Adaptive Deliberation Graphs: Generalizing Adaptive Computation Time from Recurrent Depth to Agentic Inference

Chaitanya Mishra
Independent Researcher

April 2026

Abstract

Adaptive Computation Time (ACT) asked a question that is more important in 2026 than when it appeared: how much computation should a model spend on this input? But its answer is now too narrow. ACTcan allocate extra recurrent updates along a single chain, while modern foundation model agents must distribute budget across heterogeneous computations: latent refinement, decomposition, retrieval, verification, tool use, and external action. This paper introduces ADG, a generalization of ACTfrom scalar halting on recurrent depth to budgeted control over typed computation graphs. ADGrepresents inference as growth of a deliberation graph whose nodes store latent state, optional readable traces, symbolic contract state, uncertainty, provenance, and cost. A value-of-computation controller expands the graph frontier by selecting the operator instance with the highest estimated marginal utility under a learned budget price, then halts when the upper bound on residual value falls below that price. The formulation subsumes ACTas a chain-restricted special case, replaces hand-tuned ponder penalties with dual budget control, and integrates contract-verified external actions so tool use becomes a first-class component of adaptive computation rather than an outer loop glued onto decoding. We give algorithms for graph construction, gain estimation, budget-conditioned control, and trace distillation from expensive teacher rollouts. Under adaptive submodularity assumptions we obtain greedy approximation guarantees; with bounded gain-estimation error we derive degradation bounds; with sound precondition and postcondition contracts we prove symbolic-state soundness over executed actions. The paper closes with a rigorous evaluation protocol for reasoning, coding, and web-agent tasks designed to falsify the method if its value estimates, graph abstraction, or contract interface fail. The core claim is simple: the missing successor to ACTis not deeper pondering, but a new primitive for allocating inference-time computation across a reusable, typed deliberation graph.

Empirical status. This paper contributes a formalism, algorithms, proofs, and a falsifiable evaluation program. It does not claim unrun large-scale experiments.

Contents

1 Introduction **5**

2 What ACTGot Right, and What It Could Not Yet Express **7**

2.1 The thesis of ACT 7

2.2 The elegance of the original abstraction . . . 8
2.3 The assumptions that no longer hold . . . 8
2.4 Why the correct successor is not a backbone swap . . . 9

3 Problem Formulation **10**
3.1 Budgeted agentic inference . . . 10
3.2 Deliberation graphs . . . 11
3.3 Operator instances and the frontier . . . 11
3.4 Value of computation . . . 12
3.5 Prices, not penalties . . . 12
3.6 Contracts and environment semantics . . . 13

4 Adaptive Deliberation Graphs **13**
4.1 Operator algebra . . . 13
4.2 Scoring frontier expansions . . . 14
4.3 Residual value and halting . . . 15
4.4 Graph reuse and dominance pruning . . . 15
4.5 Contract-verified external action . . . 16
4.6 Runtime algorithm . . . 17
4.7 Why graphs are the right successor to chains . . . 17

5 Learning to Control Deliberation **18**
5.1 Teacher rollouts and graphification . . . 18
5.2 Counterfactual gain labels . . . 18
5.3 Gain model and uncertainty model . . . 19
5.4 Budget-conditioned dual control . . . 19
5.5 From offline distillation to online adaptation . . . 19
5.6 Latent and readable traces . . . 20

6 Theoretical Analysis **20**
6.1 ACTas a chain-restricted special case . . . 20
6.2 Adaptive-submodular utility and greedy control . . . 22
6.3 Approximate greedy control with gain-estimation error . . . 23
6.4 Safe stopping by residual-value upper bounds . . . 23
6.5 Contract soundness . . . 24
6.6 Why these guarantees matter . . . 24

7 Systems Realization for Foundation Model Agents **24**
7.1 Runtime architecture . . . 25
7.2 Latent checkpoints and sparse surfacing . . . 25
7.3 Memory, hashing, and reuse . . . 25
7.4 Verifier cascades . . . 26
7.5 World models as optional VERIFYor REFINEoperators . . . 26
7.6 Deployment regimes . . . 26

8 Empirical Program and Evaluation Plan **26**
8.1 Task families . . . 27
8.2 Baselines . . . 27
8.3 Primary metrics . . . 29
8.4 Hypotheses and negative predictions . . . 29
8.5 Reporting standards . . . 30

9 Comparison to the Source Paper and to Modern Baselines **30**

10 Pressure Test: Where ADGBreaks **30**
10.1 Failure mode 1: Misestimated marginal value . . . 30
10.2 Failure mode 2: Frontier explosion . . . 30
10.3 Failure mode 3: Contracts that are too weak or too expensive . . . 32
10.4 Failure mode 4: Wrong graph canonicalization . . . 32
10.5 Failure mode 5: Tasks with little reusable structure . . . 32
10.6 Failure mode 6: Distribution shift in the value model . . . 32

11 Limitations **32**

12 Related Work **33**
12.1 Adaptive computation and conditional depth . . . 33
12.2 Test-time scaling and budgeted reasoning . . . 34
12.3 Metareasoning . . . 34
12.4 Search, reasoning, and action in language agents . . . 34
12.5 Verification-aware allocation . . . 34
12.6 World models, retrieval, and tool contracts . . . 34

13 Conclusion **34**

A Detailed Operator Semantics **35**
A.1 Refine . . . 35
A.2 Branch . . . 35
A.3 Retrieve . . . 35
A.4 Verify . . . 35
A.5 Merge . . . 36
A.6 Act . . . 36

B Proof Details **36**
B.1 Proof of proposition 1 . . . 36
B.2 Proof of theorem 1 . . . 36
B.3 Proof of theorem 2 . . . 36
B.4 Proof of proposition 2 . . . 37
B.5 Proof of proposition 3 . . . 37

C Implementation Recipe **37**

D Example Contracts **37**

E Notation Summary **38**

F Worked Examples **39**
F.1 A coding example with reusable certificates 39
F.2 A web-agent example with contracts 39
F.3 A mathematical reasoning example with selective verification 40

G Minimal Report Card for Future Empirical Papers **40**

1 Introduction

Inference-time computation has re-emerged as one of the dominant levers of model capability. A model can now be made stronger not only by increasing parameters or pretraining data, but by spending more computation after the prompt arrives: more latent refinement, more candidate trajectories, more retrieval, more verification, more simulation, more tool calls, or more deliberate action planning. Surveys published in 2025 captured the shift clearly: test-time scaling had become a broad research area rather than a niche efficiency trick [2, 3]. Yet the field still lacks a clean primitive for deciding *what* extra computation to buy, *where* to buy it, and *when* to stop.

Current practice is dominated by narrow control rules. Some methods allocate extra depth to selected tokens or latent states [5, 6, 7]. Some allocate more or fewer output tokens [8, 9]. Some allocate more search or verification to uncertain trajectories [11, 12]. Agent systems add yet another layer by searching over action plans, tool sequences, or environment interactions [13, 14, 15, 16]. These techniques are often useful, but they do not share a common computational object. They scale one dimension of inference at a time. They rarely treat retrieval, verification, latent reasoning, and external action as comparable budgeted operations inside one decision process.

A better starting point already existed. In 2016, Alex Graves at Google DeepMind proposed *Adaptive Computation Time* (ACT) [1]. The paper is still unusually alive in 2026 because it isolates the right question at the right level of abstraction. ACTdoes not begin with transformers, decoders, tree search, or tool use. It begins with a more primitive issue: input length and problem difficulty are different objects, so computation should not be tied rigidly to sequence position. That idea is even more relevant today than it was when ACTappeared. Modern agents face tasks whose difficulty varies by orders of magnitude, and they face it not only inside a recurrent cell but across retrieval systems, verifiers, code executors, browsers, and stateful environments.

Among older Google-lineage papers, ACTis therefore the right inheritance target. It is old enough that a genuine successor is now possible. It remains central because the underlying problem, adaptive allocation of compute under uncertainty, has become more important rather than less. It is elegant because the original mechanism is tiny, sharp, and easy to reason about. Most importantly, it is structurally unfinished. Graves's controller decides how many times to reapply one shared transition before moving on. That was the correct 2016 move, but it is not the correct 2026 abstraction.

The limiting assumptions are now clear. ACTallocates computation along a single chain. Each extra unit of compute is homogeneous: another application of the same recurrent update. Halting is local and scalar. The cost of thinking is compressed into one manually tuned ponder penalty τ. There is no branching, no reuse across hypotheses, no distinction between internal computation and external information acquisition, no verifier, no world model, no contract for tool execution, and no explicit treatment of actions whose effects change the environment. The paper itself identifies one of these weaknesses directly: the trade-off between accuracy and speed is "quite sensitive" to the time penalty parameter and should ideally be adapted automatically [1]. That open problem matters far more in 2026, when cost is multidimensional and includes not just FLOPs but latency, tool fees, rate limits, safety risk, and opportunity cost.

One could try to modernize ACTin several obvious ways. The first path is to keep the chain and swap the backbone, using latent recurrent depth or dynamic token routing inside a transformer [5, 6, 7]. The second is to keep the chain but control its textual length more carefully [8, 9, 10]. The third is to abandon the chain for a search tree, expanding and pruning textual thoughts or action plans [14, 15, 16]. Each is a real contribution. None is the missing successor. The first still treats

computation as repeated refinement of one local state. The second mostly budgets output length. The third handles branching but usually cannot reuse evidence, certificates, or partial world states across branches and often treats retrieval, verification, and action as ad hoc subroutines. What is missing underneath all three is a more general computational object.

This paper proposes that object. **Adaptive Deliberation Graphs** (ADG) generalize ACTfrom chain-valued recurrent depth to typed graph-valued inference. In ADG, the unit of adaptive computation is not "one more recurrent update" and not "one more token." It is a typed operator applied to a subgraph. Operators include latent refinement, decomposition, retrieval, verification, merge, and external action. The agent builds a deliberation graph whose nodes can store latent workspaces, readable summaries, symbolic contract state, uncertainty estimates, provenance, and accumulated cost. A controller scores frontier expansions by estimated marginal value of computation under a learned budget price. The same controller decides whether it is worth refining a latent hypothesis, branching on a hard subproblem, running a verifier, reusing shared evidence, or taking an externally visible action.

The graph, rather than a tree, is essential. Search trees duplicate work whenever two candidate plans depend on the same retrieved fact, formal certificate, or simulated state. Tool-using agents repeatedly encounter this pattern. A retrieved document may support several hypotheses. A unit test suite may certify multiple candidate programs. A browser session state may constrain many subsequent actions. A theorem checker or symbolic algebra engine may discharge a shared subgoal once and make it available to all descendant branches. ADGencodes these reusable computations directly, allowing the system to allocate budget over a DAG rather than over a chain or tree.

The second essential move is to replace ACT's fixed ponder penalty with a learned dual pricing mechanism. Modern agent inference is constrained not by a single scalar notion of "time" but by several resources at once. ADGtherefore optimizes expected utility under budget constraints using dual variables that price computation dimensions such as FLOPs, wall-clock latency, external tool cost, and safety risk. This preserves the spirit of ACTwhile removing one of its most important bottlenecks. Graves asked how to trade accuracy against speed. ADGanswers that the correct modern object is not a hand-tuned penalty but a budget-conditioned price over typed operations.

The third essential move is to treat external actions as contract-bearing computations. In many agent systems, tool use is bolted onto generation with little semantic discipline. ADGinstead makes external action a first-class operator whose applicability is guarded by preconditions, whose result must satisfy postconditions, and whose effect updates a symbolic state carried by the graph. This lets adaptive compute include interaction with the world without collapsing into unconstrained action search.

The contributions of the paper are fivefold.

1. It identifies Graves's ACTas the right older Google-lineage precursor for modern adaptive compute, and it isolates the precise assumptions that must be broken to build a true successor.

2. It introduces ADG, a new formalism for budgeted inference in which typed operator applications expand a reusable deliberation graph rather than a recurrent chain.

3. It derives a controller based on marginal value of computation with dual budget pricing, residual-value halting, dominance pruning, and contract-verified external actions.

4. It proves structural results: ACTas a chain-restricted special case of ADG, greedy approximation guarantees under adaptive submodularity, degradation bounds under gain-estimation error, and symbolic-state soundness under contract enforcement.

5. It specifies a concrete, falsifiable empirical program for reasoning, coding, and web-agent tasks, including baselines, metrics, ablations, and negative predictions.

The paper is intentionally not a benchmark patch. Its central claim is conceptual and systems-theoretic: adaptive computation for foundation model agents requires a different primitive. The question is no longer how long to ponder before emitting the next token. The question is how to allocate a finite budget across heterogeneous computations that can branch, reuse, verify, and act. ADGis an answer to that question.

2 What ACTGot Right, and What It Could Not Yet Express

2.1 The thesis of ACT

ACTbegins from a simple but deep observation: the amount of computation a model should use need not be fixed by input length. Graves operationalized this for recurrent networks by allowing the state transition function to be applied a variable number of times at each input position [1]. Let x_t denote the current input, s_t^n the intermediate state after n updates at step t, and y_t^n the corresponding intermediate output. ACTreuses the same transition function for all intermediate updates,

$$s_t^n = \begin{cases} S(s_{t-1}, x_t^1) & n = 1, \\ S(s_t^{n-1}, x_t^n) & n > 1, \end{cases} \tag{1}$$

where the input is augmented with a flag so the network can distinguish the first exposure to x_t from repeated pondering on the same input. A halting unit produces scores

$$h_t^n = \sigma(W_h s_t^n + b_h), \tag{2}$$

which are accumulated until they exceed $1 - \varepsilon$. The final halting distribution p_t^n uses a remainder term on the last step so that the intermediate outputs and states form a valid convex combination. The emitted state and output are then mean-field averages over the intermediate states and outputs.

Several aspects of the design are still admirable. First, the mechanism is minimal. The recurrent core is unchanged except for an extra halting unit and a repeated application of shared parameters. Second, the method is deterministic and differentiable, avoiding the high-variance gradient estimators that a sampled stopping rule would introduce. Third, the sharing of transition parameters separates two often conflated effects: adding more computation versus adding more parameters. The paper was careful not to claim progress merely by increasing model size.

The training objective adds a ponder cost

$$\widehat{L}(x, y) = L(x, y) + \tau P(x), \qquad P(x) = \sum_{t=1}^{T} \rho_t, \tag{3}$$

with $\rho_t = N(t) + R(t)$, encouraging the network to solve easy inputs quickly and difficult inputs slowly. The empirical section showed that variable compute could make otherwise difficult synthetic algorithmic tasks straightforward for recurrent nets, and the language modeling experiment suggested something more interesting than raw accuracy: computation time itself revealed structure in the data, concentrating on boundaries and informative transitions rather than on unpredictable noise [1]. In retrospect, that is one of the paper's most modern ideas. Compute allocation can expose

where the model believes useful structure lives.

2.2 The elegance of the original abstraction

Many historically important papers are hard to extend because their abstractions are already saturated. ACTis different. It is elegant in exactly the way that invites a successor. The abstraction can be stated in one sentence: *let the model decide how many computational updates an input deserves.* That sentence is still live in 2026. The reason the paper remains relevant is not the recurrent architecture, the synthetic tasks, or the particular halting equation. It is that the paper cleanly decomposes performance into two currencies, prediction quality and computation, and makes the exchange rate learnable.

This is why ACTis a better predecessor for modern adaptive inference than many more famous Google-lineage papers. A paper like *Attention Is All You Need* changed the dominant architecture, but its core abstraction has been expanded by countless descendants. ACT, by contrast, asked a more primitive question that modern systems still answer badly. The paper is therefore both deep and unfinished. It is deep because the question is right. It is unfinished because the mechanism is chained to the limitations of 2016 recurrent modeling.

There is another aspect of ACTthat deserves emphasis. The paper does not treat "thinking longer" as intrinsically good. More computation is justified only when it improves the output. The language modeling analysis explicitly distinguishes structure from irreducible unpredictability: the model learns not to spend extra computation on random ID numbers even though they are hard to predict, because extra effort will not help [1]. That attitude is exactly what 2026 agent systems need. The goal is not maximal internal activity. The goal is positive *marginal value of computation*.

2.3 The assumptions that no longer hold

The limitations of ACTare not accidental flaws. They are the assumptions required to make adaptive compute fit the technology of its time.

A single active locus of compute. At each input position, ACTassumes there is one state worth refining. This excludes decomposition into subproblems, parallel hypotheses, and deliberate exploration of alternatives.

Homogeneous compute steps. Every extra unit of computation is the same transition $S(\cdot, \cdot)$ applied again. Modern systems have heterogeneous compute primitives: search, retrieval, verification, sandboxed code execution, browser interaction, world-model simulation, and latent reflection. These are not interchangeable repeated steps.

Chain structure. Intermediate states form a linear sequence. But modern reasoning often has branching and sharing. Two candidate proofs may rely on the same lemma. Two tool plans may depend on the same retrieved schema. Two candidate programs may share the same test suite.

Scalar stopping. ACThalts by accumulating local halting mass. For agents, the relevant question is not only whether the current thought is done, but whether any further computation anywhere in the deliberation structure has positive net value under the remaining budget.

A fixed scalar ponder penalty. The original method uses one parameter τ to price computation. This is the assumption Graves himself singled out as fragile. In modern agents, the issue is worse. Computation is not one-dimensional. Latent FLOPs, wall-clock time, API cost, memory pressure, and risk of unsafe action all matter.

No explicit external action semantics. ACTnever has to distinguish internal computation from world-changing action. Agent systems do, and the distinction cannot be a mere implementation detail. A browser click or API mutation changes the environment and therefore the future value of computation.

No reusable evidence. Because the structure is a chain, the method cannot express that one retrieved document or one verifier certificate should be shared by several downstream hypotheses.

No failure-aware contracts. External tools in agent systems often fail because the wrong action is attempted, the output is malformed, or a hidden precondition was violated. ACThas no way to represent or enforce such constraints because they are outside its abstraction.

These are not reasons to reject the original paper. They are precisely the reasons it deserves a successor.

2.4 Why the correct successor is not a backbone swap

The easiest way to modernize ACTis to keep the core control rule and change the substrate. One can let a transformer allocate depth unevenly across tokens [5], run a recurrent latent block more times at test time [6], or iteratively refine attention weights to a fixed point [7]. This line is important, and ADGexplicitly reuses it through the REFINEoperator. But it is only a partial answer because it still assumes that the main thing worth buying is more internal refinement of one hypothesis.

A second partial successor treats the control problem as managing the length or segmentation of a textual reasoning trace. Recent test-time scaling methods optimize how many tokens to generate, where to allocate fixed token budgets, or when to stop generating more thoughts [8, 9, 10]. Again, useful, but still narrow. Tokens are at best one surface manifestation of computation, not the primitive itself.

A third partial successor moves from chains to trees by searching over thoughts, actions, or plans [14, 15, 16]. Trees are more expressive than chains, but they still do not fully solve the problem. Tree search usually duplicates shared evidence, lacks a unified cost model for retrieval versus verification versus action, and often treats world interaction as a poorly typed branch expansion.

The missing leverage point is not more scale, more prompts, more memory, more retrieval, or more agents. It is the computational object. Once we replace the chain with a typed graph and replace local halting with marginal-value scheduling under explicit budget prices, many modern techniques become special operators inside one framework rather than separate outer-loop hacks.

Table 1: From the original ACTabstraction to the requirements of modern agentic inference. The key gap is not model family but the unit and structure of adaptive computation.

Dimension	ACT(2016)	Modern requirement
Unit of extra compute	Reapply one shared recurrent update	Choose among heterogeneous operators: refine, branch, retrieve, verify, merge, act
Structure of inference	Linear chain at each input step	Reusable DAG with branching and shared evidence
Stopping signal	Local accumulated halting mass	Global residual value under remaining budget
Cost model	Scalar ponder penalty τ	Budget-priced multidimensional cost: FLOPs, latency, tool fees, risk
Interaction with environment	None	First-class external actions with contracts and rollback paths
Reuse across hypotheses	None	Shared documents, certificates, simulations, and state snapshots
Failure awareness	Implicit through prediction loss	Explicit legality checks, postcondition validation, and falsifiable halting calibration

3 Problem Formulation

3.1 Budgeted agentic inference

We consider a task instance consisting of an input x, an optional initial environment state e_0, and a budget specification B. The hidden task outcome, correct answer, or external environment dynamics are summarized by a latent variable ω. The agent may spend computation to produce internal latent states, readable traces, retrieved evidence, verifier certificates, or externally visible actions. It eventually terminates with a committed output a and, in interactive settings, a final environment state e_T.

The central object is a policy over computations rather than a policy over tokens. A computational policy π should maximize expected utility subject to budget and safety constraints:

$$\pi^* \in \arg\max_{\pi} \mathbb{E}\big[U(a, e_T, \omega)\big] \quad \text{such that} \quad \mathbb{E}[C_\pi] \preceq B, \qquad \Pr\left[V_\pi = 1\right] \leq \delta, \tag{4}$$

where $C_\pi \in \mathbb{R}_+^m$ is a vector of costs, $B \in \mathbb{R}_+^m$ is a vector budget, and V_π denotes a safety or contract-violation indicator. In the simplest deployment $m = 1$ and the cost is scalar. In realistic agent settings, the dimensions may include latent FLOPs, wall-clock latency, external API cost, and action risk.

The important modeling choice is that the policy does not directly decide "which token next." It decides which *computation* next. Output generation itself becomes one possible operator among others.

3.2 Deliberation graphs

Definition 1 (Deliberation graph)**.** At computation step k, the agent maintains a directed acyclic graph

$$G_k = (V_k, E_k, \zeta_k), \tag{5}$$

where V_k is a set of typed nodes, E_k a set of typed edges, and ζ_k global metadata such as remaining budget, tool handles, and environment snapshots.

Each node $v \in V_k$ carries a tuple

$$\psi(v) = (\tau_v, z_v, r_v, s_v, u_v, p_v, c_v), \tag{6}$$

with the following semantics.

- $\tau_v \in \mathcal{T}$ is a node type. Examples include `hypothesis`, `subgoal`, `evidence`, `certificate`, `action`, `state-snapshot`, and `commit`.
- z_v is a latent workspace state. It may be a hidden-state tensor, a pooled representation, or a structured latent object.
- r_v is an optional readable trace such as a textual rationale, a tool query, a concise summary, or a formal expression. It is optional by design. ADGdoes not require all internal computation to be surfaced as text.
- s_v is a symbolic state fragment. It stores typed facts used for contracts, obligations, variable bindings, and environment constraints.
- u_v stores uncertainty and value-related metadata, including confidence, calibration statistics, or estimated residual value.
- p_v stores provenance: parent pointers, source citations, verifier identities, timestamps, and hashes needed for reuse or audit.
- $c_v \in \mathbb{R}_+^m$ is cumulative cost metadata up to the node.

Edges are typed as well. We distinguish at least four edge kinds: `derives` for refinement and decomposition, `supports` for evidence attachment, `certifies` for verifier outputs, and `transitions` for environment-changing actions. The DAG restriction is not cosmetic. It guarantees acyclic provenance even when external actions create new state snapshots. Temporal feedback is represented by introducing a new snapshot node rather than by making the graph cyclic.

3.3 Operator instances and the frontier

Let $\mathcal{O}$ denote a typed operator library. Each operator $o \in \mathcal{O}$ has an input signature, output type constraints, a transition kernel, and a vector cost model. An operator instance is a pair

$$u = (o, \mathbf{v}), \tag{7}$$

where $\mathbf{v}$ is an ordered tuple of input nodes satisfying the operator's type signature. Applying u to graph G yields a random next graph

$$G' \sim T_o(G, \mathbf{v}). \tag{8}$$

The set of legal operator instances defines the current frontier,

$$\text{Frontier}(G) = \big\{(o, \mathbf{v}) : o \in \mathcal{O},\ \mathbf{v} \subseteq V,\ \text{legal}(o, \mathbf{v}, G) = 1\big\}. \tag{9}$$

Legality includes both typing constraints and contract checks. A VERIFYoperator may require a candidate hypothesis and an associated evidence set. An ACToperator may require that preconditions hold in the current symbolic state snapshot. A MERGEoperator may require sibling hypotheses defined over the same subgoal.

3.4 Value of computation

The right modern analogue of ACT's halting signal is not a scalar probability of local completion. It is the *marginal value of computation.* Let

$$U^*(G, b) = \sup_{\pi: C_\pi \preceq b} \mathbb{E}\big[U(\text{Commit}(G^\pi), \omega) \mid G\big] \tag{10}$$

be the optimal future utility achievable from current graph G under remaining budget b. For a legal operator instance u with cost $c(u)$, define the optimal marginal value

$$\Delta^*(u \mid G, b) = \mathbb{E}\big[U^*(G \oplus u, b - c(u)) - U^*(G, b) \mid G\big], \tag{11}$$

where $G \oplus u$ denotes the random graph after applying u.

Exact computation of (11) is generally intractable. ADGtherefore learns a gain model $\mu_\phi(u \mid G, b)$ and an uncertainty estimate $\sigma_\phi(u \mid G, b)$. These are used both for action selection and for conservative stopping.

3.5 Prices, not penalties

Instead of a single ponder penalty τ, ADGuses dual prices over budget dimensions. Let $\lambda \in \mathbb{R}^m_+$ be a nonnegative price vector and define the scalarized priced cost

$$\bar{c}_\lambda(u) = \lambda^\top c(u). \tag{12}$$

The constrained objective (4) admits the Lagrangian

$$\mathcal{L}(\pi, \lambda, \nu) = -\mathbb{E}\big[U_\pi\big] + \lambda^\top(\mathbb{E}[C_\pi] - B) + \nu(\mathbb{E}[V_\pi] - \delta), \tag{13}$$

where $\nu \geq 0$ prices safety violations. The difference from ACTis structural. The original ponder penalty is a scalar coefficient added to the loss. Here the prices are dual variables for explicit deployment constraints and can be conditioned on the user budget, the environment, or service-level objectives.

In practical deployments one need not expose the full vector to the controller. The controller can operate on the scalarized cost $\bar{c}_\lambda(u)$ while a separate budget module updates λ online or per episode.

3.6 Contracts and environment semantics

Every external action operator $a \in \mathcal{O}_{\text{ext}}$ is associated with a contract

$$\mathcal{C}_a = (\text{pre}_a, \text{schema}_a, \text{post}_a, \text{effect}_a, \text{rollback}_a). \tag{14}$$

Here pre_a is a predicate on the current symbolic state, schema_a describes the expected output structure, post_a validates semantic properties of the returned observation, effect_a updates the symbolic state if the action succeeds, and rollback_a specifies what happens when validation fails. The contract layer turns tool use from an informal string-emitting habit into a typed computational transition.

4 Adaptive Deliberation Graphs

4.1 Operator algebra

ADGuses a small operator basis. The basis is deliberately chosen to preserve the spirit of ACT: a small set of primitives should cover a large class of adaptive computations.

Table 2: Core ADGoperator types. The operator set is small by design; the goal is to make heterogeneous computation comparable under one value function rather than to define a new specialized planner for every task.

Operator	Typical inputs	Primary effect	Examples
Refine	Hypothesis or subgoal node	Improve latent state, local plan, or confidence without changing global branching structure	Latent recurrent depth, short hidden-thought block, local repair of a candidate solution
Branch	Hypothesis or unresolved subgoal	Create several child hypotheses, decompositions, or action candidates	Split a proof into lemmas, propose multiple API plans, enumerate candidate programs
Retrieve	Query-bearing node	Attach evidence nodes from corpora, memory, or schema stores	Document retrieval, API schema lookup, browser search, memory recall
Verify	Candidate subgraph and optional evidence	Produce a certificate, score, or refutation signal	Unit tests, theorem checking, consistency checks, world-model simulation
Merge	Sibling hypotheses or evidence set	Reuse or consolidate shared computation	Fuse evidence, collapse equivalent states, aggregate branch votes
Act	Action node plus state snapshot	Execute a world-changing external operation under contract checks	API call, code execution, browser click, database mutation
Commit	Candidate answer or plan	Materialize the current best output and expose it to downstream evaluation	Final answer emission, finalized tool plan, executable program

The operator basis is expressive enough to subsume many existing techniques. A latent recurrent reasoning model is a pure sequence of Refineoperators. A tree-of-thought search is mostly Branch, Refine, and Commit. A retrieval-augmented verifier is Retrievefollowed by Verify. Tool agents interleave Branch, Retrieve, Act, and Verify. The point is not that these methods become identical. The point is that their computational decisions become comparable because they are scored by the same marginal-value machinery.

4.2 Scoring frontier expansions

At graph G_k with remaining scalarized budget b_k, the controller scores every legal frontier expansion $u \in \mathrm{Frontier}(G_k)$ using an optimistic marginal-utility estimate:

$$S_\phi(u \mid G_k, b_k, \lambda_k) = \mu_\phi(u \mid G_k, b_k) + \beta_k \sigma_\phi(u \mid G_k, b_k) - \bar{c}_{\lambda_k}(u), \tag{15}$$

where $\beta_k \geq 0$ controls optimism or exploration during inference. The selected operator is

$$u_k = \arg\max_{u \in \text{Frontier}(G_k)} S_\phi(u \mid G_k, b_k, \lambda_k). \tag{16}$$

The controller then applies u_k, updates the graph, decrements budget, and repeats.

Two points matter here. First, the controller is not tied to a single architectural locus. A REFINEoperator may act on a latent model state, a RETRIEVEoperator may query an external retriever, and a VERIFYoperator may call a theorem prover or simulator. Second, costs are explicit. The same scoring rule can compare, for example, one extra hidden-state refinement against one expensive browser action. ACTcould not express that comparison because all extra computation was of one type.

4.3 Residual value and halting

ACThalts when local halting mass exceeds a threshold. ADGhalts when *further computation is not worth its price.* Let $\mathcal{R}^*(G, b)$ denote the optimal residual value,

$$\mathcal{R}^*(G, b) = U^*(G, b) - U(\text{Commit}(G), \omega). \tag{17}$$

ADGlearns a residual-value upper bound $\widehat{\mathcal{R}}_\phi(G, b)$ as well as local gain estimates. The controller halts when both conditions hold:

$$\max_{u \in \text{Frontier}(G_k)} S_\phi(u \mid G_k, b_k, \lambda_k) \leq \eta_k, \tag{18}$$

$$\widehat{\mathcal{R}}_\phi(G_k, b_k) \leq \eta_k^{\text{res}}, \tag{19}$$

for small nonnegative thresholds η_k and η_k^{res}. The first clause is local. The second guards against myopic stopping by estimating the remaining global upside in the entire graph.

This rule is the first place where ADGdeliberately breaks with the strict differentiability ethos of ACT. In the 2016 setting, a smooth halting unit was appropriate. In 2026 agent inference, the right stopping object is a calibrated estimate of residual value over a discrete, heterogeneous computation space. Differentiability is no longer the main objective. Correct budgeting is.

4.4 Graph reuse and dominance pruning

The graph representation matters not only for expressivity but for efficiency. If two frontier hypotheses share the same retrieved document or the same verifier certificate, a tree representation duplicates work. ADGinstead identifies equivalent or dominated substructures and reuses them.

We define a task-dependent dominance relation Dom over frontier nodes. Intuitively, node v dominates node w if both represent the same subgoal or state abstraction, but v has no worse confidence, stronger or equal support, and lower or equal accumulated cost. The exact definition varies by application. For program synthesis, two nodes may be considered equivalent if they induce the same executable semantics on the available tests. For web agents, two action-state nodes may be equivalent if they induce the same DOM abstraction and permissions. For retrieval-heavy QA, equivalence may be defined by the same subquestion plus a superset of evidence with no lower confidence.

Dominance pruning is then applied after each expansion:

$$G_{k+1} \leftarrow \text{Prune}(G_{k+1}; \text{Dom}). \tag{20}$$

This is one of the places where the graph abstraction earns its keep. In a tree, pruning dominated siblings still leaves duplicated subtrees. In a DAG, equivalent branches can merge and continue sharing downstream computation.

4.5 Contract-verified external action

ADGmakes external action explicit rather than incidental. Suppose an ACTinstance proposes action a in state snapshot node v_s. The action is only legal if the precondition holds:

$$\text{legal}(a, v_s, G) = 1 \iff s_{v_s} \models \text{pre}_a. \tag{21}$$

If legal, the runtime executes the action, obtains observation o_a, validates its schema and postconditions, and only then updates the symbolic state:

$$s' = \begin{cases} \text{effect}_a(s, o_a) & \text{if } o_a \models \text{schema}_a \wedge \text{post}_a(s, o_a), \\ \text{rollback}_a(s, o_a) & \text{otherwise.} \end{cases} \tag{22}$$

The resulting observation, certificate, and next state snapshot are attached as nodes in the graph. This yields three benefits.

1. **Safety.** Invalid tool outputs do not silently contaminate future reasoning.
2. **Value estimation.** The controller can learn that some tools are high-variance and only worth using when the expected upside exceeds the price.
3. **Auditability.** Every action and validation result becomes part of the graph's provenance.

4.6 Runtime algorithm

Algorithm 1 ADGinference

Require: input x, initial environment state e_0, budget B, operator library $\mathcal{O}$, base model f_θ, controller ϕ, budget prices λ_0

1: initialize root graph G_0 with input node, state snapshot node for e_0, and initial candidate commit node
2: $b_0 \leftarrow B$
3: **for** $k = 0, 1, 2, \ldots$ **do**
4: construct legal frontier $\text{Frontier}(G_k)$ using typing rules and contracts
5: **if** $\text{Frontier}(G_k) = \varnothing$ **then**
6: **break**
7: **end if**
8: **for all** $u \in \text{Frontier}(G_k)$ **do**
9: compute $\mu_\phi(u \mid G_k, b_k)$, $\sigma_\phi(u \mid G_k, b_k)$, and $\bar{c}_{\lambda_k}(u)$
10: compute score $S_\phi(u \mid G_k, b_k, \lambda_k)$ via (15)
11: **end for**
12: estimate residual upper bound $\widehat{\mathcal{R}}_\phi(G_k, b_k)$
13: **if** $\max_u S_\phi(u \mid G_k, b_k, \lambda_k) \le \eta_k$ **and** $\widehat{\mathcal{R}}_\phi(G_k, b_k) \le \eta_k^{\text{res}}$ **then**
14: **break**
15: **end if**
16: select $u_k \leftarrow \arg\max_{u \in \text{Frontier}(G_k)} S_\phi(u \mid G_k, b_k, \lambda_k)$
17: execute u_k to obtain next graph G_{k+1} and realized cost c_k
18: apply dominance pruning and graph merge rules to G_{k+1}
19: update budget $b_{k+1} \leftarrow b_k - c_k$
20: update price vector λ_{k+1} if using online dual control
21: **if** $b_{k+1} \preceq 0$ **then**
22: **break**
23: **end if**
24: **end for**
25: return best commit node $v^* \in \text{Commit}(G_k)$ according to calibrated answer utility

4.7 Why graphs are the right successor to chains

The graph formulation is not merely a larger search space. It changes the semantics of compute allocation.

In a chain, all future computation refines one evolving state. In a graph, computation can create, compare, certify, and reuse several partial objects at once. This matters most when the best use of the next unit of budget is neither “think longer” nor “sample again,” but something structurally different: retrieve a missing schema, verify a shared lemma, execute one reversible action to disambiguate the world state, or merge two branches that turned out to be equivalent. These are the cases where modern systems either over-generate tokens or resort to brittle orchestration logic.

5 Learning to Control Deliberation

A new computational primitive is only useful if it can be learned. ADGrequires two kinds of learning: learning a base model interface that exposes useful latent and symbolic summaries, and learning a controller that predicts the marginal value of operator applications under budget constraints. The paper focuses on the second problem while remaining agnostic about the exact base model.

5.1 Teacher rollouts and graphification

The most direct source of supervision is an expensive teacher policy that overspends compute offline. For closed-form reasoning, the teacher may use broad search, self-consistency, verifier cascades, or expensive latent-depth schedules. For program synthesis, it may enumerate candidate programs and test them extensively. For tool-use and web tasks, it may run a powerful planner with larger search width, stronger verifiers, or more environment interaction than the final deployment budget allows.

These teacher traces are then converted into graphs by a procedure we call GRAPHIFY. The procedure aligns segments of the trace with operator types and merges repeated or shared structures. A textual decomposition step becomes a BRANCHnode if it creates multiple candidate subtasks, but it becomes a REFINEnode if it simply strengthens one existing hypothesis. A repeated retrieval query becomes a shared evidence node rather than duplicated branch-local text. Verifier outputs are extracted as certificate nodes with explicit provenance. Action outcomes are converted into state-snapshot transitions, carrying both raw observations and validated symbolic effects.

This step is more than data cleaning. It is the moment where the implicit compute structure of an expensive reasoning trace becomes explicit and reusable. Many current methods store teacher traces as flat token sequences. ADGinstead treats them as partial computation graphs. That choice is necessary if the controller is to learn reuse and nonlocal stopping rather than just imitation of long outputs.

5.2 Counterfactual gain labels

The controller needs targets for marginal value. Simple imitation of teacher actions is insufficient because expensive teachers often expand computations that are only worthwhile under their own larger budget. We therefore attach a counterfactual gain label to each operator instance observed in a teacher graph.

Let u_t be a teacher-selected operator at partial graph G_t. A useful target is the difference between the realized teacher utility and the utility achievable if u_t were removed and the remaining budget spent by a repair policy:

$$y_t = U\big(\textsc{TeacherSearch}(G_t \oplus u_t, b_t - c(u_t))\big) - U\big(\textsc{ReplayRepair}(G_t, b_t)\big). \tag{23}$$

The repair policy need not be exact. In practice it can be a bounded local search, a model-based simulator, or a logged-behavior replay with doubly robust correction. The purpose is to estimate the incremental utility attributable to taking u_t at that moment rather than to imitate the teacher's full future trace blindly.

This counterfactual view changes the learning problem. The controller is not rewarded for matching a verbose trajectory. It is rewarded for identifying which computations mattered under a specified budget.

5.3 Gain model and uncertainty model

The controller learns two functions.

1. A gain predictor $\mu_\phi(u \mid G, b)$ estimating expected marginal utility.
2. An uncertainty predictor $\sigma_\phi(u \mid G, b)$ estimating epistemic or predictive uncertainty in that gain estimate.

A practical parameterization uses a graph encoder over G, an operator-specific scoring head, and a small budget module. The graph encoder must summarize latent node states, symbolic obligations, recent tool outcomes, and provenance features such as evidence overlap. The operator head then reads both the graph embedding and the typed local inputs of u.

The uncertainty estimate is not optional. Halting decisions are much more brittle than action ranking. If the controller is overconfident about low residual value, it stops too early; if it is underconfident, it overthinks. Uncertainty estimates can be obtained by lightweight ensembles, dropout-based approximations, or quantile regression over gain targets. The method is modular about how these are implemented. What matters is that the stopping rule has calibrated upper bounds, not point estimates alone.

5.4 Budget-conditioned dual control

Training with a fixed ponder-style penalty would reproduce one of ACT's key weaknesses. ADGinstead optimizes under explicit budgets. Let $\pi_\theta(\cdot \mid B)$ denote a budget-conditioned controller and let (λ, ν) be dual variables for cost and violation constraints. We solve the saddle problem

$$\min_{\theta} \max_{\lambda \succeq 0,\, \nu \geq 0} \Big[-\mathbb{E}[U_{\pi_\theta}] + \lambda^\top (\mathbb{E}[C_{\pi_\theta}] - B) + \nu(\mathbb{E}[V_{\pi_\theta}] - \delta) \Big]. \tag{24}$$

A simple online dual update after each training episode is

$$\lambda \leftarrow \Big[\lambda + \eta_\lambda (C_{\pi_\theta} - B) \Big]_+, \tag{25}$$

$$\nu \leftarrow \Big[\nu + \eta_\nu (V_{\pi_\theta} - \delta) \Big]_+. \tag{26}$$

The budget price becomes part of the state seen by the controller. This lets one trained system interpolate across budgets instead of relearning a separate τ for every deployment regime.

5.5 From offline distillation to online adaptation

The learning pipeline naturally has three stages.

1. **Offline graph distillation.** Build teacher graphs, derive counterfactual gain labels, and train the gain model plus residual-value head.
2. **Budget-conditioned imitation and ranking.** Train the controller to reproduce high-value operator orderings under sampled budgets.
3. **Online fine-tuning.** Interact with the true task environment, update the controller using utility and constraint feedback, and recalibrate uncertainty and residual estimates.

For noninteractive tasks, stage 3 can be lightweight because offline search traces already cover much of the distribution. For web or tool agents, stage 3 is more important because environment

dynamics and tool failures introduce deployment-specific distribution shift.

Algorithm 2 Training ADGcontrollers from expensive teacher rollouts

Require: training tasks $\mathcal{D}$, teacher search policy TeacherSearch, repair estimator ReplayRepair, operator library $\mathcal{O}$

1: **for all** $(x, e_0, \omega) \in \mathcal{D}$ **do**
2: run teacher policy under relaxed budget to obtain expensive trace τ^*
3: graphify trace: $G^* \leftarrow \textsc{Graphify}(\tau^*, \mathcal{O})$
4: **for all** operator instances u_t in G^* **do**
5: estimate counterfactual gain y_t via (23)
6: record training tuple $(G_t, u_t, b_t, y_t, U_t^{\text{res}}, C_t, V_t)$
7: **end for**
8: **end for**
9: train graph encoder and gain model (μ_ϕ, σ_ϕ) on gain targets
10: train residual-value head on U_t^{res}
11: train budget-conditioned controller to rank operator instances by predicted net value
12: fine-tune controller with online budget-constrained reinforcement learning using (24)
13: calibrate stopping thresholds η and η^{res} on held-out tasks

5.6 Latent and readable traces

A modern successor to ACTshould not force all adaptive computation into visible text. Some useful computation is latent. Some must be explicit for audit or tool invocation. ADGtreats readable traces as optional node attributes rather than as the only representation of thought. This makes the framework compatible with both latent recurrent reasoning approaches [6] and action-oriented textual agents [13].

In practice, one can expose only a sparse subset of internal nodes to text: nodes that trigger retrieval, verifier calls, or externally inspectable commitments. The rest can remain latent. This design avoids conflating interpretability policy with compute allocation policy.

6 Theoretical Analysis

This section provides structural guarantees. The purpose is not to claim that real agent tasks satisfy every assumption exactly. The point is to show that ADGhas a clean theoretical core rather than being a pile of heuristics.

6.1 ACTas a chain-restricted special case

Proposition 1 (ACTis a special case of ADG)**.** *Consider an ADGinstance with the following restrictions:*

1. *the graph at each input step is constrained to a single active path $v_t^1 \to v_t^2 \to \cdots$,*
2. *the operator set is $\{\textsc{Refine}, \textsc{Halt}\}$,*
3. *each* Refine*operator applies the same shared state transition $S(\cdot, \cdot)$ to the same input x_t with a first-step flag,*
4. *the controller emits halting masses $h_t^n \in (0, 1)$ accumulated until they exceed $1 - \varepsilon$,*

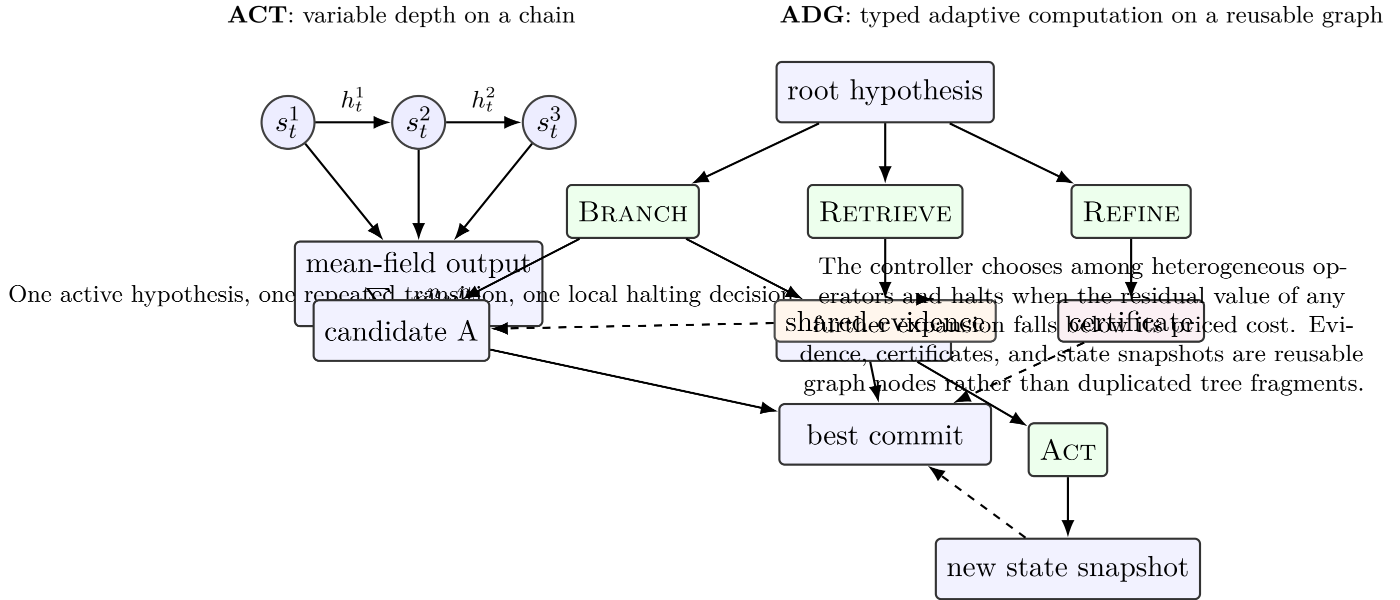

Figure 1: The structural leap from ACTto ADG. ACTadapts computation depth on a chain. ADGadapts computation type, location, and reuse pattern on a graph.

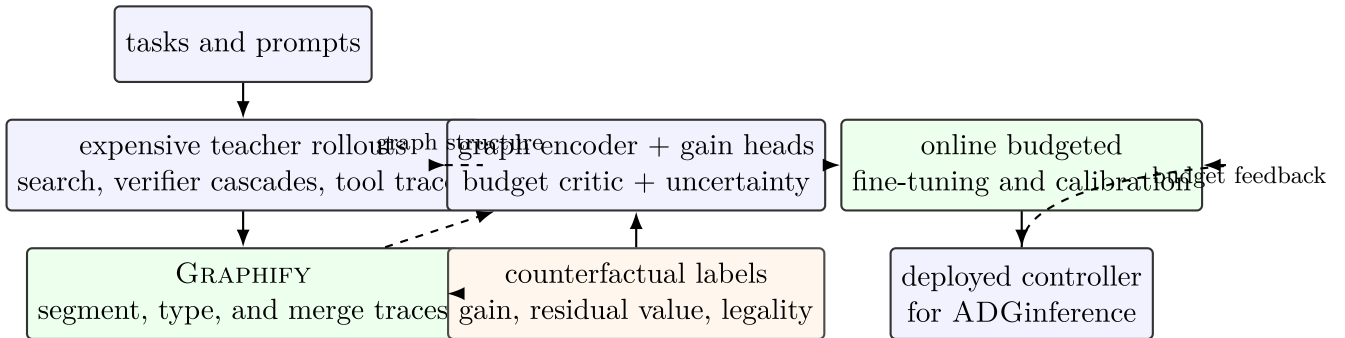

Figure 2: Training pipeline for ADG. Expensive teacher rollouts are converted into typed computation graphs, labeled with counterfactual gain and residual-value targets, and distilled into a budget-conditioned controller that is later calibrated online.

5. *the final emitted state and output are the convex combination of intermediate states and outputs weighted by the resulting halting distribution.*

Then the resulting ADGexecution is exactly Graves's ACTrecurrence with ponder cost $P(x) = \sum_t \rho_t$.

Proof sketch. Map each intermediate ACTstate s_t^n to a graph node v_t^n. Because the graph is restricted to a single path and the only nonterminal operator is REFINEwith shared parameters, each expansion reproduces one recurrent update of ACT. The HALTcontroller produces the same halting masses, the same stopping index $N(t)$, and the same remainder term $R(t)$ once the cumulative mass crosses $1 - \varepsilon$. Since the output node is constrained to be the convex combination of intermediate outputs, the emitted state and prediction match the mean-field output rule of ACT. The cumulative priced cost reduces to the ponder cost under a scalar budget price. □

Interpretation. ADGis therefore not a rejection of ACTbut a strict generalization. The original paper's abstraction survives. What changes is the computational object from chain to graph, the operator set from homogeneous to typed, and the cost model from a fixed scalar penalty to explicit budget prices.

6.2 Adaptive-submodular utility and greedy control

To obtain approximation guarantees, we use a metareasoning view. Each legal operator instance reveals a random observation and produces a random utility increment. Let a partial realization ψ encode the observed outcomes of previously executed operators. Suppose the utility of the final committed graph can be written as a set function over executed operator instances and their observations.

Assumption 1 (Adaptive monotonicity)**.** For every partial realization ψ and legal operator u, the conditional expected marginal utility is nonnegative:

$$\Delta(u \mid \psi) \geq 0. \tag{27}$$

Assumption 2 (Adaptive submodularity)**.** For any two partial realizations $\psi \subseteq \psi'$ and any legal operator u feasible under both, the conditional expected marginal utility decreases with more information:

$$\Delta(u \mid \psi) \geq \Delta(u \mid \psi'). \tag{28}$$

These assumptions are idealizations, but they are not absurd. Verification, retrieval, and local refinement often show diminishing returns once the graph already contains strong evidence or certificates.

Theorem 1 (Greedy approximation under adaptive submodularity)**.** *Suppose the utility induced by legal operator instances is adaptively monotone and adaptively submodular, and suppose all operator costs are unit costs or are converted to unit costs by standard cost splitting. Then the greedy ADGpolicy that repeatedly chooses the frontier operator with largest conditional expected marginal utility achieves at least a* $(1 - 1/e)$ *fraction of the optimal policy's expected utility under any fixed budget* B.

Proof sketch. Under the stated assumptions, the operator-selection problem reduces to adaptive submodular maximization under a cardinality constraint. The result then follows from the standard

adaptive greedy guarantee of Golovin and Krause [22]. The graph structure does not break the reduction because a partial graph together with realized observations is simply a partial realization. Shared nodes matter computationally, but the utility guarantee depends only on the executed operator set and observations. □

Cost-sensitive extension. When costs are nonuniform, the same argument applies to a greedy rule that chooses the operator with largest expected marginal utility per unit priced cost, subject to the usual truncation caveats from submodular knapsack settings. In practice ADGuses the net-value form in (15); the ratio form is a useful theoretical surrogate.

6.3 Approximate greedy control with gain-estimation error

Real controllers do not have access to exact marginal gains. Suppose the controller's gain predictor satisfies

$$\left|\mu_\phi(u \mid G, b) - \Delta^*(u \mid G, b)\right| \leq \varepsilon_g \tag{29}$$

for all legal (u, G, b) encountered under a budget of at most B, and suppose the chosen operator at each step is within additive ξ of the maximizer of the predicted net value. Then the controller behaves like an approximate greedy policy.

Theorem 2 (Degradation under gain-estimation error)**.** *Under the assumptions of theorem 1, the expected utility of an approximate greedy ADGpolicy $\widehat{\pi}$ satisfies*

$$\mathbb{E}[U(\widehat{\pi})] \geq (1 - 1/e)\,\mathbb{E}[U(\pi^*)] - B(2\varepsilon_g + \xi). \tag{30}$$

Proof sketch. At each decision step, the best true marginal gain can exceed the chosen true marginal gain by at most $2\varepsilon_g + \xi$: one ε_g for optimistic misranking of the chosen action, one ε_g for pessimistic misestimation of the true best action, and ξ for approximate maximization over the predicted score. Standard approximate-greedy analysis for submodular maximization then yields an additive degradation linear in the budget. □

The theorem is intentionally blunt. If the gain model is poor, the controller can lose linearly with budget. That is not a weakness of the proof. It is the right warning. Adaptive computation is only as good as its estimate of marginal value.

6.4 Safe stopping by residual-value upper bounds

Stopping is harder than action selection because it makes an irrevocable claim about all unexecuted computations. Let $\overline{\Delta}(u \mid G, b)$ be a high-probability upper bound on the true marginal value of operator u, and suppose the utility is adaptively submodular.

Proposition 2 (Halting gap bound)**.** *Assume that with probability at least $1-\alpha$, every legal operator instance satisfies*

$$\Delta^*(u \mid G, b) \leq \overline{\Delta}(u \mid G, b). \tag{31}$$

Let $M(b)$ be an upper bound on the number of further unit-cost operator executions allowed by remaining budget b. If the controller halts whenever

$$\max_{u \in \mathrm{Frontier}(G)} \overline{\Delta}(u \mid G, b) \leq \eta, \tag{32}$$

then with probability at least $1 - \alpha$ the residual optimality gap is at most $M(b)\eta$.

Proof sketch. Adaptive submodularity implies that no future sequence of at most $M(b)$ operator executions can gain more than the sum of the best one-step gains encountered along that sequence, each of which is upper-bounded by η. Hence the total additional value is at most $M(b)\eta$. □

This proposition makes the role of calibrated uncertainty explicit. The controller does not need perfect knowledge of future upside. It needs upper bounds honest enough that stopping at small estimated residual value is meaningful.

6.5 Contract soundness

Assumption 3 (Sound contracts)**.** For every external action a, if a symbolic state s is sound with respect to the environment and the precondition $\text{pre}_a(s)$ holds, then any observation o_a satisfying both schema_a and $\text{post}_a(s, o_a)$ yields an updated symbolic state $\text{effect}_a(s, o_a)$ that is also sound.

Proposition 3 (Symbolic-state soundness under contract enforcement)**.** *Assume the initial symbolic state is sound and all external actions in ADGare admitted only when their preconditions hold and only update the symbolic state after schema and postcondition validation. Then every symbolic state reachable in the deliberation graph is sound.*

Proof. By induction on the number of external action nodes. The base case is immediate from sound initialization. For the inductive step, all non-action operators leave symbolic soundness unchanged by construction. For an external action, legality ensures the precondition holds in a sound state. The state is updated only if the resulting observation passes schema and postcondition validation, and sound contracts preserve soundness under exactly those conditions. Hence the next symbolic state is sound. □

The proposition is modest but useful. It says that ADGcan reason over a symbolic abstraction of the environment without that abstraction silently drifting away from validated action effects.

6.6 Why these guarantees matter

The theoretical results do not claim that real reasoning tasks are perfectly adaptive-submodular or that all tool contracts are complete. They serve a more practical purpose. They show that ADGhas a small number of load-bearing components, each with a clean failure mode.

- If adaptive diminishing returns approximately hold, greedy frontier expansion is principled rather than ad hoc.
- If gain estimation is noisy, the degradation is explicit and linear in the error.
- If residual upper bounds are miscalibrated, stopping fails in a measurable way.
- If contracts are weak, symbolic-state soundness breaks exactly where validation failed.

This is preferable to a system whose success depends on several opaque heuristics interacting in unknown ways.

7 Systems Realization for Foundation Model Agents

The previous sections define ADGabstractly. This section describes how the abstraction maps onto modern foundation model systems.

7.1 Runtime architecture

A practical ADGruntime has five components.

1. A **base foundation model** f_θ that can produce latent summaries, readable outputs, and optionally action proposals.
2. A **graph controller** that encodes the current deliberation graph and scores frontier operators.
3. A **memory and provenance store** that indexes node states, evidence hashes, certificates, and state snapshots for reuse.
4. A **verifier and contract layer** that validates candidate answers, tool outputs, and external action preconditions and postconditions.
5. A **tool and environment interface** that executes retrieval, code, browser, database, or simulator operations.

This decomposition preserves modularity. The controller does not need to be the same model as the base generator. In resource-constrained deployments it can be a lightweight graph network reading compressed node summaries. In higher-end deployments it can share a trunk with the base model. The paper does not require a single architecture.

7.2 Latent checkpoints and sparse surfacing

The most immediate systems challenge is graph size. If every internal hidden state becomes a graph node, the runtime becomes impossible. The correct solution is sparse surfacing. REFINEcan internally run several latent micro-steps before writing a single summarized node back into the graph. Only decision-relevant checkpoints become durable nodes: branch points, retrieved evidence, verifier outcomes, contract-relevant symbolic states, and candidate commits.

This leads to a useful separation.

- **Fast latent loops** happen inside a REFINEoperator.
- **Structural decisions** happen at graph boundaries when the controller chooses the next operator type.

This is precisely how ADGavoids turning into either a giant textual trace or a dense hidden-state log.

7.3 Memory, hashing, and reuse

Graph reuse requires canonicalization. The runtime therefore maintains a hash or canonical key for each durable node. The key depends on task type.

- For retrieved evidence, the key may be a content hash plus normalized query role.
- For program candidates, the key may be an execution signature on a held-out lightweight test set.
- For action-state snapshots, the key may be an abstract DOM representation or normalized API state.
- For proof obligations, the key may be a normalized symbolic expression or theorem statement.

If a new node collides with an existing key and does not improve dominance statistics, the runtime reuses the existing node. If it dominates the old node, the graph can redirect dependents or mark the old node obsolete.

This is a major difference from most tree-search agents, where repeated rediscovery of the same evidence or state is treated as unavoidable overhead.

7.4 Verifier cascades

Not all verification is worth its full cost. ADGtherefore supports verifier cascades. A candidate can first pass through a cheap heuristic checker, then a stronger symbolic or execution-based verifier only if the expected gain justifies it. This makes verification itself an adaptive computation rather than a fixed post-processing stage.

For example, code synthesis can use a cascade of static linting, lightweight sampled tests, and full hidden tests. Mathematical reasoning can use format checks, local algebraic consistency, then expensive formal verification where applicable. Web agents can use DOM-schema validation before costly environment rollouts. Each verifier call is simply another VERIFYoperator with its own price and expected marginal value.

7.5 World models as optional Verifyor Refineoperators

Recent work on world-model-based agents shows that internal simulation can improve tool use in stateful environments [17, 18]. ADGdoes not force world modeling into a separate algorithmic family. A world-model simulation step is either a VERIFYoperator, if it evaluates a proposed action sequence, or a REFINEoperator, if it updates a latent state estimate. This is another advantage of typing computations rather than hard-coding planners.

7.6 Deployment regimes

ADGsupports several deployment regimes.

1. **Reasoning-only.** No external action, mostly REFINE, BRANCH, VERIFY, and COMMIT.
2. **Retriever-verifier.** RETRIEVEand VERIFYdominate, with occasional BRANCH.
3. **Tool agent.** Significant ACTand state-snapshot transitions, often with contract checks and rollback.
4. **Multimodal agent.** Visual or audio operators are added as typed retrieval or refinement primitives.

The same controller family can handle these regimes by swapping the operator library and the contract set, not by rewriting the control problem.

8 Empirical Program and Evaluation Plan

A strong idea should state clearly how it can fail. Because this paper does not fabricate large-scale experiments, the empirical section is written as an executable program: what to measure, what to compare against, and what outcomes would falsify the design.

8.1 Task families

The evaluation suite should cover three qualitatively different settings.

Closed-form reasoning. These are tasks where computation is entirely internal until a final answer is committed. Suitable examples include GSM8K, MATH-500, olympiad-style subsets, and science QA such as GPQA. The purpose is to test whether ADGcan allocate budget between latent refinement, branching, and verification better than fixed-length or token-budget baselines.

Code generation and repair. Benchmarks such as HumanEval, MBPP, and newer execution-based coding suites are ideal because verification is natural and strong. ADGshould decide when to generate more candidates, when to locally repair one candidate, and when to spend budget on tests versus further synthesis.

Stateful tool and web agents. WebArena, ToolBench, and ALFWorld capture situations in which the agent must interleave planning, retrieval, and external actions [19, 20, 21]. Here the point is not just answer quality but budgeted interaction with an environment. This is where contracts and state snapshots become indispensable.

8.2 Baselines

The baseline set should reflect the partial successor directions discussed earlier.

Chain-based adaptive compute. ACT[1], PonderNet [4], Mixture-of-Depths [5], latent recurrent reasoning [6], and adaptive textual stopping methods such as ACTS [8], MRT [9], and Plan-and-Budget [10].

Search and agent baselines. ReAct [13], Tree of Thoughts [14], LATS [15], and ToolTree [16]. These test whether the graph formulation buys anything beyond stronger tree search.

Verification-aware baselines. Uniform verification, confidence-threshold early exit, and anytime verified schemes such as AVA [12]. These test whether ADGtruly improves compute allocation rather than simply adding more verification.

Ablation baselines. The most important baselines are internal ablations.

1. **Chain ablation:** disable BranchHand Merge, reducing ADGto adaptive refinement plus stopping.
2. **Tree ablation:** disable reuse and graph merge, forcing a tree.
3. **No-contract ablation:** let external actions execute without precondition and postcondition checks.
4. **No-price ablation:** replace learned dual pricing with a fixed ponder-style scalar penalty.
5. **No-residual ablation:** use local stopping only.

These ablations reveal whether the paper's new primitives are load-bearing or ornamental.

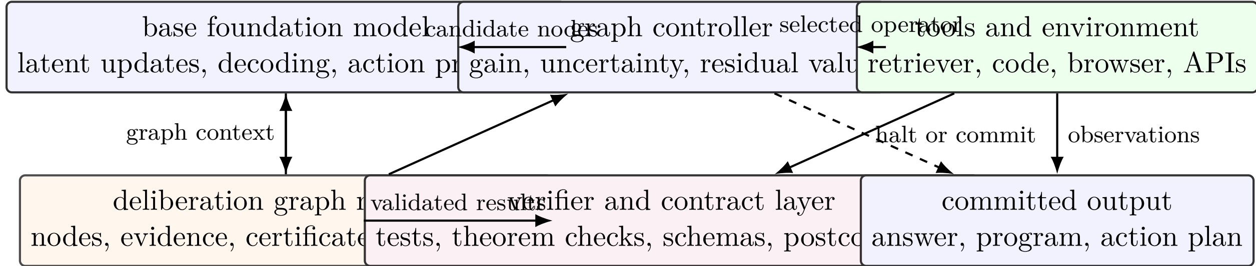

Figure 3: A reference ADGruntime. The controller sees a typed graph, not just the current token prefix. It can choose latent refinement, retrieval, verification, or external action under a unified value and budget model.

Table 3: Recommended benchmark matrix for evaluating ADG. The last column states what must be true for the method to be credible on that family.

Family	**Representative benchmarks**	**Key budget axes**	**Credibility condition**
Closed-form reasoning	GSM8K, MATH-500, GPQA-style subsets	Tokens, latent steps, verifier calls	Better utility–budget frontier than fixed CoT, self-consistency, and adaptive-length baselines
Code generation	HumanEval, MBPP, execution-based repair suites	Candidate count, test budget, repair depth	Higher pass@k at equal total execution budget or same pass@k at lower total budget
Web and tool agents	WebArena, ToolBench, ALFWorld	Environment actions, API calls, latency, rollback cost	Fewer invalid actions and better success under equal action and latency budgets
Retrieval-heavy QA	HotpotQA-style multi-hop tasks, schema-guided QA	Retrieval calls, verifier depth, merge cost	Demonstrable benefit from graph reuse over tree search or repeated retrieval

8.3 Primary metrics

Anytime utility. The central metric is performance as a function of budget. Report full utility–budget curves, not a single point. Area under the utility–budget curve should be the headline statistic.

Cost at target quality. For each task family, report the minimal budget needed to reach a fixed target utility or success rate. This reveals whether the controller is learning to spend compute in the right places.

Residual stopping error. At the moment of halting, estimate the best available continuation value using a stronger oracle or a deeper search. Compare it to the controller's predicted residual value. Large positive gaps indicate premature stopping.

Graph reuse ratio. Measure how much computation is saved by shared nodes relative to a tree execution with identical frontier decisions. If reuse remains negligible, then the DAG formulation is not earning its complexity.

Invalid action rate and rollback rate. For agent tasks, measure how often the controller attempts illegal actions, how often contracts reject tool outputs, and how often rollback is triggered.

Calibration. Evaluate calibration not only of final-answer confidence but of marginal-value predictions and residual-value bounds. A controller with sharp but miscalibrated gains is dangerous.

8.4 Hypotheses and negative predictions

A credible empirical program should state not only what ought to improve, but where the method should fail.

H1: Graph reuse is valuable only when substructure is real. On tasks with shared evidence or shared verifiers, ADGshould outperform tree methods at equal budget. On tasks with almost no reusable substructure, the graph advantage should collapse. If it does not, the gain model may be gaming the budget metric.

H2: Contracts matter most in stateful, failure-prone environments. In benchmark families with reliable external tools, contract overhead may offer only modest gains. In noisy web or API settings, the no-contract ablation should suffer more invalid states and more wasted compute.

H3: Residual halting matters most when verification is expensive. If verifiers are cheap and decisive, local stopping may be enough. If verifiers are expensive or ambiguous, residual-value stopping should reduce overthinking and underthinking.

H4: ADGshould lose on some open-ended generation tasks. In domains with weak verifiers, diffuse reward, and low subproblem reuse, simple long generation or broad sampling may remain superior. This is a feature of the evaluation plan, not a weakness. The method is designed for structured adaptive compute, not all language generation.

8.5 Reporting standards

Any serious evaluation of ADGshould report the following.

1. Full frontier traces for a representative sample of tasks.
2. Cost breakdown by operator type.
3. Calibration plots for gain and residual-value estimates.
4. Contract violation logs with root-cause categories.
5. Reuse statistics showing how often graph merge actually occurs.
6. Pareto frontiers against at least three strong baselines per task family.

Without these analyses, it would be impossible to tell whether success comes from better control, hidden extra compute, or dataset-specific heuristics.

9 Comparison to the Source Paper and to Modern Baselines

Tables 4 and 5 clarify the intended contribution. ADGis not "better tree search" and not "better stopping." It is a redefinition of the unit of adaptive compute. Existing methods often excel on one axis. ADGis aimed at the missing common substrate.

10 Pressure Test: Where ADGBreaks

The point of a successor paper is not to protect the idea. It is to isolate where the idea earns its complexity and where it does not.

10.1 Failure mode 1: Misestimated marginal value

ADGstands or falls on the quality of its gain estimates. If μ_ϕ cannot reliably predict whether one more verification call or one more refinement step helps, the controller will waste budget in exactly the way it was designed to avoid. This failure is especially likely in open-ended tasks with sparse reward and weak local signals. The approximate-greedy bound in theorem 2 makes the risk explicit: utility degrades linearly with gain-estimation error.

Decisive test. Compare predicted marginal values against oracle continuation values on held-out tasks. If rank correlation is weak and calibration poor, the paper's central mechanism fails.

10.2 Failure mode 2: Frontier explosion

A graph can be more expressive than a tree precisely because it permits branching and reuse. That same expressivity can cause uncontrolled frontier growth. If BRANCHfires too often or if equivalence and dominance relations are weak, the runtime becomes a large best-first search with extra bookkeeping.

Decisive test. Measure frontier size and effective branching factor as a function of budget. If the frontier grows superlinearly without commensurate utility gains, the graph abstraction is overfitting the idea of flexibility.

Table 4: Comparison I: chain-based adaptive-compute baselines. The comparison is about control primitives, not claimed benchmark superiority.

Method	Adaptive unit	Structure	Primary limitation addressed by ADG
ACT[1]	Recurrent updates	Chain	Homogeneous compute, scalar cost, no branching or reuse
PonderNet [4]	Recurrent pondering steps	Chain	Better halting learning, but still chain-restricted
Mixture-of-Depths [5]	Token-layer depth	Layered DAG over tokens	Adapts internal depth, not heterogeneous agent compute
Latent recurrent reasoning [6]	Latent refinement depth	Chain	Stronger latent compute, but no typed branching or contracts
ACTS [8]	Output length	Token stream	Stops textual generation, not general deliberation
Plan-and-Budget [10]	Token budgets across subquestions	Planned chain or tree	Useful budget allocation, but no reusable graph primitive
MRT [9]	Token-stream progress	Chain	Optimizes progress in textual traces, not general operator value

Table 5: Comparison II: search, verification, and agentic baselines. ADGis meant to supply a missing common substrate rather than to erase the value of specialized search methods.

Method	Adaptive unit	Structure	Tool and verifier support	Primary limitation addressed by ADG
ReAct [13]	Prompted reasoning and action steps	Chain	Yes	Interleaves thought and action, but lacks unified value pricing
Tree of Thoughts [14]	Thought expansion	Tree	Indirect	Branches well, but duplicates shared evidence and certificates
LATS [15] and ToolTree [16]	Search-tree expansions	Tree	Yes	Stronger planning, but not a general typed DAG with reusable nodes
AVA [12]	Search and verification allocation	Pipeline or tree hybrid	Yes	Strong adaptive verification, but narrower than full deliberation graphs
ADG(this paper)	Typed operator instances	Reusable DAG	Yes, with contracts	Unifies latent refinement, search, retrieval, verification, and action under one budgeted control rule

10.3 Failure mode 3: Contracts that are too weak or too expensive

Contracts help only if they capture real failure modes. Weak contracts create false confidence. Overly strict contracts reject useful actions. Expensive contracts can dominate the entire budget, turning the system into a bureaucracy that verifies more than it thinks.

Decisive test. Run no-contract, weak-contract, and strong-contract variants in noisy tool environments. If strong contracts do not improve success or reduce invalid states enough to pay for their cost, then contracts should be restricted to high-risk operators rather than used universally.

10.4 Failure mode 4: Wrong graph canonicalization

Reuse depends on recognizing when two nodes are "the same enough" to merge. This is notoriously domain-dependent. Overaggressive canonicalization can merge genuinely distinct states and erase valid branches. Underaggressive canonicalization leaves the graph equivalent to a tree.

Decisive test. Audit merge decisions. If merged nodes later diverge in external behavior or verifier outcomes, the canonicalization scheme is unsound. If almost no merges occur, the graph has no practical advantage.

10.5 Failure mode 5: Tasks with little reusable structure

Some tasks simply do not expose the kinds of substructure ADGis built to exploit. A single-shot open-ended essay question with no retrieval, no verifiers, no tool use, and no clean decomposition may not benefit from typed graph control. In those cases the method can only add overhead.

Decisive test. Evaluate on deliberately low-structure tasks. ADGshould either match simple baselines with small overhead or lose gracefully. If it claims gains there, the gains are likely coming from hidden confounders rather than the graph abstraction.

10.6 Failure mode 6: Distribution shift in the value model

Because ADGlearns from expensive teacher traces, it may inherit the teacher's compute preferences even when deployment budgets or task distributions differ. This is a classic offline-to-online mismatch. A teacher that searches widely may label exploratory branching as valuable under a budget regime where it is actually wasteful.

Decisive test. Stress the controller under budget shifts and tool-latency shifts. If it cannot reprice computations quickly, dual conditioning is not solving the right problem.

11 Limitations

The strongest version of this paper is still limited in several important ways.

No large-scale empirical validation is reported here. This is a conceptual and theoretical paper with a concrete empirical program, not an executed benchmark campaign. The absence of run results is real. The merit of the paper therefore rests on the sharpness of the abstraction, the usefulness of the algorithms, and the falsifiability of the proposed tests.

The framework depends on typed operators and contracts. Some domains admit clean operator typing and explicit preconditions. Others do not. If the domain offers no meaningful decomposition between refinement, retrieval, verification, and action, then ADGmay collapse back to a chain or tree with extra overhead.

Adaptive submodularity is an approximation, not a law. Real agent tasks often have complementarities rather than diminishing returns. Two individually weak computations may become useful only together. Greedy value estimates can struggle in that regime.

Canonicalization is hard. Reuse is a central selling point of the graph abstraction, but reliable merge rules are task-specific. There is no universal canonicalization function for web states, symbolic proofs, code candidates, and latent hypotheses.

Contracts do not solve semantic ambiguity by themselves. A tool output can be schema-valid and still semantically misleading. Contract enforcement prevents a class of failures, not all failures.

Controller overhead can dominate on easy tasks. Any adaptive controller introduces overhead relative to a single-pass baseline. ADGmust therefore be tested not only on hard tasks but also on easy tasks where the correct behavior is to spend almost no extra compute.

Latent reasoning raises audit questions. The framework intentionally allows some computations to remain latent and only sparsely surfaced. That is often necessary for efficiency and policy reasons, but it complicates human inspection. Provenance helps, yet it is not equivalent to full transparency.

The paper does not claim universal superiority. The method is designed for structured adaptive compute where branching, verification, reuse, and action all plausibly matter. On purely generative or stylistic tasks, simpler inference schemes may remain better.

12 Related Work

12.1 Adaptive computation and conditional depth

ACTis the direct ancestor of this paper [1]. Later adaptive-compute methods improved its stopping rule, probabilistic interpretation, or architectural substrate. PonderNet learned a distribution over pondering depth [4]. Mixture-of-Depths dynamically allocates transformer compute across tokens and layers [5]. Latent recurrent reasoning revisited the idea of scaling test-time computation through repeated latent updates rather than explicit token generation [6]. Change of Thought and related fixed-point refinement methods similarly increase internal depth without expanding output length [7]. These papers validate the continuing relevance of adaptive compute, but they remain focused on internal depth allocation rather than a general budgeted graph over heterogeneous operator types.

12.2 Test-time scaling and budgeted reasoning

The 2025 surveys by Zhang et al. and Alomrani et al. show how rapidly test-time scaling became a primary research direction [2, 3]. Recent methods control the length of reasoning traces, allocate per-subquestion budgets, or fine-tune models to make progress at each stage of generation [8, 10, 9]. These are part of the same broad movement as ADG, but most treat computation as longer or shorter generation rather than typed deliberation over graphs.

12.3 Metareasoning

The idea that computation itself should be chosen rationally predates modern deep learning. Russell and Wefald's work on limited rationality and metareasoning remains foundational [23]. More recent rational metareasoning work for language models makes the connection explicit again [24]. ADGis best understood as a modern metareasoning architecture: computation is an action selected for expected value under resource constraints.

12.4 Search, reasoning, and action in language agents

Tree of Thoughts, ReAct, and LATS all expand the space of deliberation beyond a single linear chain [14, 13, 15]. ToolTree extends this trend to more deliberate tool planning [16]. These methods are crucial stepping stones. The main difference is structural. They usually organize search as a tree, whereas ADGuses a DAG with typed reusable nodes and a single cost-sensitive controller over all operator kinds.

12.5 Verification-aware allocation

Anytime Verified Agents and related verifier-allocation work move closest in spirit to ADGby treating search and verification as resources to allocate under budget [12, 11]. ADGagrees with that direction but widens the scope. Verification is one operator class among several, not the whole control problem.

12.6 World models, retrieval, and tool contracts

Recent world-model approaches for LLM-based agents show that internal simulation can improve performance in stateful environments [17, 18]. Tool frameworks such as ToolLLM and ToolBench expanded the scale of tool-use datasets and evaluation [20]. ToolGate and related work on verified tool execution argue for stronger semantics around tool use [25]. ADGincorporates these ideas by making world-model simulation, retrieval, and contract-verified action typed computations inside a single deliberation graph.

13 Conclusion

Adaptive Computation Time asked the right question a decade too early. Its core insight, that a model should decide how much compute an input deserves, has only become more important as foundation models have turned into reasoning systems and agents. But the original answer now constrains more than it enables. Modern adaptive inference cannot be reduced to repeated recurrent updates on a chain with a fixed ponder penalty.

This paper proposed Adaptive Deliberation Graphs as the successor primitive that modern AI needed but never got. ADGpreserves what made ACTelegant: computation is a budgeted quantity whose value must justify its cost. It then generalizes the computational object from a chain of homogeneous updates to a reusable graph of typed operator applications. The framework makes it possible to compare latent refinement, branching, retrieval, verification, merge, and external action inside one controller. It replaces hand-tuned ponder penalties with dual budget pricing. It treats tool use as first-class adaptive computation with contracts rather than as an untyped appendage to decoding. It provides theoretical structure, algorithmic clarity, and a falsifiable empirical program.

The most important claim is not that graphs are always better than chains, or that one controller can dominate every inference scheme. The claim is narrower and sharper. Once reasoning systems become agents, the missing primitive is no longer "more thinking" in the abstract. The missing primitive is a way to allocate budget across heterogeneous computations that can branch, reuse, verify, and act. ADGis one such primitive. If the empirical program confirms the theory, then the natural descendant of ACTis not another halting rule. It is a budgeted deliberation graph.

A Detailed Operator Semantics

This appendix makes the operator semantics more concrete.

A.1 Refine

REFINEoperates on one hypothesis or subgoal node. It may perform several latent micro-steps before emitting one durable node. The output is typically a strengthened latent representation, an updated local plan, or a repaired candidate. REFINEshould be preferred when the controller believes that further progress is local and does not require new external information.

A.2 Branch

BRANCHexplicitly trades compute for exploration. It creates several children representing decompositions, alternative proofs, candidate programs, or candidate tool plans. The controller should learn not only how wide to branch but whether branching is worth it at all under the current budget price.

A.3 Retrieve

RETRIEVEattaches external information to a node. This can include documents, API schemas, cached memory items, or relevant previous state snapshots. Unlike standard retrieval-augmented generation, the retrieved object is a reusable graph node rather than unstructured prompt text.

A.4 Verify

VERIFYproduces a certificate node. Certificates need not be binary. They may encode partial test coverage, theorem-checking status, static analysis warnings, or probabilistic confidence from an external critic. The key is that certificate nodes are sharable and can participate in future dominance checks.

A.5 Merge

MERGEconsolidates equivalent or near-equivalent nodes. This can mean collapsing two tool states that reached the same abstract world configuration, fusing evidence from repeated retrieval, or merging candidate answers that only differ superficially. Merge is the most graph-specific operator and the easiest to implement badly; it therefore deserves application-specific care.

A.6 Act

ACTexecutes a world-changing operation. The result is a new state-snapshot node plus an observation node. Contracts govern legality and state update. Reversible actions can expose particularly high value because they reduce uncertainty while preserving recoverability.

B Proof Details

B.1 Proof of proposition 1

Let the active path at input position t be $v_t^1, \ldots, v_t^{N(t)}$. By restriction, every nonterminal operator is REFINEwith the shared state transition S. Therefore the latent state stored in v_t^n is exactly the intermediate ACTstate s_t^n. The halting controller assigns masses h_t^n until the cumulative sum exceeds $1 - \varepsilon$, so the induced stopping index and remainder term match ACTby construction. The output node is constrained to take the same convex combination of intermediate outputs that ACTuses. Finally, with scalar cost and fixed price $\lambda = \tau$, the cumulative priced cost equals the ponder cost. Thus every ADGexecution corresponds one-to-one with an ACTexecution.

B.2 Proof of theorem 1

We reduce ADGoperator selection to adaptive submodular maximization. A partial execution of ADGyields a partial realization ψ consisting of selected operator instances and their realized observations. The expected utility of the best current commit node after any completion depends only on this partial realization. Under adaptive monotonicity and adaptive submodularity, the conditional expected gain from adding a legal operator exhibits diminishing returns. The standard adaptive greedy theorem of Golovin and Krause then applies directly, yielding the $(1-1/e)$ guarantee under a cardinality budget. Cost splitting reduces the nonuniform-cost case to the unit-cost setting at the expense of standard approximation looseness.

B.3 Proof of theorem 2

At any decision point, let u^* be the best legal operator under the true marginal gain and let $\hat{u}$ be the operator selected by the controller. By bounded prediction error,

$$\Delta^*(u^*) \leq \mu_\phi(u^*) + \varepsilon_g, \tag{33}$$

$$\Delta^*(\hat{u}) \geq \mu_\phi(\hat{u}) - \varepsilon_g. \tag{34}$$

If the controller chooses an operator within additive ξ of the predicted maximizer, then

$$\mu_\phi(\hat{u}) \geq \mu_\phi(u^*) - \xi. \tag{35}$$

Combining these inequalities yields

$$\Delta^*(u^*) - \Delta^*(\hat{u}) \le 2\varepsilon_g + \xi. \tag{36}$$

Applying the standard approximate-greedy recursion for submodular maximization over at most B steps yields the final additive degradation bound.

B.4 Proof of proposition 2

By the upper-bound assumption, every legal operator has true marginal value at most η when the controller halts. Adaptive submodularity implies that after executing any further operator, all remaining operators still have marginal value at most η. Therefore any sequence of at most $M(b)$ future operator executions adds at most $M(b)\eta$ total utility. Since the optimal continuation is one such sequence, the residual gap is bounded by $M(b)\eta$.

B.5 Proof of proposition 3

Immediate by induction. Non-action operators leave symbolic state unchanged or refine it without claiming new validated external facts. For an action node, legality checks guarantee the precondition in a sound state. Validation of schema and postconditions ensures that the action effect is applied only where the contract is sound. Hence soundness is preserved.

C Implementation Recipe

A plausible first implementation of ADGcan be built with modest engineering.

1. Use a base language model that exposes hidden-state checkpoints and function-calling interfaces.
2. Represent the deliberation graph in a lightweight graph store with node embeddings, text summaries, and typed metadata.
3. Train a small controller on graphified teacher traces rather than fine-tuning the entire base model initially.
4. Start with a minimal operator set: Refine, Branch, Retrieve, Verify, and Commit. Add Actonly after contract machinery is stable.
5. In coding tasks, use test execution as the primary verifier. In retrieval tasks, use answer-consistency and citation checks. In web tasks, use schema-valid state transitions and rollback.
6. Measure reuse aggressively. If merge rarely triggers, simplify the system before scaling it.

D Example Contracts

Calculator. Precondition: expression parses and contains only allowed operators. Schema: numeric output or explicit error. Postcondition: result satisfies recomputation in a secondary parser. Effect: attach value node and mark expression resolved.

Code execution. Precondition: code compiles in the declared language and resource limits are within sandbox policy. Schema: stdout, stderr, exit code, and test results. Postcondition: execution occurred in an unmodified sandbox and test outputs are well formed. Effect: attach certificate node with pass or fail metadata.

Browser form submission. Precondition: required DOM elements exist, required fields are populated, and user policy permits the action. Schema: confirmation page, redirect, or error. Postcondition: resulting DOM satisfies expected page pattern. Effect: attach new state snapshot or trigger rollback.

Database mutation. Precondition: authentication token valid and mutation schema satisfied. Schema: returned row identifier or transaction result. Postcondition: read-after-write consistency check passes. Effect: update symbolic state with new object binding.

These examples illustrate the role of contracts without claiming that they are universally sufficient. Domain-specific contract design remains a major engineering task.

E Notation Summary

Table 7: Core notation used throughout the paper.

Symbol	Meaning
x	input prompt, problem instance, or user instruction
e_0, e_T	initial and final environment state
ω	latent task outcome or world variable
G_k	deliberation graph at computation step k
V_k, E_k	node and edge sets of G_k
$\psi(v)$	node state tuple containing type, latent state, readable trace, symbolic state, uncertainty, provenance, and cost
$\mathcal{O}$	typed operator library
$u = (o, \mathbf{v})$	operator instance consisting of operator type o and input node tuple $\mathbf{v}$
$\text{Frontier}(G)$	set of legal operator instances at graph G
$c(u)$	vector cost of operator instance u
B	vector budget
λ	dual price vector over budget dimensions
$U^*(G, b)$	optimal future utility from graph G under remaining budget b
$\Delta^*(u \mid G, b)$	true marginal value of operator u at graph G under remaining budget b
μ_ϕ, σ_ϕ	predicted marginal gain and uncertainty under controller parameters ϕ
$\widehat{\mathcal{R}}_\phi(G, b)$	estimated residual value upper bound used for halting
V_π	violation indicator under policy π

F Worked Examples

F.1 A coding example with reusable certificates

Consider a program-synthesis task where the model must write a function satisfying a hidden test suite. A pure chain-based strategy can spend more or fewer tokens, but it does not represent that the same test execution result is useful for several nearby candidate programs. In ADG, the agent can proceed as follows.

1. Create a root hypothesis node containing the problem statement and a first candidate program.
2. Use BRANCHto produce three candidate implementations: recursive, dynamic-programming, and greedy.
3. Use VERIFYto run a lightweight public test suite on all three. Each test result becomes a certificate node.
4. Discover that the recursive and dynamic-programming solutions fail the same boundary test. Because the failure certificate is reusable, the controller can apply REFINEto both candidates with the same error explanation rather than rediscovering the issue twice.
5. Apply MERGEto collapse two nearly identical repaired variants that now have identical execution signatures on the lightweight tests.
6. Spend the final budget on stronger hidden-test execution for the remaining candidate only.

A tree-search planner can imitate part of this behavior, but it will typically duplicate the shared certificate or treat it as branch-local text. ADGmakes the certificate a first-class reusable object. This is exactly the kind of compute reuse that motivates the graph abstraction.

F.2 A web-agent example with contracts

Consider a booking workflow in a browser environment. The agent must navigate to a site, select a route, fill a form, and confirm a purchase. A typical tool-using agent interleaves textual reasoning with browser actions, but it may still issue illegal actions such as clicking a button before the required fields are filled.

In ADG, the workflow becomes explicit.

1. A state-snapshot node represents the current DOM abstraction and session metadata.
2. RETRIEVEfetches site-specific schema information such as required fields or allowable route constraints.
3. BRANCHproduces several candidate action plans: direct booking, price comparison, or route modification.
4. Before any click or submit action, the ACToperator checks the action contract against the current symbolic state. If the contract fails, the action is not executed. Instead the controller can spend a small amount of budget on REFINEor RETRIEVEto repair the plan.
5. After a successful action, the resulting DOM observation is validated and converted into a new state-snapshot node. If validation fails, rollback attaches an error node and the controller can replan from the previous valid snapshot.

This example illustrates why explicit action semantics matter. Without contracts, invalid actions pollute the state and waste environment budget. Without graph snapshots, replanning after partial failure becomes opaque and hard to audit.

F.3 A mathematical reasoning example with selective verification

Suppose the task is a difficult algebra problem. A naive agent might generate many long candidate proofs or many samples for self-consistency. ADGcan instead reason about which kind of compute is worth buying.

1. The controller first applies REFINEto obtain a compact latent plan.
2. Uncertainty remains high on one symbolic manipulation, so it applies BRANCHto create two candidate derivations.
3. It then uses VERIFYon the disputed manipulation only, for example with a symbolic algebra engine.
4. Because the certificate resolves the contested step, the second branch becomes dominated and is pruned.
5. The controller halts once the residual value of any further branching or verification falls below the priced cost.

The point is not that math reasoning always needs formal verification. The point is that verification becomes a selective operator with explicit marginal value rather than a fixed outer loop.

G Minimal Report Card for Future Empirical Papers

If a future empirical paper claims to instantiate ADGat scale, the minimal report card should include the following tables and plots.

1. A utility–budget curve for each task family.
2. A per-operator cost breakdown showing how often REFINE, BRANCH, RETRIEVE, VERIFY, MERGE, and ACTwere chosen.
3. A merge and reuse analysis quantifying graph-specific savings relative to a tree execution.
4. Calibration plots for gain estimates and residual-value stopping bounds.
5. A contract analysis showing precondition failures, postcondition failures, rollback frequency, and downstream success after rollback.
6. A sensitivity analysis over budget vectors, not only a scalar budget.

Without this report card, an ADGimplementation could hide its true behavior behind aggregate scores. The method's claim to novelty lies in compute allocation structure, so the structure must be measured directly.

References

[1] Alex Graves. Adaptive Computation Time for Recurrent Neural Networks. *arXiv preprint arXiv:1603.08983*, 2016.

[2] Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, Irwin King, Xue Liu, and Chen Ma. A Survey on Test-Time Scaling in Large Language Models: What, How, Where, and How Well? *arXiv preprint arXiv:2503.24235*, 2025.

[3] Mohammad Ali Alomrani, Yingxue Zhang, Derek Li, Qianyi Sun, Soumyasundar Pal, Zhanguang Zhang, Yaochen Hu, Rohan Deepak Ajwani, Antonios Valkanas, Raika Karimi, Peng Cheng, Yunzhou Wang, Pengyi Liao, Hanrui Huang, Bin Wang, Jianye Hao, and Mark Coates. Reasoning on a Budget: A Survey of Adaptive and Controllable Test-Time Compute in LLMs. *arXiv preprint arXiv:2507.02076*, 2025.

[4] Andrea Banino, Jan Balaguer, and Charles Blundell. PonderNet: Learning to Ponder. *arXiv preprint arXiv:2107.05407*, 2021.

[5] David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. Mixture-of-Depths: Dynamically Allocating Compute in Transformer-Based Language Models. *arXiv preprint arXiv:2404.02258*, 2024.

[6] Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up Test-Time Compute with Latent Reasoning: A Recurrent Depth Approach. *arXiv preprint arXiv:2502.05171*, 2025.

[7] Mrinal Mathur, Mike Doan, Barak Pearlmutter, and Sergey Plis. Change of Thought: Adaptive Test-Time Computation. *arXiv preprint arXiv:2507.13569*, 2025.

[8] Taneesh Gupta, Rahul Madhavan, Rishabh Tiwari, Xuchao Zhang, Chetan Bansal, Saravan Rajmohan, and Kurt Keutzer. ACTS: Adaptive Control for Test-time Scaling. *OpenReview submission to ICLR 2026*, 2025.

[9] Yuxiao Qu, Matthew Y. R. Yang, Amrith Setlur, Lewis Tunstall, Edward Emanuel Beeching, Ruslan Salakhutdinov, and Aviral Kumar. Optimizing Test-Time Compute via Meta Reinforcement Finetuning. *Proceedings of the International Conference on Machine Learning*, 2025.

[10] Junhong Lin, Xinyue Zeng, Jie Zhu, Song Wang, Julian Shun, Jun Wu, and Dawei Zhou. Plan and Budget: Effective and Efficient Test-Time Scaling on Reasoning Large Language Models. *arXiv preprint arXiv:2505.16122*, 2025.

[11] Ahsan Bilal, Ahmed Mohsin, Muhammad Umer, Ali Subhan, Hassan Rizwan, Ayesha Mohsin, and Dean Hougen. What If We Allocate Test-Time Compute Adaptively? *arXiv preprint arXiv:2602.01070*, 2026.

[12] Anonymous. Anytime Verified Agents: Adaptive Compute Allocation for Reliable LLM Reasoning under Budget Constraints. *Under review at Transactions on Machine Learning Research*, 2025.

[13] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629*, 2022.

[14] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *arXiv preprint arXiv:2305.10601*, 2023.

[15] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models. *arXiv preprint arXiv:2310.04406*, 2023.

[16] Shuo Yang, Soyeon Caren Han, Yihao Ding, Shuhe Wang, and Eduard Hoy. ToolTree: Efficient LLM Agent Tool Planning via Dual-Feedback Monte Carlo Tree Search and Bidirectional Pruning. *arXiv preprint arXiv:2603.12740*, 2026.

[17] Shangmin Guo, Omar Darwiche Domingues, Rapha"el Avalos, Aaron Courville, and Florian Strub. World Modelling Improves Language Model Agents. *arXiv preprint arXiv:2506.02918*, 2025.

[18] Xiao Yu, Baolin Peng, Ruize Xu, Yelong Shen, Pengcheng He, Suman Nath, Nikhil Singh, Jiangfeng Gao, and Zhou Yu. Reinforcement World Model Learning for LLM-based Agents. *arXiv preprint arXiv:2602.05842*, 2026.

[19] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A Realistic Web Environment for Building Autonomous Agents. *arXiv preprint arXiv:2307.13854*, 2023.

[20] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. *arXiv preprint arXiv:2307.16789*, 2023.

[21] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. *International Conference on Learning Representations*, 2021.

[22] Daniel Golovin and Andreas Krause. Adaptive Submodularity: Theory and Applications in Active Learning and Stochastic Optimization. *Journal of Artificial Intelligence Research*, 42:427-486, 2011.

[23] Stuart J. Russell and Eric Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, 1991.

[24] C. Nicolò De Sabbata, Theodore R. Sumers, Badr AlKhamissi, Antoine Bosselut, and Thomas L. Griffiths. Rational Metareasoning for Large Language Models. *arXiv preprint arXiv:2410.05563*, 2025.

[25] Yanming Liu, Xinyue Peng, Jiannan Cao, Xinyi Wang, Songhang Deng, Jintao Chen, Jianwei Yin, and Xuhong Zhang. ToolGate: Contract-Grounded and Verified Tool Execution for LLMs. *arXiv preprint arXiv:2601.04688*, 2026.

Table 6: Failure modes, symptoms, and decisive falsification tests for ADG. A serious evaluation should report these before headline benchmark wins.

Failure mode	Observable symptom	Decisive test
Poor gain estimation	Early stopping on hard problems or overthinking on easy ones	Compare predicted marginal gains with oracle continuation values and report calibration
Frontier explosion	Compute spent mostly on bookkeeping and branch maintenance	Track effective branching factor and utility per added node under increasing budgets
Weak or costly contracts	High validation overhead or silent invalid states	Evaluate no-contract, weak-contract, and strong-contract variants in noisy environments
Bad canonicalization	Incorrect merges or no practical reuse	Audit merge errors and compare reuse ratio against a tree with identical frontier policy
Little reusable structure	Overhead with no accuracy gain	Evaluate on low-structure tasks and check whether ADGdegrades gracefully
Budget-distribution shift	Controller spends as if teacher budget still applies	Change latency, tool cost, or action caps at test time and measure adaptation