Lecture 2: Costs and Processing

Vertical Slices, MVPs, Crawling, Walking, and Running

University of Chicago

January 14, 2026

Outline

Outline

System Thinking: Breaking Down AI Tasks

- Before coding, we need to **think systematically**:
  1. How do we abstract and organize our code?
  2. How do we break tasks into inputs and outputs?
  3. How do we organize these into functions?
- **Plan before you code**
- Break down required tasks, *then* start coding
- How do we do this? What does it mean to "break tasks down"?

The Development Process

Three-Step Approach:

1. **Define a vertical slice**
   - What is the minimal end-to-end flow?
   - Start simple, prove it works
2. **Define the systems (functions) required**
   - Know the inputs and outputs of each piece
   - Simpler is better
3. **Then: Crawl $\rightarrow$ Walk $\rightarrow$ Run**
   - Start minimal, add features incrementally

Example Problem: Time Series Analysis

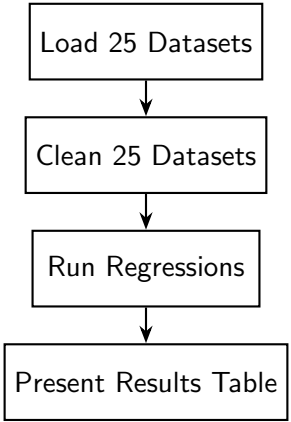
Task: Run a regression on 25 datasets

Context:

- We have data from 25 countries
- Each CSV contains education and GDP data over time
- We want to run a time series analysis on the relationship

Question: How do we break this down?

System Design: Complete Problem



This is the **complete problem**, but where do we start?

Inputs and Outputs: The Missing Piece

For each system component, we need to define:

- **Load Data**: What format for each column? How to handle dates? Any transformations?
- **Clean Data**: What cleaning steps? How do we handle missing values?
- **Run Regression**: What type? Linear? Time series?
- **Present Output**: Table? Graph? JSON? What metrics?

Inputs and Outputs: The Missing Piece

For each system component, we need to define:

- **Load Data**: What format for each column? How to handle dates? Any transformations?
- **Clean Data**: What cleaning steps? How do we handle missing values?
- **Run Regression**: What type? Linear? Time series?
- **Present Output**: Table? Graph? JSON? What metrics?

This is the complete problem, but we never want to start here!

We want to start with a **vertical slice**.

What is a Vertical Slice?

A vertical slice:

- Contains **everything** needed to go from top to bottom
- Works on a **subset** of inputs
- Proves the **end-to-end flow** works
- Is the **simplest possible** complete solution

When we build AI systems, we want to start with vertical slices.

Why Vertical Slices for AI?

AI systems have unique challenges:

- **Complex**: Many moving parts and dependencies
- **Uncertain failure points**: We aren't sure where things will break
- **Non-deterministic**: Errors appear in unexpected places
- **Iterative prompting**: Prompts need testing and refinement

With vertical slices:

- We build a viable end-to-end solution that works on a **subset** of inputs
- We discover integration issues **early**
- We can demonstrate **value quickly**

The Opposite: Horizontal Development

Horizontal development means:

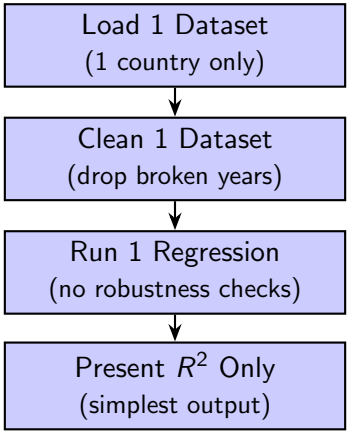
- Building for **all contingencies** at each level
- Handling **every possibility** before moving forward
- Creating **complete infrastructure** before testing end-to-end

For our regression example:

- Load all 25 datasets with full error handling
- Clean all datasets with every edge case covered
- Build multiple regression types and robustness checks
- Create multiple output formats and visualizations

Problem: You spend weeks building infrastructure before knowing if the core idea works!

Vertical Slice: Regression Example



This proves it works end-to-end with minimal effort!

Example: Personalized Marketing Emails

Task: Send AI-generated personalized marketing emails to leads

Vertical Slice Approach:

- Load **one** lead from the database
- Generate **one** email with LLM
- Print email to console for manual review
- If it looks good, move to next step

Horizontal Approach (what NOT to do):

- Build complete database integration for all leads
- Create email templates for all scenarios
- Build full SMTP sending infrastructure
- Set up monitoring and analytics before sending anything

Principle: Well-Defined Inputs and Outputs

- Each component should have:
  1. **Clear input contract**: What data does it expect? In what format?
  2. **Clear output contract**: What data does it return? In what format?
  3. **Single responsibility**: Does one thing well

- **Benefits:**
  - Easy to understand what each piece does
  - Easy to test each piece independently
  - Easy to replace or improve individual pieces
  - Clear error boundaries

Methodology: Crawl, Walk, Run

- **Crawl**: What is the minimum code to verify it's possible?
  - Hardcoded examples
  - Manual verification
  - Proof of concept that it can work in *some* instances
- **Walk**: What is the minimum code to verify it works in most cases?
  - Handle common edge cases
  - Basic error handling
  - Works reliably for typical inputs
- **Run**: What does the full production system look like?
  - Comprehensive error handling
  - Monitoring and logging
  - Scale and performance optimization

Crawl Phase in Detail

Goal: Prove the concept works at all

Characteristics:

- Use **hardcoded** or sample data
- **No error handling** (let it fail if something breaks)
- **Manual testing** (print statements, visual inspection)
- **Happy path only** (assume everything works)

Example - Resume Screening:

- Hardcode path to one resume PDF
- Extract skills with LLM, print to console
- Compare to hardcoded job requirements
- Print match score

If this doesn't work, nothing else matters!

Building Vertical Slices First

- **Vertical Slice**: End-to-end functionality for a narrow use case
  - Example: Process ONE resume through the entire pipeline
  - Proves the concept works end-to-end
  - Surfaces integration issues early
- **Then Build Horizontally**: Expand capabilities
  - Add batch processing
  - Add error handling
  - Add more sophisticated matching
  - Add caching and optimization

Key Takeaways: Building with AI

1. **Define the vertical slice** – what is the minimum thing you can get working?
   - Prove end-to-end flow works with simplest case
   - Identify integration issues early
2. **Figure out the inputs and outputs**
   - Each function needs clear contracts
   - Simpler is better
3. **Figure out the pricing**
   - Know your costs before scaling
4. **Expand horizontally after vertical slice works**
   - Add batch processing, error handling, features
   - Don't optimize or add features prematurely

Outline

Problem Background

- We have been hired by a company to help with their software developer hiring process
- Our goal is to build an AI-assisted resume scoring system
- For each resume, we want to score it so that we don't waste time on resumes we are unlikely to hire
- We are also cost conscious.
- Scores should be on a 0-100 scale.

Brainstorming

- How are we going to proceed here?
- What's our vertical slice?
- What are the system components?
- What are the inputs and outputs?
- How much will our decisions cost?

Let's Look at Our Inputs

- **Data**: `resumes_final.csv` has one row per resume (anonymized, taken from Kaggle)
- **Functions**: Our notebook has two functions already:
  1. `load_resumes()` - loads all the resumes into a dictionary
  2. `analyze_resume()` - runs a prompt against a resume with structured output
- There are two job reqs in the directory also: `job_req_senior.md` and `job_req_junior.md`
- Start with **Senior**

Define our system

- Do we have a good definition of each component?
- Do we know the inputs and outputs and what has to happen inside each system?

Next Step: Crawling!
Start with the simplest possible version

Resume Screening System: Crawl, Walk, Run

Phase 1: Vertical Slice (Crawl)

- Generate Score for a Single Resume for a single Job Req

Phase 2: Horizontal Expansion (Walk)

- Generate Score for **most** resumes for a single Job Req

Phase 3: Production (Run)

- Work on **all** resumes for **all** job reqs
- Optimize prompts and costs
- Add caching for repeated skills
- Add human-in-the-loop review
- Monitor accuracy and performance

Next Steps: Hands-On Practice

Goal: Build a working resume scorer

Your Task:

1. **Form teams** (2-3 people)
2. **Open the notebook**: `resume_screening.ipynb`
3. **Work through the examples**:
   - Load resume data
   - Use structured outputs to extract information
   - Match resumes to job requirements
4. **Key questions to answer**:
   - Can we create a score (0-100)?
   - How do we do that?
   - What prompt and schema do we need?
   - How much does it cost per resume?

Focus on the Crawl First!

Start simple, prove it works