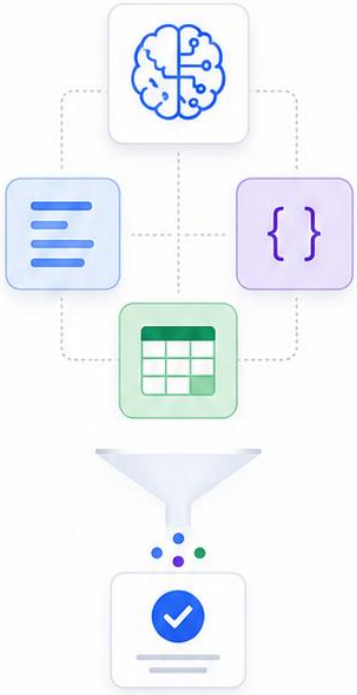
UsageTap

USAGETAP COMMUNITY WHITEPAPER

The LLM Cost Reduction Recipe Book



Practical patterns for reducing cost per successful task in AI applications

- Stop runaway workflows
- Select the right context
- Compact structured data
- Compress prose selectively

CASE STUDY

One developer. One test user.

$299.72 of application-testing spend in a day.

$1,427.86 projected over the next 7 days.

Cause: a few agentic workflows got stuck in loops.

usagetap.com

The LLM Cost Reduction Recipe Book

A practical, engineering-focused field guide for teams building AI features, agents, copilots, and LLM-powered applications. The goal is not to win a token-count contest. The goal is to complete useful work at the lowest sustainable cost while preserving quality, latency, reliability, and customer value.

COMMUNITY RESOURCE

This guide is intentionally broader than UsageTap's products. It describes practices that belong in application code, agent frameworks, model gateways, data systems, and evaluation pipelines. UsageTap is included where it can help implement or measure a recipe.

Who this is for

- Founders and engineering leaders seeing AI spend rise faster than product usage.
- Application developers building agents, assistants, RAG systems, or AI-enabled workflows.
- Platform teams responsible for model access, governance, observability, or cost control.
- Product and finance teams designing allowances, limits, pricing, and customer profitability.

How to use the book

Read the recipes in order the first time. The order matters: prevent unnecessary work before making individual calls cheaper. After that, use the recipe checklists as a diagnostic index. Each recipe includes the failure it addresses, implementation guidance, what to measure, common mistakes, and how UsageTap can help.

Important qualification

Savings vary by model, provider, workload, tokenizer, prompt shape, caching behavior, and quality requirements. The percentages and ratios in this guide are observations and practical ranges, not universal guarantees. Always validate changes against the downstream task.

CONTENTS

THE CENTRAL IDEA

The smallest prompt is not the goal. The cheapest successful task is.

Optimize cost per successful task

Token counts matter, but they are an incomplete proxy for application economics. The better metric is the total cost of producing an accepted result at the required quality and latency.

COST PER SUCCESSFUL TASK

Input + cached input + visible output + reasoning + tools + transformations + retries + fallbacks, divided by accepted tasks.

This framing changes the order of operations. A smaller prompt is not a win when it causes a retry. A cheaper model is not a win when it triggers an expensive escalation. A compressed context is not a win when it deletes a required fact. And a runaway workflow is not fixed simply because each loop costs less.

Four economic realities

INPUT-HEAVY APPS	HIDDEN WORK	BEST SAVING	CONTEXT
10-30x	**Reasoning**	**Avoid**	**Purpose**
Observed input-to-visible-output ratios in some context-heavy workloads	May substantially exceed the visible response	The cheapest model call is the one you do not make	Compression quality depends on the downstream task

Recommended order of operations

1. Bound financial exposure and runaway behavior.
2. Avoid calls that can be answered by cache, code, or existing data.
3. Define the purpose of the task and its acceptance criteria.
4. Select only the context the task can use.
5. Route the task to the least expensive model and reasoning level that reliably works.
6. Bound generated output, tool loops, retries, and agent duration.
7. Compact structured context before removing information.
8. Compress long redundant prose selectively and treat it as lossy.
9. Create reusable, purpose-specific context assets.
10. Verify downstream task quality and measure cost per accepted result.

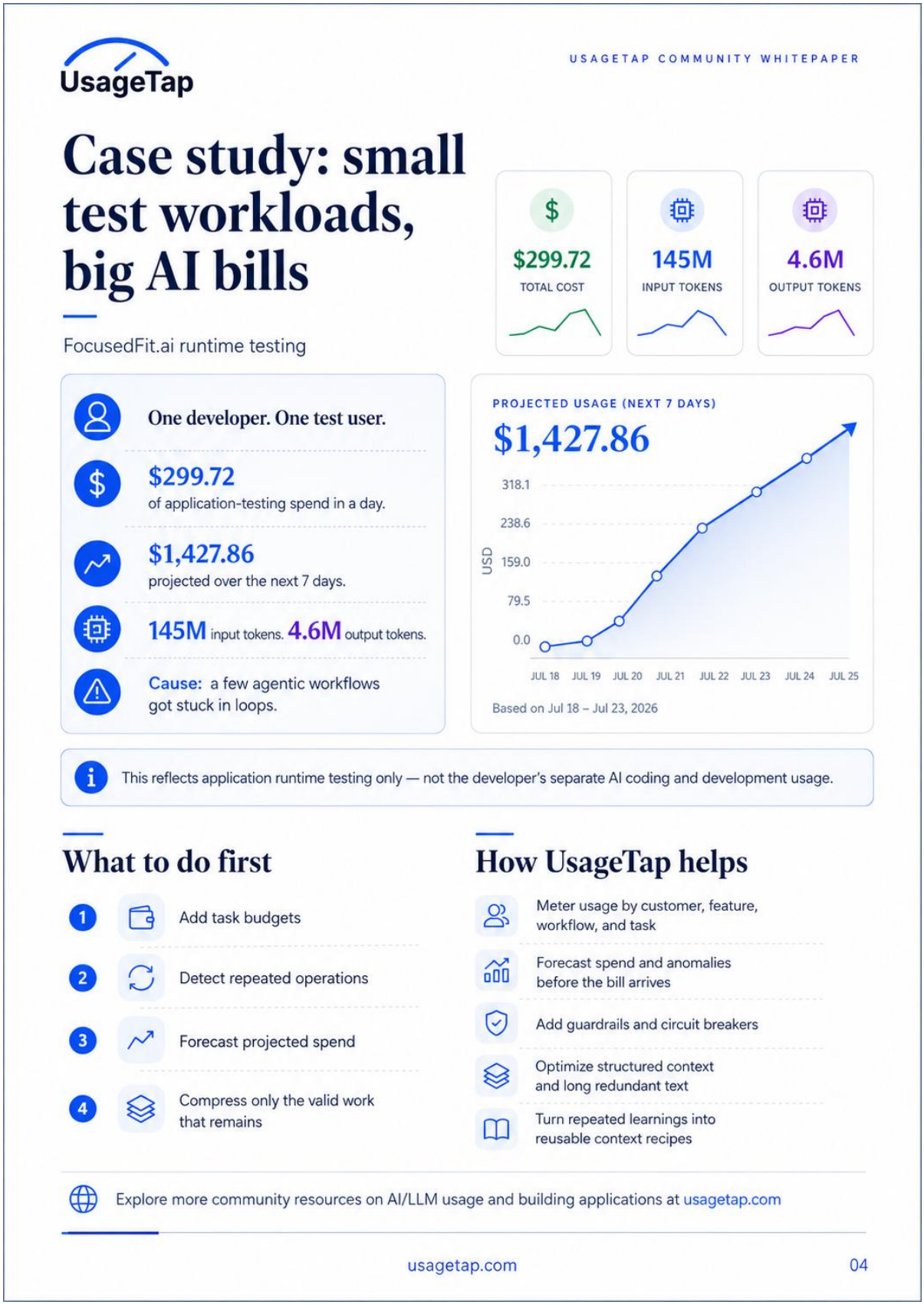


Figure 1. Cost per successful task and the practical order of operations.

Small test workloads can create big AI bills

The most expensive failure was not a high model price. It was unbounded application behavior.

APPLICATION TESTING	INPUT TOKENS	OUTPUT TOKENS	NEXT 7 DAYS
$299.72	145M	4.6M	$1,427.86
Spend observed in one day	Application runtime traffic	Visible output traffic	Forecast from recent behavior

An early-stage startup was testing its application with one developer and one test user. The application produced 145 million input tokens and 4.6 million output tokens. Daily application-testing spend reached $299.72, and the projected spend for the next seven days was $1,427.86.

WHAT THE NUMBERS DO NOT INCLUDE

This reflects application runtime testing only. It does not include the developer's separate use of AI coding assistants, development models, or other tools used to build the product.

The cause was not customer growth. A few agentic workflows became stuck in loops, repeatedly calling models and tools without recognizing that they were no longer making useful progress. The input-to-visible-output ratio was approximately 31.5 to 1, but prompt size was only part of the problem. The application was performing work that should not have existed.

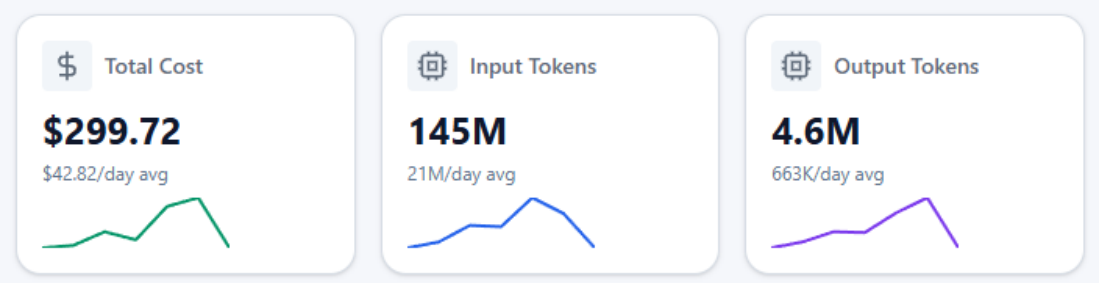


Figure 2. Application-testing usage summary.

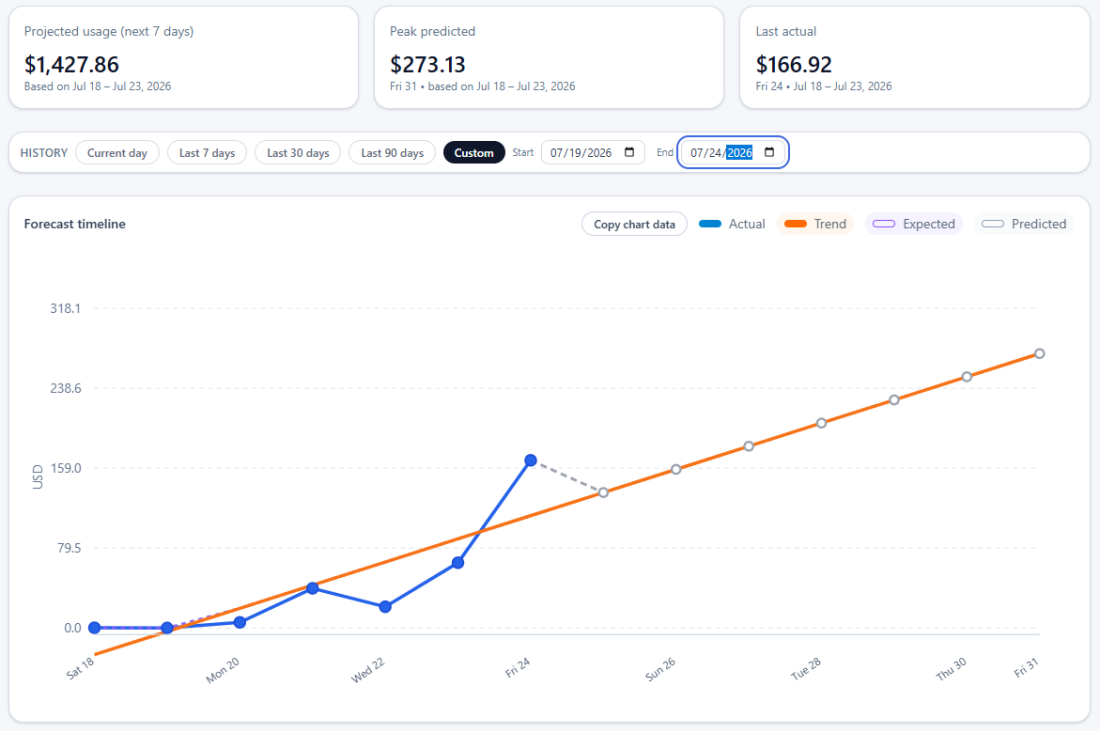


Figure 3. Seven-day spend projection based on recent application behavior.

The correct response

11. Stop the loops and add hard task-level limits.
12. Add workflow, tenant, and account-level spend guardrails.
13. Detect repeated operations and lack of progress.
14. Forecast near-term financial exposure and alert early.
15. Only then optimize the context and models used by valid calls.

LESSON

Before making every AI call cheaper, make sure every AI call should exist.

Figure 4. Ten recipes in the recommended order.

RECIPE 01

Put a budget around every task

THE RULE

Every AI task should have a maximum cost, duration, and amount of work.

Agentic and multi-call workflows are especially vulnerable to cost multiplication. A single task may resend prior context, generate hidden reasoning, call tools, retry failures, and branch into additional calls. Without explicit boundaries, a small coding mistake or confused agent can turn one user action into hundreds of paid operations.

What to budget

Boundary	What it protects against	Example action
Maximum task cost	Expensive models, huge contexts, retries, tool fees	Stop and return a partial result
Maximum model calls	Unbounded agent loops	Stop, escalate, or ask for clarification
Maximum tool calls	Repeated searches, writes, and external APIs	Block duplicate or low-value actions
Maximum elapsed time	Long-running or stalled workflows	Cancel and record a timeout reason
Maximum input tokens	Context growth across turns	Compact state or restart with a smaller context
Maximum generated tokens	Verbose output and hidden reasoning	Lower effort or enforce output limits
Maximum retries	Repeated failure without learning	Switch strategy or surface the failure
Repeated-operation limit	Same tool and arguments repeated	Trigger a circuit breaker

Implementation pattern

Create a task or run identifier before the first model call. Maintain local counters for spend, calls, tool actions, elapsed time, and tokens. Check the remaining budget before each expensive operation. The budget decision should be fast and resilient; for many applications, this means SDK-local enforcement with asynchronously reported usage.

A CALL LIMIT IS NOT A COST LIMIT

Five calls with a frontier reasoning model and a million-token context may cost more than fifty small classification calls. Use a dollar budget as the ultimate safety boundary.

Graceful stopping behavior

- Return the best validated partial result.
- Ask the user for missing information or permission to continue.
- Switch to a deterministic fallback.
- Escalate once to a more appropriate model rather than repeating the same failed strategy.
- Queue the task for asynchronous processing with an explicit budget.
- Record the stop reason so the workflow can be improved.

Measure

- Cost per task and cost per accepted task.
- Calls and tool actions per task.
- Percentage of tasks that hit each boundary.
- Spend prevented by circuit breakers.
- Completion and user-acceptance rates after graceful stopping.

Common mistakes

- Relying on the agent to decide when it is done.
- Using only a request-per-minute limit.
- Failing closed for every telemetry outage without considering application availability.
- Blocking a task without returning a useful user-facing explanation.

RECIPE 02

Avoid the call

THE RULE

The cheapest model call is the one the application does not make.

Many teams begin with model optimization because model calls are visible. The larger opportunity may be removing calls that should have been cache hits, calculations, lookups, or deterministic transformations.

Use this decision ladder

16. Is an accepted answer already cached for the same inputs and source version?
17. Can application code calculate or validate the result exactly?
18. Can a database query, search index, or rule engine return the answer?
19. Can a small model or embedding lookup solve a bounded subtask?
20. Only then call a larger generative model.

High-value avoidance patterns

Pattern	Best for	Primary risk
Exact response cache	Identical requests with stable data	Serving stale or cross-tenant results
Request coalescing	Concurrent identical work	Incorrect equivalence or delayed responses
Deterministic validation	Schemas, ranges, policy checks	Rules drift from product behavior
Precomputed artifacts	Catalogs, policy summaries, embeddings	Invalidation and versioning
Templates	Routine reports and user messages	Overfitting or loss of flexibility
Database projection	Known entities and fields	Missing interpretation the user actually needs

DO NOT USE AN LLM TO REDISCOVER WHAT THE APPLICATION ALREADY KNOWS

A model should add interpretation, synthesis, or generation. It should not be the default implementation for exact arithmetic, static lookup, or a rule the product already owns.

Cache identity

A safe cache key often needs more than the user prompt. Include tenant, source-data version, model or behavior version, tool availability, policy version, and any user-specific context that can change the answer. Record why a response is reusable and how it will be invalidated.

Measure

- Calls avoided and spend avoided.
- Cache hit rate by feature and customer.
- Invalidation frequency and stale-result incidents.
- Latency improvement from avoiding model calls.
- Acceptance rate of deterministic substitutes.

RECIPE 03

Define the purpose

THE RULE

Do not ask how to make context smaller until you know what the next task must preserve.

There is no universally correct compressed version of a document. The same source can require a different representation for support, extraction, compliance review, recommendation, planning, or classification.

Define an acceptance contract

- The task or question the model must answer.
- Required facts, entities, relationships, and exact values.
- Constraints, permissions, prohibitions, and exceptions.
- Required output schema and confidence behavior.
- Allowed latency and cost.
- What constitutes an accepted result.
- What the workflow should do when the evidence is insufficient.

Context Recipe

A Context Recipe is a versioned specification for preparing source material for a known downstream purpose. It can include selection, filtering, extraction, compaction, summarization, categorization, protection, caching, and validation.

Recipe field	Purpose
Purpose	Names the downstream task or family of compatible tasks
Inputs	Defines source types, schemas, and expected versions
Required content	Lists facts, relations, and constraints that must survive
Protected content	Defines terms, identifiers, numbers, code, or exact language
Allowed transforms	Selection, compaction, formatting, compression, summarization
Output contract	Target schema, representation, and token budget
Quality checks	Deterministic checks and task-level evaluation
Fallback	Original context, larger budget, stronger model, or human review
Cache policy	Source hash, recipe version, freshness, and invalidation

GENERAL SUMMARIES ARE OFTEN THE WRONG ASSET

A general summary preserves what appears broadly important. A task-specific representation preserves what the application will actually use.

Example

A customer record prepared for a support response may preserve active products, recent incidents, contract entitlements, and the current question. The same record prepared for churn classification may emphasize usage trends, support sentiment, renewal date, and account history. One universal summary is likely to be wrong for at least one of those tasks.

RECIPE 04

Send only relevant context

THE RULE

Context selection beats context compression.

The safest token is often the token never retrieved. Selection uses application knowledge to decide which records, fields, documents, turns, and tools the current task can use.

Selection opportunities

Source	Common waste	Better selection
Database records	All fields, all history, all related entities	Project only required fields and date range
RAG	Too many chunks or low-relevance neighbors	Use tighter retrieval, reranking, and diversity controls
Conversation	Entire chat replayed every turn	Retain recent turns plus verified state
Tools	Every tool definition exposed	Expose tools applicable to the current step
Tool results	Whole documents or raw API payloads	Return a task-specific projection
HTML pages	Navigation, chrome, scripts, repeated labels	Extract the relevant article or data region
Agent state	Plans, observations, errors, duplicate notes	Carry forward decisions and unresolved items

Which 50%?

A provocative observation is that a large share of application context may not affect the current task. The dangerous part is treating that as a universal percentage. The irrelevant portion changes with the question, user, workflow step, and acceptance criteria.

DO NOT COMPRESS DATA YOU SHOULD NEVER HAVE RETRIEVED

Selection is usually easier to reason about than lossy summarization because the application can explain why a record, field, or section is out of scope.

Ablation testing

When importance is uncertain, evaluate removal hypotheses against representative tasks. Remove one section or field family at a time, repeat the downstream task, and compare required facts, decisions, schema validity, and acceptance. Because responses are nondeterministic, use repeated runs or deterministic evaluators where appropriate.

Measure

- Tokens removed by selection before compression.
- Task accuracy and required-fact retention.
- Retrieval precision and useful-chunk rate.

RECIPE 05

Route to the right model

THE RULE

Use the least expensive model and reasoning level that reliably meets the task's acceptance criteria.

Not every task needs a frontier model, and not every task needs deep reasoning. Model routing is most effective when the task is well-defined, quality is measurable, and fallback behavior is included in the economics.

Good candidates for smaller models

- Classification against a stable taxonomy.
- Extraction into a fixed schema.
- Formatting, rewriting, and tone adjustment.
- Routine summarization with explicit required fields.
- Validation and policy checks with clear rules.
- Intent detection and straightforward tool selection.

Tasks that may justify stronger reasoning

- Ambiguous or conflicting evidence.
- Multi-step planning with dependencies.
- High-consequence trade-offs.
- Novel synthesis across many sources.
- Incomplete information requiring careful uncertainty handling.

CHEAP-FIRST CAN BE EXPENSIVE

A weak first attempt plus a retry plus an expensive fallback may cost more and take longer than starting with the appropriate model.

Routing evaluation

Metric	Why it matters
Accepted-task rate	A cheap model that fails often is not cheap
Escalation rate	Shows how often the first route was wrong
Total task latency	Captures retries and sequential cascades
Total task cost	Includes all attempts, not just the first
Error severity	Some failures are harmless; others are unacceptable
Customer segment	Different plans may justify different quality and latency tiers

Practical routing strategy

21. Group traffic into stable task families.
22. Define acceptance tests for each family.
23. Benchmark candidate models and reasoning levels.
24. Choose the lowest-cost route that meets the target reliability.
25. Define one explicit escalation path.
26. Measure the combined cost and latency of route and fallback.
27. Re-evaluate when prompts, tools, models, or source data change.

RECIPE 06

Bound reasoning, output, and agent work

THE RULE

Do not pay for deep thinking, verbose answers, or repeated tool work when the task does not need them.

The visible answer may be only a fraction of generated work. Reasoning tokens, tool calls, self-correction, structured-output overhead, and agent iterations can dominate cost.

Control generated work

- Choose an appropriate reasoning effort for the task.
- Set realistic output-token limits.
- Request a schema or concise fields instead of an essay.
- Do not ask for explanations the application will discard.
- Cap tool and agent iterations.
- Define a stopping condition before the workflow starts.
- Stop once the acceptance condition is met.
- Avoid repeated self-critique loops unless measured quality requires them.

Agent progress signals

Signal	Possible interpretation	Action
Same tool and equivalent arguments	Repeated work	Stop or ask what changed
Same error repeated	Retry without learning	Change strategy or surface failure
State hash unchanged	No progress	Trigger circuit breaker
Alternating actions	Two-state loop	Stop and escalate
Context grows, result score does not	Accumulation without value	Compact state or restart
Repeated retrieval of same source	Poor memory or planning	Cache result and mark consumed

PASS STATE FORWARD - DO NOT REPLAY THE ENTIRE JOURNEY

Long-running workflows should carry explicit state, decisions, evidence, and unresolved items. They should not depend on replaying every plan, tool call, and failed attempt.

Measure

- Visible output tokens and reasoning tokens by task.
- Tool calls and model calls per accepted task.
- Time and cost spent after the task was already answerable.
- Percentage of generated output consumed by the application.
- Loop and repeated-action incidents.

RECIPE 07

Compact structured context

THE RULE

Remove representation overhead before removing information.

Application prompts often contain valuable data represented inefficiently for a model: repeated JSON keys, XML tags, HTML wrappers, indentation, boilerplate, and large objects with irrelevant fields. This is often a structure problem before it is a language problem.

Three distinct transformations

Transformation	Definition	Risk level
Lossless minification	Same data and format, unnecessary whitespace removed	Low when parsed and validated
Deterministic value-preserving transformation	Representation changes while required values and relationships remain	Moderate; task dependent
Lossy compression	Words, spans, or details may be removed or summarized	Higher; needs evaluation

Safe order

28. Project away fields and records the task cannot use.
29. Parse and validate the source format.
30. Minify within the original format.
31. Apply a compact deterministic representation when eligible.
32. Verify values, ordering, relationships, and protected content.
33. Measure savings with the target tokenizer.
34. Return the original when the transform is unsafe or not worthwhile.

Examples

- Minify JSON and remove repeated indentation.
- Convert compatible arrays of similar objects into a tabular or compact record representation.
- Extract visible article content from HTML while removing navigation and page chrome.
- Normalize verbose tool outputs into fields the next step consumes.
- Replace repeated field names with a schema plus row values when relationships remain unambiguous.

BE PRECISE WITH THE WORD 'LOSSLESS'

Changing JSON into another representation may preserve all required values and relationships without being byte-for-byte reversible. Describe this as deterministic or value-preserving unless true round-trip equivalence is verified.

Where 30-50% can appear

Eligible JSON, HTML, XML-like, and repeated-record payloads can contain substantial representation overhead. In some observed workloads, deterministic compaction removed roughly 30-50% of input tokens. The result depends on schema shape, repeated keys, whitespace, source cleanliness, and tokenizer.

RECIPE 08

Compress prose selectively

THE RULE

Prompt compression is useful for long redundant language, but it is lossy and must be evaluated.

Extractive prompt-compression models estimate which tokens or spans are less important and remove them. This can work well on redundant natural language, but it is not equally safe for every content type.

Good candidates

- Long transcripts and meeting notes.
- Repetitive conversation histories.
- Large retrieved passages.
- Verbose documents and website text.
- Repeated explanations in tool output.
- Logs with substantial natural-language redundancy.

Poor or high-risk candidates

- Code and configuration.
- Exact quotations or legal language.
- Identifiers, account numbers, dates, money, and measurements.
- Compact schemas and short prompts.
- Negations, exceptions, permissions, and prohibitions.
- Exact-output instructions and format contracts.

Protect before compressing

Protection class	Examples
Identifiers	IDs, order numbers, ticket references, SKU values
Numbers	Dates, times, percentages, prices, quantities, thresholds
Constraints	must, must not, never, unless, only, at least
Code	Inline code, fenced blocks, expressions, commands
Entities	Customer names, product names, domain terminology
Verbatim spans	Quotations, contractual text, user-marked no-compress regions

A COMPRESSION RESULT IS A CANDIDATE UNTIL THE TASK PROVES IT WORKS

Text similarity is not enough. A compressed prompt can look semantically similar while changing a negation, entity-value relationship, exception, or numerical constraint.

Latency matters

Model-based compression adds compute and latency. It is most attractive when the compressed context will be reused, when the downstream call is expensive, when the source is very large, or when compression can run asynchronously. A deep compression step that saves a few hundred tokens but adds seconds to every inline request may worsen the product.

RECIPE 09

Transform once and reuse

THE RULE

Repeated context should not be repeatedly selected, summarized, compacted, or compressed.

If the same source supports the same recurring purpose, create a versioned context asset once and reuse it until the source, recipe, or quality requirements change.

Reusable context assets

- A product catalog prepared for recommendation.
- A policy set prepared for customer-support answers.
- A customer profile prepared for account planning.
- A rolling conversation state for a long-running assistant.
- Compact tool documentation for a known workflow.
- A document representation prepared for a specific extraction task.

Cache identity

A reusable representation should be keyed by tenant, source hash or version, recipe version, purpose, protected-content configuration, target representation, and tokenizer or model family where relevant.

Key component	Why it matters
Tenant	Prevents cross-customer leakage
Source version	Invalidates when data changes
Recipe version	Invalidates when transformation logic changes
Purpose	Prevents reuse for an incompatible task
Protection config	Captures tenant terminology and exact-value requirements
Target model/tokenizer	Allows choices to follow model economics
Freshness policy	Defines when the derived asset becomes stale

KEEP THE CANONICAL SOURCE

A compressed or summarized representation is normally a derived artifact. Keep the original when audit, reprocessing, legal, or alternative-task requirements demand it.

Complement provider prompt caching

Stable instructions and tool definitions should remain byte-stable when provider caching rewards repeated prefixes. Transform variable application context without needlessly changing the stable prefix. Context transformation and provider caching are complementary.

Measure

- Transformation cache hit rate.
- Repeated transformation cost avoided.
- Latency saved on future requests.
- Storage used by canonical and derived assets.
- Invalidation and stale-asset incidents.

RECIPE 10

Verify the task, not the text

THE RULE

A smaller prompt is successful only when the application still produces the right result.

Semantic similarity between original and transformed context is not a sufficient quality gate. The real question is whether the downstream application still completes the intended task correctly.

Compare four conditions

35. Unchanged baseline context.
36. Deterministic selection and compaction only.
37. Lossy compression only.
38. Combined pipeline.

Evaluation dimensions

Dimension	Examples
Task correctness	Classification accuracy, extracted fields, accepted answer
Required-fact retention	Entities, dates, quantities, obligations, exceptions
Constraint compliance	Permissions, prohibitions, safety rules, format rules
Citation and grounding	Correct source, quote, section, or evidence
Schema validity	Parseable and complete structured output
Operational behavior	Retries, fallbacks, tool loops, latency
Economics	Cost per accepted result, not tokens per call

TEXT SIMILARITY CAN HIDE CRITICAL DAMAGE

A one-word change can reverse a prohibition, remove an exception, or attach a value to the wrong entity while leaving the rest of the text almost unchanged.

Handling nondeterminism

- Use repeated runs for tasks with variable outputs.
- Set temperature and sampling controls consistently where available.
- Use deterministic validators for required facts and schemas.
- Use model-based judges only with calibration and human-reviewed examples.
- Compare distributions of quality, cost, and latency rather than a single run.
- Maintain held-out traffic not used to design the recipe.

Promotion criteria

A transformation should move from experiment to production only after it meets explicit quality thresholds, saves enough cost or latency to matter, and has a defined fallback. Continue monitoring because model, prompt, data, and traffic changes can invalidate earlier results.

Measure, diagnose, apply, validate, repeat

One-off optimization exercises decay quickly. Efficient AI applications need a continuous operating loop connecting usage telemetry, task outcomes, engineering changes, and financial impact.

Stage	Questions	Outputs
Measure	Where do calls, tokens, reasoning, tools, and retries accumulate?	Task-level cost and quality baseline
Diagnose	Is the waste from loops, retrieval, context, model choice, or output?	Ranked hypotheses
Apply	Which recipe addresses the cause with the lowest risk?	Versioned change or context recipe
Validate	Did quality, cost, and latency improve together?	Promotion or rollback decision
Repeat	What changed in traffic, models, prompts, or data?	Updated priorities and thresholds

Build a task taxonomy

Cost and quality become actionable when traffic is grouped into recurring task families such as extraction, support response, document review, research, recommendation, or agent planning. Task identity should travel through model calls, tool calls, retries, and transformations.

Minimum event context

- Tenant, customer, user, project, feature, workflow, and task identifiers.
- Provider, model, reasoning level, and request mode.
- Input, cached-input, output, and reasoning tokens where available.
- Tool calls, external charges, retries, and fallback attempts.
- Compression or transformation mode, latency, and tokens saved.
- Task outcome, acceptance, error class, and stop reason.

USAGE WITHOUT OUTCOME IS ONLY HALF THE DATA

A dashboard can show that tokens fell. Only task outcomes can show whether the application became more efficient.

Rank opportunities by economic impact

A 60% reduction on a rare low-cost task may matter less than a 10% reduction on a high-volume workflow. Prioritize by total spend, projected growth, customer margin, quality risk, and implementation effort.

A staged path from visibility to adaptive context operations

Phase 1 - Make task economics visible

- Assign task and workflow identifiers.
- Measure full cost across calls, reasoning, tools, retries, and fallbacks.
- Define accepted-task outcomes.
- Add spend forecasts and anomaly alerts.

Phase 2 - Add hard safety boundaries

- Implement local task budgets and circuit breakers.
- Set tenant, feature, and workflow limits.
- Detect repeated operations and lack of progress.
- Return graceful partial results or explicit failures.

Phase 3 - Remove obvious waste

- Cache exact repeat work.
- Replace deterministic model calls with code or queries.
- Select smaller record, field, document, and tool sets.
- Bound output and reasoning for routine tasks.

Phase 4 - Optimize context

- Apply deterministic structured compaction.
- Protect values, code, constraints, and tenant terminology.
- Use lossy prose compression only on eligible content.
- Create reusable purpose-specific context assets.

Phase 5 - Profile and automate

- Sample representative production traffic.
- Classify context segments by purpose and sensitivity.
- Run ablation and downstream evaluation.
- Promote proven findings into Context Recipes.
- Monitor drift and roll back automatically when quality changes.

START WITH CONTROL, NOT COMPLEXITY

A task ID, cost budget, duplicate-action detector, and a few selection rules can prevent more spend than a sophisticated compression model applied to an uncontrolled workflow.

Production checklist

Task economics

☐ Every model and tool call carries a task or run identifier.

☐ Cost includes retries, fallbacks, reasoning, and external tools.

☐ Task acceptance or outcome is recorded.

☐ Forecasts and anomaly alerts exist for critical tenants and workflows.

Guardrails

☐ Each agent task has call, time, token, retry, and dollar limits.

☐ Repeated operations can trigger a circuit breaker.

☐ Failure behavior is explicit and user-friendly.

☐ Fail-open and fail-closed behavior is configured intentionally.

Context

☐ Database fields and document sections are selected before compression.

☐ Stable prompt prefixes are preserved for caching where appropriate.

☐ Structured transforms are parsed, validated, and gated.

☐ Lossy compression protects critical values and exact spans.

Evaluation

☐ Baseline and transformed variants are compared.

☐ Required facts and constraints have deterministic checks.

☐ Cost per accepted task is reported.

☐ Recipes have versions, thresholds, fallback, and rollback.

How UsageTap Helps

Put a budget around every task: UsageTap can meter active tasks, forecast near-term spend, apply tenant and workflow policies, and signal when a task should warn, throttle, downgrade, or stop. The application still owns the graceful exit and the semantic definition of progress.

Define the purpose of the task and its acceptance criteria. UsageTap can meter active tasks, forecast near-term spend, apply tenant and workflow policies, and signal when a task should warn, throttle, downgrade, or stop. The application still owns the graceful exit and the semantic definition of progress.

Select only the context the task can use.: UsageTap can profile recurring context segments, attribute their cost, compare downstream outcomes with and without them, and recommend fields or sections that may be safe to omit for a defined task.

Route the task to the least expensive model and reasoning level that reliably works.: UsageTap Meter can attribute cost and outcomes by task, model, provider, customer, and feature. UsageTap Compress may help a smaller model succeed by presenting denser, more relevant context, but compression should not be treated as an automatic model-downgrade guarantee.

Bound generated output, tool loops, retries, and agent duration.: UsageTap can meter reasoning where providers expose it, count tool and model iterations, detect abnormal task trajectories, forecast spend, and apply task or workflow guardrails.

Compress long redundant prose selectively and treat it as lossy.: UsageTap Compress segments input, protects critical content, skips poor candidates, applies model compression only when eligible, reports what changed, and can return the original when thresholds fail.

Create reusable, purpose-specific context assets.: UsageTap Compress can cache content-addressed transformations, version outputs by recipe, report reuse, and preserve stable prompt sections while optimizing variable context.

Verify downstream task quality and measure cost per accepted result.: UsageTap can record baselines, transformed variants, protected-content checks, downstream outcomes, cost, and fallback decisions. The long-term opportunity is to promote proven traffic-specific findings into reusable Context Recipes.

Example task budget policy

```
{
  "policy": "standard-agent-task",
  "limits": {
    "max_cost_usd": 1.50,
    "max_model_calls": 15,
    "max_tool_calls": 25,
    "max_elapsed_seconds": 240,
    "max_input_tokens": 300000,
    "max_output_tokens": 30000,
    "max_retries_per_step": 2,
    "max_repeated_operation": 2
  },
  "actions": {
    "at_75_percent": "warn",
    "at_90_percent": "notify",
    "at_100_percent": "block"
  },
  "fallback": "return_best_validated_partial_result"
}
```

Policy design notes

- The values are examples, not defaults for every workload.
- The application should estimate the next operation before allowing it.
- The workflow must define a useful stop or fallback behavior.
- Strict prepaid workloads may fail closed; ordinary observability may fail open.
- Policy changes should be versioned and auditable.

Example usage event

```
{
  "tenant_id": "tenant_123",
  "customer_id": "cust_456",
  "feature": "research-agent",
  "workflow_id": "workflow_789",
  "task_id": "task_abc",
  "provider": "provider-name",
  "model": "model-name",
  "input_tokens": 18420,
  "output_tokens": 620,
  "reasoning_tokens": 4100,
  "tool_calls": 3,
  "estimated_cost_usd": 0.42,
  "compression": {
    "mode": "deterministic_first",
    "original_tokens": 24200,
    "result_tokens": 18420,
    "saved_tokens": 5780
  },
  "outcome": "accepted",
  "stop_reason": "completed"
}
```

Context Recipe template

Field	Questions to answer
Name and version	What is this recipe called, and what changed?
Purpose	Which downstream task or task family is it for?
Inputs	Which source types, schemas, and roles are accepted?
Selection	Which records, fields, turns, tools, or sections are in scope?
Protected content	Which exact values, words, spans, and structures must survive?
Transformations	Which deterministic and lossy steps are allowed?
Token and latency budget	How much context and transformation time are acceptable?
Output contract	What representation or schema is produced?
Validation	Which deterministic checks and task evaluations must pass?
Fallback	When is the original used, or the task escalated?
Cache identity	Which source, tenant, purpose, and version fields define reuse?
Monitoring	Which metrics trigger review or rollback?

Promotion checklist

- Representative traffic sample reviewed.
- Required facts and protected spans defined.
- Baseline results captured.
- Deterministic-only variant evaluated.
- Lossy variant evaluated separately.
- Combined pipeline evaluated.
- Cost, quality, and latency thresholds met.
- Fallback and rollback tested.
- Recipe version and owner recorded.

Research foundations

Jiang et al. (2023). LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models.

Introduced a prompt-compression framework aimed at reducing prompt length while preserving model performance.

Pan et al. (2024). LLMLingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression.

Frames prompt compression as token classification and provides an efficient extractive approach.

Jiang et al. (2023). LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression.

Explores question-aware compression for long-context tasks.

Liu et al. (2023). Lost in the Middle: How Language Models Use Long Contexts.

Shows that relevant information placement within long contexts can affect model use and retrieval.

Practical principle

SEND THE RIGHT CONTEXT - NOT MERELY LESS CONTEXT

Efficient AI applications combine operational control, task-aware selection, appropriate model effort, deterministic compaction, selective lossy compression, reuse, and downstream verification.

About UsageTap

UsageTap helps AI application teams meter usage by customer and feature, forecast spend, add guardrails, and optimize context. UsageTap Compress applies deterministic-first context optimization, selective text compression, protected content, inspection, and fallback. The broader goal is to make AI usage predictable, controlled, and profitable.

Community resources

This whitepaper is part of an ongoing UsageTap series on AI/LLM application economics, context engineering, usage controls, forecasting, and sustainable product design. Visit usagetap.com for updates, practical tools, and additional field notes.