Mutations: A Framework for Bounded Architectural Adaptation in Shared-Base Language Models

Nicola Flavio Quatraro

Independent Researcher
nicolaflavioquatraro@gmail.com

July 9, 2026

Abstract

This paper proposes Mutations, a framework for modular experience-derived adaptation in frozen language-model agents. The framework is motivated by the goal of enabling capable agentic behavior on customer hardware, such as 16GB or 24GB GPUs and unified-memory Apple Silicon systems, where very large models and very long context windows may be impractical. A mutation is defined broadly as a bounded adaptation unit that can be routed, validated, and composed with other adaptations while preserving the frozen base model as the general reasoning engine. Episodic and procedural mutations are introduced as early examples, but the framework is intended to support many adaptation types. Rather than claiming empirical superiority, this paper defines the architecture, its intended efficiency advantages, and a research program for evaluating it against retrieval, task-dependent skills and memory, larger vanilla models, and other adaptation strategies.

1 Introduction

Modern language-model agents can solve increasingly complex tasks when they are backed by large models, long context windows, extensive retrieval systems, tool-use policies, and multi-step orchestration [44, 33, 27]. However, these capabilities often assume computational resources that are not available on ordinary customer hardware. A practical adaptation framework should therefore ask a different question: can a relatively small frozen model become more capable over repeated experience without requiring a larger model, a much longer context window, or full-model retraining?

The Mutations framework explores this direction. Its central idea is to keep a base language model frozen while converting selected experience into small, bounded, reusable adaptation modules called mutations. At runtime, a router or arbitration mechanism selects which mutations are likely to help, obtains compact structured support from them, validates this support, and presents it to the base model before the final decision. This creates a modular form of adaptation between ordinary retrieval and monolithic fine-tuning.

The intended deployment target is deliberately modest: systems with roughly 16GB or 24GB of GPU memory or unified memory. This includes many consumer workstations and Apple Silicon laptops. The framework is therefore designed around memory efficiency, limited context budgets, and selective activation of small adaptation units.

2 Motivation and Scope

Retrieval-augmented generation and its derivatives are powerful non-parametric memory mechanisms [16, 7, 1, 40], but they also expose several limitations. Retrieved text consumes context budget, retrieval quality depends on chunking and indexing decisions, and downstream reasoning can degrade when the retrieved context is noisy, redundant, stale, poorly positioned, or poorly matched to the current decision [21]. More elaborate retrieval harnesses can mitigate these issues, but they often add additional engineering complexity, latency, and memory pressure.

Language models are themselves strong retrieval and association systems: much of their usefulness comes from the ability to activate relevant patterns from their parameters given an appropriate prompt. This suggests a complementary strategy. Instead of storing all experience as text and retrieving it at inference time, recurring patterns may be distilled into small parameter-efficient adaptations that improve the quality of the model's internal retrieval and decision support without modifying the base model [8, 18, 15, 9, 29]. Because the base model remains frozen, new mutations can be added, removed, compared, or retired without the same catastrophic-forgetting risks associated with continual modification of the main model [31, 22, 38].

This paper is a framework proposal rather than an experimental performance claim. It introduces definitions, architecture, and evaluation methodology for studying mutations. The empirical question is not whether mutations can defeat every possible retrieval system, since retrieval systems can be engineered in many ways. The more focused question is whether routed modular adaptations can provide useful capability and efficiency tradeoffs under fixed memory, context, and inference budgets.

3 The Mutations Framework

A mutation is any bounded model modification derived from experience, task structure, or domain knowledge that preserves at least one element of an original frozen base model. The preserved element may be large, such as the first n transformer layers, an embedding stack, a tokenizer and input interface, or most of the original parameterization; or it may be smaller, such as a shared submodule, routing interface, attention block, representation space, or frozen component used by a specialized branch. Under this definition, a system is a mutation of the base model when it remains genealogically and operationally tied to that base rather than being an independently trained model with no retained component.

This definition includes both parameter-level and architectural changes. A mutation may be a PEFT adapter, a set of changed weights, a merged task vector, a modified attention block, a specialized head, branched later layers, added layers, replaced submodules, altered routing components, a representation-level intervention, or a distilled module that reuses a frozen prefix of the base model [9, 20, 41]. The key requirement is not that every mutation use the same mechanism, but that each mutation remain bounded, inspectable, composable with the framework, and explicitly connected to the base model artifact it modifies or extends.

Other mechanisms in the surrounding system may support mutations without being mutations themselves. Prompt programs, learned routers, memory updaters, tool-use policies, verifiers, and retrieval sub-systems may decide when a mutation should be used, supply evidence to it, validate its outputs, or arbitrate among several mutations. They become mutations only when they are themselves implemented as bounded modifications of, or extensions to, a retained component of the frozen base model.

The term "mutation" therefore denotes the research object rather than only an adapter check-

point. A mutation packages together what model elements are preserved, what elements are modified or added, what it is for, when it should be used, what data produced it, how it emits support, and how its usefulness is measured. Episodic and procedural mutations are early examples, not an exhaustive taxonomy.

The framework is loosely inspired by the way different brain areas specialize for different functions while participating in coordinated behavior. The analogy is not intended as a biological claim. It motivates a system design in which many specialized modules can be orchestrated through fixed rules, learned routing models, or adaptive arbitration algorithms to unlock capabilities that may be difficult for a small base model to express alone. Conditional-computation systems such as sparsely gated mixture-of-experts models show one precedent for activating specialized capacity selectively, although mutations differ by emphasizing inspectable adaptation artifacts and lifecycle management rather than only scaling model capacity [34, 6].

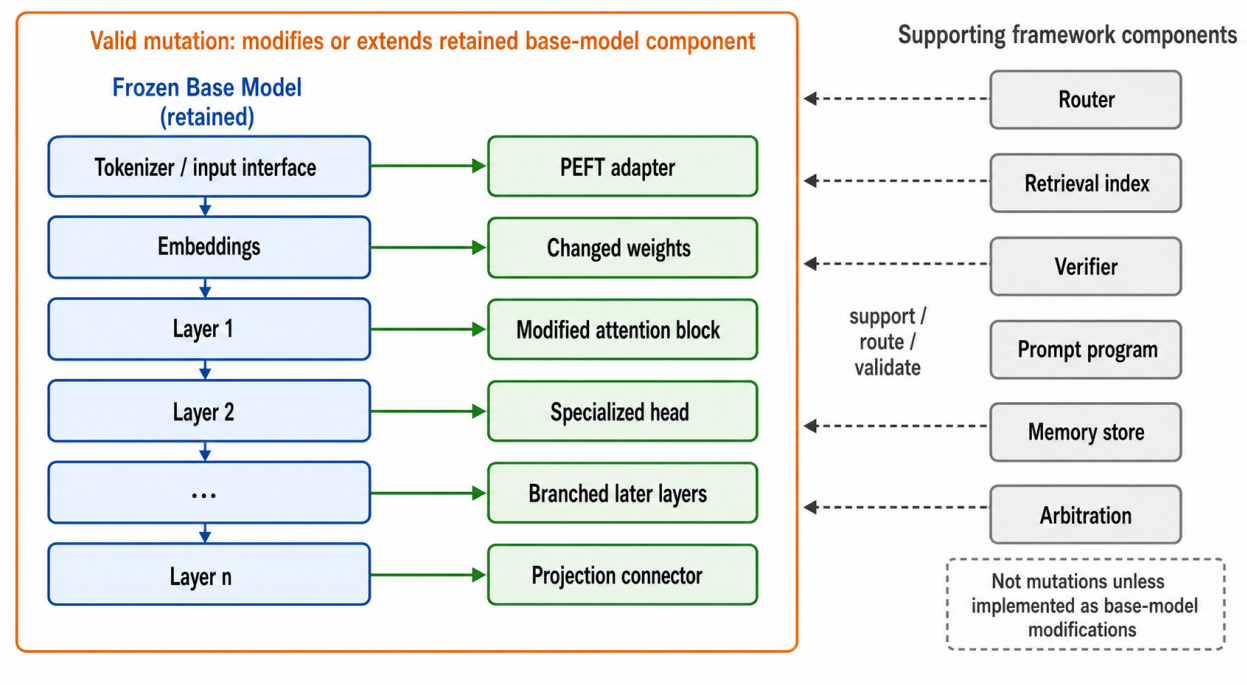

Figure 1: Mutation boundary. A mutation is a bounded modification or extension of a retained frozen base-model component. Routers, retrieval systems, verifiers, prompt programs, memory stores, and arbitration mechanisms support mutations but are not themselves mutations unless implemented as base-model modifications.

3.1 Frozen Base Model

The frozen base model is the stable reference system from which mutations inherit part of their identity. Its parameters are not updated during ordinary mutation creation, routing, or deployment. This constraint gives the framework a fixed behavioral anchor: mutations may specialize, correct, or extend the system, but the base model remains the common coordinator, language interface, and general reasoning engine.

At inference time, the base model receives the current task state together with any validated support produced by active mutations. It is responsible for the final action or answer unless a particular deployment explicitly delegates a bounded decision to a specialized mutation. This

preserves a simple control structure: mutations provide specialized capability or evidence, while the base model integrates that support with the current context, user instruction, tool state, and safety constraints.

Freezing the base model also makes mutation evaluation more tractable. Because the underlying model is stable, changes in behavior can be attributed to the routed mutation set, the router, the arbitration policy, or the surrounding memory system rather than to continual drift in the main model. This enables direct comparison among alternative mutations, rollback of harmful mutations, and measurement of marginal utility under fixed memory, latency, and context budgets.

3.2 Mutation Specifications

A mutation should be represented by an explicit specification rather than only by a checkpoint or code artifact. The specification defines the mutation's relation to the base model, its intended scope, and the conditions under which it may be used. At minimum, it should identify the compatible base model, the retained base-model elements, the modified or added components, the training or construction procedure, and the runtime interface exposed to the router.

A useful `MutationSpec` contains several classes of metadata. Compatibility metadata records the base model version, tokenizer assumptions, retained layers or submodules, parameter format, quantization requirements, and hardware constraints. Provenance metadata records the traces, documents, human feedback, synthetic examples, or external datasets that produced the mutation. Scope metadata states the tasks, domains, failure modes, or environment states for which the mutation is intended, along with known exclusions.

Runtime metadata specifies triggers, expected cost, input schema, output schema, activation constraints, and composition rules. For example, a mutation may declare that it expects a recent observation, a task goal, and a small set of retrieved facts, and that it emits a bounded context packet containing candidate actions, warnings, or subgoal guidance. Validation metadata defines tests for schema conformance, grounding, consistency, behavioral usefulness, and regression risk. Evaluation metadata records benchmark results, held-out trace performance, latency, active memory cost, and known interactions with other mutations.

This specification is part of the mutation as a research object. Without it, a modified model component may still be an artifact, but it is difficult to route, compare, audit, compose, or retire. With it, mutation management becomes closer to package management for model modifications: a local system can decide whether a mutation is compatible, trustworthy, useful, and worth activating under current constraints.

3.3 Context Packets

Context packets are compact, structured outputs produced by active mutations for consumption by the base model or by an arbitration layer. They are not raw hidden states or unconstrained natural-language continuations. Their purpose is to expose the useful result of a mutation in a form that can be inspected, validated, merged with other packets, and placed into the base model's context without consuming excessive tokens.

A context packet should include the mutation identifier, the packet type, a confidence or priority signal, the relevant source references, and a bounded payload. The payload may contain candidate actions, repaired commands, subgoals, constraints, warnings, retrieved facts, uncertainty estimates, or short procedural guidance, depending on the mutation's declared interface. The packet should also make clear whether its contents are observations, inferences, recommendations, or prohibitions.

Source awareness is important because mutation outputs should not be treated as ungrounded authority. When a packet is derived from a trace, document, user correction, synthetic example, or benchmark result, the packet should preserve enough provenance for the system to audit why it was emitted. In privacy-sensitive settings, this provenance may be a local reference or hashed identifier rather than raw source text.

The main risk is that useful mutation outputs can themselves become a new form of context bloat. If many mutations emit long packets, the framework may recreate the same bottleneck it is meant to avoid: excessive prompt length, slower inference, higher memory use, and weaker attention to the most relevant evidence. In a consumer-hardware setting, context packets should therefore be treated as scarce budgeted objects rather than free explanatory text. Packet schemas should include size limits, priority fields, expiration rules, and validation checks that penalize unnecessary verbosity.

The context-packet abstraction gives the framework a common integration layer across heterogeneous mutations. A PEFT mutation, a branched-layer mutation, and a specialized-head mutation may have very different internal implementations, but each can emit a packet whose size, schema, and validation criteria are known. Arbitration can then discard stale packets, resolve contradictions, rank packets by expected usefulness, and pass only the most relevant support to the frozen base model.

3.4 Routing and Arbitration

Routing is the mechanism that decides which mutations to consult for a given situation. It may begin as a fixed heuristic, such as always invoking a procedural mutation in a known class of environment states. It may later become an adaptive model that estimates expected usefulness, cost, and risk. Arbitration then merges or rejects mutation outputs before the base model receives them. Adapter composition methods such as AdapterFusion and LoRAHub suggest that learned or search-based composition can be useful when multiple task adaptations are available, but the Mutations framework also requires activation to be evaluated against runtime cost, provenance, and packet-level validation [28, 10].

This routing layer is essential to the efficiency argument. A system with many mutations should not activate all of them at every step. It should activate only the adaptations expected to improve the current decision enough to justify their memory, latency, and context costs.

The policies that create, revise, remove, route, or combine mutations need not be produced by the same small model that runs at deployment time. In one possible configuration, a larger and more capable model acts as an offline supervisor that analyzes traces, proposes mutation candidates, selects training objectives, writes validation criteria, or tunes routing algorithms for a particular task family. The resulting mutation system can then run locally with much lower recurring cost. This separates occasional expensive supervision from frequent inexpensive inference.

4 Episodic and Procedural Adaptation

The first version of the framework distinguishes episodic and procedural adaptation because they represent two common kinds of agent experience. Episodic adaptation concerns facts about a particular environment instance, run, or task. Procedural adaptation concerns reusable operations or strategies that may transfer across instances.

This split is useful for initial experiments, but it should not be treated as fundamental or complete. Other mutation types may target perception, planning, command repair, uncertainty estimation, safety checking, tool selection, compression, verification, curriculum generation, or

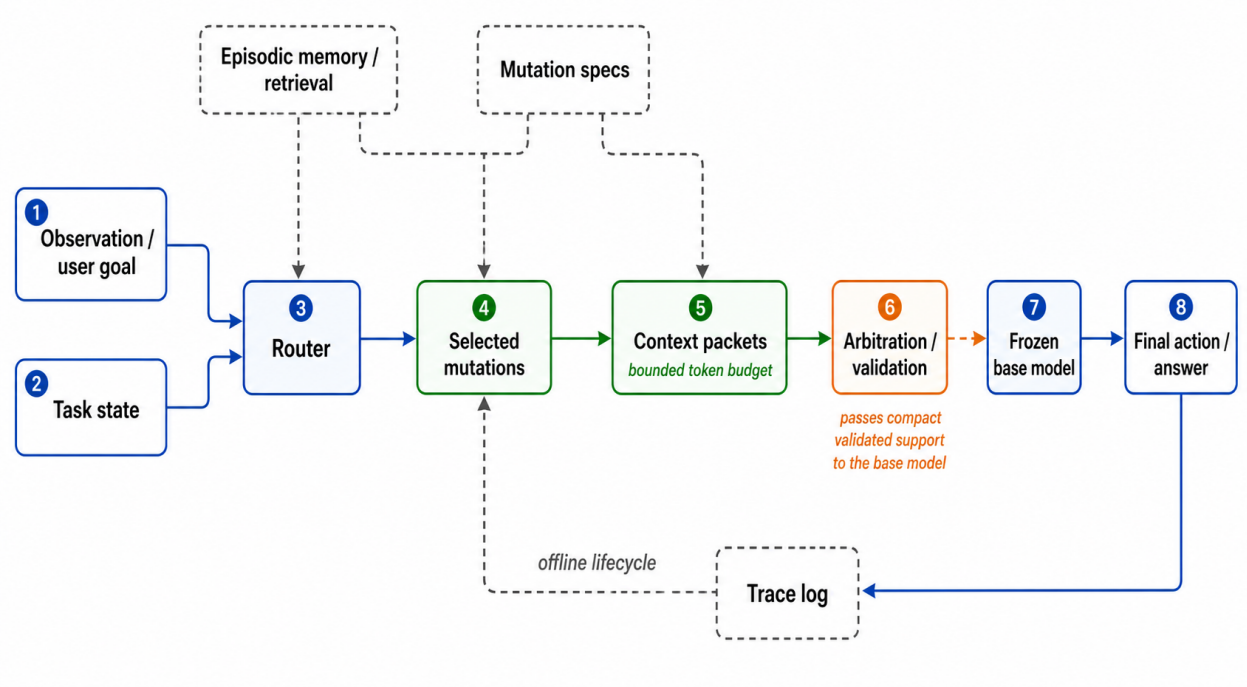

Figure 2: Runtime flow. The router selects mutations using task state, episodic memory, and mutation specifications. Active mutations emit bounded context packets, arbitration validates and merges them, and the frozen base model produces the final action or answer.

domain-specific skills. The general framework only requires that each mutation expose a bounded interface and be evaluated under controlled conditions.

4.1 Episodic World Models

Episodic adaptation concerns information tied to a particular run, environment instance, user, project, or recent interaction history. Examples include the current state of a game world, facts discovered during a task, user preferences revealed in conversation, files inspected during a coding session, or failed actions and their observed consequences. This information is often volatile: it may become stale quickly, conflict with later observations, or matter only inside one local context.

For this reason, the first representation for episodic knowledge should usually be non-parametric. Traces, structured notes, summaries, databases, and retrieval indexes are easier to update, delete, inspect, and correct than model weights. A local agent can maintain an episodic world model as a set of source-aware documents: observations, inferred state, uncertainty notes, action logs, user corrections, and task-specific summaries. Retrieval can then expose the relevant pieces to the router, mutations, or base model when needed. Agent-memory systems that store observations, reflections, and retrieved experiences provide useful precedents for this non-parametric layer [26, 35].

Within the Mutations framework, episodic data becomes mutation material when repeated patterns justify a bounded model modification. For example, a system may observe that a user repeatedly organizes files in a particular way, that a game environment follows recurring map conventions, or that a class of tasks requires the same state-update rule. The raw facts may remain in external memory, while a mutation learns how to interpret, compress, update, or query that episodic state more effectively. In this sense, episodic world models provide both runtime evidence and training material for later mutations.

The case is stronger when episodic information is effectively immutable. If a corpus, manual, map, API specification, codebase snapshot, game guide, or completed project archive is not expected to change during deployment, then distilling some of its structure into a mutation may reduce the need to retrieve across many external documents at every step. Related memory-as-model work such as MeMo similarly argues that encoding corpus-derived knowledge into a dedicated model can capture cross-document relationships and make inference cost less dependent on corpus size [29]. In the Mutations framework, such an artifact counts as a mutation only when it modifies or extends a retained component of the frozen base model. The mutation does not need to memorize every fact verbatim; it may instead learn stable abstractions, lookup strategies, source organization, entity relationships, or task-relevant regularities that make retrieval smaller, more targeted, or unnecessary for common decisions. Immutable episodic sources therefore sit between ordinary external memory and procedural knowledge: they are instance-specific, but stable enough that a bounded model modification can amortize their access cost across many future uses.

The evaluation of episodic support should therefore separate freshness from generalization. External documents and retrieval should be judged by factual accuracy, update latency, source grounding, and deletion behavior. Episodic mutations should be judged by whether they improve state tracking, relevance selection, compression, or decision support without memorizing stale private details in a way that cannot be audited or removed.

4.2 Procedural Policy Support

Procedural adaptation concerns reusable ways of acting rather than facts about a single episode. It captures patterns such as how to repair invalid commands, how to decompose a task, how to recover from a repeated failure mode, how to choose tools, how to check preconditions before acting, or how to translate a high-level intention into environment-specific operations. These patterns are strong candidates for mutation because they often transfer across many instances while remaining narrow enough to train and evaluate separately. ReAct, Reflexion, Voyager, Toolformer, Gorilla, and API-Bank illustrate related ways of coupling language models to action traces, tool use, self-correction, and reusable skill libraries [44, 35, 37, 33, 27, 17].

A procedural mutation may modify the base model through an adapter, changed weights, a specialized head, a branch over later layers, or another bounded architecture that preserves part of the frozen base. Its runtime output should usually be small: a candidate action, a repaired command, a warning, a short plan fragment, or a ranked set of next-step constraints. The goal is not to replace the base model's general reasoning, but to give it reliable local skill in situations where repeated experience has exposed a stable procedure.

Command repair is a simple example. An agent operating in a text environment may repeatedly propose actions that are semantically reasonable but syntactically invalid. A procedural mutation can be trained on failed intentions, valid command forms, and successful repairs, then emit a compact context packet suggesting the corrected command or the rule that makes the original command invalid. Similar mutations could specialize in tool argument repair, API call sequencing, navigation routines, test-failure triage, or safety checks before irreversible actions.

Procedural mutations should be evaluated on held-out tasks and adversarial near misses, not only on the traces that produced them. A useful procedural mutation improves success rate, reduces repeated failures, and lowers token or latency cost without overfitting to one environment instance. It should also expose its scope clearly: a command-repair mutation trained for one game, shell, API, or workflow should not be treated as a universal policy unless transfer has been measured.

4.3 Token Compression and Projection

Context compression is a natural third target for mutation research. Episodic mutations decide what facts should be available, and procedural mutations decide what reusable behavior should be reinforced; compression mutations decide how much of that support can be delivered to the frozen base model under strict token, memory, and latency limits. This is especially important on consumer hardware, where the framework cannot assume that every useful trace, retrieved fact, and mutation output can be placed directly into the prompt. Prompt and context compression methods provide useful baselines for this part of the framework [12, 13, 19, 30].

The simplest compression approaches are algorithmic. The router or arbitration layer can discard low-priority packets, remove stale packets, suppress packets that duplicate stronger evidence, cap the number of active mutations, or select only the highest expected-utility payloads under a token budget. It can also merge compatible packets into a shorter summary, preserve only the decision-relevant fields of a packet, or ask mutations to emit different levels of detail depending on the current context budget.

More ambitious approaches are learned. A specialized compression mutation could learn to map many tokens of retrieved evidence, episodic state, or mutation output into a smaller representation that remains useful to the frozen base model. One version would emit a short textual packet. Another version would operate below the text interface by projecting compressed representations directly into the base model's early layers or embedding space. This resembles the way vision encoders are attached to already trained language models through learned projection layers: an external encoder produces compact features, an MLP or related connector maps those features into the language model's representational space, and the frozen model consumes them as if they were part of its own input stream.

Such learned projection modules would count as mutations when they preserve and attach to a retained component of the frozen base model. Their evaluation should measure not only task success, but compression ratio, information loss, latency, active memory cost, robustness to irrelevant inputs, and whether the projected representation introduces misleading evidence. The research question is whether a small learned connector can preserve the decision-relevant content of many context tokens more efficiently than textual summaries or longer prompts.

5 Runtime Protocol

At a high level, a running Mutations framework instance alternates between two phases.

1. **Interaction and data gathering.** The agent interacts with the environment, records observations, actions, outcomes, intermediate model outputs, mutation activations, failures, and task-level results. During this phase, the system behaves like an ordinary agent augmented by whatever mutations are currently deployed.

2. **Elaboration and mutation update.** The system processes the saved data, constructs or updates documents, mines recurring patterns, generates synthetic examples, trains new mutations or revises existing ones, validates their outputs, and updates routing or arbitration metadata.

The interaction phase can run under strict runtime constraints, while the elaboration phase can be scheduled between episodes, after failures, during idle periods, or on a separate machine. This separation is important for local deployment because training and data synthesis may be too expensive to perform at every decision step, but feasible between runs.

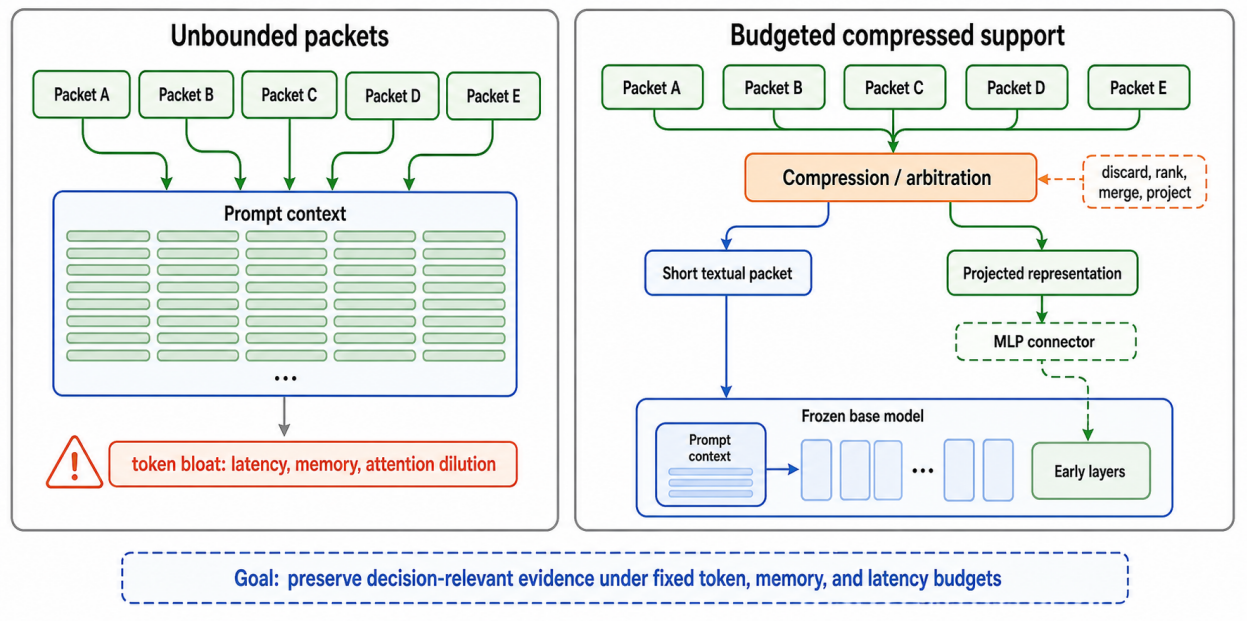

Figure 3: Context-packet budgeting and compression. Unbounded mutation outputs can recreate the context-bloat problem. Compression and arbitration can instead discard, rank, merge, or project packet content into compact textual or learned representations under fixed token, memory, and latency budgets.

6 Learning and Mutation Lifecycle

The mutation lifecycle begins with saved interaction data. Traces are converted into structured documents, summaries, state updates, failure reports, and candidate training examples. A synthesis component then produces mutation-specific datasets. These datasets should match the desired runtime behavior of the mutation: for example, a command-repair mutation should learn to map failed intentions to valid commands, while a planning mutation should learn to emit compact subgoal guidance.

Recent open-source generative models, including diffusion-style language models, may make synthetic data generation increasingly practical on consumer hardware. This matters because the framework depends on generating many small, targeted datasets rather than one large monolithic fine-tuning corpus. Local synthetic data generation also supports privacy-sensitive deployments in which raw interaction traces should not leave the user's machine.

A much larger model, open or proprietary, can also guide the lifecycle without being part of the deployed local system. Such a supervisor may inspect traces to identify recurring failure modes, decide whether a problem calls for episodic memory, procedural specialization, compression, or no mutation at all, propose the retained base-model components to modify, choose training objectives, generate synthetic edge cases, define packet schemas, and write validation tests. It may also compare candidate mutations, suggest routing labels, estimate expected utility under token and latency budgets, detect overlapping mutations that should be merged, or recommend retiring a mutation whose gains no longer justify its cost. For example, a large supervisor could analyze a set of invalid tool calls and propose a command-repair mutation; inspect a stable API manual and propose an immutable-source episodic mutation; or review many verbose context packets and propose a learned compression connector.

After training, a mutation is validated before deployment. Validation should check schema conformance, grounding in source data, bounded output size, consistency with current observations, and measured usefulness on held-out traces or tasks. Mutations that fail validation can be revised, retired, or kept out of the router.

Mutation creation and removal can therefore be treated as an optimization problem over task performance, memory budget, latency, reliability, and transfer. A capable offline supervisor, a local heuristic, a learned controller, or a community-maintained evaluation process may decide that a mutation should be created when recurring failures are detected, revised when its support becomes stale, merged when it overlaps with another mutation, or removed when it no longer improves decisions under the current budget.

This also enables distributed specialization. Multiple Mutations framework instances may solve different parts of a larger problem by training or selecting different mutation sets. One instance might specialize in long-context retrieval, another in planning, another in command repair, and another in verification. Their outputs can be composed by an upper-level coordinator or shared as reusable mutation artifacts.

Human feedback can also be treated as a first-class signal in the mutation lifecycle. A user interacting with the system may approve or reject actions, correct mistaken assumptions, label useful or harmful mutation outputs, identify goals that the system has misunderstood, or request that certain behaviors be reinforced or suppressed. These signals can influence whether mutations are added, updated, merged, or removed. They can also improve synthetic dataset generation, validation criteria, and routing labels.

This creates a path for progressive personalization. Rather than requiring the user to manually engineer prompts or memory rules, the system can translate feedback into structured training examples, preference signals, negative examples, and routing constraints. Over time, a human can adapt a local Mutations framework instance toward personal goals, workflows, and tolerances while keeping the base model frozen.

Not all mutations need to add new behavior. Some may act as suppressors or corrective modules that reduce known failure modes. A negative mutation could detect repeated loops, inhibit unsafe tool calls, flag hallucinated state claims, reduce overconfident retrieval use, or block invalid command patterns. This broadens the framework from capability acquisition to behavior shaping: mutations can accelerate useful patterns, but they can also dampen harmful or unproductive ones.

As mutation libraries grow, the system must manage a mutation ecology. Mutations may cooperate, compete, overlap, conflict, become obsolete, or lose value as the base model or task distribution changes. The lifecycle should therefore include pruning unused mutations, merging highly overlapping mutations, detecting contradictory support packets, ranking mutations by marginal utility, and preventing mutation bloat. The goal is not to accumulate every possible adaptation, but to maintain a compact and useful population under explicit resource constraints.

6.1 Mutation Compression and Work Orders

Over time, a system may accumulate many small mutations whose combined overhead becomes larger than their individual benefits. The framework should therefore support mutation compression and consolidation. Related mutations may be distilled into a smaller set of modules, merged into a single adaptation, or reorganized into a hierarchy of coarse and fine-grained mutations. This is especially relevant when several mutations activate together frequently or when their context packets contain overlapping guidance.

Existing model and adapter merging techniques suggest several possible mechanisms for this process. Weight averaging approaches such as model soups show that multiple fine-tuned models

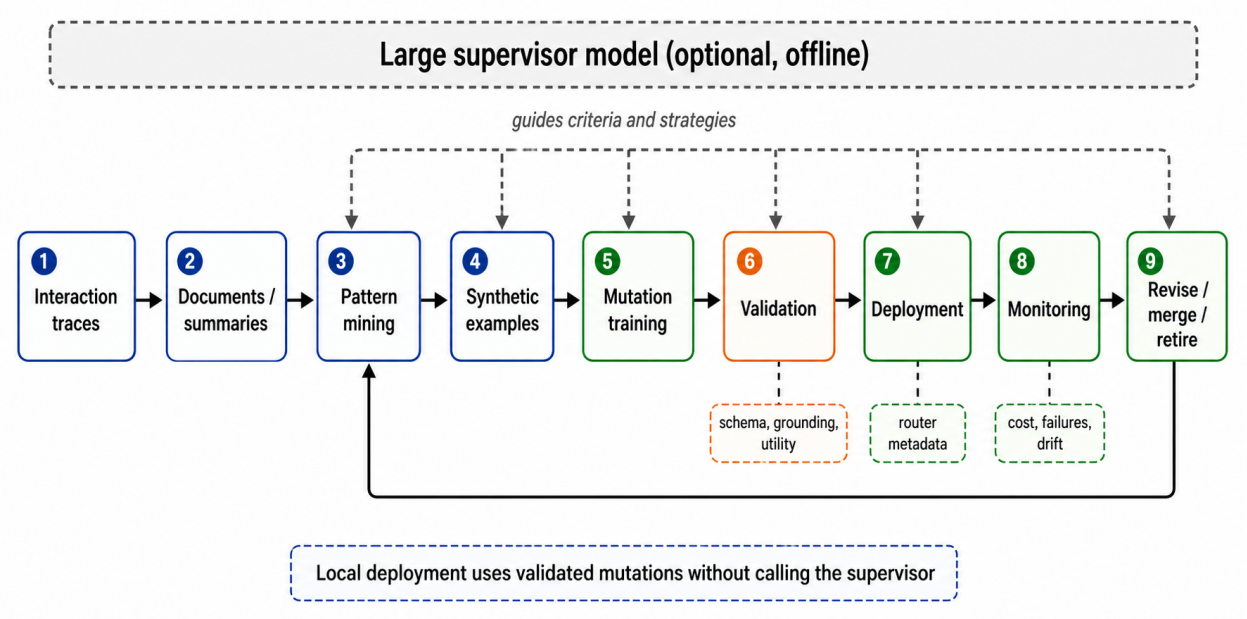

Figure 4: Mutation lifecycle. Interaction traces are transformed into documents, mined for patterns, converted into synthetic examples, trained into mutations, validated, deployed, monitored, and revised, merged, or retired. A larger optional supervisor can guide criteria and strategies offline while local deployment uses only validated artifacts.

can sometimes be averaged without increasing inference cost [39]. Task-vector methods represent fine-tuning deltas as directions that can be added or subtracted in weight space [11]. More recent merging methods such as TIES-Merging address interference between merged parameter updates [42]. Adapter-specific approaches, including AdapterFusion, AdapterSoup, and LoRA composition methods such as LoRAHub, are particularly relevant because mutations may often be implemented as small adapters rather than full model copies [28, 2, 10]. These techniques do not solve mutation consolidation by themselves, but they provide a starting vocabulary for studying when multiple adaptations can be combined without losing their specialized value.

The same lifecycle can also be used proactively through mutation work orders. A larger supervisor, a human user, or a planning component may decompose a difficult objective into requests for new or revised mutations: train a verifier for a failure mode, create a retrieval mutation for long documents, build a suppressor for repeated loops, refine a planner for a task family, or merge several redundant command-repair mutations. Work orders turn the framework into an active adaptation process rather than a passive repository of modules.

7 Efficiency Argument

The core efficiency hypothesis is that a smaller frozen model plus routed mutations can, in some repeated-task settings, approach or exceed the practical performance of larger vanilla models while using less memory and shorter prompts. Instead of increasing the base model size or placing all prior experience into context, the system activates compact adaptation units only when they are expected to help.

For PEFT-style mutations, most of the base model remains unchanged. The additional memory cost is concentrated in small adapter parameters or other bounded artifacts [8, 9, 18, 15, 20]. If the

runtime can load, swap, merge, drop, or selectively activate these adaptations efficiently, the agent may accumulate a growing library of capabilities without paying the full inference cost of a larger model at every step [32].

Sharing a common base model also creates opportunities for parallel inference. If multiple mutations reuse the same early layers and differ only in later layers, adapters, heads, or specialized branches, the runtime can compute the shared prefix once or batch related computations across mutations. In architectures where only the last n layers or selected submodules differ, this may reduce the marginal cost of consulting several mutations compared with running fully separate models.

This efficiency goal is especially important for local deployment. A useful target is operation on consumer hardware with approximately 16GB or 24GB of GPU memory or unified memory. Under this constraint, long-context retrieval-heavy systems and larger vanilla models may be possible only with compromises in speed, quantization, context length, or concurrent capability. Quantized adaptation methods such as QLoRA show that strong fine-tuning can be made substantially more memory efficient, and mutation-based systems inherit this motivation while adding routing and lifecycle constraints [5]. Mutations aim to make adaptation more granular: the agent pays for the specialized support it needs rather than for a globally larger model or a globally larger context.

8 Mutation Libraries and Community Training

Because mutations are modular artifacts with explicit specifications, they can in principle be trained asynchronously by different people, organizations, or automated systems. A public mutation library could contain adaptations specialized for long-context retrieval, navigation, game playing, code repair, scientific reading, tool use, verification, or domain-specific reasoning.

Such a library would be more than a repository of checkpoints. Each shared mutation should include metadata about its base-model compatibility, training sources, intended scope, runtime contract, output schema, validation results, known limitations, license, and expected resource cost. A local Mutations framework instance could then select and combine mutations as needed for a given task, subject to compatibility and trust constraints.

Community libraries also amortize expensive supervision. A larger open or proprietary model may be used to design, train, audit, or validate a mutation once, while many local deployments repeatedly use the resulting artifact without calling that model at runtime. This makes external supervision a shared construction cost rather than a recurring inference cost. For example, a community could use a strong model to generate edge cases for a code-repair mutation, audit a navigation mutation for unsafe actions, or compare several compression connectors before publishing the best validated artifact.

The same library structure supports a public-private split. Public mutation libraries can provide generic capabilities, while private mutations capture user-specific preferences, environments, documents, workflows, or immutable local sources. A user's private interaction traces can remain on the local device, where they are converted into documents, synthetic examples, and personal mutations without uploading raw data. This privacy boundary is especially important for personal agents, enterprise workflows, and any setting where experience traces may contain sensitive information.

Community-trained mutations would make the framework closer to an ecosystem than a single agent design. Users could compose a local system from a common base model and a small set of relevant mutations, while researchers could publish new mutations independently of the full application in which they were trained.

9 Evaluation Methodology

The framework should be evaluated in interactive settings where repeated experience can plausibly produce reusable adaptations. Static question answering is useful for isolated tests, but it does not expose the full lifecycle of traces, failures, mutation creation, routing, context packets, and later reuse. Good evaluation domains should therefore include sequential decisions, partial observability, repeated task families, measurable outcomes, and enough variation to test transfer rather than memorization.

Early controlled environments may include text-based games, TextWorld-style tasks, embodied text environments, small web-navigation tasks, software-engineering workflows, tool-use benchmarks, and emulator-like game environments. TextWorld and ALFWorld are especially relevant because they provide sandboxes for interactive text-game or text-conditioned embodied agents, controllable difficulty, state tracking, reward assignment, and sets of varied but related tasks for studying generalization and transfer [3, 36]. WebShop, Mind2Web, API-Bank, and SWE-bench provide complementary settings for web interaction, API use, and software-engineering repair [43, 4, 17, 14]. These domains make it possible to record observations, actions, invalid commands, state changes, and final outcomes. They also allow families of related tasks: different maps, goals, APIs, repositories, websites, or user preferences can be held out to measure whether a mutation captures a reusable procedure rather than one episode.

The central evaluation design should compare systems under fixed resource budgets. A mutation-based system should not be compared only to an unconstrained larger model or an unconstrained long-context agent. It should be evaluated under explicit limits on GPU or unified memory, context length, latency, active parameters, number of mutation activations, and total context-packet tokens. This matches the target deployment scenario and makes the claimed efficiency tradeoff testable.

Each experiment should distinguish three phases. First, a data-gathering phase records traces from the frozen base model, a baseline agent, or a supervised exploration policy. Second, an elaboration phase constructs episodic documents, synthetic examples, candidate mutations, routing labels, and validation tests. Third, a deployment phase freezes the produced artifacts and evaluates them on held-out tasks without further adaptation unless the experiment explicitly studies online updating.

9.1 Baselines

The minimum baseline is the frozen base model with the same tool interface and prompt budget as the mutation system. This measures what the base model can do without added episodic memory, procedural specialization, or mutation routing. A second baseline should use the same base model with summaries or retrieved documents but no model modifications, in order to test whether mutation gains are better explained by ordinary memory.

Retrieval baselines should be strong rather than symbolic placeholders. They may include raw trace retrieval, summarized episodic documents, vector retrieval over observations and outcomes, hybrid retrieval, retrieval with reranking, retrieval-augmented generation, retrieval-enhanced pretraining, and retrieval-heavy large-language-model variants [16, 7, 1, 40]. These baselines should be given comparable context-token budgets and should report how many retrieved tokens are actually inserted into the prompt.

Model-scale baselines are also important. A larger vanilla model with no mutations can show whether modular specialization offers a useful tradeoff against simply using more parameters. Where feasible, a larger model with retrieval should be included as a high-resource reference point,

while preserving a separate analysis under consumer-hardware constraints.

Mutation-specific baselines should include episodic-only systems, procedural-only systems, compression-only systems, routed combinations, and oracle routing. Oracle routing is not a deployable method, but it estimates the upper bound of a mutation library when the correct mutation is known in advance. Irrelevant-mutation and random-routing baselines are useful negative controls: a good router should avoid activating mutations that do not help.

9.2 Metrics

Primary task metrics depend on the environment: success rate, final score, reward, pass rate, solved subgoals, number of steps, and time to completion. For interactive agents, invalid action rate and recovery rate are especially important because many procedural mutations are expected to reduce repeated action failures rather than merely improve final answers.

Efficiency metrics should be reported alongside performance. These include prompt tokens, generated tokens, context-packet tokens, active mutation count, active parameter count, adapter or branch size, peak memory footprint, load or swap cost, latency per step, and total wall-clock time. When compression mutations are used, evaluations should report compression ratio, reconstruction or faithfulness error where measurable, and downstream performance after compression.

Routing metrics should measure whether the system activates useful mutations at the right time. Useful measures include activation precision, activation recall against oracle labels where available, marginal utility per activation, activation cost, conflict rate among packets, discarded-packet rate, and the fraction of decisions made with no mutation support. These metrics help distinguish a good mutation from a good router.

Reliability metrics should track hallucinated packet content, unsupported claims, stale episodic facts, contradictions between packets, regressions on tasks previously solved by the base model, and harmful overconfidence. Transfer should be measured by the gap between performance on tasks similar to the mutation training traces and performance on held-out tasks with new layouts, goals, users, APIs, or environment variants.

9.3 Ablations

Ablations should isolate the contribution of each part of the framework. Episodic-only variants test whether external state and retrieval are sufficient. Procedural-only variants test whether reusable action patterns improve behavior without episodic memory. Compression-only variants test whether token reduction preserves performance. Full systems test whether these components compose.

Routing ablations should compare learned routing, heuristic routing, random routing, oracle routing, and forced activation of all available mutations. Forced activation is useful because it reveals whether mutations are individually helpful but collectively harmful due to context bloat, latency, or contradictory packets. Routing-disabled variants can show whether mutation selection is doing real work or whether improvements come from simply adding more support to the prompt.

Budget ablations are necessary for the consumer-hardware claim. Experiments should sweep context-token limits, maximum active mutations, maximum context-packet tokens, memory budgets, quantization levels, and latency targets. A method that wins only when it can activate many large packets is less relevant to the proposed deployment setting than a method that remains useful under tight budgets.

Negative controls should include irrelevant mutations, stale episodic packets, intentionally over-broad procedural mutations, and mutations trained on related but mismatched domains. These

controls test whether the system can reject misleading support and whether validation catches mutations that appear plausible but degrade decisions.

10 Relationship to Existing Approaches

Mutations are related to retrieval-augmented generation, reflection, long-context agents, PEFT adapters, continual learning, tool use, mixture-of-experts systems, model editing, and agent memory architectures.

Compared with retrieval-augmented generation and model-based memory approaches such as MeMo [29], mutations shift some recurring experience from textual memory into bounded adaptive modules while preserving an explicit relation to a frozen base model. This does not eliminate retrieval. In fact, fast-changing episodic state may remain better represented as external documents or retrieval indexes. The framework instead asks which kinds of experience should remain non-parametric, which should be encoded in separate memory models, and which should be distilled into base-model-linked modifications.

Compared with ordinary PEFT, the framework treats an adapter, prefix, prompt vector, low-rank update, or representation intervention as only one part of a larger lifecycle [8, 18, 15, 9, 20, 41]. A mutation includes source filtering, synthetic data generation, runtime prompts, output schemas, validation rules, routing labels, and evaluation metrics. This makes the unit inspectable and replaceable rather than merely a checkpoint.

Compared with model editing methods such as MEND, ROME, and MEMIT, mutations emphasize bounded adaptation artifacts with explicit activation and validation rather than direct unconditional modification of a model's internal factual associations [25, 23, 24]. Knowledge editing is therefore an important neighboring literature, especially for understanding when a small targeted change should be embedded in model parameters and when it should remain an externally routed module.

Compared with agent memory and skill-library approaches, mutations ask which parts of reflected experience, self-correction, tool use, or procedural skill should remain as natural-language memory or code, and which should become base-model-linked adaptation artifacts [26, 35, 37]. This distinction is especially important for local agents that must balance inspectability, deletion, privacy, and inference cost.

Compared with larger vanilla models, mutations aim to improve the capability of smaller models through modular specialization. The relevant comparison is therefore not only raw task success, but the Pareto frontier of success, memory footprint, active parameters, context tokens, latency, and transfer.

Compared with model merging and adapter composition, mutations add a higher-level lifecycle around when adaptations should be created, routed, merged, retired, or exposed as reusable artifacts. Merging techniques may be used inside the framework, but the framework also accounts for provenance, validation, context-packet behavior, human feedback, privacy, and runtime arbitration.

11 Limitations and Risks

The flexibility of the framework introduces several risks. If mutations are created too easily, a system may accumulate many overlapping or low-value modules that increase routing complexity without improving task performance. If negative or suppressive mutations are too aggressive, they may block useful exploration or make the agent overly conservative. If pruning is too aggressive, the system may discard rare but important capabilities.

Privacy-preserving local adaptation also requires care. Keeping traces local reduces exposure, but trained mutations may still encode sensitive information from the data that produced them. Personal mutations should therefore be treated as potentially private artifacts, and shared mutations should include provenance, licensing, and privacy review where possible.

Human-in-the-loop adaptation introduces additional risks. Users may provide inconsistent feedback, optimize the system for narrow preferences that reduce generality, or accidentally reinforce unsafe behavior. The framework should preserve feedback provenance and support review, rollback, and evaluation of personalized mutations before they are trusted in higher-stakes settings.

12 Conclusion

This paper has proposed Mutations as a framework for studying modular adaptation in frozen language-model agents. The central claim is architectural rather than empirical: useful experience need not be represented only as longer prompts, larger retrieval stores, or monolithic updates to a base model. Some recurring patterns may be better treated as bounded adaptation units that remain tied to a frozen base model, expose explicit specifications, emit compact context packets, and can be routed, validated, composed, revised, or retired over time.

The framework is motivated by a practical constraint: capable local agents should be able to improve under fixed memory, context, and latency budgets on ordinary customer hardware. In this setting, the relevant comparison is not simply whether mutations outperform every retrieval system or every larger model. The more precise question is whether a small frozen model plus a managed library of routed adaptations can offer a better tradeoff among task success, active parameters, prompt length, latency, privacy, and controllability for repeated interactive tasks.

Several design principles follow from this view. The frozen base model provides a stable behavioral anchor. Mutation specifications make adaptations inspectable and comparable. Context packets prevent heterogeneous modules from flooding the base model with unbounded output. Routing and arbitration make activation a budgeted decision rather than a default. Lifecycle management, including compression, merging, suppression, pruning, and human feedback, treats adaptation as an ongoing ecology of modules rather than a one-time fine-tuning event.

The proposal also clarifies where empirical work is needed. Mutations should be tested against strong retrieval, memory, PEFT, larger-model, and routing baselines under matched resource budgets. Evaluation should separate episodic state tracking from procedural transfer, measure both performance and efficiency, and include negative controls for stale, irrelevant, or overbroad mutations. Only such experiments can determine when experience should remain external, when it should be distilled into a base-model-linked modification, and when the overhead of modularity outweighs its benefits.

If successful, the Mutations framework would provide a path toward local agents that become more capable through experience without continually expanding their base model or carrying ever larger histories in context. It would also support a broader ecosystem in which public mutations provide reusable capabilities, private mutations capture local preferences and environments, and larger offline supervisors help construct artifacts that cheaper deployment systems can use repeatedly. The long-term aim is not to replace retrieval, fine-tuning, or larger models, but to add a disciplined intermediate layer: modular, inspectable, resource-aware adaptation for frozen-model agents.

References

[1] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens, 2022.

[2] Alexandra Chronopoulou, Matthew E. Peters, Alexander Fraser, and Jesse Dodge. Adaptersoup: Weight averaging to improve generalization of pretrained language models, 2023.

[3] Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games, 2019.

[4] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web, 2023.

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[6] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022.

[7] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Realm: Retrieval-augmented language model pre-training, 2020.

[8] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR, 09–15 Jun 2019.

[9] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[10] Chengsong Huang, Qian Liu, Bill Yuchen Lin, Tianyu Pang, Chao Du, and Min Lin. Lorahub: Efficient cross-task generalization via dynamic lora composition, 2024.

[11] Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. Editing models with task arithmetic, 2023.

[12] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models, 2023.

[13] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression, 2024.

[14] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.

[15] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning, 2021.

[16] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[17] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms, 2023.

[18] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation, 2021.

[19] Yucheng Li, Bo Dong, Chenghua Lin, and Frank Guerin. Compressing context to enhance inference efficiency of large language models, 2023.

[20] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning, 2022.

[21] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023.

[22] Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning, 2018.

[23] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in gpt, 2023.

[24] Kevin Meng, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. Mass-editing memory in a transformer, 2023.

[25] Eric Mitchell, Charles Lin, Antoine Bosselut, Chelsea Finn, and Christopher D. Manning. Fast model editing at scale, 2022.

[26] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023.

[27] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis, 2023.

[28] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning, 2021.

[29] Ryan Wei Heng Quek, Sanghyuk Lee, Alfred Wei Lun Leong, Arun Verma, Alok Prakash, Nancy F. Chen, Bryan Kian Hsiang Low, Daniela Rus, and Armando Solar-Lezama. Memo: Memory as a model, 2026.

[30] Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling, 2019.

[31] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks, 2022.

[32] Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. Adapterdrop: On the efficiency of adapters in transformers, 2021.

[33] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

[34] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017.

[35] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

[36] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning, 2021.

[37] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023.

[38] Zifeng Wang, Zizhao Zhang, Chen-Yu Lee, Han Zhang, Ruoxi Sun, Xiaoqi Ren, Guolong Su, Vincent Perot, Jennifer Dy, and Tomas Pfister. Learning to prompt for continual learning, 2022.

[39] Mitchell Wortsman, Gabriel Ilharco, Samir Yitzhak Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S. Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, and Ludwig Schmidt. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time, 2022.

[40] Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers, 2022.

[41] Zhengxuan Wu, Aryaman Arora, Zheng Wang, Atticus Geiger, Dan Jurafsky, Christopher D. Manning, and Christopher Potts. Reft: Representation finetuning for language models, 2024.

[42] Prateek Yadav, Derek Tam, Leshem Choshen, Colin Raffel, and Mohit Bansal. Ties-merging: Resolving interference when merging models, 2023.

[43] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023.

[44] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.