ZeroPaper: An Autonomous Research System

Alejandro Lopez-Lira

May 1, 2026

Abstract

ZeroPaper is an autonomous system that takes a research domain as input and produces a publication-candidate manuscript as output, with no human intervention between launch and completion. The system runs ten numbered stages and six adversarial gates, coordinates roughly thirty specialized agents, and produces finished LaTeX for economics at roughly two dollars per paper amortized under a flat-fee Claude Code Max subscription (approximately zero marginal cost per additional paper). From ten premises about how LLMs behave and what they cost, we derive six design principles for long-running autonomous systems: keep state and control flow outside the model, spend context carefully, delegate aggressively, verify rather than trust, make termination mechanical, and run independent work in parallel. ZeroPaper runs on Claude Code, OpenAI Codex, and Gemini CLI, with variants for finance and macro and extensions for empirical analysis and LLM experiments.

Keywords: autonomous research, AI for science, LLM agents, adversarial evaluation, system design, Markov systems

1 Introduction

Top-five economics journals tripled in paper length and saw acceptance rates fall from 15% to 6% between 1970 and 2012 [Card and DellaVigna, 2013], and across science the number of researchers required to sustain Moore's-Law-grade progress has grown roughly eighteen-fold since the early 1970s [Bloom et al., 2020]. The diagnosis is robust: a unit of research progress costs more inputs than it used to. Generative AI is the most natural new input to put against that constraint, and the question is whether it can be marshaled into a pipeline that produces a finished, refereeable paper without supervision.

This paper describes one such pipeline. ZeroPaper is an autonomous large-language-model system that, given a research domain and a set of target journals, runs end-to-end from

literature survey to typeset LaTeX, coordinating roughly thirty specialized agents through ten sequential stages and six adversarial gates, under any of three host runtimes (Claude Code, OpenAI Codex CLI, or Gemini CLI). Under a single Claude Code Max subscription, it finishes roughly one hundred papers per month at around two dollars per paper on average. The paper makes two contributions. The first is the system itself, walked through in Section 4. The second is the design discipline that makes a pipeline of this length terminate without drift: ten premises about how LLMs behave and what they cost (Section 2) and six principles that follow from those premises (Section 3).

The principles are not stylistic preferences. Long-running autonomous LLM pipelines fail in characteristic ways: self-bias, long-context degradation, coherence drift, stochastic error, and a path-of-least-resistance bias toward declaring premature completion. A pipeline that ignores these fails silently or loops forever. ZeroPaper does not because three architectural choices, each forced by one or more principles, neutralize the failure modes that would otherwise dominate. Every substantive advance is gated by an adversarial agent that has not seen the generator's reasoning, so self-bias has no surface to attach to. Every loop has a mechanical termination predicate, so the orchestrator cannot rationalize "one more try" indefinitely. Every gate runs at least two evaluators with distinct framings (a structured re-derivation paired with a free-form skeptical reader at math gates, three referees with different priors at the publication gate), so a single evaluator's blind spot or noisy verdict does not propagate. The pipeline also runs under any of three host runtimes (Claude Code, OpenAI Codex, Gemini CLI), so the architecture is not tied to any one vendor's model or session-discipline conventions. Section 5 situates these choices relative to existing autonomous-pipeline systems and to the closest econ-domain predecessors, including Korinek [2025] on agentic LLM systems for economic research and the companion paper Lopez-Lira and Seyfi [2026] on optimal journal policy under AI-augmented submission.

Not every ZeroPaper run produces a publishable paper, and this paper does not claim otherwise. The scorer, the self-attacker, and the simulated referees are agents I wrote,

calibrated to my reading of what target-journal referees would accept. Whether a human referee at a top finance journal would agree is an empirical question that requires submission and is not yet answered. The narrower claim, the one this paper develops, is that the pipeline is structured to avoid the failure modes most often degrading automated-research prototypes, and that the principles it implements are the ones those failure modes call for. The architecture is the contribution; the empirical rate is measurable but not yet measured.

2 Premises

A pattern language — a vocabulary of named design commitments a system can be audited against (Section 5) — is only as strong as the premises it answers. This section states ten structural facts about LLMs and deployment that any long-running autonomous pipeline inherits. Each of the six principles in Section 3 traces to at least one premise. If a proposed principle does not trace, it is decoration.

2.1 Weaknesses

Premise 1 (self-bias). An LLM that has produced context gets pulled toward defending and continuing it. It rationalizes its prior output rather than evaluating it freshly. The same instance cannot reliably grade its own work.

Premise 2 (long-context degradation). As context grows, recall and reasoning quality degrade. The model misses details in the middle of long inputs, forgets instructions stated at the top, and conflates similar items. Anything the model does not strictly need, hurts.

Premise 3 (coherence drift). Across many steps, invariants get forgotten, overridden, or silently reinterpreted. Small local departures compound. Rules that must hold across a whole run need redundant enforcement, not a single statement.

Premise 4 (stochastic error). Every output is a noisy sample of a latent quality, not the quality itself. Even on tasks the model can do, some fraction of attempts fail: a sign slip, an off-by-one, a dropped constraint, a misread token. A single pass is evidence, not a measurement.

Premise 5 (path-of-least-resistance). LLMs are trained to fulfill objectives. When several paths satisfy the letter of an instruction, the model prefers the cheapest: a shortcut, surface-level compliance, a premature "done." This pattern shows up as specification gaming, skipping available tools because they are unfamiliar, and declaring a task complete before the harder part is attempted. Self-reports are not evidence.

2.2 Capabilities

Premise 6 (reads-any-text). An LLM makes sense of prose, tables, JSON, and mixed formats. Contracts between components can specify what information must appear, not its exact shape.

Premise 7 (judges open-ended predicates). Given a well-posed question, an LLM can read an artifact and return a verdict ("is this sound?", "does this meet the criteria?"). Routing and verification are not limited to mechanical rules over state; they can be semantic.

Premise 8 (fresh instances are less correlated, not independent). Two calls with different prompts and no shared context sample errors at substantially lower correlation than a single continuation, but not at zero: they still share the model's weights, training biases, and systematic failure modes. A new subagent is a real reset of session-local context. Multi-verifier variance reduction has a floor at the model-level correlation. Crossing different models lowers the floor further.

2.3 Deployment

Premise 9 (tokens and time cost). Every token charges dollars and latency; every second of wall-clock charges latency. Costs are linear in input. At equal correctness, the cheaper design wins.

Premise 10 (infrastructure fails independently of the work). Long autonomous runs accumulate transient failures (tool timeouts, rate limits, malformed outputs from a flake, network blips) that are exogenous to the task signal. Any predicate or counter fed by task signal must distinguish exogenous from endogenous failures, or the downstream decision is noisier than the signal warrants.

3 Principles

Six principles, each derived from the premises in Section 2. The principles are stated at the architectural level; their corollaries cover the specific places the simple rule does not reach. The mapping from premises to principles:

- **Principle 1 (state):** premises 1, 2, 3, 10. **Principle 1 (control flow):** premises 1, 3, 5.
- **Principle 2:** premises 2, 3, 9.
- **Principle 3:** premises 1, 2, 3, 4, 5, 6, 8.
- **Principle 4:** premises 1, 4, 5, 7, 8.
- **Principle 5:** premises 5, 6, 7, 9, 10.
- **Principle 6:** premise 9.

3.1 Principle 1: The pipeline is a Markov machine with external control flow

Two system properties are forced. State carries history explicitly, in a compact JSON object that every stage reads and updates, not in a growing transcript. Control flow lives outside the worker: the routing graph is structure, not the worker's judgment. The pipeline's always-loaded configuration file is where both live (in Claude Code this is `CLAUDE.md`; in Codex and Gemini the equivalent file differs). The orchestrator is itself an LLM session with CLAUDE.md always loaded; each step it reads state, dispatches work to a subagent or loads a doc, updates state, and transitions. Premise 2 (long-context degradation) forbids keeping history in the conversation; premises 1 (self-bias) and 5 (path-of-least-resistance) together forbid the worker from routing itself, since a self-routing agent both rationalizes its prior output and reaches for shortcuts when several paths satisfy the letter of the instruction; premise 3 (coherence drift) demands the routing rules be explicit; premise 10 (infrastructure fails) demands that state be durable and resume-safe.

The orchestrator's loop is short:

```
while state.status == "running":
    stage = state.current_stage
    doc = read(f"docs/{stage}.md")
    verdict = run_stage(doc, state)
    state = transition(state, verdict)
    commit(state)
```

Ten corollaries cover the surfaces this short loop opens.

State must be fresh. If state is the sole carrier of history, a stale file silently breaks the Markov property: a crashed-then-resumed run finds previous outputs in place and consumes them as this run's work. Stages must verify intermediate inputs are current before consuming.

The big-picture graph lives in CLAUDE.md. The orchestrator needs to see the pipeline shape at a glance, so the always-loaded file pays the token cost of the overall stage-and-gate

graph; per-stage procedures do not. The graph is cheap; procedures are not.

CLAUDE.md is not a manual. Stage-by-stage procedures, examples, and edge-case notes are not control flow. They belong in per-stage docs loaded on demand.

CLAUDE.md routes, does not work. The orchestrator dispatches stage-level work to fresh-context subagents. The moment it does the work itself, its context fills with domain material and the failure modes the separation kept at arm's length take hold.

Routing state is not observability. Logs, dashboards, and process records exist for humans, not for routing; keep them in separate files with separate freshness budgets. A stale dashboard is ugly; a stale state file silently breaks the Markov property.

Environmental ground truth is its own artifact. Facts about the environment (which data sources exist, which tools have credentials, which services are up) are captured once at pipeline entry and read by stages, not re-derived. Otherwise they drift across stages and shortcuts fill the gaps.

Resumability is a property, not a feature. A long-running pipeline outlives any single session, so each transition must commit atomically and durably before the next begins. Batching multiple logical transitions into one write leaves the resume point ambiguous after a crash.

Seed a valid initial state. Ship a valid initial state pre-committed so the orchestrator's entry point is identical on a fresh run and a resume: read state, and if running with `current_stage` set, continue.

State schema is declared, not emergent. Across stages an orchestrator asked to "update state" can add fields, rename keys, or retype values until some stage reads a shape no stage explicitly wrote. Declare the schema alongside the pipeline graph; schema changes are their own transition.

Rollback is a transition, not an edit. Undoing a committed stage (stale input found late, bad artifact, manual revert) must commit atomically like any other transition, recording which artifacts were deleted and which survived. Otherwise the next stage cannot distinguish

stale orphans from reusable upstream outputs.

3.2 Principle 2: Context is costly

Every always-loaded byte is a bet that its value exceeds its cost, and the cost rises faster than length. The direct cost is linear in length (premise 9), but attention degradation (premise 2) and coherence drift (premise 3) degrade the reliability of everything already loaded. Adding a marginal line taxes the prior content's recall and the prior invariants' hold, so total context cost is superlinear in length.

CLAUDE.md design becomes a budget problem. Every line is judged on three tests: is this load-bearing for this step; is this load-bearing for every step; will this still be needed on step 1000? Domain invariants that fire every step pass the bar. Ornamental prose does not. Agents receive minimal inputs: file paths rather than file bodies when the agent can fetch for itself. State is compact; running details get summarized; raw transcripts do not live in the state JSON. The always-loaded pipeline graph is the exception that proves the rule: every step needs to know where it sits in the whole.

3.3 Principle 3: Delegate

Principles 1 and 2 together force delegation: the machinery lives somewhere else, loaded or spawned only when needed. There are four vehicles, each with a different cost and isolation profile.

Docs are content the orchestrator reads on demand. Cheapest to author, zero isolation: the content lands in the current context. Use for stage procedures, reference material, and content the orchestrator itself needs to act on. In economic terms, docs are the standing operating procedure — the rule the principal consults before acting, at the cost of a context hit.

Scripts are deterministic code invoked directly by the orchestrator or an agent. No model tokens during execution, no coherence drift inside the call. Use for idempotent transforms,

file generation, mechanical checks, schema validation. Scripts play the role of a deterministic enforcement mechanism: once invoked, the output is independent of whoever invoked it.

Skills are self-contained modules (instructions, often with scripts or tools) loaded by the harness on trigger. They land in whoever is currently running. Use for reusable capabilities that multiple workers need. Skills are the analog of a shared technology: any worker that activates one inherits its capabilities without additional principal delegation.

Agents are fresh-context sub-conversations with their own system prompt. Substantial (not total) context isolation: premise 8 (fresh instances are less correlated, not independent) is what makes isolation real rather than rhetorical. Use for work needing independent judgment (premise 1), long work that would pollute parent context (premise 2), parallel execution, or any place stochastic-error re-sampling matters (premise 4). Agents are the analog of a specialist: hired, given a brief, and evaluated on their output — not on their reasoning process.

The orchestrator's loop becomes short: read state, pick a vehicle, dispatch, update state. It does not do the work itself; it decides where the work happens. A corollary: load-bearing invariants (rules whose breach is silent, cascading, and reachable from more than one surface) are restated at each surface the rule can enter from, not only at the dispatch site.

3.4 Principle 4: Verify, do not trust

Workers defend their own output (premise 1) and prefer cheap paths (premise 5), so self-reports are not evidence the work got done. And any verdict, from a worker or a verifier, is a noisy sample of the underlying quality (premise 4): one draw is not enough. Verify everything the orchestrator will route on using other LLMs (premise 7: judges predicates), not the same instance (premise 8: fresh-instance sampling).

The principle has five corollaries that every verification stage must honor.

Verification is a distinct stage, not a sub-step. Fresh context is not enough: a worker that spawns its own verifier inside its own stage has not escaped self-bias. The worker chooses

the framing, the inputs, and what to show. Verification must be a stage the orchestrator dispatches separately.

At least two verifiers, more when the signal is noisier. Variance falls with N only in the independent limit; extra verifiers on the same model and framing buy less than the ideal bound.

Distinct framings. Two verifiers given identical instructions share the blind spots those instructions force them into. Vary posture (structured step-by-step vs. skeptical-reader holistic), phrasing, rubric. Framing is the floor; different models reduce correlation further.

At least one free-form. A numeric score is cheap to route on, but easy to game when the worker can see it. A free-form critique has no single number to climb; its feedback is qualitative and open-ended. Ship both: structured verdict for routing, free-form critique for content.

Each verifier is framed adversarially. A verifier told "check whether this is correct" drifts toward confirming. State the job as finding errors, not evaluating correctness.

3.5 Principle 5: Termination must be mechanical

The orchestrator is a program. Sequences, conditionals, loops, and early exits are all fair game. The constraint is on the *predicate* gating each branch. Two kinds of predicate are legitimate: mechanical (numeric rules over state, for example: if error attempts plateau for two or more consecutive rounds, escalate) and LLM-judged (the orchestrator reads an agent's output and decides). Both are fine for routing; termination is the exception.

Runaway loops need a mechanical termination. Any loop that could run forever must have at least one mechanical branch on its termination path. An LLM orchestrator will rationalize "one more try" indefinitely if termination depends only on its own judgment.

Termination triggers when marginal value stops, not only when some cap is hit. Three strategies, ordered by the data they require: an absolute cap when the loop is new and there is no history, a budget cap at the count where marginal value historically saturates, and a

delta trigger that escalates when $\Delta(\text{score}, \text{feedback-novelty})$ falls below a threshold.

Signal retries are separate from noise retries. Infrastructure failures (tool timeout, rate limit, network blip) get retried at the dispatch layer; task-signal failures (verifier REJECT, solver gave up, malformed model output) feed the termination counter. Collapsing them breaks the pipeline in both directions.

Verdicts on hot paths, prose on rare branches. Structured verdicts (`PASS/FAIL`) are cheap to route on and earn their keep on frequently-visited paths. On rare branches, the orchestrator can read the full artifact and decide; no verdict token is needed. Verdicts buy efficiency, not correctness.

Prefer self-recovery to escalation. Mechanical termination must exist and must rarely fire. Every escalation imposes a cost on someone who was not there. Design the pipeline so cheap self-recovery paths (fresh-instance retry, framing swap, coarser fallback) run out before termination fires.

Graceful degradation is a signal event. A stage that substitutes a weaker input for a missing one (fallback score, cached stand-in, skipped optional check) must record the downgrade in state and count it against the signal-failure budget. Silent fallbacks are broken stages.

3.6 Principle 6: Parallelize independent dispatches

When two dispatches have no data dependency, run them concurrently. Run quality is unchanged; wall-clock time falls. Latency is part of premise 9's cost surface, so cutting it without changing token load is a pure gain. The constraint is that parallelism is in dispatch, not in state mutation (a state-coherence concern that traces to premise 3): parallel branches write distinct keys, or the orchestrator gathers writes after both return. Concurrent writes to the same field race.

4 ZeroPaper: the worked instantiation

4.1 System overview

The pipeline has three layers: a host runtime that acts as orchestrator, a roster of specialized agents that do the research work, and a versioned artifact tree that the orchestrator and agents communicate through.

4.1.1 The orchestrator

The orchestrator is not a separate process. It is the top-level CLI session (Claude Code, Codex, or Gemini) with a system prompt that instructs it to behave as a pipeline coordinator rather than a general-purpose assistant. The system prompt is assembled at setup time from a runtime-agnostic core template and a runtime-specific session-discipline block. The core template defines the stages, the gates, the escalation table, the commit protocol, and the pipeline-state schema. The runtime-specific block defines how the orchestrator launches subagents under that particular CLI (e.g., the `Agent` tool under Claude Code, the `codex` subagent protocol under OpenAI Codex).

The orchestrator does four things on every step. It reads `process_log/pipeline_state.json` to determine the current stage. It decides which agent to launch next, based on the stage's rules and the artifact state. It launches the agent, waits for the output, and writes the output to the artifact tree. It updates the pipeline state (including appending a timestamped entry to the history array) and commits. Every state change is a separate commit, so the run's provenance is reconstructable line by line.

4.1.2 The agent roster

The pipeline relies on roughly thirty specialized agents. An agent is defined by a prompt body (a markdown file that describes the agent's role, constraints, outputs, and failure modes) plus a metadata block (which model to use, which tools the agent has access to,

and a one-line description that the orchestrator reads when deciding which agent to launch). Agents are dispatched as fresh subprocesses with clean context; they see only the artifacts they are explicitly given, not the orchestrator's session history or other agents' reasoning.

This isolation is load-bearing for adversarial evaluation. A math-auditor launched with only the theory draft in context has no way to rationalize around the generator's reasoning chain, because it never saw that chain. A novelty-checker with only the theory's claim and web access cannot be talked out of flagging a known result by the generator's internal arguments, because the generator is not in the room.

4.1.3 The artifact tree

All communication between the orchestrator and the agents happens through files under `output/`. Each stage writes to its own subdirectory (e.g., `output/stage0/`, `output/stage2/`). Within a stage, iterated artifacts are versioned (e.g., `theory_v1.md`, `theory_v2.md`); canonical versions of "current" artifacts are also written at the top level of the stage directory and are the authoritative inputs for downstream stages. Agent outputs go into the tree; the orchestrator reads them, decides the next step, and the cycle continues.

The artifact tree doubles as the process log. Because every agent output is committed, and because the pipeline state's history array records every transition with a timestamp, a completed run's git log is a linear record of who did what and when. The dashboard (`dashboard.html`) reads this history and renders it for the user.

4.1.4 The state file

`pipeline_state.json` is the single source of truth for where the pipeline is. A representative subset of its schema is shown below; the full schema additionally tracks `reject_cosmetic_round`, `fix_empirics_round`, `bib_verify_round`, `polish_round`, `regeneration_round`, `pivot_resolved`, `pivot_history`, `triaged_lit_implications`, `seeded`, `stage2b_theory_version`, and `target_journal_ti`

```
{
```

```
  "current_stage": "stage_0",
  "problem_attempt": 1,
  "idea_round": 0,
  "theory_attempt": 1,
  "theory_version": 1,
  "referee_round": 0,
  "pivot_round": 0,
  "status": "not_started",
  "scores": {},
  "stage1_candidates": [],
  "history": []
}
```

The orchestrator reads this file at the start of every turn and updates it after every stage transition. Because it is JSON and committed, another orchestrator session can resume an interrupted run, and the resumption is deterministic: whatever stage the file points to is where the pipeline continues from.

4.1.5 The division of labor

The orchestrator produces no research artifacts of its own. It never writes proofs, never writes prose, never runs empirical analyses; its judgment is exercised over routing (which agent to launch, when to escalate, which gap to develop from the literature scout's map, whether a problem statement meets the Gate 0 bar), not over content. Its only outputs are state updates, dispatch decisions, and bookkeeping. All substantive output is produced by specialized agents. This split has two consequences. First, the orchestrator's context stays small and stable across a long run, because it is not accumulating the full content of every agent's output, only the file paths and verdicts. Second, debugging a bad paper reduces to debugging a bad agent, not a bad orchestrator: if the scorer is consistently generous, or the self-attacker is consistently shallow, the fix is in that agent's prompt, not in the coordinator's logic.

The orchestrator does retain discretion over three things: which stage to enter next (constrained by the gate decisions), how to respond to escalations (constrained by the escalation table), and when to launch the scribe agent (which records course corrections and discussion). These are judgment calls that I have not yet found a way to automate without losing the ability to recover from unanticipated states.

4.2 The pipeline: stages and gates

The base pipeline has ten numbered stages (Stage 0 through Stage 9) and six adversarial gates (Gates 0 through 5). A candidate artifact advances by clearing the gate that follows its stage; a failing gate sends the candidate back, either to the same stage for revision or to an earlier stage for regeneration. Extensions add additional stages and gates (notably `Gate 3a-feasibility` under the empirical extension); these are described in Section 4.5. Letter suffixes (`2b`, `3a`, `3b`) denote stages and gates that share a numbered block but do not always execute in alphabetical order: `Gate 3a-feasibility` runs before Stage 3 as the empirical extension's pre-check; Stage 3a (empirical analysis) runs after Stage 3; Stage 2b (Theory Exploration) runs after Gate 3 (Novelty), not between Stages 2 and 3. The numbering reflects topical grouping, not strict execution order. Each stage and gate is described below.

4.2.1 Stage 0: Problem Discovery

The pipeline starts without a predetermined topic. Stage 0 produces a problem statement by running two agents in sequence. The *literature-scout* performs a broad survey of recent publications in the target journals and produces a literature map. The orchestrator then selects a gap from the map and launches *gap-scout*, which performs a focused deep-dive on the adjacent literature, identifies the closest existing competitor, and validates that the gap is real (i.e., not already filled by a paper the broad scout missed). The stage outputs a problem statement naming a specific unresolved question, the closest relevant prior work, and a first-pass feasibility assessment.

Gate 0: Problem Viability. The orchestrator evaluates whether the problem statement meets a minimum bar (well-defined, non-trivially open, plausibly tractable in a theory paper). Failures return to Stage 0 with a different gap; after five failed attempts the system picks the best-scoring candidate and proceeds.

4.2.2 Stage 1: Idea Generation

Given a problem statement, Stage 1 produces a ranked list of candidate ideas and selects one to develop. The *idea-generator* writes several candidate sketches, each a short description of a mechanism and a proposed first result. The *idea-reviewer* critiques and ranks them.

Gate 1: Idea Review. Generator and reviewer iterate up to five rounds; if the reviewer has not converged on a top-K selection (with $1 \leq K \leq 3$) after five rounds, the orchestrator picks the best available and advances.

Gates 1b and 1c: Parallel screening. The top-K ideas are then screened in parallel by two further agents. Gate 1b dispatches K novelty-checkers concurrently, each performing independent web search on one candidate; candidates flagged KNOWN are dropped. Gate 1c dispatches idea-prototypers on the survivors, each performing a quick math feasibility check; candidates flagged BLOCKED are dropped. Negative results accumulate in a persistent file so future runs inherit the knowledge. Remaining survivors are tiebroken (novelty tier, surprise tier, reviewer rank); the winner's artifacts are promoted to canonical filenames and the runners-up retained as pre-vetted fallbacks. If all K candidates fail, Stage 1 restarts with the feedback recorded.

4.2.3 Stage 2: Theory Development

The *theory-generator* produces a full theory draft: model, assumptions, propositions, proofs, corollaries. The draft is versioned (`theory_v1.md`, `theory_v2.md`, and so on); every subsequent audit or revision produces a new version. On Stage 2 re-entry after a failed gate, the

generator can choose to mutate, crossover, or regenerate fresh depending on what the failure feedback suggests.

Gate 2: Math Audit. Every theory version is audited for mathematical correctness by two agents in sequence. The structured *math-auditor* re-derives each proposition step by step, checking algebra, limits, and edge cases. If the structured audit passes, the *math-auditor-freeform* reads the theory as a skeptical reader with no checklist and flags anything that feels wrong. A theory version advances only after both auditors pass. Three failed audit attempts abandon the theory version and send the pipeline back to Stage 2 with the specific failure feedback.

Gate 3: Novelty on the full theory. A novelty-checker searches the web for prior work matching the full theoretical claim (not just the idea sketch). Verdicts are NOVEL, INCREMENTAL, or KNOWN. Three consecutive INCREMENTAL verdicts after rework abandon the idea and return to Stage 1.

Stage 2b: Theory Exploration. A dedicated *theory-explorer* runs numerical verification on the propositions, calibrates the model to reasonable parameter ranges, and produces plots. Failures of numerical verification (sign flips, reversed comparative statics, failure at plausible calibrations) return to Stage 2 as substantive, not tool-related, failures. The pipeline treats an unexpected sign reversal as a potential new contribution rather than a bug.

Gate 3a-feasibility: Empirical Feasibility. If the project was set up with the empirical extension, the orchestrator now checks that the theory's predictions can actually be tested with available data. Failure here sends the pipeline back to Stage 1 for a different idea, since a theory whose predictions cannot be tested is not one this variant of the pipeline can finish.

4.2.4 Stage 3: Implications

For each theory prediction, the orchestrator launches a focused *gap-scout* search to classify it as NOVEL, PUZZLE-CANDIDATE, SUPPORTED (already shown), or DEAD (already contradicted by data). The stage's output is a tagged list of predictions that feeds into Stage 4 and, where applicable, into the empirical extension's analysis stages.

4.2.5 Extension stages (optional)

If the empirical extension is active, Stage 3a runs the *empiricist* agent on the predictions, executes the analysis code, and audits the results with *empirics-auditor*. If the theory-llm extension is active, Stage 3b runs the *experiment-designer* and *experiment-reviewer* pair to design and validate LLM experiments testing the theory's behavioral predictions. Either extension can produce empirical or experimental contradictions; these trigger the Puzzle Triage machinery.

4.2.6 Puzzle Triage

When empirics or experiments contradict the theory, the pipeline does not silently accept the contradiction. It launches *puzzle-triager*, which classifies the situation into one of six resolutions: NORMAL-PROCEED (the contradiction is minor, proceed), FIX-EMPIRICS (the analysis was misfit, re-run), RECONCILE (add a scope condition, re-audit), BACK-TO-IDEA (the theory is wrong, return to Stage 1), PIVOT (rebuild the theory around the contradiction, up to two pivots), or HONEST-NULL (the theory fails, present with limitations or return to Stage 0). This triage prevents the pipeline from either suppressing contradictions or over-reacting to them.

4.2.7 Stage 4: Self-Attack

Before the paper is written, the *self-attacker* reads the full theory plus empirical or experimental results and produces the harshest critique it can generate, explicitly simulating a

hostile referee. The *triager* then classifies each concern into Address, Scope-Out, or Drop buckets with written justifications for any downgrades, so the generator does not waste cycles on items the triager judges out-of-scope; only items tagged `[FIX]` feed back into theory revision.

Gate 4: Scorer Decision. The *scorer* reads the theory and the self-attack and produces a numerical score plus a verdict: ADVANCE (75 or above for the default top-5 target tier; 65 for field journals, 55 for letters), REVISE, MAJOR REWORK, or ABANDON. The scorer tracks the score trajectory across versions and uses delta thresholds: a positive delta of 3 or more justifies another iteration in the same band; a delta under 3 triggers escalation. The pipeline also runs a *scorer-freeform* in parallel, a second scorer with no rubric whose verdict is weighed against the structured one, guarding against rubric-gaming. A *branch-manager* is launched after both scorers return and is a mandatory third component of Gate 4; no gate decision is valid without a fresh branch-manager report for the current version. Plateaus in the 55 to 74 range trigger either a deepening playbook (further mathematical development) or, if branch-manager recommends Regenerate and no regeneration round has yet fired, a regeneration round: the orchestrator writes a learnings file, archives any existing draft, and re-enters Stage 1 with the learnings fed to idea-generator. At most one regeneration round per problem; a second plateau falls back to the deepening playbook. Scores below the abandon threshold trigger the ABANDON escalation path, which returns to Stage 0 only after five ABANDON verdicts on the same problem.

4.2.8 Stage 5: Paper Writing

The *paper-writer* converts the final theory, exploration results, empirical or experimental findings, and scope conditions into a full LaTeX manuscript with sections, proofs, tables, and references. The introduction and conclusion are written last, after the other sections are stable, to ensure they match what the paper actually delivers rather than what was initially

planned.

4.2.9 Stage 6: Referee Simulation

Three referee agents read the full paper independently and produce independent reports. The *referee* agent simulates a domain-appropriate journal referee with the variant's target-journal standards in mind. The *referee-freeform* agent reads without a rubric and produces an editorial-style assessment. The *referee-mechanism* agent focuses specifically on whether the paper's claimed mechanism actually matches what the math and empirics deliver, flagging MISATTRIBUTED (the claimed mechanism is not what's driving the result) or DECORATIVE (the mechanism framing adds nothing the math doesn't already say) verdicts.

Gate 5: Referee Decision. After the three referees return, an *editor* agent aggregates their reports. The orchestrator does not read the three reports directly to form a verdict; the editor is the verdict aggregator. The editor applies hard aggregation rules: any single Reject vote from the structured or freeform referee produces an aggregated Reject verdict (with one escape: the editor may downgrade to Major Revision if the rejection is on journal-fit grounds only); a MISATTRIBUTED or DECORATIVE verdict from the mechanism referee forces Major Revision (the mechanism referee's verdict space is VALID / MISATTRIBUTED / DECORATIVE rather than Reject / Major / Minor / Accept, by design: mechanism mismatch is treated as fixable in revision rather than fatal, so its strongest signal caps at Major Revision). The editor produces a single output containing the aggregated verdict, a mechanism verdict pass-through, a canonical comment list (the merged, deduplicated comment list across all three reports), and a journal-fit verdict. Gate 5 routes on the editor's aggregated verdict, not on any individual referee's. Accept or Minor Revision advances to Stage 7. Major Revision fires a *triager* on the editor's canonical comment list, which classifies each comment before revision proceeds; revision cycles run up to ten rounds. Reject routes through the never-abandon path: Stage 6 fires only after Stage 5, so a paper draft always

exists, and Reject under structural concerns invokes the deepening playbook with up to ten revision rounds rather than returning to Stage 0. Mechanism referee deadlocks at round ten trigger either a forced adoption of the identified driver or a narrow-path rewrite, depending on the run mode.

4.2.10 Stage 7: Style Check

The *style* agent reads the finished paper and flags violations of the style guide: unsupported superlatives, unhedged claims, redundant sections, vague transitions, and stylistic markers the variant's style guide forbids. Violations are fixed in place.

4.2.11 Stage 8: Bibliography Verification

The *bib-verifier* agent reads every citation and confirms that the reference exists, the authors are correct, the venue is correct, and the year is correct. Hallucinated references are a well-documented failure mode of LLM-generated papers; this stage exists to catch them before publication. Failures are fixed in place and re-verified, up to a bounded number of rounds.

4.2.12 Stage 9: Polish

Stage 9 is the final substantive pass before the pipeline marks the paper complete. Six polish agents are dispatched in parallel, each with a narrow, focused checklist. Where Stage 6's three referees and the editor evaluate the paper holistically against journal standards, the polish agents do narrow mechanical checks (per-equation re-derivation, per-citation prose-vs-source verification, per-numerical-example recomputation) that reward focused scope and would dilute a referee's broader assessment; for the upstream theory-draft auditors (math-auditor, scorer, self-attacker), polish additionally catches errors introduced when the draft was typeset into LaTeX. The six agents are: *polish-consistency* (cross-section contradictions, heading-vs-text mismatches, intro framings qualified away later); *polish-formula* (re-derivation of every numbered equation, lemma, proposition, and corollary in the rendered paper, to catch

sign flips, subscript errors, and FOC mistakes introduced during typesetting; returns no findings on papers without formal mathematical content, as is the case for the present paper); *polish-numerics* (recomputation of every numerical example and calibration claim); *polish-institutions* (verification of real-world facts, regulatory mechanisms, and characterizations of cited papers' frameworks); *polish-equilibria* (detection of unstated multiple equilibria, missing continuum assumptions, and welfare benchmarks that do not match the model's principal); and *polish-bibliography* (per-citation prose-claim verification against the cited paper's actual content). After the six agents return, a *triager* aggregates their findings into Apply, Investigate, and Drop buckets. The *paper-writer* is then re-invoked to apply the triaged fixes. Stage 9 owns the `"status": "complete"` flag in the pipeline state; the pipeline is not done until polish has either applied its fixes or judged that nothing actionable remains.

4.2.13 The gates as a unit

The six gates are not uniform in kind. Gates 0, 1, and 4 evaluate *worth*: is this problem worth pursuing, is this idea worth developing, is this theory worth publishing? (Gates 1b and 1c, which screen individual idea candidates in parallel within Stage 1, are sub-gates of Gate 1 and use the same worth-evaluation logic.) Gates 2 and 3 evaluate *correctness*: does the math hold, is the claim actually new? Gate 5 evaluates *presentation and standing*: would a reviewer at a target journal accept this? Each gate's design reflects its kind. Worth gates use scoring and verdicts. Correctness gates use structured re-derivation plus a freeform skeptical reader. Presentation gates use a panel of three referees with different priors and a meta-rule against mechanism drift. All six gates are concrete instances of principle 4: an evaluator with no shared context with the generator, framed adversarially, returning a structured verdict plus a free-form critique.

4.3 Agents

The pipeline's behavior is determined more by its agents than by its orchestrator. This section describes the roster and the three design properties I rely on: isolation, specialization, and adversarial pairing.

4.3.1 Roster

Agents fall into three groups: shared agents that are identical across variants, variant-specific agents whose prompts change with the research domain, and extension agents that are added only when the corresponding extension is enabled at setup time.

Shared agents are domain-agnostic. They receive variant-specific context (target journals, paper type, domain areas) as parameters at setup, but their prompt bodies are the same for a finance run and a macro run. The roster includes: *literature-scout* (broad literature survey at Stage 0), *gap-scout* (deep search on a specific gap, reused at Stage 0 and per-prediction at Stage 3), *idea-prototyper* (quick math feasibility check at Gate 1c), *theory-explorer* (numerical verification and plots at Stage 2b), *math-auditor* (structured step-by-step derivation check), *math-auditor-freeform* (skeptical re-read with no checklist), *novelty-checker* (web-search novelty verification), *scorer-freeform* (holistic quality assessment at Gate 4), *referee-freeform* (editorial-style referee at Stage 6), *referee-mechanism* (mechanism-fit referee at Stage 6), *editor* (aggregates the three referee reports into the Gate 5 verdict; produces the canonical comment list for the triager), *triager* (classifies adversarial findings at Gate 4, Gate 5 Major, and Stage 9 polish; produces structured triage with written justifications), *paper-writer*, *style*, *bib-verifier*, *branch-manager* (launched at every Gate 4 after scorers return, at Gate 5 Reject, and every third theory version inside the Stage 2 audit loop), *puzzle-triager* (handles empirical contradictions), *debugger* (diagnoses tool-execution failures), *scribe* (launched after every stage transition and gate decision; runs in the background so the pipeline continues uninterrupted), *polish-consistency*, *polish-formula*,

polish-numerics, *polish-institutions*, *polish-equilibria*, and *polish-bibliography* (the six Stage 9 polish agents, each with a disjoint, focused checklist; see Stage 9). Math-touching agents (*math-auditor*, *math-auditor-freeform*, *idea-prototyper*, *theory-explorer*, *self-attacker*, *referee*, *referee-mechanism*, *polish-formula*, *polish-equilibria*, and *polish-numerics*) preload a `sympy` skill for symbolic verification and escalate to *codex-math* only when the problem requires extended reasoning rather than computation.

Variant-specific agents are instantiated per variant from a shared core body with variant-context fragments spliced into placeholders at setup time, so the deployed prompt differs across variants while the structural skeleton is reused. The current variants are `finance` and `macro`. The variant-specific agents are *idea-generator* (needs domain-specific brainstorming patterns), *idea-reviewer* (needs domain-specific evaluation criteria), *theory-generator* (needs domain-specific model-structure guidance), *scorer* (needs domain-specific score calibrations), *self-attacker* (needs domain-specific attack vectors), and *referee* (needs domain-specific journal standards).

Extension agents are added at setup time when the user passes an extension flag. The `--ext empirical` flag adds *empiricist* (variant-specific) and *empirics-auditor* (shared). The `--ext theory_llm` flag adds *experiment-designer* and *experiment-reviewer* (both shared). Extensions also inject new pipeline stages (Gate 3a-feasibility and Stage 3a for empirical; Stage 3b for theory-llm) and extend the puzzle-triage machinery to handle their contradictions.

4.3.2 Isolation

Every agent runs in a fresh subprocess with a clean context window. The agent sees the prompt body, the files the orchestrator explicitly hands it, and nothing else. In particular, it does not see the orchestrator's system prompt, the prior agents' reasoning, or the pipeline-state history. This isolation is deliberate. If a math-auditor inherited the generator's chain

of reasoning, it would be biased toward accepting the generator's framing of the problem; a fresh reader without that context is more likely to notice gaps.

Isolation does come at a cost. Because agents cannot coordinate directly, the orchestrator has to do the coordination through artifacts, and artifacts have to be self-contained enough that a cold reader can evaluate them. This isolation is a forcing function on artifact quality: if the theory draft can only be evaluated by someone who was present when it was generated, it is not finished.

4.3.3 Specialization

Each agent has one job, a known set of outputs, and a bounded context. Compressing multiple roles into a single agent degrades quality. A math-auditor that was also asked to score novelty, for instance, would produce shallower math audits because its attention was split; splitting them into two agents that each see the full relevant context appeared to give better results on both dimensions in test runs.

Specialization also makes debugging tractable. When a run produces a thin section, the offending artifact is traceable to a specific agent and the fix is in that agent's prompt. When an agent is used in multiple places (gap-scout runs at Stage 0 and at every Stage 3 prediction), changes to its prompt propagate to every call site uniformly.

4.3.4 Adversarial pairing

Several of the gates rely on pairing a generator agent with an evaluator whose explicit job is to disagree with the generator. The pairs are: idea-generator with idea-reviewer at Stage 1; theory-generator with math-auditor, math-auditor-freeform, and novelty-checker at Stage 2; theory-generator with self-attacker at Stage 4, and theory plus self-attack with scorer, scorer-freeform, and branch-manager at Gate 4; paper-writer with three referees (referee, referee-freeform, referee-mechanism) whose outputs are aggregated by the editor at Stage 6. The evaluator is not given the generator's reasoning, only the generator's output, and is

explicitly instructed to find flaws rather than to find merit.

This adversarial pairing is an engineered dynamic, not an emergent one. LLM agents default to cooperating with the framing they are handed; to get adversarial evaluation, the prompt has to tell the evaluator that its job is to find the strongest objection a human referee would make. Even mild cooperative framing in an evaluator's prompt (e.g., "provide constructive feedback") softens the audit. The evaluator prompts therefore use harsher language ("your job is to find every weakness a referee would find") and require explicit verdicts, not recommendations.

4.3.5 Model selection

Agents can be assigned different underlying models. The default deployment uses a strong model for generator agents (idea-generator, theory-generator, paper-writer) and evaluator agents that need deep reasoning (math-auditor, self-attacker, referee). Cheaper models are used for agents whose job is primarily pattern-matching or retrieval (literature-scout, bib-verifier). The `--light` flag at setup time downgrades all subagents to a faster, cheaper model while keeping the orchestrator at the strong tier, trading quality for cost on runs where the user wants to iterate rapidly.

4.4 Quality enforcement

The quality of an autonomous pipeline's output is bounded by the quality of its worst agent and its worst coordination rule. In this section I describe the three mechanisms I rely on to keep that bound high: adversarial evaluation at every gate, an explicit escalation table that prevents infinite loops without compromising the standard, and the never-abandon rule that forces the pipeline to finish what it starts once a draft exists.

4.4.1 Adversarial evaluation

Every substantive advance in the pipeline passes through at least one evaluator that did not generate the content being evaluated and was given no access to the generator's reasoning. I described the pairings in Section 4.3; here I describe why the design works.

Cooperative evaluation, where a single agent both generates and assesses, fails in a predictable way: the agent rationalizes its own output. A theory-generator that was also asked to audit its theory would find its proofs convincing, because it was convinced of them when it wrote them. Isolating the evaluator breaks that rationalization. The math-auditor has no access to the chain of reasoning that led to a particular lemma; it only sees the lemma statement and the proof, and asks whether the proof actually proves the statement. When it does not, the math-auditor does not have a story about why it almost does.

I also duplicate evaluators where I can afford to. The structured math-auditor pairs with a free-form math-auditor reading the same theory as a skeptical human; the scorer pairs with a scorer-freeform reading without a rubric; Stage 6 runs three independent referees with different priors. Duplication is expensive, but it catches the systematic failure where a single evaluator's blind spot matches the generator's blind spot.

4.4.2 Escalation

A pipeline that iterates indefinitely is a pipeline that never ships. I enforce termination through an escalation table that is consulted after every gate failure. The rules that matter most:

- **Idea review iterates:** five rounds, then advance with the best available.
- **Gate 3 novelty INCREMENTAL:** three rework attempts at Stage 2, then abandon the idea and return to Stage 1.
- **Math audit fails:** three attempts, then abandon that theory version.

- **Scorer delta under 3 (plateau or decline):** escalate one level (REVISE to MAJOR REWORK to ABANDON).
- **Scorer plateau 55 to 74 for two consecutive rounds:** switch to the deepening playbook rather than regenerating.
- **Theory scored ABANDON five times on the same problem:** change the problem (return to Stage 0).
- **Problem viability fails five times:** pick the best scoring candidate and proceed anyway.
- **Referee Major Revision with structural concerns:** up to ten rounds; use the deepening playbook, but keep going as long as each round surfaces any new issue.
- **Mechanism referee deadlock at round ten:** force adoption or narrow-path rewrite, depending on the situation; do not loop further.

The table encodes a view about what kinds of failure are recoverable and at what rate. Novelty failures are structural: if three rework attempts have not made the theory novel, the fourth will not either, and the efficient move is a different idea. Scoring failures are more gradual: a plateau in the 55 to 74 range usually means the core idea is sound but thin, and the right move is to deepen the mathematics rather than throw the idea out. The table reflects what I learned from earlier runs, not a first-principles derivation.

4.4.3 The never-abandon rule

Before Stage 5 (Paper Writing), the pipeline can abandon a topic and return to Stage 0. After Stage 5, it cannot. The reasoning is asymmetric investment: once a draft exists, the cost of producing it is sunk, but the expected value of polishing that draft into a finished paper exceeds the expected value of starting over with a new topic and reaching Stage 5 again. This is true even when the current draft scores badly, because a mediocre finished paper still

documents real work and still produces a submittable artifact, whereas an abandoned draft produces nothing.

The never-abandon rule is enforced by the escalation table, not by agent prompts. The scorer is a Gate-4 agent and is not re-invoked post-Stage-5; the post-Stage-5 surface where abandonment would otherwise be tempting is Gate 5. Editor Reject verdicts at Stage 6 are preserved as the editor's verdict but routed through the deepening playbook: a deepen directive synthesized from each rejecting referee's "what would be publishable" section is passed to the theory-generator (and to the empirical or experimental extension where active), and a branch-manager judges the deepening as substantive or cosmetic. Two consecutive cosmetic verdicts trigger a Regeneration Round (a fresh idea cycle informed by a learnings file) on eligible runs; ineligible runs fall back to standard Major Revision under the ten-round cap. The orchestrator cannot bypass these rules because they are checked at every stage transition.

This rule matters because without it, the pipeline has a pathology I have seen in practice: as quality pressure mounts, the pipeline retreats from hard problems toward easy ones, because each retreat locally looks rational (the new topic has a higher expected score than the struggling one). The result is a sequence of shallow starts, none of which finishes. The never-abandon rule breaks that pathology by making retreat unavailable at the point where finishing is the better policy.

4.4.4 Deepening playbook

The deepening playbook is the escape valve that lets the pipeline respond to late-stage quality pressure without abandoning. When the scorer plateaus in the 55 to 74 range, or the referee gives Major Revision with structural concerns, the orchestrator selects one or two mathematically hard extensions that might deepen the core result: continuous-time reformulations, incomplete-markets or heterogeneity extensions, learning or incomplete-information layers, general preference classes, higher-dimensional versions, perturbation analysis with

formal error bounds, and so on. The extensions are run through Gate 2 and Gate 4 as first-class theory work, not as bolt-ons. If an extension produces a counterexample, the counterexample is as valuable as a positive result and is incorporated into the paper.

The playbook distinguishes itself from "polish" in that every extension is a new proof or a new result, not a rewrite of existing prose. The pipeline's failure mode in the absence of a playbook is to respond to depth criticism by rewriting the introduction in stronger language; the playbook makes that response unavailable and forces actual new content.

4.4.5 Post-pipeline math audit

After the pipeline is formally complete, if the manuscript is edited (co-author additions, referee response changes, manual revisions), any new or modified proposition, lemma, or corollary must pass a fresh math audit before being committed. I added this rule after observing that post-pipeline edits to mathematical content had a high error rate compared to edits made during the pipeline, presumably because the editing context was less rigorous. The rule is enforced by a commit protocol: post-pipeline mathematical content is staged in an audit area, audited, and only then moved into the paper sections.

4.5 Runtimes, variants, and extensions

ZeroPaper is deployed by running a setup script in an empty directory. The script assembles a self-contained project by combining a runtime-agnostic core, a runtime-specific packaging layer, a variant that selects the research domain, and any extensions the user requests.

4.5.1 Multi-runtime packaging

The same pipeline runs under three host CLIs: Claude Code, OpenAI Codex CLI, and Gemini CLI. Each host has different subagent semantics (how to launch a fresh agent, how to pass it context, how to retrieve its output), different tool primitives (file editing, shell execution, web search), and different system-prompt conventions. I factor the pipeline along

this split.

The runtime-agnostic layer consists of: the orchestrator core (pipeline stages, gates, escalation table, state schema, commit protocol); the agent prompt bodies (markdown files describing each agent's role and outputs); and the scoring calibrations (what a 75 means in a finance run versus a macro run). These files are the same across runtimes.

The runtime-specific layer consists of: a session-discipline block that tells the host how the orchestrator should behave in its idiom (which tool to use for agent dispatch, how to handle streaming output, how to interpret the host's tool restrictions); an agent metadata file that specifies per-runtime overrides (model IDs, tool allowlists, subagent invocation mechanics); and an assembler script that combines the shared body with the runtime metadata to produce the host's expected agent-definition format.

At setup time, runtime-specific assembler scripts run in sequence, one set per runtime, each producing a complete set of agent definitions in the host's expected format: `.claude/agents/*.md` for Claude Code, `.codex/agents/*.toml` for Codex, `.gemini/agents/*.md` for Gemini. The user can then launch any of the three runtimes in the project directory and get the same pipeline.

The runtime-agnostic/runtime-specific split matters because it decouples pipeline evolution from host evolution. When Claude Code adds a new tool, I update the Claude session-discipline block and the agent metadata; the pipeline logic does not change. When a new pipeline stage is added, I edit the core template once; all three runtimes inherit it automatically.

4.5.2 Variants

A variant is a specialization of the pipeline for a research domain. Currently supported variants are `finance` (target: Journal of Finance, Journal of Financial Economics, Review of Financial Studies) and `macro` (target: American Economic Review, Econometrica, Quarterly Journal of Economics, Journal of Political Economy, Review of Economic Studies, Journal

of Monetary Economics).

A variant specifies four things: the paper type (theory paper, empirical paper, etc.), the list of target journals, the domain areas that Stage 0 literature survey should cover, and the scoring calibration that tells the Gate 4 scorer how to weight correctness, novelty, and significance in this domain. The variant-specific agents (idea-generator, idea-reviewer, theory-generator, scorer, self-attacker, referee) have different prompt bodies per variant because the underlying activity differs: brainstorming in finance emphasizes asset-pricing and market-microstructure patterns, while macro brainstorming emphasizes general equilibrium and policy-relevance; the scorer in finance weights empirical testability differently than the macro scorer, and so on.

Adding a new variant means: authoring the variant-specific agent bodies, writing a scoring calibration file, and adding a configuration block to the setup script with the paper type, target journals, and domain areas. It does not require changes to the runtime layer, the shared agents, or the pipeline stages.

4.5.3 Extensions

Extensions are optional layers that add pipeline stages and extra agents. An extension is activated by passing `--ext <name>` at setup time; multiple extensions can be combined.

The `empirical` extension adds access to financial databases (CRSP, Compustat, FRED, WRDS) through a long-running local socket server that keeps the WRDS authentication session warm across queries. It adds an *empiricist* agent (variant-specific) that runs empirical analyses against the theory's predictions, an *empirics-auditor* agent (shared) that verifies the analysis code and results, and three new pipeline elements: Gate 3a-feasibility (empirical feasibility check before heavy work), Stage 3a (full empirical analysis after Stage 3), and an extension to the Puzzle Triage machinery that handles empirical contradictions specifically.

The `theory_llm` extension adds infrastructure for running LLM-based experiments that test a theory's behavioral predictions (does an LLM agent actually behave the way the theory

says, under the theory's parameterization?). It adds an *experiment-designer* agent that writes experimental protocols, an *experiment-reviewer* agent that validates methodology, and Stage 3b for experiment design and review. The extension includes its own LLM client and protocol runner, installed into the project at setup.

Both extensions extend the Puzzle Triage step so that contradictions from their analyses route through the same six-way classification used for theoretical contradictions. This keeps the pipeline's response to negative evidence uniform across sources.

4.5.4 Seeded mode

By default the pipeline starts at Stage 0 with no preconceptions. The `--seed` flag changes this: the user drops their own idea files into `output/seed/` before launching, and the pipeline starts at a seed-triage stage rather than at problem discovery. The seeded run respects the user's topic as a hard constraint: if downstream gates produce results that diverge from the seed's framing, the pipeline prefers a narrow honest presentation of what it can deliver under the seed topic rather than adopting an unrelated mechanism. Seeded mode lets researchers apply the adversarial machinery to an existing idea rather than running fully autonomously.

4.5.5 Setup.sh walkthrough

The setup script takes a project name, a variant flag, and optional extension flags. It clones the template repository into a temporary location, runs the runtime-specific assembler scripts (one set per runtime, plus skill assembly), combines the selected variant's agent bodies with the runtime metadata, applies any extensions by layering in their agents and stages, injects the variant context (paper type, target journals, domain areas) into the relevant agents, installs Python dependencies (primarily sympy and matplotlib) via `uv pip install`, sets up the output directory structure and initial pipeline state, removes template infrastructure from the project, removes the remote origin so the project is no longer linked to the template repository, and produces a commit in the new project with the initial state. The script's

output is a standalone project folder that has no runtime dependency on the template.

The `--manual` flag activates a research toolkit mode that assembles the full agent roster and skill set without creating an autonomous pipeline state or autonomous orchestrator session. In manual mode the user interacts with the agents directly as needed, without a coordinating orchestrator driving the research process. The setup script detects which of three project shapes applies when `--manual` is passed: an existing paper living in a nested git repository (the outer project adds a `paper/` ignore line), flat `.tex` files in `paper/` (the toolkit operates on them directly), or no existing paper (the user can begin drafting from scratch using the assembled agents). Manual mode does not initialize a `pipeline_state.json` file.

Once setup completes, the user launches any of the three runtimes in the project directory, asks the orchestrator to run the pipeline, and walks away. The run produces commits, stage transitions, and artifact updates until either the dashboard shows a completed paper or the orchestrator has exhausted the escalation table's bounds.

5 Related work

ZeroPaper sits at an intersection of four literatures: methodological pieces on LLMs as tools for economic research; empirical work using LLMs to do finance research tasks; the macro literature on generative-AI productivity in knowledge work; and meta-science work on the economics of research production itself. I situate the system relative to each, then turn to the small set of prior autonomous-pipeline systems and the software-engineering pattern-language tradition that supplies the principles' framing.

The most direct implication of the research-cost trend documented in the introduction is that any technology capable of reducing the marginal cost of producing a submission-quality artifact changes the equilibrium volume and composition of the research pool. ZeroPaper is one systems response to that implication: an attempt to drop the marginal cost of producing a paper-length artifact by orders of magnitude, conditional on quality gates that screen the

output. It is not the only response on offer, and the ones from inside economics are more measured.

Korinek [2023] catalogs dozens of LLM use cases across six research domains (ideation, writing, background research, data analysis, coding, mathematical derivations) and rates each from experimental to highly useful. Korinek [2025] extends the survey to agentic systems, autonomous multi-step LLM workers, and the practical machinery (including Claude Code, OpenAI Codex, and Deep Research) on which they run. ZeroPaper is a concrete instantiation of the agentic-systems architecture that survey describes: not a toolkit a researcher reaches for, but a single pipeline that coordinates the full agent roster end-to-end. Ludwig et al. [2025] is a complementary contribution; it formalizes when LLM-based measurement and prediction yield valid econometric inference, identifying failure modes such as training leakage and LLM measurement error in concept extraction. That work addresses LLMs as measurement instruments in empirical research; ZeroPaper's adversarial gates address a distinct set of failure modes that arise when LLMs serve as the researcher rather than the instrument. Cowen and Tabarrok [2023] occupy an adjacent slot focused on teaching and learning rather than research production.

A separate line of work uses LLMs as the empirical apparatus inside finance papers. Lopez-Lira and Tang [2023] score news headlines with GPT-4 out-of-sample and find the scores predict initial market reactions and subsequent drift, with returns to the strategy decaying as the technology diffuses. Jha et al. [2024] build a firm-level LLM investment score from earnings-call text that forecasts capital expenditure up to nine quarters ahead with information beyond Tobin's q. Eisfeldt and Schubert [2024] survey the broader finance-and-AI landscape from finance-native authors. These papers use LLMs to do finance research; ZeroPaper produces finance research using LLMs. The two contributions sit on opposite sides of the same divide.

The macro literature on generative-AI productivity gives the system a reference rate. Noy and Zhang [2023] document a 40% time saving and an 18% quality gain on midlevel

writing tasks under random assignment to ChatGPT. Brynjolfsson et al. [2025] find a 15% productivity gain among customer-support agents in a field rollout, with the largest gains accruing to novices because the AI disseminates the practices of skilled workers, a mechanism close to what the agent roster encodes. Eloundou et al. [2024] estimate that research-adjacent occupations sit toward the high-exposure end of the LLM-impact distribution. The skeptical case is Acemoglu [2024]: under reasonable extrapolation, current AI implies at most a 0.66% TFP gain over a decade. ZeroPaper is one bet, in one high-complementarity domain (formal academic writing under fixed quality gates), about whether the realized gain in that domain exceeds the aggregate.

The pipeline's gate machinery has a more local motivation than systems-engineering hygiene. Empirical economics has a documented credibility problem: bimodal distributions of test statistics consistent with p-hacking [Brodeur et al., 2016]; replication failure rates near 40% in social-science experiments published in top journals [Camerer et al., 2018]; review delays and standards that vary systematically across author characteristics [Hengel, 2022]; and reform efforts (pre-registration, code/data sharing, transparency) that address some of these but not all [Christensen and Miguel, 2018]. These failures persist within an institutional setting where top-five placement carries outsized weight in tenure decisions [Heckman and Moktan, 2020], sharpening the incentive to publish under the existing standards rather than to reform them. ZeroPaper's adversarial evaluation is not a substitute for that institution. It is a partial automation of it: math auditors, novelty checkers, and three independent referee agents evaluate every artifact under fixed criteria the operator can inspect, and they evaluate it without knowing the author. Horton et al. [2023] validate LLMs as simulated human survey respondents, providing a partial precedent for treating LLM outputs as proxies for human judgment; the principle 4 corollaries in Section 3 state the additional conditions under which that analogy extends to adversarial evaluation of research content.

Several recent projects build autonomous or semi-autonomous research pipelines. Lu et al. [2024] describe a system that produces machine-learning papers end-to-end with sim-

ulated peer review; the architectural shape is similar (sequential stages, automated review) but the domain is empirical machine learning, with experiments rather than derivations as the central artifact, and evaluation is a single automated-reviewer pass rather than the multi-evaluator gate machinery here. Gottweis et al. [2025] build a multi-agent system for biomedical hypothesis generation and ranking; the surviving candidates are meant for human follow-up rather than developed into a finished paper. Dawid et al. [2025] propose an agentic workflow for economic research that is closest to ZeroPaper in domain (covering ideation, modeling, empirical analysis, and interpretation) but operates under human-in-the-loop checkpoints rather than autonomously. ZeroPaper differs on three dimensions across these comparisons: the target domain is a formal-theory paper in finance or macro (propositions, proofs, comparative statics); the evaluation gates are autonomous and adversarial rather than self-review or HITL; and the same pipeline runs under three different host runtimes.

Lopez-Lira and Seyfi [2026] develop a general-equilibrium model of academic journals under AI-augmented research production: AI lowers paper-production cost, raises mean paper quality, and (where AI referees are deployed) lowers reviewing cost. The paper characterizes optimal submission fees and journal capacity as functions of the AI level, and shows that without policy adjustment the quality compression that comes with cheap near-uniform AI output floods the journal and welfare collapses; with optimal fees and an expanded capacity (rising roughly 52% in their calibration), welfare rises monotonically. ZeroPaper is the supply-side instantiation of the technology that paper models: a concrete pipeline that pushes paper-production cost down and mean quality up. The two papers are independent (the companion paper does not derive from this system, and this system was not designed against the model), but they describe the same phenomenon from opposite sides of the journal.

The principles framing in Section 3 takes its cue from the architectural-patterns literature in software engineering, beginning with Alexander et al. [1977] and extending through the

object-oriented patterns of Gamma et al. [1994] and the enterprise-integration patterns of Hohpe and Woolf [2003]. The analogy is deliberate: a pattern language does not build systems, it supplies a vocabulary and a set of structural commitments against which designs can be audited. The six principles play the role those traditions reserve for patterns, not mandatory rules but commitments a system that violates one of them should document why.

The closest econ-domain predecessors describe how to use LLMs [Korinek, 2023, 2025] or build human-supervised workflows [Dawid et al., 2025]; the closest CS-domain systems target machine learning or biomedicine [Lu et al., 2024, Gottweis et al., 2025]. What is missing across both is an end-to-end autonomous pipeline that produces formal finance or macro theory under adversarial verification. ZeroPaper occupies that slot. The contribution is not the orchestrator, which is an off-the-shelf agentic CLI, and not the individual agents, most of which have analogues elsewhere; the contribution is the pipeline shape (adversarial gates, mechanical termination, multi-runtime packaging) applied to a domain where the unit of progress is a proved proposition rather than a measured effect.

6 Limitations

6.1 Output quality: calibrated but not externally validated

While the finance variant of the pipeline is designed to target the top-three finance journals (JF, JFE, RFS), my conjecture based on finance-refereeing experience is that the best of the finance-variant papers it has produced would receive a revise-and-resubmit at a peer-reviewed finance field journal such as the Journal of Financial and Quantitative Analysis under blinded review.[1] I state this as a referee's calibration, not as an independently verified claim; the natural test is to hand a blinded sample to other referees or to submit, and I have not done either. The macro variant's outputs have not been calibrated against macro-journal

[1] A companion gallery of autonomously generated papers, with their PDFs and source, is available at `https://alejandroll10.github.io/zeropaper-gallery/` (repository: `https://github.com/alejandroll10/zeropaper-gallery`); readers can form their own calibration from the sample.

standards in the same way.

6.2 Self-referential evaluation

The pipeline's quality gates are themselves LLM agents whose standards I calibrated. When the scorer returns 75, that is a prediction about a human referee's response, not the response itself. The scorer's calibration file is based on my reading of what target-journal referees would accept, and the scorer has been tuned against that reading. This calibration loop is self-referential in a way I have not fully closed: a pipeline that scores its own output against its own standards cannot detect cases where its standards are misaligned with the field's. The partial mitigation is to run three referees with different priors at Stage 6, and the mechanism referee specifically, but they are still my agents with my calibrations. A genuine external signal (submission to a real journal) is the only cure; the pipeline does not yet include that loop. The structural advantage the gates do offer over a conventional review process is mechanical rather than substantive: they apply identical criteria across runs and are blind to author identity by construction [Hengel, 2022, Camerer et al., 2018].

6.3 Domain coverage

The pipeline currently supports two variants (finance, macro) and two extensions (empirical, theory_llm). Other domains (experimental economics, development, labor, industrial organization, microeconomic theory outside the finance-macro set) would require new variant agents and, in most cases, new extensions to connect to domain-specific data sources.

6.4 Compute and latency

A fast run takes around 8 hours of wall-clock time from launch to finished paper, but that is the median path; runs with multiple scorer revision cycles or deep Stage 6 loops average 20 to 30 hours per paper. Under a single Claude Code Max subscription at the $200/month

tier, I can run three to four orchestrator sessions in parallel on one machine, each driving an independent pipeline. In practice this setup yields roughly one hundred finished papers per month per subscription at the average run length, or on the order of two dollars per paper on average (the marginal cost of one additional paper under a flat-fee subscription is approximately zero). The figures depend on the runtime, the variant, and how many revision cycles the scorer and referee trigger; empirical and theory-LLM extensions lengthen runs, and deep referee-revision loops can extend wall-clock time substantially on a single run.

At roughly two dollars per paper, the pipeline's output is not limited by inference budget but by the quality of the ideas the generator agents produce and by the human evaluation loop that would confirm or refute the quality conjecture stated above. For this pipeline's output volume, compute is no longer the scarce resource. The institutional consequences of throughput at this scale, including the journal-side response when many such pipelines run at once, are modeled in companion work [Lopez-Lira and Seyfi, 2026].

6.5 Reproducibility

Two runs with the same setup will produce different papers. This is expected (the generator agents have non-zero temperature) and mostly desirable (different runs explore different mechanisms), but it complicates evaluation. I do not yet have a good protocol for comparing a ZeroPaper output against a deterministic baseline, because the pipeline is not deterministic and the baseline (a human researcher) is also not deterministic.

6.6 Hallucinated references

The bib-verifier at Stage 8 checks every reference against the OpenAlex academic graph, confirming that the cited paper exists and that title, authors, venue, and year match. References that do not resolve are flagged and either corrected or dropped before the manuscript is finalized, and this catches the large majority of outright hallucinations. The harder problem, citing a real paper for a claim it does not actually support, is partially addressed. Web

search during citation typically surfaces the cited paper's abstract, and when the full text is not paywalled it surfaces that as well, so the paper-writer and novelty-checker agents usually have abstract-level evidence of semantic fit and occasionally full-text evidence. The remaining gap is paywalled full texts, where verification is bounded to what the abstract states. A tighter semantic-fit check (retrieving a specific chunk of the cited text that supports the claim, and failing the citation if no such chunk exists) is a natural next extension and is not yet implemented.

6.7 Mechanism drift

Theoretical papers can drift between the mechanism the text claims is at work and the mechanism the math actually delivers. The referee-mechanism agent at Stage 6 exists specifically to detect this, but it is an adversarial single agent; its misses are hard to observe. The escalation for repeated mechanism failure (force-adoption or narrow-path rewrite at round ten) is a blunt instrument that I have used when the mechanism referee cannot be satisfied, but it represents a genuine limitation: the system can produce a paper where the framing says "X is the mechanism" while the math delivers "a property consistent with X in this parameter region." Identifying mechanism drift reliably is an open problem.

6.8 Dependency on host runtime behavior

The pipeline's behavior inherits from the host runtime's behavior (Claude Code, Codex, or Gemini). When a host updates its subagent semantics, tool availability, or system-prompt handling, the runtime-specific packaging layer has to track the change. I have not automated this tracking; a runtime update that silently changes subagent isolation guarantees, for instance, would undermine the adversarial evaluation machinery in ways that would not show up until specific failure modes accumulated. This is a real operational risk for any multi-runtime system.

7 Conclusion

ZeroPaper is an autonomous pipeline for end-to-end research paper production. The system executes ten numbered stages and six adversarial gates, coordinating roughly thirty specialized agents through a versioned artifact tree, under any of three host runtimes, at around two dollars per paper amortized under a flat-fee max subscription. The three central design choices are adversarial isolation of evaluators, mechanical termination that prevents infinite loops, and multi-evaluator gates with distinct framings; multi-runtime packaging then keeps the pipeline portable across host CLIs.

The paper's second contribution is the pattern language those choices come from. Ten premises about LLM behavior and deployment economics (self-bias, long-context degradation, coherence drift, stochastic error, path-of-least-resistance, reads-any-text, judges-predicates, fresh-instance sampling, tokens-cost, infrastructure-fails) force six principles (Markov machine with external control flow; context is costly; delegate; verify, do not trust; termination must be mechanical; parallelize independent dispatches). The principles explain which architectural choices in ZeroPaper were forced and which were contingent. They are falsifiable: a pipeline that implements the principles and still fails either violates a premise the language missed or implements a principle incorrectly, and the language should be revised.

Not every run produces a publishable paper. The pipeline's structure avoids the failure modes I have most often seen degrade automated-research prototypes, but whether the surviving papers clear a real journal's bar is a question the system cannot answer from inside itself. Submission is the only test. In the interim, and based on my refereeing experience (Section 6), the best produced papers would receive a revise-and-resubmit at a finance field journal under blinded review.

I release the system at `https://github.com/alejandroll10/zeropaper` so other researchers can run the pipeline autonomously, develop a seed idea of their own into a paper, or modify the variant, host runtime, or extension layer for domains the paper does not cover. The repository is open, the setup is one script, and the pipeline runs the same under three

different hosts. The system produces a starting point for a human researcher's revision, not a replacement for the researcher; the name marks the absence of human contribution to the generation.

References

Daron Acemoglu. The simple macroeconomics of AI. *Economic Policy*, 40(121):13–58, 2024.

Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

Nicholas Bloom, Charles I. Jones, John Van Reenen, and Michael Webb. Are ideas getting harder to find? *American Economic Review*, 110(4):1104–1144, 2020.

Abel Brodeur, Mathias Lé, Marc Sangnier, and Yann Zylberberg. Star wars: The empirics strike back. *American Economic Journal: Applied Economics*, 8(1):1–32, 2016.

Erik Brynjolfsson, Danielle Li, and Lindsey Raymond. Generative AI at work. *Quarterly Journal of Economics*, 140(2):889–942, May 2025. Earlier version: NBER Working Paper 31161.

Colin F. Camerer et al. Evaluating the replicability of social science experiments in Nature and Science between 2010 and 2015. *Nature Human Behaviour*, 2:637–644, 2018.

David Card and Stefano DellaVigna. Nine facts about top journals in economics. *Journal of Economic Literature*, 51(1):144–161, 2013.

Garret Christensen and Edward Miguel. Transparency, reproducibility, and the credibility of economics research. *Journal of Economic Literature*, 56(3):920–980, 2018.

Tyler Cowen and Alexander T. Tabarrok. How to learn and teach economics with large language models, including GPT. Working paper, George Mason University, Mercatus Center, 2023. SSRN 4391863.

Herbert Dawid, Philipp Harting, Hankui Wang, Zhongli Wang, and Jiachen Yi. Agentic workflows for economic research: Design and implementation. *arXiv preprint arXiv:2504.09736*, 2025.

Andrea L. Eisfeldt and Gregor Schubert. AI and finance. Working Paper 33076, National Bureau of Economic Research, 2024.

Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. GPTs are GPTs: Labor market impact potential of LLMs. *Science*, 384:1306–1308, 2024.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Juraj Gottweis et al. Towards an AI co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.

James J. Heckman and Sidharth Moktan. Publishing and promotion in economics: The tyranny of the top five. *Journal of Economic Literature*, 58(2):419–470, 2020.

Erin Hengel. Publishing while female: Are women held to higher standards? evidence from peer review. *Economic Journal*, 132(648):2951–2991, 2022.

Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

John J. Horton, Apostolos Filippas, and Benjamin Manning. Large language models as simulated economic agents: What can we learn from Homo Silicus? Working Paper 31122, National Bureau of Economic Research, 2023.

Manish Jha, Jialin Qian, Michael Weber, and Baozhong Yang. ChatGPT and corporate policies. Working Paper 32161, National Bureau of Economic Research, 2024.

Anton Korinek. Generative AI for economic research: Use cases and implications for economists. *Journal of Economic Literature*, 61(4):1281–1317, 2023.

Anton Korinek. AI agents for economic research. Working Paper 34202, National Bureau of Economic Research, 2025.

Alejandro Lopez-Lira and Seyed Mohammad Sina Seyfi. One prompt, one paper: Optimal journal policy and the AI submission flood. *SSRN Working Paper*, (6337880), 2026. `https://ssrn.com/abstract=6337880`.

Alejandro Lopez-Lira and Yuehua Tang. Can ChatGPT forecast stock price movements? return predictability and large language models. *Review of Financial Studies*, 2023. SSRN 4412788; forthcoming.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI Scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Jens Ludwig, Sendhil Mullainathan, and Ashesh Rambachan. Large language models: An applied econometric framework. Working Paper 33344, National Bureau of Economic Research, 2025.

Shakked Noy and Whitney Zhang. Experimental evidence on the productivity effects of generative artificial intelligence. *Science*, 381(6654):187–192, 2023.