INF 385T · FALL 2026

TEXAS
The University of Texas at Austin

INF385T: Special Topics in Information Science: Language Models and Agentic Workflows

Week 4: Deterministic Agentic Workflows

Dr. Abhijit Mishra

Announcement

- Project group formation – DONE
- First Group Proposal Due Next week

1. Evidence-Grounded Research Brief Workflow

a. Agentic Unlearning: Krutik, Akshat, Nathan

b. Nikhil: Nikhil P

c. Agents make life better? : Boyue, Fangyuan

2. Support-Ticket Triage and Resolution Workflow

a. OP-US: Haren, Girishkaushik, Disha

b. Group AAI: Adi, Arjun , Ishika

3. Document Compliance and Quality Review Workflow

a. Byte Me: Robin, Joe

b. Superstar: Jenny, Kayla

4. Data Analysis and Reporting Workflow

a. RAG to Riches🤑💸: Ashutosh, Abhee, Monisha

b. Bgentic: Yilun, Jiayu, Chengkang

5. Meeting-to-Action Workflow

a. OC: Ziyuan, Wenxuan, Xuan

b. Chechi: Chitse, CHENG-CHIN

Week 3: Recap

- Context Engineering: Retrieval, Memory, and State
  - Emphasize distinctions among retrieval, memory, state, and context
  - Context vs Prompt engineering
  - Retrieval augmented generation basics

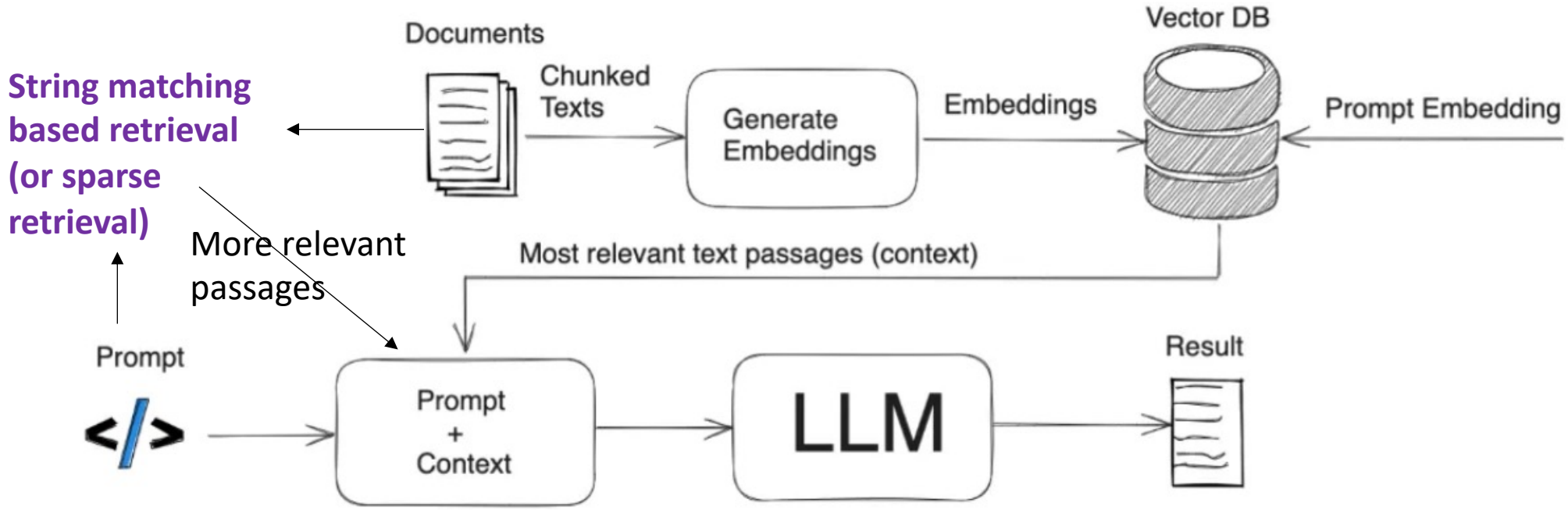
Week 3: Recap

- **Retrieval:** *What does the outside world say that's relevant?*
- **State:** *Where are we in solving this task?*
- **Memory:** *What have I learned about you from previous tasks?*

Recap: Prompt Engineering vs Context Engineering

- **Prompt engineering:** how the task and response behavior are expressed
- **Context engineering:** what information, tools, history, state, and constraints are supplied
- Good wording cannot compensate for absent or irrelevant evidence
- Context engineering is a system-level activity

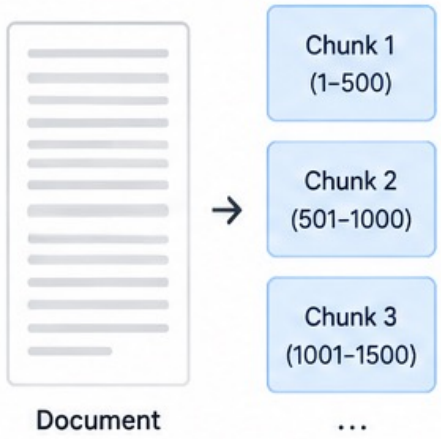
RAG: A typical RAG workflow



https://medium.com/@kushalsharma0508/rag-workflow-retrieval-augmented-generation-ai-5baee937d71c

1. Fixed-size Chunking

Split text into equal-sized chunks (e.g., 500 tokens).

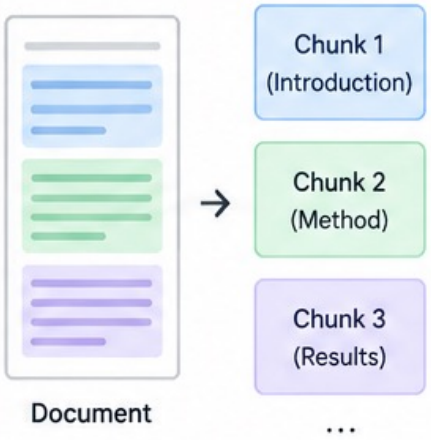


Simple and fast. Good baseline.

Can split in the middle of a sentence or idea.

2. Semantic Chunking

Split at natural boundaries (e.g., paragraphs, headings, or meaning-based splits).

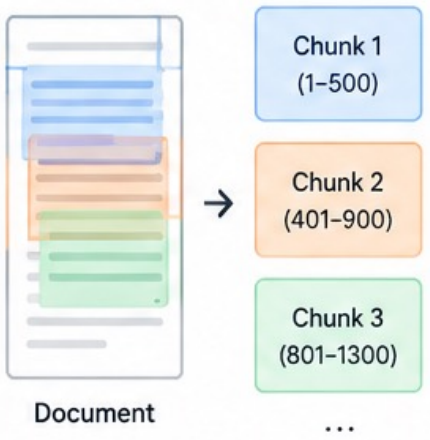


Keeps related content together.

Better context and coherence.

3. Overlapping Chunking

Split into chunks with overlaps (e.g., 500 tokens with 100 token overlap).

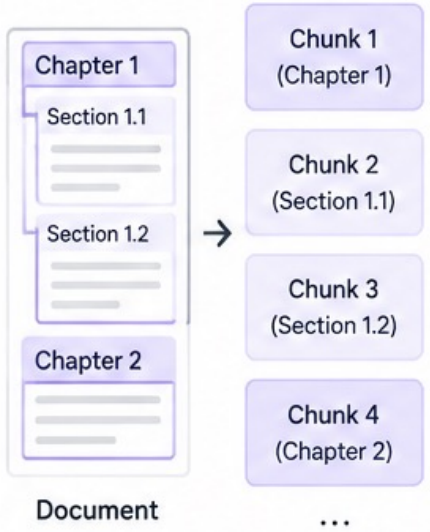


Reduces loss of context at chunk boundaries.

Common and effective in practice.

4. Hierarchical Chunking

Split by document structure (e.g., chapters → sections → paragraphs).

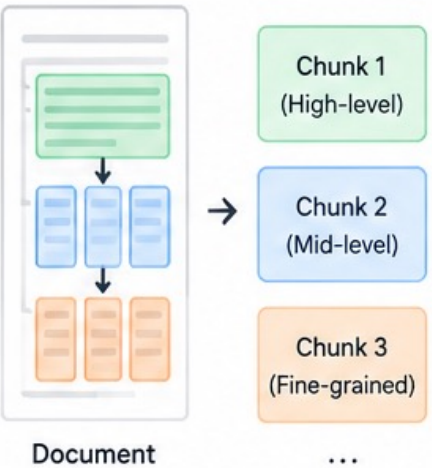


Preserves document structure and hierarchy.

Useful for long, structured documents (e.g., reports, manuals).

5. Recursive Chunking

Try larger semantic units first, then recursively split until chunks fit a target size.



Balances meaning and size automatically.

Adapts to content of different types and lengths.

Common chunking methods

Recap: Dense Passage Retrieval using Document Embeddings

- Treat the query as a tiny document
- Represent the query and every document each as a word vector in a word space
- Rank documents according to their proximity to the query in the word space
- Vector Distance or Similarity is Key

$$d(X^1, X^2) = \sqrt{\sum_{i=1}^{N} (x_i^1 - x_i^2)^2}$$

$$\text{cosine}\,(X^1 `` X^2) = 1 - \frac{\sum_{i=1}^{N} x_i^1 x_i^2}{\sqrt{\sum_{i=1}^{N} (x_i^1)^2} \cdot \sqrt{\sum_{i=1}^{N} (x_i^2)^2}}$$

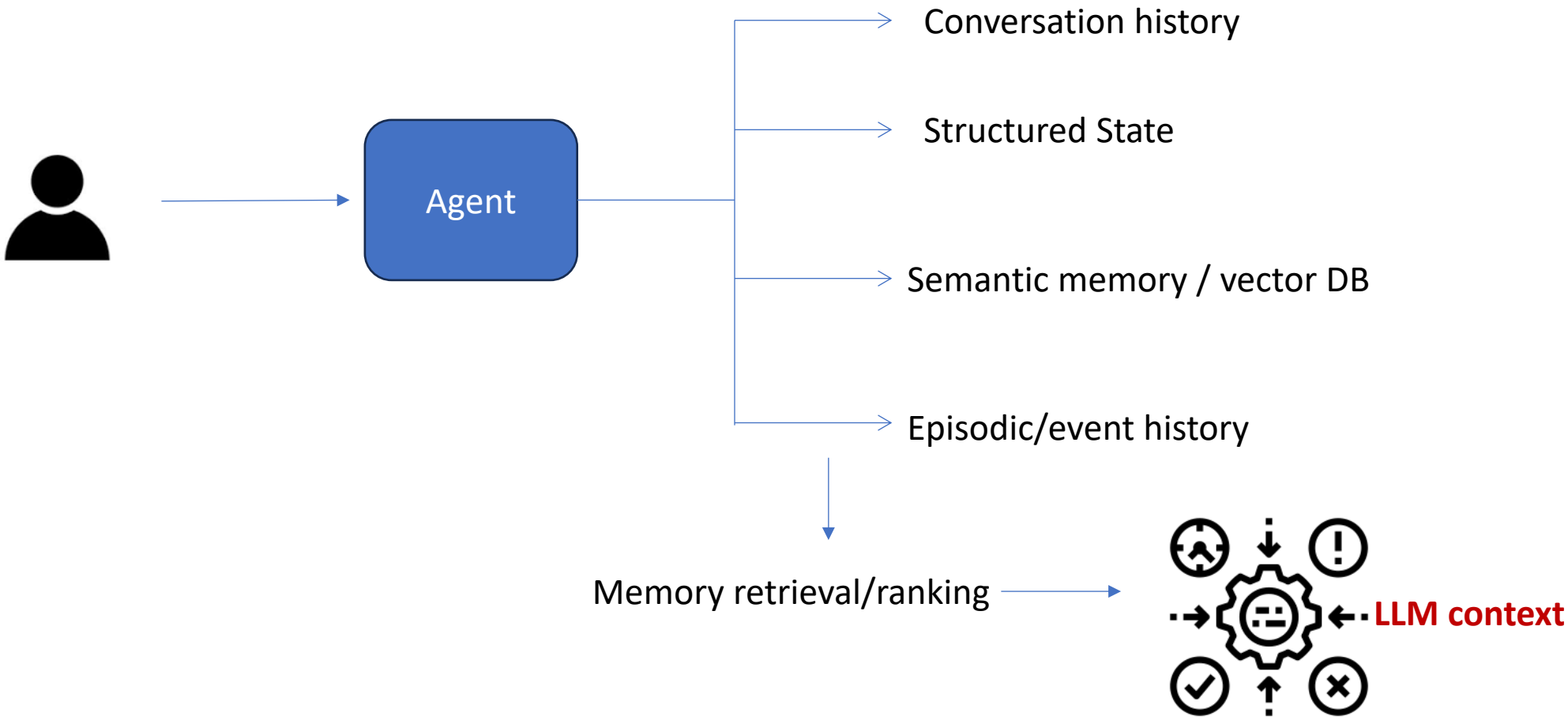
Recap: Memory

- Could be plain text, JSONS or database
- Relevant memory could be retrieved using database querying, dense / sparse retrieval

Recap: Memory types

- Working / short-term memory
  This is usually just the agent's current context
  Preferred format: string (part of the prompt)
- Long-term semantic memory
  Information worth remembering across sessions is stored externally
  Preferred format: Databases + SQL
- Episodic memory
  Instead of storing facts, the agent remembers things that happened
  Summarizing Compressing agent actions before storing in memory
  Preferred format: Markdown files
- Structured State/Memory
  For many agents, this is actually the most important type
  JSON schema

Recap: A production architecture



Week 3 -> Week 4

- Week 3 built an information pipeline
  Pipelines already contain stages, evidence, state, and validation
- Week 4 generalizes this into an explicit orchestration architecture

Central question: which decisions should remain deterministic?

Week 4: Roadmap

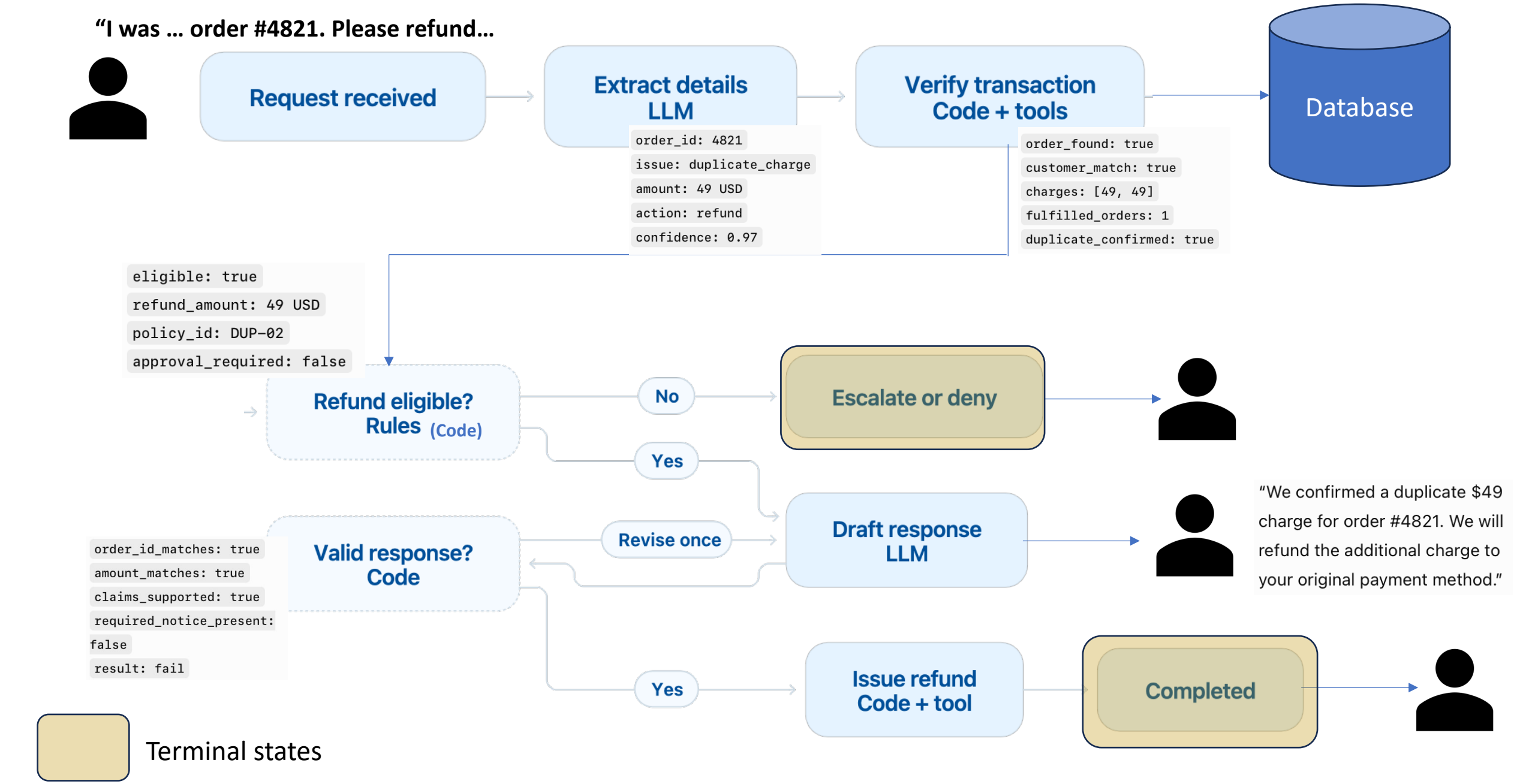
- Deterministic Workflows vs Autonomous agents
- Prompt chaining and conditional routing
- Parallelization and aggregation
- Evaluator–optimizer patterns
- State machines and task graphs
- Deterministic control around nondeterministic models.

Deterministic Workflow

- A workflow is a controlled sequence or graph of stages that specifies how a task moves from input to completion.
- Typically has:
  - Ordered or graph-structured stages
  - Defined inputs and outputs
  - Explicit transitions between stages
  - Known control logic
  - Clear terminal conditions
  - Observable execution state

Example: Customer Refund Workflow

“I was charged twice for order #4821. Please refund the extra $49 charge.”


"I was … order #4821. Please refund…
Request received
Extract details
LLM
order_id: 4821
issue: duplicate_charge
amount: 49 USD
action: refund
confidence: 0.97
Verify transaction
Code + tools
order_found: true
customer_match: true
charges: [49, 49]
fulfilled_orders: 1
duplicate_confirmed: true
Database
eligible: true
refund_amount: 49 USD
policy_id: DUP-02
approval_required: false
Refund eligible?
Rules (Code)
No
Escalate or deny
Yes
Draft response
LLM
Revise once
Valid response?
Code
order_id_matches: true
amount_matches: true
claims_supported: true
required_notice_present: false
result: fail
"We confirmed a duplicate $49 charge for order #4821. We will refund the additional charge to your original payment method."
Yes
Issue refund
Code + tool
Completed
Terminal states

Exercise

- **Group exercise:** Design a deterministic workflow for processing a university course-add request, identifying the LLM calls, code-controlled decisions, validation gates, human escalation path, and terminal states

- Submit to canvas (1 PDF document with your diagram in it)
- **Time limit:** 15 minutes

Deterministic Workflow vs Adaptive Agent

- **Determinstic Workflow**: code determines the allowable route and transition logic
- **Agent**: model may choose the next approved action from observations
- Both may use models, tools, state, memory, and evaluators
- Difference lies primarily in who controls the next step

“Deterministic does not mean model-free”

- Model calls remain nondeterministic components
- Python (Code) fixes placement, schemas, branches, budgets, and gates
- Same workflow graph can process variable model outputs
- Reliability comes from bounded interfaces and checks, not identical prose

which concept from last three weeks apply here?

Why begin with workflows (and not adaptive agents)?

- Predictable execution paths
- Easier testing and debugging
- Lower coordination overhead
- Clearer permissions and approvals
- More stable cost and latency
- Appropriate when stages are understood in advance

Design the workflow before prompting



Decompose by responsibility

- Each stage should have one coherent purpose
- Separate retrieval, transformation, judgment, execution, and verification
- Keep deterministic operations in code
- Use models where semantic flexibility provides value

Core workflow patterns

Typed stage boundaries

- Named input and output fields
  - **Example**: Input: customer_message → Output: issue_type
- Required and optional values
  - **Example**: order_id required; customer_note optional
- Closed status values and explicit errors
  - status ∈ {approved, denied, escalated}; error: ORDER_NOT_FOUND
- Evidence **provenance** carried forward
  - refund decision linked to transaction TX-4821
- Typed boundaries prevent prose from silently controlling downstream stages
  - Only validated refund_amount can trigger payment

Preconditions, postconditions, invariants, and state

- **Precondition:** what must be true before a stage runs
  - *Example: Check refund eligibility only after verifying the order and charges.*
- **Postcondition:** what the stage must establish
  - *Example: The eligibility stage returns approved or denied with a reason.*
- **Invariant**: what must remain true throughout execution
  - *Example: Never refund more than the verified duplicate charge.*
- **State** records the run ID, current stage, artifacts, errors, retry count, budget, and terminal status
  - *Example: Stage: validation | Retries: 1 | Status: running*
- **Workflow safeguards:** Keep the original request, limit model calls, and connect every decision to valid evidence.

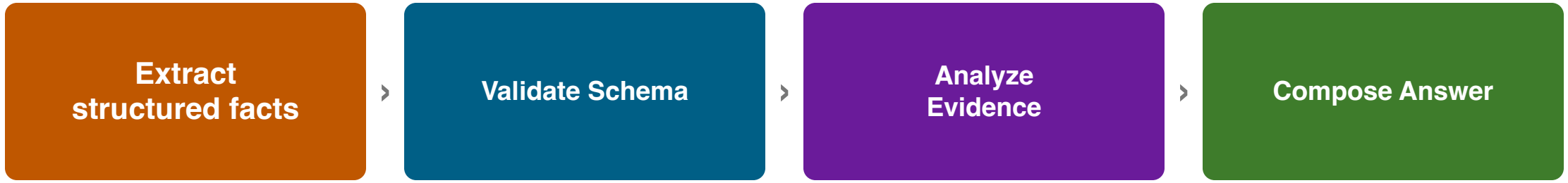
Observable transitions

- **Record** the source stage and destination stage
- **Record** why the transition occurred
- **Link** the input and output artifact identifiers
- **Sequence** or timestamp each event
- **Exclude** secrets and private model reasoning
- **Transition** logs support replay, testing, and audit

Prompt chaining

- Split one compound model request into sequential bounded stages
- Output from one stage becomes validated input to the next
- Appropriate when later work depends on a preceding artifact

Prompt chaining



Each stage has one responsibility. A failed gate blocks every dependent stage.

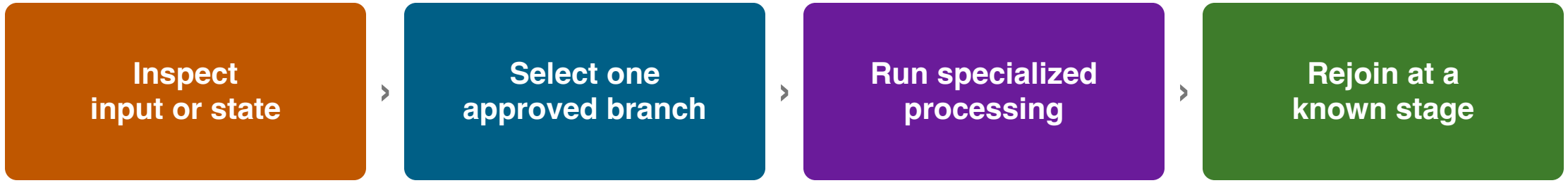
Example “I was charged twice for order #4821.”

- **Extract structured facts**
  Order: #4821 | Issue: duplicate charge | Amount: $49 | Request: refund
- **Validate schema**
  Confirm that the order number, issue type, amount, and requested action are present and correctly formatted.
- **Analyze evidence**
  Compare the extracted claim with payment records and confirm that two $49 charges exist for one order.
- **Compose answer**
  “We confirmed a duplicate $49 charge for order #4821 and approved a refund to your original payment method.”

Prompt-chain risks and invariants

- Additional calls increase latency, tokens, and cost
- Early errors can propagate into later stages
- Transformations may discard important information
- Every stage consumes validated data
- Evidence identifiers persist across the chain
- Intermediate outputs remain bounded

Conditional routing



Use code for clear business rules. Use a model classifier only when the category requires semantic judgment.

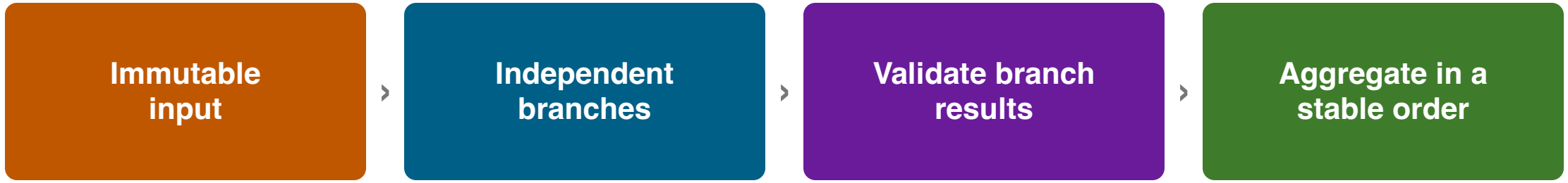
Example: “I was charged twice for order #4821.”

- **Inspect input or state**
  An LLM classifies the issue as **duplicate_charge** with 97% confidence.
- **Select one approved branch**
  Code maps **duplicate_charge** to the predefined **Billing Dispute** branch.
- **Run specialized processing**
  The billing branch retrieves the payment records and checks for duplicate charges.
- **Rejoin at a known stage**
  The branch returns **duplicate_confirmed: true** to the shared **Draft Response** stage.

Routing logic and failures

- Define precedence when multiple routes match
- Handle no-match and ambiguous-match cases explicitly
- Test rule boundaries without model calls
- Keep consequential routes behind stronger gates
- Common failures: wrong branch, overlapping definitions, silent defaults, or incompatible branch outputs

Parallel fan-out and fan-in



Parallel execution can reduce **wall-clock latency.** It does not reduce calls, tokens, or total cost.

Example: “Order #4821 has two $49 charges. The customer requests a refund.”

- **Immutable input**
  Every branch receives the same unchanged customer request and transaction record.
- **Independent branches**

  Run three checks simultaneously:

Payment check: Was the customer charged twice?
Policy check: Does the refund policy cover duplicate charges?
Fraud check: Does the account show suspicious activity?

Example: “Order #4821 has two $49 charges. The customer requests a refund.”

- **Validate branch results**
  Confirm that every branch returned the required fields, evidence IDs, and an approved status value.
- **Aggregate in a stable order**
  Combine results as payment → policy → fraud, then produce: Duplicate confirmed | Refund allowed | Fraud risk low

Fan-out and fan-in contracts

- Assign stable branch identifiers
- Use one result schema across branches
- Mark branches as required or optional
- Set a timeout for each branch
- Define the partial-failure policy
- Sort results deterministically before aggregation

Aggregation

- **Validate** every branch result before merging
- **Preserve** the source of each finding
- **Deduplicate** overlapping findings
- **Surface** conflicts instead of hiding them
- **Constrain** the final artifact format
- The aggregator cannot invent evidence from a missing branch

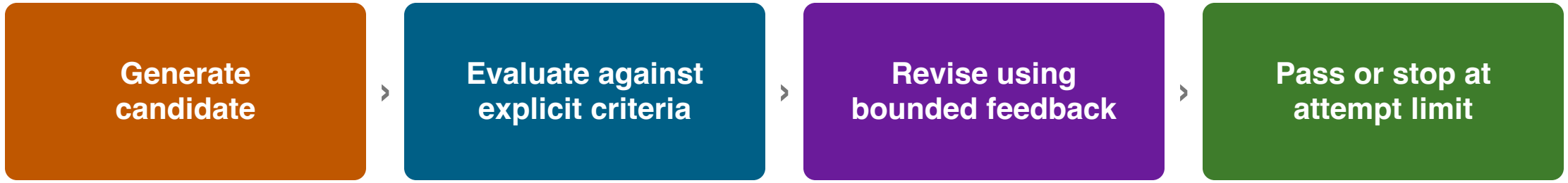
Validation gates

- Schema, type, and required-field checks
- Evidence membership and provenance checks
- Policy and authorization checks
- Length, token, latency, and cost limits
- Domain invariants
- Run the gate before output changes state or reaches another system

Three validation layers

- **Deterministic checks** cover reproducible, observable properties
- **LLM Judge:** Model evaluators score semantic quality against a written rubric
- **Human reviewers** handle consequential or subjective judgments
- A model critic can inform a decision but cannot serve as an authorization boundary

Evaluator and optimizer



This pattern fits revisable outputs with clear evaluation criteria and no irreversible side effects.

Example: “Improving a Refund Response”

- **Generate candidate**
  The LLM drafts:
  “Your $49 refund has been approved.”
- **Evaluate against explicit criteria**
  The evaluator checks: correct amount, supporting evidence, required processing-time notice, and no unsupported promises.
  Result: Fail. The response omits the processing time.

Example: “Improving a Refund Response”

- **Revise using bounded feedback**
  The LLM revises:
  “Your $49 refund has been approved. It should appear on your original payment method within 5–7 business days.”
- **Pass or stop at attempt limit**
  The revised response passes all criteria and moves forward. If it still failed after two revisions, the workflow would stop and escalate to a human.

A bounded evaluator loop

- Set a maximum revision count
- Keep the rubric stable across attempts
- Return a structured evaluation result
- Tie feedback to evidence and specific criteria
- Preserve candidates and scores
- Stop when output repeats or no longer improves

State machines and task graphs

- A state machine defines lifecycle states, events, guards, updates, and terminal states
- A task graph shows execution dependencies, branches, joins, and bounded cycles
- One implementation may use both views
- State remains the source of truth
- Reject every illegal transition

Workflow states

- **Typical progress states:** received, classified, analyzed, drafted, and validated
- **Terminal success state:** completed
- **Waiting state:** clarification_needed
- **Terminal exception states:** blocked or failed
- Each transition requires a named event and a satisfied guard

Failure belongs in the architecture

- Model timeout or invalid structured output
- Tool or service error
- Missing parallel branch result
- Validation rejection
- Budget exhaustion
- Human approval unavailable
- Classify failures as transient or permanent before choosing recovery

Retry, revise, fallback, escalate, or stop

- Retry repeats a transient and safe operation
- Revise repairs an artifact that failed evaluation
- Fallback selects an alternate bounded path
- Escalate transfers the decision to an authorized person or system
- Stop terminates when the workflow cannot establish success safely

Control-flow evaluation and content evaluation

- **Control tests:** correct route, legal transitions, required stages, prohibited stages, joins, retries, terminal policy, and budget
- Use deterministic mocks to test the control graph without paid model calls
- **Content tests:** correctness, grounding, completeness, constraints, semantic rubric, and human judgment
- The same control graph can contain strong or weak model outputs

(More on evaluations next week)

Summary

- Deterministic workflows keep control flow in code while using LLMs for bounded semantic tasks.
- Typed inputs and outputs create reliable boundaries between stages.
- Chaining, routing, parallelization, and evaluator loops support different task structures.
- Validation gates prevent unchecked model outputs from changing state or triggering actions.
- Explicit state, terminal conditions, retry limits, make workflows testable and recoverable.
- Evaluate workflow behavior separately from the quality of model-generated content.

Takeaway: Use LLMs where judgment and language flexibility help, while code controls the routes, rules, limits, and actions.

Offline exercise

- Revisit the diagram you made and mention what changes you would make based on the content discussed today

Over to Tutorial