DYNA Fall 2025
A DYALOG USER MEETING

Dyalog + AI = ...?

Stefan Kruger

Who am I?

- New APLer (first decade) -- 3 years at Dyalog
- Previously at IBM for long stint
- Computer Science background
- @ Dyalog: provider of outside perspective

Points

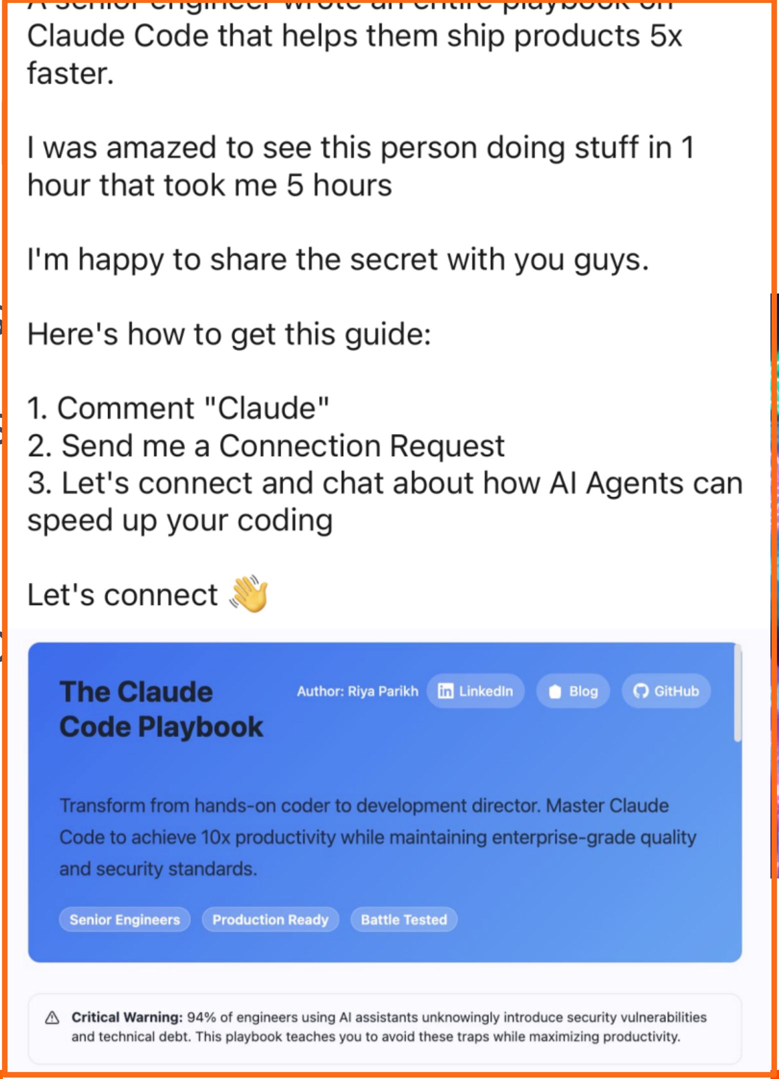
- AI for developer productivity
- Applications
  - Semantic search
  - Retrieval-Augmented Generation (RAG)
- Client libraries

AI for developer productivity

- Value proposition:
  - 10x yourself!
  - Remove 'drudge'!
  - ... (mumbles)
  - $PROFIT!

AI for deve

- Value propos
  - 10x yours
  - Remove '
  - … (mumb
  - $PROFIT!



Claude Code that helps them ship products 5x faster.

I was amazed to see this person doing stuff in 1 hour that took me 5 hours

I'm happy to share the secret with you guys.

Here's how to get this guide:

1. Comment "Claude"
2. Send me a Connection Request
3. Let's connect and chat about how AI Agents can speed up your coding

Let's connect 👋

The Claude Code Playbook

Author: Riya Parikh

Transform from hands-on coder to development director. Master Claude Code to achieve 10x productivity while maintaining enterprise-grade quality and security standards.

⚠ **Critical Warning:** 94% of engineers using AI assistants unknowingly introduce security vulnerabilities and technical debt. This playbook teaches you to avoid these traps while maximizing productivity.

Reality check

AI *does* improve developer productivity

Reality check

…if you're in its wheelhouse

Reality check

Small, constrained tasks; "bookwork"

Reality check

Mechanical refactoring of code

Reality check

Improves *velocity*, not *competency*!

Reality check

Does *not* turn interns into senior devs

Reality check

APL: *improving (but must try harder)*

Pace of improvement: off charts

Pace of improvement: off charts

- World looks different today from ~~18 months ago~~ last week

Pace of improvement: off charts

- World looks different today from ~~18 months ago~~ last week
- Claude Opus 4.1 perfectly capable of explaining APL code

Pace of improvement: off charts

- World looks diff today from ~~18 months ago~~ last week
- Claude Opus y capable of explaining APL code

Latest LLM
Anthropic

Pace of improvement: off charts

- World looks different today from ~~18 months ago~~ last week
- Claude Opus 4.1 perfectly capable of explaining APL code
- Writes mostly correct APL in constrained domains

Pace of improvement: off charts

- World looks different today from ~~18 months ago~~ last week
- Claude Opus 4.1 perfectly capable of explaining APL code
- Writes mostly correct APL in constrained domains
- 18-24 months behind the curve

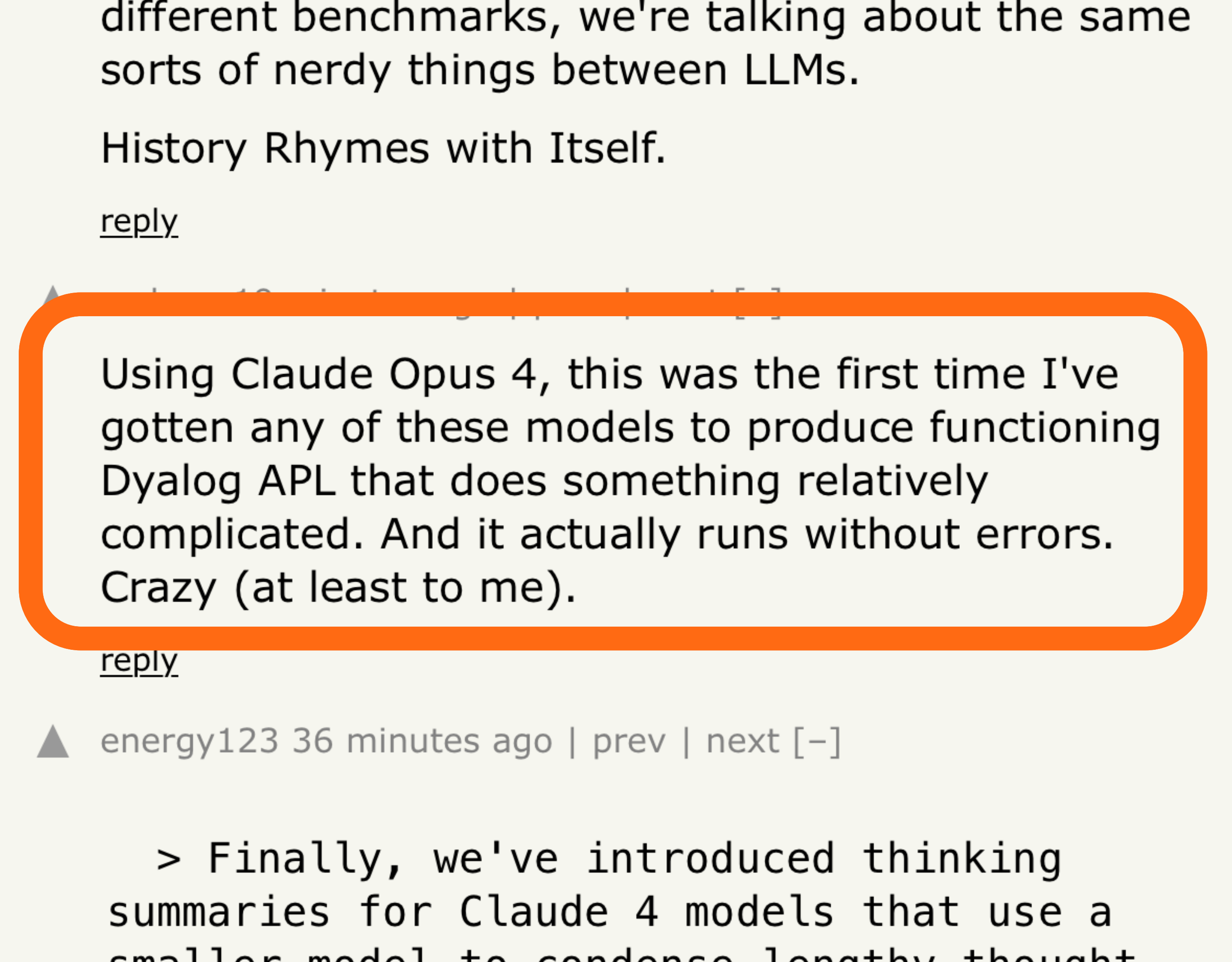
Pace of improvement: off charts

- World looks different today from ~~18 months ago~~ last week
- Claude Opus 4.1 perfectly capable of explaining APL code
- Writes mostly correct APL in constrained domains
- 18-24 months behind the curve

CONJECTURE

Pace

- Wo
- Clau
- Wri
- 18-



different benchmarks, we're talking about the same sorts of nerdy things between LLMs.

History Rhymes with Itself.

reply

Using Claude Opus 4, this was the first time I've gotten any of these models to produce functioning Dyalog APL that does something relatively complicated. And it actually runs without errors. Crazy (at least to me).

reply

▲ energy123 36 minutes ago | prev | next [–]

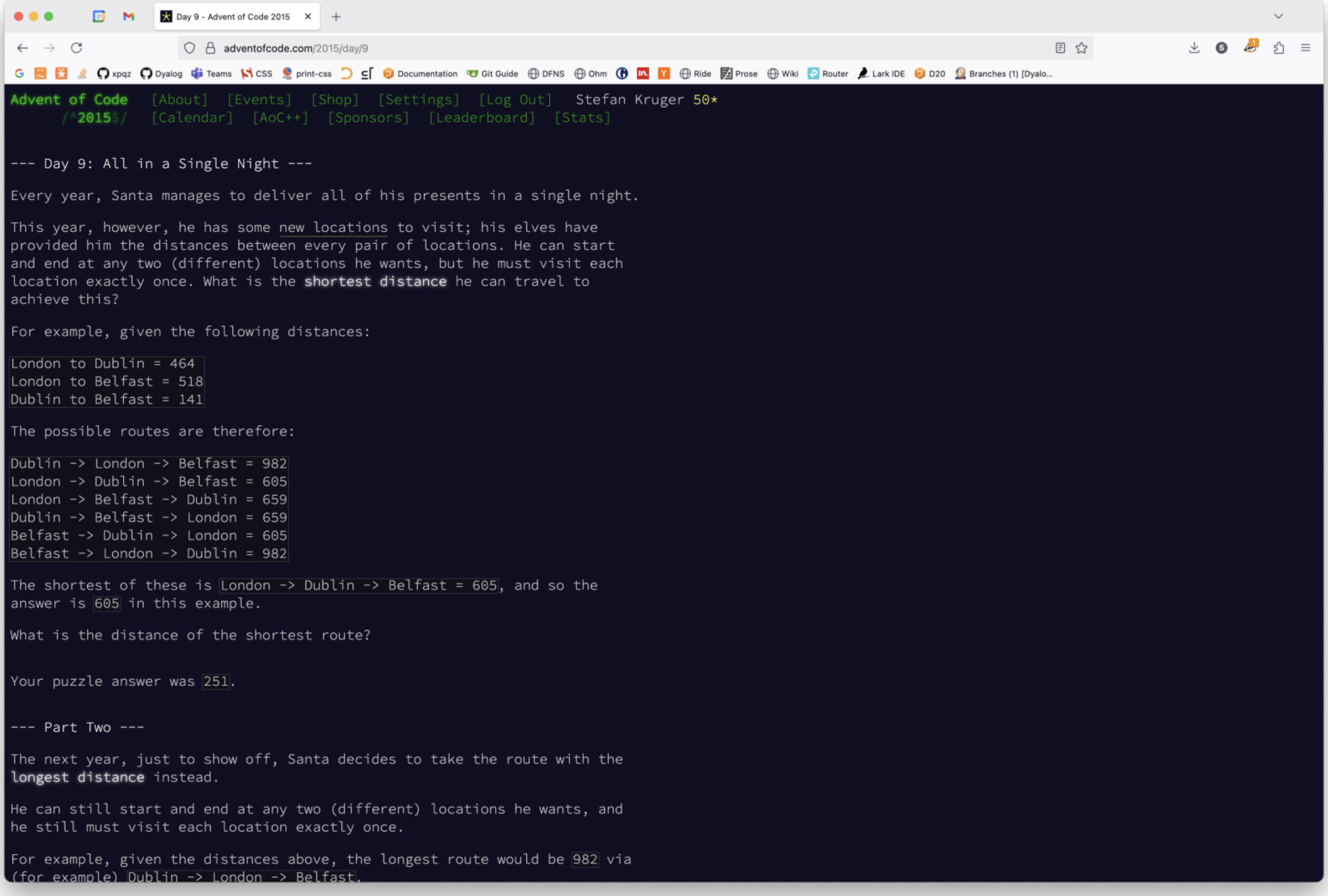
```
  > Finally, we've introduced thinking
summaries for Claude 4 models that use a
smaller model to condense lengthy thought
```

ek

CLAUDE
ME
GPT-5-CODEX

Pace of improvement: off charts

- World looks different today from ~~18 months~~ ago last week
- Claude Opus 4.1 perfectly capable of explaining APL code
- Writes mostly correct APL in constrained domains
- 18-24 months behind the curve


Day 9 - Advent of Code 2015
adventofcode.com/2015/day/9
xpqz
Dyalog
Teams
CSS
print-css
Documentation
Git Guide
DFNS
Ohm
Ride
Prose
Wiki
Router
Lark IDE
D20
Branches (1) [Dyalo...
Advent of Code
/^2015$/
[About] [Events] [Shop] [Settings] [Log Out]
Stefan Kruger 50*
[Calendar] [AoC++] [Sponsors] [Leaderboard] [Stats]

--- Day 9: All in a Single Night ---

Every year, Santa manages to deliver all of his presents in a single night.

This year, however, he has some new locations to visit; his elves have provided him the distances between every pair of locations. He can start and end at any two (different) locations he wants, but he must visit each location exactly once. What is the **shortest distance** he can travel to achieve this?

For example, given the following distances:

```
London to Dublin = 464
London to Belfast = 518
Dublin to Belfast = 141
```

The possible routes are therefore:

```
Dublin -> London -> Belfast = 982
London -> Dublin -> Belfast = 605
London -> Belfast -> Dublin = 659
Dublin -> Belfast -> London = 659
Belfast -> Dublin -> London = 605
Belfast -> London -> Dublin = 982
```

The shortest of these is `London -> Dublin -> Belfast = 605`, and so the answer is `605` in this example.

What is the distance of the shortest route?

Your puzzle answer was `251`.

--- Part Two ---

The next year, just to show off, Santa decides to take the route with the **longest distance** instead.

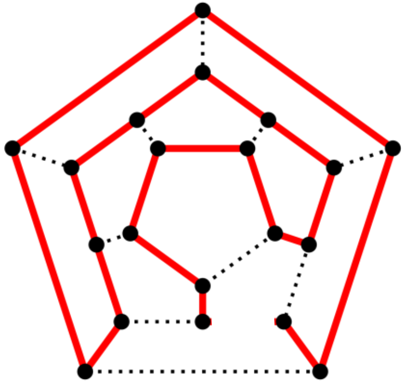
He can still start and end at any two (different) locations he wants, and he still must visit each location exactly once.

For example, given the distances above, the longest route would be `982` via (for example) `Dublin -> London -> Belfast`.

Start and end at any two (different) locations. Must visit each location **exactly once**. What is the **shortest distance**?

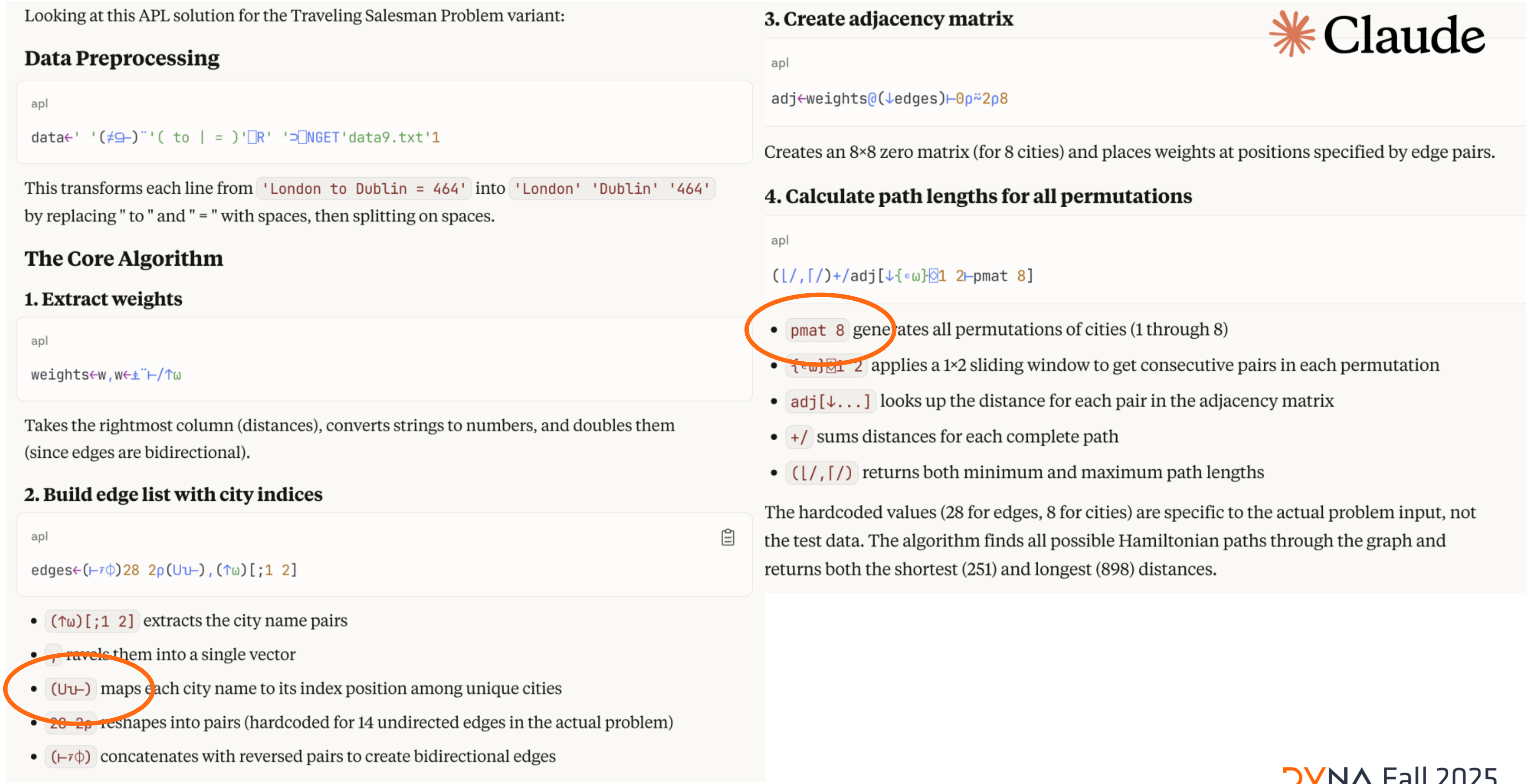
Given the following distances:

```
London to Dublin  = 464
London to Belfast = 518
Dublin to Belfast = 141
```



The shortest is `London → Dublin → Belfast = 605`

```apl
'pmat'⎕CY'dfns'
data←' '(≠⊆⊢)¨','(to|=)'⎕R' '⊃⎕NGET'data9.txt'1

Day9←{
    weights←w,w←⍎¨⊢/↑ω
    edges←(⊢⍪⌽)28 2⍴(∪⍳⊢),(↑ω)[;1 2]
    adj←weights@(↓edges)⊢0⍴⍨2⍴8
    (⌊/,⌈/)+/adj[↓{∊ω}⌺1 2⊢pmat 8]
}
```



Looking at this APL solution for the Traveling Salesman Problem variant:

Data Preprocessing

```apl
data←' '(≠⊆⊢)¨'( to | = )'⎕R' '⊃⎕NGET'data9.txt'1
```

This transforms each line from `'London to Dublin = 464'` into `'London' 'Dublin' '464'` by replacing " to " and " = " with spaces, then splitting on spaces.

The Core Algorithm

1. Extract weights

```apl
weights←w,w←⍎¨⊢/↑⍵
```

Takes the rightmost column (distances), converts strings to numbers, and doubles them (since edges are bidirectional).

2. Build edge list with city indices

```apl
edges←(⊢⍪⌽)28 2⍴(∪⍳⊢),(↑⍵)[;1 2]
```

- `(↑⍵)[;1 2]` extracts the city name pairs
- `,` ravels them into a single vector
- `(∪⍳⊢)` maps each city name to its index position among unique cities
- `28 2⍴` reshapes into pairs (hardcoded for 14 undirected edges in the actual problem)
- `(⊢⍪⌽)` concatenates with reversed pairs to create bidirectional edges

3. Create adjacency matrix

```apl
adj←weights@(↓edges)⊢0⍴⍨2⍴8
```

Creates an 8×8 zero matrix (for 8 cities) and places weights at positions specified by edge pairs.

4. Calculate path lengths for all permutations

```apl
(⌊/,⌈/)+/adj[↓{∊⍵}⌺1 2⊢pmat 8]
```

- `pmat 8` generates all permutations of cities (1 through 8)
- `{∊⍵}⌺1 2` applies a 1×2 sliding window to get consecutive pairs in each permutation
- `adj[↓...]` looks up the distance for each pair in the adjacency matrix
- `+/` sums distances for each complete path
- `(⌊/,⌈/)` returns both minimum and maximum path lengths

The hardcoded values (28 for edges, 8 for cities) are specific to the actual problem input, not the test data. The algorithm finds all possible Hamiltonian paths through the graph and returns both the shortest (251) and longest (898) distances.

AI Tooling: towards greater autonomy

- Original Co-pilot: contextual auto-complete
- Copy & paste ~~Stack Overflow~~ ChatGPT
- AI-integrated IDEs (Cursor, Zed, Windsurf...)
- Console/cloud agents (Claude Code, Aider, Codex...)

AI in cli-environment

AI Tooling: Dyalog's Challenge

- Tight integration into Dyalog Integrated Development Environment (IDE)?
- Uncomfortably fast-moving goal posts
- *Not* in the 'move fast & break things' camp!
- Models still below-par for APL

Console Agents + Link = ❤️

- Good match for Link-driven workflows
- Already watching for changes = fast feedback loop
- No UI overlays or chat panels in the Dyalog environment

Console Agents

- LLM in a loop + tools use in the CLI
- Runs your whole tool chain: git, test runner, CI
- High autonomy: set it off, go for lunch
- ...come back to PR ready for review

Console Agents

- LLM in a loop + tools use in the CLI
- Runs your whole tool chain: git, test runner, CI
- High autonomy: se
- ...come back to PR ready for review

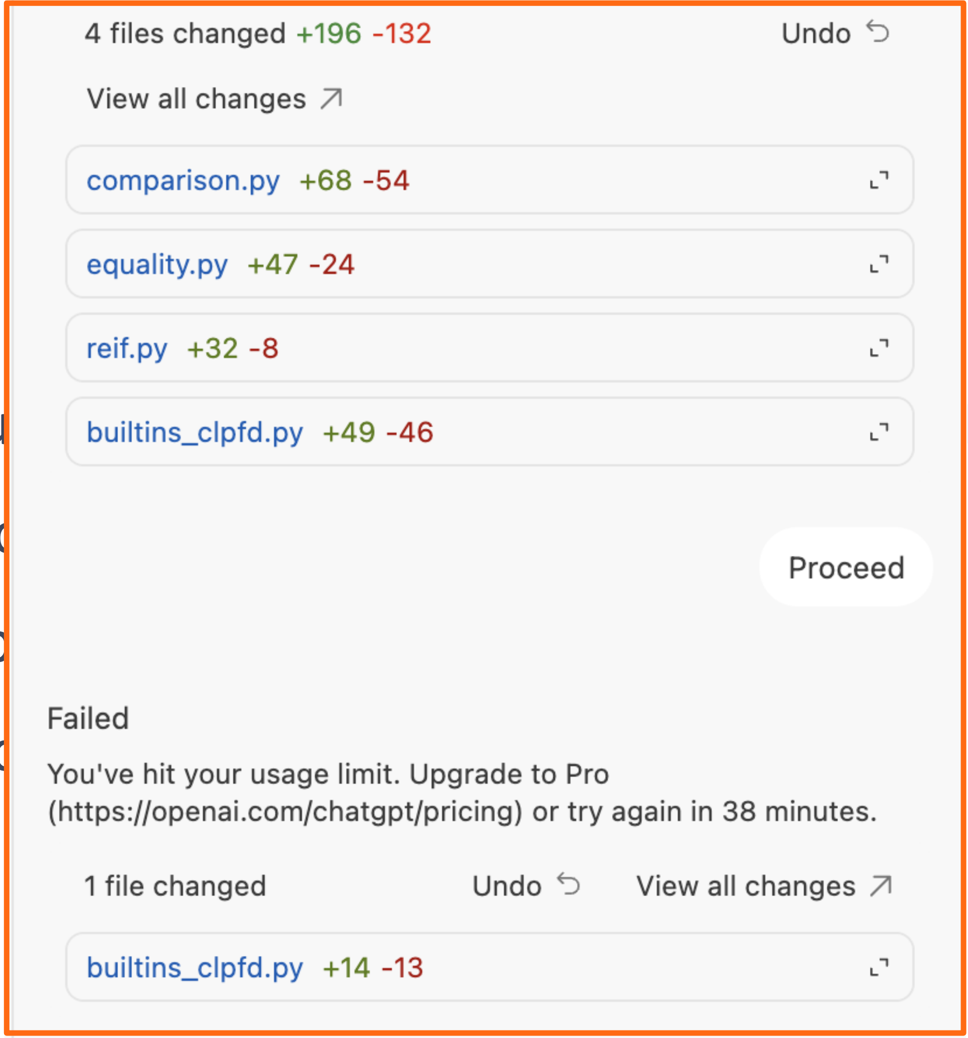
Suggested changes to a project's code

Console Agents

- LLM in a loop + tools use in the CLI
- Runs your whole tool chain: git, test runner, CI
- High autonomy: set it off, go for lunch
- ...come back to PR ready for review
- Visceral cost-~~benefit~~ feedback

Console

- LLM in a
- Runs you
- High auto
- ...come b
- Visceral c



4 files changed +196 -132

View all changes ↗

comparison.py +68 -54

equality.py +47 -24

reif.py +32 -8

builtins_clpfd.py +49 -46

Failed

You've hit your usage limit. Upgrade to Pro (https://openai.com/chatgpt/pricing) or try again in 38 minutes.

1 file changed

builtins_clpfd.py +14 -13

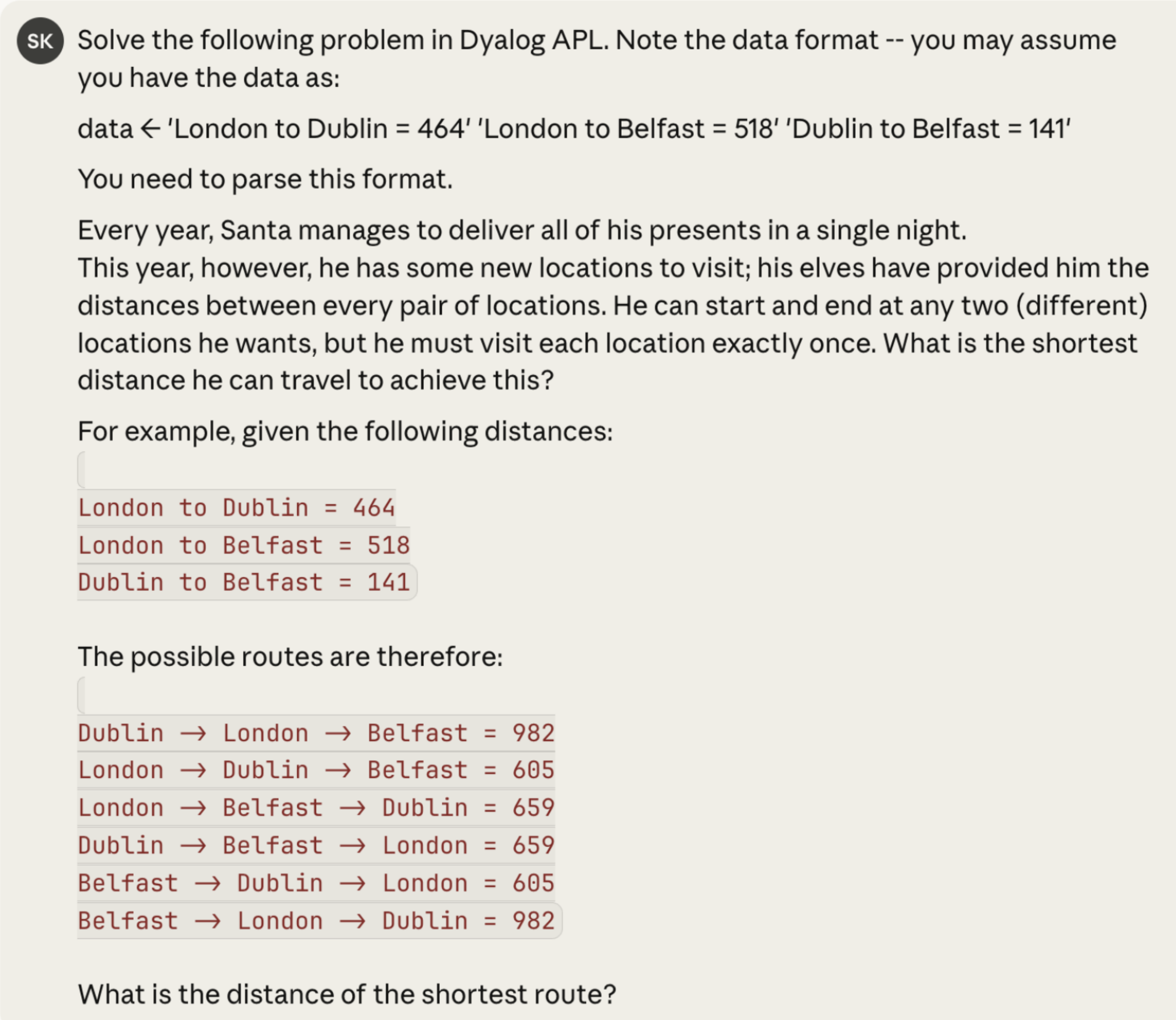
Console Agents: Implications

- Storing your code as text files
- Ability to run APL from the command line
- Everything accessible and introspectable via CLI
- Rich error reporting

Claude

Dyalog + Claude Code Agent

AI in cli-environment



SK

Solve the following problem in Dyalog APL. Note the data format -- you may assume you have the data as:

data ← 'London to Dublin = 464' 'London to Belfast = 518' 'Dublin to Belfast = 141'

You need to parse this format.

Every year, Santa manages to deliver all of his presents in a single night.
This year, however, he has some new locations to visit; his elves have provided him the distances between every pair of locations. He can start and end at any two (different) locations he wants, but he must visit each location exactly once. What is the shortest distance he can travel to achieve this?

For example, given the following distances:

```
London to Dublin = 464
London to Belfast = 518
Dublin to Belfast = 141
```

The possible routes are therefore:

```
Dublin → London → Belfast = 982
London → Dublin → Belfast = 605
London → Belfast → Dublin = 659
Dublin → Belfast → London = 659
Belfast → Dublin → London = 605
Belfast → London → Dublin = 982
```

What is the distance of the shortest route?

Claude

```
parse ← {
  parts ← (⍵='=')⊆⍵
  route ← ⊃parts
  dist ← ⍎(⊃⌽parts)
  cities ← ((route≠' ')∧~(¯2⌽0 0 1 1 0)⌽⍨≢route)⊆route
  (⊃cities)(⊃⌽cities)dist
}

edges ← parse¨data
cities ← ∪⊃,/(2↑)¨edges
n ← ≢cities
dist ← (n,n)⍴0

{
  (c1 c2 d) ← ⍵
  i j ← cities⍳c1 c2
  dist[i;j] ← dist[j;i] ← d
}¨edges
```

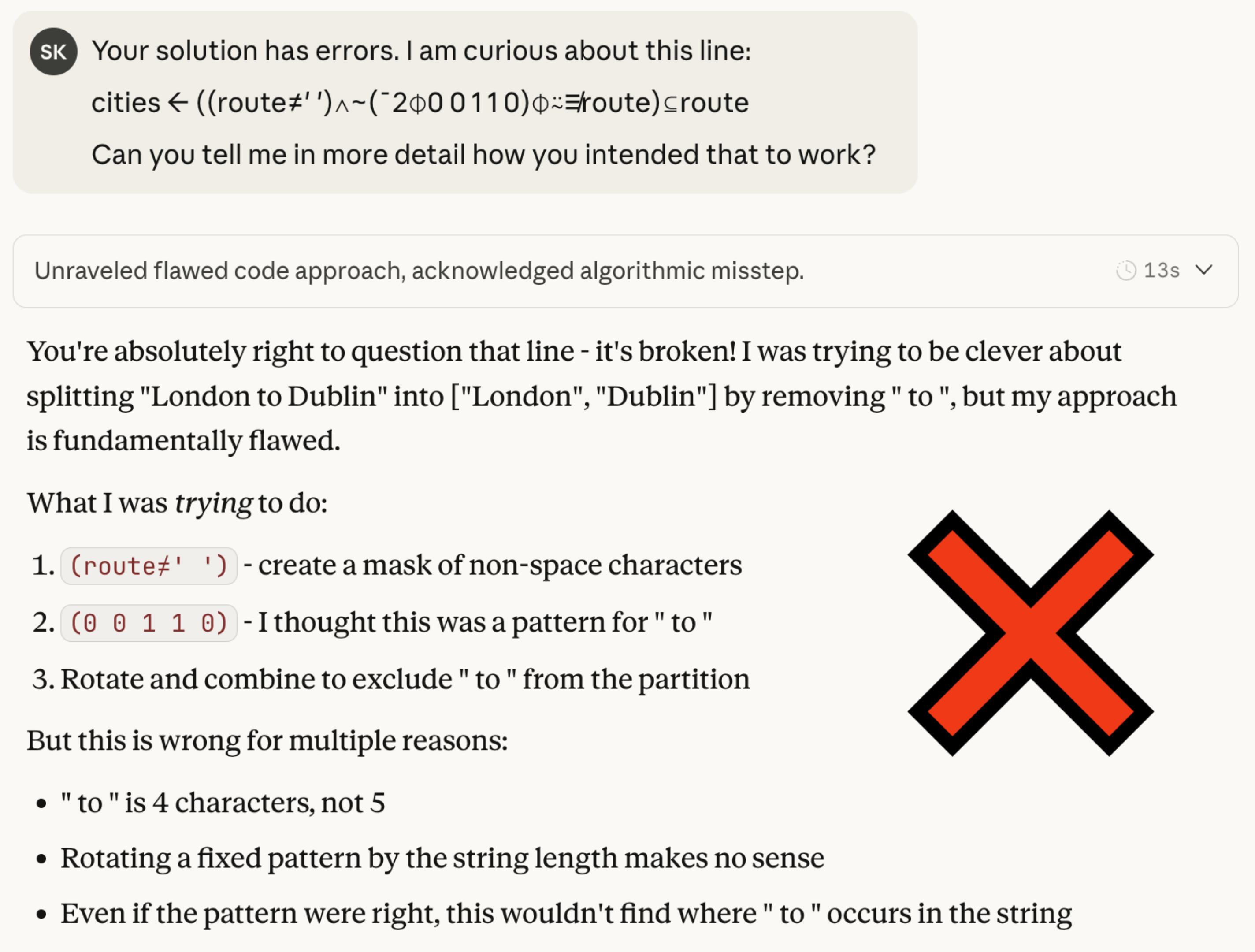
Claude

```
perm ← {
  ⍵=0: ,⊂⍬
  ⊃,/{⍵,¨∇(⍳⍺)~⍵}/⍳⍵
}

routes ← perm n
routeDist ← {
  ⍵≡⍬: 0
  pairs ← 2,/⍵
  +/dist[⊃¨pairs;⊃¨⌽¨pairs]
}
distances ← routeDist¨routes
⌊/distances
```

```
parse ← {
  parts ← (⍵='=')⊆⍵
  route ← ⊃parts
  dist ← ⍎(⊃⌽parts)
  cities ← ((route≠' ')∧~(¯2⌽0 0 1 1 0)⌽⍨≢route)⊆route
  (⊃cities)(⊃⌽cities)dist
}

e
c
n
d

{
  (c1 c2 d) ← ⍵
  i j ← cities⍳c1 c2
  dist[i;j] ← dist[j;i] ← d
}¨edges
```

```
perm ← {
  ⍵=0: ,⊂⍬
  ⊃,/{⍵,¨∇(⍳⍺)~⍵}/⍳⍵
}

routes ← perm n
routeDist ← {
  ⍵=⍬: 0
```

```
LENGTH ERROR
parse[4] cities←((route≠' ')∧~(¯2⌽0 0 1 1 0)⌽⍨≢route)⊆route
                                                     ∧
```



Your solution has errors. I am curious about this line:

```
cities ← ((route≠' ')∧~(¯2⌽0 0 1 1 0)⌽⍨≢route)⊆route
```

Can you tell me in more detail how you intended that to work?

Unraveled flawed code approach, acknowledged algorithmic misstep.

You're absolutely right to question that line - it's broken! I was trying to be clever about splitting "London to Dublin" into ["London", "Dublin"] by removing " to ", but my approach is fundamentally flawed.

What I was *trying* to do:

1. `(route≠' ')` - create a mask of non-space characters
2. `(0 0 1 1 0)` - I thought this was a pattern for " to "
3. Rotate and combine to exclude " to " from the partition

But this is wrong for multiple reasons:

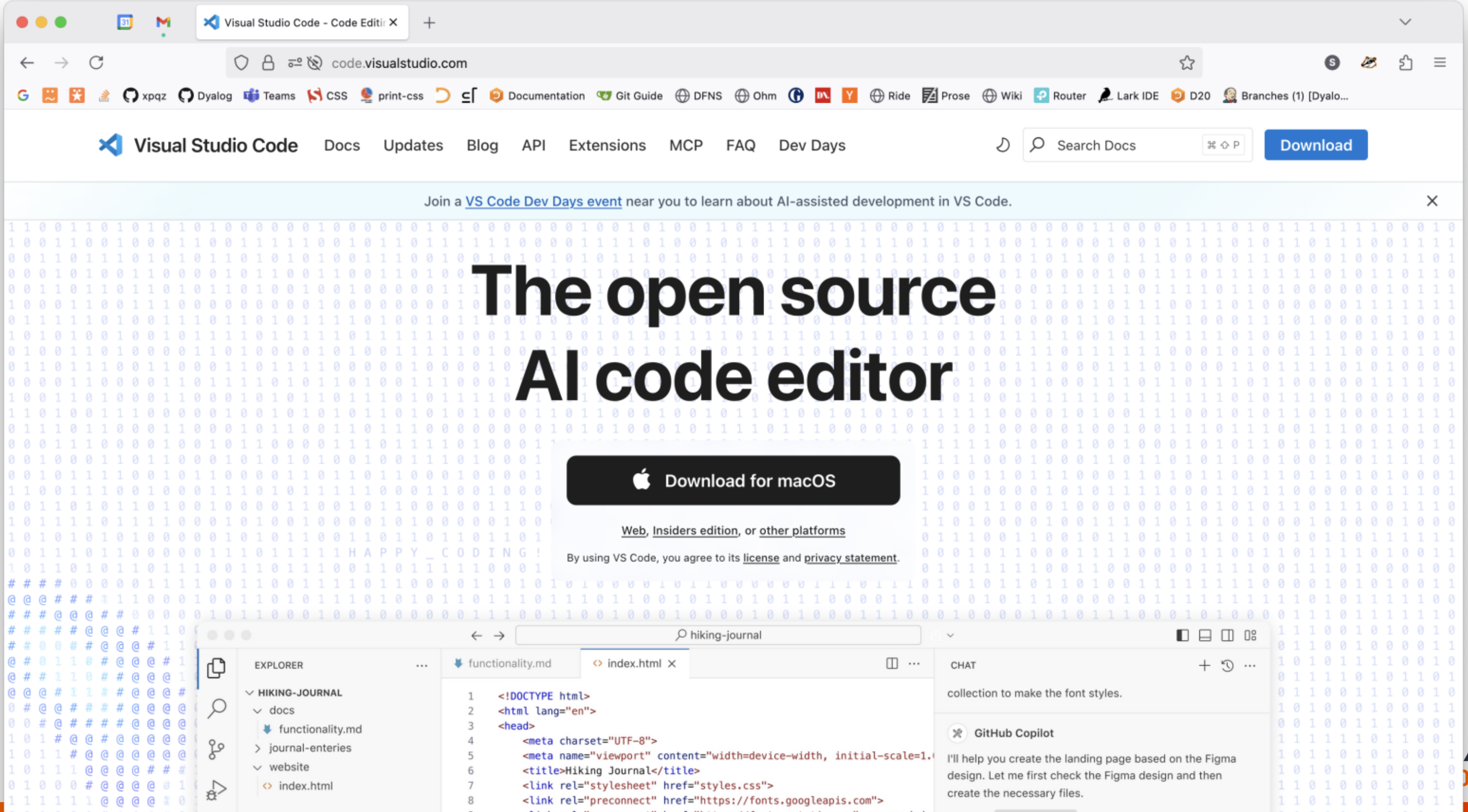
- " to " is 4 characters, not 5
- Rotating a fixed pattern by the string length makes no sense
- Even if the pattern were right, this wouldn't find where " to " occurs in the string

Claude Code: Stacking the Deck

Claude

- Carefully tuned instructions (CLAUDE.md)
- Dyalog evaluation
- APLCart lookup
- Dyalog documentation
- Console-only unit testing

Visual Studio Code (VS Code)



Join a VS Code Dev Days event near you to learn about AI-assisted development in VS Code.

The open source AI code editor

Web, Insiders edition, or other platforms

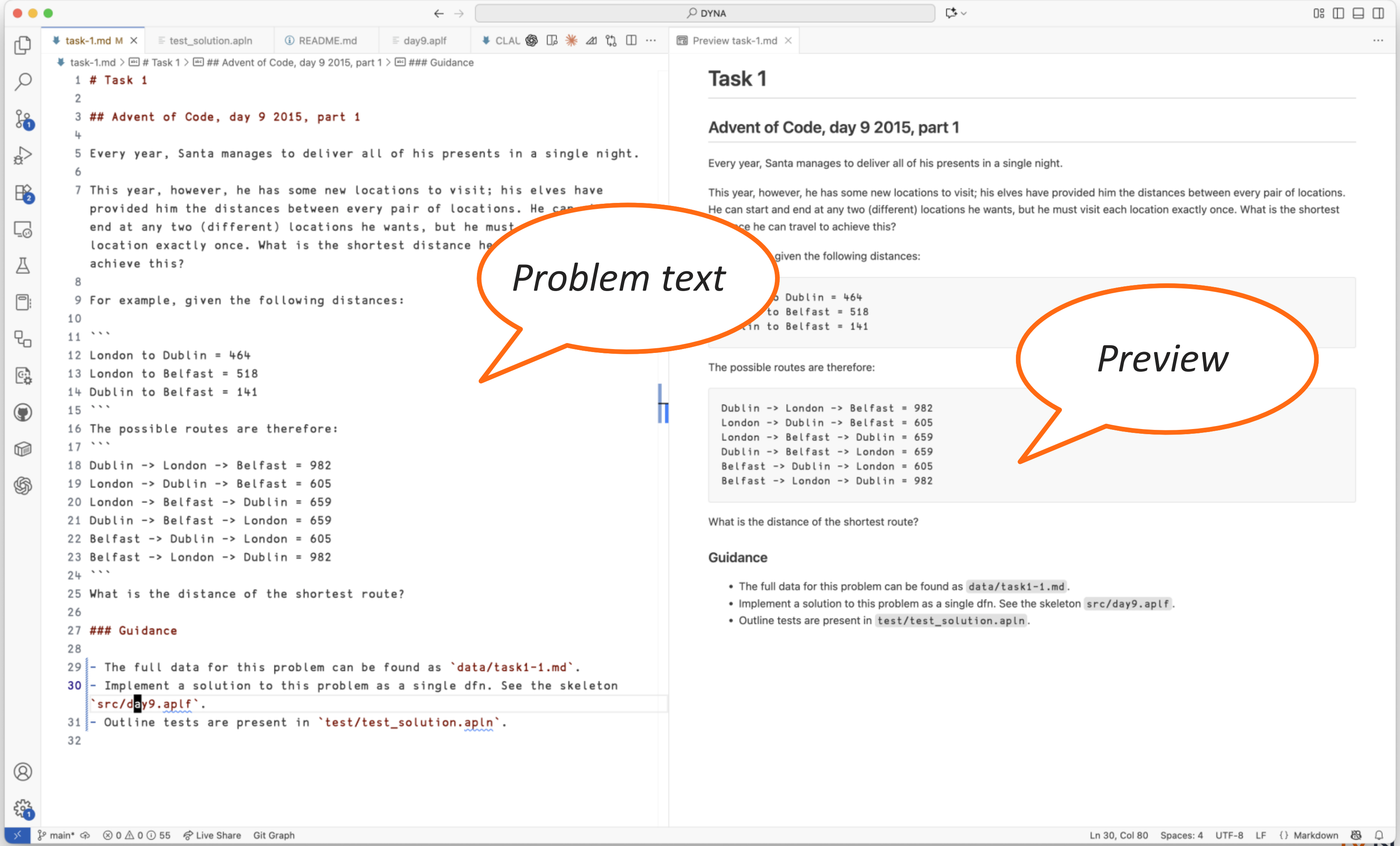
By using VS Code, you agree to its license and privacy statement.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
    <title>Hiking Journal</title>
    <link rel="stylesheet" href="styles.css">
    <link rel="preconnect" href="https://fonts.googleapis.com">
```

collection to make the font styles.

GitHub Copilot

I'll help you create the landing page based on the Figma design. Let me first check the Figma design and then create the necessary files.


DYNA
task-1.md M
test_solution.apln
README.md
day9.aplf
CLAL
Preview task-1.md
task-1.md > # Task 1 > ## Advent of Code, day 9 2015, part 1 > ### Guidance
Problem text
Preview
main* ⓧ 0 ⚠ 0 ⓘ 55 Live Share Git Graph
Ln 30, Col 80 Spaces: 4 UTF-8 LF {} Markdown

```
# Task 1

## Advent of Code, day 9 2015, part 1

Every year, Santa manages to deliver all of his presents in a single night.

This year, however, he has some new locations to visit; his elves have
provided him the distances between every pair of locations. He can
end at any two (different) locations he wants, but he must
location exactly once. What is the shortest distance he
achieve this?

For example, given the following distances:

```
London to Dublin = 464
London to Belfast = 518
Dublin to Belfast = 141
```
The possible routes are therefore:
```
Dublin -> London -> Belfast = 982
London -> Dublin -> Belfast = 605
London -> Belfast -> Dublin = 659
Dublin -> Belfast -> London = 659
Belfast -> Dublin -> London = 605
Belfast -> London -> Dublin = 982
```
What is the distance of the shortest route?

### Guidance

- The full data for this problem can be found as `data/task1-1.md`.
- Implement a solution to this problem as a single dfn. See the skeleton
`src/day9.aplf`.
- Outline tests are present in `test/test_solution.apln`.

```

Task 1

Advent of Code, day 9 2015, part 1

Every year, Santa manages to deliver all of his presents in a single night.

This year, however, he has some new locations to visit; his elves have provided him the distances between every pair of locations. He can start and end at any two (different) locations he wants, but he must visit each location exactly once. What is the shortest ce he can travel to achieve this?

given the following distances:

```
o Dublin = 464
to Belfast = 518
in to Belfast = 141
```

The possible routes are therefore:

```
Dublin -> London -> Belfast = 982
London -> Dublin -> Belfast = 605
London -> Belfast -> Dublin = 659
Dublin -> Belfast -> London = 659
Belfast -> Dublin -> London = 605
Belfast -> London -> Dublin = 982
```

What is the distance of the shortest route?

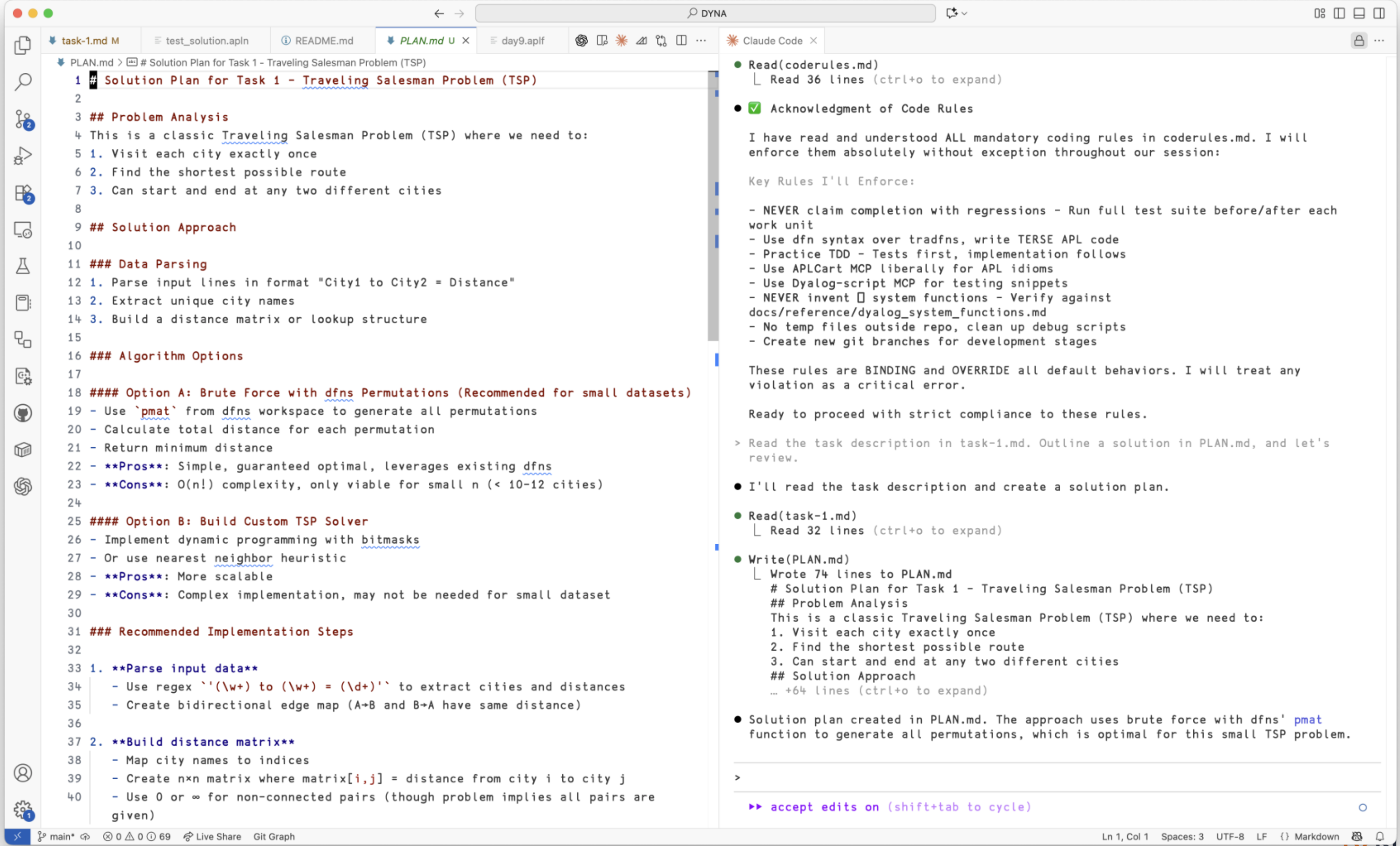
Guidance

- The full data for this problem can be found as `data/task1-1.md`.
- Implement a solution to this problem as a single dfn. See the skeleton `src/day9.aplf`.
- Outline tests are present in `test/test_solution.apln`.

```
test_given_example ← {
  data ← 'London to Dublin = 464' 'London to Belfast = 518' 'Dublin to Belfast = 141'
  expected ← 605

  result ← #.Day9 data
  'Wrong result'#.dytest.Assert result = expected:
  1
}

test_solution ← {
  data ← ⊃⎕NGET'data/task1-1.txt'1
  expected ← 251
  result ← #.Day9 data
  'Wrong result'#.dytest.Assert result = expected:
  1
}
```

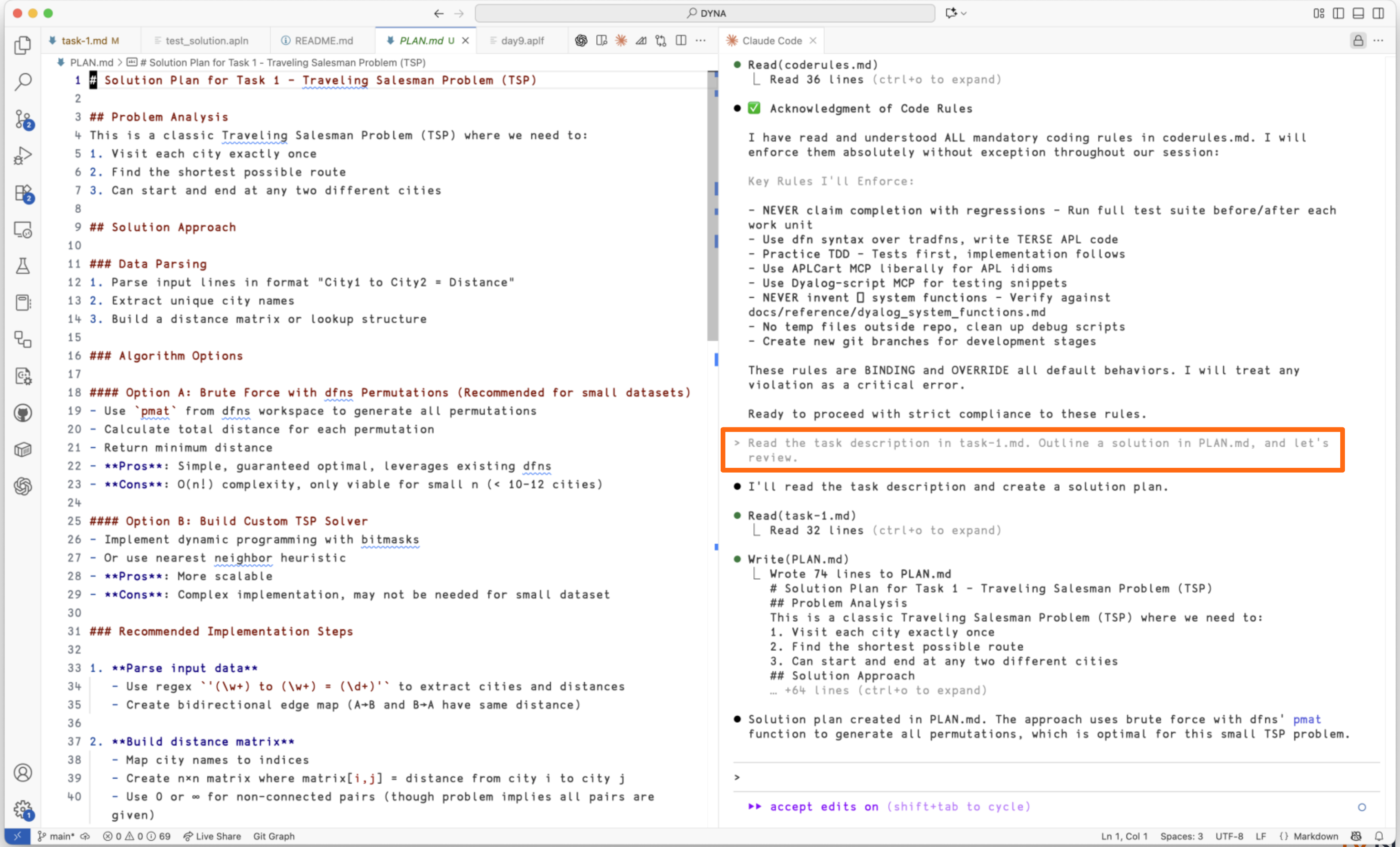
Two prompts only

- Read the task description in task-1.md. Outline a solution in PLAN.md, and let's review.
- Proceed with implementing your plan. Ensure to implement tests for intermediate steps, or ad-hoc testing with the 'dyalogscript' tool.
- [goes for lunch]



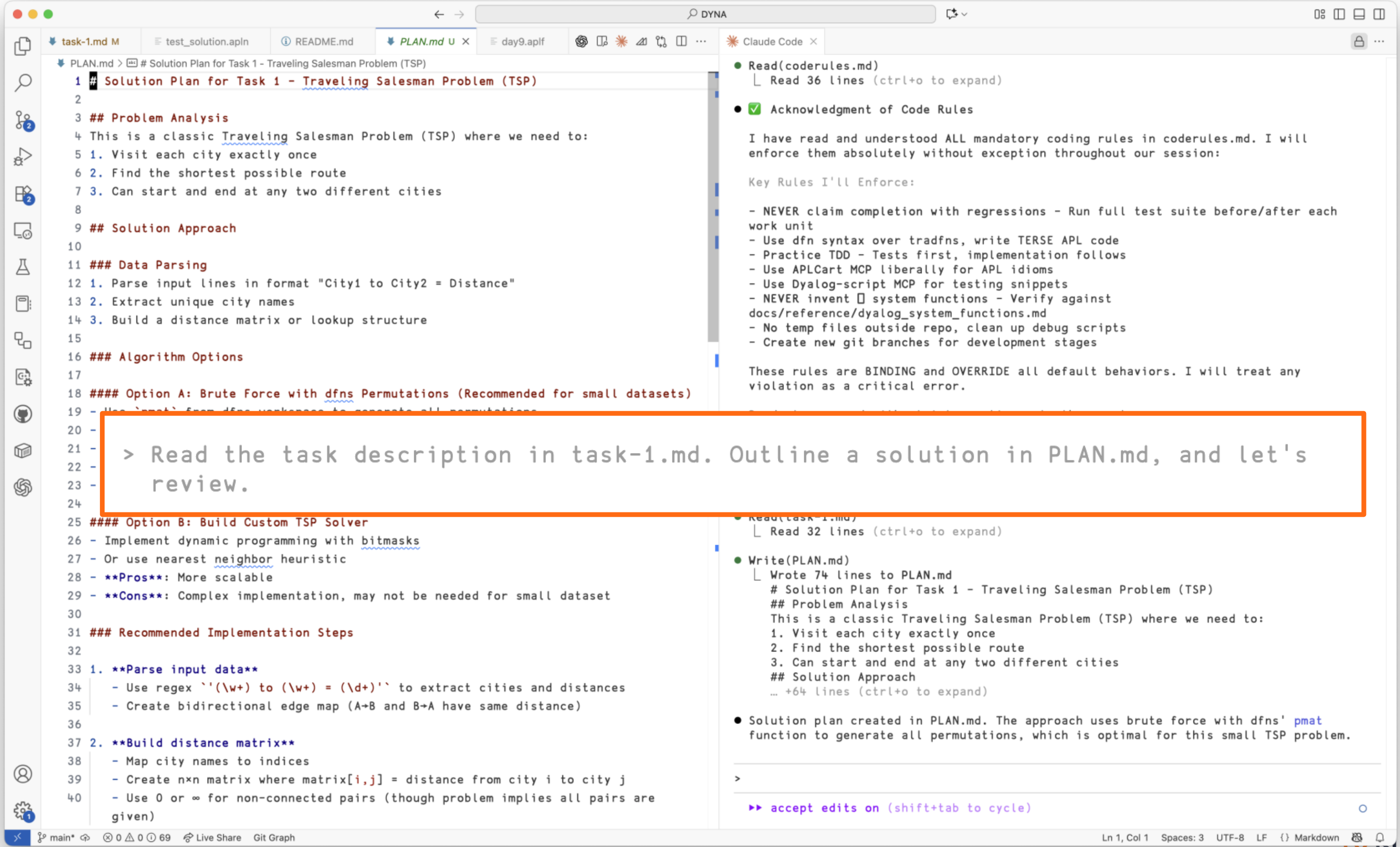
```markdown
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
   given)
```

```
● Read(coderules.md)
  ⎿ Read 36 lines (ctrl+o to expand)

● ✅ Acknowledgment of Code Rules

  I have read and understood ALL mandatory coding rules in coderules.md. I will
  enforce them absolutely without exception throughout our session:

  Key Rules I'll Enforce:

  - NEVER claim completion with regressions - Run full test suite before/after each
  work unit
  - Use dfn syntax over tradfns, write TERSE APL code
  - Practice TDD - Tests first, implementation follows
  - Use APLCart MCP liberally for APL idioms
  - Use Dyalog-script MCP for testing snippets
  - NEVER invent ⎕ system functions - Verify against
  docs/reference/dyalog_system_functions.md
  - No temp files outside repo, clean up debug scripts
  - Create new git branches for development stages

  These rules are BINDING and OVERRIDE all default behaviors. I will treat any
  violation as a critical error.

  Ready to proceed with strict compliance to these rules.

> Read the task description in task-1.md. Outline a solution in PLAN.md, and let's
  review.

● I'll read the task description and create a solution plan.

● Read(task-1.md)
  ⎿ Read 32 lines (ctrl+o to expand)

● Write(PLAN.md)
  ⎿ Wrote 74 lines to PLAN.md
     # Solution Plan for Task 1 - Traveling Salesman Problem (TSP)
     ## Problem Analysis
     This is a classic Traveling Salesman Problem (TSP) where we need to:
     1. Visit each city exactly once
     2. Find the shortest possible route
     3. Can start and end at any two different cities
     ## Solution Approach
     … +64 lines (ctrl+o to expand)

● Solution plan created in PLAN.md. The approach uses brute force with dfns' pmat
  function to generate all permutations, which is optimal for this small TSP problem.

>

  ▸▸ accept edits on (shift+tab to cycle)
```



```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are given)
```

```
● Read(coderules.md)
  ⎿ Read 36 lines (ctrl+o to expand)

● ✅ Acknowledgment of Code Rules

  I have read and understood ALL mandatory coding rules in coderules.md. I will
  enforce them absolutely without exception throughout our session:

  Key Rules I'll Enforce:

  - NEVER claim completion with regressions - Run full test suite before/after each
  work unit
  - Use dfn syntax over tradfns, write TERSE APL code
  - Practice TDD - Tests first, implementation follows
  - Use APLCart MCP liberally for APL idioms
  - Use Dyalog-script MCP for testing snippets
  - NEVER invent ⎕ system functions - Verify against
  docs/reference/dyalog_system_functions.md
  - No temp files outside repo, clean up debug scripts
  - Create new git branches for development stages

  These rules are BINDING and OVERRIDE all default behaviors. I will treat any
  violation as a critical error.

  Ready to proceed with strict compliance to these rules.

> Read the task description in task-1.md. Outline a solution in PLAN.md, and let's
  review.

● I'll read the task description and create a solution plan.

● Read(task-1.md)
  ⎿ Read 32 lines (ctrl+o to expand)

● Write(PLAN.md)
  ⎿ Wrote 74 lines to PLAN.md
    # Solution Plan for Task 1 - Traveling Salesman Problem (TSP)
    ## Problem Analysis
    This is a classic Traveling Salesman Problem (TSP) where we need to:
    1. Visit each city exactly once
    2. Find the shortest possible route
    3. Can start and end at any two different cities
    ## Solution Approach
    … +64 lines (ctrl+o to expand)

● Solution plan created in PLAN.md. The approach uses brute force with dfns' pmat
  function to generate all permutations, which is optimal for this small TSP problem.

>

  ▸▸ accept edits on (shift+tab to cycle)
```

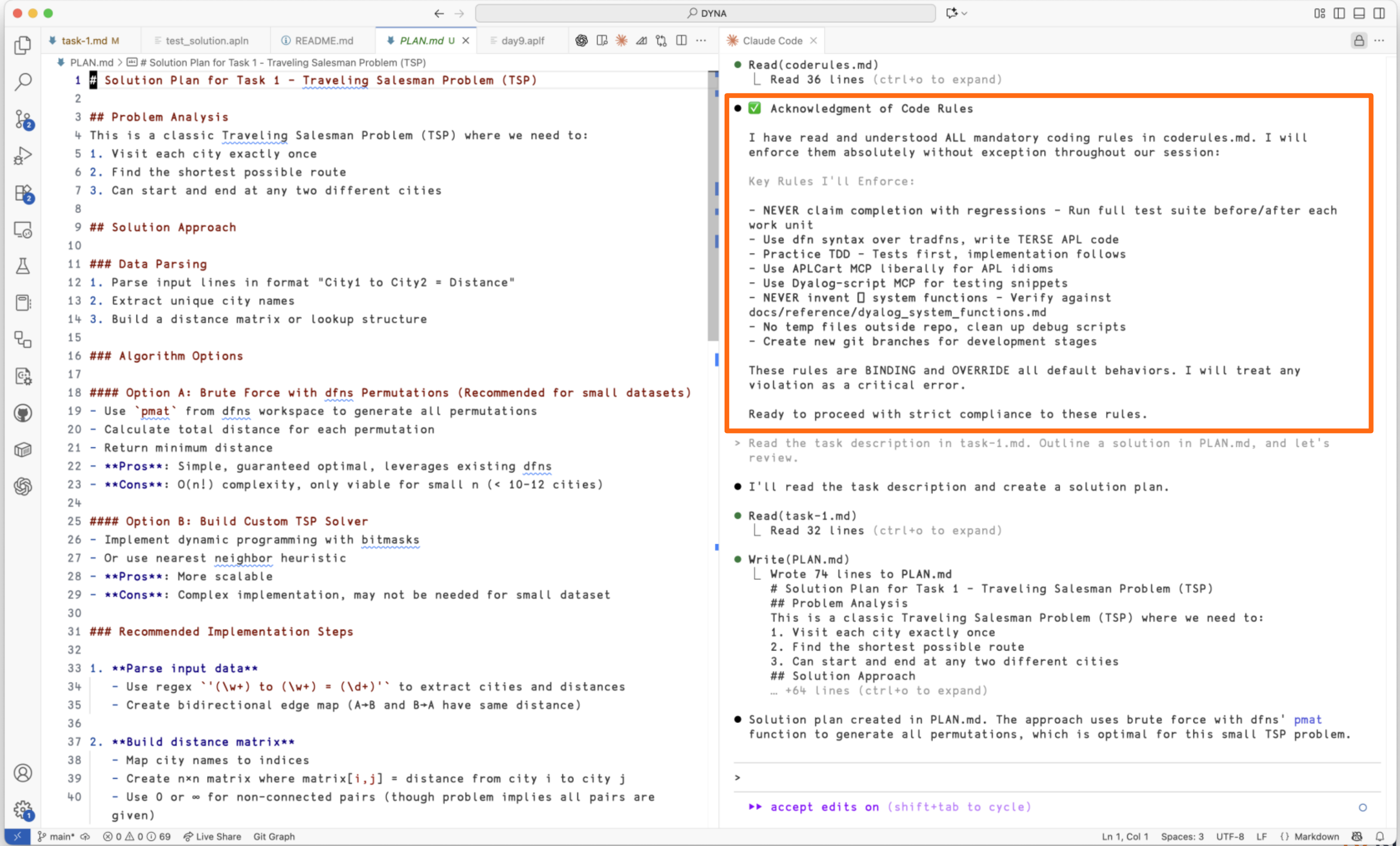


task-1.md M
test_solution.apln
README.md
PLAN.md U
day9.aplf
DYNA
Claude Code
PLAN.md > # Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
-
-
-
-

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
   given)
```

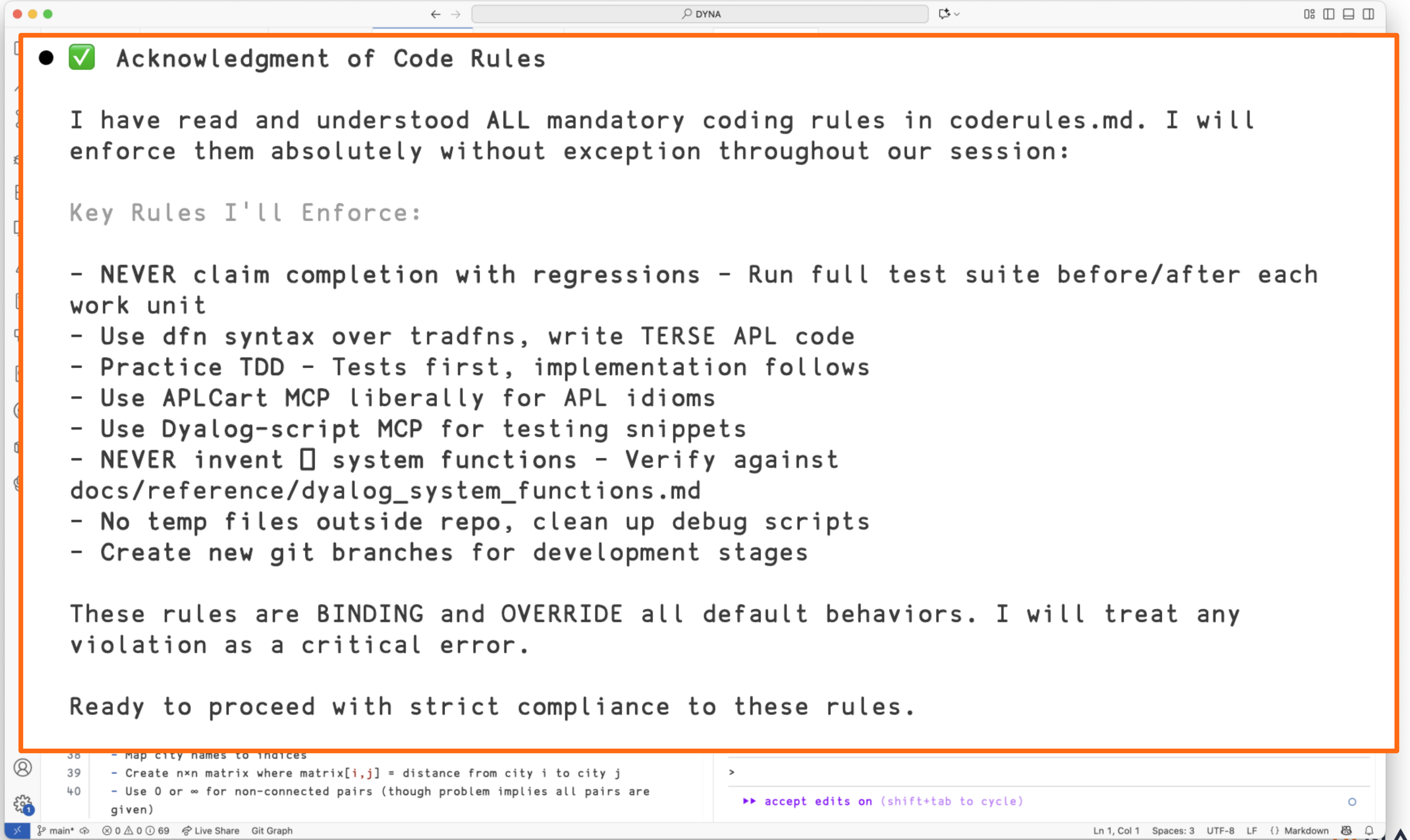
```
● Read(coderules.md)
  ⎿ Read 36 lines (ctrl+o to expand)

● ✅ Acknowledgment of Code Rules

  I have read and understood ALL mandatory coding rules in coderules.md. I will
  enforce them absolutely without exception throughout our session:

  Key Rules I'll Enforce:

  - NEVER claim completion with regressions - Run full test suite before/after each
  work unit
  - Use dfn syntax over tradfns, write TERSE APL code
  - Practice TDD - Tests first, implementation follows
  - Use APLCart MCP liberally for APL idioms
  - Use Dyalog-script MCP for testing snippets
  - NEVER invent ⎕ system functions - Verify against
  docs/reference/dyalog_system_functions.md
  - No temp files outside repo, clean up debug scripts
  - Create new git branches for development stages

  These rules are BINDING and OVERRIDE all default behaviors. I will treat any
  violation as a critical error.

● Read(task-1.md)
  ⎿ Read 32 lines (ctrl+o to expand)

● Write(PLAN.md)
  ⎿ Wrote 74 lines to PLAN.md
     # Solution Plan for Task 1 - Traveling Salesman Problem (TSP)
     ## Problem Analysis
     This is a classic Traveling Salesman Problem (TSP) where we need to:
     1. Visit each city exactly once
     2. Find the shortest possible route
     3. Can start and end at any two different cities
     ## Solution Approach
     … +64 lines (ctrl+o to expand)

● Solution plan created in PLAN.md. The approach uses brute force with dfns' pmat
  function to generate all permutations, which is optimal for this small TSP problem.

>

  ▸▸ accept edits on (shift+tab to cycle)
```

```
> Read the task description in task-1.md. Outline a solution in PLAN.md, and let's
  review.
```



```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are given)
```

```
● Read(coderules.md)
  ⎿ Read 36 lines (ctrl+o to expand)

● ✅ Acknowledgment of Code Rules

  I have read and understood ALL mandatory coding rules in coderules.md. I will
  enforce them absolutely without exception throughout our session:

  Key Rules I'll Enforce:

  - NEVER claim completion with regressions - Run full test suite before/after each
  work unit
  - Use dfn syntax over tradfns, write TERSE APL code
  - Practice TDD - Tests first, implementation follows
  - Use APLCart MCP liberally for APL idioms
  - Use Dyalog-script MCP for testing snippets
  - NEVER invent ⎕ system functions - Verify against
  docs/reference/dyalog_system_functions.md
  - No temp files outside repo, clean up debug scripts
  - Create new git branches for development stages

  These rules are BINDING and OVERRIDE all default behaviors. I will treat any
  violation as a critical error.

  Ready to proceed with strict compliance to these rules.

> Read the task description in task-1.md. Outline a solution in PLAN.md, and let's
  review.

● I'll read the task description and create a solution plan.

● Read(task-1.md)
  ⎿ Read 32 lines (ctrl+o to expand)

● Write(PLAN.md)
  ⎿ Wrote 74 lines to PLAN.md
    # Solution Plan for Task 1 - Traveling Salesman Problem (TSP)
    ## Problem Analysis
    This is a classic Traveling Salesman Problem (TSP) where we need to:
    1. Visit each city exactly once
    2. Find the shortest possible route
    3. Can start and end at any two different cities
    ## Solution Approach
    … +64 lines (ctrl+o to expand)

● Solution plan created in PLAN.md. The approach uses brute force with dfns' pmat
  function to generate all permutations, which is optimal for this small TSP problem.

>

▸▸ accept edits on (shift+tab to cycle)
```



● ✅ Acknowledgment of Code Rules

I have read and understood ALL mandatory coding rules in coderules.md. I will enforce them absolutely without exception throughout our session:

Key Rules I'll Enforce:

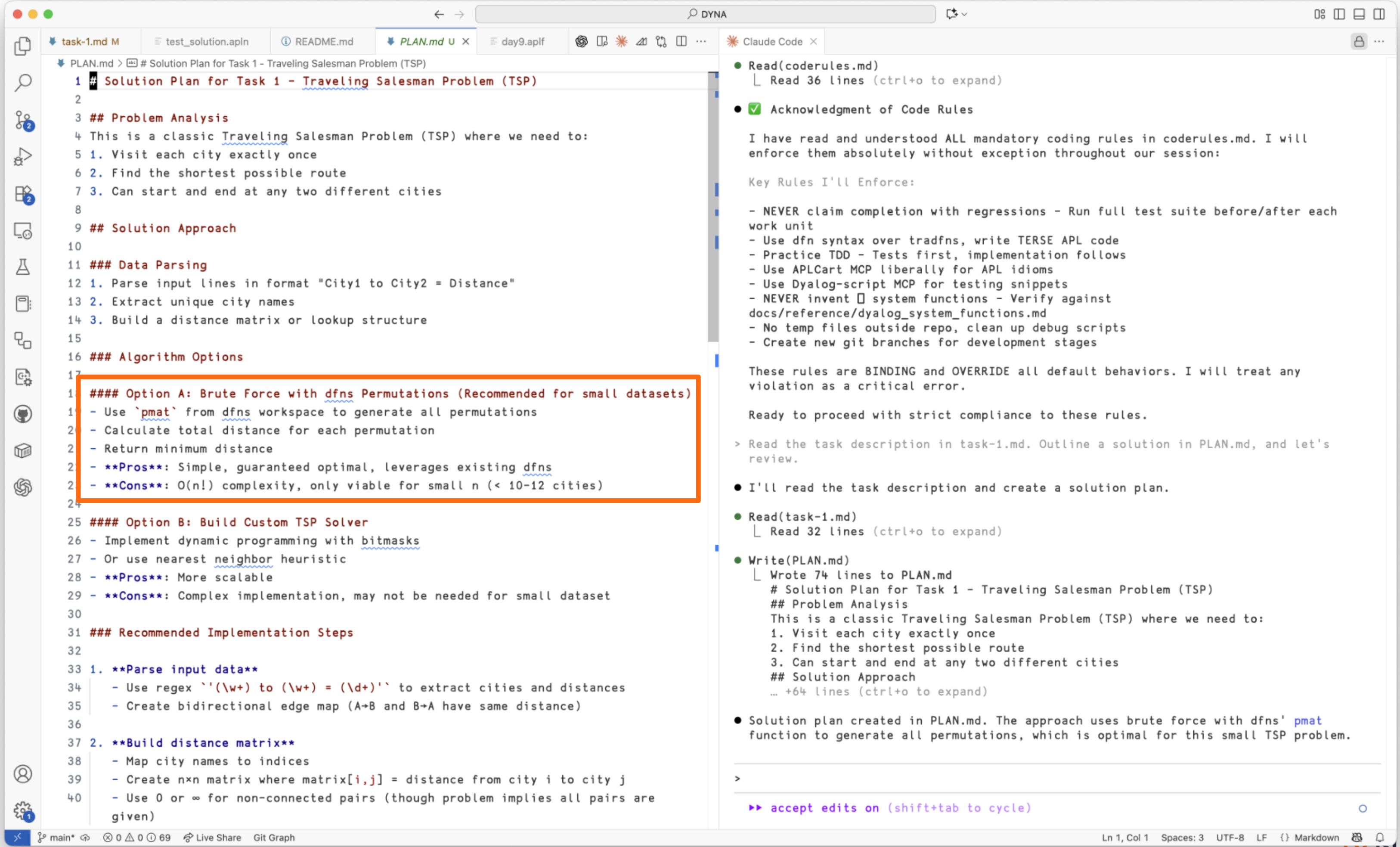
- NEVER claim completion with regressions - Run full test suite before/after each work unit
- Use dfn syntax over tradfns, write TERSE APL code
- Practice TDD - Tests first, implementation follows
- Use APLCart MCP liberally for APL idioms
- Use Dyalog-script MCP for testing snippets
- NEVER invent ⎕ system functions - Verify against docs/reference/dyalog_system_functions.md
- No temp files outside repo, clean up debug scripts
- Create new git branches for development stages

These rules are BINDING and OVERRIDE all default behaviors. I will treat any violation as a critical error.

Ready to proceed with strict compliance to these rules.

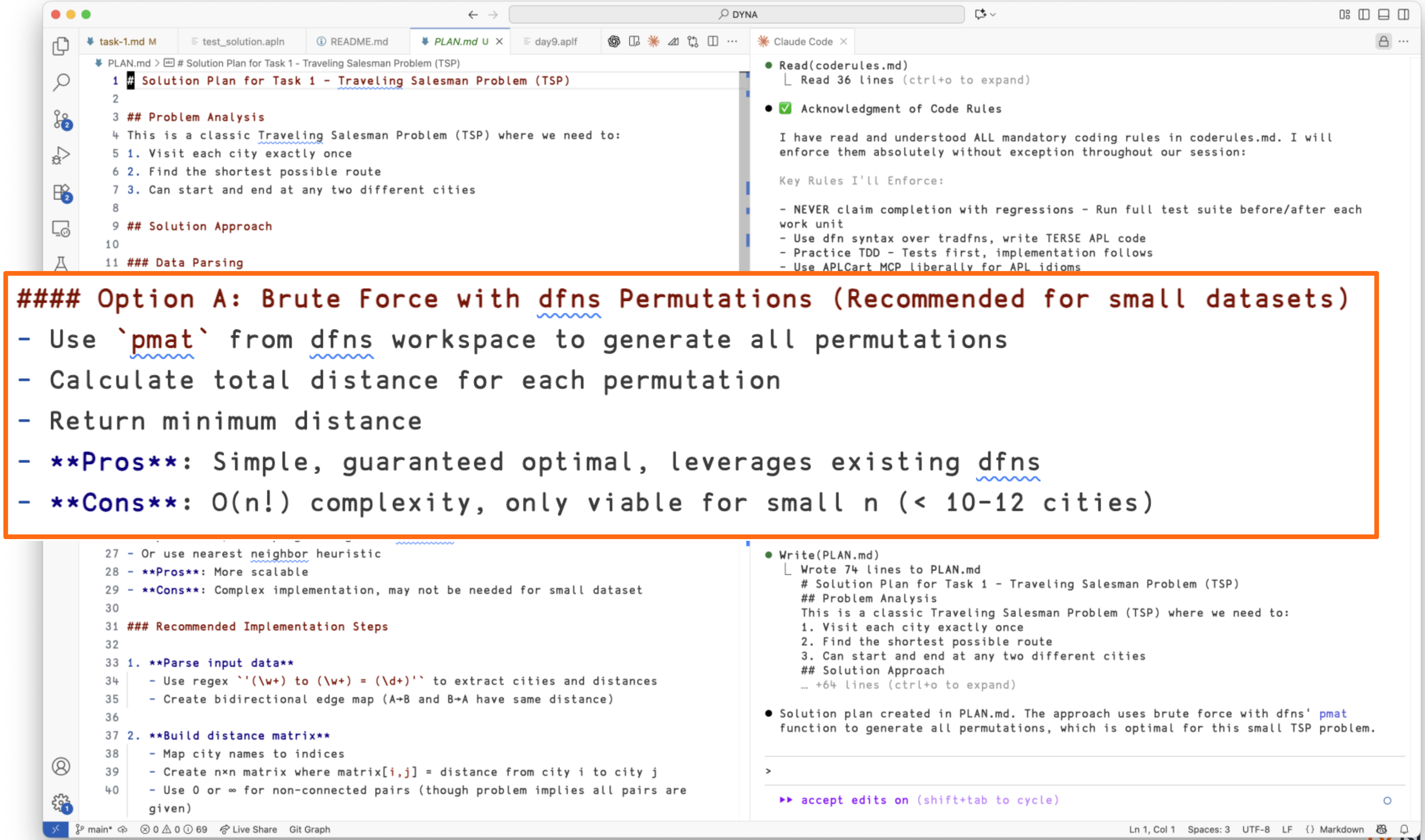
```
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
given)
```

```
>
▸▸ accept edits on (shift+tab to cycle)
```



```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
given)
```

```
● Read(coderules.md)
  ⎿ Read 36 lines (ctrl+o to expand)

● ✅ Acknowledgment of Code Rules

  I have read and understood ALL mandatory coding rules in coderules.md. I will
  enforce them absolutely without exception throughout our session:

  Key Rules I'll Enforce:

  - NEVER claim completion with regressions - Run full test suite before/after each
  work unit
  - Use dfn syntax over tradfns, write TERSE APL code
  - Practice TDD - Tests first, implementation follows
  - Use APLCart MCP liberally for APL idioms
  - Use Dyalog-script MCP for testing snippets
  - NEVER invent ⎕ system functions - Verify against
  docs/reference/dyalog_system_functions.md
  - No temp files outside repo, clean up debug scripts
  - Create new git branches for development stages

  These rules are BINDING and OVERRIDE all default behaviors. I will treat any
  violation as a critical error.

  Ready to proceed with strict compliance to these rules.

> Read the task description in task-1.md. Outline a solution in PLAN.md, and let's
  review.

● I'll read the task description and create a solution plan.

● Read(task-1.md)
  ⎿ Read 32 lines (ctrl+o to expand)

● Write(PLAN.md)
  ⎿ Wrote 74 lines to PLAN.md
     # Solution Plan for Task 1 - Traveling Salesman Problem (TSP)
     ## Problem Analysis
     This is a classic Traveling Salesman Problem (TSP) where we need to:
     1. Visit each city exactly once
     2. Find the shortest possible route
     3. Can start and end at any two different cities
     ## Solution Approach
     … +64 lines (ctrl+o to expand)

● Solution plan created in PLAN.md. The approach uses brute force with dfns' pmat
  function to generate all permutations, which is optimal for this small TSP problem.

>

  ▸▸ accept edits on (shift+tab to cycle)
```



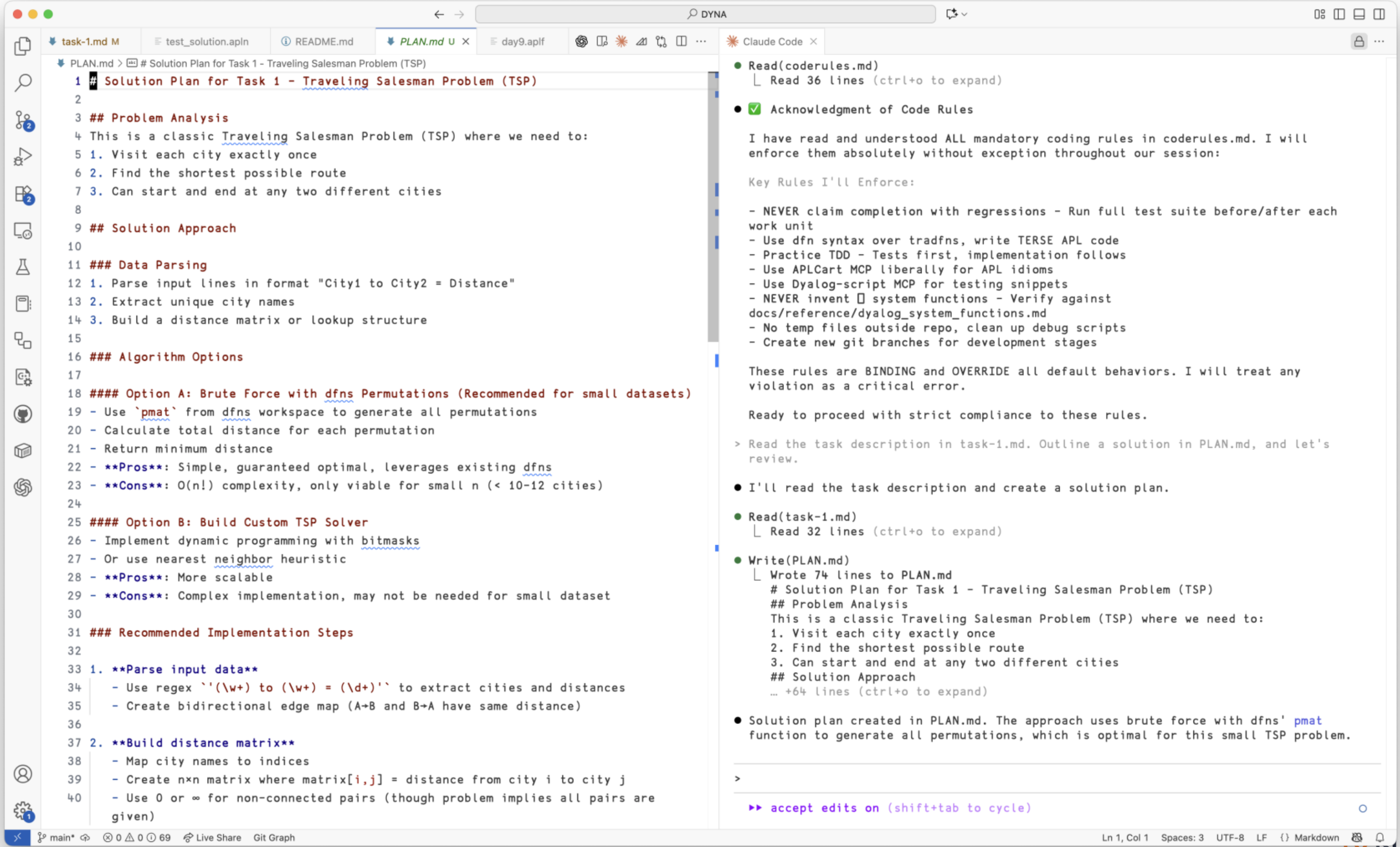
```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
```

```
#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)
```

```
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
   given)
```

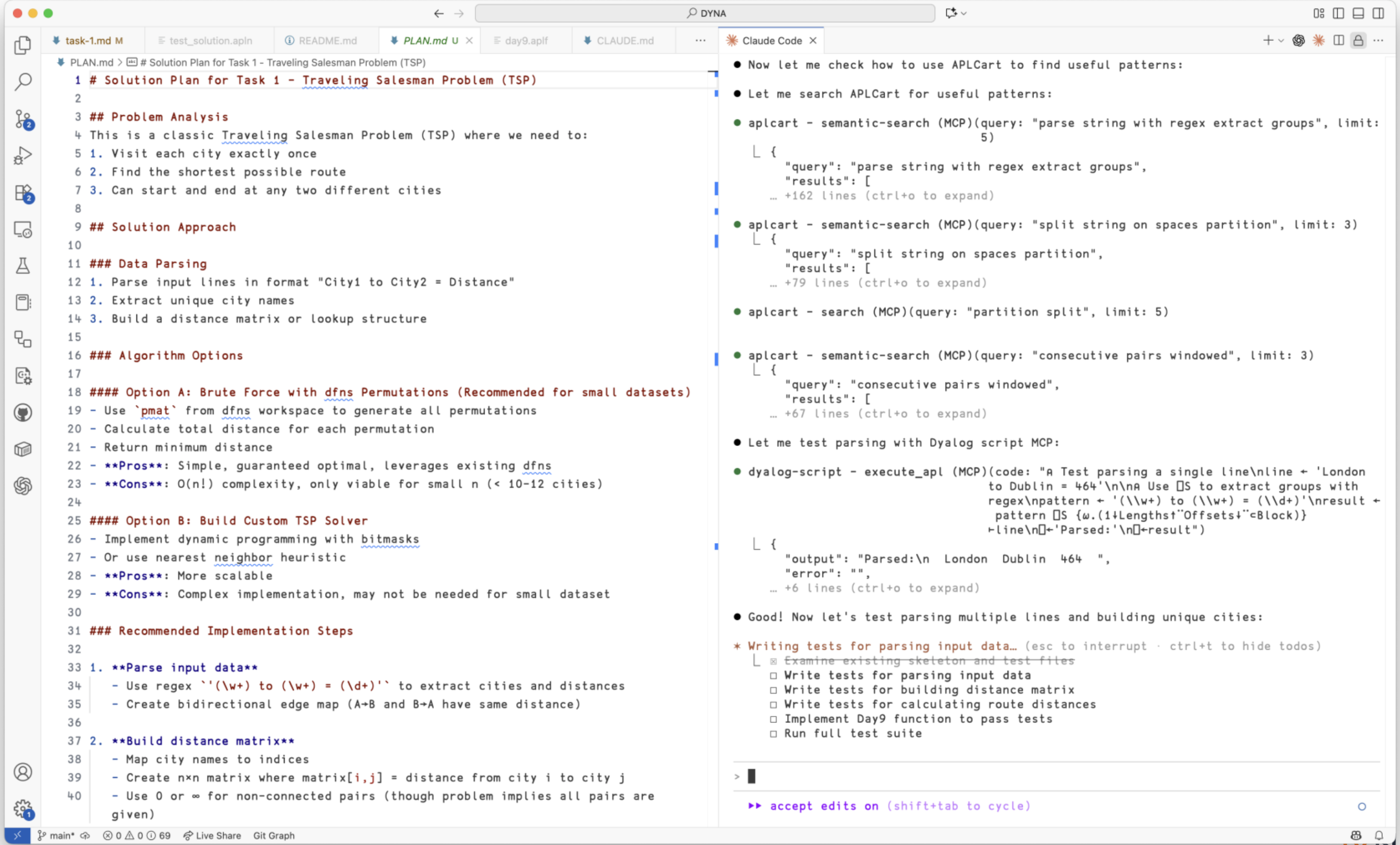
```
● Read(coderules.md)
  ⎿ Read 36 lines (ctrl+o to expand)

● ✅ Acknowledgment of Code Rules

  I have read and understood ALL mandatory coding rules in coderules.md. I will
  enforce them absolutely without exception throughout our session:

  Key Rules I'll Enforce:

  - NEVER claim completion with regressions - Run full test suite before/after each
  work unit
  - Use dfn syntax over tradfns, write TERSE APL code
  - Practice TDD - Tests first, implementation follows
  - Use APLCart MCP liberally for APL idioms
```

```
● Write(PLAN.md)
  ⎿ Wrote 74 lines to PLAN.md
     # Solution Plan for Task 1 - Traveling Salesman Problem (TSP)
     ## Problem Analysis
     This is a classic Traveling Salesman Problem (TSP) where we need to:
     1. Visit each city exactly once
     2. Find the shortest possible route
     3. Can start and end at any two different cities
     ## Solution Approach
     … +64 lines (ctrl+o to expand)

● Solution plan created in PLAN.md. The approach uses brute force with dfns' pmat
  function to generate all permutations, which is optimal for this small TSP problem.

>

  ▸▸ accept edits on (shift+tab to cycle)
```



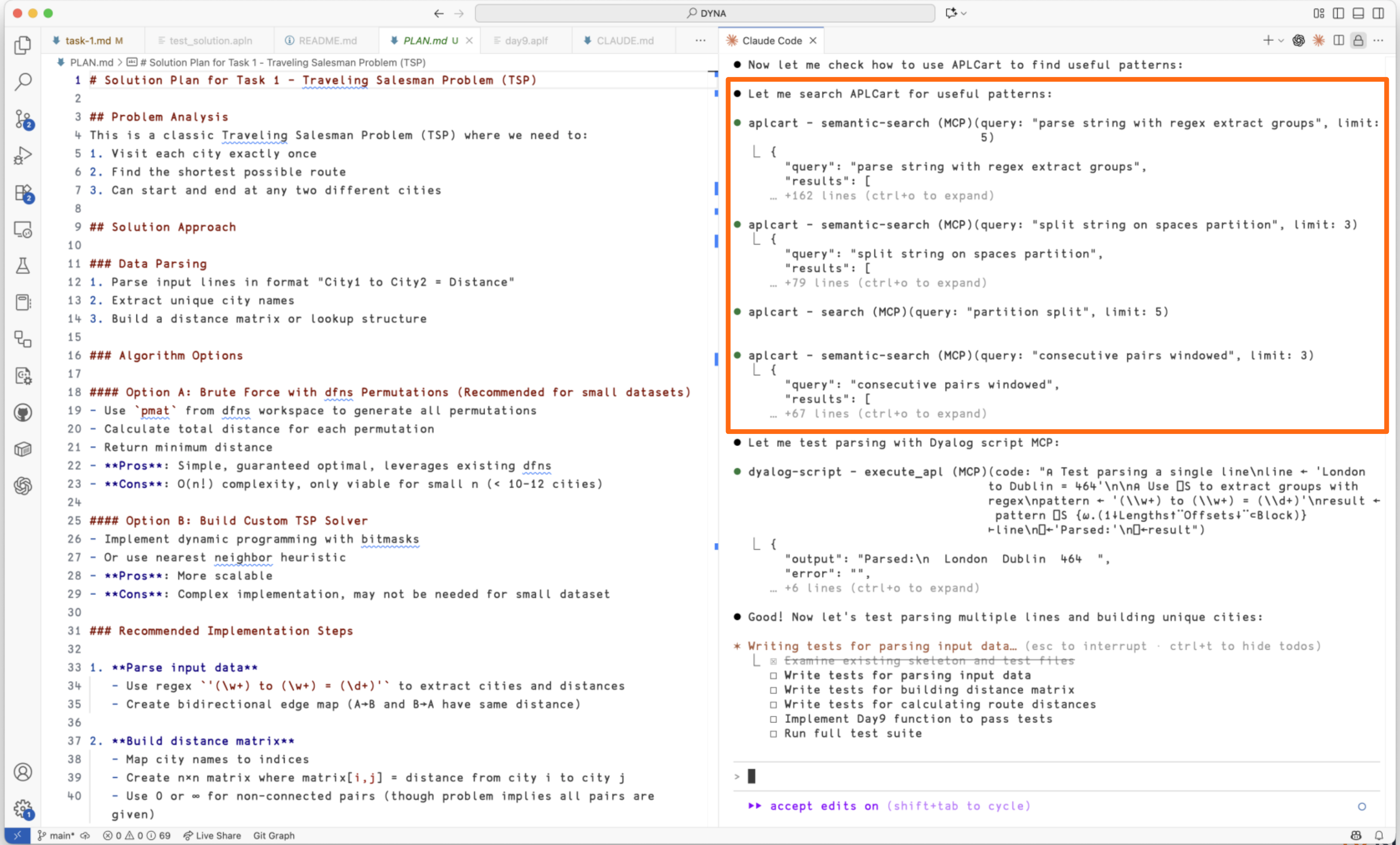
```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
given)
```

```
● Read(coderules.md)
  ⎿ Read 36 lines (ctrl+o to expand)

● ✅ Acknowledgment of Code Rules

  I have read and understood ALL mandatory coding rules in coderules.md. I will
  enforce them absolutely without exception throughout our session:

  Key Rules I'll Enforce:

  - NEVER claim completion with regressions - Run full test suite before/after each
  work unit
  - Use dfn syntax over tradfns, write TERSE APL code
  - Practice TDD - Tests first, implementation follows
  - Use APLCart MCP liberally for APL idioms
  - Use Dyalog-script MCP for testing snippets
  - NEVER invent ⎕ system functions - Verify against
  docs/reference/dyalog_system_functions.md
  - No temp files outside repo, clean up debug scripts
  - Create new git branches for development stages

  These rules are BINDING and OVERRIDE all default behaviors. I will treat any
  violation as a critical error.

  Ready to proceed with strict compliance to these rules.

> Read the task description in task-1.md. Outline a solution in PLAN.md, and let's
  review.

● I'll read the task description and create a solution plan.

● Read(task-1.md)
  ⎿ Read 32 lines (ctrl+o to expand)

● Write(PLAN.md)
  ⎿ Wrote 74 lines to PLAN.md
     # Solution Plan for Task 1 - Traveling Salesman Problem (TSP)
     ## Problem Analysis
     This is a classic Traveling Salesman Problem (TSP) where we need to:
     1. Visit each city exactly once
     2. Find the shortest possible route
     3. Can start and end at any two different cities
     ## Solution Approach
     … +64 lines (ctrl+o to expand)

● Solution plan created in PLAN.md. The approach uses brute force with dfns' pmat
  function to generate all permutations, which is optimal for this small TSP problem.

>

  ▸▸ accept edits on (shift+tab to cycle)
```



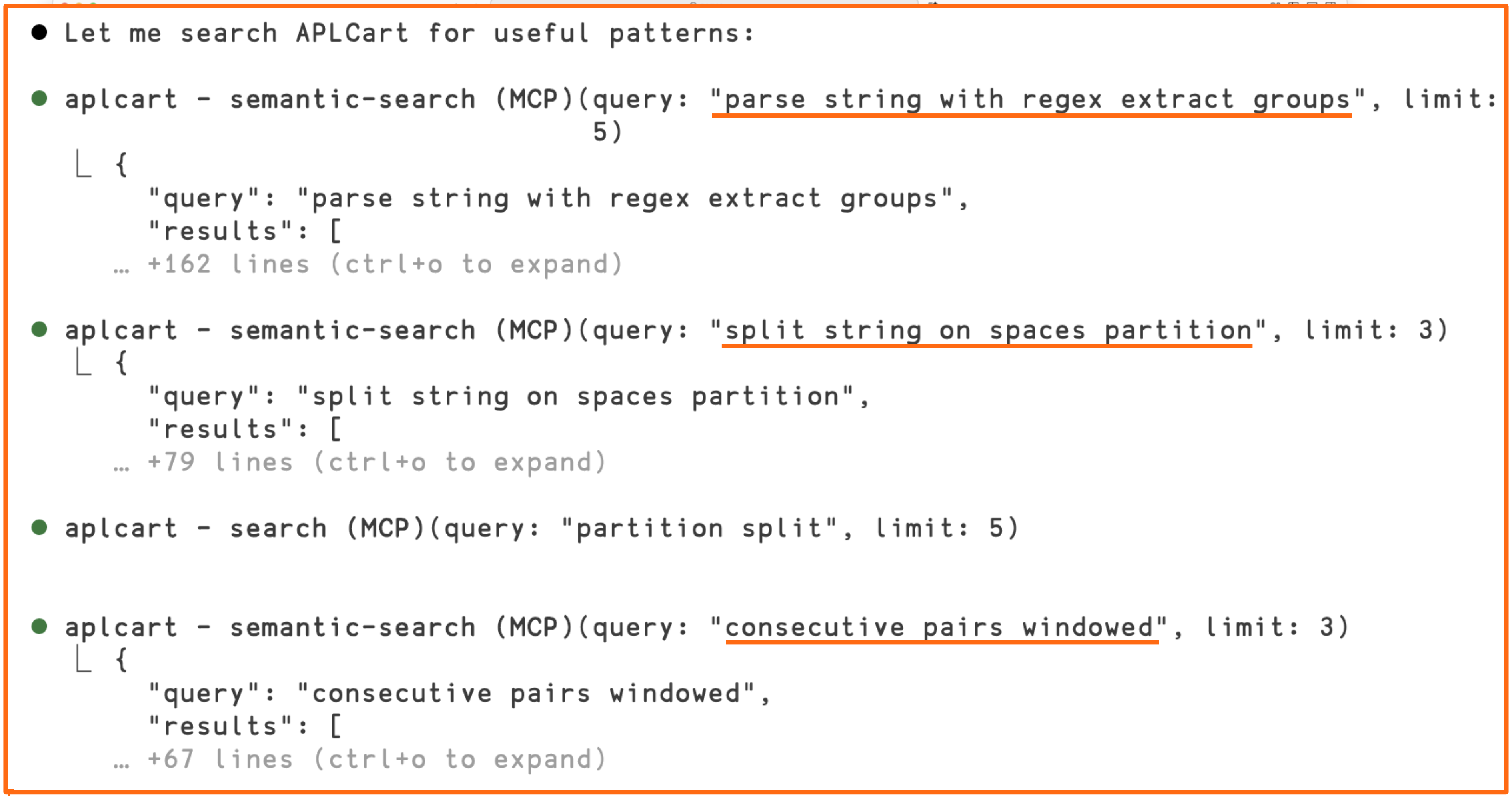
```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
given)
```

```
● Now let me check how to use APLCart to find useful patterns:

● Let me search APLCart for useful patterns:

● aplcart - semantic-search (MCP)(query: "parse string with regex extract groups", limit:
                                 5)
  └ {
      "query": "parse string with regex extract groups",
      "results": [
    … +162 lines (ctrl+o to expand)

● aplcart - semantic-search (MCP)(query: "split string on spaces partition", limit: 3)
  └ {
      "query": "split string on spaces partition",
      "results": [
    … +79 lines (ctrl+o to expand)

● aplcart - search (MCP)(query: "partition split", limit: 5)

● aplcart - semantic-search (MCP)(query: "consecutive pairs windowed", limit: 3)
  └ {
      "query": "consecutive pairs windowed",
      "results": [
    … +67 lines (ctrl+o to expand)

● Let me test parsing with Dyalog script MCP:

● dyalog-script - execute_apl (MCP)(code: "⍝ Test parsing a single line\nline ← 'London
                                   to Dublin = 464'\n\n⍝ Use ⎕S to extract groups with
                                   regex\npattern ← '(\\w+) to (\\w+) = (\\d+)'\nresult ←
                                    pattern ⎕S {⍵.(1↓Lengths↑¨Offsets↓¨⊂Block)}
                                   ⊢line\n⎕←'Parsed:'\n⎕←result")
  └ {
      "output": "Parsed:\n  London  Dublin  464  ",
      "error": "",
    … +6 lines (ctrl+o to expand)

● Good! Now let's test parsing multiple lines and building unique cities:

✶ Writing tests for parsing input data… (esc to interrupt · ctrl+t to hide todos)
  └ ☒ ~~Examine existing skeleton and test files~~
    ☐ Write tests for parsing input data
    ☐ Write tests for building distance matrix
    ☐ Write tests for calculating route distances
    ☐ Implement Day9 function to pass tests
    ☐ Run full test suite

> █

  ▸▸ accept edits on (shift+tab to cycle)                                              ○
```

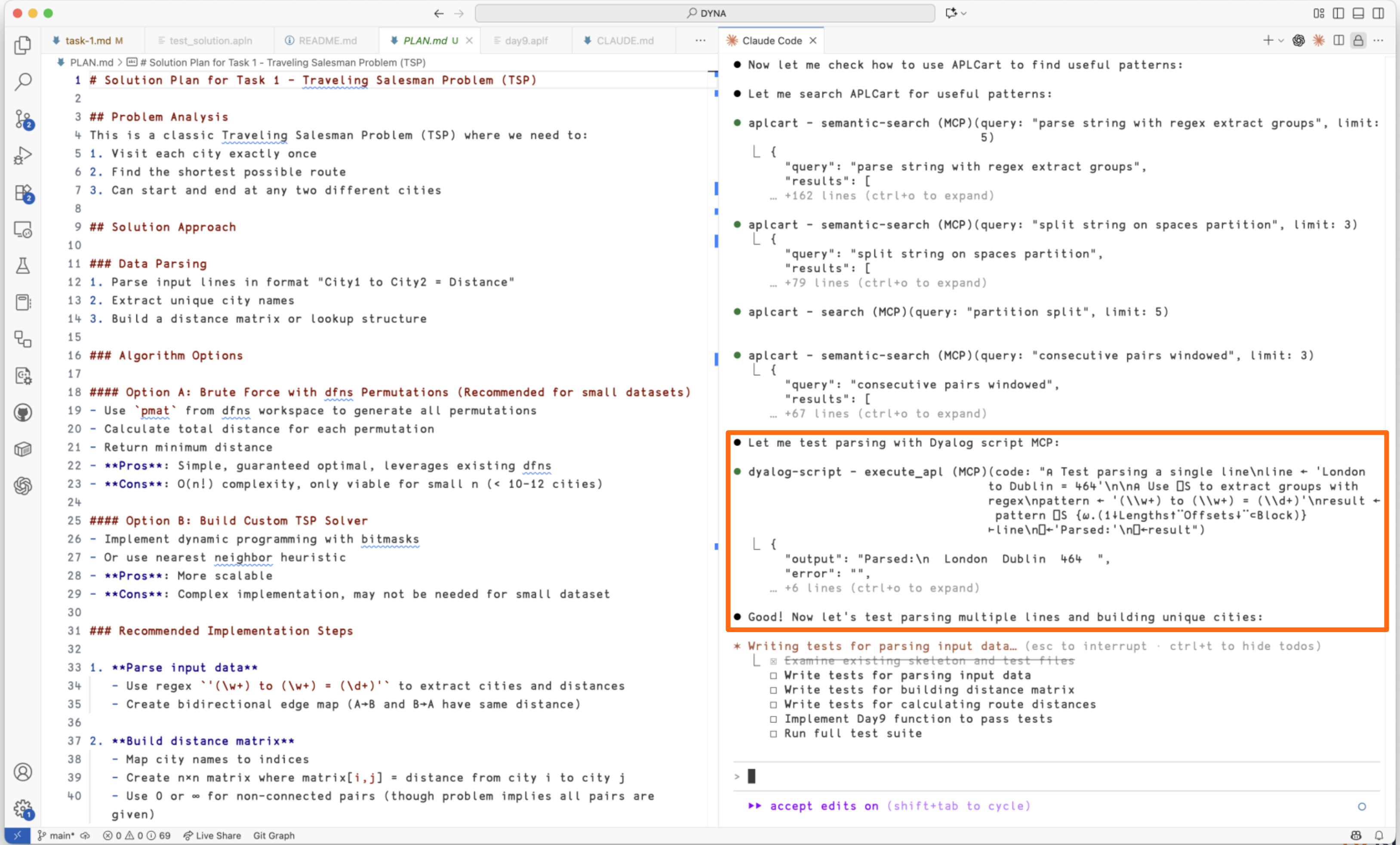


```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are given)
```

```
● Now let me check how to use APLCart to find useful patterns:

● Let me search APLCart for useful patterns:

● aplcart - semantic-search (MCP)(query: "parse string with regex extract groups", limit:
                                5)
  ⎿ {
       "query": "parse string with regex extract groups",
       "results": [
    … +162 lines (ctrl+o to expand)

● aplcart - semantic-search (MCP)(query: "split string on spaces partition", limit: 3)
  ⎿ {
       "query": "split string on spaces partition",
       "results": [
    … +79 lines (ctrl+o to expand)

● aplcart - search (MCP)(query: "partition split", limit: 5)

● aplcart - semantic-search (MCP)(query: "consecutive pairs windowed", limit: 3)
  ⎿ {
       "query": "consecutive pairs windowed",
       "results": [
    … +67 lines (ctrl+o to expand)

● Let me test parsing with Dyalog script MCP:

● dyalog-script - execute_apl (MCP)(code: "⍝ Test parsing a single line\nline ← 'London
                                   to Dublin = 464'\n\n⍝ Use ⎕S to extract groups with
                                   regex\npattern ← '(\\w+) to (\\w+) = (\\d+)'\nresult ←
                                    pattern ⎕S {⍵.(1↓Lengths↑¨Offsets↓¨⊂Block)}
                                   ⊢line\n⎕←'Parsed:'\n⎕←result")
  ⎿ {
       "output": "Parsed:\n  London  Dublin  464  ",
       "error": "",
    … +6 lines (ctrl+o to expand)

● Good! Now let's test parsing multiple lines and building unique cities:

* Writing tests for parsing input data… (esc to interrupt · ctrl+t to hide todos)
  ⎿ ☒ ~~Examine existing skeleton and test files~~
    ☐ Write tests for parsing input data
    ☐ Write tests for building distance matrix
    ☐ Write tests for calculating route distances
    ☐ Implement Day9 function to pass tests
    ☐ Run full test suite

>

▸▸ accept edits on (shift+tab to cycle)
```

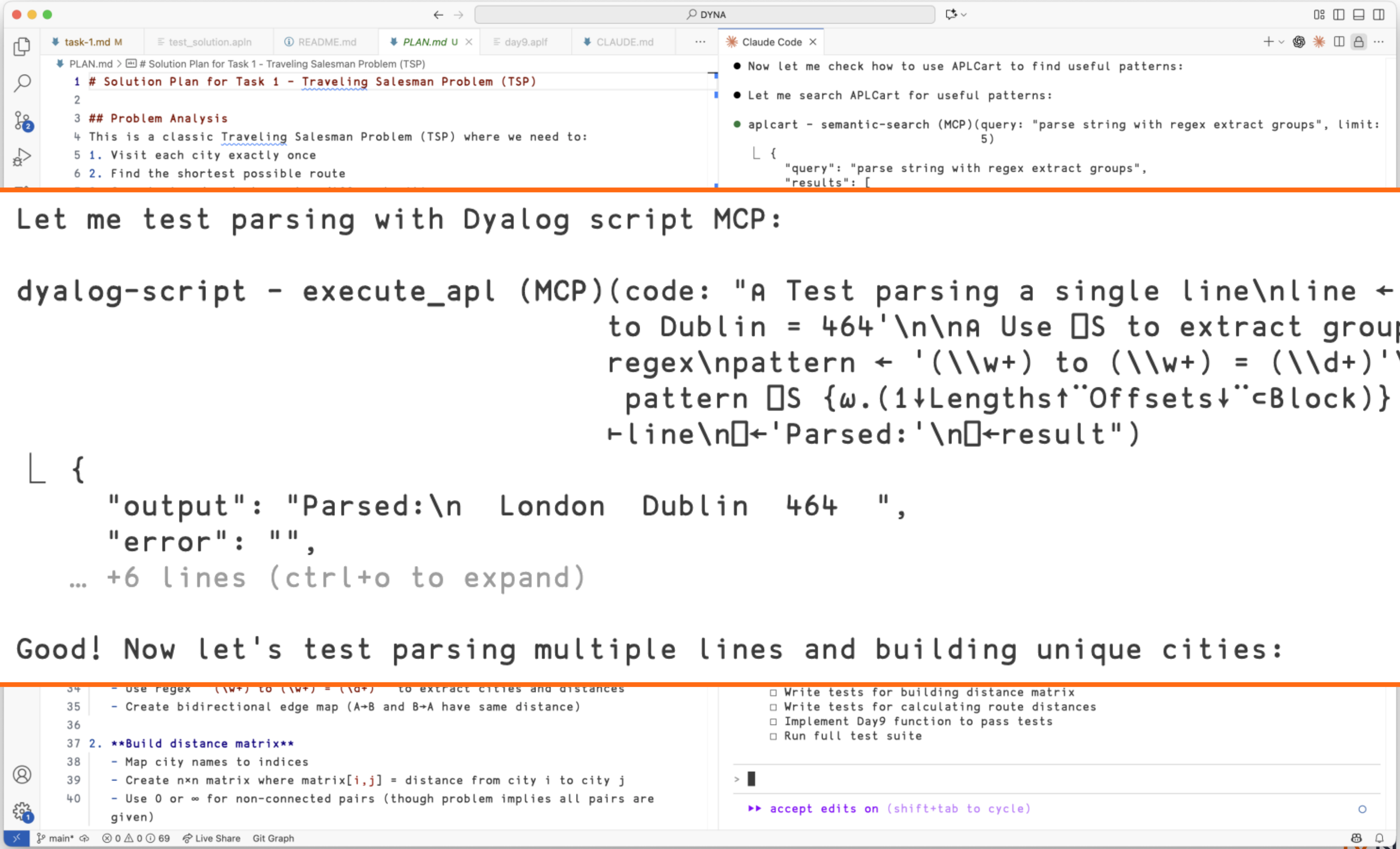


```
● Let me search APLCart for useful patterns:

● aplcart - semantic-search (MCP)(query: "parse string with regex extract groups", limit:
                                 5)
  ⎿ {
      "query": "parse string with regex extract groups",
      "results": [
    … +162 lines (ctrl+o to expand)

● aplcart - semantic-search (MCP)(query: "split string on spaces partition", limit: 3)
  ⎿ {
      "query": "split string on spaces partition",
      "results": [
    … +79 lines (ctrl+o to expand)

● aplcart - search (MCP)(query: "partition split", limit: 5)


● aplcart - semantic-search (MCP)(query: "consecutive pairs windowed", limit: 3)
  ⎿ {
      "query": "consecutive pairs windowed",
      "results": [
    … +67 lines (ctrl+o to expand)
```



```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
given)
```

```
● Now let me check how to use APLCart to find useful patterns:

● Let me search APLCart for useful patterns:

● aplcart - semantic-search (MCP)(query: "parse string with regex extract groups", limit:
                                5)
  ⎿ {
      "query": "parse string with regex extract groups",
      "results": [
    … +162 lines (ctrl+o to expand)

● aplcart - semantic-search (MCP)(query: "split string on spaces partition", limit: 3)
  ⎿ {
      "query": "split string on spaces partition",
      "results": [
    … +79 lines (ctrl+o to expand)

● aplcart - search (MCP)(query: "partition split", limit: 5)

● aplcart - semantic-search (MCP)(query: "consecutive pairs windowed", limit: 3)
  ⎿ {
      "query": "consecutive pairs windowed",
      "results": [
    … +67 lines (ctrl+o to expand)

● Let me test parsing with Dyalog script MCP:

● dyalog-script - execute_apl (MCP)(code: "⍝ Test parsing a single line\nline ← 'London
                                   to Dublin = 464'\n\n⍝ Use ⎕S to extract groups with
                                   regex\npattern ← '(\\w+) to (\\w+) = (\\d+)'\nresult ←
                                    pattern ⎕S {⍵.(1↓Lengths↑¨Offsets↓¨⊂Block)}
                                   ⊢line\n⎕←'Parsed:'\n⎕←result")
  ⎿ {
      "output": "Parsed:\n  London  Dublin  464  ",
      "error": "",
    … +6 lines (ctrl+o to expand)

● Good! Now let's test parsing multiple lines and building unique cities:

* Writing tests for parsing input data… (esc to interrupt · ctrl+t to hide todos)
  ⎿ ☒ ~~Examine existing skeleton and test files~~
    ☐ Write tests for parsing input data
    ☐ Write tests for building distance matrix
    ☐ Write tests for calculating route distances
    ☐ Implement Day9 function to pass tests
    ☐ Run full test suite

>

▸▸ accept edits on (shift+tab to cycle)
```



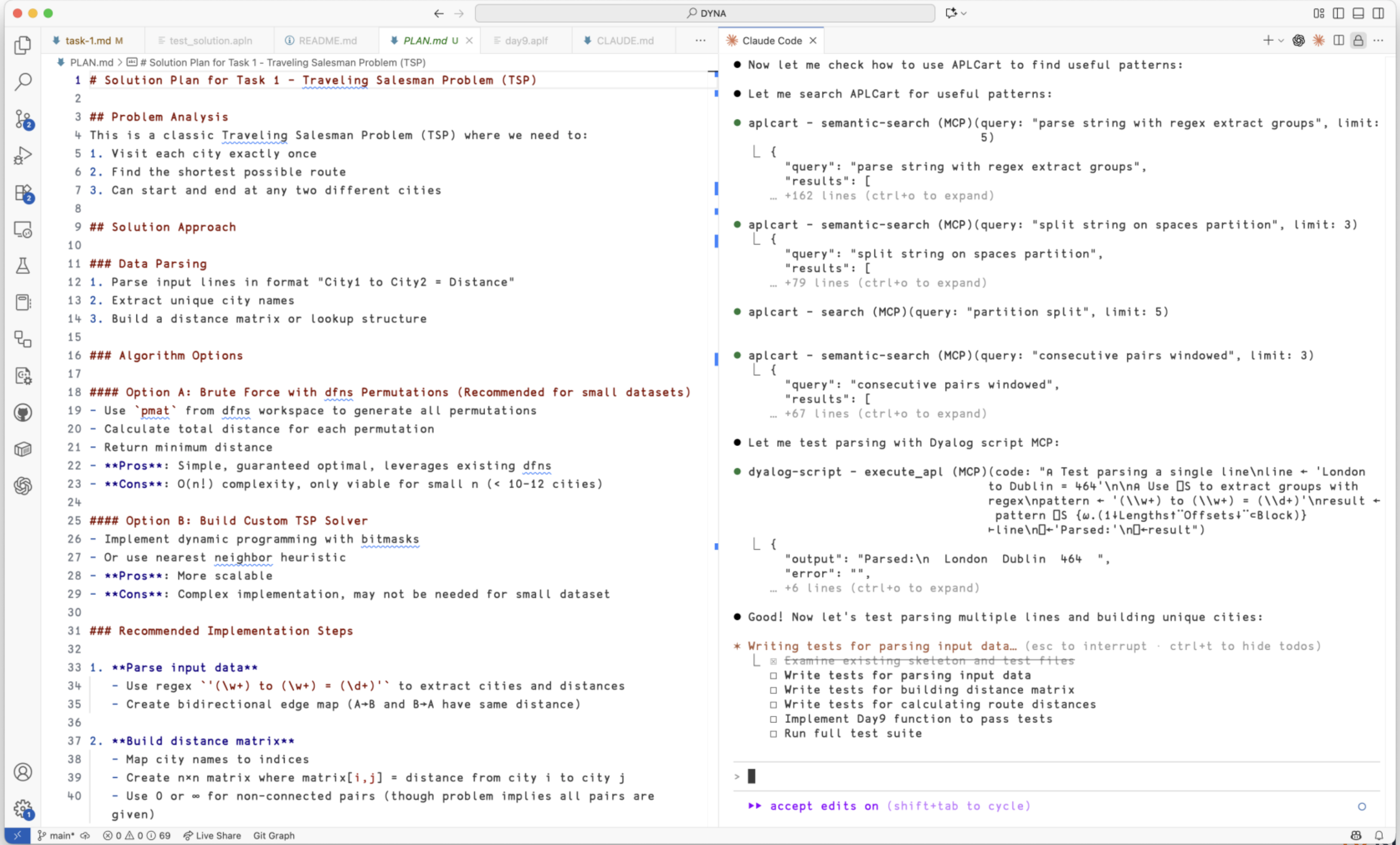
```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
```

```
● Now let me check how to use APLCart to find useful patterns:

● Let me search APLCart for useful patterns:

● aplcart - semantic-search (MCP)(query: "parse string with regex extract groups", limit:
                                  5)
  ⎿ {
      "query": "parse string with regex extract groups",
      "results": [
```

```
● Let me test parsing with Dyalog script MCP:

● dyalog-script - execute_apl (MCP)(code: "⍝ Test parsing a single line\nline ← 'London
                                   to Dublin = 464'\n\n⍝ Use ⎕S to extract groups with
                                   regex\npattern ← '(\\w+) to (\\w+) = (\\d+)'\nresult ←
                                    pattern ⎕S {⍵.(1↓Lengths↑¨Offsets↓¨⊂Block)}
                                   ⊢line\n⎕←'Parsed:'\n⎕←result")
  ⎿ {
      "output": "Parsed:\n  London  Dublin  464  ",
      "error": "",
    … +6 lines (ctrl+o to expand)

● Good! Now let's test parsing multiple lines and building unique cities:
```

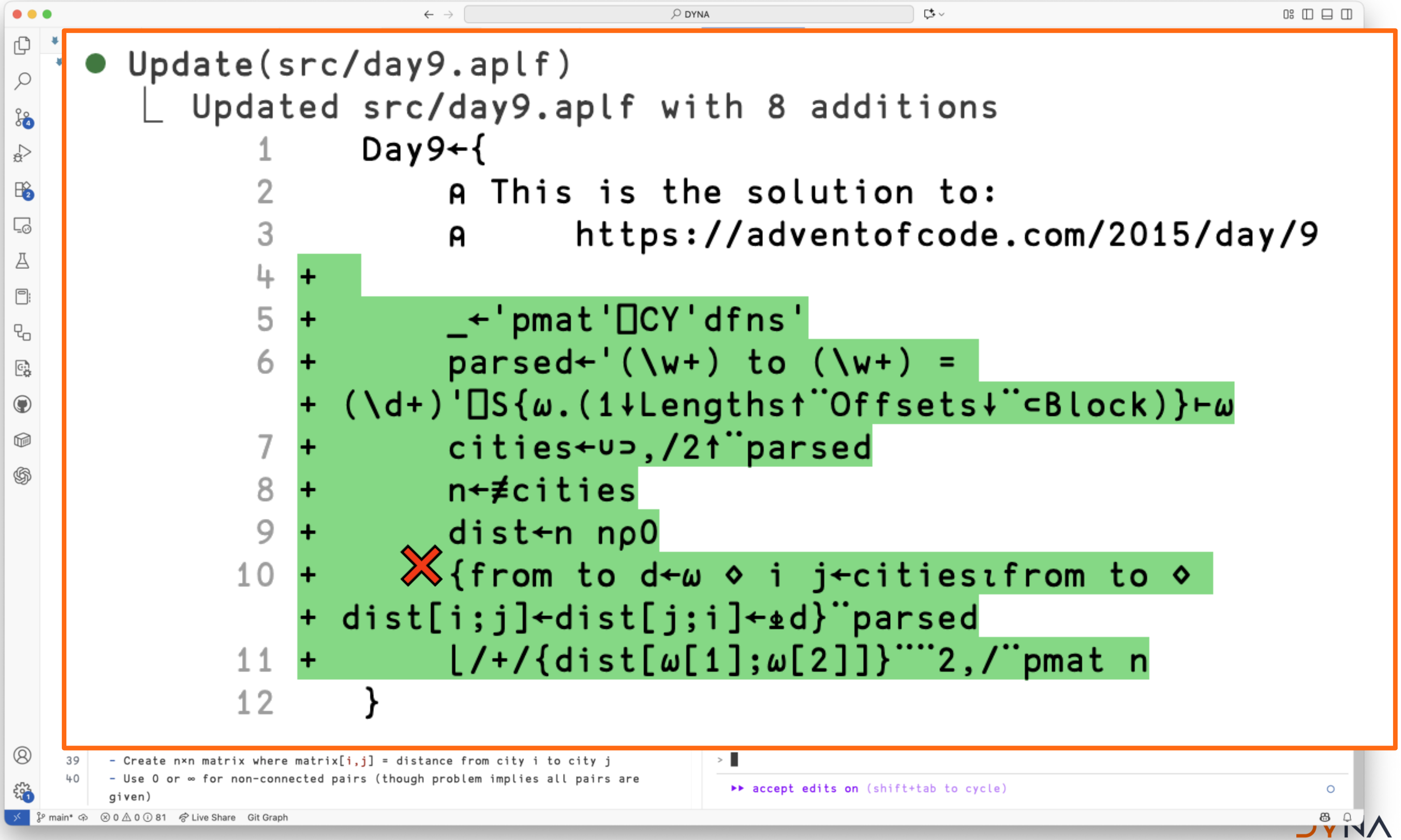
```
   - Use regex `(\w+) to (\w+) = (\d+)` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
   given)
```

```
     ☐ Write tests for building distance matrix
     ☐ Write tests for calculating route distances
     ☐ Implement Day9 function to pass tests
     ☐ Run full test suite

> ▌

 ▸▸ accept edits on (shift+tab to cycle)
```



```
# Solution Plan for Task 1 - Traveling Salesman Problem (TSP)

## Problem Analysis
This is a classic Traveling Salesman Problem (TSP) where we need to:
1. Visit each city exactly once
2. Find the shortest possible route
3. Can start and end at any two different cities

## Solution Approach

### Data Parsing
1. Parse input lines in format "City1 to City2 = Distance"
2. Extract unique city names
3. Build a distance matrix or lookup structure

### Algorithm Options

#### Option A: Brute Force with dfns Permutations (Recommended for small datasets)
- Use `pmat` from dfns workspace to generate all permutations
- Calculate total distance for each permutation
- Return minimum distance
- **Pros**: Simple, guaranteed optimal, leverages existing dfns
- **Cons**: O(n!) complexity, only viable for small n (< 10-12 cities)

#### Option B: Build Custom TSP Solver
- Implement dynamic programming with bitmasks
- Or use nearest neighbor heuristic
- **Pros**: More scalable
- **Cons**: Complex implementation, may not be needed for small dataset

### Recommended Implementation Steps

1. **Parse input data**
   - Use regex `'(\w+) to (\w+) = (\d+)'` to extract cities and distances
   - Create bidirectional edge map (A→B and B→A have same distance)

2. **Build distance matrix**
   - Map city names to indices
   - Create n×n matrix where matrix[i,j] = distance from city i to city j
   - Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
given)
```

```
● Now let me check how to use APLCart to find useful patterns:

● Let me search APLCart for useful patterns:

● aplcart - semantic-search (MCP)(query: "parse string with regex extract groups", limit:
                                  5)
  └ {
      "query": "parse string with regex extract groups",
      "results": [
    … +162 lines (ctrl+o to expand)

● aplcart - semantic-search (MCP)(query: "split string on spaces partition", limit: 3)
  └ {
      "query": "split string on spaces partition",
      "results": [
    … +79 lines (ctrl+o to expand)

● aplcart - search (MCP)(query: "partition split", limit: 5)

● aplcart - semantic-search (MCP)(query: "consecutive pairs windowed", limit: 3)
  └ {
      "query": "consecutive pairs windowed",
      "results": [
    … +67 lines (ctrl+o to expand)

● Let me test parsing with Dyalog script MCP:

● dyalog-script - execute_apl (MCP)(code: "⍝ Test parsing a single line\nline ← 'London
                                   to Dublin = 464'\n\n⍝ Use ⎕S to extract groups with
                                   regex\npattern ← '(\\w+) to (\\w+) = (\\d+)'\nresult ←
                                    pattern ⎕S {⍵.(1↓Lengths↑¨Offsets↓¨⊂Block)}
                                   ⊢line\n⎕←'Parsed:'\n⎕←result")
  └ {
      "output": "Parsed:\n  London  Dublin  464  ",
      "error": "",
    … +6 lines (ctrl+o to expand)

● Good! Now let's test parsing multiple lines and building unique cities:

✶ Writing tests for parsing input data… (esc to interrupt · ctrl+t to hide todos)
  └ ☒ ~~Examine existing skeleton and test files~~
    ☐ Write tests for parsing input data
    ☐ Write tests for building distance matrix
    ☐ Write tests for calculating route distances
    ☐ Implement Day9 function to pass tests
    ☐ Run full test suite

>

 ▸▸ accept edits on (shift+tab to cycle)                                              ○
```



```
● Update(src/day9.aplf)
  └ Updated src/day9.aplf with 8 additions
         Day9←{
               ⍝ This is the solution to:
               ⍝       https://adventofcode.com/2015/day/9
     +
     +         _←'pmat'⎕CY'dfns'
     +         parsed←'(\w+) to (\w+) =
     + (\d+)'⎕S{⍵.(1↓Lengths↑¨Offsets↓¨⊂Block)}⊢⍵
     +         cities←∪⊃,/2↑¨parsed
     +         n←≢cities
     +         dist←n n⍴0
     +         {from to d←⍵ ⋄ i j←cities⍳from to ⋄
     + dist[i;j]←dist[j;i]←⍎d}¨parsed
     +         ⌊/+/{dist[⍵[1];⍵[2]]}¨¨2,/¨pmat n
         }
```

```
- Create n×n matrix where matrix[i,j] = distance from city i to city j
- Use 0 or ∞ for non-connected pairs (though problem implies all pairs are
given)
```

```
>
▸▸ accept edits on (shift+tab to cycle)
```

```
Day9←{
    _←'pmat'⎕CY'dfns'
    parsed←'(\w+) to (\w+) = (\d+)'⎕S{⍵.(1↓Lengths↑¨Offsets↓¨⊂Block)}⊢⍵
    cities←∪⊃,/2↑¨parsed
    n←≢cities
    dist←n n⍴0
    _←{
        from to d←⍵
        i j←cities⍳from to
        dist[i;j]←dist[j;i]←⍎d
    }¨parsed
    ⌊/{+/{dist[⊃⍵;⊃⌽⍵]}¨2,/⍵}¨↓pmat n
}
```

```
'pmat'⎕CY'dfns'
data←' '(≠⊆⊢)¨'(to|=)'⎕R' '⊃⎕NGET'data9.txt'1
Day9←{
    weights←w,w←⍎¨⊢/↑⍵
    edges←(⊢⍪⌽)28 2⍴(∪⍳⊢),(↑⍵)[;1 2]
    adj←weights@(↓edges)⊢0⍴⍨2⍴8
    (⌊/,⌈/)+/adj[↓{∊⍵}⌺1 2⊢pmat 8]
}
```

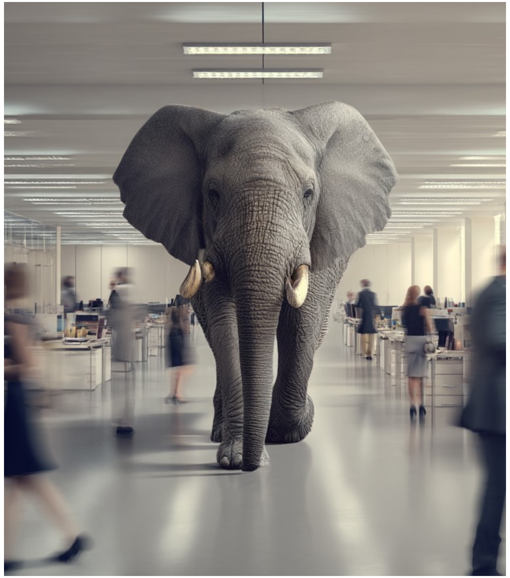
Agent loop + tools use > single shot

Productivity gain: net-negative

About a day's worth of tweaking

The Elephant in the Room

- Models just aren't good enough at APL
- ...yet.
- Lack of training data
- Lack of incentives
- No APL-specific tokenisers

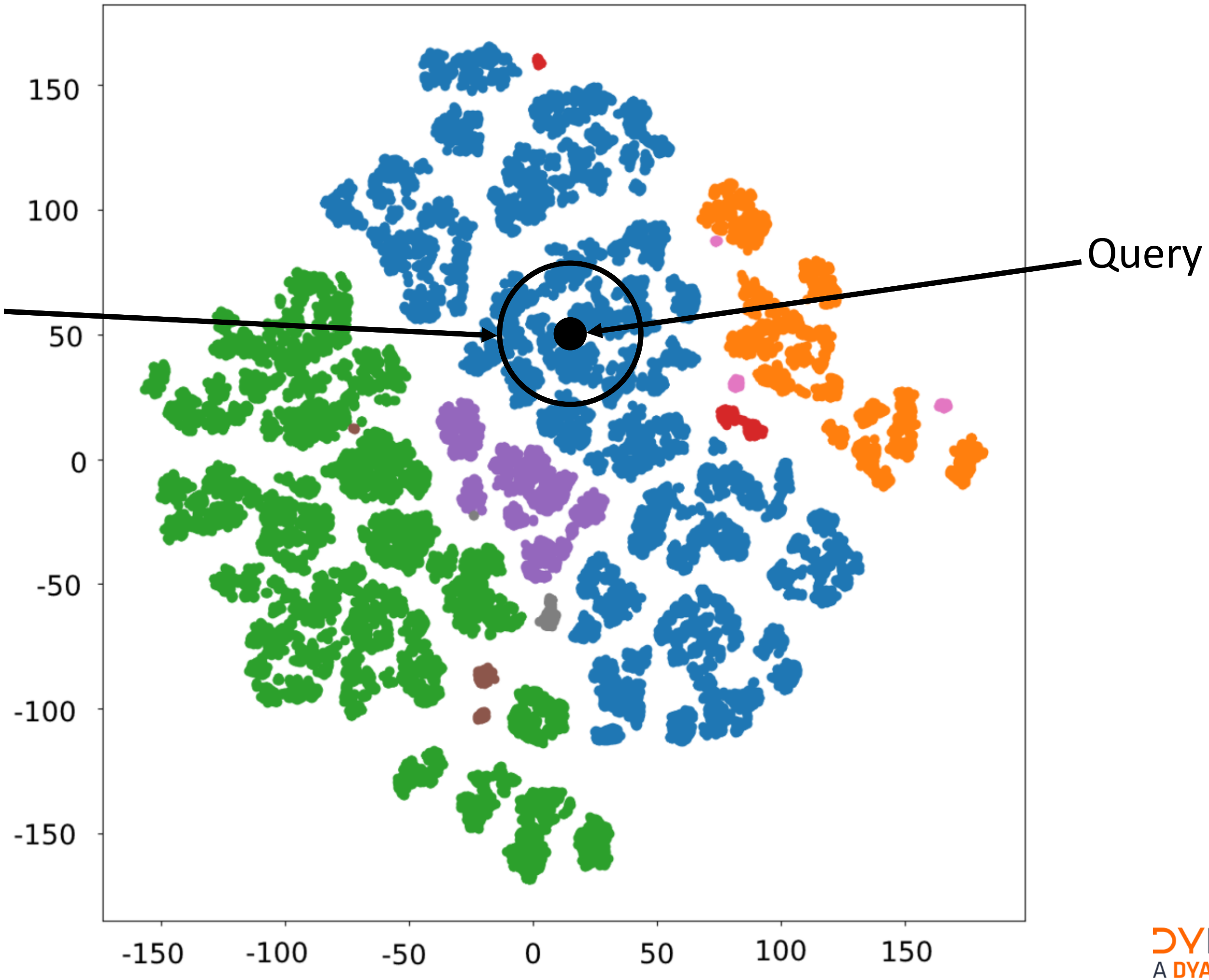


Claude

Semantic Search; RAG

AI Applications: Semantic Search

- Semantic search: "what I mean, not what I say"
- Use a model to create vector embeddings to encode *meaning*
- Query becomes a nearest neighbor problem in a vector space


Closest in meaning
Query
150
100
50
0
-50
-100
-150
-150
-100
-50
0
50
100
150

Search Dyalog's Docs

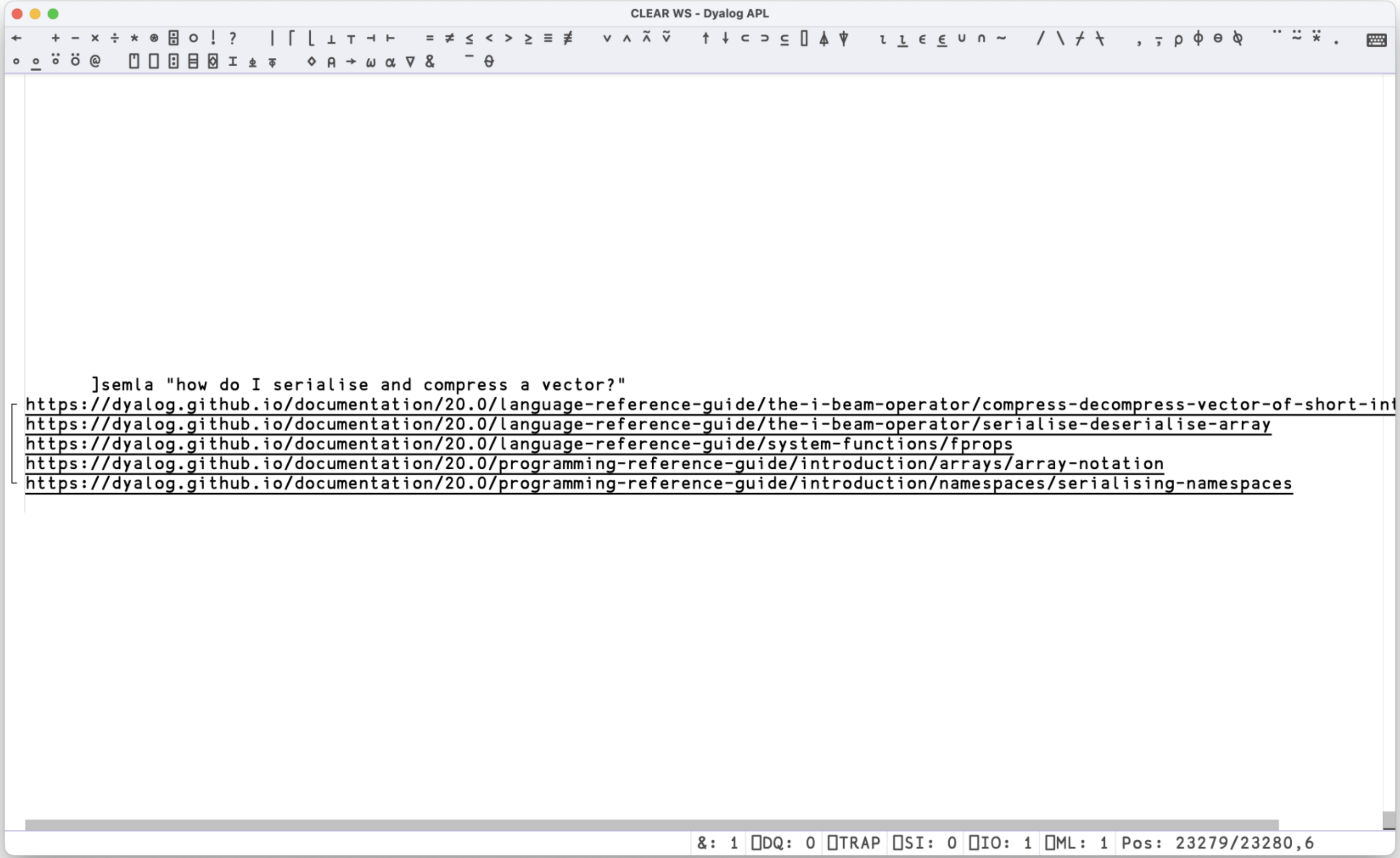
```
curl -s -X POST "http://localhost:8000/search" \
     -H "Content-Type: application/json" \
     -d '{"query": "how do I serialise and compress a vector?",
```

Search Dyalog's Docs

```
{
  "urls": [
    "https://dyalog.github.io/documentation/20.0/language-reference-guide/the-i-beam-operator/compress-decompress-vector-of-short-integers",
    "https://dyalog.github.io/documentation/20.0/language-reference-guide/the-i-beam-operator/serialise-deserialise-array",
    "https://dyalog.github.io/documentation/20.0/language-reference-guide/system-functions/fprops",
    "https://dyalog.github.io/documentation/20.0/programming-reference-guide/introduction/arrays/array-notation",
    "https://dyalog.github.io/documentation/20.0/programming-reference-guide/introduction/namespaces/serialising-namespaces"
  ],
  "scores": [
    0.5287122130393982,
    0.4902539551258087,
    0.42364850640296936,
    0.40454745292663574,
    0.3980979919433594
  ]
}
```

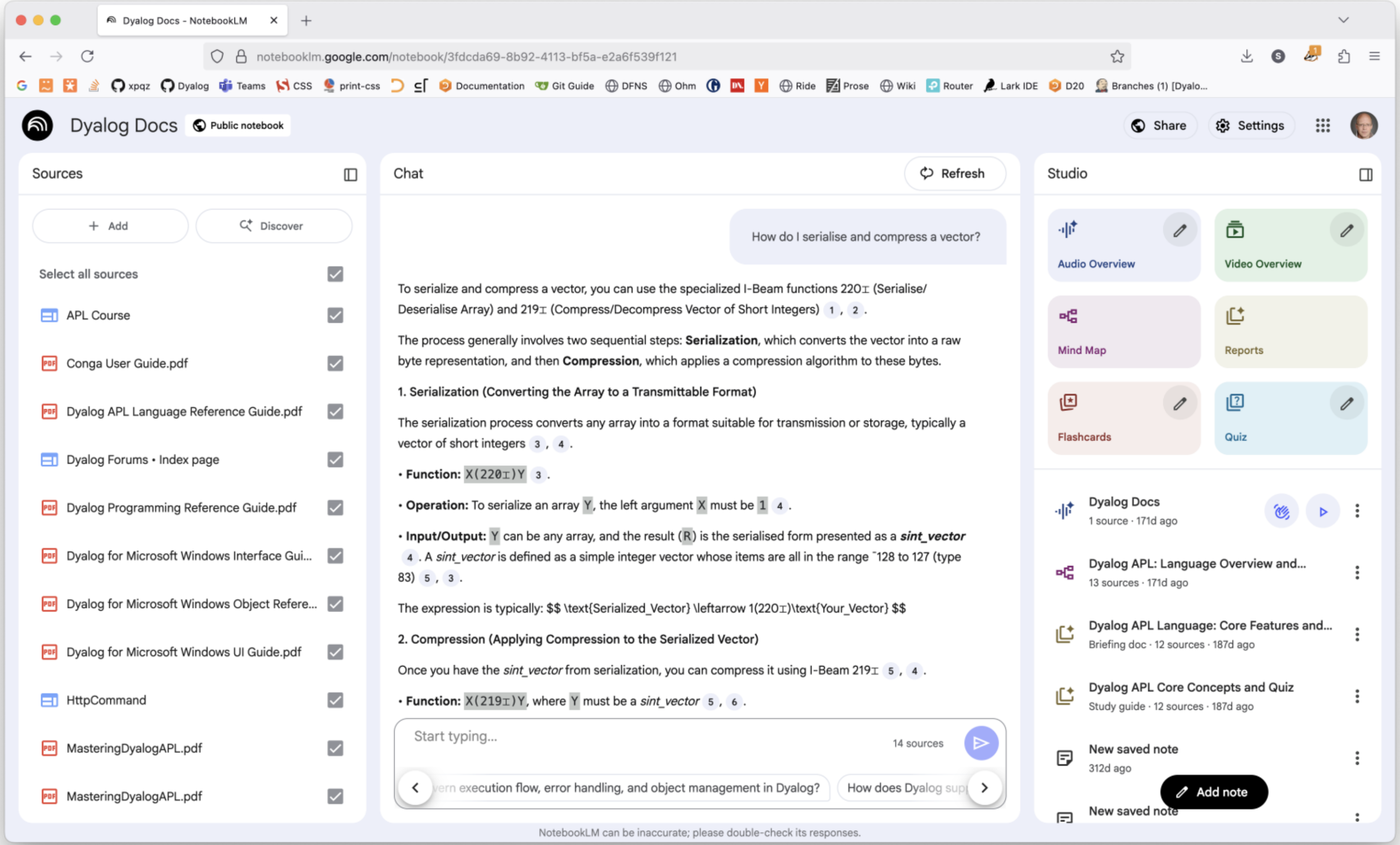
Search Dyalog's Docs

```
{
  "urls": [
    ".../the-i-beam-operator/compress-decompress-vector-of-short-integers",
    ".../the-i-beam-operator/serialise-deserialise-array",
    ".../system-functions/fprops",
    ".../introduction/arrays/array-notation",
    ".../introduction/namespaces/serialising-namespaces"
  ],
  "scores": [
      ...
  ]
}
```



```
      ]semla "how do I serialise and compress a vector?"
┌https://dyalog.github.io/documentation/20.0/language-reference-guide/the-i-beam-operator/compress-decompress-vector-of-short-int
│https://dyalog.github.io/documentation/20.0/language-reference-guide/the-i-beam-operator/serialise-deserialise-array
│https://dyalog.github.io/documentation/20.0/language-reference-guide/system-functions/fprops
│https://dyalog.github.io/documentation/20.0/programming-reference-guide/introduction/arrays/array-notation
└https://dyalog.github.io/documentation/20.0/programming-reference-guide/introduction/namespaces/serialising-namespaces
```

Gemini


Dyalog Docs - NotebookLM
notebooklm.google.com/notebook/3fdcda69-8b92-4113-bf5a-e2a6f539f121
xpqz
Dyalog
Teams
CSS
print-css
Documentation
Git Guide
DFNS
Ohm
Ride
Prose
Wiki
Router
Lark IDE
D20
Branches (1) [Dyalo...
Dyalog Docs
Public notebook
Share
Settings
Sources
Add
Discover
Select all sources
APL Course
Conga User Guide.pdf
Dyalog APL Language Reference Guide.pdf
Dyalog Forums • Index page
Dyalog Programming Reference Guide.pdf
Dyalog for Microsoft Windows Interface Gui...
Dyalog for Microsoft Windows Object Refere...
Dyalog for Microsoft Windows UI Guide.pdf
HttpCommand
MasteringDyalogAPL.pdf
MasteringDyalogAPL.pdf
Chat
Refresh
Studio
Audio Overview
Video Overview
Mind Map
Reports
Flashcards
Quiz
Dyalog Docs
1 source · 171d ago
Dyalog APL: Language Overview and...
13 sources · 171d ago
Dyalog APL Language: Core Features and...
Briefing doc · 12 sources · 187d ago
Dyalog APL Core Concepts and Quiz
Study guide · 12 sources · 187d ago
New saved note
312d ago
Add note
New saved note
Start typing...
14 sources
...ern execution flow, error handling, and object management in Dyalog?
How does Dyalog sup...
NotebookLM can be inaccurate; please double-check its responses.

How do I serialise and compress a vector?

To serialize and compress a vector, you can use the specialized I-Beam functions 220⌶ (Serialise/Deserialise Array) and 219⌶ (Compress/Decompress Vector of Short Integers) [1], [2].

The process generally involves two sequential steps: **Serialization**, which converts the vector into a raw byte representation, and then **Compression**, which applies a compression algorithm to these bytes.

1. Serialization (Converting the Array to a Transmittable Format)

The serialization process converts any array into a format suitable for transmission or storage, typically a vector of short integers [3], [4].

- **Function:** `X(220⌶)Y` [3].
- **Operation:** To serialize an array `Y`, the left argument `X` must be `1` [4].
- **Input/Output:** `Y` can be any array, and the result (`R`) is the serialised form presented as a ***sint_vector*** [4]. A *sint_vector* is defined as a simple integer vector whose items are all in the range ¯128 to 127 (type 83) [5], [3].

The expression is typically: $$ \text{Serialized_Vector} \leftarrow 1(220⌶)\text{Your_Vector} $$

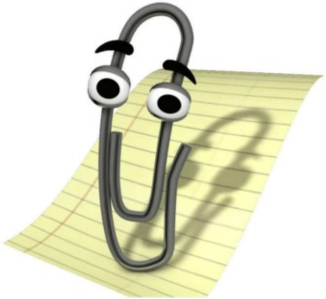
2. Compression (Applying Compression to the Serialized Vector)

Once you have the *sint_vector* from serialization, you can compress it using I-Beam 219⌶ [5], [4].

- **Function:** `X(219⌶)Y`, where `Y` must be a *sint_vector* [5], [6].

Client libraries

Client libraries



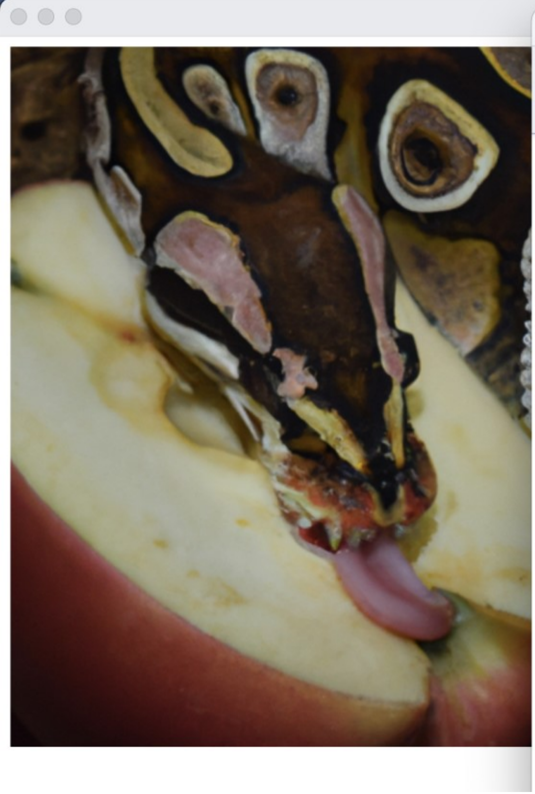
- Build AI-enabled applications in Dyalog APL
- "Smart data", forecasts, chatbots, assistants...

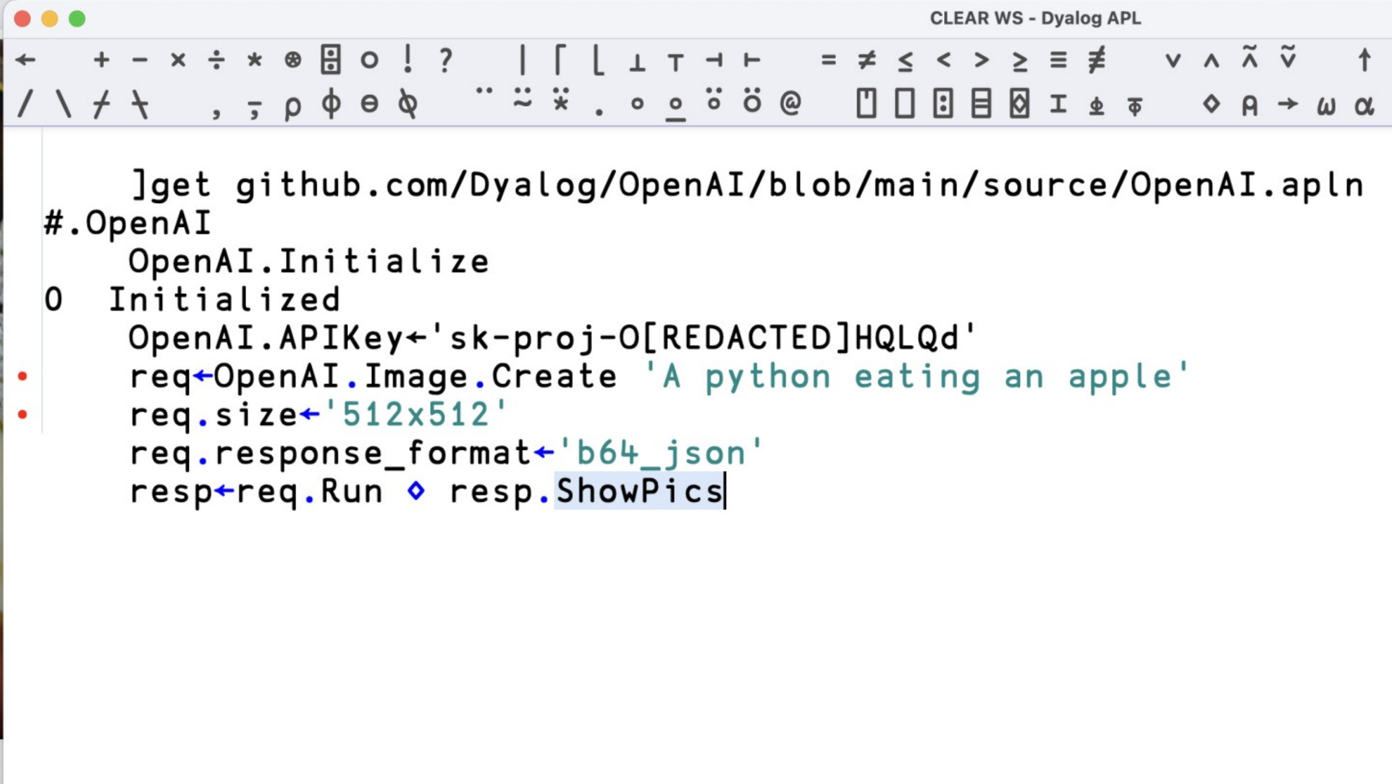
Client libraries

- Dyalog provides basic client libraries for OpenAI-compatible providers

https://dyalog.github.io/OpenAI

- Bigger job than it seems: API *spec* alone is 40k lines...





```
      ]get github.com/Dyalog/OpenAI/blob/main/source/OpenAI.apln
#.OpenAI
      OpenAI.Initialize
0  Initialized
      OpenAI.APIKey←'sk-proj-O[REDACTED]HQLQd'
      req←OpenAI.Image.Create 'A python eating an apple'
      req.size←'512x512'
      req.response_format←'b64_json'
      resp←req.Run ⋄ resp.ShowPics
```

State of the APL-AI nation

- Situation is rapidly improving, but...
- ...fair way to go before improving dev productivity
- Agents looks like the shape of AI-for-development
- Semantic search, RAG: solid today
- Build your own AI-applications with Dyalog

What will Dyalog do?

- Improve CLI story: testing, code evaluation
- New AI-friendly documentation format
- Improve error messaging, stack traces
- Client libraries
- Examples, training materials
- Publish more APL code

I LIKE
APL
KEN WANTS
YOU FOR
APL

Lack of APL code 'out there'

Can *you* help? Talk to us.