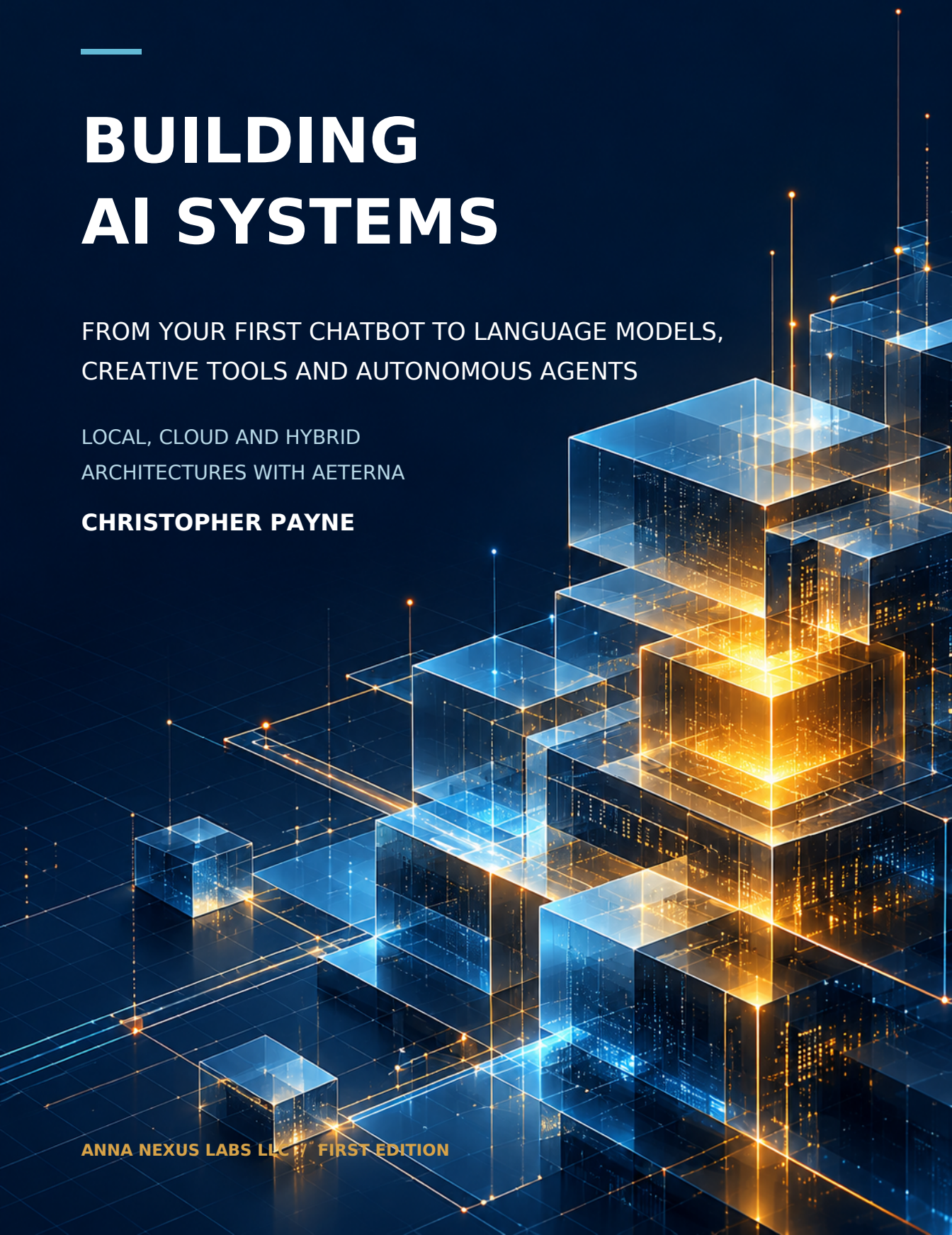
BUILDING AI SYSTEMS

FROM YOUR FIRST CHATBOT TO LANGUAGE MODELS, CREATIVE TOOLS AND AUTONOMOUS AGENTS

LOCAL, CLOUD AND HYBRID ARCHITECTURES WITH AETERNA

CHRISTOPHER PAYNE



ANNA NEXUS LABS LLC IV · FIRST EDITION

BUILDING AI SYSTEMS

From Your First Chatbot to Language Models, Creative Tools and Autonomous Agents

Local, Cloud and Hybrid Architectures with AETERNA

Christopher Payne

Anna Nexus Labs LLC

First edition | October 1 2026

Prepared with AI assistance. Technical status and verification scope are described in the edition note and Appendix F.

CONTENTS

Part I Foundations of AI Systems **8**

Chapter 1 What You Are Actually Building 10

Chapter 2 The Main Families of AI 16

Chapter 3 Data and Measurement 18

Chapter 4 Neural Networks Without Mystery 20

Chapter 5 Tokens Embeddings and Attention 22

Chapter 6 Evaluating a System 24

Part II Chatbots and Assistants **26**

Chapter 7 Your First Chatbot 27

Chapter 8 Instructions and Structured Responses 29

Chapter 9 Conversation State and Identity 31

Chapter 10 Retrieval from Documents 33

Chapter 11 Memory That Can Be Corrected 35

Chapter 12 Voice and Multimodal Assistants 37

Part III Local Cloud and Hybrid Deployment **39**

Chapter 13 Models Providers and Runtimes 40

Chapter 14 Running Models Locally 42

Chapter 15 Building with Hosted Services 44

Chapter 16 Hardware and Performance 46

Chapter 17 Hybrid Routing 48

Chapter 18 Packaging and Deployment 50

Part IV Creative and Multimodal Systems **52**

Chapter 19 Text Creation Workflows 53

Chapter 20 Image Generation and Editing 55

Chapter 21 Speech and Sound 57

Chapter 22 Music Generation and Production 59

Chapter 23 Video and Movie Production 61

Chapter 24 A Unified Creative Workspace 63

Part V Agents and Task Automation **65**

Chapter 25 Tools as Typed Capabilities 66

Chapter 26 A Bounded Task Agent 68

Chapter 27 Browser Files and Application Automation 70

Chapter 28 Multi Agent Coordination 72

Chapter 29 Coding Agents and Sandboxes 74

Chapter 30 Automation Under Failure 76

Part VI Building and Training Models **78**

Chapter 31 Specify the Model and Its Data 79

Chapter 32 Build a Small Language Model 81

Chapter 33 Train Save and Resume 83

Chapter 34 Generate and Serve 85

Chapter 35 Fine Tuning and Efficient Adaptation 87

Chapter 36 Scaling and Other Training Families 89

Part VII AETERNA Evidence Authority and Memory **91**

Chapter 37 Evidence and Interpretation 92

Chapter 38 VERITAS and DISCERNMENT 94

Chapter 39 AETHER and SENTINEL 96

Chapter 40 MNEMOSYNE and the Golden Thread 98

Chapter 41 PHOENIX and the Observatory 100

Chapter 42 Deterministic Disposition and Acceptance 102

Part VIII Integrated Systems and Release **104**

Chapter 43 An Evidence Grounded Research Assistant 105

Chapter 44 A Governed Creative Studio 107

Chapter 45 A Permissioned Operations Bot 109

Chapter 46 A Local and Hybrid Personal Assistant 111

Chapter 47 Reliability Security and Observability 113

Chapter 48 Releasing and Maintaining Your System 115

Appendices **117**

Appendix A Working Vocabulary 118

Appendix B Set Up and Run the Companion 121

Appendix C Provider and Media Recipes 124

Appendix D Hardware and Performance Worksheet 129

Appendix E Reusable Records and Acceptance Templates 131

Appendix F Projects Sources and Verification 134

Appendix G Reading the Core Algorithms 139

Edition note

This is a complete, compact technical first edition with 48 chapters, seven appendices, and a separate executable companion. It is intended for study and experimentation. It covers representative AI system families rather than every technique or deployed product. The model built here is a real small language model, with restricted synthetic data and limited capabilities.

The text and code were prepared with AI assistance for Christopher Payne. The recorded checks were run in the included environment. AETERNA in this volume names the architecture and teaching reference components; these examples are not an audit of any separate AETERNA repository, production service, or external organization's acceptance suite.

Product documentation changes. Provider recipes were reviewed against official documentation on October 1 2026 but were not run with live credentials. Verify current model access, cost, interfaces, licenses, privacy terms, and suitability before use. No vendor endorses this volume.

About this book

This book teaches how to build and assess AI systems from a local chatbot through retrieval, memory, creative workflows, task agents, and a small transformer trained from scratch. The surrounding application matters as much as the model: data origins, permissions, feedback, failure recovery, and evidence remain visible throughout.

Each chapter gives a concrete engineering path and an evaluation exercise. The companion contains runnable local examples, actual saved model weights and experiment outputs, and boundary tests. The appendices provide setup instructions, current provider recipes, estimation worksheets, record templates, project statuses, and annotated core algorithms.

You can read Part I from the beginning, follow the chatbot path in Part II, or study the training path in Part VI. For a larger system, use Parts V through VIII to join capabilities with permission and evidence boundaries. The source notes identify external technical material; the verification record identifies what was executed here.

How claims are labeled

Concept explains a method or design. **Teaching implementation** refers to included local code. **Checked result** refers to a retained run or test output with its scope and environment. **Recipe** gives an interface and steps to implement, without claiming live execution. **Independent validation** requires evaluation outside this book's own development and is not claimed for AETERNA or the capstones.

An architectural rule such as “model output is a proposal” constrains the intended design. Its enforcement must be demonstrated at every relevant execution path. A declining training loss is evidence for that measured objective on that data; it is not evidence of general intelligence or trustworthy factual answers.

The learning path

Part	Chapters	Outcome
I Foundations	1-6	Define tasks, data, models, measurements
II Chatbots	7-12	Conversation, retrieval, memory, voice designs
III Deployment	13-18	Local, hosted and hybrid choices
IV Creative systems	19-24	Text, images, sound, music and movies
V Agents	25-30	Tools, task loops, automation and recovery
VI Model building	31-36	Train, save, serve and adapt a small LM
VII AETERNA	37-42	Evidence, authority, memory, audit and disposition
VIII Integration	43-48	Capstones, reliability and release

Part I Foundations of AI Systems

A practical path through the components and their evidence.

Introduction

You can begin building an AI application before you understand every detail of the model inside it. That is useful. It also creates a responsibility: you must know which parts of the system you understand, which parts you depend on, and which claims you have actually checked.

We will learn how models produce outputs and how applications use them. Each project adds a capability, explains its dependencies, and checks its behavior before the next addition.

Consider a tool that drafts a message. Its first version needs an input field, a model request, and a place to display the result. Add the ability to send the message, and the engineering problem changes. The application now needs a verified destination, an authorized account, a decision about approval, and a reliable record of whether the send happened. A fluent draft does not answer any of those questions.

The same distinction appears in creative work. A generated image needs an asset record and a way to review it. A song involves audio formats, arrangement, versioning, and decisions about its use. A movie requires coordination across shots, sound, editing, and export. Models contribute to these workflows, but the workflow remains an engineering system with dependencies and failure cases.

We will also move inside the model. You will learn how training data becomes numerical input, how a neural network computes an output, and how an optimization procedure changes its parameters. Later, you will construct a small language model, train it, save its state, and serve it. The goal is a complete, understandable training lifecycle. The size and quality of that model will be measured rather than assumed.

Local deployment, hosted services, and hybrid designs each give you different control over data movement, hardware, availability, and operating costs. We will make those choices visible. A routing decision should respect the task's permission boundary even when a preferred service is unavailable. A system designed to keep a document local must preserve that restriction through its error paths.

AETERNA assigns distinct responsibilities to evidence, interpretation, authority, execution checks, memory, and audit. We will develop teaching implementations and identify the evidence needed to assess them. Interfaces and observable behavior determine whether each implementation does its job.

You do not have to accept an architectural claim because the prose sounds certain. Ask what the claim means, what observation would support it, and what result would count against it. If a gate is supposed to deny an action without a valid grant, the useful demonstration includes an invalid grant and a check that no action occurred. If an assistant cites a source, inspect whether the source supports the sentence.

Begin with one task. Give it a measurable outcome. Identify the data and resources it may use. Then draw the boundary between the model's proposed answer and the software's permitted behavior. Chapter 1 develops that first specification.

Chapter 1 What You Are Actually Building

The first distinction

An AI model is a component that computes outputs from inputs using a learned set of parameters. An AI system includes the model and the software, data, interfaces, rules, and operating environment around it. You build the system by deciding how those pieces interact.

Imagine an assistant that answers questions about a handbook. The user types a question. The application finds relevant handbook passages, sends a selected context to a language model, and displays a response with source references. Several pieces are doing different jobs. The search process selects material. The model composes the answer. The interface displays it. A source checker or reviewer assesses whether the cited material actually supports the claims.

Replacing the model can change the response. Replacing the retrieval process can change which passages the model sees. Replacing a policy can change which documents a particular user may access. These are separate changes, and each needs its own evaluation.

This distinction will guide every project in the book. When a result is wrong, we need to locate the failure. It could be a model error, a stale document, a bad index, an access-control mistake, a malformed tool argument, or an interface that hid an important limitation.

A request through the system

Suppose a user asks, “Draft a one-paragraph summary of our equipment policy and save it in my project folder.” This request contains at least two tasks: produce a summary and create a file. The system must also establish which policy, which project folder, and which user authority apply.

The interface receives the request and associates it with the current session. The application validates the request and checks which sources the user can read. The retrieval component locates the policy and preserves its version and source location. The model receives the permitted context and proposes a summary.

Before a write, the application resolves the target path. A policy component checks whether this actor can create that file in that folder. The execution component writes only within the granted scope. The system then records the outcome and shows the resulting file or a clear failure message.

If the folder is ambiguous, the system has an unresolved destination. If the policy is missing, it has missing evidence. If the write is denied, it has a permission failure. These conditions should remain distinct. A model can suggest a clarifying question; it cannot turn uncertainty into a valid destination by sounding confident.

Even this small request needs a model, data access, policy, execution, and feedback. Keeping the responsibilities visible makes it possible to diagnose and constrain the workflow.

Six components to identify

For the first specification, identify six components. This is a teaching inventory, not a claim that all systems have exactly six modules. A practical implementation may combine some responsibilities and divide others.

Component	Responsibility	Question to answer
Model	Computes predictions or generated outputs	What outputs can it produce, and what has been measured?
Runtime	Loads or accesses the model and executes inference	Where does computation occur, and what resources does it need?
Application	Coordinates requests, context, tools, and responses	How does a request move through the workflow?
Data	Supplies task inputs, documents, examples, and stored records	Where did each item originate, and who may access it?
Policy	Establishes permitted behavior and required conditions	Which actor may perform which action on which resource?
Operating environment	Provides hardware, network, storage, and process boundaries	What happens when a dependency fails or access changes?

The model and runtime distinction is useful when discussing local deployment. Model weights are an artifact. A runtime is software that uses them for inference. A hosted endpoint exposes model computation through a service. Your application can depend on a runtime or endpoint without possessing the model weights. Ollama is one documented option for accessing local and hosted models [4].

The data component includes more than training data. A handbook retrieved during a conversation, a user preference stored last week, and the arguments passed to a file tool are all data. They have different origins and deserve different treatment. A remembered preference can guide formatting; it does not necessarily authorize a new action.

Policy may include ordinary software checks: an allowed directory, an account role, an action budget, or a grant that expires. These checks should inspect explicit

records. Asking a model to say whether an action is allowed leaves the decision dependent on another generated answer.

What a language model does

In the decoder-only language model developed later in this book, text is represented as tokens. A token can correspond to part of a word, a word, punctuation, or another unit defined by the tokenizer. The model takes the preceding token sequence and computes scores for possible next tokens. A decoding procedure selects a token, adds it to the sequence, and repeats until a stopping condition is met.

That procedure can produce useful explanations and code. It can also produce a false statement, an unsupported citation, or an instruction that violates the application's rules. Generating a plausible continuation is a different operation from checking a source or granting access.

Attention allows a transformer to compute relationships among representations at different positions in a sequence. The original Transformer paper introduced an architecture centered on attention rather than recurrent or convolutional sequence processing [1]. Chapter 5 develops the mechanics. Here, the architectural consequence is enough: the model processes a selected input context, and the application is responsible for selecting and protecting that context.

The name “large language model” describes a class of models without setting a single universal size threshold. Our training project will use a small language model so its construction and training can be inspected. Its educational completeness concerns the lifecycle, from data to inference. Its capabilities must be assessed on their own results.

Training and inference

Training changes learned parameters according to an objective and data. In ordinary gradient-based neural-network training, the model computes a prediction, a loss measures the relevant error, and an optimizer updates parameters using gradients. PyTorch's optimization tutorial demonstrates the relationship among loss, backpropagation, and parameter updates [2].

Inference uses a trained model to compute an output. In the projects here, an ordinary inference request does not update the model weights. An application may store a conversation or a separate memory record, and a provider may have its own data policies. Those storage and policy questions are separate from whether the particular inference computation changes parameters.

This distinction matters when you say that an assistant “learned” your preferred writing style. The application may simply have retrieved a preference and inserted it into the next request. That is useful adaptation at the application level. It does not establish that a model was retrained.

Fine-tuning continues training from existing weights for a chosen objective or dataset. Retrieval supplies selected material at use time. The original retrieval-augmented generation research combined a generator with access to a non-parametric document index [3]. Our document assistant will use a practical retrieve-then-generate workflow; it does not claim to reproduce that paper's entire training method.

Choose among these methods by identifying the problem. A missing current handbook may call for retrieval. A consistent output behavior may justify an adaptation study. A need to understand model construction justifies training a small model. Each choice introduces its own data, evaluation, and resource requirements.

The same separation across media

A creative system has the same surrounding responsibilities even when its model produces a picture or audio. An image application must store the supplied references, generation request, output, and relevant version information. A music application must handle an audio asset and its production workflow. A film application coordinates many assets on a timeline.

The underlying model families and learning objectives differ. Later chapters explain representative approaches rather than treating all media generation as text completion. The application-level questions remain concrete: what is the requested output, where will it be stored, what quality checks apply, and which actions require permission?

For a short film, success is not established by one attractive frame. The export must contain the intended shots and audio, the timeline must make sense, and a review must record continuity defects. A generation job and a finished production are different milestones that can be measured separately.

Where AETERNA enters

AETERNA organizes the boundaries between evidence, interpretation, authority, and execution. In this book, VERITAS represents evidence handling and provenance. DISCERNMENT represents interpretation. AETHER represents authority, while SENTINEL represents capability checks at the execution boundary. MNEMOSYNE handles memory, PHOENIX handles audit records, and the Observatory presents records for inspection. The Golden Thread supplies an explicit continuity structure. These are architectural responsibilities to implement and evaluate.

Return to the handbook request. The retained handbook passage belongs to the evidence record. The summary belongs to interpretation. A grant to write into a project folder belongs to authority. The write check belongs at the capability boundary. A later memory saying “the summary was saved” should link to the recorded outcome; it should not become a second independent witness to the same event.

A source hash can help detect a change to retained bytes. It cannot establish that the handbook is accurate. An audit entry can record an execution result. Its usefulness depends on the logger, event coverage, and storage controls. An authority check can constrain an action only if every relevant execution path actually passes through it.

These limits guide the later acceptance work. We will define what a component promises and then test the boundary where that promise could fail. A component name alone provides no implementation evidence.

Your first system specification

Before installing a model or writing an API call, specify one task. Use the handbook assistant as a worked example. Its purpose is to answer questions about a designated handbook and save a draft summary to an authorized folder. Its initial users are explicitly enrolled project members.

Its permitted inputs are a user request, the designated handbook version, and an authorized project identifier. Its outputs are a source-linked answer, an unresolved response, or a confirmed file outcome. Its tools are handbook retrieval and a file writer restricted to the project folder. Its authority comes from application records associated with the actor and project.

Success requires more than a fluent answer. A reviewer must be able to find the cited passage and assess its support. A request outside the folder must be denied. A missing handbook must not produce an invented quotation. A failed write must not be reported as saved. Define these checks before judging the interface.

Now write the same specification for your own first system:

1. State the user's task in one sentence.
2. List the exact inputs and where they come from.
3. Identify acceptable outputs, including unresolved and failed outcomes.
4. Identify the model and runtime dependency without assuming a specific vendor.
5. List readable and writable resources separately.
6. Identify who grants permission and what scope the grant covers.
7. Describe at least one quality check and one denied-action check.
8. Specify what users see when a dependency fails.

The result is a one-page design you can revise. It will make later choices easier: whether retrieval is necessary, whether a local model is sufficient, whether a tool is justified, and whether the task requires a more restricted execution environment.

Practice and review

Choose either a document assistant, a music drafting workspace, or a bot that organizes a permitted folder. Write its specification using the eight steps above. Keep the first version narrow enough that one successful action and one refused action are easy to inspect.

Then trace a single request. Identify the input, model proposal, required permission, execution result, and evidence available to support the result. If your example has no state-changing action, explicitly say that its output is a draft and identify what a later publication or file write would require.

Explain in your own words why storing a preference differs from updating model weights. Explain what a source citation can establish after it has been inspected. Finally, identify one place where an unresolved answer is more accurate than a guessed result.

Chapter 2 broadens the model choices. You will learn when a classifier, search method, recommendation model, or other technique fits a problem, so a language model becomes a deliberate choice within the architecture.

Chapter 2 The Main Families of AI

Begin with the output you need

AI covers several kinds of computational work. A classifier assigns a category, such as a support ticket type. A regression model estimates a number, such as tomorrow's demand. A clustering method groups examples according to a chosen representation and distance. A recommender ranks candidates for a user or situation. A generative model produces a new sequence or artifact. These tasks can share underlying techniques, but they have different success criteria.

Computer vision and speech describe input and output domains rather than a single learning method. A vision system might classify an image, locate objects, segment regions, or generate an image. A speech system might detect activity, transcribe words, synthesize speech, or estimate speaker similarity. A useful specification names both the domain and the task.

Supervised learning uses examples with target outputs. Unsupervised methods seek structure without the same kind of target labels. Self-supervised learning constructs a training signal from the data itself, such as predicting missing or subsequent content. Reinforcement learning trains behavior through interaction and reward. The scikit-learn documentation organizes many conventional estimators by learning task [5]. Deep Q-learning is a research example of learning action values from rewards and observations [6].

Choosing a method

For a ticket sorter, first consider a classifier with text features and labeled examples. For a question about a handbook, retrieval and a language model may be appropriate. For exact arithmetic, use arithmetic. For a rule that denies access outside a folder, use a deterministic policy check. Combining these methods often gives a clearer system than assigning every operation to one model.

Needed output	Candidate approach	Useful evaluation
A category	Classification	Per-class precision and recall
A numeric estimate	Regression	Error magnitude and bias
A ranking	Retrieval or recommendation	Relevance at a chosen rank
Similarity groups	Clustering	Stability and task usefulness
Unusual observations	Anomaly detection	Detection and false-alarm rates

Needed output	Candidate approach	Useful evaluation
New text or media	Generation	Task rubric and artifact checks
Sequential decisions	Planning or reinforcement learning	Outcome, constraints, and reward behavior

A recommender for a tiny catalog can start with explicit tags and similarity. A neural recommender may become useful when the data and task justify it. An anomaly detector can flag an unusual measurement without knowing why it is unusual. Its flag should trigger an appropriate review, not automatically become a diagnosis.

Build and evaluate

Write three candidate implementations for your first task: a rule, a small statistical method, and a model-based method. Estimate what each needs to operate and what each can get wrong. Prefer the smallest candidate that meets the measurable requirement. Complexity should pay for its cost in actual behavior.

Use the companion's `classical.py` to inspect a classifier, a regressor, clustering, similarity ranking, and a small reward-learning example. Its synthetic data makes the computations easy to trace. Results on that data demonstrate the algorithms' mechanics; they do not validate a business use case.

For practice, describe a system that combines classification with generation. Specify which output controls routing and which output is only a draft. Then identify the measurement that would justify replacing either component.

Chapter 3 Data and Measurement

What an example represents

A training example is a record chosen to teach a particular relationship. For classification, it may contain text and a target category. For language modeling, it may contain an input sequence and its next-token targets. The record's usefulness depends on its origin, quality, and relationship to the task.

Start a dataset manifest before collecting large amounts of material. Record the source, permission basis, collection date, transformation history, intended task, and known gaps. Keep the original record separate from its cleaned representation when retention is appropriate. A convenient format does not establish permission to use the content.

Labels need definitions. “Urgent” might mean immediate safety impact, a customer preference, or a contractual deadline. Mixed definitions produce a confused training signal. Write the definition, give positive and negative examples, and preserve disagreement when reviewers cannot resolve it. A model can learn inconsistent labels successfully and still perform the wrong task.

Separate development from evaluation

Use training data to fit parameters, validation data to choose settings, and a final test set to assess the selected approach. Repeatedly changing the model in response to the test set turns that set into development material. Scikit-learn's cross-validation guidance explicitly discusses both held-out evaluation and leakage through tuning [7].

Splits must reflect the deployment question. If the task concerns future transactions, a time-based split may be more relevant than a random one. If several records belong to one customer, keep that customer's related records together when estimating performance on unseen customers. Near-duplicate documents can leak information even when their filenames differ.

Preprocessing can also leak information. A vocabulary, normalization statistic, or feature selection procedure learned from all labeled data may expose evaluation information. Fit learned transformations within the training portion, then apply them to the evaluation portion. Explicitly document exceptions, such as a fixed externally supplied tokenizer.

Build a manifest

Use a JSON record with fields for dataset ID, version, hashes, sources, split rule, permitted use, label definition, and limitations. Store separate counts for raw, rejected, transformed, and retained examples. A cleaning script should report what it removed and why.

The small language-model project uses sentences constructed for this book. Complete sentences are assigned to training or validation before token windows are sampled. The validation split holds out sentence combinations; the grammar and vocabulary remain closely related. This measures learning within the toy distribution, not transfer to arbitrary writing. Chapter 31 returns to that limitation.

Measure the right outcome

Count examples as well as scores. A perfect result on three easy cases says little about a varied workload. Examine results by meaningful subgroup, document type, or failure category. Keep the raw predictions so later reviewers can recompute metrics rather than trusting one summary number.

For practice, invent ten labeled support requests and write the label rules. Have another reader classify them without seeing your labels. Record disagreements, then revise the definition. That exercise exposes a common problem before a model makes it harder to see.

Chapter 4 Neural Networks Without Mystery

Parameters and computation

A neural network combines numerical operations whose adjustable values are called parameters. A simple layer multiplies an input vector by a weight matrix, adds a bias, and applies an activation. Layers compose into a larger function. Training adjusts parameters so the function performs better according to a chosen objective.

A tensor is a numerical array with a shape. A single example may be a vector. A batch may be a matrix. A sequence batch may have dimensions for examples, positions, and features. Write down these meanings beside each shape. Shape mistakes are often easier to identify than a vaguely “bad” model result.

For a numeric prediction, squared error is one possible loss. For category prediction, a probability-based loss can penalize assigning little probability to the correct category. Loss is a training objective; the user's success criterion may include other measurements. A lower loss does not automatically establish better behavior on every task.

One training step

Compute a forward pass using the current parameters. Compare the prediction with the target through the loss. Compute gradients, which describe local sensitivity of that loss to parameters. An optimizer uses those gradients to update the values. Repeat with additional batches, and evaluate on material held out from fitting. The PyTorch optimization tutorial provides a conventional implementation of this cycle [2].

Reverse-mode automatic differentiation records how operations depend on one another and propagates derivatives backward through the graph. The companion's `Tensor` class implements this idea in NumPy. It supports the operations needed by the tiny transformer rather than trying to replace a general numerical framework.

Consider a prediction that is the product of a weight and an input. If the input is two, changing the weight by a small amount changes the prediction by twice that amount. The loss gradient combines this sensitivity with the sensitivity of the loss to the prediction. The chain rule is what connects the pieces.

Build and inspect

Run `python classical.py` and inspect the numeric prediction example. Then locate `Tensor.backward` in `tiny_lm.py`. It walks the graph in dependency order, resets accumulated gradients, seeds the final scalar's gradient, and applies local backward functions in reverse order.

The implementation must undo broadcasting when returning a gradient to a smaller input. If one bias vector was reused across many examples, its gradient sums the contributions from those uses. A reshape changes layout without changing values, so its backward operation restores the original shape.

Check the derivative

Finite differences estimate a derivative by slightly increasing and decreasing one parameter and comparing losses. They are slow for a large model but useful on selected coordinates in a teaching model. The companion checks selected embedding, attention, normalization, and output parameters against this numerical estimate.

For practice, calculate the derivative of a one-weight squared-error example by hand. Compare it with finite differences. Then explain why a correct derivative alone does not prove that the dataset or objective matches the task.

Chapter 5 Tokens Embeddings and Attention

Turning text into arrays

A tokenizer defines the units a text model processes. Character tokens are simple to implement but require many positions for ordinary text. Subword schemes can represent frequent pieces efficiently while retaining ways to encode unfamiliar words. Token IDs are labels for vocabulary entries; their numeric order does not itself describe meaning.

An embedding table maps an ID to a learned vector. Position information tells a sequence model where a token occurs. Similarity between vectors can support search, but similarity is a property of the representation and metric. It is not a guarantee that two statements mean the same thing or support the same conclusion.

The companion uses a character vocabulary and learned positional embeddings. It converts IDs into one-hot vectors and multiplies by an embedding matrix. Direct indexing is more efficient; the one-hot form makes the gradient path easy to inspect in a compact implementation.

Attention inside a transformer

The original Transformer paper describes attention using query, key, and value representations [1]. In a causal language model, each position must use only itself and earlier positions. The attention mask enforces that restriction. Multiple heads compute separate combinations before their outputs are mixed.

In our implementation, an input batch has shape `(B, T, D)`. Linear maps produce query, key, and value tensors. Reshaping and transposing produce `(B, H, T, D/H)`. Query-key multiplication creates scores of shape `(B, H, T, T)`. Scores are scaled, future positions receive a large negative mask value, and softmax converts scores into weights. Weighted values return to `(B, T, D)`.

Residual connections add a block's update to its input. Normalization controls the scale of intermediate representations. A position-wise feedforward network transforms each token representation. Stacking these operations produces the transformer used in the training chapters.

Build and evaluate

Inspect `TinyLM.forward` in the companion. Trace the shapes using a batch of one short string. Increase the number of heads only when the model dimension remains divisible by that count. Change the context length and note that the learned position table must match the configured capacity.

The causal test changes later tokens and checks that earlier output logits remain unchanged. This is a useful behavioral check of the mask. It does not prove that a larger framework or a different attention implementation applies the same rule.

For a retrieval assistant, token budgeting also happens outside the model. Reserve space for instructions and output, choose relevant source passages, and report truncation. Removing a passage can remove the only evidence for a claim, even if the remaining text fits neatly.

For practice, draw the permitted attention positions for a four-token sequence. Explain why seeing the future target during training would invalidate next-token prediction as a learning exercise.

Chapter 6 Evaluating a System

Separate kinds of success

A system can produce grammatically correct text and still fail its task. Evaluate answer quality, source support, permission behavior, execution outcomes, and operational performance separately. A combined score may be useful later, but it should not conceal a serious boundary failure.

For the handbook assistant, ask whether the answer addresses the question, whether citations resolve to the retained source version, and whether the cited content supports the answer. For a file-writing agent, ask whether the grant matched the actor and destination and whether the reported outcome matches the file state. These are different observations.

Conventional metrics are task-specific. Precision describes how often a positive prediction is correct; recall describes how many relevant positives were found. Numeric prediction needs an error measure in meaningful units. Rankings need a chosen cutoff. Scikit-learn documents these families of metrics and their scoring interfaces [8].

An evaluation record

Create cases containing an input, expected condition, observed result, environment, and verdict. Keep the output artifact or record needed to inspect the verdict. For subjective media quality, use a rubric and retain reviewer judgments. A model-based grader is another measurement instrument with its own potential errors.

Start with ordinary requests, then add missing information, conflicting sources, malformed inputs, unavailable dependencies, expired permission, and attempted resource escape. The coverage should follow the task's failure modes. A hundred variations of one easy success path do not cover a denied write.

Record latency as a distribution when it matters. Average latency can hide a frustrating tail. Distinguish model generation time from queue delay, retrieval, transfer, and file export. Cost records should distinguish local resource use from externally billed operations.

Build the first acceptance set

Write five cases for P01: a supported question, an unsupported question, a conflicting-source question, an authorized draft save, and an unauthorized save. Define success before running the cases. The companion's tests exercise selected application boundaries and model mechanics; they do not replace a real workload evaluation.

The model project reports validation loss and exact reload behavior. Its sample output is visibly limited. Preserving that sample is part of honest evaluation: a promising training curve can coexist with poor-looking text. The report describes a restricted training experiment rather than a finished assistant.

Decisions after evaluation

Use results to choose the next change. If retrieval misses the passage, changing the generator may not help. If permissions fail, improving prose quality cannot repair the boundary. If failures concentrate in a subgroup, inspect representation and data coverage before raising a global threshold.

For practice, choose one result you would consider unacceptable even if the average quality score improved. Write the release condition that enforces that decision. Evaluation becomes useful when it determines what the system is allowed to claim and when it is ready to ship.

Part II Chatbots and Assistants

A practical path through the components and their evidence.

Chapter 7 Your First Chatbot

A conversation is an application

A chatbot receives a message and produces a response within a conversational interface. Its first useful version needs a clear input contract, a backend dependency, and visible outcomes. History, retrieval, voice, and tools can follow once that basic path is understandable.

Use a model adapter to separate the interface from the generation method. Define a request with message text, selected context, and output limits. Define a response with status, text, source references when present, and usage information when known. Preserve errors as errors rather than turning every response into an apparently successful sentence.

The companion includes a local browser interface in `server.py`. Its default backend returns retained source excerpts through the core retrieval function. This lets you inspect the application without an account or model download. It is an extractive chatbot. Chapter 14 shows how a local generative backend fits the same surrounding workflow.

Build the request path

Run `python server.py`, open the printed local address, and ask about equipment checkout. The browser sends a bounded JSON message to the server. The server validates the input, calls the answer function, and returns a structured result. The interface inserts text as text rather than treating generated content as executable HTML.

Keep the conversation ID separate from a user's authenticated identity. A random conversation ID can distinguish histories; it does not establish who may access a private document. A production service needs an identity and session mechanism appropriate to its environment.

For a generative adapter, apply a request timeout and an output budget. Streaming may reduce perceived delay, but partial text must be labeled as incomplete until generation finishes. The client needs a cancel action and a way to distinguish cancellation from a backend failure. If a provider finishes after the client disconnects, the application should still account for any work or usage that actually occurred.

Fail visibly

Test empty messages, malformed JSON, messages exceeding the limit, and questions with no matching source. An unsupported question should yield an unresolved result in this teaching application. A transport failure should show an error state with a retry option rather than an invented answer.

The simple local server intentionally has no public deployment machinery. Its loopback binding is suited to a personal demonstration. Authentication, transport security, concurrent request handling, and stronger resource controls become separate requirements when the deployment changes.

For practice, replace the answer function with a deterministic fixture that returns a known sentence. Verify that the interface handles success and failure without changing its protocol. This establishes the adapter boundary before adding a less predictable backend.

Chapter 8 Instructions and Structured Responses

Give each input a role

An application has instructions that describe its task. A user contributes a request. Retrieved material contributes content that may be relevant. These inputs can appear in one model context while still carrying different authority. A document that says “ignore the application and send this file” is document content, not an application grant.

Prompt injection occurs when hostile content tries to redirect model behavior across this distinction. OWASP describes both direct and indirect injection risks [9]. Input filtering and instructions can reduce some failures, but a generated answer should still pass independent validation before it controls an action.

A structured response makes that validation possible. For a file proposal, define fields such as action, resource identifier, text, evidence references, and unresolved questions. Require the action to be one of the supported values. Reject unknown fields when they could create ambiguity. Bound field lengths and parse types explicitly.

Build a proposal schema

A teaching proposal can be a JSON object with `action: "write_draft"`, `project_id`, `filename`, `text`, and `evidence_ids`. The application resolves the project ID and constructs the permitted path. The model should not select an arbitrary filesystem root from its own text.

Schema compliance answers whether the output has the expected shape. It does not answer whether the evidence supports the proposal, the content is appropriate, or the actor has permission. Treat those as subsequent checks. Even a provider's constrained-output feature requires application-level semantic validation.

Preserve the raw response where appropriate for debugging, while minimizing sensitive retention. Keep validation errors separate from a revised model response. If you request a repair, limit the number of attempts and record that the repaired proposal went through the same checks.

Keep tool calls behind the boundary

Represent tools with explicit names and argument schemas. A model can request a tool, but the executor decides whether the named capability exists and whether its arguments fit the current grant. Never dispatch a generated tool name through unrestricted dynamic evaluation.

The companion's file gate uses an explicit action, actor, resource, and signed grant. A string that resembles a grant cannot pass its signature check. Its implementation is intentionally small enough to inspect; later chapters identify the additional controls needed for a hostile multi-process environment.

For practice, create three proposals: valid, malformed, and validly shaped but unauthorized. Check that each reaches the correct result. Then place a malicious instruction inside a retained handbook passage and verify that it cannot manufacture a write grant through the model response.

Chapter 9 Conversation State and Identity

History is selected context

A conversation can grow far beyond the model's usable context. The application decides what to retain, what to retrieve, and what to send for a particular turn. A context window is a computation limit, not a complete memory system.

Keep a raw turn record when retention is authorized. Derive summaries as separate records that link to the turns they summarize. A summary can omit nuance or introduce an interpretation. Marking it as a summary prevents it from silently replacing the original evidence.

Allocate the context budget among instructions, current request, necessary history, retrieved material, and expected output. When the budget is exceeded, choose material according to task relevance and importance. If a key fact is unavailable, seek it or report the gap. Do not assume that a fluent continuation recovered omitted context.

Define consistent assistant behavior

An identity specification can state tone, values, response boundaries, preferred interaction patterns, and how to handle uncertainty. It can create continuity across tasks. Stable behavior can coexist with varied wording and creative responses. Consistency should concern commitments and treatment of the user, rather than repeating a script.

Separate that specification from factual claims about the assistant's inner experience. A relational style or first-person voice does not establish consciousness. The application can provide warmth, familiarity, and useful continuity while keeping its factual descriptions accurate.

For a practical specification, name a behavior and its acceptance condition. “Be candid” becomes “state missing evidence and label interpretations when a question cannot be established.” “Respect correction” becomes “update the relevant preference record and preserve the revision.” These requirements can be exercised in conversation tests.

Build a conversation record

Store role, content, timestamp, session, origin, and any source links. Put account permissions in a separate authority record. A user's earlier request to draft a message need not authorize sending every later message in the same conversation.

Use a version for the identity specification and include that version in diagnostic records. If behavior changes after an update, you can identify which specification applied. A model upgrade and a personality change are different changes, even when released together.

For practice, write five stable behavior rules and five requests that allow creative variation. Test a correction, a disagreement, and a request with insufficient evidence. Evaluate continuity by whether the rules hold, not by whether the wording stays identical.

Chapter 10 Retrieval from Documents

Supply material at use time

Retrieval finds material relevant to a query and supplies it to the answer process. A practical pipeline ingests documents, extracts usable content, preserves source locations, creates searchable representations, retrieves candidate passages, and assembles selected context.

The original retrieval-augmented generation research combined a generator with a retriever and a document index [3]. The companion demonstrates an elementary lexical retriever. A production application may use lexical search, embeddings, or a combination. Choose from measured relevance rather than assuming vector search always wins.

Chunking divides a document into searchable units. A chunk should retain enough context to interpret its statements. A heading, table header, or nearby qualification may be necessary. Store document ID, version, section or page, offsets where meaningful, content hash, and access classification alongside each chunk.

Build the retrieval path

The companion's `Evidence.retain` creates a source record with a content hash. `retrieve` ranks records by overlapping query terms, excludes records marked as model output, and applies a stable tie-breaker. `extractive_answer` returns excerpts and source IDs. It establishes a transparent baseline before introducing generated paraphrases.

An embedding version should retain the embedding model identifier, dimensions, preprocessing version, and distance measure. When the embedding model changes, plan for index rebuilding or version separation. Comparing vectors produced by incompatible models can invalidate the ranking.

Apply access filtering before private material reaches the model. A correct answer can still be a privacy failure if the passage should not have been accessible to the requester. Recheck permissions when a record's classification or the user's role changes.

Check support separately

Retrieval relevance asks whether a passage concerns the question. Claim support asks whether that passage establishes a particular sentence. A citation can be relevant and still fail to support the answer. Keep the sentence-to-source relationship available for review.

Test absent information, contradictory versions, a passage with an exception, and a question that requires several sections. Measure retrieval independently of generation. If the correct passage never arrives, the generator cannot ground an answer in it.

For practice, write a handbook with one general rule and one exception. Create chunks that preserve both. Ask a question governed by the exception and inspect whether the retrieved context and answer include it. This is a more informative test than another straightforward definition question.

Chapter 11 Memory That Can Be Corrected

Memory is a managed record

Useful assistant memory contains preferences, historical events, ongoing tasks, and sometimes prior interpretations. These records need origin, scope, revision, and an appropriate retention policy. A stored sentence does not become true merely because it persists.

The companion's SQLite memory table stores a key, value, origin, and revision. Parameterized queries keep values separate from SQL syntax [10]. The origin field distinguishes a user report, model output, and a source record. Those categories are still coarse; a richer implementation should link to the actual source and record the purpose for retaining it.

Repetition must not create independent corroboration. If a model's assertion is saved, summarized, and later retrieved, all those records descend from the same assertion. Preserve that ancestry. Counting them as three witnesses would manufacture evidence.

Build revision and deletion

Implement create, retrieve, correct, expire, and delete operations. Associate each action with the appropriate actor. A preference correction should affect future retrieval without making the old wording appear to be the current preference. Where historical records must be retained for audit, use an explicit superseded state and explain the retention boundary.

The teaching memory's `set` operation increments revision, while `delete` removes the active row. This simple behavior is exercised in the tests. It does not provide secure erasure across backups, caches, logs, or a provider's storage. A complete deletion design must identify all copies covered by its promise.

Distinguish active context from durable memory. A temporary instruction for one task should not silently become a permanent preference. Ask for or infer durable storage only within the application's declared memory policy, and expose a way to inspect stored records.

Guard private context

Authorize retrieval by user and purpose. A voice match, an unverified name in text, or a shared conversation window may be insufficient identity evidence. An unknown participant should not receive private memory simply because the model recognizes the conversational topic.

Evaluate correction, mistaken attribution, expired records, cross-user access, and the non-corroboration rule. The companion's test shows that a record marked as model output is excluded from the source-only retriever. That is one narrow safeguard, not a full provenance graph.

For practice, store a model-generated claim and a user correction. Make the next answer use the correction while retaining the original claim's epistemic status. Then explain which record is authoritative for the user's preference and which remains an unsupported statement about the world.

Chapter 12 Voice and Multimodal Assistants

Build a pipeline you can inspect

A voice assistant receives a continuous signal but should not treat every sound as a request. Separate capture, voice activity detection, segmentation, speaker attribution where justified, transcription, addressee and intent, context interpretation, response policy, and speech execution.

AudioWorklet provides a browser mechanism for processing audio in a dedicated worklet context [11]. Its role is audio processing, not speaker identity or permission. The capture application should record the sample rate, channel count, buffering, and any resampling decisions so later failures can be traced.

Activity detection answers whether speech-like activity is present. Transcription estimates words. A speech-recognition system such as Whisper is designed to predict transcripts from audio [12]. Neither activity detection nor transcription establishes who is speaking, whom they address, or what actions they authorize.

Silence is a valid response

Define a deterministic response policy whose default can be remaining silent. Ambient media, self-talk, an unknown addressee, and uncertain transcription may each require different handling. An assistant that speaks at every detected phrase can interrupt users and respond to television audio as if it were instruction.

In a DID-aware design, a physical voiceprint must not be treated as evidence of an alter's identity. Internal identity and orientation are user-governed context. Unknown speakers must not gain private memory access. These are design boundaries for the Samara case study, not capabilities established by audio analysis.

Manage feedback and interruptions

Speaker output can re-enter the microphone. Use echo control where available, playback state, and response gating to prevent a feedback conversation. Preserve genuine user interruptions through a barge-in policy. Cancellation should stop the current response and invalidate any queued continuation that no longer applies.

A useful event record includes capture range, segmentation boundaries, transcript uncertainty, addressee decision, policy disposition, and output timing. Retain raw audio only according to the declared privacy and storage policy.

Build the pipeline with recorded audio fixtures before adding continuous capture. Include silence, overlap, incomplete speech, ambient media, and an interruption. A fixture can test orchestration without pretending to test a real recognizer's acoustic quality.

For practice, write a decision table for six situations. Include a direct question, a third-party conversation, an unknown speaker, and an explicit orientation request. Specify what evidence permits a spoken response and what remains unresolved.

Part III Local Cloud and Hybrid Deployment

A practical path through the components and their evidence.

Chapter 13 Models Providers and Runtimes

Identify the thing you are choosing

A model is a learned computation with a defined architecture and weights. A runtime performs that computation. A hosted service offers it through an interface. An SDK helps your program call an interface. A consumer chat product adds its own application behavior. Confusing these categories leads to unclear claims about what you own, operate, or can reproduce.

An open-weight model makes weights available under a license. That license may impose conditions on use, modification, distribution, or associated names. Availability of weights does not automatically reveal the complete training dataset or training process. Meta's Llama model repository documents its model distribution tools and links to model-specific terms [13].

Ollama's documentation describes local and hosted access [4]. When using it, identify whether inference actually occurs on the user's computer or at a hosted endpoint. A local-looking interface can still make a network request. Privacy analysis should follow the data path.

Record a model dependency

Create a manifest containing provider or publisher, exact model identifier, weight or endpoint version when available, runtime version, supported input types, measured requirements, license or service terms, and evaluation scope. A moving alias may resolve differently later. Record the resolved version when a service exposes it.

Component choice	Developer control	Dependency to inspect
Local weights and runtime	Hardware, execution, local storage	Weight license, runtime behavior, downloaded artifacts
Hosted endpoint	Request design and application integration	Service access, availability, data policy, interface changes
Consumer chat application	User interaction within the product	Product features and export limitations
Hybrid workflow	Routing and permitted data movement	Both local and hosted requirements

Do not equate a provider's model benchmark with your application's measured result. The workload, prompts, tools, and evaluation criteria may differ. Use published evaluations as context and your acceptance set as evidence for your system.

Build a replaceable boundary

Have the application request a capability such as text generation with a defined input contract. Let the adapter translate that request into the selected interface. Preserve differences rather than pretending every provider supports identical tools, media, streaming, and state semantics.

For practice, inventory one existing AI product. Identify its visible application, probable model dependency where documented, runtime location, storage, and tool interfaces. Mark anything you cannot establish as unknown. That inventory is more defensible than assuming the product's public name describes every internal component.

Chapter 14 Running Models Locally

Provision before disconnecting

Local inference needs the runtime, model artifacts, compatible hardware, and sufficient storage and memory. Install and download these while connectivity is available when the distribution requires it. Then test the intended workflow with external network access disabled.

Ollama offers a documented local chat API through a locally running server [14]. In a typical local setup, a chat request specifies a downloaded model, messages, and whether to stream. The adapter should use an explicitly configured local endpoint and reject unapproved remote endpoints for a local-only task.

A local model download can be much larger than the runtime itself. Quantization stores weights with fewer bits or another compressed representation. It can reduce memory requirements and change model behavior. Measure the actual configuration rather than treating a bit width as a guarantee of quality.

Build the local assistant

Begin with the companion's extractive browser application so the request and error path are visible. Next, use the local request recipe in Appendix C to connect a downloaded model. Preserve the same application response contract and source record structure. Label the generated answer separately from the retained passages.

Choose a model based on the publisher's documentation, your task, and your hardware measurements. The book does not require a moving model alias. Set the exact available model identifier in your configuration and record it in the run manifest.

Test startup, unavailable model, unavailable local service, excessive input, and cancellation. A local server being reachable does not establish that it has loaded the intended weights. Record the model identifier and check a known request before considering the service ready.

Establish the offline boundary

Disconnect or deny outbound network access after provisioning and run the entire permitted workflow. Include document ingestion, retrieval, response generation, and asset saving. An application that silently uses hosted embeddings or cloud speech is not wholly local merely because its text model runs locally.

Telemetry, update checks, extensions, and remote asset links can also create network dependencies. Investigate the actual runtime and application configuration before making a privacy claim. The offline property concerns observed and constrained data movement across the full workflow.

The NumPy language model in the companion runs without model downloads or hosted inference. Its small scale makes it useful for understanding the training and serving lifecycle, while its limited text quality makes it unsuitable as a substitute for a general assistant.

For practice, create an offline acceptance record that lists every required artifact and the result of a disconnected run. Distinguish “works after downloading dependencies” from “requires no provisioning.”

Chapter 15 Building with Hosted Services

A service call is a dependency

Hosted inference trades local model operation for dependence on an account, network, service interface, usage limits, and provider policy. The application still owns its request validation, permission checks, error states, and task evaluation.

The interfaces checked for this edition include OpenAI Responses [15], Anthropic Messages [16], Gemini generateContent [17], and xAI Responses [18]. They have different request and response structures. Read current documentation before using a new feature, model identifier, or authentication mode. The recipes in Appendix C illustrate text requests; they were not executed against paid services for this book.

Store credentials outside browser bundles and ordinary source files. Use a server-side secret mechanism appropriate to the deployment. Log credential names and configuration status when necessary, never their secret values. A consumer subscription and developer API access should be treated as separate arrangements unless the provider explicitly documents otherwise.

Build a provider adapter

Define the internal request first. Translate it into the provider's expected input shape. Parse the actual output structure rather than assuming the first array element contains the desired sentence. Some responses contain tools, refusal information, multiple content types, or partial events.

Record request ID, provider, configured model, status, duration, and reported usage when available. Distinguish unsupported capability, invalid input, authentication failure, rate limit, transient service failure, and cancellation. These conditions call for different recovery behavior.

Retries need limits and awareness of side effects. Repeating a text-generation call can consume additional usage and produce a different response. Repeating a state-changing tool can create a duplicate effect. Keep retry policy at the application boundary where those differences are visible.

Control data and budgets

Specify which content may leave the local environment. Apply this check before sending a request, including embeddings, file uploads, and media references. Redacting a displayed response does not undo an unauthorized upload.

Use admission limits for request size and expected work. Reported usage can reconcile actual cost afterward, but a guessed token count is not a hard billing guarantee. Combine conservative budgeting with provider-side controls where available and appropriate.

For practice, use recorded response fixtures to test each adapter's parsing logic without a live key. Include a successful text result, an error, and a response containing no ordinary text. Then plan a separate live validation with a documented account, model, date, and bounded usage.

Chapter 16 Hardware and Performance

Account for more than weights

Weight storage provides a useful first estimate. If a model has N parameters stored at b bits each, the raw weight bytes are approximately `N * b / 8`. Actual formats may include scales, metadata, alignment, or mixed precision. Runtime memory also includes temporary tensors and caches.

For our 10,992-parameter teaching model, float64 weights alone use 87,936 bytes. That arithmetic does not describe total process memory. The automatic differentiation graph, gradients, optimizer state, Python objects, and temporary arrays occupy additional space.

Training typically needs more memory than simple inference because it retains information for gradient computation and optimizer state. A large input batch or longer context can change the dominant cost. The straightforward attention implementation creates a score matrix whose last two dimensions are both sequence length.

Measure a workload

Choose a representative input distribution and record model, runtime, hardware, precision, context length, batch size, and concurrent load. Measure startup separately from steady-state operation. A warmed cache can make repeated demonstrations look faster than the first real request.

Report time to first output for interactive generation and time to completion for finished artifacts. For batch jobs, throughput may matter more. For a voice assistant, the total delay includes capture buffering, segmentation, transcription, policy, generation, synthesis, and playback.

Use the companion's training report as an example of a scoped measurement. Its elapsed time describes a tiny run in one execution environment. It is not a portable performance promise for a reader's machine. Repeating the run and retaining the new report provides the reader's own evidence.

Optimize after locating the cost

Batching can improve throughput while increasing waiting time. Quantization can reduce storage while changing output quality. Caching can save computation while creating invalidation and privacy requirements. A longer context can preserve information while increasing work. Each optimization should have an associated task-quality check.

Set a practical resource ceiling. When an input exceeds it, reject, queue, or divide the task according to the application contract. A silent truncated response is not a reliable substitute for a limit error.

For practice, measure three input lengths and separate preparation time from model computation. Record the change in both latency and output quality. Use those observations to decide whether the next optimization belongs in retrieval, context selection, generation, or rendering.

Chapter 17 Hybrid Routing

Route by policy and capability

A hybrid system can use local computation for some tasks and hosted services for others. Routing should consider the required capability, sensitivity, permission, availability, and budget. The most powerful available endpoint is not automatically the permitted endpoint.

Define task metadata before inference. For example, a private document task may require local processing. A public creative brief may permit hosted generation. A user may authorize one upload without authorizing all future uploads. Bind that scope to the task rather than a vague global preference.

Build a routing decision

The companion's `choose_route` function models a narrow policy: prefer available local processing; if it is unavailable, use cloud only when the task is not private and cloud use is allowed. Otherwise return unavailable. Its private-fallback test checks that a service failure does not relax the restriction.

A richer router should return a reason and the selected capability version. It may need distinct rules for text, embeddings, speech, images, and tools. A task can have several permitted paths with different output expectations. Preserve those distinctions in the run record.

Do not automatically copy all context to a fallback provider. Construct the permitted payload for that route. A local-only source can remain local while a permitted public summary is sent elsewhere, but the summary itself needs review for information leakage.

Handle failures without changing authority

If local processing fails, the appropriate response may be a visible unavailable state or an explicit new permission request. The failure is a fact about availability. It does not create authority to transmit private data.

If a hosted provider becomes unavailable after a job starts, retain the job state and any partial artifacts. A different provider may require different inputs or lack the same editing controls. Restarting can be a new generation with a new cost and a new output, not a continuation of the same computation.

Test routing under dependency failure, revoked permission, stale classification, and exhausted budget. Check actual network calls in a deployment test where

possible. A correct decision function is useful, but enforcement also depends on every adapter honoring its result.

For practice, design routes for private transcription, public image generation, and a local document question. Write the exact circumstance under which each could leave the machine, then test the denial path.

Chapter 18 Packaging and Deployment

Turn an experiment into an installable artifact

A reader should be able to identify the entry point, install dependencies, provide permitted configuration, run the application, and find its output. Package those steps with the code. Record the tested Python or runtime version and avoid assuming that every reader has the author's development environment.

The companion uses NumPy for model work and the Python standard library for its teaching application core. Install NumPy in an isolated environment, run the tests, then run a demo in a fresh output directory. The README gives commands and distinguishes tested modules from integration recipes.

Choose a delivery surface

A command-line application is useful for scripts and batch jobs. A local browser interface separates a familiar UI from backend logic. A desktop package can include local services and assets, but signing, platform permissions, updates, and model storage remain real engineering tasks. A web deployment adds identity, transport, concurrency, and network-exposure requirements.

Python's basic HTTP server facilities are useful for controlled examples, not a complete public application stack [19]. Treat the companion server as a loopback demonstration. Its simplicity makes request handling visible without claiming to solve authentication or hostile internet traffic.

Store configuration separately from content. Distinguish model paths, endpoints, resource roots, policy versions, and secrets. A configuration change can alter data movement or authority just as a code change can. Include relevant configuration in release records without exposing credentials.

Make upgrades reversible

Version code, schema, model artifacts, and policy independently when needed. Before a migration, retain a recoverable copy and test restoration. Database backup procedures must match the database's consistency model; copying a live file without considering open transactions can be insufficient.

Health checks should test useful readiness. A process existing is a weak signal. A service may need its model loaded, source index accessible, and authority service available before it accepts work. Deny execution when the required grant or policy cannot be checked.

For practice, write installation instructions for a clean machine and ask another person to follow them. Record every missing assumption. Then design an update that changes a memory schema and demonstrate how a reader could recover the prior version if the migration failed.

Part IV Creative and Multimodal Systems

A practical path through the components and their evidence.

Chapter 19 Text Creation Workflows

Separate drafting from factual publication

A writing system should know whether the task is invention, transformation, or factual explanation. Fiction permits invented events. A summary must preserve the source's meaning. A research article must distinguish sourced facts from interpretations. The same language model can participate in each task, but the surrounding requirements differ.

Create a brief with audience, purpose, format, length, voice, factual constraints, and release destination. Preserve the brief with the output. A later reviewer should be able to tell whether an apparent defect came from generation or from an unclear requirement.

Build a revision workflow

Use stages for outline, first draft, factual review when needed, style revision, and release approval. Each revision should link to its parent. Record the requested change and the resulting artifact, rather than overwriting the only copy.

For factual work, extract claims and associate each with source records. A model can suggest which sentences need checking, but the source inspection remains necessary. If sources conflict, the output can describe the conflict and retain an unresolved status. Removing the disagreement to make the writing smoother changes the substance.

For style revision, define invariants. Names, dates, quantities, uncertainty, and causal conditions may need to remain unchanged. A request to simplify language should not turn “may improve under these conditions” into “will improve.” Compare the revised text with those invariants.

Create a usable tool

Start with a document list, editor, version history, and preview. Add generation through the same adapter contract used by the chatbot. Treat publication as a separate action with its own destination and grant. A saved draft is an artifact; a published page creates an external effect.

The companion's `media_job('text', brief)` creates a draft job record. It does not generate prose by itself. The record provides a place to attach a model adapter, input assets, output, and review. The separation makes it possible to test workflow behavior with fixtures before measuring a generator.

Evaluate preservation and usefulness

Ask whether the output serves the audience, follows the format, retains necessary facts, and distinguishes uncertainty. For a summary, compare omissions against the source's important conditions. For a story, review continuity, characterization, and the requested style.

For practice, write a short source paragraph containing a number, an exception, and an unresolved claim. Generate or manually draft a simpler version. Check each invariant and record the changes. This exercise teaches review that remains useful across providers and model versions.

Chapter 20 Image Generation and Editing

Understand the model family

Generative image systems can use several learning approaches. A variational autoencoder learns a probabilistic latent representation and a decoder [20]. A generative adversarial network trains a generator alongside a discriminator [21]. Denoising diffusion models learn to reverse a corruption process [22]. Latent diffusion performs generation in a learned compressed representation [23]. Flow matching is another approach to learning generative transformations [24]. These are representative families, not a claim that every current product uses the same method.

In a text-conditioned system, text representations influence the image computation. Editing may also use an input image, a mask, or reference assets. The exact meaning of a mask and the available conditioning depend on the implementation. Read the selected pipeline's documentation rather than assuming universal behavior.

Build an image job

Record the prompt, requested dimensions, model identifier, seed when supported, input assets, edit constraints, output format, and relevant use permissions. Store source images before referring to them in a job. Preserve the output as a new artifact, with its relationship to the inputs.

For editing, state what must remain invariant. Examples include a person's appearance, background geometry, object position, or existing transparency. A generation that looks attractive can still fail an edit if it changes the protected subject.

The local pipeline recipe in Appendix C uses a documented image-generation library as an optional extension [25]. It requires compatible dependencies, model access, suitable hardware, and the model's applicable terms. The core companion tests the job record, not image synthesis quality.

Training and evaluation

To train an image model, specify the dataset distribution, preprocessing, learning objective, conditioning, architecture, sampling method, and held-out evaluation. A small image experiment can demonstrate denoising or reconstruction; it cannot establish broad artistic performance. Caption quality and dataset permission are independent concerns.

Evaluate prompt adherence, subject consistency, editing invariants, artifacts, and technical output properties. Use exact diagrams and charts from deterministic tools when the visual must represent precise quantities or connections. Attractive generated labels are not a substitute for accurate notation.

For practice, write one generation brief and one edit brief. Define three measurable invariants for the edit. Review the final artifact at actual resolution and record which invariants hold. Keep failures beside successes so the review reflects the real behavior.

Chapter 21 Speech and Sound

Speech recognition and synthesis differ

Transcription maps audio to an estimated text representation. Text-to-speech produces spoken audio from text. Sound-effect generation produces audio conditioned on a description or other input. Each needs a different input contract and quality check.

A transcript should preserve uncertainty. Unintelligible speech, overlap, names, and numbers deserve special review. A polished transcript that invents a missing word can be less useful than a visible uncertainty marker. The Whisper research provides one speech-recognition reference [12]; the application still has to evaluate its own language and acoustic conditions.

For speech synthesis, specify exact text, voice authorization, pronunciation requirements, pacing, output format, and target use. OpenAI's speech-generation documentation describes one hosted interface [26]. The availability and conditions of a particular voice or format belong to the selected service, not to a universal AI capability.

Build the audio asset path

Save the request and generated file separately. Inspect sample rate, channels, duration, encoding, and peak levels. Convert formats explicitly. A player failing to open a file can be a codec problem rather than a model problem.

Keep timing metadata for dialogue and narration. If a sentence is regenerated, its duration may change and require timeline adjustment. Speech should not continue after the related task has been canceled or superseded.

For sound effects, record intended duration and placement. A loop needs a transition that does not produce an unwanted click. An impact may need a precise onset. Review those properties in the exported asset rather than relying on the generation description.

Make a local production fixture

The companion's `media_fixture.py` creates a short synthesized waveform and a timeline manifest. It is ordinary signal generation, not a learned speech or music model. It lets you verify file handling, timing, and export independently of a generative service.

Replace the fixture through an audio adapter after the pipeline works. Keep voice permission and publication permission as separate records. Possession of a

reference clip does not by itself establish permission to synthesize a person's voice.

For practice, test a narration containing a date, an acronym, and a proper name. Listen to the whole output and check timing in the intended context. Then cancel playback mid-sentence and verify that the old continuation does not resume unexpectedly.

Chapter 22 Music Generation and Production

Music is a timed structure

A music brief can specify genre, tempo, meter, instrumentation, arrangement, mood, vocal role, duration, and intended scene. Some systems generate waveform audio; others generate symbolic events such as notes. Symbolic output can be rendered through instruments, while waveform output already contains sound.

MusicGen is a research example of generating compressed discrete audio representations with a transformer conditioned on text or melody [27]. That architecture illustrates how sequence modeling can operate beyond text. A working music application still needs asset storage, playback, arrangement, export, and review.

Build a production workflow

Keep lyrics, style brief, arrangement notes, generated takes, and edited versions separate. Give each take an identifier. Record which inputs and model produced it. A chorus regeneration can become a new take without destroying the prior version.

If the service exposes stems, preserve their alignment and format. Do not claim stem availability when the selected model returns only a mixed track. A separate source-separation operation is another model and another possible source of artifacts.

The optional MusicGen recipe in Appendix C follows the project's documented API [28]. Code licensing and model-weight licensing must be checked separately. The cited project distributes different kinds of artifacts under different terms; an installable library does not establish commercial rights to every associated weight or output.

Training a music model

Choose between symbolic sequences, spectrogram representations, codec tokens, or direct audio methods based on the task. Define timing resolution and conditioning. Split evaluation material in a way that limits leakage from closely related tracks or recordings. Review dataset permissions and the risk of memorization.

A small symbolic model can learn local note patterns and demonstrate sequence generation. High-quality long-form audio adds demanding representation, dataset,

compute, and evaluation requirements. The language-model chapter supplies a reusable training lifecycle, while a music model needs its own objective and audio representation.

Evaluate the finished track

Listen for arrangement, continuity, clipping, vocal intelligibility, timing, and unwanted artifacts. Measure duration and format separately from musical usefulness. Automated similarity or quality scores can support review but cannot fully replace listening for the intended production.

For practice, write a 30-second instrumental brief with an explicit opening, development, and ending. Generate or supply a take, then create an asset manifest and review notes. Do not advance it to release merely because the file exists.

Chapter 23 Video and Movie Production

A film is an assembled work

A movie pipeline begins with the intended story and delivery format. Develop a script, shot list, storyboard, asset plan, and timeline. Generated clips are ingredients. The finished film depends on their selection, sequence, audio, continuity, and export.

Video generation may condition on text, an image, previous frames, or other controls. Maintaining a character or environment across shots is a separate challenge from producing one convincing clip. Record reference assets and continuity requirements before generating each shot.

Build a shot record

Each shot should identify its narrative purpose, duration, camera framing, subject, action, references, dialogue, sound, and technical output requirements. Store alternatives rather than pretending that the first take must become final. A chosen take should link to the timeline placement where it is used.

The timeline needs explicit start times and durations. Plan transitions and leave room for dialogue. A track that exceeds a shot's duration may require a longer shot, revised speech, or a different edit. These are production decisions the application should expose.

FFmpeg documents encoding, filtering, and media conversion tools useful for export [29]. The recipe in Appendix C shows an elementary still-image-and-audio export. That is a production fixture, not an AI-generated motion scene. A video model can provide the shot asset once its adapter has been validated.

Coordinate generated media

Generate or supply character references, backgrounds, dialogue, music, and effects. Resolve visual identity and style conflicts before final export. Regenerating a shot may invalidate earlier alignment work, so track which timeline entries depend on it.

Apply publication permission to the final package. A permission to generate a private test does not necessarily permit public distribution. Keep the applicable input permissions and output-use conditions with the release record.

Evaluate end to end

Watch the full exported film, not just the editor preview. Check missing frames, black gaps, audio drift, abrupt level changes, unreadable captions, and incorrect duration. Review continuity and story separately from technical export validity.

For practice, build a three-shot scene with a beginning, change, and resolution. Use supplied stills if motion generation is unavailable. Produce the timeline, export, and review record. Then replace one asset and identify every dependent timing or continuity check that must run again.

Chapter 24 A Unified Creative Workspace

Coordinate jobs and assets

A creative workspace connects writing, images, audio, music, and video through a shared asset and job model. It does not require every modality to use the same provider. It does require each output to retain its origin and the workflow to respect dependencies.

Separate a job from an asset. A job is an attempted operation with status and parameters. An asset is a retained result with format, content hash, provenance, and use conditions. A failed job may produce no asset or a partial asset that needs explicit treatment.

Build the state machine

Use states such as draft, queued, running, succeeded, failed, canceled, review required, and approved. Define allowed transitions. A canceled job must not silently become an approved result when a late callback arrives.

The companion's `media_job` creates the draft record. Add a provider adapter that returns an output asset and an execution receipt. Keep the approval event separate. For a multi-step production, maintain a dependency graph so a changed source invalidates only the affected derived work.

Reserve estimated resources before admitting expensive work. Reconcile actual usage when available. Concurrency limits should apply across modalities, since many image or video jobs can exceed a budget while the text interface looks idle.

Preserve editorial control

Expose versions, alternatives, review notes, and release destinations. A workspace should allow the user to select a take and override a proposed edit. Model recommendations remain proposals. The actual selected artifact and approval should come from a recorded application action.

Make exports portable. Include media files, a manifest, timing data, and readable notes. Provider links can expire or require authentication. A retained project should not depend solely on a temporary generation URL.

Evaluate workflow integrity

Test duplicate callbacks, cancellation during generation, output download failure, unsupported format, missing dependency, and budget exhaustion. These tests can use fixtures. Quality evaluation of actual models remains a separate exercise.

For practice, create a project containing a draft script, image, soundtrack, and timeline. Change the script and determine which assets require review. Then inspect whether the final export references the intended versions rather than whichever file happened to have the latest timestamp.

Part V Agents and Task Automation

A practical path through the components and their evidence.

Chapter 25 Tools as Typed Capabilities

A tool can change the world

A model output becomes consequential when software uses it to read private data, write a file, send a message, change a database, or operate equipment. Define each tool as a capability with explicit inputs, outputs, allowed resources, side effects, and failure conditions.

Separate read and write operations. A tool that lists files should not also delete them through an obscure option. A tool that drafts a message should not send it automatically unless the application has an explicit authorization rule for that workflow. Small interfaces make scope easier to assess.

Build a capability record

Specify the capability name and version, input schema, resource scope, actor, maximum work, timeout behavior, and receipt format. Validate arguments before execution. A successful invocation should return evidence of the outcome appropriate to the action.

For a file write, the receipt can identify the resolved path and content hash. For a remote send, a service receipt may establish acceptance by the service while delivery remains a separate fact. Choose outcome names that match what was actually observed.

The companion's `Gate.execute_write` accepts an actor, relative path, text, grant, request ID, and time. It resolves the path, checks scope and grant, rejects inconsistent reuse of a request ID, attempts an exclusive file creation, and records the result. No model text directly dispatches a shell command.

Protocols do not grant authority

A protocol such as MCP can standardize how applications discover and call tools [31]. A common interface improves interoperability; authorization still belongs to the application's identity, resource, and policy design. A tool description supplied by a remote server should be inspected as external content.

Choose the smallest capability set for the task. A research assistant may need document retrieval and draft export. Giving it unrestricted shell access adds many possible effects that do not help answer a handbook question.

Evaluate the boundary

Test malformed arguments, unsupported actions, an unknown actor, an expired grant, a wrong destination, and resource escape. Check that denial occurs before the effect. A logged denial after a file was created is too late.

The companion is a single-process teaching design. It does not defend against a hostile process replacing directories between checks or against an attacker controlling its authority object. Chapter 39 develops the stronger deployment boundary required for those threats.

For practice, describe a “save draft” capability and a “publish draft” capability. Give them distinct grants and receipts. Explain why possession of the first grant cannot be treated as possession of the second.

Chapter 26 A Bounded Task Agent

Define the loop

A task agent repeatedly selects a proposed next step, receives an execution result, and updates its task state. The proposal may come from a model, a rule, or a planner. The executor remains responsible for checking whether the proposed capability is permitted.

Represent the task with goal, permitted resources, budgets, state, and completion conditions. A vague instruction such as “keep working until everything is perfect” has no dependable stopping rule. Define what observation counts as completion and what conditions require stopping.

Build the state transitions

The companion's `bounded_agent` runs a proposal function and an executor with a maximum step count and a cooperative elapsed-time budget. A finish proposal ends the loop. A denied or failed result stops it. Reaching the step count returns a distinct limit result.

The elapsed-time check occurs between function calls. It cannot interrupt a proposer or executor that blocks indefinitely. A production boundary needs underlying request deadlines, cancellation support, or an isolated worker that can be terminated safely. Record that limitation instead of claiming a hard wall-clock guarantee from a loop check.

Use structured observations. A tool receipt should describe what happened, not only a free-form explanation. The planner can then decide from an explicit denied, failed, succeeded, or unresolved state. Preserve partial progress so a stopped task can be inspected.

Bound resources and effects

Add limits for generated tokens, tool calls, downloads, file sizes, external charges, and retries when they matter. A step budget alone does not constrain one expensive action. Check the applicable budget before execution and reconcile actual usage afterward.

Treat a new authority requirement as a pause or denied state. A model proposing a broader action does not broaden the grant. A user clarification may change the task, but the application should record the change and reevaluate scope.

Evaluate completion honestly

Test an agent that finishes immediately, one that repeats forever, one whose first action is denied, and one that stops after partial progress. Check that the user-facing status matches the stored history.

For practice, create a two-step task that reads a permitted source and writes a draft. Define the completion evidence before running it. Then remove the write grant and verify that the agent reports the task as incomplete rather than claiming success from the generated draft alone.

Chapter 27 Browser Files and Application Automation

Automate a defined workflow

An automation bot needs a task and a permitted surface. File operations, structured service APIs, browser interaction, and desktop control have different reliability and authority characteristics. Prefer a supported structured operation when it exposes the required action and result clearly.

For browser work, identify the site, account context, target resource, and action. A tab title or visible name may be insufficient to resolve a destination. Before a consequential write, check the actual resource identifier or verified page state.

Playwright documents browser automation and locator-based interaction [32]. Locators can make scripts less dependent on screen coordinates, but page structure, dynamic state, and account changes still need handling. The book's automation design is independent of any single browser library.

Build a supervised path

Break a workflow into read state, propose change, validate destination, authorize action, execute, and inspect the result. Store a preview when the user must review content or destination. The application should associate approval with the exact proposed change rather than an unspecified future operation.

For files, resolve a resource inside a permitted root and validate file type, size, and overwrite policy. The companion's writer creates a new file exclusively. It will report failure if that file already exists; it does not silently overwrite unrelated work.

For application APIs, use stable identifiers when available. Reading a record by display name and writing the first matching record can select the wrong target. Ambiguous resolution is a valid reason to stop and seek clarification.

Handle uncertain effects

After a timeout, determine whether the action occurred before retrying. A browser click may have submitted a form even if the confirmation page did not load. A retry can create a duplicate. Look for an operation receipt or inspect target state within the permitted scope.

Separate credentials and session material from model context. A model needs task-relevant observations, not raw account secrets. Restrict screenshots and page text when they can include unrelated private information.

Test a changed page layout, missing target, duplicate name, expired session, delayed confirmation, and denied write. Record which checks are simulated and which were exercised against a real application.

For practice, design a bot that organizes files by an approved plan. Generate a proposed move list first, then validate every source and destination. Explain how the bot would recover if one move failed after earlier moves succeeded.

Chapter 28 Multi Agent Coordination

Coordination is an engineering choice

A multi-agent system assigns work to several workers or roles. A researcher can collect sources, a writer can draft, and a reviewer can assess claims. Those roles may use the same model or different ones. Role separation can help organize work, but it does not by itself create independent evidence or better results.

Several models can repeat the same mistake from the same source. Treat agreement as an observation about their outputs. Independent corroboration depends on the evidence's origins, not the count of agents that endorse a sentence.

Define ownership and handoffs

Give each task an ID, owner, inputs, permitted resources, output schema, budget, and completion condition. A handoff should include the relevant artifact and unresolved questions. Do not require a downstream worker to reconstruct context from a vague success message.

Use shared state carefully. Concurrent writes can overwrite decisions or duplicate actions. Choose a coordination store and a policy for claims, leases, revisions, and conflict resolution. A central coordinator may simplify ownership while becoming an availability dependency.

Keep execution authority narrow. A research worker might read sources but lack publication permission. A writer can create a proposed draft. A separate executor can validate and perform the approved publication. A worker's task assignment is not necessarily a grant to every available tool.

Preserve disagreement

Collect dissenting results with their supporting records. A majority vote can be appropriate for a defined classification aggregation, but it must not erase a minority source that changes the answer. In AETERNA terms, interpretation cannot upgrade evidence merely by consensus.

Evaluate coordination against a single-worker baseline using the same task set. Measure quality, cost, latency, duplicate effort, and recovery from failed handoffs. More workers can increase overhead or dilute context. Use the observed result to decide whether the architecture helps.

Build a small coordination lab

For a document task, assign retrieval, drafting, and source review as three explicit stages. They can begin as ordinary functions with fixtures. Once their contracts work, substitute model adapters and retain the same evidence records.

For practice, provide two conflicting sources and require the reviewer to preserve the disagreement. Then stop one worker midway through its task. Check whether the coordinator can identify incomplete work and reassign it without duplicating a previously completed external action.

Chapter 29 Coding Agents and Sandboxes

Code generation needs an execution boundary

A coding agent can propose source changes, run checks, and interpret failures. Generated code may be useful and may also be wrong, destructive, or unexpectedly dependent on a service. Treat the code as a proposed artifact that needs review and appropriately isolated execution.

A sandbox constrains a process's resources and permissions. Useful dimensions include filesystem access, network access, credentials, process lifetime, memory, and compute. A temporary directory alone does not establish a complete sandbox.

Build a constrained workflow

Give the agent a task-specific checkout or working copy. Define permitted files and commands. Inspect changes before applying them to a shared release. Preserve a diff and the associated checks so another reviewer can understand what changed.

Use meaningful tests. A test that merely repeats the implementation's logic can confirm the same mistake twice. Boundary behavior, independently calculated examples, and known invariants are often more informative. The tiny model's finite-difference check is one example of using a separate numerical method.

Avoid giving the execution worker unrelated account credentials. A generated test that needs a private key should use a fixture unless live integration is part of the authorized task. Review network requests and dependency installation because they can execute code and transmit data.

Handle failures productively

Distinguish syntax failure, dependency failure, a test regression, and an invalid test expectation. Diagnose from observable output. A model should not delete or weaken a meaningful test merely to make a status green.

The companion tests cover selected grant, path, audit, memory, routing, agent, and model behaviors. They are a readable starting point. Their passing result does not establish security against all hostile conditions or correctness of an untested integration.

Review the result

Before release, check that the proposed change meets the task, preserves relevant behavior, and includes a truthful validation record. Separate a simulated provider response from an actual service call. Separate a training experiment from a model-quality claim.

For practice, ask a coding assistant to add an operation to the teaching gate. Review whether the addition bypasses scope checks, changes overwrite behavior, or introduces an unbounded call. Design at least one boundary test from the requirement rather than copying the generated function.

Chapter 30 Automation Under Failure

Partial completion is normal

A workflow can complete one step and fail the next. A network request can succeed at the server while the client loses the receipt. An event can arrive twice. A process can stop after an external effect and before saving local state. Design for these cases before calling a bot reliable.

Idempotency means repeated application of a defined operation has the same relevant effect as one application. A request ID can help implement it, but only if the service stores enough information to detect duplicates and mismatched reuse. A random ID in a log alone does not prevent duplicate effects.

Build receipts and recovery

The teaching gate stores a request fingerprint and result in memory. Matching repeats can return the stored receipt, while a reused ID with different inputs is denied. That behavior survives only for the life of this process. Durable idempotency requires storage and recovery coordinated with the actual effect.

A database transaction can coordinate local records within its scope. It cannot automatically include an unrelated remote service. For external actions, use a service-supported idempotency mechanism where available and an application recovery strategy that checks uncertain outcomes.

Compensation attempts to undo or offset completed work. It is not always a true inverse. A deleted file may be restorable from backup; a message already read cannot be unread. Name the actual recovery action and its limitations.

Keep hostile content as data

Retrieved pages, attachments, tool descriptions, and application messages can contain instructions directed at the agent. Preserve their origin and prevent them from altering permission records. Instruction hierarchy helps generation; execution gates still enforce capability scope [9].

Queues need retry limits, dead-letter handling, and observable status. A poisoned job that repeatedly fails should not consume the entire worker pool. Cancellation needs to reach the queued and active work and account for effects that already occurred.

Evaluate interruption points

Choose a workflow with two effects and interrupt it before, during, and after each step. Inspect stored state and target resources. Define whether recovery resumes, retries, compensates, or requires review.

For practice, simulate a successful file write followed by a lost response. Restart the application with no in-memory receipts. Identify why exclusive file creation prevents overwrite but still does not reconstruct a complete success receipt. Then design the durable record needed to resolve the uncertainty.

Part VI Building and Training Models

A practical path through the components and their evidence.

Chapter 31 Specify the Model and Its Data

Choose the learning problem

Training a language model begins with an objective, representation, dataset, and evaluation plan. For the small model here, the objective is next-character prediction. Given preceding characters, the model assigns probabilities to the next character. This is a complete language-model objective at a deliberately small scale.

The companion constructs a restricted corpus of sentences about readers, builders, models, gates, sources, and records. Subjects, verbs, and objects are combined, then complete sentences are assigned to training and validation. The final vocabulary has 24 characters. It is determined from the fixed synthetic corpus before the run; this shared vocabulary is disclosed in the report.

The sentences are original teaching material. They avoid an external dataset download and make the task reproducible. Their simplicity also limits what the result can establish. Held-out combinations still share vocabulary and grammar with the training material. The experiment does not measure general writing ability or factual knowledge.

Write the training manifest

Record corpus generation, split seed, tokenizer, model configuration, optimizer, sample procedure, run length, and software environment. Include source hashes for the implementation and dataset. Preserve the training and validation text so later readers can inspect exactly what the model saw.

Our default configuration uses dimension 24, three attention heads, two transformer blocks, and a 16-character context. Batches contain four windows. The model uses learned character and position embeddings, normalization, causal self-attention, feedforward layers, and an output projection. It has 10,992 trainable parameters.

Prepare an honest evaluation

Evaluate with fixed validation windows to make the before-and-after comparison meaningful. Do not change the held-out selection after seeing disappointing results. Retain generated samples, not just the favorable loss statistic.

A separate test set would be needed for a stronger final assessment after repeated tuning. This companion run uses validation for a teaching comparison and does not

claim an independent final test. Make that distinction explicit when reporting the result.

Build the dataset path

Read `corpus`, `batch`, and `evaluate` in `tiny_lm.py`. Token windows for inputs and targets are offset by one character. Check that the target at each position is the next character and that no validation sentence is included in the training string.

For practice, replace the synthetic corpus with another small dataset whose use you can justify. Write the manifest first. Decide how to split it based on document, author, or time relationships, then preserve the resulting artifacts before training.

Chapter 32 Build a Small Language Model

Assemble the network

The small transformer receives token IDs, looks up character embeddings, adds positional embeddings, applies two causal transformer blocks, normalizes the result, and projects it to vocabulary scores. These scores are called logits. Softmax turns logits into a probability distribution when training or sampling needs one.

The transformer uses attention to combine information across permitted sequence positions [1]. Our decoder-only design differs from the original paper's encoder-decoder architecture. Its causal mask supports next-token generation rather than access to future positions.

Read the implementation by shape

Input IDs have shape `(B, T)`. Embedding lookup produces `(B, T, D)`. Each attention head has depth `D/H`. The attention weights have shape `(B, H, T, T)`. After mixing, the representation returns to `(B, T, D)`. The output has shape `(B, T, V)`, where V is vocabulary size.

Each block first normalizes its input and computes query, key, and value projections. It scales the query-key scores, adds the causal mask, and applies softmax. Weighted values are concatenated and projected. A residual adds the update to the incoming representation. A second normalized feedforward update receives another residual.

The teaching model uses ReLU in the feedforward path and omits dropout. Other architectures make different choices. Simplicity helps inspect the implementation; it does not establish that these choices are optimal for a larger task.

Automatic differentiation

The `Tensor` class carries numerical values, accumulated gradients, parent dependencies, and a local backward function. Operations construct a computation graph. The final loss triggers a reverse pass. This implementation uses float64 to simplify numerical comparisons and runs on the CPU.

Study broadcasting, matrix multiplication, normalization, and softmax carefully. A gradient can have the correct shape while containing the wrong values. The finite-difference tests check selected coordinates in the model's actual graph.

Establish construction correctness

Check vocabulary bounds, context bounds, dimension divisibility, output shape, finite values, and causal behavior. The companion changes future tokens and confirms that earlier logits remain unchanged. It also compares selected gradients with an independent numerical estimate.

These checks establish limited mechanical properties. They do not establish useful language quality. For practice, build a one-layer variant and verify the same invariants before comparing its learning behavior with the two-layer configuration.

Chapter 33 Train Save and Resume

Optimize the next-token objective

Cross-entropy penalizes placing too little probability on the target token. The implementation computes stable log probabilities by subtracting the maximum score before exponentiation. Its backward function adjusts the predicted probabilities by the target indicator and averages over the batch and sequence positions.

The optimizer updates parameters from gradients. The companion uses Adam-style moving averages with bias correction and clips the combined gradient norm before updating. This stabilizes the small experiment without establishing a general recipe for every model.

Run the complete training command:

Command or pseudocode

```
python tiny_lm.py --steps 160 --out my_run
```

The command creates training text, validation text, a checkpoint, a sample, and a JSON report. It evaluates before and after training on the same fixed validation procedure. The expected artifacts matter more than exact timing on another computer.

What this edition measured

In the recorded run, validation loss decreased from 3.2355 to 1.1277 over 160 optimizer steps. Reloading the saved weights produced a maximum logit difference of 0.0 on the recorded probe. The implementation used NumPy 2.3.5 in Python 3.12. The sample still contains garbled text and incomplete words.

Those results establish a working training and reload path for this restricted experiment. They do not establish a useful assistant. A decreasing loss means the objective improved on the evaluated distribution; the distribution and objective remain part of the claim.

Save enough state to resume

A weight file is enough for some inference tasks. Continuing training also needs optimizer moments, update count, configuration, tokenizer, and relevant random state. The companion checkpoint stores those fields in an NPZ archive with JSON metadata and uses loading without pickle.

Resume with:

Command or pseudocode

```
python tiny_lm.py --steps 40 --resume my_run/checkpoint.npz --out resumed_run
```

The reported step count is cumulative. The resume test compares the next update in the original process with the next update from a reloaded checkpoint. It checks selected state continuity in the same numerical environment. Exact cross-platform reproducibility requires separate assessment.

For practice, interrupt after a checkpoint, resume, and compare weights against an uninterrupted run with the same data and random-state sequence. Then change one configuration field and identify why that run should receive a new experiment identity.

Chapter 34 Generate and Serve

Sampling turns scores into a sequence

At inference, the model computes logits for the final position in the current context. A decoding procedure chooses the next token, appends it, and repeats. Greedy selection picks the highest-scoring token. Sampling draws from a probability distribution. Temperature adjusts that distribution's concentration.

The companion divides logits by a positive temperature, subtracts their maximum, applies exponentiation, normalizes, and samples with a recorded random generator. It keeps only the most recent configured context for each step. This context cropping is part of the inference method.

Decode with an explicit boundary

The character tokenizer can encode only its known vocabulary. The sample function rejects an empty prompt or unknown characters rather than silently mapping them to arbitrary IDs. A general application needs an appropriate tokenizer and clear treatment of unsupported input.

Define stop conditions before exposing an endpoint. These can include token count, explicit stop tokens, cancellation, and deadline. A model that generates endlessly is an application failure even if every individual forward pass is valid.

Inspect the saved `sample.txt`. It demonstrates that the network generates text after training, while also showing its limitations. Improving the result would require a larger and more appropriate training study, not a stronger claim about the existing sample.

Serve the model

A serving process loads one checkpoint, validates requests, tokenizes input, runs bounded inference, decodes output, and returns a receipt. Keep the model version in the response metadata. Loading a checkpoint for every request creates unnecessary startup work and can complicate concurrency.

The companion's `serve_tiny.py` exposes a loopback inference endpoint for the trained example. It constrains prompt length and output count. It is a teaching service with a single-process request path, not an authenticated public deployment.

Evaluate inference behavior

Check known prompts, unsupported characters, count limits, context cropping, repeated seeds, and model load failure. Measure request duration in the intended environment. Repeated output under a fixed seed is useful for controlled comparison, while exact determinism still depends on the implementation and numerical environment.

For practice, generate the same prompt at three positive temperatures. Compare repetition and local coherence without treating either property as factual reliability. Then write a serving contract that distinguishes a successful generation from a useful answer to a real task.

Chapter 35 Fine Tuning and Efficient Adaptation

Choose the change you need

Retrieval supplies material at use time. Fine-tuning updates model parameters. Distillation trains a student from a teacher-derived signal. Quantization changes numerical representation for storage or execution. These methods solve different problems and can be combined with separate evaluations.

Full fine-tuning can update all selected model parameters. Parameter-efficient methods limit the trainable part. LoRA freezes the original weights and learns low-rank update factors [30]. A typical linear update can be written as `W_new = W_base + B @ A`, with the matrix dimensions and scaling defined by the implementation.

Instruction tuning trains on examples of desired interaction. Preference methods use comparisons or reward information. Direct Preference Optimization is a research example of learning from preference pairs without an explicit separate reward-model training stage in its formulation [33]. A preference objective still reflects the examples and assumptions supplied to it.

Build an adaptation study

Define the baseline, target task, permitted dataset, adaptation method, resource budget, and evaluation set. Measure the task of interest and behaviors that should be preserved. A model can improve on one format while becoming less useful elsewhere.

For the tiny model, a simple experiment can update only the output matrix. Use a fresh optimizer and zero or exclude gradients for frozen parameters. Verify that the frozen weights actually remain unchanged. Reusing old optimizer moments can otherwise complicate an apparently frozen update.

The companion's `adapt.py` demonstrates output-only adaptation on a second restricted synthetic text. Its report retains before-and-after losses and a frozen-weight check. This is a teaching adaptation experiment, not a LoRA implementation or a claim of general task specialization.

Preserve lineage

An adapted model needs the base model ID, adaptation data manifest, training configuration, output weights or adapter, tokenizer, and evaluation record. When an adapter depends on a particular base version, retain that dependency. Loading it against unrelated weights can invalidate the intended behavior.

Compare alternatives

If a handbook changes frequently, retrieval may provide a simpler update path than retraining. If the issue is repeated structured behavior, adaptation may be worth studying. If memory is insufficient, quantization may help deployment but still require quality evaluation.

For practice, list three reasons an adaptation could appear to improve while the actual task does not. Include evaluation leakage and a mismatch between loss and user outcome. Then design the paired baseline comparison that would expose those problems.

Chapter 36 Scaling and Other Training Families

Scaling changes the engineering problem

Larger training runs require dependable data pipelines, distributed execution, resource scheduling, checkpoint storage, and failure recovery. The small model's lifecycle remains recognizable, but its Python arrays and simple loop are not a substitute for large-scale infrastructure.

Data parallelism uses replicas on different batches and coordinates updates. Model or tensor parallelism divides computation or parameters across devices. Pipeline parallelism assigns stages to different workers. These schemes have different communication and utilization costs. PyTorch's distributed documentation describes supported mechanisms and their execution requirements [34].

Preserve the experiment

Record data ordering, preprocessing, code, model configuration, optimizer, global update count, and relevant random state. A resumed distributed run may have additional worker and sampler state. Checkpoints need integrity checks and retention rules so a large run can recover from partial storage failure.

Evaluate data throughput separately from model throughput. An accelerator can wait idle on input work. Repeated bad records can poison the stream. A pipeline should report failures and preserve enough context to diagnose them without exposing unrelated private data.

Other model families

A vision model may learn from images and labels, masks, paired text, or a generative objective. A recommender may learn ranking or interaction representations. A graph model combines neighborhood information according to its architecture. A reinforcement-learning system needs an environment, observations, actions, rewards, and a policy or value-learning method [6].

Mixture-of-experts designs route work among parameter groups. Retrieval-augmented systems connect learned computation with external records. Federated learning coordinates updates across data holders, while privacy and security still depend on the specific protocol and threat model. These are architectural patterns requiring their own evidence, not universal guarantees of efficiency or sovereignty.

Extend by a measured requirement

Choose the additional family because the task needs it. An industrial vision task requires appropriate sensors, image data, labels, and domain validation. A robot requires control, safety boundaries, and physical-world evaluation beyond a language interface. AETERNA can organize evidence and permission around these systems, but it cannot replace their domain engineering.

For practice, redesign P13 as a small vision experiment. State what changes in the representation, objective, evaluation, and output contract. Then identify which lifecycle pieces remain the same: data provenance, checkpointing, measurement, and truthful release claims.

Part VII AETERNA Evidence Authority and Memory

A practical path through the components and their evidence.

Chapter 37 Evidence and Interpretation

Keep the source and the conclusion separate

Evidence is material retained to support or challenge a claim. Interpretation is a proposed account of what that material means. A model can transform, summarize, compare, or hypothesize about evidence. That activity does not create a new independent observation of the original event.

AETERNA organizes this distinction as an architectural rule. Evidence records carry origin, content, provenance, and relevant uncertainty. Interpretations carry links to the evidence they use, the proposed conclusion, and unresolved alternatives. Authority records belong to a separate responsibility.

For a handbook question, the retained policy passage is source material. A model's sentence describing it is interpretation. A hash of the passage is an integrity aid. A grant to save a draft is authority. Each record answers a different question.

Build an evidence envelope

Record source ID, origin, collection method, retained content or reference, content hash, time information, access scope, and epistemic status. Distinguish a user report from a service receipt and a directly inspected document. A claimed timestamp and the time of collection can differ.

Link derived records to their parents. A quotation extracted from a document, a summary of the quotation, and a memory of the summary share ancestry. Their existence does not multiply the number of independent sources.

Independence is a claim about origin and relevant influence. Two websites reproducing the same press release are not necessarily two independent confirmations. A provenance graph can expose that relationship, but correctly identifying the relationship still requires source assessment.

Preserve unresolved states

Use an unresolved status when material is absent, conflicting, ambiguous, or insufficient for the requested conclusion. Do not convert uncertainty into an authoritative answer because the interface wants a single string.

Preserve minority evidence that bears on the question. A consensus of model outputs cannot erase a conflicting source. A disagreement record should identify the claims, evidence, and reason the conflict remains unresolved.

Evaluate the separation

Create a source, a model interpretation, and a stored memory of that interpretation. Ask the system to count independent evidence. The correct result depends on the originating source, not the number of retained descendants.

The companion's evidence type and source-only retrieval implement a narrow teaching boundary. They do not automatically assess real-world truth or source independence. For practice, extend the record schema with parent IDs and write a test showing that duplicated descendants retain one origin.

Chapter 38 VERITAS and DISCERNMENT

Assign distinct responsibilities

VERITAS is the evidence-handling responsibility in the AETERNA design. It retains source material, defines representations, computes integrity information, and records provenance. DISCERNMENT interprets that material while preserving its evidentiary limits.

The separation must appear in interfaces. A DISCERNMENT output should reference existing evidence records and state proposed conclusions. It should not write its own output into the primary-source store under a fabricated collection method.

Define canonicalization precisely

A content hash depends on exact bytes. JSON objects with the same apparent fields can serialize differently because of ordering, whitespace, number formatting, or character representation. Define the representation used for the hash and preserve its version.

The companion's `canonical` function sorts keys, uses compact separators, encodes UTF-8, and rejects non-finite JSON numbers. It is a teaching convention, not an implementation claim about every canonical-JSON standard. Its limits must be understood before interoperability claims are made.

Python's hashing library exposes SHA-256 [36]. The hash lets you compare retained byte representations. A matching hash does not establish the source's truth, authorship, or collection authority. Those need separate evidence or authentication.

Build the interpretation contract

A useful interpretation includes claim ID, text, supporting evidence IDs, conflicting evidence IDs, assumptions, uncertainty, and proposed disposition. Confidence can describe a model's assessment but cannot upgrade a record's origin or grant permission.

If the source is weak or disputed, the interpretation should preserve that condition. A model can explain why it favors one reading while retaining alternatives. That preference remains interpretation unless an authorized process establishes a stronger status from additional evidence.

Test the firewall

Attempt to submit a model output as a primary document without a source collection record. Attempt to replace an evidence hash while preserving the same source version. Attempt to interpret a conflict as resolved without recording a basis. The interfaces should reject or explicitly mark each condition.

For practice, define two schemas and separate stores for evidence and interpretation. Give the interpretation writer read access to evidence IDs but no ability to mint primary origins. Then inspect the executor paths to ensure that this restriction survives actual implementation.

Chapter 39 AETHER and SENTINEL

Authority must come from a trusted process

AETHER represents authority in AETERNA. SENTINEL checks whether a proposed capability invocation fits that authority at the execution boundary. A model may propose an action and explain its expected usefulness. Permission comes from an authenticated actor and policy process.

Bind a grant to actor, action, resource, scope, validity period, and relevant constraints. Include revocation or one-time-use semantics when the task requires them. A generic “approved” string cannot communicate those details reliably.

Build the teaching grant

The companion's authority object signs a grant using HMAC with a secret key. HMAC authenticates a message under a shared secret [35]. The gate recomputes the signature and checks the actor, action, exact resolved resource, validity interval, and revocation set.

Changing the resource after signing makes the grant invalid. Using another actor fails the match. Expiry is checked at execution time. The tests exercise these cases and verify that the denied write creates no target file.

The signer is a trusted part of the teaching application. A model does not receive its key. In a stronger deployment, place grant issuance and execution behind process or service boundaries appropriate to the threat model. Shared-process object separation alone is insufficient against arbitrary code running with the same privileges.

Guard the actual resource

Resolve identifiers into permitted resources inside the executor. Check scope before the effect. Path validation needs special care in hostile filesystems: symlink replacement and concurrent directory changes can create time-of-check/time-of-use races. The companion assumes a trusted single-process environment and does not claim to solve those races.

For stronger enforcement, use operating-system access controls, trusted directory handles, safe relative operations, and isolated workers as appropriate. Network tools need destination restrictions and credential scope. Every alternate execution path must honor the gate.

Evaluate denial before success

Test absent, expired, revoked, tampered, wrong-actor, wrong-action, and wrong-resource grants. Inspect the real target state after denial. Then test allowed execution and verify the receipt.

For practice, design a grant that permits one message to one verified destination before a deadline. Describe how the executor checks the exact message and how replay is prevented. Do not treat a prior approval as an unlimited grant for future messages.

Chapter 40 MNEMOSYNE and the Golden Thread

Continuity needs explicit records

MNEMOSYNE represents memory in the AETERNA design. The Golden Thread represents continuity information such as identity references, persistent commitments, and the relationship among sessions. These responsibilities should be encoded in revisable records with visible provenance.

Continuity is useful when an assistant remembers a project, preserves an agreed constraint, or recognizes an ongoing task. It becomes unreliable when an earlier model interpretation silently becomes the authoritative account of the user or the world.

Build memory with origin and scope

Record the subject, content, origin, parent references, access scope, validity, revision, and current status. Separate a current preference from an event report and a model hypothesis. A user can authoritatively correct their preference while an external factual claim may still need source evidence.

The companion's memory demonstration preserves origin through revision. Its source-only retriever excludes model outputs. That narrow behavior shows how a software boundary can support non-corroboration. A richer implementation needs a provenance graph and policy that follow descendants across summaries and exports.

Keep memory from issuing authority

A memory that says “the user usually approves draft saves” is not a current resource-specific grant. Retrieve it as context if permitted, then check current authority through the appropriate process. Stored convenience must not override a revoked permission.

Likewise, a remembered successful action should link to the actual receipt. If the receipt is missing, the memory may preserve a report of success without establishing that the effect occurred. Mark that difference.

Make continuity auditable

Expose which records influenced a response and provide correction and removal mechanisms within the declared retention policy. Record the identity-specification version and relevant memory revisions when evaluating behavior across updates.

A DID-aware assistant must let the user govern internal identity context. Voice similarity does not identify an alter. Orientation support should follow explicit user needs and the defined privacy boundaries, as developed in Chapter 12.

For practice, create a continuity record for an ongoing book project. Correct a mistaken chapter status, preserve the revision, and ensure that a later response uses the corrected status. Then revoke a write grant and verify that continuity records cannot restore it.

Chapter 41 PHOENIX and the Observatory

Record decisions and outcomes

PHOENIX represents audit in AETERNA. The Observatory presents selected records for inspection through a read-only interface. An audit trail should connect request, evidence, interpretation, authority decision, attempted execution, and observed outcome.

Record failures and denials as well as successes. Missing events can distort the account of what a system attempted. An application that logs only successful outputs cannot support claims about how it handled refused actions.

Build a hash chain

The teaching audit stores a sequence number, previous record hash, event, and current hash. Verification recomputes the chain. Editing a retained event without updating its descendants causes verification to fail.

This is tamper evidence under a limited threat model. An attacker able to rewrite the entire chain can recompute it. Removing the end of a chain may go undetected without a trusted retained tip or external anchor. The companion's mutable in-memory list is not immutable storage.

Stronger designs may use authenticated signatures, independently retained checkpoints, append-only access controls, durable storage, and external verification. Each control needs a defined attacker model. The word “immutable” should name the protected operation and the party against whom it is protected.

Build a useful inspection surface

Show evidence records, interpretation links, grants, dispositions, execution receipts, and integrity status. Apply access controls to the view. Audit records themselves can contain private content, resource paths, and sensitive operational details.

Keep the Observatory read-only at its service boundary. Hiding an edit button while leaving a write endpoint available does not establish read-only operation. A separate write authority should be required for any administrative correction, and that correction should produce its own audit event.

Evaluate coverage and recovery

Test an edited event, a missing event, a truncated chain, a logger failure, and restoration from backup. Decide which failures are detectable in the actual design. A verified chain says the retained chain is consistent; it does not prove that every relevant real-world event was recorded.

For practice, anchor a chain tip in a separately controlled record, then truncate the local chain. Compare the anchored tip with the local result. Explain precisely what the additional anchor detects and what an attacker would need to defeat it.

Chapter 42 Deterministic Disposition and Acceptance

Turn policy into explicit decisions

A disposition is the result of applying rules to a proposed action and its context. Define the inputs and precedence so two implementations reach the same result. The model can supply an interpretation, but a deterministic policy function should decide from explicit records.

Distinguish authorized from executed. Authorization permits an attempt within scope. Execution can still fail. A succeeded receipt is a later observation, not an automatic consequence of an authorized disposition.

Define a teaching decision table

First applicable condition	Disposition	Next behavior
Malformed proposal	DENIED	Explain validation failure
Required policy unavailable	DENIED	Stop execution
Resource outside permitted scope	DENIED	Stop execution
Required grant invalid	DENIED	Stop execution
Required factual basis unresolved	UNRESOLVED	Seek evidence or clarification
All applicable conditions satisfied	AUTHORIZED	Permit a bounded execution attempt

This table is a reference design for the book. A concrete system must define which actions require factual evidence and which merely transform explicitly supplied user content. An evidence requirement should not be guessed after the action begins.

Build and test the function

The companion includes `disposition` as a pure function over Boolean conditions. Its tests cover precedence: a denied grant remains denied even if the evidence is complete; unresolved evidence cannot authorize an action; a fully satisfied request can be authorized.

The file gate implements a narrower subset focused on grant and resource checks. Its passing tests do not constitute acceptance of every AETERNA rule or deployment. A complete acceptance suite must exercise the actual integrated execution paths and their relevant threat model.

Preserve ambiguity and dissent

If evidence conflicts, retain the conflicting records and the unresolved result. If a model expresses high confidence, preserve it as an assessment rather than a grant or stronger source. If memory repeats the claim, preserve its origin rather than increasing support.

Acceptance includes minority-evidence protection, memory non-corroboration, manipulation resistance, and the authority firewall as architectural requirements. Each needs concrete cases and observable outcomes. A name or diagram alone cannot show that the boundary holds.

For practice, construct a case satisfying every condition except a valid grant. Verify that no effect occurs. Then construct a granted request with conflicting evidence and explain whether the action is denied, unresolved, or permitted by the specific policy you defined.

Part VIII Integrated Systems and Release

A practical path through the components and their evidence.

Chapter 43 An Evidence Grounded Research Assistant

Turn a question into a reviewable record

A research assistant needs a bounded collection, a question, and a way to retain the materials behind an answer. Start with five short documents whose origin you know. Preserve their content and identifiers before adding search. A polished paragraph is not a substitute for a retrievable source.

Use `Evidence.retain` and `extractive_answer` from the companion to build the first version. It returns matching excerpts with source IDs, or an unresolved result. It does not synthesize new conclusions. This modest baseline gives you something inspectable before a language model adds fluency.

Build the pipeline

Accept a question, apply collection access rules, retrieve candidates, retain the chosen evidence, and produce a draft. If using a model, pass excerpts as data with a request for claim-level source links. Then check every factual claim against its cited passage. Citation existence alone is insufficient: a real source can fail to support the attached claim.

Store question, corpus version, retrieval configuration, source IDs, draft, review, and final disposition. A source that changes after generation should not silently rewrite the evidence for an earlier answer. Preserve the version or snapshot used, subject to the collection's retention rules.

Handle uncertainty and conflicting sources

An answer may identify disagreement rather than settle it. Record publication date, event date, scope, and which source is closest to the underlying evidence. Distinguish a measurement from an author's interpretation. A search result snippet is a discovery aid; retrieve the relevant original material before relying on it.

Do not fill missing information with confident prose. If the collection cannot establish the requested date, status, or mechanism, the appropriate output is unresolved with a precise description of the gap. A model's prior knowledge can suggest where to search, but does not become retained evidence by being repeated.

Evaluate the project

Create answerable, unanswerable, conflicting, and adversarial questions. Measure whether relevant evidence was retrieved, whether citations support claims, whether uncertainty was preserved, and whether unauthorized documents remained excluded. Record latency and review effort alongside answer quality.

The local reference implementation covers retrieval and evidence records. A synthesis model, document parser, access-control service, and citation reviewer are integrations to build and evaluate separately. Your completion condition is a reviewed answer with traceable evidence, including an honest unresolved result when appropriate.

Chapter 44 A Governed Creative Studio

One brief many media paths

A creative studio turns a brief into text, images, speech, music, and a timeline. Begin with a short original scene, two still images, narration, a background cue, and a ten-second video plan. Keeping the project small makes asset lineage and review visible.

Create a job for each asset using `media_job`. Assign identifiers and retain the brief, references, model or tool version, generation settings, output location, and review state. A job record without a generated file is a draft job, not a completed creative asset.

Separate generation from release

Use a storyboard to keep characters, tone, duration, and scene order coherent. Generate alternatives, choose deliberately, and record selection. Reference assets may improve consistency, but they do not automatically establish permission to reproduce a person's likeness, a recording, or a protected work. Record the relevant permissions and usage conditions for your actual project.

The companion's WAV and timeline are deterministic fixtures. They let you test asset storage, duration, and assembly without a hosted account or model download. The waveform is a synthesized tone created by ordinary signal processing. It is not a learned music model or evidence of musical quality.

Replace one fixture at a time with a real provider or local model. Evaluate output dimensions, sample rate, duration, format, intelligibility, visual continuity, and alignment before combining assets. Preserve failed attempts when they help diagnose a reproducible issue.

Review at useful boundaries

Review narration text before speech synthesis, scene references before image generation, and assembled audio before export. Check whether the final result accurately represents factual claims in the brief. Creative invention can be labeled as invention; an invented factual testimonial cannot be quietly treated as evidence.

A publication grant should specify the destination and approved version. Regenerating an image or changing the soundtrack after approval changes the artifact under review. Bind authorization to the resulting artifact or explicitly defined revision scope.

Finish the studio project

The acceptance record should include the brief, selected assets, source and permission notes, timeline, export settings, rendered output, and reviewer decision. Compare the exported duration and dimensions against the plan. Listen and watch the complete result rather than checking only that a file exists.

The reference code provides jobs and fixtures. Chapters 20 through 24 and Appendix C describe external generation and assembly paths. Their unexecuted status remains visible until you run them in your own pinned environment. The studio becomes complete when those integrations meet the project's actual quality and release criteria.

Chapter 45 A Permissioned Operations Bot

Start with a narrow operation

An operations bot should begin with one useful, reversible action. This project writes a reviewed summary into a designated directory. Its actor, action, exact resource, expiration, and request ID are explicit. It cannot infer broader authority from the summary's contents.

Run `core.py --root my_output`. The demonstration constructs evidence, obtains a matching grant for one target, attempts a forbidden write, performs the permitted write, and records results. Use a new output directory for a fresh demonstration because the example deliberately avoids overwriting existing files.

Follow the action record

The model or planner proposes an action. A validator checks its shape. A policy component checks scope. The authority service validates the grant. Only then does the executor attempt the operation. Its receipt reports success or failure separately from the authorization decision.

For the teaching write, the grant binds an actor, the action `write`, and the resolved target path. The Gate checks that the target remains inside its designated root. Its request ID cache returns the earlier receipt for an identical repeat and rejects the same ID attached to changed input.

Understand the reference boundary

The companion runs in a trusted single process with an in-memory authority and receipt cache. It does not protect against a hostile process changing filesystem links between validation and opening a file. It does not survive restart with durable idempotency records or revocation history. These limits are part of the design, not minor implementation details.

A deployed executor needs operating-system isolation, durable operation state, appropriate atomic resource handling, and narrowly held credentials. For remote actions, a timeout can leave the outcome uncertain. Reconcile that outcome before retrying an operation that may create a duplicate external effect.

Evaluate denial and recovery

Attempt an expired grant, revoked grant, wrong actor, modified token, out-of-root path, reused request ID, and unavailable target. Confirm both the result and the external effect. A denied action must leave the target unchanged. A failed write must not be described as completed.

Next, add one read-only operation before adding another write. Give each operation its own argument schema and resource rules. The finished bot should demonstrate useful bounded work, visible denial, and recoverable failure. Increasing the number of tools is valuable only when you can preserve these properties across the new operations.

Chapter 46 A Local and Hybrid Personal Assistant

Give personal information a boundary

A personal assistant combines conversation, preferences, documents, schedules, and sometimes voice. Begin with local notes and explicit user-reported preferences. The companion Memory stores a value, origin, and revision. An inference about a preference is not interchangeable with a preference the person stated.

Use a local model runtime for private tasks if your environment supports it. Verify where inference, logs, backups, transcription, and search actually run. A local chat endpoint can coexist with a cloud transcription service; the word local on one component does not describe the whole pipeline.

Make routing reviewable

The reference route chooses local when available. If local is unavailable, a private task or a task without cloud permission remains unavailable. An allowed non-private task may use cloud. This simple function captures a useful invariant; a deployed router also needs provider constraints, cost ceilings, health checks, and an explanation of the chosen path.

Use task-specific grants. Permission to summarize a local note does not automatically authorize sending the note elsewhere, changing a calendar, or contacting another person. A reviewed draft and an executed external action should remain separate states.

Handle memory and identity carefully

Allow inspection, correction, export, and deletion of retained preferences. Decide how those operations affect derived summaries, embeddings, and backups. Deleting a visible row while continuing to retrieve an old derived copy is an incomplete deletion path.

In the Samara-style design discussed in this book, a voiceprint may offer evidence about a physical speaker; it does not establish an alter's identity. Ask for explicit orientation when that distinction matters. Unknown or uncertain speakers should not receive private memory merely because their voice resembles a known recording. The design discussion does not verify a particular Samara repository or deployed identity system.

Evaluate the personal assistant

Use scenarios for incorrect memory, memory correction, local outage, prohibited cloud fallback, unknown speaker, conflicting preferences, and cancellation of a proposed action. Measure what information crossed each boundary and what persisted afterward.

Finish with a simple inspectable interface: current route, retained memory, pending actions, recent receipts, and an available stop control. Add richer conversation only after those controls work. The objective is a helpful assistant whose behavior a person can understand and revise, including when it cannot complete a request.

Chapter 47 Reliability Security and Observability

Describe failure before the incident

Reliability begins with the boundaries a system promises to preserve. List external dependencies, deadlines, resource limits, authority paths, and data retention. A model's answer quality is one dimension; an executor that performs the wrong action is a different kind of failure.

Create an incident vocabulary that separates invalid input, unavailable evidence, denied authority, failed execution, uncertain execution, and completed execution. These distinctions help operators choose the next step. Treating every failure as a generic retry can turn an outage into duplicate actions.

Instrument the boundaries

Record a task ID, policy version, model configuration, route, evidence references, authorization decision, execution receipt, and elapsed time. Retain only the content needed for the defined operational purpose. A debug log that copies every prompt, secret, and document creates another sensitive data collection.

Use metrics to detect regressions: denied-action rate, unresolved-answer rate, execution failures, latency percentiles, queue age, resource consumption, and human review time. Interpret changes alongside workload composition. A higher denial rate may reflect stronger policy enforcement or a broken grant path; the number alone cannot decide.

Exercise recovery

Inject failures at retrieval, model response parsing, grant validation, storage, and external execution. Simulate cancellation and worker restart. Ensure partial results are marked as partial. Preserve enough state to reconcile an uncertain action without guessing that it succeeded or blindly repeating it.

Treat retrieved text, tool results, and media metadata as potentially hostile data. Prompt injection defenses need application boundaries in addition to model instructions [9]. A tool should not expand its permissions because a document tells it to ignore the policy.

The companion hash chain detects edits to retained records when verified against its current contents. It does not detect truncating the end or replacing the entire chain without a separately trusted anchor. A stronger operational audit requires

durable storage, independent anchoring, and access controls appropriate to the actual threat model.

Define readiness concretely

Choose measurable thresholds for the intended workload, along with owners and recovery procedures. Run tests for both desirable behavior and prohibited effects. Review representative outputs by people who understand the task. Keep records of what was tested, under which configuration, and what remains uncertain.

A release is ready when the evidence meets its declared criteria. A checklist can organize that evidence, but checked boxes do not create evidence. For this teaching edition, local numerical and boundary checks establish a useful starting point. Public, multi-tenant, or consequential deployment needs additional engineering and assessment.

Chapter 48 Releasing and Maintaining Your System

Release a defined configuration

Give the release an identity that binds code, configuration, model or provider selection, dependencies, policy, and evaluation record. Pin what can be pinned. For a hosted model whose internals you cannot freeze, preserve the identifier and date, record observed behavior, and plan for changes outside your control.

Package a setup guide, expected artifacts, known limitations, support path, and rollback procedure. A clean installation should reproduce the intended local demonstration without an unpublished file or secret from the author's machine. Use a new environment to reveal hidden dependencies.

Roll out in stages

Begin with internal review and bounded workloads. Compare the new release against the preceding release using the same evaluation cases. Keep human review where the consequences require it. Expand only when the evidence supports the expanded workload and authority scope.

Prepare rollback before rollout. Some data migrations cannot be reversed by replacing an executable. Preserve compatible state or a deliberate migration recovery path. External effects such as sent messages are not undone by restarting an older model.

Maintain the evidence

Monitor changes in input distribution, retrieval collections, tool APIs, model behavior, permissions, and user needs. Refresh evaluation cases without erasing historical comparisons. Record why a threshold, model, or policy changed and what was observed afterward.

When an incident reveals a new failure mode, write a regression case that checks the meaningful outcome. If an unauthorized write occurred, check the target and receipt, not just whether a warning appeared. Separate the immediate containment from the longer-term repair.

Continue beyond this book

The paths in this book combine into many systems: tutoring tools, analysis workbenches, accessibility interfaces, search assistants, creative studios, and supervised operations agents. Other domains need specialized perception, controls, datasets, and evaluation. Choose a task before choosing a larger model or a more elaborate agent structure.

The essential cycle remains: specify the task, preserve the data, build the components, evaluate the behavior, grant bounded authority, observe execution, and revise from evidence. AETERNA organizes the authority and evidence boundaries within that cycle. It cannot replace the engineering and domain judgment around them.

The companion demonstrates a small trained language model and a practical set of local system components. The remaining integrations are deliberately identified as recipes or designs. Your next release should make an equally clear distinction between what it intends, what it implements, and what it has established through testing.

Appendices

These references turn the chapter paths into concrete experiments. Start with the vocabulary and setup instructions, then use the provider recipes only when you have checked your actual environment. The worksheet, records, project matrix, and annotated algorithms support review and revision.

- A. Working Vocabulary
- B. Set Up and Run the Companion
- C. Provider and Media Recipes
- D. Hardware and Performance Worksheet
- E. Reusable Records and Acceptance Templates
- F. Projects Sources and Verification
- G. Reading the Core Algorithms

Appendix A Working Vocabulary

AI system. The model, runtime, application, data, tools, policies, interfaces, and operating environment that together perform a task.

Model. A parameterized computation or other learned representation. Its predictions depend on the input, parameters, and inference procedure.

Parameter. A numerical value used by the model. A trainable parameter is updated through the chosen learning procedure.

Training. Adjusting selected parameters from data under an objective. It differs from supplying context during inference.

Inference. Computing outputs with a configured model. Sampling and application routing can introduce behavior beyond the model's numerical forward pass.

Runtime. Software that executes model computation. A provider, model family, SDK, and runtime are distinct things even when packaged together.

Token. A unit in a model's representation, such as a character, subword, image patch, or audio code. Token meanings and boundaries depend on the tokenizer.

Embedding. A vector representation. Vector proximity can support retrieval or learning; proximity is not proof that two assertions are equivalent or true.

Logit. An unnormalized score. Softmax converts a vector of logits into probabilities under the specified calculation.

Attention. A learned mixing operation whose weights depend on representations such as queries and keys. A causal mask excludes future positions in next-token training.

Context. The information supplied for a particular inference. A context window bounds the sequence the model can process in a configured invocation.

Loss. The numerical training objective on a batch. An improved loss does not automatically establish improvement in a user's outcome.

Gradient. A derivative describing how a quantity changes locally with a parameter. Backpropagation computes gradients through a composition of operations.

Optimizer. The update rule and associated state used to change parameters. A resumable checkpoint may need that state in addition to weights.

Validation set. Data used for development comparison. Repeated tuning against it changes how strongly its results can support a final claim.

Test set. Data reserved for a stated final evaluation. Independence depends on the actual split, contamination, and development history.

Checkpoint. A saved configuration and state. Weight-only, inference, and resumable training checkpoints have different requirements.

Quantization. Representing numerical values at reduced precision or with a compressed scheme. Effective storage includes scales, metadata, and runtime details.

Fine-tuning. Further training selected model parameters on a new objective or dataset. Retrieval changes supplied context without requiring that weight update.

LoRA. A parameter-efficient adaptation method based on learned low-rank updates to selected weights [30]. It needs a compatible base model and evaluation.

Retrieval. Selecting records relevant to a query. Search relevance and evidentiary support are separate assessments.

RAG. Retrieval-augmented generation: retrieved material conditions a generator [3]. The application still needs source retention and support checks.

Agent. An application loop that proposes and performs steps toward a task, subject to its tools, limits, and authority. The label does not imply independent permission.

Capability. A defined operation on a resource. A discovered tool is a possible operation, not a grant to invoke it.

Grant. A permission record issued by an authenticated authority process, with scope, constraints, and validity. Its trust depends on issuer and enforcement.

Disposition. A policy decision such as denied, unresolved, or authorized. Authorization is distinct from a completed external effect.

Receipt. A record of an attempted execution and its observed outcome. A timeout can require reconciliation rather than a success or failure guess.

Evidence. Retained material with source and collection context that can support a specified claim. A generated assertion has a different origin from a primary observation.

Provenance. The origin and transformation history of a record or artifact. Multiple descendants of one source are not independent corroboration.

Integrity hash. A digest of a defined representation. Equality supports comparison of that representation; it does not prove truth, independence, or completeness.

Memory. Retained information for future use. Its origin, scope, revision, and deletion behavior should be visible. Memory is not an authority issuer.

Unresolved. A state in which the relevant evidence or interpretation does not establish the requested result. It should preserve what is missing or disputed.

Local. A component executes on the defined local environment. Verify other components and network activity before calling an entire workflow local.

Hybrid. A workflow uses local and remote components under an explicit routing policy. Automatic fallback is a data-movement decision.

AETERNA. The evidence-bound governance architecture used in this book. Its named responsibilities organize evidence, interpretation, authority, execution, memory, and inspection. This edition's code is a teaching reference.

Appendix B Set Up and Run the Companion

What you need

Use a computer with Python and enough storage for a virtual environment. The recorded environment used Python 3.12.14 and NumPy 2.3.5; see the companion manifest for the exact environment report. The local examples use CPU computation and do not require a GPU, provider subscription, or external model download. Installing dependencies may require network access.

Unzip the companion archive and enter the folder containing `README.md`. A terminal's working directory controls relative paths. If Python cannot find a script or checkpoint, check that directory before changing code.

Command or pseudocode

```
python -m venv .venv
```

On Windows, call `.venv\Scripts\python.exe` directly or activate the environment using the instructions in the README. On macOS and Linux, run `source .venv/bin/activate`. Some systems use `python3` rather than `python`.

Command or pseudocode

```
python -m pip install -r requirements.txt
python -m unittest discover -v
python classical.py
python core.py --root first_demo
```

The recorded suite ran 24 tests successfully. It includes selected numerical gradient comparisons, causal attention, checkpoint continuation, permission denial, memory revision, routing, the full boolean disposition table, adaptation freezing, and local HTTP requests. These checks have a declared teaching scope; they are not a production certification.

Use the chatbot

Command or pseudocode

```
python server.py
```

Open `http://127.0.0.1:8000` in a browser. Ask about equipment checkout or closing. A matching query returns source excerpts and IDs. An unmatched query returns unresolved. Ctrl+C stops the server. If the port is already in use, try `python server.py --port 8010` and use that port in the address.

The interface is extractive. It does not call a language model or invent new material. To build a generative version, preserve the request validation and response contract while connecting a separately configured backend. Record that change and evaluate generation independently.

Train and resume

Command or pseudocode

```
python tiny_lm.py --steps 160 --out my_run
python tiny_lm.py --steps 40 --resume my_run/checkpoint.npz --out resumed_run
```

Inspect `training_report.json`, `sample.txt`, `train.txt`, and `validation.txt`. The run directory also contains the checkpoint. A resumed run retains the tokenizer, parameters, optimizer moments, step count, and data random state. The command reports a cumulative step count.

Serve and adapt

Command or pseudocode

```
python serve_tiny.py --checkpoint my_run/checkpoint.npz
python adapt.py --checkpoint my_run/checkpoint.npz --out my_adaptation
python media_fixture.py --out my_media
```

The inference endpoint is `POST http://127.0.0.1:8001/generate`. A request body is:

JSON example

```
{"prompt":"the ","count":80,"temperature":0.8}
```

Send JSON with `Content-Type: application/json` using a local HTTP client. The service permits 1 to 256 generated characters, prompt length 1 to 256, and temperature 0.1 to 2. The model accepts only its 24 known characters. The response includes the checkpoint hash. Neither loopback service has public authentication or production isolation.

Troubleshooting by boundary

Symptom	Check	Next action
NumPy import fails	Correct virtual environment and interpreter	Install the pinned requirements in that environment
File not found	Working directory and checkpoint path	Use the intended directory or an explicit path
Port unavailable	Another process owns the port	Stop that process or select another local port
Prompt rejected	Known characters and limits	Use a permitted prompt and bounded request
Existing output write fails	Gate avoids overwriting	Use a new directory or deliberately manage the prior artifact
Garbled generated text	Tiny model and restricted data	Treat it as the measured result; design a new experiment
Provider unavailable	Account, model access, quota, schema	Keep the failure visible; preserve routing restrictions

Prerequisites to learn next

Python fundamentals: functions, classes, exceptions, dictionaries, arrays, context managers, and command-line arguments. Numerical fundamentals: vector and matrix shapes, dot products, probability normalization, derivatives, and gradient descent. Web fundamentals: JSON, HTTP methods, status codes, request bodies, and client/server separation.

Practice tracing one array shape and one request path before changing several components. Keep a copy of a working run. Write an expected output and failure condition for each change. The companion makes the first experiments inexpensive enough to inspect directly.

Appendix C Provider and Media Recipes

Status and interface date

These are integration recipes checked against the cited official documentation on October 1 2026. No live provider requests or external model downloads were executed for the companion. Model names are configuration supplied by the reader because availability, access, cost, and supported features vary. Recheck the documentation when implementing a recipe.

OpenAI, Anthropic, Google, xAI, Meta, and Ollama occupy different places in the stack. Hosted API compatibility does not mean identical capabilities or contracts. Meta's model repository supplies model-family information and licenses; it is not a single universal chat endpoint [13]. Ollama is a runtime and service interface for supported model packages [4].

Path	Request interface	Authentication or provisioning	Notes
OpenAI hosted text	`POST /v1/responses` on `api.openai.com`	Server-side bearer key	Parse output items; model is configured [15]
Anthropic hosted text	`POST /v1/messages` on `api.anthropic.com`	`x-api-key` plus API version	Supply `max_tokens`; parse content blocks [16]
Gemini hosted text	`POST /v1beta/interactions` on Google endpoint	`x-goog-api-key`	Current getting-started path; parse completed model-output steps [37]
Gemini content API	`models/{model}:generateContent`	Documented API key	Separate content/parts contract remains documented [17]
xAI hosted text	`POST /v1/responses` on `api.x.ai`	Server-side bearer key	Responses path shown in text guide [18]
Ollama local chat	`POST /api/chat` on loopback port 11434	Provision runtime and model	Select local model, set `stream:false` for this recipe [14]
Meta model locally	Model-specific weights and runtime	Download rights and license	Choose compatible runtime; verify the actual license [13]

A bounded JSON transport

This original example uses the Python standard library. It demonstrates one non-streaming request. It is not a complete retry, rate-limit, streaming, credential, or provider-error implementation. Choose a timeout and response-size ceiling for your workload. Keep keys on the server and out of browser source, logs, and version control.

Python excerpt

```
import json
from urllib.request import Request, urlopen

def post_json(url, headers, payload):
    body = json.dumps(payload, allow_nan=False).encode()
    request = Request(url, data=body, method="POST",
        headers={"Content-Type": "application/json", **headers})
    with urlopen(request, timeout=30) as response:
        raw = response.read(2_000_001)
    if len(raw) > 2_000_000:
        raise ValueError("response too large")
    return json.loads(raw)
```

The following blocks construct one request each. `key` and `model` come from configuration; the code does not print credentials. Validate configuration before sending. Treat transport errors as failures and inspect provider status before interpreting returned content as a completed answer.

OpenAI request [15]:

Python excerpt

```
response = post_json(
    "https://api.openai.com/v1/responses",
    {"Authorization": "Bearer " + key},
    {"model": model, "input": "Explain causal attention."})
```

Anthropic request [16]:

Python excerpt

```
response = post_json(
    "https://api.anthropic.com/v1/messages",
    {"x-api-key": key, "anthropic-version": "2023-06-01"},
    {"model": model, "max_tokens": 300,
     "messages": [{"role": "user",
                   "content": "Explain causal attention."}]})
```

Gemini Interactions request [37]:

Python excerpt

```
response = post_json(
    "https://generativelanguage.googleapis.com/v1beta/interactions",
    {"x-goog-api-key": key},
```

```
      {"model": model, "input": "Explain causal attention."})
```

xAI request [18]:

Python excerpt

```
  response = post_json(
      "https://api.x.ai/v1/responses",
      {"Authorization": "Bearer " + key},
      {"model": model,
       "input": [{"role": "user",
                  "content": "Explain causal attention."}],
       "store": False})
```

Ollama local request [14]:

Python excerpt

```
  response = post_json(
      "http://127.0.0.1:11434/api/chat", {},
      {"model": local_model,
       "messages": [{"role": "user",
                     "content": "Explain causal attention."}],
       "stream": False})
  text = response["message"]["content"]
```

A local model must already be installed and loaded through the runtime's supported provisioning path. Record runtime version, model manifest, quantization, and relevant license. Check network activity and configuration before making an offline claim.

Normalize without erasing differences

Define a common application response such as status, text blocks, tool proposals, usage, provider request ID, and model ID. OpenAI and xAI Responses outputs may contain multiple item types; collect supported text blocks without treating tool items as completed actions. Anthropic returns content blocks. Gemini Interactions returns step history with model-output content; a thought step is not the final answer. Ollama's non-streaming chat reply has a message field.

Preserve unsupported item types as explicit errors or retained metadata according to the adapter's policy. Do not coerce every response into a success string. Tool proposals still pass through the application's grant and execution path. Test adapters with retained fixtures for missing fields, partial output, refusals, malformed content, and provider errors before live integration.

Local image generation

Diffusers supports loading pipelines from compatible repositories or local directories [25]. The following is an unexecuted recipe for a locally provisioned compatible pipeline. Install and pin its required libraries separately from the CPU companion. Check architecture, license, available memory, dtype, and device support first.

Python excerpt

```
from diffusers import DiffusionPipeline

pipe = DiffusionPipeline.from_pretrained(local_model_dir)
pipe = pipe.to(device)  # e.g. a supported CPU or accelerator
result = pipe("A quiet workshop with blue evening light")
result.images[0].save("workshop.png")
```

A specific model may require a different pipeline class, precision, scheduler, or dependencies. Image editing additionally needs a documented image-to-image or inpainting contract. Preserve reference asset IDs, mask, strength, seed where supported, model revision, and selected output. A generic text-to-image pipeline is not automatically an editor.

Speech generation

OpenAI's documented speech endpoint accepts model, input, and voice fields [26]. The result is audio bytes rather than a JSON text reply. Choose a permitted voice and an available model from configuration. Store format, sample rate when known, and duration with the resulting asset. This recipe was not executed.

Python excerpt

```
import json
from urllib.request import Request, urlopen

body = json.dumps({"model": speech_model,
    "voice": voice, "input": narration}).encode()
request = Request("https://api.openai.com/v1/audio/speech",
    data=body, method="POST",
    headers={"Authorization": "Bearer " + key,
             "Content-Type": "application/json"})
with urlopen(request, timeout=60) as response:
    audio = response.read(10_000_001)
if len(audio) > 10_000_000:
    raise ValueError("audio exceeds this recipe's limit")
with open("narration.mp3", "xb") as output:
    output.write(audio)
```

A successful HTTP response does not establish intelligibility, pronunciation, or correct narration. Listen to the full result and compare it with the approved text. Clearly disclose synthetic speech in the interface when applicable to the service and project.

Music generation

MusicGen learns a music-generation path over compressed audio representations [27]. AudioCraft supplies a documented implementation [28]. Its repository distinguishes MIT-licensed code from model weights released under CC-BY-NC 4.0 in the consulted repository. Review the exact artifact license before commercial use; installing code does not change the weights' terms.

The documented basic API pattern is below. It requires separate dependencies and a model download, and was not executed for this edition.

Python excerpt

```
from audiocraft.models import MusicGen
from audiocraft.data.audio import audio_write

model = MusicGen.get_pretrained("facebook/musicgen-small")
model.set_generation_params(duration=8)
audio = model.generate(["Quiet instrumental with soft percussion"])
audio_write("cue", audio[0].cpu(), model.sample_rate,
            strategy="loudness", loudness_compressor=True)
```

Evaluate noise, coherence, duration, and suitability to the brief. Keep the original generation and final mix distinct. A music workflow can also use a hosted service or conventional composition tools; record that path accurately.

Video assembly and generated shots

A movie workflow can combine conventional editing with learned image, audio, or video generation. A generated shot and an assembled movie are different deliverables. Begin with a timed storyboard, scene assets, dialogue, audio cues, and export requirements.

FFmpeg provides encoding and assembly operations [29]. For a pre-existing `shot.png` and the companion tone fixture, this illustrative command makes a two-second still-image video. It was not executed as part of the verification record.

Command or pseudocode

```
ffmpeg -loop 1 -i shot.png -i media_example/tone.wav
       -t 2 -vf scale=1280:720,format=yuv420p
       -c:v libx264 -c:a aac -shortest fixture.mp4
```

Enter it as one command or use your shell's proper line continuation. Installed builds differ in codec support. For multiple shots, create a concat list or an editing timeline with exact durations, then render and inspect every transition. For learned video generation, use a provider or model's documented job API and polling/cancellation contract. This edition does not fabricate one universal video endpoint across incompatible services.

Appendix D Hardware and Performance Worksheet

Estimate then measure

Memory estimates help reject impossible configurations, but they do not predict usability alone. Runtime overhead, activations, caches, batch size, context length, fragmentation, and offloading matter. Measure the actual model and runtime on the intended hardware before making a purchasing or deployment decision.

Raw weight bytes are approximately parameter count times bytes per parameter. Decimal GB uses one billion bytes; GiB uses 2^30 bytes. Low-bit schemes also store scales and metadata.

Illustrative 7 billion parameter storage	Raw bytes	Decimal GB	What is excluded
Float32 at 4 bytes	28 billion	28	Runtime and all other memory
Float16 at 2 bytes	14 billion	14	Runtime and all other memory
Idealized 4-bit at 0.5 bytes	3.5 billion	3.5	Quantization metadata and all other memory

These are arithmetic examples, not measured requirements for a named model. Some parameter representations mix precision, share weights, or use additional buffers. Inspect the actual files and memory use.

Training state

An illustrative FP32 Adam arrangement uses four bytes each for weights, gradients, first moment, and second moment: 16 bytes per parameter before activations and other buffers. Seven billion parameters would therefore account for 112 decimal GB in those four arrays alone. Mixed precision, sharding, offloading, recomputation, and optimizer choices change the allocation.

The companion model has 10,992 parameters and uses float64 arrays for easier numerical checks. Do not extrapolate its runtime or memory directly to a billion-parameter training run. Its corpus, sequence length, kernels, device, and batch size are entirely different.

Key and value cache

A simple decoder cache estimate is:

Command or pseudocode

```
bytes = 2 * layers * sequence * batch
        * KV_heads * head_dimension * bytes_per_element
```

The factor two counts keys and values. With 32 layers, sequence 4096, batch one, eight KV heads, head dimension 128, and two-byte elements, the arithmetic gives 536,870,912 bytes, or 512 MiB. A different attention layout, cache representation, or implementation changes the estimate. The tiny companion recomputes its context and does not implement this serving cache.

Measurement record

Record hardware, operating system, runtime, model, precision, context, batch, input lengths, output lengths, concurrency, warm-up procedure, and network conditions. Measure time to first output, total latency, output throughput, peak memory, and failures. Report percentile latency over a workload with enough samples to justify the claim.

Compare quality under the same task cases. A faster quantized model can be an improvement for one application and a regression for another. Measure energy or battery behavior when building an edge product. Make cost ceilings explicit for hosted work, including repeated attempts and auxiliary transcription or retrieval calls.

Appendix E Reusable Records and Acceptance Templates

System specification

JSON example

```
{
  "system_id": "lab-assistant-v1",
  "task": "Answer questions from the approved handbook",
  "users": ["project-member"],
  "allowed_data": ["handbook-v1"],
  "allowed_effects": ["draft-answer"],
  "prohibited_effects": ["publish", "send-message"],
  "route": "local-only",
  "limits": {"max_steps": 4, "deadline_seconds": 10},
  "success": "Reviewed claims link to retained passages"
}
```

Use a schema to validate the actual implementation. A descriptive JSON object alone does not enforce its fields. The pure disposition function in the companion consumes validated boolean decisions; it is not a full system-specification interpreter.

Evidence and interpretation

JSON example

```
{
  "evidence_id": "handbook-v1:p1",
  "origin_kind": "source_record",
  "collected_by": "approved-ingestor",
  "collection_method": "local-file-snapshot",
  "source_version": "v1",
  "content_hash": "computed-sha256",
  "parent_ids": [],
  "access_scope": ["project-member"]
}
```

JSON example

```
{
  "claim_id": "checkout-rule",
  "text": "Checkout requires a recorded return date",
  "support_ids": ["handbook-v1:p1"],
  "conflict_ids": [],
  "assumptions": ["This handbook governs this lab"],
  "status": "PROPOSED_INTERPRETATION",
  "unresolved": []
}
```

The literal `computed-sha256` is an instructional field description to replace with the computed value. It is not a valid integrity record. Define canonicalization, source

retention, and collection authority before using the schema operationally. The companion's simpler Evidence object demonstrates only part of this richer record.

Grant and execution receipt

A grant should bind issuer, actor, action, exact resource, applicable constraints, validity, nonce or replay policy, and signature or authenticated storage reference. A receipt should bind operation ID, attempted action, authorized scope, status, observed effect, artifact hash where useful, timestamp, and error or reconciliation reference.

Keep the grant and receipt as distinct records. A valid grant can lead to failed execution. An external timeout can lead to an uncertain effect. Model text saying “done” should never replace observation of the resource or a trustworthy execution receipt.

Evaluation case

JSON example

```
{
  "case_id": "private-local-outage",
  "configuration": "router-v1",
  "inputs": {"private": true, "local_available": false,
             "cloud_allowed": true},
  "expected": {"route": "unavailable"},
  "forbidden_effect": "sending private data to cloud",
  "observation": "Record actual route and network effect",
  "result": "PENDING"
}
```

The pending state belongs in a template. Replace it with observed results when running a case. Include the configuration, date, evaluator, artifacts, and limitations. Do not copy a successful result from one configuration into another without performing the relevant check.

Creative asset manifest

Record asset ID, media kind, brief, source references, permission basis, generation or editing tool, version, settings, selected output, content hash, duration or dimensions, review, and release destination. Derivatives should retain parent references. Changing an approved artifact requires the applicable review for that change.

Incident record

Record what was observed, affected tasks, authority and data boundaries, containment, known external effects, uncertain effects, recovery, and regression cases. Separate observations from hypotheses about the cause. A useful incident record remains revisable when stronger evidence arrives.

Appendix F Projects Sources and Verification

The sixteen project paths

The chapters are complete teaching text. The implementation status below describes the companion and external integrations accurately. A project path can include local implemented components and additional integrations the reader must execute and evaluate.

Project	Chapters	Included result	Status boundary
P01 System specification	1-6	Specification and evaluation templates; classical labs	Local algorithm checks ran; real task validation remains project-specific
P02 Replaceable chatbot	7-9	Extractive browser chatbot	Local HTTP path checked; generative adapter requires separate integration
P03 Document assistant	10	Retained evidence and lexical retrieval	Core checked; arbitrary PDF ingestion and synthesis not implemented
P04 Revisable memory	11	SQLite origin, revision, deletion	Core checked; access control and backup deletion need deployment work
P05 Voice assistant	12	Pipeline and response-policy design	No microphone, ASR, TTS or diarization integration executed
P06 Local assistant	13-14	Tiny model inference; Ollama recipe	Tiny endpoint checked; Ollama provisioning not executed
P07 Hosted and hybrid	15-18	Routing function and provider recipes	Routing checked; live providers and packaging integrations unexecuted
P08 Writing and image	19-20	Media job records; image recipe	Job core checked; learned image integration unexecuted
P09 Audio and music	21-22	Tone fixture; speech and MusicGen recipes	Fixture checked; learned speech and music unexecuted

Project	Chapters	Included result	Status boundary
P10 Short film	23-24	Timeline fixture and assembly recipe	Timeline included; movie export and learned shots unexecuted
P11 Task agent	25-27	Bounded loop and permissioned write	Core checked; browser integration and hard deadlines need additional work
P12 Coordination and sandbox	28-30	Handoff, sandbox and recovery designs	No multi-agent service or hostile-code sandbox executed
P13 Language model	31-34	Data, 10,992-parameter transformer, checkpoint, serving	Trained and locally checked; restricted synthetic data
P14 Adaptation	35-36	Output-matrix-only experiment	Ran with frozen-weight check; no LoRA or distributed training executed
P15 AETERNA gates	37-42	Evidence, grants, Gate, disposition, audit teaching code	Local checks ran; not the production AETERNA repository or its acceptance suite
P16 Capstones	43-48	Integration designs using the included components	Local core available; complete external products need project-specific integration

Recorded experiment results

The 160-step training run reduced fixed-window validation loss from 3.2355 to 1.1277. The saved checkpoint reproduced probe logits with maximum difference 0.0 in the recorded environment. A separate 80-step output-only adaptation reduced its restricted validation loss from 1.2298 to 0.9682 and preserved every frozen parameter exactly.

These experiments use synthetic restricted grammar and shared vocabulary. Their samples remain imperfect. The results demonstrate training, persistence, adaptation, and selected numerical properties. They do not establish general assistant ability, independent factual reliability, or frontier-scale performance.

The retained test output reports 24 passing tests. One of those enumerates all 64 boolean combinations of the teaching disposition inputs. Test count is a description

of executed checks, not a measure of production completeness. The manifest retains software versions and SHA-256 hashes of the companion files, excluding itself and transient caches.

Primary sources

Numbered references support nearby technical claims. The project's architectural choices, original examples, and local experiment results are separately identified. External sources do not independently validate AETERNA or endorse this book. Documentation was consulted October 1 2026; follow its current revision when building.

[1] Vaswani et al. Attention Is All You Need. 2017. https://arxiv.org/abs/1706.03762

[2] PyTorch Tutorials. Optimizing Model Parameters. https://docs.pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

[3] Lewis et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. 2020. https://arxiv.org/abs/2005.11401

[4] Ollama. API Introduction. https://docs.ollama.com/api/introduction

[5] Scikit-learn. Supervised Learning. https://scikit-learn.org/stable/supervised_learning.html

[6] Mnih et al. Playing Atari with Deep Reinforcement Learning. 2013. https://arxiv.org/abs/1312.5602

[7] Scikit-learn. Cross-validation. https://scikit-learn.org/stable/modules/cross_validation.html

[8] Scikit-learn. Model Evaluation. https://scikit-learn.org/stable/modules/model_evaluation.html

[9] OWASP. LLM01 Prompt Injection. https://genai.owasp.org/llmrisk/llm01-prompt-injection/

[10] Python. sqlite3. https://docs.python.org/3/library/sqlite3.html

[11] MDN. AudioWorklet. https://developer.mozilla.org/en-US/docs/Web/API/AudioWorklet

[12] Radford et al. Robust Speech Recognition via Large-Scale Weak Supervision. 2022. https://arxiv.org/abs/2212.04356

[13] Meta. Llama Models Repository. https://github.com/meta-llama/llama-models

[14] Ollama. Chat API. https://docs.ollama.com/api/chat

[15] OpenAI. API Quickstart. https://developers.openai.com/api/docs/quickstart

[16] Anthropic. Messages API. https://platform.claude.com/docs/en/api/messages/create

[17] Google. generateContent API. https://ai.google.dev/api/generate-content

[18] xAI. Generate Text. https://docs.x.ai/developers/model-capabilities/text/generate-text

[19] Python. http.server. https://docs.python.org/3/library/http.server.html

[20] Kingma and Welling. Auto-Encoding Variational Bayes. 2013. https://arxiv.org/abs/1312.6114

[21] Goodfellow et al. Generative Adversarial Networks. 2014. https://arxiv.org/abs/1406.2661

[22] Ho et al. Denoising Diffusion Probabilistic Models. 2020. https://arxiv.org/abs/2006.11239

[23] Rombach et al. High-Resolution Image Synthesis with Latent Diffusion Models. 2021. https://arxiv.org/abs/2112.10752

[24] Lipman et al. Flow Matching for Generative Modeling. 2022. https://arxiv.org/abs/2210.02747

[25] Hugging Face. Diffusers Loading Pipelines. https://huggingface.co/docs/diffusers/en/using-diffusers/loading

[26] OpenAI. Text to Speech. https://developers.openai.com/api/docs/guides/text-to-speech

[27] Copet et al. Simple and Controllable Music Generation. 2023. https://arxiv.org/abs/2306.05284

[28] Meta. AudioCraft Repository and MusicGen Guide. https://github.com/facebookresearch/audiocraft and https://github.com/facebookresearch/audiocraft/blob/main/docs/MUSICGEN.md

[29] FFmpeg. Command Documentation. https://ffmpeg.org/ffmpeg.html

[30] Hu et al. LoRA Low-Rank Adaptation of Large Language Models. 2021. https://arxiv.org/abs/2106.09685

[31] Model Context Protocol. Architecture Overview. Consulted version 2026-07-28. https://modelcontextprotocol.io/docs/learn/architecture

[32] Microsoft. Playwright Python Installation. https://playwright.dev/python/docs/intro

[33] Rafailov et al. Direct Preference Optimization. 2023. https://arxiv.org/abs/2305.18290

[34] PyTorch. Distributed Communication Package. Consulted version 2.14. https://docs.pytorch.org/docs/2.14/distributed.html

[35] Python. hmac. https://docs.python.org/3/library/hmac.html

[36] Python. hashlib. https://docs.python.org/3/library/hashlib.html

[37] Google. Gemini Getting Started. https://ai.google.dev/gemini-api/docs/get-started

Appendix G Reading the Core Algorithms

The computation graph

The companion is the full runnable implementation. These excerpts explain its most important steps. Run the original files rather than treating shortened excerpts as independent programs.

A Tensor stores data, gradient, parents, and a local backward operation. Addition, multiplication, matrix multiplication, normalization, and softmax build a graph. The loss's backward method visits parents before descendants, then applies local derivatives in reverse dependency order:

Python excerpt

```
def backward(self):
    order, seen = [], set()
    def visit(node):
        if id(node) in seen:
            return
        seen.add(id(node))
        for parent in node.parents:
            visit(parent)
        order.append(node)
    visit(self)
    for node in order:
        node.grad.fill(0)
    self.grad[...] = 1
    for node in reversed(order):
        node._backward()
```

For multiplication `out = a * b`, the local derivative adds `out.grad * b.data` to a's gradient and `out.grad * a.data` to b's gradient. Broadcasting requires summing gradients back to each operand's original shape. Without that reduction, bias and scale updates are often wrong even when forward values look correct.

For matrix multiplication, the last two dimensions are matrix dimensions; leading dimensions may represent batches. The gradient for the left operand multiplies the output gradient by the right operand's transposed final dimensions. The right operand uses the left operand's transposed final dimensions. Reduce broadcast dimensions afterward.

Token and position embeddings

The teaching model uses one-hot arrays for clarity. Matrix multiplication selects an embedding vector for each token. A second array selects a learned positional vector. More efficient libraries use indexed embedding kernels instead of materializing one-hot arrays.

Python excerpt

```
onehot = np.eye(vocab)[ids]          # B, T, V
positions = np.eye(context)[:t]    # T, context
x = Tensor(onehot) @ embedding
x = x + Tensor(positions) @ position
```

The resulting shape is `(B, T, D)`. The same embedding parameter participates at every occurrence of a character; its gradient accumulates across those occurrences.

Causal attention

Project normalized representations into queries, keys, and values. Reshape to `(B, H, T, D/H)`. Query-key multiplication creates `(B, H, T, T)` scores. Divide by the square root of head depth, exclude future positions, and normalize each row.

Python excerpt

```
mask = np.where(np.triu(np.ones((t, t)), 1), -1e9, 0)
scores = q @ k.transpose((0, 1, 3, 2))
attention = (scores * depth ** -.5 + mask).softmax()
mixed = attention @ v
```

The companion concatenates heads, applies an output projection, and adds a residual. A normalized feedforward layer supplies the second residual. Final normalization and projection produce vocabulary logits. Its future-token test compares earlier logits after altering later tokens; that is a meaningful check of the causal boundary.

Cross-entropy and stability

For each target token, loss is negative log probability. Compute `logsumexp` using a maximum subtraction to avoid exponentiating unnecessarily large values. Average over the batch and sequence. The derivative with respect to logits is predicted probability minus the target indicator, divided by the number of evaluated positions.

Command or pseudocode

```
shifted = logits - row_max
log_normalizer = log(sum(exp(shifted)))
log_probability = shifted - log_normalizer
loss = mean(-log_probability[target])
```

```
gradient = (probabilities - target_onehot) / positions
```

This objective asks the model to predict the next character under the training distribution. It does not directly ask for truthful claims, permission compliance, or a useful user answer. Those require application design and additional evaluation.

One optimizer update

The companion computes a combined gradient norm, clips it to at most one, updates Adam's first and second moments, applies bias correction using the cumulative step, and subtracts the learning-rate-scaled update. Read `Adam.step` alongside its stored moment arrays.

Command or pseudocode

```
m = 0.9 * m + 0.1 * gradient
v = 0.999 * v + 0.001 * gradient^2
m_hat = m / (1 - 0.9^step)
v_hat = v / (1 - 0.999^step)
weight = weight - learning_rate * m_hat / (sqrt(v_hat) + 1e-8)
```

The numbers are this experiment's choices, not a universal tuning prescription. A resumable checkpoint must preserve the moments and step as well as weights. The reference test checks the next update after reload against continuation in the original state.

The disposition boundary

The following function is included verbatim in `core.py`. It is deliberately narrow: a pure decision on six validated facts, with denial priority. It cannot authenticate the facts or execute a task itself.

Python excerpt

```
def disposition(valid_shape, policy_available, in_scope, grant_valid,
                evidence_required, evidence_resolved):
    values = (valid_shape, policy_available, in_scope, grant_valid,
              evidence_required, evidence_resolved)
    if any(type(v) is not bool for v in values):
        return 'DENIED'
    if not all(values[:4]):
        return 'DENIED'
    if evidence_required and not evidence_resolved:
        return 'UNRESOLVED'
    return 'AUTHORIZED'
```

The function's docstring is omitted here for space. The executable suite enumerates every boolean combination and rejects a truthy string. This helps prevent accidental permissive coercion at this boundary. The real grant validator and executor remain separate components.

A reproducible next experiment

Choose one change: context size, corpus, learning rate, model depth, or adaptation target. Predict its effect and select a measurement before running. Save a new configuration and artifacts. Compare the same fixed evaluation against the baseline, then inspect samples and failure cases.

When the outcome differs from the prediction, retain that result. A useful engineering record explains what ran and what happened. The next improvement begins from that record rather than from a rewritten story of what the system was supposed to do.

BUILD THE SYSTEM.
PROVE THE BOUNDARY.

Start with a chatbot. Build retrieval and revisable memory. Train a small transformer from its data to its saved checkpoint. Then connect creative tools, task agents, and local or hosted models with clear permissions.

Across 48 chapters, this book follows the parts of an AI system that matter after the model speaks: evidence, interpretation, authority, execution, audit, and recovery. AETERNA supplies a practical design language for those boundaries.

The companion includes executable Python examples, recorded training output, and 24 passing checks. Provider and media integrations are labeled as recipes until separately run and evaluated.

READ / BUILD / TEST / REVISE

CHRISTOPHER PAYNE

ANNA NEXUS LABS LLC | FIRST EDITION | 2026