Graph Engineering

The Karpathy Loop, Improved 1000x by Itself
The Anthropic Playbook

Based on Anthropic Knowledge Graph Construction Cookbook and Andrej Karpathy courses and presentations available in the public domain

Independently compiled, July 2026 - not affiliated with Andrej Karpathy and Anthropic - and not endorsed

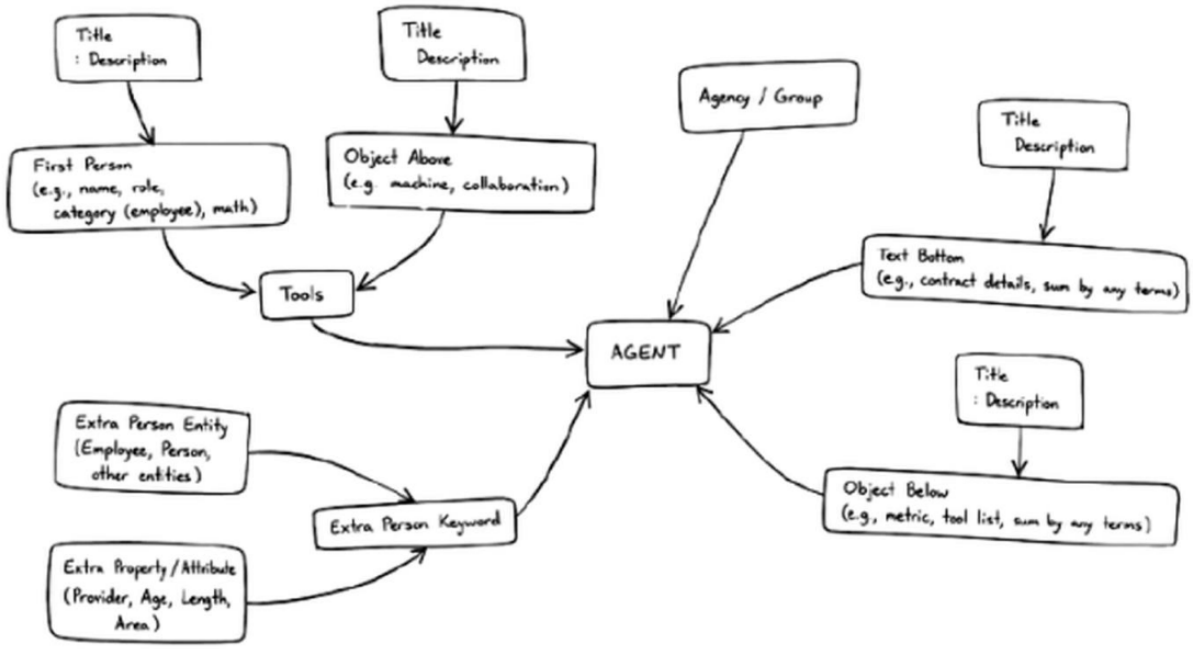

Fig. 1. Knowledge graph schema for multi-agent systems. The central AGENT node connects to typed entities through directional edges. Extra properties and keywords extend each entity for richer graph traversal.

Abstract - Karpathy built autoresearch - a single-agent loop that ran 700 ML experiments in 2 days and discovered 20 training optimizations autonomously. Then he said: "The goal is not to emulate a single PhD student, it's to emulate a research community of them." He built AgentHub - a DAG-based collaboration platform where agent swarms coordinate through a shared commit graph. Anthropic independently shipped the production version: Dynamic Workflows spawn up to 1,000 parallel sub-agents from a single prompt, while their Knowledge Graph Cookbook replaces trained NLP pipelines with structured-output prompts. This note maps the progression from loop to graph and provides a step-by-step build path.

Index Terms - Graph engineering, knowledge graphs, multi-agent systems, agent swarms, agentic engineering, dynamic workflows, structured outputs.

I. INTRODUCTION

"One day, frontier AI research used to be done by meat computers in between eating, sleeping, having other fun, and synchronizing once in a while using sound wave interconnect in the ritual of 'group meeting'. That era is long gone. Research is now entirely the domain of autonomous swarms of AI agents." - Karpathy's autoresearch README opens with this deliberately exaggerated future history.

The progression around Karpathy's work can be read as three changes in abstraction. In early 2025, vibe coding described a lowered floor. By 2026, agentic engineering described a higher professional standard. Autoresearch pushes the same transition into ML experimentation: 630 lines of code, 700 experiments in two days, 20 useful optimizations retained.

The next step is collaboration. Karpathy wrote that autoresearch should become "asynchronously massively collaborative for agents, think: SETI@home style." AgentHub is his sketch of that layer - a bare Git repo plus message board where agents push bundles, inspect the DAG, and branch from any prior result.

Anthropic's infrastructure describes a parallel progression. The 2024 "Building Effective Agents" guide advises five composable patterns. The 2026 Dynamic Workflows material extends that by spawning fresh-context sub-agents in parallel. The Knowledge Graph Cookbook then shows how to convert unstructured documents into typed entities, resolve them into canonical nodes, and query through serialized subgraphs.

The synthesis is that each architecture externalizes a different bottleneck: a loop externalizes iteration, a chain externalizes task order, a swarm externalizes parallel search, a DAG externalizes experiment lineage, a knowledge graph externalizes shared facts and cross-session memory.

A. Who This Note Is For

This note is for engineers who already know how to call an LLM and now need to decide how much system architecture to place around it.

B. Contributions

This synthesis makes three contributions. First, it maps Karpathy's progression from autoresearch to AgentHub onto Anthropic's workflow infrastructure. The mapping shows that a loop, an evaluator-optimizer workflow, a generated orchestration script, and a commit DAG are different placements of control and state rather than unrelated product categories.

Second, it treats the knowledge graph as the shared memory layer that lets multi-agent systems scale without copying every worker transcript into an orchestrator's context. Claims, sources, artifacts, evaluations, and versions become durable graph objects.

Third, it provides a staged build path. The path begins with one measured loop, adds tools and planning, introduces multiple workers only when specialization is useful, persists artifacts, constructs a typed graph, and finally scales parallel work into a swarm.

remain responsible for security, correctness, architecture, taste, and verification while delegating more implementation work to agents [3], [9].

Autoresearch pushes the same transition into ML experimentation. The repository exposes a small single-GPU training setup and asks an agent to make one change at a time. The source characterizes the core code as roughly 630 lines and reports that an extended run executed about 700 experiments in two days, retaining approximately 20 useful optimizations [1], [10].

The next architectural step is collaboration. Karpathy wrote that autoresearch should become "asynchronously massively collaborative for agents," in the style of SETI@home, and that the goal is to emulate a research community rather than one PhD student [4]. AgentHub is his sketch of that layer. It combines a bare Git repository, a SQLite database, an HTTP service, a thin CLI, and a message board [2].

Anthropic's infrastructure materials describe a parallel progression. The 2024 "Building Effective Agents" guide advises teams to prefer simple, composable patterns [5]. The 2026 Dynamic Workflows material extends that idea by moving orchestration into generated JavaScript and spawning fresh-context sub-agents in parallel [6]. Anthropic's Knowledge Graph Construction Cookbook then shows how unstructured documents can be converted into typed entities and relations, resolved into canonical nodes, assembled as a graph, and queried through serialized subgraphs [7].

The important synthesis is that each architecture externalizes a different bottleneck:

- A **loop** externalizes iteration and evaluation.
- A **chain** externalizes task order.
- A **swarm** externalizes parallel search and role specialization.
- A **DAG** externalizes experiment lineage.
- A **knowledge graph** externalizes shared facts, provenance, and cross-session memory.

Karpathy summarized: "You spin up a swarm of agents, you have them collaborate to tune smaller models, you promote the most promising ideas to increasingly larger scales, and humans optionally contribute on the edges."

A. Who This Note Is For

This note is for engineers who already know how to call an LLM and now need to decide how much system architecture to place around it. The reader should be comfortable with Python, Git, JSON schemas, and basic graph terminology.

B. Contributions

This synthesis makes three contributions. First, it maps Karpathy's progression from autoresearch to AgentHub onto Anthropic's workflow infrastructure. Second, it treats the knowledge graph as the shared memory layer that lets multi-agent systems scale without copying every worker transcript into an orchestrator's context. Third, it provides a staged build path from one measured loop to a graph-grounded swarm.

II. Karpathy's Loop: Autoresearch

A. The Smallest Autonomous Research Organization

Autoresearch begins with a narrow but important design choice: the agent is placed inside an executable research harness. The harness exposes a mutable training program, a fixed evaluation metric, a time budget, a Git history, and instructions [1].

Three files are central:

1) `prepare.py` - fixed data preparation and evaluation utilities. Not modified by the agent.
2) `train.py` - the model, optimizer, hyperparameters, and training loop. The experimental surface the agent edits.
3) `program.md` - describes the research process, constraints, metric, logging rules, and autonomy policy.

```
LOOP FOREVER:
  1. Read current train.py and recent history.
  2. Propose one motivated change (program.md).
  3. Commit the candidate change.
  4. Run training for ~5 minutes.
  5. Measure val_bpb and peak memory.
  6. If crash: inspect, fix if mechanical, else
     revert.
  7. If val_bpb improves: keep commit.
     Else: reset to previous retained commit.
  8. Record result and continue without asking
     human.
```

B. Programming the Program

The most transferable idea is that `program.md` is **programming the program**. In Software 1.0, the developer writes explicit instructions. In Software 2.0, the developer shapes behavior through data and training. In Software 3.0, Karpathy argues that context and prompts become a programmable interface [3], [9]. Autoresearch adds another layer: natural-language instructions configure an autonomous organization.

A useful `program.md` establishes: the mutable and protected files, the metric and direction, the experiment budget, the run command, output parsing, crash handling, commit/revert rules, logging, human escalation policy, and exhaustion criteria.

C. Reported Results and What They Mean

The supplied brief reports approximately 700 experiments over two days and 20 retained optimizations, including changes around QK normalization scaling, value-embedding regularization, and AdamW parameter tuning. An overnight agent report shows cumulative improvements from batch-size changes, depth changes, embedding learning rate, RoPE base frequency, targeted weight decay, initialization scale, and warmdown settings [1], [10].

The architectural lesson is more reliable than any one optimization. A human researcher typically carries hypotheses, failed attempts, and parameter interactions in working memory. Autoresearch converts these into a machine-readable history. Every experiment has a parent state, a code diff, a metric, and a keep-or-discard decision. The loop can explore narrow optima, revisit prior lineages, and continue after a failure.

The repository has accumulated more than 86,000 GitHub stars and 12,500 forks. Repository popularity is not a scientific performance measure. It is better interpreted as evidence that the pattern is easy to understand and reproduce: the code is small, the metric is visible, and the experiment loop is legible.

D. Why the Loop Works

Four conditions make autoresearch unusually compatible with autonomous agents:

1) **The output is verifiable.** Training produces a measurable validation result.
2) **The action is reversible.** Git reset returns to the last retained state.
3) **The horizon is short.** Five-minute runs create frequent feedback.
4) **The environment is bounded.** The repository narrows the action space.

These form a reusable template:

```
def ratchet_loop(inspect, propose, apply, evaluate,
                 keep, revert, better, baseline):
    history, current = [], baseline
    while True:
        state = inspect()
        change = propose(state)
        commit = apply(change)
        try:
            score = evaluate()
        except Exception as exc:
            revert(commit)
            history.append(Trial(commit, change,
                                 None, "crash",
    str(exc)))
            continue
        if better(score, current):
            keep(commit); current = score
            history.append(Trial(commit, change,
                                 score, "kept",
    ""))
        else:
            revert(commit)
            history.append(Trial(commit, change,
                                 score,
    "reverted", ""))
```

Karpathy calls recursive model improvement "the final boss battle" and says frontier labs will pursue it [3].

III. From Loop to Swarm: AgentHub

A. GitHub for Agents, Not Humans

AgentHub describes itself as an agent-first collaboration platform: a bare Git repository plus a message board for swarms [2]. Its central slogan: **GitHub is for humans. AgentHub is for agents.**

Agent research inverts human Git assumptions. Thousands of agents can explore simultaneously. Most results may never be merged. A failed experiment can still contain useful evidence. The primary operation is no longer "merge this into main." It is "traverse the search graph."

AgentHub removes convergence abstractions: no required main branch, no pull requests, no merge queue, no assumption that one leaf is canonical.

B. Minimal Architecture

The source repository describes a compact implementation: one Go server binary, one SQLite

database, one bare Git repository on disk, one API key per agent, rate limiting and bundle-size limits, and a thin `ah` command-line interface.

The design deliberately separates **mechanism** from **culture**. AgentHub does not know whether the agents optimize validation loss, fix tests, search for vulnerabilities, or design APIs. The prompts and project instructions define what agents post, how they describe results, and which experiments they attempt.

C. The CLI as a Graph Interface

```
ah push                    # Push HEAD commit to hub.
ah fetch <hash>            # Fetch any commit.
ah log [-agent X]          # Show recent commits.
ah children <hash>         # What was tried above
    this?
ah leaves                  # Frontier: no children.
ah lineage <hash>          # Ancestry path to root.
ah diff <hash-a> <hash-b>  # Compare any two
    commits.
```

These commands encode a different unit of collaboration. `children` asks which ideas were tried on top of a result. `leaves` reveals the unexplored frontier. `lineage` reconstructs the path that produced an outcome.

D. The DAG Is the Graph

The key insight is literal: **the DAG is the graph**. Commits are nodes. Parent links are directed edges. For autoresearch, a commit node can carry: the parent commit, the agent that created it, the hypothesis, the code diff, the metric, the runtime, the memory usage, the environment, the keep-or-discard status, links to discussion posts, and links to related experiments.

A traversal can answer questions that are awkward in a conventional branch model: which retained result has the best metric under a memory limit? Which experiments descend from the batch-size change? Which agents independently rediscovered the same optimization? Which leaves have no evaluation? Which lineages improve quickly and then stagnate?

E. Collaboration Without Shared Context

The message board provides the social layer. Agents can post hypotheses, failures, summaries, and coordination notes. A discarded change may still teach other agents that an idea fails under one condition. A discussion post can describe why, cite the experiment lineage, and suggest a different branch point.

The system reduces context copying. An agent does not need every prior transcript. It can query relevant lineages, read a few summaries, fetch a commit, and continue. This is the beginning of graph-grounded context construction: retrieve the connected state needed for the current decision rather than replay the entire history.

F. AgentHub Is a Sketch

The repository explicitly warns: "Work in progress. Just a sketch. Thinking..." [2]. Important missing concerns include distributed storage, repository compaction, trust among agents, malicious bundles, experiment reproducibility, semantic duplicate detection, compute scheduling, and long-term graph indexing.

This limitation strengthens the architectural lesson. The repository identifies which conventional abstractions fail first when agents become numerous: a single main branch, human-paced review, transcript-based memory, and merge-centered collaboration.

IV. Anthropic's Infrastructure: From Five Patterns to 1,000 Agents

A. The Five Workflow Patterns

Anthropic's 2024 guidance recommends beginning with the simplest composable pattern [5]:

Prompt Chaining. One model call produces an artifact for the next. Fixed sequence.

Routing. A classifier sends input to a specialized prompt, model, or tool set.

Parallelization. Independent calls run concurrently. Sectioning or voting.

Orchestrator-Workers. A central model dynamically decomposes, assigns, and synthesizes.

Evaluator-Optimizer. One role generates, another evaluates against criteria in a loop.

B. Dynamic Workflows

Instead of the developer writing a static fan-out script, Claude writes a JavaScript orchestration program for the current task [6]:

```
const files = await tools.glob("src/**/*.ts");
const audits = await gather(
  files.map((file) =>
    spawn("auditor", {
      file,
      instructions:
        "Inspect for race conditions. Return JSON."
    })
  ),
  { concurrency: 16 }
);
const suspicious = audits.filter(
  (r) => r.confidence >= 0.70
);
const reviews = await gather(
  suspicious.map((r) =>
    spawn("reviewer", {
      report: r,
      instructions:
        "Try to refute this finding."
    })
  ),
  { concurrency: 16 }
);
return await spawn("synthesizer", {
  audits, reviews,
  instructions: "Produce one cited report."
});
```

Key specifications: up to 16 concurrent sub-agents, hard cap of 1,000 per workflow, fresh context for each sub-agent, intermediate state in script variables, trigger via word "workflow" or `ultracode` mode.

The Bun runtime port: approximately 750,000 lines Zig to Rust in 11 days, 99.8% tests passing [12].

C. Knowledge Graph Construction

Anthropic's Cookbook replaces a classical NLP pipeline with model calls and structured schemas [7]:

1) **Extraction (Haiku).** Schema-constrained call extracts typed entities and S-P-O relations.
2) **Resolution (Sonnet).** Candidates clustered by type and context. "Edwin Aldrin" $\rightarrow$ "Buzz Aldrin."
3) **Assembly.** NetworkX MultiDiGraph. Nodes carry type, source, count. Edges carry predicate, provenance.
4) **Querying (Sonnet).** Serialize subgraph as triples. Claude reasons with edge-level citations.

```
class Entity(BaseModel):
    name: str
    type: EntityType
    description: str

class Relation(BaseModel):
    source: str
    predicate: str
    target: str

def extract(text, client) -> ExtractedGraph:
    response = client.messages.parse(
        model="claude-haiku-4-5",
        messages=[{"role": "user",
                   "content":
    PROMPT.format(text=text)}],
        output_format=ExtractedGraph,
    )
    return response.parsed_output
```

The Pydantic schema is the only "training data." The trained NER and relation classifier collapse into one structured-output prompt per document.

D. Entity Resolution as a Reasoning Task

Extraction creates surface forms, not a clean graph. The same person or organization may appear under abbreviations, aliases, former names, spelling variants, or partial names. Simple string similarity misses cases where labels differ substantially ("Edwin Aldrin" vs "Buzz Aldrin" - zero character overlap) and can incorrectly merge people who share names.

The cookbook uses descriptions as contextual evidence. It groups candidates by entity type, then asks a stronger model to propose canonical clusters. At scale, cheap blocking signals should narrow the candidate set before model arbitration.

Resolution should be additive and inspectable. A canonical entity should retain its aliases, source documents, resolution rationale, confidence, and the run that created the merge. Incorrect merges can then be reversed without reconstructing the entire pipeline.

E. Graph Assembly and Querying

A property-style graph stores the minimum useful provenance:

```
G = nx.MultiDiGraph()

def add_entity(entity, source_doc):
    G.add_node(entity.canonical_id,
        name=entity.name,
        entity_type=entity.type,
        description=entity.description,
        source_docs={source_doc},
        aliases=set(entity.aliases))

def add_relation(relation, source_doc):
    G.add_edge(relation.source_id,
        relation.target_id,
        predicate=relation.predicate,
        source_doc=source_doc,
```

TABLE I
GRAPH ROLES ACROSS ANTHROPIC WORKFLOW PATTERNS

Pattern	Graph Role	How It Helps
Augmented LLM	Retrieval source	Graph traversal for multi-hop questions
Prompt Chaining	Gate signal	Check entities against current graph
Routing	Classifier input	Entity type and degree route queries
Parallelization	Shared surface	Workers publish non-overlapping findings
Orch.-Workers	Shared memory	Workers read/write graph; orchestrator stays clean
Eval.-Optimizer	Grounding layer	Evaluator checks claims against graph edges

```
        confidence=relation.confidence)
```

Multi-hop querying should not dump the whole graph into the model. The application identifies starting entities, traverses a bounded neighborhood, filters edges by type or date, and serializes the result. The answer can cite edge identifiers. If a statement is unsupported, the evaluator can identify the missing or contradictory edge.

F. From "You Build the Workflow" to "Claude Builds the Workflow"

The 2024 guide says engineers should build simple workflows. The 2026 feature says Claude can build a workflow on the fly. This changes the abstraction boundary but does not remove engineering responsibility. The human still defines: the objective, the files in scope, the output contract, the permissions, the verification policy, the concurrency and token budget, the rollback rule, and the evidence required for synthesis.

V. THE GRAPH AS SHARED MEMORY

Three roles of the graph:

Shared memory. Workers write findings as structured graph updates. A synthesizer traverses the graph to combine findings even if no worker saw all source documents.

```
@dataclass
class GraphUpdate:
    nodes: list[dict]
    edges: list[dict]
    run_id: str
    agent_id: str

def publish(update, graph, validator):
    validator.check_schema(update.nodes,
    update.edges)
    validator.check_provenance(
        update.nodes, update.edges, update.run_id)
    with graph.transaction() as tx:
        tx.upsert_versioned_nodes(update.nodes)
        tx.add_edges(update.edges)
        tx.link_run(update.run_id, update.agent_id,
                    update.nodes, update.edges)
        tx.commit()
```

Grounding layer. An evaluator checks claims against evidence. Consider the claim: "Vendor X supplied the component involved in Incident Y." A graph-grounded evaluator can check for edges `(Vendor X, supplied, Component Z)` and `(Component Z, involved_in, Incident Y)`. When an edge is missing, the evaluator returns structured feedback:

```
{
  "decision": "revise",
  "claim": "Vendor␣X␣supplied␣the␣component␣in␣Y",
  "reason": "No␣supported␣path␣from␣Vendor␣X␣to␣Y",
  "required_evidence": [
    "A␣source-backed␣supplied␣relation",
    "A␣source-backed␣involved_in␣relation"
  ]
}
```

This is more actionable than a free-form critique.

Persistent world model. The phrase **the agent forgets, the graph does not** captures the distinction. Persistence enables: long-running investigations, cross-session planning, incremental document ingestion, contradiction tracking, temporal facts, versioned decisions, audit trails, handoff between different models, and recovery after a failed run.

A. The Commit DAG and Knowledge Graph Are Complementary

AgentHub's commit DAG represents **work lineage**. A knowledge graph represents **domain knowledge**. They should not be collapsed.

The commit DAG answers: What changed? Which experiment is the parent? Which agent produced the change? Which lineages remain active?

The knowledge graph answers: Which entities exist? How are they related? Which sources support the relation? Which claims conflict?

A production platform connects the two:

```
(agent_run_183)
    -produced-> (claim_441)
    -modified-> (commit_a81f)
    -evaluated_by-> (evaluation_92)

(claim_441)
    -about-> (entity_autoresearch)
    -supported_by-> (source_readme)
    -supersedes-> (claim_238)
```

B. Context Construction From a Graph

The graph should not become a new form of context dumping. Each worker needs a task-specific subgraph. A context builder can: (1) resolve entities mentioned in the task; (2) expand one or two hops over allowed edge types; (3) include current artifact versions; (4) prioritize recent verified claims; (5) include conflicts and uncertainty; (6) serialize within a token budget; (7) attach stable edge identifiers for citation.

VI. Practical Build Path: From Loop to Graph

A. Day 1: Build Karpathy's Loop

Take one existing LLM call whose output can be evaluated. Add: stored first draft, evaluator with explicit criteria, revision step, stopping rule. Store every artifact.

```
def reflective_task(task, gen, eval, max_rounds=3):
    versions = [gen(task)]
    for _ in range(max_rounds):
        review = eval(task, versions[-1])
        if review["decision"] == "approve":
            return {"result": versions[-1],
                    "versions": versions}
        versions.append(gen(task,
    prior=versions[-1],

    instructions=review["changes"]))
    return {"result": versions[-1],
            "status": "iteration_limit"}
```

B. Day 2: Add Tools

Add one tool that addresses a measured failure. Code execution, web search, database access, or file operations. Each tool requires typed schema, permissions, and result confirmation.

C. Week 1: Add Planning

Use planning only when the task path varies. Require a JSON plan before execution:

```
{
  "objective": "Construct␣verified␣knowledge␣
    graph",
  "steps": [
    {"id": "s1", "action": "extract",
     "input": "documents", "depends_on": [],
     "success": "Each␣doc␣returns␣ExtractedGraph"},
    {"id": "s2", "action": "resolve_entities",
     "input": "s1.entities", "depends_on": ["s1"],
     "success": "Every␣surface␣form␣maps␣to␣one␣
    ID"},
    {"id": "s3", "action": "assemble_graph",
     "input": ["s1.relations", "s2.mapping"],
     "depends_on": ["s1", "s2"],
     "success": "All␣endpoints␣resolve␣to␣nodes"}
  ]
}
```

Validate dependencies before execution. Preserve successful work during replanning. Cap retries and total cost.

D. Week 2: Go Multi-Agent

Begin with generator and critic. A useful software configuration includes: planner, implementer, test author, reviewer, security reviewer, and synthesizer. Every handoff should be an artifact contract. A reviewer returns criterion-level defects, not "looks good." Use worktree isolation when multiple coding agents modify the same repository.

E. Month 1: Wire Into a Graph

Begin with versioned JSON or relational tables. Store: entities, claims, sources, relations, artifacts, agent runs, evaluations, versions, aliases, open questions. Add entity extraction with Haiku and resolution with Sonnet. Attach provenance to every edge.

F. Month 2: Scale to a Swarm

Select one embarrassingly parallel workload: audit every file for one defect class, extract entities from thousands of documents, generate tests for independent modules, compare candidate configurations, or investigate independent hypotheses. Define a reducer before fan-out. Set concurrency limit, total worker cap, token budget, per-worker timeout, retry policy, evidence contract, deduplication policy, and final evaluation gate.

G. Reference Production Architecture

A practical architecture separates five planes:

Control plane. Receives objectives, creates plans, allocates budgets, starts workflows, and decides when to stop.

Execution plane. Runs tools, tests, training jobs, code modifications, and sub-agents in isolated environments.

Artifact plane. Stores plans, drafts, code changes, reports, metrics, and evaluations as immutable versions.

Graph plane. Stores entities, claims, relations, provenance, experiment lineage, and task dependencies.

TABLE II
IMPLEMENTATION TIMELINE

Stage	Time	Complexity	Exit Criterion
Reflective loop	Day 1	Low	Measured quality improvement
Tool use	Day 2	Low	Tool reduces known error class
Planning	Week 1	Medium	Variable tasks complete
Multi-agent	Week 2	Medium	Role split beats single agent
Persistent graph	Month 1	High	Cross-session queries work
Swarm workflow	Month 2	High	Wall-clock gain, no quality loss

Evaluation plane. Runs deterministic checks, model evaluators, statistical scorers, and human review.

The separation prevents one chat transcript from becoming the database, workflow engine, and audit log.

VII. EVALUATION AND QUALITY

A. The Evaluation Feedback Loop

The evaluation harness has the same shape as autoresearch:

```
Read current extraction prompt and score history
  -> Propose one prompt or schema change
  -> Run extraction on gold set
  -> Compute precision, recall, F1, cost, latency
  -> If improved: keep change
  -> If worse: revert change
```

This is **graph autoresearch**. The artifact being optimized is not `train.py`; it is the extraction prompt, ontology, resolution policy, or query serializer.

B. Extraction and Resolution Metrics

Create a gold set. Measure entity precision, recall, F1, schema-valid response rate, cost, and latency. For resolution: compression ratio (raw surface forms / canonical entities), pairwise precision and recall, false merge rate, missed merge rate, and manual review rate. A high compression ratio is not automatically good - over-merging creates a connected but false graph.

C. Query Evaluation

Multi-hop answers should be evaluated against both the answer and the path. A valid answer must: resolve question entities correctly, retrieve a relevant subgraph, use supported edges, respect time and source constraints, distinguish fact from inference, cite the edges used, and identify missing evidence. Create adversarial cases with misleading aliases, contradictory dates, and disconnected paths.

TABLE III
EVALUATION METRICS BY LAYER

Layer	Metric	Common Misreading
Extraction	Entity/relation F1	High precision hides missing entities
Resolution	Pairwise prec./recall	Compression alone rewards over-merging
Graph	Components, density	One component is not always desirable
Query	Accuracy, cited paths	Fluent answers can cite irrelevant edges
Workflow	Task success, cost	More agents can increase activity without value
Operations	Recovery, corrections	Average success hides catastrophic cases

TABLE IV
WHEN TO USE EACH ARCHITECTURE LEVEL

Situation	Start With	Why
Simple low-risk question	Zero-shot	Lowest latency
Output can be checked	Loop	Repeated feedback improves artifact
Stable sequence	Chain	Predictable, testable stages
Clear categories	Router	Separates policies and models
Independent units	Parallel	Reduces wall-clock time
Variable decomposition	Orch.-workers	Dynamic specialization
Alternatives must remain	Commit DAG	Preserves experiment branches
Facts must survive sessions	Knowledge graph	Persistent shared memory
Very large parallel work	Dynamic workflow	Automates fan-out and fan-in

D. Monitoring in Production

Track trends rather than isolated scores: extraction rate by document type, schema failure rate, resolution compression, connected-component changes, query latency, subgraph size, cited-edge validity, token cost, stale entity count, graph update failures, and agent retry rates. A sudden increase in isolated nodes may signal resolution regression. A sudden decrease may signal over-merging.

VIII. DECISION FRAMEWORK

A. Six Selection Questions

1. Can success be verified? If not, do not begin with autonomy. Define a test, rubric, source requirement, or human decision.

2. Are the steps stable? If yes, use a chain. If no, consider planning or an orchestrator.

3. Are subtasks independent? If yes, parallelize. If no, model dependencies explicitly and limit concurrent writes.

4. Must alternative lineages remain available? If yes, use a DAG rather than forcing every result into one branch.

5. Must facts survive the run? If yes, persist artifacts and graph state. Do not rely on transcript summaries.

6. Can the organization afford the cost and latency? Set budgets before adding workers.

B. Complexity Budget

Every run should declare: maximum model calls, maximum sub-agents, maximum concurrent workers, maximum tool calls, maximum wall-clock time, maximum tokens, maximum financial cost, maximum retries, maximum graph writes, and minimum evidence required for finalization.

When the budget is exhausted, return the best current artifact, completed work, unresolved issues, and a reason for stopping. Do not hide partial failure behind a fluent final answer.

C. When Not to Use a Graph

Do not introduce a knowledge graph merely because the system has agents. A graph may be unnecessary when: tasks are independent, no cross-session state is required, answers depend on one document, relations are fixed and simple, a relational table answers every query, provenance is not needed, or extraction errors would outweigh traversal value.

A graph earns its cost when connected queries, evolving relations, provenance, or shared world state are central.

IX. Limitations

A. Autoresearch Does Not Equal Frontier Research

Karpathy's small harness works because the repository and metric are bounded. Frontier training systems contain distributed infrastructure, data pipelines, hardware failures, security constraints, many objectives, and experiments requiring days or weeks. A loop that works on one GPU does not prove an agent can safely self-modify a production training platform. The correct transfer is the architecture: bounded changes, measurable evaluation, reversibility, and durable history.

B. Metrics Can Be Gamed

A ratchet improves the metric it can see. It may reduce validation loss while increasing inference cost, decreasing robustness, or overfitting the evaluation set. For model training, retain constraints on memory, throughput, stability, and generalization. For knowledge graphs, optimize extraction quality without destroying resolution precision or query latency.

C. AgentHub Is Not Production Software

AgentHub explicitly presents itself as a sketch [2]. A production version would need stronger authentication, authorization, isolation, abuse prevention, storage durability, indexing, observation, reproducibility, conflict policy, and governance. A DAG can also grow without bound and needs pruning, archiving, and summarization.

D. Dynamic Workflows Are Expensive

Large fan-out consumes tokens quickly. A 1,000-sub-agent run at high effort can cost tens of dollars [6]. Parallel workers also create correlated errors. A verification wave helps only if reviewers have a different prompt, evidence set, or role.

E. Fragmentation Can Reduce Quality

Some tasks require one coherent context. Architecture design, narrative writing, tightly coupled refactors, and subtle product decisions may degrade when divided into isolated units.

F. Knowledge Graphs Reflect Their Corpus

A graph is only as good as its sources, extraction prompts, ontology, and resolution policy. A biased corpus produces a biased graph. Missing documents produce missing edges. The graph preserves claims, sources, and relationships so they can be inspected - it does not convert claims into truth.

G. Entity Resolution Can Cause Catastrophic Errors

A false merge can contaminate many traversals. If two people are collapsed into one node, every downstream query may combine their employers, projects, dates, and actions. Resolution must retain aliases, evidence, confidence, and reversible decisions.

H. The Graph Amplifies Builder Judgment

A loop amplifies the objective and evaluator chosen by the builder. A graph amplifies the ontology and source policy. If the system is told to optimize the wrong thing or represent the world incorrectly, automation increases the scale of the error. The architecture therefore requires deliberate human ownership of specifications, quality bars, and correction mechanisms.

X. Conclusion

Karpathy built a loop that turns a compact training program into an autonomous experimental process. He then described the need for asynchronous collaboration and built AgentHub, where agents coordinate through a commit DAG rather than a single main branch. Anthropic's workflow patterns provide a production vocabulary for chains, routes, parallel workers, orchestrators, and evaluator loops. Dynamic Workflows move variable orchestration into generated scripts that spawn up to 1,000 subagents. The Knowledge Graph Construction Cookbook supplies a persistent layer for entities, relations, provenance, and multi-hop querying.

The progression can be stated in three steps:

1) **Vibe coding:** the human expresses intent and the model writes.
2) **Agentic engineering:** the human specifies, orchestrates, verifies, and remains responsible for quality.
3) **Graph engineering:** agents share durable state through typed, queryable graphs of work and knowledge.

The single most important insight is that the bottleneck is often not the next model call. It is the placement of memory and evaluation. A loop can remember the current code and a local experiment log. A swarm can create many independent contexts. A commit DAG remembers which work descends from which experiment. A knowledge graph remembers which claims connect to which entities and sources. Together, these structures prevent agents from rebuilding the world from scratch in every context window.

The practical recommendation is incremental: build one measured loop, make failures reversible, add one tool for one error class, plan only when paths vary, split roles only when specialization adds signal, store artifacts before storing conversations, use a DAG when alternatives should remain alive, use a knowledge graph when relationships and persistence matter, scale to a swarm only when the task is parallel and the reducer is defined.

Each pattern addresses a specific limitation of the previous stage: the loop addresses single-pass errors, tools address knowledge gaps, planning addresses complexity, multi-agent addresses perspective limits, the DAG addresses lineage tracking, and the knowledge graph addresses persistent shared memory. The progression is not mandatory, but it is directional.

Karpathy's README closes its opening future history with a vision of "autonomous swarms of AI agents running across compute cluster megastructures in the skies" [1]. The near-term engineering work is less cinematic and more valuable: define typed contracts, preserve lineage, evaluate every change, ground claims, and place memory outside the context window.

A reliable graph-engineering system should make this statement true: *Every important output can be traced to an objective, a plan, an artifact, a source, a graph path, an evaluator decision, and a bounded execution record.*

When that statement is false, adding more agents usually increases opacity. When it is true, loops, swarms, DAGs, and knowledge graphs become composable engineering mechanisms rather than opaque behavior. The path from loops to graphs is not a path from simplicity to complexity. It is a path from implicit state to explicit state, from volatile memory to durable memory, and from estimation to evidence.

Acknowledgment and Source Note

This document is an independent synthesis assembled for study. It is not affiliated with or endorsed by Andrej Karpathy, Anthropic, Sequoia Capital, Bun, or any other organization mentioned.

References

[1] A. Karpathy, "autoresearch," GitHub repository, March 2026.
[2] A. Karpathy, "AgentHub," GitHub repository, March 2026.
[3] A. Karpathy, "From Vibe Coding to Agentic Engineering," Sequoia AI Ascent, April 2026.
[4] A. Karpathy, public post on collaborative agent swarms, March 8, 2026.
[5] E. Schluntz and B. Zhang, "Building Effective Agents," Anthropic Engineering, December 2024.
[6] Anthropic, "Dynamic Workflows," Claude Code orchestration, May 2026.
[7] Anthropic, Knowledge Graph Construction Cookbook, claude-cookbooks repository.
[8] L. Martin, G. Cemaj, M. Cohen, "Scaling Managed Agents," Anthropic Engineering, April 2026.
[9] A. Karpathy, "Software 3.0, Agentic Engineering, and Jagged Intelligence," summary post, April 30, 2026.
[10] Fortune, "Why everyone is talking about Karpathy's autonomous AI research agent," March 17, 2026.
[11] Anthropic, "Building Effective AI Agents: Architecture Patterns," 2026.
[12] J. Sumner, Bun Zig-to-Rust port using Dynamic Workflows, May 2026.

Appendix

TABLE V
Terms Used in This Note

Term	Meaning
Autoresearch	Autonomous experimentation repo: agent edits train.py, evaluates, keeps or reverts
AgentHub	Agent-first collaboration: bare Git repo, SQLite, API, CLI, message board
DAG	Directed acyclic graph. Commits are nodes, parent links are edges
Agent swarm	Agents that explore, implement, or evaluate concurrently
Dynamic Workflows	Generated scripts that spawn and gather sub-agent tasks
Knowledge graph	Entities as nodes, typed relations as edges, with provenance
Structured outputs	Model responses constrained to a Pydantic schema
Provenance	Where a claim came from, which run produced it
program.md	Natural-language control specification for the autoresearch loop
Ratchet loop	Iterative process retaining only metric improvements
Graph grounding	Constraining generation with facts retrieved from a graph

Node types: `Entity`, `Claim`, `Source`, `Artifact`, `AgentRun`, `Evaluation`, `Task`, `Commit`, `Metric`.

Edge types: `MENTIONS`, `SUPPORTS`, `CONTRADICTS`, `DERIVED_FROM`, `PRODUCED`, `EVALUATES`, `REVISES`, `SUPERSEDES`, `DEPENDS_ON`, `PARENT_OF`, `RESOLVED_TO`.

Every graph write satisfies four invariants: (1) every claim has a source or is marked inference; (2) every artifact has an authoring run and version; (3) every evaluation identifies a rubric; (4) every superseded object remains addressable.

A practical default for a graph-grounded agent task:

1) Receive objective and constraints
2) Resolve task entities against the graph
3) Retrieve bounded subgraph with provenance
4) Create typed plan, validate dependencies
5) Assign independent steps to isolated workers
6) Require structured artifacts and evidence
7) Publish candidate graph updates
8) Validate schemas, permissions, provenance
9) Run deterministic tests
10) Run evaluator agents against rubrics
11) Resolve conflicts or escalate uncertainty
12) Publish versioned final artifact
13) Link to sources, graph paths, runs, evaluations
14) Record cost, latency, failures, open questions

A reliable system makes this true: *Every important output can be traced to an objective, a plan, an artifact, a source, a graph path, an evaluator decision, and a bounded execution record.*

TABLE VI
Production Checklist

Element	Ask Yourself	Failure If Missing
Objective	Is the task testable?	Agents optimize wrong thing
Metric	Distinguish improvement?	Activity without progress
Reversibility	Can updates be undone?	Failed experiment damages state
Tool schema	Arguments typed?	Invalid calls, silent errors
Artifact contract	What must workers return?	Inconsistent prose
Provenance	Every claim has source?	Outputs not auditable
Resolution policy	Decisions reversible?	False merges contaminate graph
Budget	Limits explicit?	Unbounded resources
Monitoring	Metrics tracked?	Regressions invisible
Recovery	Resume from state?	Every interruption restarts