state summary and patches its graph before accepting new objectives.

The critical design rule is the **write-through principle for bypassed levels**: when a level is collapsed out of the active path, the outcomes of tasks that *would have* passed through it are still recorded in its decision memory as *bypass entries* - lightweight records noting the goal, the outcome, and the fact that the level was not consulted. This ensures that when the level is later rehydrated, its case library is not missing a gap of unrecorded history. Without this write-through, a rehydrated L2 would have no record of the architectural changes that occurred while it was dormant, and its decision memory would be misleadingly optimistic about its own historical coverage. The bypass-entry mechanism is cheap - it writes only the contract-level summary, not full internal state - and it preserves the invariant that **every level's decision memory is a complete chronicle of all goals that fell within its scope, whether or not it was active when they executed.**

---

9. Anti-Patterns and Failure Modes

9.1 Leaky Abstraction

Leaky abstraction. The strategy agent starts reasoning about gripper torque. The moment an upper level references a lower level's primitives, the firewall is breached and context bloat returns. Detect it by auditing whether each level's vocabulary stays within its own semantic graph.

9.2 Too Many Thin Levels

Too many thin levels. If adjacent levels share most of their eight contents, you have over-stratified. Each boundary costs a translation and a round trip; pay it only where the vocabulary genuinely changes.

9.3 The God Agent

The god agent. A single powerful agent quietly bypasses the hierarchy and reaches all the way down to primitives "because it's faster." This works in the demo and collapses in production, because it reintroduces every flat-design pathology under the disguise of a pyramid. The god-agent failure is insidious precisely because it often *improves* short-term performance - a frontier model that can see everything and act everywhere will outperform a disciplined pyramid on simple benchmarks - while silently destroying the properties (context isolation, fault containment, independent testability) that make the pyramid viable at scale.

Preventing the god agent requires **architectural enforcement**, not just policy. Five concrete mechanisms, drawn from standard software-engineering practice, make the hierarchy physically difficult to bypass:

1. **API gateway routing with adjacency enforcement.** Each level's skills are exposed through a gateway (or service mesh) that enforces a strict adjacency rule: a level-N agent can call only level-(N–1) endpoints. The L3 strategy agent's credentials literally cannot resolve L0 tool endpoints. This is the same principle as network segmentation in zero-trust security - the topology itself prevents unauthorized reach-through. In an LLM-agent framework, this translates to **scoped tool registries**: the tool list injected into each agent's system prompt or function-calling schema contains *only* the skills exposed by its immediate subordinate level, not the union of all tools in the system.

2. **Scoped tool provisioning per agent instance.** When an agent is instantiated, it receives a tool manifest that is generated from the contract of the level below - and *nothing else*. The L2 orchestrator's tool manifest contains `deliver_feature(story)`, `run_integration_tests(branch)`, and other L1-contract skills; it does not contain `edit_function`, `run_linter`, or any L0 primitive. If an L2 agent attempts to call an L0 tool, the call fails at the framework level before it reaches any model. This is the agent-framework analogue of the principle of least privilege: each agent is provisioned with exactly the capabilities its level requires, and the provisioning is performed by infrastructure, not by the agent itself.

3. **Call-graph auditing and anomaly detection.** Even with gateway enforcement, a determined god-agent pattern can emerge through indirect means - an L3 agent that encodes low-level instructions inside the goal string it passes to L2, effectively puppeteering the lower levels. Detect this by auditing the **call graph** in production: if the decomposition tree from a single L3 goal fans out to an unusually large number of L0 calls, or if L3's goal descriptions contain vocabulary from L0's semantic graph (function names, file paths, joint coordinates), the abstraction boundary is being violated through the payload. Automated vocabulary auditors - classifiers trained to flag out-of-level terms in inter-level messages - serve as a runtime guardrail against this subtler form of hierarchy bypass.

4. **Process and resource isolation.** For high-stakes deployments, run each level in a separate process, container, or (in physical-AI systems) on separate hardware. L3 runs on a cloud instance with access to strategic databases; L0 runs on edge controllers with access to actuators. The network topology physically prevents L3 from issuing motor commands. This mirrors the ISA-95 automation pyramid's practice of placing different levels on separate network segments (the Purdue model), and it provides defense in depth: even if a software-level access control is misconfigured, the network architecture prevents cross-level reach-through.

5. **Contract-only integration testing.** The test suite itself should enforce the hierarchy. Integration tests for level N should mock level N–1 through its contract interface, never by directly manipulating N–1's internals. If the only way to test L2 is to give it direct access to L0's tools, the architecture has a coupling defect. This testing discipline catches god-agent tendencies during development, before they reach production, and it doubles as documentation of the intended inter-level boundaries.

The common thread is that **the hierarchy must be enforced by the platform, not by the agents' good behavior.** An agent that is merely *instructed* not to bypass levels will eventually do so when the instruction conflicts with its objective. An agent that *cannot* bypass levels - because the tools are not in its manifest, the endpoints are not routable, and the test suite would break - is structurally constrained regardless of what its reasoning suggests.

9.4 Objective Mismatch Between Levels

Objective mismatch between levels. The classic hierarchical-RL failure: the manager's goal and the worker's reward are not aligned, so the worker optimizes something the manager didn't want. In agent terms, the downward objective binding must faithfully translate intent, and the upward report must faithfully reflect outcome - otherwise levels optimize at cross-purposes.

9.5 Lossy Upward Abstraction

Lossy upward abstraction. If the summarization that flows up discards the wrong detail, upper levels reason on a distorted world. The aggregation functions are as safety-critical as the guardrails.

9.6 Chatty Interfaces

Chatty interfaces. Levels exchange too much information per interaction, defeating the purpose of the context firewall. This is distinct from leaky abstraction, which concerns the *wrong* information leaking; chatty interfaces concern the *volume*. A level that dumps its full internal state upward on every report is obeying the abstraction boundary in principle - it never references the wrong concepts - but is still flooding the upper level's context with unnecessary detail. The fix is to define the upward report as a fixed schema with bounded cardinality, and to treat any schema growth as a design smell.

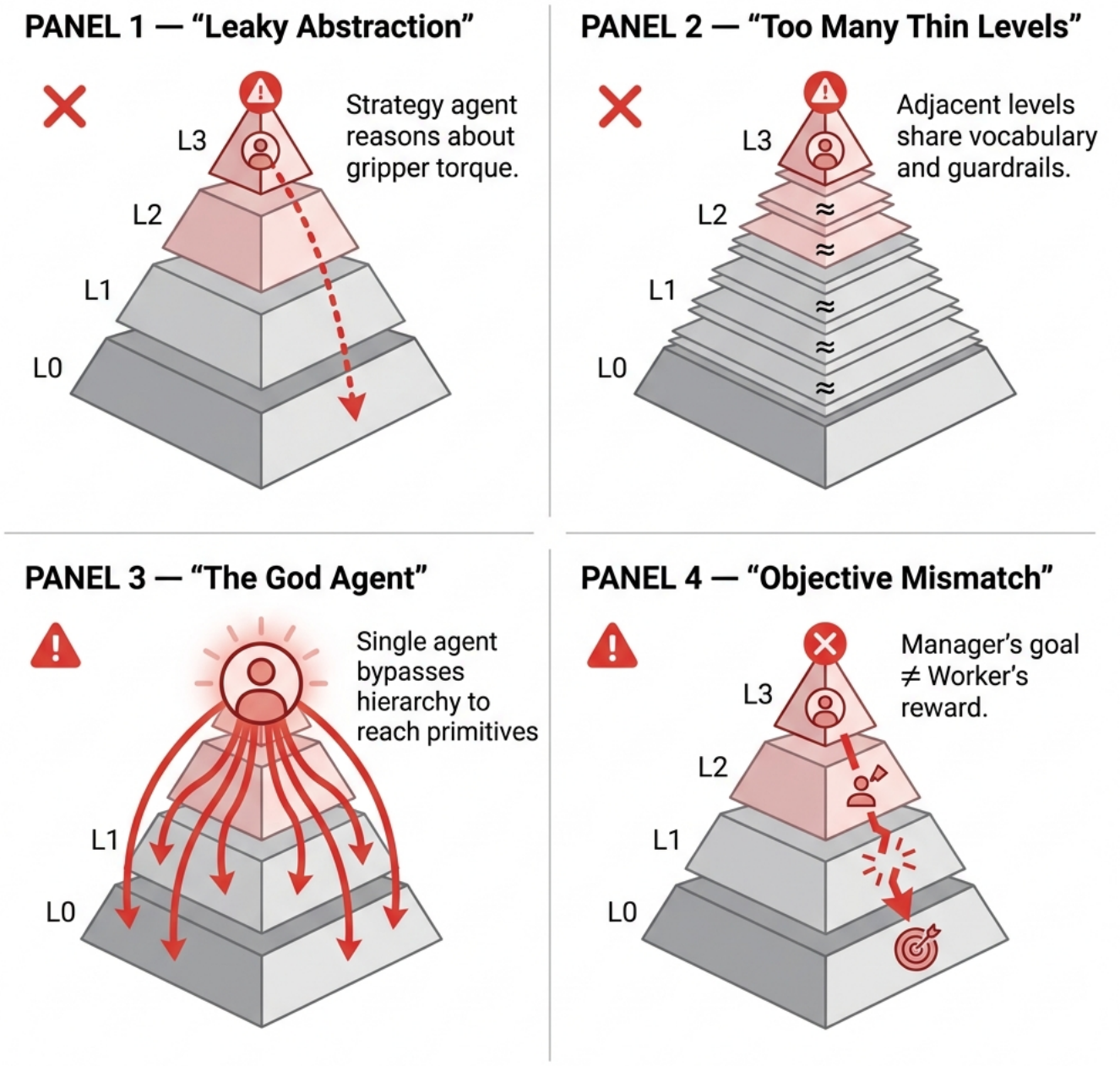

Figure 6. *Anti-patterns and failure modes. Four common ways a pyramid degrades: (1) Leaky Abstraction - upper levels reference lower-level primitives; (2) Too Many Thin Levels - adjacent levels share vocabulary; (3) The God Agent - a single agent bypasses the hierarchy; (4) Objective Mismatch - manager goals and worker rewards diverge.*

9.7 Adversarial Threats and Trust Boundaries

The anti-patterns above are *internal* failure modes - ways the system defeats itself. Production systems must also contend with *external* adversarial threats. The pyramid's layered structure provides natural defense-in-depth, but only if the boundaries are treated as **trust boundaries**, not just abstraction boundaries.

Prompt injection across levels. If a lower-level agent processes untrusted input (user-supplied code, external API responses, sensor data from an uncontrolled environment), a crafted payload could attempt to hijack the agent's reasoning and forge an upward report - for example, an L0 agent manipulated into reporting "station healthy" when the station is faulted. Defense: inter-level messages should be **schema-validated** against the contract's declared output type. Any message that fails schema validation is rejected at the boundary, not passed through.

Escalation forging. A compromised lower-level agent could fabricate escalation events to manipulate upper-level decisions - triggering an unnecessary strategy change

by reporting fictitious failures. Defense: escalation events should carry **provenance metadata** (originating level, timestamp, causal chain) that the receiving level can audit against its own observations. An L2 that receives an escalation from L1 claiming "station 4 blocked" can cross-check against its own telemetry summary before acting.

Blast-radius containment. The pyramid's architecture already limits the blast radius of a compromised agent: an L0 agent with scoped tool provisioning (§9.3, guardrail #2) can damage only its own atomic actions, not the production plan. This containment is a security property, not just a reliability property, and it should be explicitly valued as such in threat modeling.

10. Implementation Notes for LLM-Based Agents

10.1 Model Selection and Integration

Match model capability to altitude. Use small, fast, cheap models (or even non-LLM controllers) at L0 where decisions are frequent, narrow, and latency-bound; reserve frontier-scale reasoning models for L2–L3 where decisions are rare, broad, and consequential. The pyramid is also a cost-optimization structure: most calls happen at the cheap base.

Skills are tools or sub-agent calls. A level's downward consumption is implemented either as tool calls (when the lower level is deterministic) or as sub-agent invocations (when the lower level itself deliberates). The reason-then-act loop within a single level can follow patterns like ReAct (Yao et al., 2023), while the cross-level handoffs resemble the conversational orchestration of multi-agent frameworks such as AutoGen (Wu et al., 2023), CAMEL (Li et al., 2023), and LangGraph (LangChain, 2024). The contract is identical in both cases, which is what lets you replace one with the other.

The semantic graph is real infrastructure. Give each level an explicit, queryable representation of its concepts - a knowledge graph, a typed state object, a schema - rather than leaving the ontology implicit in prose prompts. Explicit semantics make guardrail-checking and upward-abstraction mechanical instead of hopeful.

Memory architecture. As established in §8.1, memory and state are level-local. For LLM-based implementations, the decision-memory slot benefits specifically from retrieve-reflect-plan memory architectures demonstrated for long-lived agents (Park et al., 2023), kept separate per level so that retrieval stays level-appropriate and the context firewall is preserved.

Upward summaries as engineered contracts. The aggregation step (§5) deserves the same care as a public API: define exactly what each level reports up, in what schema, and with what bounded cardinality. This is where the context firewall is enforced in practice.

10.2 Learning, Adaptation, and Skill Promotion

The pyramid as described so far is largely static: levels are defined, contracts are specified, and agents execute. But a production system should **learn across deployments**, and the pyramid's structure channels that learning naturally.

Skill compilation (downward promotion). When a composed procedure at level N is executed frequently with high reliability, it becomes a candidate for *compilation* into a primitive at level N–1. In the software-engineering pyramid, an L1 "add unit test for function" playbook that succeeds 98% of the time with a fixed decomposition can be promoted to an L0 atomic skill - a deterministic script or fine-tuned small model that executes the pattern without deliberation. This is the agentic analogue of JIT compilation: hot paths are hardened into faster, cheaper execution units. Voyager's ever-growing skill library (Wang et al., 2023) is an early demonstration of this principle within a single agent; the pyramid extends it across agents.

Decomposition learning (within a level). Each level's decision memory accumulates evidence about which decomposition strategies succeed and which fail. Over time, L2 learns that serializing features touching a shared module (§8.3, conflict resolution strategy (a)) works better for tightly-coupled modules, while interface-freezing (strategy (b)) works better for loosely-coupled ones. This learning is local to L2's decision memory and does not leak across levels - preserving the context firewall while still improving.

Upward promotion (error-handling enrichment). Conversely, when an L0 primitive proves unreliable for certain input classes, it may be "promoted" upward: the unreliable primitive is wrapped in an L1 composed task that includes error-handling logic, retry strategies, and fallback paths. The L0 skill is not removed; it remains available for the common case. But for the known-difficult cases, L1 interposes a more deliberate procedure. This is how the pyramid heals its weakest links without a full redesign.

10.3 Evaluation Methodology

Evaluating an agentic pyramid requires metrics at two granularities: **per-level metrics** that test each layer against its own contract, and **cross-level metrics** that test the integrated stack against end-to-end goals. Both are necessary; neither alone is sufficient.

Per-level metrics. Each level is a self-contained competence unit with a defined interface contract. Evaluate it the way you would evaluate any service - against its SLA:

Metric	What it measures	Level-specific examples
Latency	Time from goal receipt to result delivery	L0: < 100 ms per primitive. L1: < 60 s per task. L2: < 5 min per plan cycle. L3: < 1 h per strategic decision.
Decision quality	Correctness of the level's output relative to a ground-truth or expert judgment	L0: action success rate. L1: task completion with quality gates passed. L2: schedule optimality vs. offline solver. L3: goal attainment over a planning horizon.
Contract compliance	Fraction of calls where preconditions, postconditions, and cost/latency estimates are honored	All levels: measured as percentage of calls within the contract's advertised bounds.
Escalation frequency	How often the level cannot handle a situation within its authority and escalates upward	L0–L2: should be low (< 5%) under normal conditions. High escalation rates signal misdrawn level boundaries.
Guardrail violation rate	Frequency of hard-constraint breaches	All levels: target is zero. Any non-zero rate triggers immediate review.

Table 5. *Per-level evaluation metrics. Each level is tested against its own contract SLA.*

Per-level evaluation should be possible in isolation, using mock implementations of the adjacent levels - a simulated L0 when testing L1, a scripted L2 when testing L1 from above. This is the layered-testing principle: **localize failures before they compound.**

Cross-level metrics. The pyramid exists to achieve goals that span all levels. Cross-level metrics test the stack as a whole:

- **End-to-end goal attainment.** Given a strategic objective (L3), does the system achieve it within the specified constraints? This is the ultimate pass/fail, but it is slow to measure and hard to attribute when it fails.
- **Abstraction fidelity.** Does the information that flows upward faithfully represent the state of the level below? Measure by comparing the compressed summary at level N+1 against the full state at level N; the summary should preserve every fact that is decision-relevant at N+1 and discard everything else.
- **Decomposition correctness.** Does the goal decomposition flowing downward produce subgoals that, when all achieved, satisfy the parent goal? This tests the downward translation logic at each boundary.
- **Recovery time.** When a level fails and escalates, how quickly does the system return to nominal operation? This measures the escalation and replanning machinery.

Comparison against flat baselines. A pyramid justifies its complexity only if it outperforms a flat agent on the same task. The fair comparison gives the flat agent the union of all tools, knowledge, and context that the pyramid distributes across levels, and measures the same end-to-end metrics. In practice, flat agents degrade earlier on tasks with multiple timescales, larger tool sets, and longer horizons - precisely the conditions the pyramid is designed for. The crossover point - the task complexity at which the pyramid overtakes the flat agent - is itself a useful metric and should be reported for any empirical evaluation.

10.4 Cost and Latency Analysis

A key practical advantage of the pyramid is that it is also a **cost-optimization structure**. Because decision frequency decreases exponentially as you move up the hierarchy, and because lower levels can be served by smaller, cheaper models, the bulk of the compute budget is spent at the cheapest tier.

Level	Typical Model Size	Decision Frequency	Approx. Cost / 1K Calls	Latency Budget	Example
L0 Atomic Skills	Small / fine-tuned (≤ 8B params) or non-LLM controller	100–10,000 / hour	$0.01–$0.10	< 200 ms	Code edit, actuator command, order route
L1 Composed Tasks	Mid-size (8B–70B) or capable general-purpose model	10–100 / hour	$0.10–$1.00	< 60 s	Feature implementation, assembly sequence, position management
L2 Orchestration	Frontier reasoning model	1–10 / hour	$1.00–$10.00	< 5 min	Sprint planning, line balancing, portfolio rebalance
L3 Strategy	Frontier reasoning model + extended thinking	< 1 / hour	$5.00–$50.00	< 1 hour	Roadmap prioritization, production replanning, risk-budget setting

Table 6. *Cost and latency by pyramid level. Most compute happens at the cheap base - the inverted cost pyramid.*

The key insight is the **inverted cost pyramid**: although L3 decisions are individually expensive, they are so infrequent that their total cost is negligible compared to L0. In a typical software-engineering deployment, L0 might process 5,000 edit–test cycles per day at $0.05 each ($250/day), while L3 makes one or two strategic decisions at $25 each ($50/day). The total system cost is dominated by the base - and the base is where the cheapest models live.

Latency budgets compound vertically: an L2 decision that decomposes into ten L1 tasks, each decomposing into ten L0 actions, must fit within L2's total latency budget. This imposes a discipline on how deep the decomposition tree can grow and how much parallelism each level must exploit - a practical constraint that keeps the architecture honest about whether additional levels are worth their round-trip cost.

11. When *Not* to Use a Pyramid

The pyramid earns its overhead only when abstraction levels are genuinely distinct. Skip it when:

- **The problem is flat.** A task with one timescale and one vocabulary does not need layering; a single agent is simpler and faster.
- **Latency forbids round trips.** If even the highest-value decision must be made in milliseconds end to end, the inter-level handoffs may be unaffordable, and a single tight controller wins.
- **One capable agent already suffices.** If a single model handles the whole task within its context budget and reliability bar, the pyramid is premature optimization. Introduce levels when - and only when - you can name a concept boundary that a flat design is fighting against.

The honest test: *can you state, in one sentence each, the distinct vocabulary of every level you propose?* If you cannot, you do not yet have a pyramid - you have one agent and some wishful org-charting.

For a quantitative sanity check, three rules of thumb help:

- **Timescale separation.** If the decision frequencies at two proposed levels differ by less than one order of magnitude (e.g., both operate at 1–5 decisions per minute), they are probably one level. Genuine pyramid levels typically differ by 10× or more in decision frequency.
- **Context budget.** If the total tool-set and knowledge-asset inventory fits in a single model's context without degradation - typically fewer than ~30 tools and ~50K tokens of reference material - a flat agent may handle it. The pyramid earns its keep when the union of all tools and knowledge exceeds any single model's effective context.
- **Round-trip overhead.** If the inter-level handoff latency exceeds 10% of the end-to-end latency budget, the level boundary is too expensive. Each boundary must be cheap relative to the work it coordinates.

12. Conclusion

The Agentic Pyramid is, in the end, an org chart for machine cognition. It takes the oldest lesson of complex-systems engineering - that scale is managed through layered abstraction with narrow interfaces - and applies it to agents whose substrate happens to be language models. Each level is a specialist defined by eight concrete contents: a semantic graph, a skill library, a world model, a decision memory, knowledge assets, guardrails, an interface contract, and an objective binding. Commands flow down as decomposition; abstractions flow up as compression; and the boundary between levels is the firewall that keeps each agent reasoning in clean, level-appropriate concepts.

Making this pattern production-grade requires more than a clean decomposition. It requires state-lifecycle management so that dynamically reconfigured levels maintain coherent memory. It requires typed conflict resolution so that parallel agents at the same level cannot silently interfere with each other. It requires programmatic enforcement - scoped tool provisioning, API gateway adjacency rules, call-graph auditing - so that the hierarchy is maintained by the platform, not by the agents' good behavior. And it requires per-level evaluation that localizes failures before they compound, together with cost and latency discipline that keeps the inverted cost pyramid honest. The tools exist; the discipline is in applying them layer by layer.

One important boundary of the current framework deserves acknowledgment: this paper treats a single pyramid in isolation. Real enterprises may deploy **multiple pyramids** that need to coordinate - a manufacturing pyramid and a supply-chain pyramid, a frontend engineering pyramid and a backend engineering pyramid. Cross-pyramid coordination is a natural extension: it happens at the highest level (L3↔L3) through an explicit integration contract, creating a federation layer. Exploring the design patterns for this federation - and the meta-level governance it requires - is a clear direction for future work.

Flat agents pretend abstraction levels don't exist. Real factories, real vehicles, real trading desks, and real codebases insist that they do. The pyramid simply takes them at their word.

Appendix A: Practitioner's Checklist - Building Your First Pyramid

The following step-by-step guide is intended for engineers beginning to apply the Agentic Pyramid pattern to a new domain.

1. **Identify the timescales.** List the distinct decision frequencies in your system, from fastest to slowest. If you find three or more timescales that differ by at least an order of magnitude, you likely have a real pyramid. If the timescales cluster together, a flat design may suffice - revisit §11.

2. **Name the vocabulary at each timescale.** For each timescale, write down the entities, relations, and verbs that are first-class. "Functions, diffs, test results" is a different vocabulary from "modules, APIs, migrations." If two timescales share a vocabulary, they are one level.

3. **Draw the level boundaries.** Group timescales into levels (typically 3–5). Apply the eight-content test from §4: can you populate the semantic graph, skill library, world model, decision memory, knowledge assets, guardrails, interface contract, and objective binding with *materially different* content at each level? If not, merge levels until you can.

4. **Define the interface contracts first.** For each level, specify the skills it exposes upward: name, parameters, preconditions, postconditions, expected cost, and expected latency. The contract is the durable product - design it before you design the agent.

5. **Implement bottom-up.** Start at L0. Build the atomic skills, wrap them in their contract, and test them in isolation. Then build L1 on top of the real L0, not a mock. The

pyramid earns trust the same way it earns autonomy - from the base upward.

6. **Choose models by altitude.** Assign the smallest model (or non-LLM controller) that meets each level's latency and capability requirements. Use the cost/latency table in §10.3 as a starting template. Reserve frontier reasoning models for L2 and above.
7. **Engineer the upward summaries.** For each level boundary, define exactly what information crosses upward: what schema, what compression, what is discarded. This is your context firewall. Test it by asking: "Could the upper level make a correct decision using *only* this summary?"
8. **Wire the escalation paths.** Define the conditions under which each level escalates to the level above: unmet preconditions, exceeded cost/latency budgets, guardrail near-misses, repeated failures. Escalation should be typed and structured so the upper level can handle it programmatically.
9. **Add guardrails and observability.** Populate each level's guardrail slot with hard constraints (invariants that must never be violated) and soft constraints (thresholds that trigger alerts or escalation). Instrument each level to emit structured logs keyed to its own vocabulary - these logs are your explainability layer (§8.2).
10. **Evaluate per level, then end-to-end.** Test each level against its contract using mocked adjacent levels. Then test the integrated stack with cross-level metrics (§10.2). Track the per-level metrics table continuously in production; regressions in one level will show up as escalation spikes at the level above.

List of Figures

List of Tables

References

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., … Zeng, A. (2022). *Do As I Can, Not As I Say: Grounding Language in Robotic Affordances.* Conference on Robot Learning (CoRL). arXiv:2204.01691.

Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., … Kaplan, J. (2022). *Constitutional AI: Harmlessness from AI Feedback.* arXiv:2212.08073.

Brooks, R. A. (1986). A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation, 2*(1), 14–23.

Dayan, P., & Hinton, G. E. (1993). Feudal Reinforcement Learning. *Advances in Neural Information Processing Systems (NIPS) 5*, 271–278.

Erol, K., Hendler, J., & Nau, D. S. (1994). HTN Planning: Complexity and Expressivity. *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1123–1128.

Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., … Wu, C. (2024). *MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework.* International Conference on Learning Representations (ICLR). arXiv:2308.00352.

Huang, W., Xia, F., Xiao, T., Chan, H., Liang, J., Florence, P., … Hausman, K. (2023). *Inner Monologue: Embodied Reasoning through Planning with Language Models.*

Conference on Robot Learning (CoRL). arXiv:2207.05608.

International Electrotechnical Commission. (2013). *IEC 62264: Enterprise-Control System Integration* (the international standard underlying the ANSI/ISA-95 automation pyramid). Geneva: IEC.

LangChain. (2024). *LangGraph: Multi-Actor Applications with LLMs.* https://github.com/langchain-ai/langgraph.

Li, G., Hammoud, H. A. A. K., Itani, H., Khizbullin, D., & Ghanem, B. (2023). *CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society.* Advances in Neural Information Processing Systems (NeurIPS). arXiv:2303.17760.

Nau, D. S., Au, T. C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., & Yaman, F. (2003). SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research, 20*, 379–404. DOI:10.1613/jair.1141.

Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). *Generative Agents: Interactive Simulacra of Human Behavior.* ACM Symposium on User Interface Software and Technology (UIST). arXiv:2304.03442.

Pateria, S., Subagdja, B., Tan, A.-H., & Quek, C. (2021). Hierarchical Reinforcement Learning: A Comprehensive Survey. *ACM Computing Surveys, 54*(5), Article 109.

Rasmussen, J. (1983). Skills, Rules, and Knowledge; Signals, Signs, and Symbols, and Other Distinctions in Human Performance Models. *IEEE Transactions on Systems, Man, and Cybernetics, SMC-13*(3), 257–266.

Sacerdoti, E. D. (1974). Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence, 5*(2), 115–135.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence, 112*(1–2), 181–211.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., & Kavukcuoglu, K. (2017). *FeUdal Networks for Hierarchical Reinforcement Learning.* International Conference on Machine Learning (ICML). arXiv:1703.01161.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). *Voyager: An Open-Ended Embodied Agent with Large Language Models.* arXiv:2305.16291.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., … Wang, C. (2023). *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation.* arXiv:2308.08155.

Yang, J., Jimenez, C. E., Wettig, A., Liber, K., Narasimhan, K., & Press, O. (2024). *SWE-Agent: Agent-Computer Interfaces Enable Automated Software Engineering.* arXiv:2405.15793.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models.* International Conference on Learning Representations (ICLR). arXiv:2210.03629.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., … Chi, E. (2023). *Least-to-Most Prompting Enables Complex Reasoning in Large Language Models.* International Conference on Learning Representations (ICLR). arXiv:2205.10625.