Coding Assessment

Idempotent Extraction API • Time Limit: 2 Hours

QUICK SUMMARY

Task: Build an API with 2 endpoints that accepts documents, extracts structured data, and returns results idempotently.

Key Focus: System design, state modeling, error handling, and clear documentation.

Deliverable: GitHub repo with working code + comprehensive README and specs.

Tools: Use any coding assistants (Cursor, Windsurf, Codex, Claude Code, etc.) and any language/framework.

Core Requirements

API Endpoints

Endpoint	Purpose	Key Behavior
`POST /extract`	Submit document for extraction	Returns `request_id` and `status`. Same `idempotency_key` → same `request_id`
`GET /extract/{id}`	Retrieve extraction status and results	Returns status (COMPLETED/FAILED) with result or error

Extracted Data Schema

Your extractor should return JSON with these fields:

- `doc_type` → "invoice" | "receipt" | "unknown"
- `invoice_number` → string or null
- `invoice_date` → YYYY-MM-DD or null
- `total_amount` → number or null
- `currency` → 3-letter code or null

> **Implementation Choice:** Use either a mock extractor (regex/deterministic mapping) or a real LLM API based on your preference and time management.

Critical Scenario: Failure & Retry

Your implementation must handle this correctly:

1. Client submits with `idempotency_key = "xyz-999"`
2. Extractor fails (simulate this)
3. Client retries with same `idempotency_key`
4. **Required:** Returns same `request_id`. Either re-attempts extraction OR returns existing FAILED status (document your choice in README)

What We Assess

- **Entity and state modeling:** How you model data, relationships, and state transitions
- **Error handling:** Robustness under failure scenarios and edge cases
- **Design and architecture:** Overall system design, separation of concerns, and API design
- **Software design fundamentals:** Code organization, abstraction choices, and engineering principles
- **Documentation:** Quality of specs, README, and how you leverage AI coding tools in your workflow

Persistence Requirement

Data must survive server restarts. Use any storage mechanism (SQLite, PostgreSQL, JSON files, etc.).

Deliverables

Submit a public GitHub repository containing:

- **Working code** that can be run locally
- **Comprehensive README** with: setup instructions, design decisions, architecture overview, and how the failure/retry scenario is handled
- **API specification** documenting endpoints, request/response formats, and behavior
- **Data model documentation** showing entities, relationships, and state transitions
- **(Optional)** Notes on your development workflow and how you used coding assistants

Not Required: Authentication, UI, deployment configuration, background job infrastructure

Reference: Detailed Specifications

This section provides complete API contracts and examples for implementation.

Endpoint 1: POST /extract

Request Body

```
{
  "idempotency_key": "abc-123",
  "document_text": "Invoice ACME-009 for $120.00 dated 2025-12-01"
}
```

Response (First Submission)

```
{
  "request_id": "req_1",
  "status": "COMPLETED"
}
```

Response (Duplicate Idempotency Key)

```
{
  "request_id": "req_1",
  "status": "COMPLETED"
}
```

Idempotency Guarantee: Submitting the same `idempotency_key` multiple times must return the same `request_id` without re-processing. Processing can be synchronous or asynchronous—document your choice.

Endpoint 2: GET /extract/{request_id}

Response: Completed Extraction

```
{
  "request_id": "req_1",
  "status": "COMPLETED",
  "result": {
    "doc_type": "invoice",
    "invoice_number": "ACME-009",
    "invoice_date": "2025-12-01",
    "total_amount": 120.0,
    "currency": "USD"
  },
  "error": null
}
```

Response: Failed Extraction

```
{
  "request_id": "req_2",
  "status": "FAILED",
  "result": null,
  "error": {
    "code": "EXTRACTOR_TIMEOUT",
    "message": "Extractor timed out"
  }
}
```

Implementation Guidance

Extractor Options

Option A: Mock Extractor

- Use regex to parse common patterns (invoice numbers, dates, amounts)
- Or implement deterministic mappings for test inputs
- Include at least one simulated failure case

Option B: Real LLM API

- Keep prompts minimal and focused
- Validate and parse LLM responses defensively
- Handle timeouts and malformed responses

State Management Considerations

- Define clear state transitions (e.g., PENDING → COMPLETED, PENDING → FAILED)
- Ensure atomic writes when creating new extraction requests
- Consider how concurrent requests with the same idempotency_key should behave
- Document whether failed extractions can be retried or remain permanently failed

README Requirements

Your README should include:

1. **Setup Instructions:** How to install dependencies and run the service
2. **Architecture Overview:** High-level design decisions and component interactions
3. **Data Model:** Entities, relationships, and state management approach
4. **API Specification:** Clear documentation of endpoints, behaviors, and edge cases
5. **Design Rationale:** Key decisions you made and why (especially for the failure/retry scenario)
6. **Trade-offs:** What you prioritized and what you'd change with more time
7. **(Optional)** Development workflow and how coding assistants were used

> **Focus on Clarity:** We value well-documented design decisions, clear specifications, and thoughtful engineering over perfect handling of every edge case.