Week 5: Context Engineering - Sub-agents and RAG

01. Session Goals

- Understand sub-agent architecture and use cases
- Use the Task tool to spawn specialized agents
- Define custom agents with focused capabilities
- Build multi-agent workflows for GTM pipelines
- Connect to vector databases for retrieval-augmented generation

02. Block 1: Theory - Why Sub-agents? (30 min)

The Problem with Single Agents
Complex workflows strain a single agent:

- Context window fills up with intermediate steps
- Focus dilutes across tasks
- Errors cascade

The real killer: context flooding.

Imagine your agent needs to research 10 companies. Without sub-agents, every web search result, every page fetch, every intermediate analysis floods the main agent's context. By the time you're on company #5, the agent has forgotten what it learned about company #1.

What Are Sub-agents?
A sub-agent is a separate agent instance that handles a focused subtask within a larger workflow. Think of it like a boss delegating to employees: you give them a clear brief, they work independently, and they report back with results. You don't see every step they took, just the output.

The key insight: sub-agents are black boxes for context management.

When a sub-agent runs:

1. It does all its work in its own context (web scrapes, file reads, analysis)
2. All those intermediate steps stay in the sub-agent's context
3. Only the final result returns to the main agent
4. The main agent's context stays clean

```
Without sub-agents:
+-------------------------------------------------------------+
| Main Agent Context                                          |
| - Task: Research 10 companies                               |
| - WebSearch result for Company 1 (500 tokens)               |
| - WebFetch page content (2000 tokens)                       |
| - Analysis reasoning (300 tokens)                           |
| - WebSearch result for Company 2 (500 tokens)               |
| - WebFetch page content (2000 tokens)                       |
| - ... context explodes ...                                  |
+-------------------------------------------------------------+

With sub-agents:
+-------------------------------------------------------------+
| Main Agent Context                                           |
| - Task: Research 10 companies                                |
| - Company 1 result: "Acme Corp, 500 employees, Series B..."  |
| - Company 2 result: "TechCo, 200 employees, bootstrapped..." |
| - ... clean, manageable context ...                          |
+-------------------------------------------------------------+
```

Key characteristics:

Isolated context: Each sub-agent starts fresh, focused only on its task
Own conversation: Sub-agent messages don't pollute the main agent's context
Specialized tools: You can restrict which tools a sub-agent can use
Defined model: Sub-agents can use different models (Haiku for speed, Opus for complexity)
Single level: Sub-agents cannot spawn their own sub-agents

Sub-agents are essential for building production agent systems. Without them, you're limited to what fits in a single context window with a single set of instructions.

Sub-agents Solve This

```
+-----------------------------------------------------+
|                   Main Agent                        |
|  "Process these 50 leads"                           |
+-----------------------------------------------------+
          |             |             |
          ▼             ▼             ▼
+-------------+  +-------------+  +-------------+
|  Enrichment |  |   Scorer    |  |   Writer    |
|    Agent    |  |    Agent    |  |    Agent    |
|             |  |             |  |             |
| Research    |  | Score based |  | Draft       |
| companies   |  | on rubric   |  | emails      |
+-------------+  +-------------+  +-------------+
```

Benefits:

- Focused context per task
- Parallel execution possible
- Specialized tools per agent

- Isolated failures

The Task Tool
The Task tool is how you invoke sub-agents in Claude Code. When Claude uses the Task tool, it spawns a new agent instance with its own isolated conversation.

```
> Use the code-reviewer agent to review this PR
```

How Task tool works:

1. Creates isolated context for sub-agent
2. Passes the prompt and configuration you specify
3. Gives sub-agent access to its allowed tools
4. Sub-agent runs independently until completion
5. Returns final result to main agent

Task tool parameters:

Parameter	Required	Description
`prompt`	Yes	The task for the sub-agent to perform
`description`	Yes	Short summary (3-5 words) of what the agent does
`subagent_type`	Yes	Which agent type to use
`model`	No	Override model (sonnet, opus, haiku)
`max_turns`	No	Limit API round-trips
`run_in_background`	No	Run asynchronously
`resume`	No	Continue from previous execution

Foreground vs Background:

By default, sub-agents run in the foreground: you wait for them to finish. For long-running tasks, use `run_in_background: true`. Background agents return an output file path you can check later.

Built-in Sub-agent Types
Claude Code comes with built-in agents optimized for common tasks:

Agent Type	Model	Use Case	Tools	Best For
`Explore`	Haiku	Codebase exploration	Glob, Grep, Read (read-only)	Quick searches, finding files, understanding code
`Plan`	Inherited	Architecture planning	All except Edit/Write	Designing implementation before coding
`Bash`	Inherited	Command execution	Bash only	Git operations, running scripts
`general-purpose`	Inherited	Flexible tasks	All tools	Complex multi-step work

When to use each:

Explore: Use for research tasks. It's fast (Haiku) and read-only, perfect for "find all files that..." or "how does X work in this codebase?"
Plan: Use before implementing features. Creates a plan file for user approval before any code changes.
Bash: Use for terminal operations like git commits, test runs, or build commands.
General-purpose: Use when you need full capabilities but want isolated context.

Three Ways to Create Sub-agents
1. Built-in agents (Claude Code CLI):

Use the types above. No configuration needed.

2. Filesystem-based agents (Claude Code CLI):

Create agent definitions in `.claude/agents/`:

```
.claude/agents/
+-- lead-enricher.md
+-- email-writer.md
+-- data-validator.md
```

Each file uses YAML frontmatter:

```
---
name: lead-enricher
description: Research companies and enrich lead data with firmographic info
model: haiku
tools: ["Read", "WebSearch", "WebFetch", "Write"]
---

# Lead Enricher

Research the company thoroughly. Find:
- Company size and employee count
- Industry and sub-industry
- Recent news and announcements
- Key decision makers and their roles
- Technology stack (if B2B tech)

Output a structured summary in markdown.
```

3. Programmatic agents (Agent SDK):

Define agents in code when using the SDK:

```
const result = await query({
  prompt: "Process these leads",
  options: {
    agents: {
      "lead-enricher": {
        description: "Research companies and enrich lead data",
        prompt: "Research thoroughly. Find size, industry, news, key people.",
        tools: ["Read", "WebSearch", "WebFetch", "Write"],
        model: "haiku"
      }
    }
  }
});
```

Agent Configuration Options

Whether using filesystem or SDK, agents support these configuration fields:

Field	Required	Description
`name`	Yes	Identifier for the agent (lowercase, hyphens)
`description`	Yes	What it does and when to use it. Critical for automatic invocation.
`prompt`	No	System instructions for the agent
`tools`	No	Array of allowed tools. Omit for all tools.
`model`	No	`haiku`, `sonnet`, or `opus`
`permissionMode`	No	`default`, `acceptEdits`, or `bypassPermissions`
`hooks`	No	PreToolUse/PostToolUse handlers

Tool restriction patterns:

```
# Specific tools only
tools: ["Read", "Grep", "Glob"]

# All tools except dangerous ones
tools: ["Read", "Write", "Edit", "Glob", "Grep", "WebSearch", "WebFetch"]

# Bash with command prefix restriction
tools: ["Read", "Bash(python:*)"]  # Only python commands
```

Permission modes:

- `default`: Prompts for approval on sensitive actions
- `acceptEdits`: Auto-approves file edits (useful for automation)
- `bypassPermissions`: No prompts (only for trusted, sandboxed environments)

Resuming Sub-agents
Sub-agents can be resumed to continue work with their full previous context. This is useful for:

- Long-running research tasks you check on later
- Iterative workflows where you add requirements
- Following up on partial results

When a sub-agent completes, it returns an agent ID. Use this ID to resume:

```
Main agent: "Research Acme Corp"
Sub-agent: [completes with agent_id: "abc123"]

Later...
Main agent: "Resume agent abc123 and also find their competitors"
Sub-agent: [continues with full previous context]
```

In the SDK, use the `resume` parameter:

```
await query({
  prompt: "Also find their competitors",
  options: {
    resume: "abc123"  // Previous agent ID
  }
});
```

Demo
Show a multi-agent workflow:

1. Main agent receives list of leads
2. Spawns enrichment agent for each company
3. Collects and summarizes results

---

03. Block 2: Lab 1 - Using Built-in Sub-agents (30 min)

Task: Explore Codebase with Sub-agent
Use the Explore agent for codebase discovery.

Step 1: Ask a broad question:

```
> How does the authentication system work in this codebase?
```

Watch Claude spawn an Explore agent. Notice it uses Haiku for speed.

Step 2: Request parallel exploration:

```
> I need to understand three things in parallel:
> 1. How data validation works
> 2. How errors are handled
> 3. How logging is configured
```

Notice multiple sub-agents working simultaneously. This is faster than sequential research.

Step 3: Create your first custom agent

Create `.claude/agents/company-researcher.md`:

```
---
name: company-researcher
description: Research a company for sales preparation. Use when preparing for sales calls or evaluat
model: haiku
tools: ["Read", "WebSearch", "WebFetch"]
---

# Company Researcher

Research the target company and provide a concise briefing:

1. **Company Overview** - What they do, size, industry
2. **Recent News** - Last 3 months of relevant news
3. **Key People** - Leadership team and decision makers
4. **Potential Pain Points** - Based on their industry and size

Keep the briefing under 500 words. Focus on actionable intelligence for a sales conversation.
```

Step 4: Test your custom agent:

```
> Use company-researcher to research Stripe
```

Verify your agent is discovered and invoked.

Success Criteria

- [ ] Observe built-in sub-agent spawning
- [ ] See parallel execution
- [ ] Create a custom agent definition
- [ ] Successfully invoke your custom agent

Discussion Questions

1. When did Claude choose to use sub-agents automatically?
2. How did the custom agent differ from using the general-purpose agent?
3. What tools did each sub-agent have access to?

---

04. BREAK (10 min)

---

05. Block 3: Theory - Building Multi-Agent Workflows (30 min)

Workflow Patterns

Sequential Pipeline:

```
Lead -> [Enrichment] -> [Scoring] -> [Email Draft] -> Output
```

Parallel Processing:

```
        +--> [Agent 1] -->+
Lead ---+--> [Agent 2] -->+--> Combine
        +--> [Agent 3] -->+
```

Hierarchical Delegation:

```
Main Agent
+-- Research Manager
|   +-- Company Researcher
|   +-- Contact Researcher
+-- Output Manager
    +-- Report Writer
    +-- Email Writer
```

Designing Agent Responsibilities
Good decomposition:

- Each agent has clear, focused purpose
- Minimal overlap between agents
- Well-defined inputs and outputs

Anti-patterns:

- Too many tiny agents (overhead)
- Agents that do "everything"
- Circular dependencies

!Too many agents anti-pattern (images/week5-too-many-agents-anti-pattern.png)

Don't be this person. "Whimsy-injector.md" is not a real agent you need.

Agent Communication
Sub-agents return results to parent:

```
Main Agent: "Use lead-enricher to research Acme Corp"
            v
Sub-agent:  [researches, finds info]
            v
Sub-agent:  Returns: "Acme Corp: 250 employees, Series B..."
            v
Main Agent: [receives result, continues workflow]
```

Error Handling
When sub-agents fail:

- Main agent receives error message
- Can retry with different approach
- Can skip and continue with others

```
> If enrichment fails for a company, note it as "Research Failed" and continue with the next lead.
```

Context, Memory, and Retrieval

Sub-agents are powerful, but what happens when you need agents to work with large knowledge bases? This is where retrieval-augmented generation (RAG) comes in.

The evolution:

- RAG started as "embed documents, find similar chunks, stuff into prompt"
- Modern agents treat retrieval as just another tool
- Claude can decide when to search, what to search for, and how to use results

RAG vs Agentic Search:

Traditional RAG	Agentic Search
Fixed retrieval pipeline	Agent decides when to retrieve
Same query for all questions	Agent reformulates queries
Top-K results stuffed in prompt	Agent filters and synthesizes
One-shot retrieval	Iterative search and refinement

When to use vector search vs filesystem:

Use Case	Best Approach
Structured data (CSVs, databases)	SQL/grep - precise matching
Unstructured docs (PDFs, notes)	Vector search - semantic similarity
Code search	Grep + embeddings hybrid
Finding similar past deals/cases	Vector search - similarity matching

Connecting to vector databases via MCP:

For this workshop, we'll use Vectorize.io, which provides:

- A visual UI for uploading and managing documents
- Automatic vectorization of your files (PDFs, docs, text)
- Built-in MCP integration for Claude Code
- No code required for basic RAG workflows

Once connected, Claude can:

- Search for semantically similar content
- Retrieve context for complex questions
- Enrich data based on proprietary knowledge

Tracking Sub-agent Progress (SDK)

When using the Agent SDK, you can track sub-agent progress through streaming:

```
import { query } from "@anthropic-ai/claude-agent-sdk";

for await (const message of query({
  prompt: "Research these 5 companies",
  options: {
    agents: {
      "researcher": {
        description: "Research a company",
        tools: ["WebSearch", "WebFetch"]
      }
    }
  }
})) {
  // Messages from sub-agents include parent_tool_use_id
  if (message.parent_tool_use_id) {
    console.log(`Sub-agent progress: ${message.content}`);
  }

  // Tool calls show which agent is acting
  if (message.type === 'tool_use') {
    console.log(`Tool: ${message.name}`);
  }
}
```

What you can track:

- When sub-agents start and complete
- Which tools each sub-agent uses
- Intermediate results before final output
- Errors or retries within sub-agents

---

06. Block 4: Lab 2 - Build a GTM Pipeline (45 min)

Task: Create a Lead Processing Pipeline
Build a multi-agent workflow that:

1. Takes a list of leads
2. Enriches each with company research
3. Scores based on your rubric
4. Drafts personalized outreach

Step 1: Create Specialized Agents
First, create the agents that will power your pipeline.

Create `.claude/agents/lead-enricher.md`:

```
---
name: lead-enricher
description: Enrich a lead with company research. Use when you need firmographic data about a compan
model: haiku
tools: ["Read", "WebSearch", "WebFetch"]
---

# Lead Enricher

Research the company and provide:

1. **Company Size** - Employee count, revenue if public
2. **Industry** - Primary industry and sub-vertical
3. **Recent News** - Funding, product launches, leadership changes
4. **Key People** - Decision makers relevant to our product
5. **Tech Stack** - Known technologies they use

Output as structured JSON:
{
  "company": "...",
  "size": "...",
  "industry": "...",
  "news": ["..."],
  "key_people": ["..."],
  "tech_stack": ["..."]
}
```

Create `.claude/agents/email-drafter.md`:

```
---
name: email-drafter
description: Draft personalized outreach emails based on lead data. Use when you have enriched lead
model: sonnet
tools: ["Read", "Write"]
---

# Email Drafter

Write a short, personalized outreach email:

1. Reference something specific about their company (from research)
2. Connect to a pain point relevant to their industry/size
3. Clear, low-friction call to action
4. Under 100 words

Be conversational, not salesy. No generic openers like "I hope this email finds you well."
```

Step 2: Design the Workflow

Create `.claude/skills/lead-pipeline/SKILL.md`:

```
---
name: lead-pipeline
description: Process leads through enrichment, scoring, and email drafting. Use when processing a ba
---

# Lead Processing Pipeline

This skill orchestrates multiple specialized agents to process leads.

## Pipeline Stages

### Stage 1: Enrichment
For each lead, research the company:
- Company size and industry
- Recent news and announcements
- Key decision makers
- Technology stack (if B2B tech)

Use sub-agent: "Research [Company Name] thoroughly"

### Stage 2: Scoring
Apply the lead-scorer skill to score each enriched lead.

### Stage 3: Email Drafting
For leads scoring 60+, draft personalized outreach:
- Reference specific company details
- Address likely pain points
- Clear call to action

### Execution

Process leads sequentially or in small batches (3-5) to manage context.

### Output Format

| Lead | Company | Enrichment Summary | Score | Email Draft |
|------|---------|--------------------|-------|-------------|
| ... | ... | ... | ... | ... |

Save detailed results to `output/pipeline-results.json`
```

Step 2: Test with Startup Data

Use the startup funding database to find promising companies:

```
> Query the funding database for AI startups that raised Series A in 2024.
> Pick the top 5 by funding amount and run them through the lead pipeline.
```

Watch the pipeline:

1. Query database for candidates
2. Spawn enrichment sub-agents for each
3. Apply scoring based on company data

4. Generate outreach emails

Step 3: Iterate and Improve
Based on results:

- Adjust enrichment prompts
- Tune scoring weights
- Refine email templates

Deliverable

- Working lead-pipeline skill
- Screenshot of pipeline execution
- Sample output with 5 processed leads

---

07. Bonus Lab: RAG with Vectorize.io (Optional)

Scenario: Enrich Deals with Proprietary Knowledge
Your company has historical deal notes, win/loss analyses, and sales playbooks. When processing new leads, you want to:

- Find similar past deals
- Pull relevant playbook sections
- Get recommendations based on what worked before

This is a perfect use case for vector search + agents.

Step 1: Set Up Vectorize.io
Vectorize.io provides a visual workflow builder for RAG pipelines with direct Claude Code integration via MCP.

1. Create an account at vectorize.io (https://vectorize.io)
2. Create a new pipeline:
   - Click "New Pipeline"
   - Name it "Sales Knowledge Base"
   - Choose your embedding model (OpenAI or Cohere recommended)
3. Upload your documents directly in the UI:
   - Sales playbooks (PDF, Word, Markdown)
   - Historical deal summaries
   - Win/loss analysis reports
   - Product documentation
   - Competitor battlecards
4. Configure retrieval settings:
   - Set chunk size (500-1000 tokens recommended)
   - Enable metadata filtering if you want to filter by deal type, industry, etc.

Step 2: Create an MCP Agent in Vectorize
1. Navigate to Agents in the Vectorize dashboard
2. Create a new MCP Agent:

- Name: "Sales Knowledge Agent"
- Select your pipeline as the data source
- Configure the retrieval function:
- Function name: `search_knowledge_base`
- Description: "Search historical deals, playbooks, and sales documentation"

3. Generate API credentials:
   - Go to Agent API Keys
   - Create a new key named "Claude Code"
   - Copy the API key and Agent ID

Step 3: Connect to Claude Code
Run this command to add the Vectorize MCP server:

```
claude mcp add vectorize-mcp \
  --env VECTORIZE_API_KEY=YOUR_API_KEY \
  -- npx -y mcp-remote@latest \
  https://agents.vectorize.io/api/agents/YOUR_AGENT_ID/mcp \
  --header "Authorization: Bearer ${VECTORIZE_API_KEY}"
```

Verify the connection:

```
claude mcp list
```

You should see `vectorize-mcp` in the list.

Step 4: Test the RAG Integration
Start a new Claude Code session and test:

```
> Search our knowledge base for deals we won in the Healthcare industry.
> What were the common winning factors?
```

Claude will automatically use the Vectorize MCP to search your indexed documents and return relevant context.

More example queries:

```
> Find information about how we handle security objections
```

```
> What's our playbook for selling to enterprise companies with 1000+ employees?
```

```
> Search for deals where we competed against [Competitor Name]. What worked?
```

Step 5: Integrate into Your Pipeline
Create `.claude/agents/knowledge-enricher.md`:

```
---
name: knowledge-enricher
description: Enrich leads with insights from our sales knowledge base using Vectorize. Use when you
model: haiku
tools: ["mcp__vectorize-mcp__search_knowledge_base"]
---

# Knowledge Enricher

Search our sales knowledge base to provide context for new leads and deals.

## Process

1. **Identify search criteria** from the lead:
   - Industry vertical
   - Company size range
   - Likely pain points based on role/title

2. **Search the knowledge base** for:
   - Similar past deals (won and lost)
   - Industry-specific playbook sections
   - Relevant objection handling

3. **Return structured insights**:
```

{

"similar_deals": [

{"company": "...", "outcome": "...", "key_factors": "..."}

],

"playbook_guidance": "...",

"recommended_approach": "...",

"potential_objections": ["..."]

}

Step 6: Run the Full Pipeline

```
> Process this lead through the full pipeline with knowledge enrichment:
>
> Company: MedTech Solutions
> Industry: Healthcare
> Size: 300 employees
> Contact: VP of Operations
>
> 1. Use knowledge-enricher to find similar deals and playbook guidance
> 2. Apply lead scoring
> 3. Draft a personalized email that incorporates the lessons learned
```

What Makes Vectorize Different

Feature	Benefit
Visual UI	Upload files directly, no code needed
Workflow builder	Configure chunking, embedding, retrieval visually
MCP integration	Native Claude Code support via agent MCP
Hosted infrastructure	No vector DB to manage
Automatic updates	Re-upload docs and vectors update automatically

Example Queries That Shine with Vector Search
Vector search finds semantically similar content, not just keyword matches:

Query	What Vector Search Finds
"Deals where budget was the main concern"	Deals mentioning cost, pricing, ROI, budget constraints
"Skeptical technical buyers"	Deals with CTOs who needed proof, security reviews, POCs
"Fast-moving startups"	Deals with short cycles, founder-led, quick decisions

Deliverable

- [ ] Vectorize.io account created
- [ ] Documents uploaded to pipeline
- [ ] MCP agent configured and connected to Claude Code
- [ ] Knowledge enricher agent working
- [ ] Sample query showing relevant results from your docs

---

08. Wrap-Up (15 min)

Key Takeaways
1. Sub-agents = Focused Workers - Isolated context, specialized tools
2. Task Tool - Built-in way to spawn sub-agents
3. Design Matters - Clear responsibilities, defined I/O
4. Orchestration - Main agent coordinates, sub-agents execute

Homework
Part 1: Extend Your Pipeline

1. Add a new stage to your pipeline. Examples by domain:

Domain	New Stage Ideas
GTM/Sales	Competitor research, social media analysis, intent signals
Developer Tools	Security scanning, test coverage analysis, dependency check
Content/Marketing	SEO analysis, competitor content audit, distribution planning
Customer Support	Sentiment analysis, FAQ matching, priority scoring
Operations	Compliance checking, audit trail, approval routing

2. Create a custom agent definition for this stage
3. Test with 10+ items from your dataset
4. Document:
   - Pipeline diagram
   - Agent responsibilities
   - Sample results

Part 2: Multi-Source Analysis (Explore on Your Own)

Real data analysis often requires context from multiple sources. When you see an anomaly in your funding data, you might want to know: Was there industry news? Did a major competitor get acquired? Was there a market shift?

This is how professional data scientists work—they don't just query data, they synthesize information from multiple sources to explain what happened and why.

Your task: Connect at least one additional MCP server and combine it with the startup funding database.

Recommended MCPs to explore:

MCP	What It Enables	Example Use Case
Slack	Search channels, read messages	Find team discussions about companies in your database
Linear/Jira	Query issues and projects	Correlate product releases with funding trends
Notion	Query databases and pages	Pull company research notes alongside funding data
Google Drive	Search documents	Find historical analysis or meeting notes

Setup (pick one):

```
# Slack
claude mcp add slack

# Linear
claude mcp add linear

# Notion
claude mcp add --transport http notion https://mcp.notion.com/mcp
```

Example workflow:

1. Query funding database: "Which AI companies raised the largest rounds last quarter?"
2. Search Slack/Notion: "Find any team discussions or notes about these companies"
3. Synthesize: Write a 3-bullet summary combining quantitative funding data with qualitative context

Deliverable:

Document your multi-source analysis showing:

- The data query and results from the funding database
- Context gathered from your connected MCP
- Your synthesized explanation (what the data shows + what the context reveals)

Next Week Preview
Week 6: Agent SDK (TypeScript) - Running agents headlessly at scale

---

09. Facilitator Notes

Common Issues

1. Agent not found: Check that `.claude/agents/` directory exists and file has `.md` extension
2. Agent not triggering: Description doesn't match what user asked. Add more trigger keywords.
3. YAML syntax errors: Validate frontmatter. Use online YAML validator if needed.
4. Sub-agent not spawning: Task tool might not be in allowed tools list
5. Results not returning: Sub-agent may have errored. Check Claude's output for errors.
6. Slow execution: Too many sequential sub-agents. Use parallel execution where possible.
7. Context overflow: Sub-agents doing too much in one task. Break into smaller agents.
8. MCP tools not available: Background sub-agents cannot access MCP tools. Run in foreground for MCP.

Key Constraints to Emphasize

- Sub-agents cannot spawn other sub-agents (single level only)
- Background sub-agents auto-deny unpermitted operations
- Each sub-agent starts with fresh context (no memory from main agent except the prompt)
- Use Haiku for fast, simple tasks. Use Sonnet/Opus for complex reasoning.

Optimization Tips

- Batch similar tasks together (e.g., enrich 5 leads at once)
- Use parallel execution when tasks are independent
- Keep sub-agent scope narrow and well-defined
- Cache repeated lookups to avoid redundant web searches

- Use `model: haiku` for research and data gathering
- Use `model: sonnet` for writing and complex analysis

Reference: Claude Models Overview ■

Timing Adjustments

- Lab 1 can be shortened if concepts are clear
- Creating custom agents (Lab 1 Step 3-4) is essential. Don't skip.
- Lab 2 pipeline can be simplified to 2 stages if time is short
- Full pipeline with all agents can extend to homework