Module 2: Architectural Planning and System Design, Lesson 4: Prompting for Mermaid.js Architecture Diagrams

Hands-On: Three Diagrams, One Session — Live TaskFlow Architecture Demo

Overview

This demo proves three claims from the lecture in real time:

- AI-generated Mermaid syntax renders in mermaid.live with little to no editing when the prompt specifies the output format explicitly
- Iterating in a single chat session refines a diagram faster than any drag-and-drop tool — we will grow one diagram through three rounds
- Diagrams-as-code belong in the repo — every artifact we produce is a markdown-ready code block that GitHub renders natively

You will generate a request-flow flowchart, an OAuth sequence diagram, and a full Container-level C4 architecture diagram for the TaskFlow application using three prompts in the same AI chat window. The student lab extends this to produce a complete docs/architecture directory.

Before You Start

- Open your AI tool of choice: Claude.ai, Cursor, or GitHub Copilot Chat
- Open mermaid.live in a second browser tab — this is where we validate every generated diagram
- Have a scratch file ready named architecture.md — we will paste each diagram inside a ```mermaid code fence
- Keep the four-part Mermaid prompt template visible on screen or a second monitor throughout
- Do NOT start a new chat between prompts — session continuity is the point

> **Remember**
> Architecture diagrams go stale the moment someone exports the PNG. Mermaid-as-code fixes that because the diagram lives next to the source. We are going to produce three production-ready diagrams in one session and validate each one in mermaid.live before embedding it.

During Hands-On — Keep Visible if Possible

1. ROLE — senior software architect using C4 and Mermaid.js
2. INSTRUCTION — diagram type + "Return only the Mermaid code block, no prose"
3. CONTEXT — system name, components, relationships, audience
4. EXAMPLE — short reference snippet in the style you want
5. VERSION — "use valid Mermaid version 10 syntax only"

Prompt 1 — Request-Flow Flowchart

> **What to Learn**
> Here is the four-part Mermaid prompt in action. You specify the diagram type, name every component, add decision diamonds for the branching logic, and tell the AI to return only the code block. Paste into mermaid.live. A rendered diagram in under a minute.

Paste this into the AI chat:

```
ROLE: You are a senior software architect who documents systems
using Mermaid.js and writes concise, valid syntax.

INSTRUCTION: Generate a Mermaid.js flowchart using `graph TD`
showing the end-to-end request flow for a user submitting a new
task in the TaskFlow app. Use decision diamonds for JWT auth
success and failure. Return only the Mermaid code block — no
prose explanation. Use valid Mermaid version 10 syntax only.

CONTEXT:
System: TaskFlow — a team task management app with AI
prioritization.
Participants in order:
  - Browser client
  - API gateway
  - JWT auth middleware  (decision: valid vs invalid)
  - Task service
  - PostgreSQL database
  - AI prioritization service  (async, fire-and-forget)
Expected responses: 401 on auth failure, 202 Accepted on task
creation.

EXAMPLE STYLE:
```

```
graph TD
    A[Client] --> B[Gateway]
    B --> C{Auth?}
    C -->|Yes| D[Service]
    C -->|No| E[401]
```

> **What you should get**
> A graph TD block with seven to nine nodes. Browser flows to Gateway, Gateway flows to the Auth decision diamond. The Yes path reaches the Task Service, which writes to PostgreSQL and asynchronously messages the AI priority service. The No path returns 401. The Task Service returns 202 Accepted at the end.

You should ask:
"Why do we tell the AI to return only the code block?"

Answer: Without that line, the AI wraps the syntax in a paragraph explaining what it did, and you have to manually extract the code. Over dozens of diagrams that friction compounds. One line in the prompt saves you real time.

Tip: Paste the output into mermaid.live immediately. If you see a parse error, nine times out of ten it is an unescaped parenthesis or colon inside a label — wrap that label in quotes, e.g. B["API Gateway (Kong)"].

Prompt 2 — OAuth Sequence Diagram, Then Iterate for Error Paths

> **Remember**
> Same chat window. The AI already knows we are working on TaskFlow — I do not repeat the system description. I switch diagram types, specify the participants, ask for autonumber, then add error paths in a follow-up. This is the iteration pattern: happy path first, failure modes second.

Continue in the same chat session — do NOT start a new one:

```
Still in the TaskFlow context. Generate a Mermaid.js sequence
```

```
diagram for the OAuth 2.0 authorization code flow.

PARTICIPANTS: User, Browser, TaskFlow API, Google Auth.

INSTRUCTION: Show the authorization-code redirect, the code-for-
token exchange, and JWT issuance back to the browser. Use
`autonumber` so every interaction is labeled with a step number.
Return only the Mermaid code block. Use valid Mermaid version 10
syntax only.
```

> **What you should get**
> A sequenceDiagram block starting with autonumber and four declared participants. Roughly eleven numbered steps — click sign in, redirect to Google, enter credentials, redirect back with auth code, exchange code for tokens, set session cookie, load dashboard. Every arrow is labeled.

Now iterate — add error paths in one follow-up prompt:

```
Add `alt` blocks to the same diagram for:
  1. User cancels the Google consent screen
  2. Google returns invalid_grant on the token exchange
  3. TaskFlow API times out loading the user profile — with
     one retry before surfacing a 503

Return the complete updated diagram, not a diff. Same rules:
only the code block, Mermaid version 10 syntax.
```

> **What you should get**
> The same diagram with three alt blocks inside the main flow. You now have a sequence diagram that documents the happy path and the failure modes — the kind of artifact you reference in an incident post-mortem.

Validate in mermaid.live. If the alt blocks fail to render, the condition text almost always has a special character — "alt Invalid Grant" renders, "alt Invalid Grant (Google returns 401)" does not. Keep condition text simple and let the notes carry the detail.

Prompt 3 — Container-Level C4 Architecture Diagram

> **What to Learn**
> This is the capstone diagram for the whole module. One prompt gives you every service, every database, every external dependency, with protocols labeled on every arrow and components grouped by infrastructure boundary using subgraphs. This is the diagram that goes at the top of the architecture README.

Third prompt — still the same session:

```
Still TaskFlow. Generate a Mermaid.js flowchart representing
the Container-level C4 diagram for the entire system.

COMPONENTS:
  Clients:   React web app, iOS mobile app
  Backend:   API gateway, auth service, task service,
             AI priority service, notification service
  Data:      PostgreSQL, Redis, S3
  External:  OpenAI API

INSTRUCTION: Label every arrow with the protocol and data
format — e.g. "HTTPS / JSON", "gRPC / Protobuf", "SQL",
"AMQP". Group components into three `subgraph` blocks:
Client, Backend Services, Data Layer. External dependencies
render outside the subgraphs. Return only the Mermaid code
block. Use valid Mermaid version 10 syntax only.
```

> **What you should get**
> A graph LR with three labeled subgraphs. Two client nodes on the left flow into the gateway. Four backend services with explicit connections to each other and to the data stores. Three data store nodes. One external OpenAI node. Every arrow labeled with protocol and data format. Every subgraph closed with end.

Validate in mermaid.live. The most common miss is a subgraph missing its `end` keyword — the contents still render but they are not visually grouped. If that happens, the fix is one follow-up: "The Backend Services subgraph is missing its end keyword — fix it." Paste. Re-validate.

Commit the diagram to the repo:

```
mkdir -p docs/architecture && pbpaste > docs/architecture/taskflow-c4.md
```

Open the file on GitHub. The diagram renders natively inside the markdown file. No plugin, no export, no attachment. That is diagrams-as-code — living documentation that travels with the source.

Wrap-Up — The Pattern, the Module Capstone

> **Remember**
> Three prompts. One session. A request-flow flowchart, an OAuth sequence diagram with error paths, and a Container-level C4 diagram with protocol labels. Every one of them renders in mermaid.live. Every one of them commits to docs/architecture as a markdown file that GitHub renders natively. The AI did the syntax work. You owned the validation and the architectural judgment.

Key takeaways:

- Always include "Return only the Mermaid code block, no prose" — one line, compounding time savings
- Always include "Use valid Mermaid version 10 syntax only" — avoids deprecated features that render wrong
- Iterate in rounds: happy path → error paths → observability → simplify for non-technical stakeholders
- Validate every diagram in mermaid.live before embedding — a broken diagram in a README is worse than no diagram
- Commit diagrams to docs/architecture — they version with your code and cannot drift
- Module 2 wrap: PRD (Lecture 2) + schema (Lecture 3) + these diagrams = a complete design package, all in Git, all generated with AI and validated by you