Hackathon II

**The Evolution of Todo – Mastering Spec-Driven Development & Cloud Native AI**

The future of software development is **AI-native and spec-driven**. As AI agents like Claude Code become more powerful, the role of the engineer shifts from "syntax writer" to "system architect." We have already explored Spec-Driven Book Authoring. Now, we want you to master the **Architecture of Intelligence**.

In this hackathon, you will master the art of building applications iteratively—starting from a simple console app and evolving it into a fully-featured, cloud-native AI chatbot deployed on Kubernetes. This journey will teach you the [Nine Pillars of AI-Driven Development](https://ai-native.panaversity.org/docs/Introducing-AI-Driven-Development/nine-pillars), [Claude Code](https://ai-native.panaversity.org/docs/AI-Tool-Landscape/claude-code-features-and-workflows), [Spec-Driven Development with Reusable Intelligence](https://ai-native.panaversity.org/docs/SDD-RI-Fundamentals) and Cloud-Native AI technologies through hands-on implementation.

**Excel in the Hackathon and Launch Your Journey as an AI Startup Founder 🚀**We've recently launched **Panaversity (panaversity.org)**, an initiative focused on teaching cutting-edge AI courses. If you perform well in this hackathon, you may be invited for an interview to join the **Panaversity core team** and potentially step into the role of a **startup founder** within this growing ecosystem. You will get a chance to work with Panaversity founders Zia, Rehan, Junaid, and Wania and become the very best. You may also get a chance to teach at Panaversity, PIAIC, and GIAIC.

# What You Will Learn

* Spec-Driven Development using Claude Code and Spec-Kit Plus
* Reusable Intelligence: Agents Skills and Subagent Development
* Full-Stack Development with Next.js, FastAPI, SQLModel, and Neon Serverless Database
* AI Agent Development using OpenAI Agents SDK and Official MCP SDK
* Cloud-Native Deployment with Docker, Kubernetes, Minikube, and Helm Charts
* Event-Driven Architecture using Kafka and Dapr
* AIOps with kubectl-ai, kagent and Claude Code
* Develop Cloud-Native Blueprints for Spec-Driven Deployment

Research Note: Deployment Blueprints for Spec-Driven Deployment

1. [Is Spec-Driven Development Key for Infrastructure Automation?](https://thenewstack.io/is-spec-driven-development-key-for-infrastructure-automation/)
2. [ChatGPT Progressive Learning Conversation](https://chatgpt.com/share/6924914a-43dc-8001-8f67-af29c4d9617e)
3. [Spec-Driven Cloud-Native Architecture: Governing AI Agents for Managed Services with Claude Code and SpecKit](https://claude.ai/public/artifacts/6025a232-6ebe-4c42-bb51-02dbd4603e18)

## Requirements

You are required to complete the **5-Phase "Evolution of Todo" Project** using Claude Code and Spec-Kit Plus. The core deliverables are:

* **Spec-Driven Implementation:** You must implement all **5 Phases** of the project (detailed below). You are strictly required to use **Spec-Driven Development**. You must write a Markdown Constitution and Spec for every feature of the phase, and use **Claude Code** to generate the implementation.
  + *Constraint:* You cannot write the code manually. You must refine the *Spec* until Claude Code generates the correct output.
* **Integrated AI Chatbot:** In Phases III, IV, and V, you must implement a conversational interface using **OpenAI Chatkit**, **OpenAI Agents SDK**, and **Official MCP SDK**. The bot must be able to manage the user's Todo list via natural language (e.g., "Reschedule my morning meetings to 2 PM").
* **Cloud Native Deployment:** In Phases IV and V, you must deploy the chatbot locally on Minikube, and on the cloud on DigitalOcean Kubernetes (DOKS).

# Todo App Feature Progression

## Basic Level (Core Essentials)

These form the foundation—quick to build, essential for any MVP:

1. Add Task – Create new todo items
2. Delete Task – Remove tasks from the list
3. Update Task – Modify existing task details
4. View Task List – Display all tasks
5. Mark as Complete – Toggle task completion status

## Intermediate Level (Organization & Usability)

Add these to make the app feel polished and practical:

1. Priorities & Tags/Categories – Assign levels (high/medium/low) or labels (work/home)
2. Search & Filter – Search by keyword; filter by status, priority, or date
3. Sort Tasks – Reorder by due date, priority, or alphabetically

## Advanced Level (Intelligent Features)

1. Recurring Tasks – Auto-reschedule repeating tasks (e.g., "weekly meeting")
2. Due Dates & Time Reminders – Set deadlines with date/time pickers; browser notifications

Use Agentic Dev Stack for building this hackathon project.

# Hackathon Phases Overview

| **Phase** | **Description** | **Technology Stack** | **Points** | **Due Date** |
| --- | --- | --- | --- | --- |
| Phase I | In-Memory Python Console App | Python, Claude Code, Spec-Kit Plus | 100 | Dec 7, 2025 |
| Phase II | Full-Stack Web Application | Next.js, FastAPI, SQLModel, Neon DB | 150 | Dec 14, 2025 |
| Phase III | AI-Powered Todo Chatbot | OpenAI ChatKit, Agents SDK, Official MCP SDK | 200 | Dec 21, 2025 |
| Phase IV | Local Kubernetes Deployment | Docker, Minikube, Helm, kubectl-ai, kagent | 250 | Jan 4, 2026 |
| Phase V | Advanced Cloud Deployment | Kafka, Dapr, DigitalOcean DOKS | 300 | Jan 18, 2026 |
| **TOTAL** |  |  | **1,000** |  |

# Bonus Points

Participants can earn additional bonus points for exceptional implementations:

| **Bonus Feature** | **Points** |
| --- | --- |
| Reusable Intelligence – Create and use reusable intelligence via Claude Code Subagents and Agent Skills | +200 |
| Create and use Cloud-Native Blueprints via Agent Skills | +200 |
| Multi-language Support – Support Urdu in chatbot | +100 |
| Voice Commands – Add voice input for todo commands | +200 |
| **TOTAL BONUS** | **+600** |

**Timeline**

* **Submission Deadline**: On Sundays on dates as mentioned above.
* **Live Presentations**: On Sundays, December 7, 14, and 21, 2025 and on January 4 and 18, 2026 starting at 8:00 PM on Zoom. Final Live Presentation date to be determined.

Top submissions will be invited via WhatsApp to present live on Zoom.

Note: All submissions will be evaluated. Live presentation is by invitation only, but does not affect final scoring.

| **Milestone** | **Date** | **Description** |
| --- | --- | --- |
| Hackathon Start | Monday, Dec 1, 2025 | Documentation released |
| Phase I Due | Sunday, Dec 7, 2025 | Console app checkpoint |
| Phase II Due | Sunday, Dec 14, 2025 | Web app checkpoint |
| Phase III Due | Sunday, Dec 21, 2025 | Chatbot checkpoint |
| Phase IV Due | Sunday, Jan 4, 2026 | Local K8s checkpoint |
| Final Submission | Sunday, Jan 18, 2026 | All phases complete |
| Live Presentations | Sundays, Dec 7, 14, 21, and Jan 4 and 18 | Top submissions present |

**Submit and Present Your Project:**

Once you have completed the project you will submit your project here at each phase:

<https://forms.gle/KMKEKaFUD6ZX4UtY8>   
  
**Submit the following via the form for each phase (You can submit a phase before the due date):**

1. Public GitHub Repo Link
2. Published App Link for Vercel.
3. Include a demo video link (must be under 90 seconds). Judges will only watch the first 90 seconds. You can [use NotebookLM](https://www.youtube.com/watch?v=_9TgVAYP3XA) or record your demo.
4. WhatsApp number (top submissions will be invited to present live)

Everyone is welcome to join the Zoom meeting to watch the presentations. Only invited participants will present their submissions. The meetings start at 8:00 PM on Sundays.

**Join Zoom Meeting**

* Time: 08:00 PM On Sundays, December 7, 14, and 21, 2025 and on January 4, 2026 starting at 8:00 PM on Zoom. Final Live Presentation date to be determined.
* <https://us06web.zoom.us/j/84976847088?pwd=Z7t7NaeXwVmmR5fysCv7NiMbfbhIda.1>
* Meeting ID: 849 7684 7088
* Passcode: 305850

## Project Details: The Evolution of Todo

Focus and Theme: From CLI to Distributed Cloud-Native AI Systems.

Goal: Students act as Product Architects, using AI to build progressively complex software without writing boilerplate code.

### Project Overview

This project simulates the real-world evolution of software. You will start with a simple script and end with a Kubernetes-managed, event-driven, AI-powered distributed system.

### Phase Breakdown

### Phase I: Todo In-Memory Python Console App

*Basic Level Functionality*

**Objective:** Build a command-line todo application that stores tasks in memory using Claude Code and Spec-Kit Plus.

💡**Development Approach:** Use the [Agentic Dev Stack workflow](#_32bnoy5itcj): Write spec → Generate plan → Break into tasks → Implement via Claude Code. No manual coding allowed. We will review the process, prompts, and iterations to judge each phase and project.

## Requirements

* Implement all 5 Basic Level features (Add, Delete, Update, View, Mark Complete)
* Use spec-driven development with Claude Code and Spec-Kit Plus
* Follow clean code principles and proper Python project structure

## Technology Stack

* UV
* Python 3.13+
* Claude Code
* Spec-Kit Plus

## Deliverables

1. GitHub repository with:

* Constitution file
* specs history folder containing all specification files
* /src folder with Python source code
* README.md with setup instructions
* CLAUDE.md with Claude Code instructions

1. Working console application demonstrating:

* Adding tasks with title and description
* Listing all tasks with status indicators
* Updating task details
* Deleting tasks by ID
* Marking tasks as complete/incomplete

## Windows Users: WSL 2 Setup

Windows users must use WSL 2 (Windows Subsystem for Linux) for development:

# Install WSL 2

wsl --install

# Set WSL 2 as default

wsl --set-default-version 2

# Install Ubuntu

wsl --install -d Ubuntu-22.04

# Phase II: Todo Full-Stack Web Application

*Basic Level Functionality*

**Objective:** Using Claude Code and Spec-Kit Plus transform the console app into a modern multi-user web application with persistent storage.

💡**Development Approach:** Use the [Agentic Dev Stack workflow](#_32bnoy5itcj): Write spec → Generate plan → Break into tasks → Implement via Claude Code. No manual coding allowed. We will review the process, prompts, and iterations to judge each phase and project.

## Requirements

* Implement all 5 Basic Level features as a web application
* Create RESTful API endpoints
* Build responsive frontend interface
* Store data in Neon Serverless PostgreSQL database
* Authentication – Implement user signup/signin using Better Auth

## Technology Stack

| **Layer** | **Technology** |
| --- | --- |
| Frontend | Next.js 16+ (App Router) |
| Backend | Python FastAPI |
| ORM | SQLModel |
| Database | Neon Serverless PostgreSQL |
| Spec-Driven | Claude Code + Spec-Kit Plus |
| Authentication | Better Auth |

## API Endpoints

| **Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| GET | /api/{user\_id}/tasks | List all tasks |
| POST | /api/{user\_id}/tasks | Create a new task |
| GET | /api/{user\_id}/tasks/{id} | Get task details |
| PUT | /api/{user\_id}/tasks/{id} | Update a task |
| DELETE | /api/{user\_id}tasks/{id} | Delete a task |
| PATCH | /api/{user\_id}tasks/{id}/complete | Toggle completion |

Securing the REST API

*Better Auth + FastAPI Integration*

# The Challenge

Better Auth is a JavaScript/TypeScript authentication library that runs on your **Next.js frontend**. However, your **FastAPI backend** is a separate Python service that needs to verify which user is making API requests.

# The Solution: JWT Tokens

Better Auth can be configured to issue **JWT (JSON Web Token)** tokens when users log in. These tokens are self-contained credentials that include user information and can be verified by any service that knows the secret key.

# How It Works

* User logs in on Frontend → Better Auth creates a session and issues a JWT token
* Frontend makes API call → Includes the JWT token in the Authorization: Bearer <token> header
* Backend receives request → Extracts token from header, verifies signature using shared secret
* Backend identifies user → Decodes token to get user ID, email, etc. and matches it with the user ID in the URL
* Backend filters data → Returns only tasks belonging to that user

# What Needs to Change

| **Component** | **Changes Required** |
| --- | --- |
| **Better Auth Config** | Enable JWT plugin to issue tokens |
| **Frontend API Client** | Attach JWT token to every API request header |
| **FastAPI Backend** | Add middleware to verify JWT and extract user |
| **API Routes** | Filter all queries by the authenticated user's ID |

# The Shared Secret

Both frontend (Better Auth) and backend (FastAPI) must use the **same secret key** for JWT signing and verification. This is typically set via environment variable **BETTER\_AUTH\_SECRET** in both services.

# Security Benefits

| **Benefit** | **Description** |
| --- | --- |
| **User Isolation** | Each user only sees their own tasks |
| **Stateless Auth** | Backend doesn't need to call frontend to verify users |
| **Token Expiry** | JWTs expire automatically (e.g., after 7 days) |
| **No Shared DB Session** | Frontend and backend can verify auth independently |

# API Behavior Change

**After Auth:**

| All endpoints require valid JWT token |
| --- |
| Requests without token receive 401 Unauthorized |
| Each user only sees/modifies their own tasks |
| Task ownership is enforced on every operation |

# Bottom Line

The REST API endpoints stay the same (**GET /api/user\_id/tasks**, **POST /api/user\_id/tasks**, etc.), but every request now must include a JWT token, and all responses are filtered to only include that user's data.

Monorepo Organization For Full-Stack Projects With GitHub Spec-Kit + Claude Code

This guide explains how to organize your Full-Stack Projects in a monorepo to integrate **GitHub Spec-Kit** for spec-driven development with **Claude Code**. This guide explains how to organize your repository so that Claude Code and Spec-Kit Plus can effectively edit both frontend (Next.js) and backend (FastAPI) code in a single context.

# Spec-Kit Monorepo Folder Structure

hackathon-todo/

├── .spec-kit/ # Spec-Kit configuration

│ └── config.yaml

├── specs/ # Spec-Kit managed specifications

│ ├── overview.md # Project overview

│ ├── architecture.md # System architecture

│ ├── features/ # Feature specifications

│ │ ├── task-crud.md

│ │ ├── authentication.md

│ │ └── chatbot.md

│ ├── api/ # API specifications

│ │ ├── rest-endpoints.md

│ │ └── mcp-tools.md

│ ├── database/ # Database specifications

│ │ └── schema.md

│ └── ui/ # UI specifications

│ ├── components.md

│ └── pages.md

├── CLAUDE.md # Root Claude Code instructions

├── frontend/

│ ├── CLAUDE.md

│ └── ... (Next.js app)

├── backend/

│ ├── CLAUDE.md

│ └── ... (FastAPI app)

├── docker-compose.yml

└── README.md

# Key Differences from Basic Monorepo

| **Aspect** | **Without Spec-Kit** | **With Spec-Kit** |
| --- | --- | --- |
| **Specs Location** | /specs (flat) | /specs (organized by type) |
| **Config File** | None | /.spec-kit/config.yaml |
| **Spec Format** | Freeform markdown | Spec-Kit conventions |
| **Referencing** | @specs/file.md | @specs/features/file.md |

# Spec-Kit Config File

# .spec-kit/config.yaml

name: hackathon-todo

version: "1.0"

structure:

specs\_dir: specs

features\_dir: specs/features

api\_dir: specs/api

database\_dir: specs/database

ui\_dir: specs/ui

phases:

- name: phase1-console

features: [task-crud]

- name: phase2-web

features: [task-crud, authentication]

- name: phase3-chatbot

features: [task-crud, authentication, chatbot]

# CLAUDE.md Files

Create multiple CLAUDE.md files to provide context at different levels:

# Root CLAUDE.md

# Todo App - Hackathon II

## Project Overview

This is a monorepo using GitHub Spec-Kit for spec-driven development.

## Spec-Kit Structure

Specifications are organized in /specs:

- /specs/overview.md - Project overview

- /specs/features/ - Feature specs (what to build)

- /specs/api/ - API endpoint and MCP tool specs

- /specs/database/ - Schema and model specs

- /specs/ui/ - Component and page specs

## How to Use Specs

1. Always read relevant spec before implementing

2. Reference specs with: @specs/features/task-crud.md

3. Update specs if requirements change

## Project Structure

- /frontend - Next.js 14 app

- /backend - Python FastAPI server

## Development Workflow

1. Read spec: @specs/features/[feature].md

2. Implement backend: @backend/CLAUDE.md

3. Implement frontend: @frontend/CLAUDE.md

4. Test and iterate

## Commands

- Frontend: cd frontend && npm run dev

- Backend: cd backend && uvicorn main:app --reload

- Both: docker-compose up

## Frontend CLAUDE.md

# Frontend Guidelines

## Stack

- Next.js 14 (App Router)

- TypeScript

- Tailwind CSS

## Patterns

- Use server components by default

- Client components only when needed (interactivity)

- API calls go through `/lib/api.ts`

## Component Structure

- `/components` - Reusable UI components

- `/app` - Pages and layouts

## API Client

All backend calls should use the api client:

import { api } from '@/lib/api'

const tasks = await api.getTasks()

## Styling

- Use Tailwind CSS classes

- No inline styles

- Follow existing component patterns

## 

## Backend CLAUDE.md

# Backend Guidelines

## Stack

- FastAPI

- SQLModel (ORM)

- Neon PostgreSQL

## Project Structure

- `main.py` - FastAPI app entry point

- `models.py` - SQLModel database models

- `routes/` - API route handlers

- `db.py` - Database connection

## API Conventions

- All routes under `/api/`

- Return JSON responses

- Use Pydantic models for request/response

- Handle errors with HTTPException

## Database

- Use SQLModel for all database operations

- Connection string from environment variable: DATABASE\_URL

## Running

uvicorn main:app --reload --port 8000

# Example Spec Files

## /specs/overview.md

# Todo App Overview

## Purpose

A todo application that evolves from console app to AI chatbot.

## Current Phase

Phase II: Full-Stack Web Application

## Tech Stack

- Frontend: Next.js 14, TypeScript, Tailwind CSS

- Backend: FastAPI, SQLModel, Neon PostgreSQL

- Auth: Better Auth with JWT

## Features

- [ ] Task CRUD operations

- [ ] User authentication

- [ ] Task filtering and sorting

## /specs/features/task-crud.md

# Feature: Task CRUD Operations

## User Stories

- As a user, I can create a new task

- As a user, I can view all my tasks

- As a user, I can update a task

- As a user, I can delete a task

- As a user, I can mark a task complete

## Acceptance Criteria

### Create Task

- Title is required (1-200 characters)

- Description is optional (max 1000 characters)

- Task is associated with logged-in user

### View Tasks

- Only show tasks for current user

- Display title, status, created date

- Support filtering by status

## /specs/api/rest-endpoints.md

# REST API Endpoints

## Base URL

- Development: http://localhost:8000

- Production: https://api.example.com

## Authentication

All endpoints require JWT token in header:

Authorization: Bearer <token>

## Endpoints

### GET /api/tasks

List all tasks for authenticated user.

Query Parameters:

- status: "all" | "pending" | "completed"

- sort: "created" | "title" | "due\_date"

Response: Array of Task objects

### POST /api/tasks

Create a new task.

Request Body:

- title: string (required)

- description: string (optional)

Response: Created Task object

## /specs/database/schema.md

# Database Schema

## Tables

### users (managed by Better Auth)

- id: string (primary key)

- email: string (unique)

- name: string

- created\_at: timestamp

### tasks

- id: integer (primary key)

- user\_id: string (foreign key -> users.id)

- title: string (not null)

- description: text (nullable)

- completed: boolean (default false)

- created\_at: timestamp

- updated\_at: timestamp

## Indexes

- tasks.user\_id (for filtering by user)

- tasks.completed (for status filtering)

# Workflow with Spec-KitPlus + Claude Code

* Write/Update Spec → @specs/features/new-feature.md
* Ask Claude Code to Implement → "Implement @specs/features/new-feature.md"
* Claude Code reads: Root CLAUDE.md, Feature spec, API spec, Database spec, Relevant CLAUDE.md
* Claude Code implements in both frontend and backend
* Test and iterate on spec if needed

# Referencing Specs in Claude Code

# Implement a feature

You: @specs/features/task-crud.md implement the create task feature

# Implement API

You: @specs/api/rest-endpoints.md implement the GET /api/tasks endpoint

# Update database

You: @specs/database/schema.md add due\_date field to tasks

# Full feature across stack

You: @specs/features/authentication.md implement Better Auth login

# Summary

| **Component** | **Purpose** |
| --- | --- |
| **/.spec-kit/config.yaml** | Spec-Kit configuration |
| **/specs/<features>/\*\*** | What to build |
| **/CLAUDE.md** | How to navigate and use specs |
| **/frontend/CLAUDE.md** | Frontend-specific patterns |
| **/backend/CLAUDE.md** | Backend-specific patterns |

**Key Point:**

Spec-Kit Plus provides organized, structured specs that Claude Code can reference. The CLAUDE.md files tell Claude Code how to use those specs and project-specific conventions.

# Summary: Monorepo vs Separate Repos

| **Approach** | **Pros** | **Cons** |
| --- | --- | --- |
| **Monorepo ⭐** | Single CLAUDE.md context, easier cross-cutting changes | Larger repo |
| Separate Repos | Clear separation, independent deployments | Claude Code needs workspace setup |

**Recommendation:**

Use monorepo for the hackathon – simpler for Claude Code to navigate and edit both frontend and backend in a single context.

# Key Benefits of This Structure

| **Benefit** | **Description** |
| --- | --- |
| **Single Context** | Claude Code sees entire project, can make cross-cutting changes |
| **Layered CLAUDE.md** | Root file for overview, subfolder files for specific guidelines |
| **Specs Folder** | Reference specifications directly with @specs/filename.md |
| **Clear Separation** | Frontend and backend code in separate folders, easy to navigate |

# Phase III: Todo AI Chatbot

*Basic Level Functionality*

**Objective:** Create an AI-powered chatbot interface for managing todos through natural language using MCP (Model Context Protocol) server architecture and using Claude Code and Spec-Kit Plus.

💡**Development Approach:** Use the [Agentic Dev Stack workflow](#_32bnoy5itcj): Write spec → Generate plan → Break into tasks → Implement via Claude Code. No manual coding allowed. We will review the process, prompts, and iterations to judge each phase and project.

# Requirements

1. Implement conversational interface for all Basic Level features
2. Use OpenAI Agents SDK for AI logic
3. Build MCP server with Official MCP SDK that exposes task operations as tools
4. Stateless chat endpoint that persists conversation state to database
5. AI agents use MCP tools to manage tasks. The MCP tools will also be stateless and will store state in the database.

# Technology Stack

| **Component** | **Technology** |
| --- | --- |
| Frontend | OpenAI ChatKit |
| Backend | Python FastAPI |
| AI Framework | OpenAI Agents SDK |
| MCP Server | Official MCP SDK |
| ORM | SQLModel |
| Database | Neon Serverless PostgreSQL |
| Authentication | Better Auth |

# Architecture

┌─────────────────┐ ┌──────────────────────────────────────────────┐ ┌─────────────────┐

│ │ │ FastAPI Server │ │ │

│ │ │ ┌────────────────────────────────────────┐ │ │ │

│ ChatKit UI │────▶│ │ Chat Endpoint │ │ │ Neon DB │

│ (Frontend) │ │ │ POST /api/chat │ │ │ (PostgreSQL) │

│ │ │ └───────────────┬────────────────────────┘ │ │ │

│ │ │ │ │ │ - tasks │

│ │ │ ▼ │ │ - conversations│

│ │ │ ┌────────────────────────────────────────┐ │ │ - messages │

│ │◀────│ │ OpenAI Agents SDK │ │ │ │

│ │ │ │ (Agent + Runner) │ │ │ │

│ │ │ └───────────────┬────────────────────────┘ │ │ │

│ │ │ │ │ │ │

│ │ │ ▼ │ │ │

│ │ │ ┌────────────────────────────────────────┐ │────▶│ │

│ │ │ │ MCP Server │ │ │ │

│ │ │ │ (MCP Tools for Task Operations) │ │◀────│ │

│ │ │ └────────────────────────────────────────┘ │ │ │

└─────────────────┘ └──────────────────────────────────────────────┘ └─────────────────┘

# Database Models

| **Model** | **Fields** | **Description** |
| --- | --- | --- |
| **Task** | user\_id, id, title, description, completed, created\_at, updated\_at | Todo items |
| **Conversation** | user\_id, id, created\_at, updated\_at | Chat session |
| **Message** | user\_id, id, conversation\_id, role (user/assistant), content, created\_at | Chat history |

# Chat API Endpoint

| **Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| POST | /api/{user\_id}/chat | Send message & get AI response |

## Request

| **Field** | **Type** | **Required** | **Description** |
| --- | --- | --- | --- |
| conversation\_id | integer | No | Existing conversation ID (creates new if not provided) |
| message | string | Yes | User's natural language message |

## Response

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| conversation\_id | integer | The conversation ID |
| response | string | AI assistant's response |
| tool\_calls | array | List of MCP tools invoked |

# MCP Tools Specification

The MCP server must expose the following tools for the AI agent:

## Tool: add\_task

| **Purpose** | Create a new task |
| --- | --- |
| **Parameters** | user\_id (string, required), title (string, required), description (string, optional) |
| **Returns** | task\_id, status, title |
| **Example Input** | {“user\_id”: “ziakhan”, "title": "Buy groceries", "description": "Milk, eggs, bread"} |
| **Example Output** | {"task\_id": 5, "status": "created", "title": "Buy groceries"} |

## Tool: list\_tasks

| **Purpose** | Retrieve tasks from the list |
| --- | --- |
| **Parameters** | status (string, optional: "all", "pending", "completed") |
| **Returns** | Array of task objects |
| **Example Input** | {user\_id (string, required), "status": "pending"} |
| **Example Output** | [{"id": 1, "title": "Buy groceries", "completed": false}, ...] |

## Tool: complete\_task

| **Purpose** | Mark a task as complete |
| --- | --- |
| **Parameters** | user\_id (string, required), task\_id (integer, required) |
| **Returns** | task\_id, status, title |
| **Example Input** | {“user\_id”: “ziakhan”, "task\_id": 3} |
| **Example Output** | {"task\_id": 3, "status": "completed", "title": "Call mom"} |

## Tool: delete\_task

| **Purpose** | Remove a task from the list |
| --- | --- |
| **Parameters** | user\_id (string, required), task\_id (integer, required) |
| **Returns** | task\_id, status, title |
| **Example Input** | {“user\_id”: “ziakhan”, "task\_id": 2} |
| **Example Output** | {"task\_id": 2, "status": "deleted", "title": "Old task"} |

## Tool: update\_task

| **Purpose** | Modify task title or description |
| --- | --- |
| **Parameters** | user\_id (string, required), task\_id (integer, required), title (string, optional), description (string, optional) |
| **Returns** | task\_id, status, title |
| **Example Input** | {“user\_id”: “ziakhan”, "task\_id": 1, "title": "Buy groceries and fruits"} |
| **Example Output** | {"task\_id": 1, "status": "updated", "title": "Buy groceries and fruits"} |

# Agent Behavior Specification

| **Behavior** | **Description** |
| --- | --- |
| **Task Creation** | When user mentions adding/creating/remembering something, use add\_task |
| **Task Listing** | When user asks to see/show/list tasks, use list\_tasks with appropriate filter |
| **Task Completion** | When user says done/complete/finished, use complete\_task |
| **Task Deletion** | When user says delete/remove/cancel, use delete\_task |
| **Task Update** | When user says change/update/rename, use update\_task |
| **Confirmation** | Always confirm actions with friendly response |
| **Error Handling** | Gracefully handle task not found and other errors |

# Conversation Flow (Stateless Request Cycle)

1. Receive user message
2. Fetch conversation history from database
3. Build message array for agent (history + new message)
4. Store user message in database
5. Run agent with MCP tools
6. Agent invokes appropriate MCP tool(s)
7. Store assistant response in database
8. Return response to client
9. Server holds NO state (ready for next request)

# Natural Language Commands

The chatbot should understand and respond to:

| **User Says** | **Agent Should** |
| --- | --- |
| "Add a task to buy groceries" | Call add\_task with title "Buy groceries" |
| "Show me all my tasks" | Call list\_tasks with status "all" |
| "What's pending?" | Call list\_tasks with status "pending" |
| "Mark task 3 as complete" | Call complete\_task with task\_id 3 |
| "Delete the meeting task" | Call list\_tasks first, then delete\_task |
| "Change task 1 to 'Call mom tonight'" | Call update\_task with new title |
| "I need to remember to pay bills" | Call add\_task with title "Pay bills" |
| "What have I completed?" | Call list\_tasks with status "completed" |

# Deliverables

1. GitHub repository with:

* /frontend – ChatKit-based UI
* /backend – FastAPI + Agents SDK + MCP
* /specs – Specification files for agent and MCP tools
* Database migration scripts
* README with setup instructions

1. Working chatbot that can:

* Manage tasks through natural language via MCP tools
* Maintain conversation context via database (stateless server)
* Provide helpful responses with action confirmations
* Handle errors gracefully
* Resume conversations after server restart

# OpenAI ChatKit Setup & Deployment

## Domain Allowlist Configuration (Required for Hosted ChatKit)

Before deploying your chatbot frontend, you must configure OpenAI's domain allowlist for security:

1. **Deploy your frontend first to get a production URL:**

* Vercel: `https://your-app.vercel.app`
* GitHub Pages: `https://username.github.io/repo-name`
* Custom domain: `https://yourdomain.com`

1. **Add your domain to OpenAI's allowlist:**

* Navigate to: <https://platform.openai.com/settings/organization/security/domain-allowlist>
* Click "Add domain"
* Enter your frontend URL (without trailing slash)
* Save changes

1. **Get your ChatKit domain key:**

* After adding the domain, OpenAI will provide a domain key
* Pass this key to your ChatKit configuration

## Environment Variables

NEXT\_PUBLIC\_OPENAI\_DOMAIN\_KEY=your-domain-key-here

*Note: The hosted ChatKit option only works after adding the correct domains under Security → Domain Allowlist. Local development (`localhost`) typically works without this configuration.*

# Key Architecture Benefits

| **Aspect** | **Benefit** |
| --- | --- |
| **MCP Tools** | Standardized interface for AI to interact with your app |
| **Single Endpoint** | Simpler API — AI handles routing to tools |
| **Stateless Server** | Scalable, resilient, horizontally scalable |
| **Tool Composition** | Agent can chain multiple tools in one turn |

### Key Stateless Architecture Benefits

* **Scalability:** Any server instance can handle any request
* **Resilience:** Server restarts don't lose conversation state
* **Horizontal scaling:** Load balancer can route to any backend
* **Testability:** Each request is independent and reproducible

# Phase IV: Local Kubernetes Deployment (Minikube, Helm Charts, kubectl-ai, Kagent, Docker Desktop, and Gordon)

*Cloud Native Todo Chatbot with Basic Level Functionality*

**Objective:** Deploy the Todo Chatbot on a local Kubernetes cluster using Minikube, Helm Charts.

💡**Development Approach:** Use the [Agentic Dev Stack workflow](#_32bnoy5itcj): Write spec → Generate plan → Break into tasks → Implement via Claude Code. No manual coding allowed. We will review the process, prompts, and iterations to judge each phase and project.

## Requirements

* Containerize frontend and backend applications (Use Gordon)
* Use Docker AI Agent (Gordon) for AI-assisted Docker operations
* Create Helm charts for deployment (Use kubectl-ai and/or kagent to generate)
* Use kubectl-ai and kagent for AI-assisted Kubernetes operations
* Deploy on Minikube locally

*Note: If Docker AI (Gordon) is unavailable in your region or tier, use standard Docker CLI commands or ask Claude Code to generate the docker run commands for you.*

## Technology Stack

| **Component** | **Technology** |
| --- | --- |
| Containerization | Docker (Docker Desktop) |
| Docker AI | Docker AI Agent (Gordon) |
| Orchestration | Kubernetes (Minikube) |
| Package Manager | Helm Charts |
| AI DevOps | kubectl-ai, and Kagent |
| Application | Phase III Todo Chatbot |

## AIOps

Use [Docker AI Agent (Gordon)](https://docs.docker.com/ai/gordon/) for intelligent Docker operations:

# To know its capabilities

docker ai "What can you do?"

Enable Gordon: Install latest Docker Desktop 4.53+, go to Settings > Beta features, and toggle it on.

Use [kubectl-ai](https://github.com/GoogleCloudPlatform/kubectl-ai), and [Kagent](https://github.com/kagent-dev/kagent) for intelligent Kubernetes operations:

# Using kubectl-ai

kubectl-ai "deploy the todo frontend with 2 replicas"

kubectl-ai "scale the backend to handle more load"

kubectl-ai "check why the pods are failing"

# Using kagent

kagent "analyze the cluster health"

kagent "optimize resource allocation"

Starting with kubectl-ai will make you feel empowered from day one. Layer in Kagent for advanced use cases. Pair them with Minikube for zero-cost learning and work.

**Research Note: Using Blueprints for Spec-Driven Deployment**

Can Spec-Driven Development be used for infrastructure automation, and how we may need to use blueprints powered by Claude Code Agent Skills.

1. [Is Spec-Driven Development Key for Infrastructure Automation?](https://thenewstack.io/is-spec-driven-development-key-for-infrastructure-automation/)
2. [ChatGPT Progressive Learning Conversation](https://chatgpt.com/share/6924914a-43dc-8001-8f67-af29c4d9617e)
3. [Spec-Driven Cloud-Native Architecture: Governing AI Agents for Managed Services with Claude Code and SpecKit](https://claude.ai/public/artifacts/6025a232-6ebe-4c42-bb51-02dbd4603e18)

# Phase V: Advanced Cloud Deployment

*Advanced Level Functionality on Azure (AKS) or Google Cloud (GKE) or Azure (AKS)*

**Objective:** Implement advanced features and deploy first on Minikube locally and then to production-grade Kubernetes on Azure/Google Cloud/Oracle and Kafka within Kubernetes Cluster or with a managed service like Redpanda Cloud.

💡**Development Approach:** Use the [Agentic Dev Stack workflow](#_32bnoy5itcj): Write spec → Generate plan → Break into tasks → Implement via Claude Code. No manual coding allowed. We will review the process, prompts, and iterations to judge each phase and project.

## Part A: Advanced Features

* Implement all Advanced Level features (Recurring Tasks, Due Dates & Reminders)
* Implement Intermediate Level features (Priorities, Tags, Search, Filter, Sort)
* Add event-driven architecture with Kafka
* Implement Dapr for distributed application runtime

## Part B: Local Deployment

* Deploy to Minikube
* Deploy Dapr on Minikube use Full Dapr: Pub/Sub, State, Bindings (cron), Secrets, Service Invocation

## Part C: Cloud Deployment

* Deploy to Azure (AKS)/Google Cloud (GKE)
* Deploy Dapr on GKE/AKS use Full Dapr: Pub/Sub, State, Bindings (cron), Secrets, Service Invocation
* Use Kafka on Confluent/Redpanda Cloud. If you have any trouble with kafka access you can add any other PubSub Component with Dapr.
* Set up CI/CD pipeline using Github Actions
* Configure monitoring and logging

## Microsoft Azure Setup (AKS)

**US$200 credits for 30 days, plus 12 months of selected free services:**

Sign up at <https://azure.microsoft.com/en-us/free/.%22>?

1. Create a Kubernetes cluster
2. Configure kubectl to connect with Cluster
3. Deploy using Helm charts from Phase IV

## Oracle Cloud Setup (Recommended - Always Free)

Sign up at https://www.oracle.com/cloud/free/

- Create OKE cluster (4 OCPUs, 24GB RAM - always free)

- No credit card charge after trial

- Best for learning without time pressure

## Google Cloud Setup (GKE)

**US$300 credits, usable for 90 days for new customers:**

Sign up at <https://cloud.google.com/free?hl=en>

Kafka Use Cases in Phase

**Event-Driven Architecture for Todo Chatbot**

# 1. Reminder/Notification System

┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐

│ │ │ │ │ │ │ │

│ Todo Service │────▶│ Kafka Topic │────▶│ Notification │────▶│ User Device │

│ (Producer) │ │ "reminders" │ │ Service │ │ (Push/Email) │

│ │ │ │ │ (Consumer) │ │ │

└─────────────────┘ └─────────────────┘ └─────────────────┘ └─────────────────┘

When a task with a due date is created, publish a reminder event. A separate notification service consumes and sends reminders at the right time.

# 2. Recurring Task Engine

┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐

│ │ │ │ │ │

│ Task Completed │────▶│ Kafka Topic │────▶│ Recurring Task │

│ Event │ │ "task-events" │ │ Service │

│ │ │ │ │ (Creates next) │

└─────────────────┘ └─────────────────┘ └─────────────────┘

When a recurring task is marked complete, publish an event. A separate service consumes it and auto-creates the next occurrence.

# 3. Activity/Audit Log

┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐

│ │ │ │ │ │

│ All Task │────▶│ Kafka Topic │────▶│ Audit Service │

│ Operations │ │ "task-events" │ │ (Stores log) │

│ │ │ │ │ │

└─────────────────┘ └─────────────────┘ └─────────────────┘

Every task operation (create, update, delete, complete) publishes to Kafka. An audit service consumes and maintains a complete history.

# 4. Real-time Sync Across Clients

┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐

│ │ │ │ │ │ │ │

│ Task Changed │────▶│ Kafka Topic │────▶│ WebSocket │────▶│ All Connected │

│ (Any Client) │ │ "task-updates" │ │ Service │ │ Clients │

│ │ │ │ │ │ │ │

└─────────────────┘ └─────────────────┘ └─────────────────┘ └─────────────────┘

Changes from one client are broadcast to all connected clients in real-time.

# Recommended Architecture

┌──────────────────────────────────────────────────────────────────────────────────────┐

│ KUBERNETES CLUSTER │

│ │

│ ┌─────────────┐ ┌─────────────┐ ┌─────────────────────────────────────────────┐ │

│ │ Frontend │ │ Chat API │ │ KAFKA CLUSTER │ │

│ │ Service │──▶│ + MCP │──▶│ ┌─────────────┐ ┌─────────────────────┐ │ │

│ └─────────────┘ │ Tools │ │ │ task-events │ │ reminders │ │ │

│ └──────┬──────┘ │ └─────────────┘ └─────────────────────┘ │ │

│ │ └──────────┬────────────────────┬────────────┘ │

│ │ │ │ │

│ ▼ ▼ ▼ │

│ ┌─────────────┐ ┌─────────────────┐ ┌─────────────────┐ │

│ │ Neon DB │ │ Recurring Task │ │ Notification │ │

│ │ (External) │ │ Service │ │ Service │ │

│ └─────────────┘ └─────────────────┘ └─────────────────┘ │

└──────────────────────────────────────────────────────────────────────────────────────┘

# Kafka Topics

| **Topic** | **Producer** | **Consumer** | **Purpose** |
| --- | --- | --- | --- |
| **task-events** | Chat API (MCP Tools) | Recurring Task Service, Audit Service | All task CRUD operations |
| **reminders** | Chat API (when due date set) | Notification Service | Scheduled reminder triggers |
| **task-updates** | Chat API | WebSocket Service | Real-time client sync |

# Event Schema Examples

## Task Event

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| event\_type | string | "created", "updated", "completed", "deleted" |
| task\_id | integer | The task ID |
| task\_data | object | Full task object |
| user\_id | string | User who performed action |
| timestamp | datetime | When event occurred |

## Reminder Event

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| task\_id | integer | The task ID |
| title | string | Task title for notification |
| due\_at | datetime | When task is due |
| remind\_at | datetime | When to send reminder |
| user\_id | string | User to notify |

# Why Kafka for Todo App?

| **Without Kafka** | **With Kafka** |
| --- | --- |
| Reminder logic coupled with main app | Decoupled notification service |
| Recurring tasks processed synchronously | Async processing, no blocking |
| No activity history | Complete audit trail |
| Single client updates | Real-time multi-client sync |
| Tight coupling between services | Loose coupling, scalable |

# Bottom Line

Kafka turns the Todo app from a simple CRUD app into an **event-driven system** where services communicate through events rather than direct API calls. This is essential for the advanced features (recurring tasks, reminders) and scales better in production.

**Key Takeaway:**

Kafka enables decoupled, scalable microservices architecture where the Chat API publishes events and specialized services (Notification, Recurring Task, Audit) consume and process them independently.

Kafka Service Recommendations

# For Cloud Deployment

| **Service** | **Free Tier** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Redpanda Cloud ⭐** | Free Serverless tier | Kafka-compatible, no Zookeeper, fast, easy setup | Newer ecosystem |
| Confluent Cloud | $400 credit for 30 days | Industry standard, Schema Registry, great docs | Credit expires |
| CloudKarafka | "Developer Duck" free plan | Simple, 5 topics free | Limited throughput |
| Aiven | $300 credit trial | Fully managed, multi-cloud | Trial expires |
| Self-hosted (Strimzi) | Free (just compute cost) | Full control, learning experience | More complex setup |

# For Local Development (Minikube)

| **Option** | **Complexity** | **Description** |
| --- | --- | --- |
| **Redpanda (Docker) ⭐** | Easy | Single binary, no Zookeeper, Kafka-compatible |
| Bitnami Kafka Helm | Medium | Kubernetes-native, Helm chart |
| Strimzi Operator | Medium-Hard | Production-grade K8s operator |

# Primary Recommendation: Self-Hosted Kafka in Kubernetes

You can deploy Kafka directly within your K8s cluster using the Strimzi operator. Best for hackathon because:

* Free cost
* Dapr PubSub makes Kafka-swappable - same APIs, clients work unchanged
* No Zookeeper - simpler architecture
* Fast setup - under 5 minutes
* REST API + Native protocols

# Self-Hosted on Kubernetes (Strimzi)

Good learning experience for students:

# Install Strimzi operator

kubectl create namespace kafka

kubectl apply -f https://strimzi.io/install/latest?namespace=kafka

# kafka-cluster.yaml

apiVersion: kafka.strimzi.io/v1beta2

kind: Kafka

metadata:

name: taskflow-kafka

namespace: kafka

spec:

kafka:

replicas: 1

listeners:

- name: plain

port: 9092

type: internal

storage:

type: ephemeral

zookeeper:

replicas: 1

storage:

type: ephemeral

# Create Kafka cluster

kubectl apply -f kafka-cluster.yaml

# Redpanda Cloud Quick Setup

| **Step** | **Action** |
| --- | --- |
| 1 | Sign up at redpanda.com/cloud |
| 2 | Create a Serverless cluster (free tier) |
| 3 | Create topics: task-events, reminders, task-updates |
| 4 | Copy bootstrap server URL and credentials |
| 5 | Use standard Kafka clients (kafka-python, aiokafka) |

# Python Client Example

Standard kafka-python works with Redpanda:

from kafka import KafkaProducer

import json

producer = KafkaProducer(

bootstrap\_servers="YOUR-CLUSTER.cloud.redpanda.com:9092",

security\_protocol="SASL\_SSL",

sasl\_mechanism="SCRAM-SHA-256",

sasl\_plain\_username="YOUR-USERNAME",

sasl\_plain\_password="YOUR-PASSWORD",

value\_serializer=lambda v: json.dumps(v).encode('utf-8')

)

# Publish event

producer.send("task-events", {"event\_type": "created", "task\_id": 1})

# Summary for Hackathon

| **Type** | **Recommendation** |
| --- | --- |
| **Local: Minikube** | Redpanda Docker container |
| **Cloud** | Redpanda Cloud Serverless (free) or Strimzi self-hosted |

Dapr Integration Guide

# What is Dapr?

**Dapr (Distributed Application Runtime)** is a portable, event-driven runtime that simplifies building microservices. It runs as a **sidecar** next to your application and provides building blocks via HTTP/gRPC APIs.

# Dapr Building Blocks for Todo App

| **Building Block** | **Use Case in Todo App** |
| --- | --- |
| **Pub/Sub** | Kafka abstraction – publish/subscribe without Kafka client code |
| **State Management** | Conversation state storage (alternative to direct DB calls) |
| **Service Invocation** | Frontend → Backend communication with built-in retries |
| **Bindings** | Cron triggers for scheduled reminders |
| **Secrets Management** | Store API keys, DB credentials securely |

# Architecture: Without Dapr vs With Dapr

## Without Dapr (Direct Dependencies)

┌─────────────┐ ┌─────────────┐ ┌─────────────┐

│ Frontend │────▶│ Backend │────▶│ Kafka │

│ │ │ (FastAPI) │────▶│ Neon DB │

└─────────────┘ └─────────────┘ └─────────────┘

│

Tight coupling:

- kafka-python library

- psycopg2/sqlmodel

- Direct connection strings

## With Dapr (Abstracted Dependencies)

┌─────────────┐ ┌─────────────────────────────────┐ ┌─────────────┐

│ Frontend │ │ Backend Pod │ │ │

│ + Dapr │────▶│ ┌─────────┐ ┌───────────┐ │ │ Dapr │

│ Sidecar │ │ │ FastAPI │◀──▶│ Dapr │──┼────▶│ Components │

└─────────────┘ │ │ App │ │ Sidecar │ │ │ - Kafka │

│ └─────────┘ └───────────┘ │ │ - Neon DB │

└─────────────────────────────────┘ │ - Secrets │

└─────────────┘

Loose coupling:

- App talks to Dapr via HTTP

- Dapr handles Kafka, DB, etc.

- Swap components without code changes

# Use Case 1: Pub/Sub (Kafka Abstraction)

Instead of using kafka-python directly, publish events via Dapr:

**Without Dapr:**

from kafka import KafkaProducer

producer = KafkaProducer(bootstrap\_servers="kafka:9092", ...)

producer.send("task-events", value=event)

**With Dapr:**

import httpx

# Publish via Dapr sidecar (no Kafka library needed!)

await httpx.post(

"http://localhost:3500/v1.0/publish/kafka-pubsub/task-events",

json={"event\_type": "created", "task\_id": 1}

)

**Dapr Component Configuration:**

apiVersion: dapr.io/v1alpha1

kind: Component

metadata:

name: kafka-pubsub

spec:

type: pubsub.kafka

version: v1

metadata:

- name: brokers

value: "kafka:9092"

- name: consumerGroup

value: "todo-service"

# Use Case 2: State Management (Conversation State)

Store conversation history without direct DB code:

**Without Dapr:**

from sqlmodel import Session

session.add(Message(...))

session.commit()

**With Dapr:**

import httpx

# Save state via Dapr

await httpx.post(

"http://localhost:3500/v1.0/state/statestore",

json=[{

"key": f"conversation-{conv\_id}",

"value": {"messages": messages}

}]

)

# Get state

response = await httpx.get(

f"http://localhost:3500/v1.0/state/statestore/conversation-{conv\_id}"

)

**Dapr Component Configuration:**

apiVersion: dapr.io/v1alpha1

kind: Component

metadata:

name: statestore

spec:

type: state.postgresql

version: v1

metadata:

- name: connectionString

value: "host=neon.db user=... password=... dbname=todo"

# Use Case 3: Service Invocation (Frontend → Backend)

Built-in service discovery, retries, and mTLS:

**Without Dapr:**

// Frontend must know backend URL

fetch("http://backend-service:8000/api/chat", {...})

**With Dapr:**

// Frontend calls via Dapr sidecar – automatic discovery

fetch("http://localhost:3500/v1.0/invoke/backend-service/method/api/chat", {...})

# Use Case 4: Dapr Jobs API (Scheduled Reminders)

Why Jobs API over Cron Bindings?

* Cron Bindings | Poll every X minutes, check DB
* Dapr Jobs API | Schedule exact time, callback fires

Schedule a reminder at exact time:

```python

import httpx

async def schedule\_reminder(task\_id: int, remind\_at: datetime, user\_id: str):

"""Schedule reminder using Dapr Jobs API (not cron polling)."""

await httpx.post(

f"http://localhost:3500/v1.0-alpha1/jobs/reminder-task-{task\_id}",

json={

"dueTime": remind\_at.strftime("%Y-%m-%dT%H:%M:%SZ"),

"data": {

"task\_id": task\_id,

"user\_id": user\_id,

"type": "reminder"

}

}

)

Handle callback when job fires:

@app.post("/api/jobs/trigger")

async def handle\_job\_trigger(request: Request):

"""Dapr calls this endpoint at the exact scheduled time."""

job\_data = await request.json()

if job\_data["data"]["type"] == "reminder":

# Publish to notification service via Dapr PubSub

await publish\_event("reminders", "reminder.due", job\_data["data"])

return {"status": "SUCCESS"}

Benefits:

* No polling overhead
* Exact timing (not "within 5 minutes")
* Scales better (no DB scans every minute)
* Same pattern works for recurring task spawns

# Use Case 5: Secrets Management

Securely store and access credentials (Optionally you can use Kubernetes Secrets):

* K8s Secrets directly: Simple, already on K8s, fewer moving parts
* Dapr Secrets API: Multi-cloud portability, unified API across providers

Dapr Secrets becomes valuable when targeting multipleplatforms (K8s + Azure + AWS).

**Dapr Component (Kubernetes Secrets):**

apiVersion: dapr.io/v1alpha1

kind: Component

metadata:

name: kubernetes-secrets

spec:

type: secretstores.kubernetes

version: v1

**Access in App:**

import httpx

response = await httpx.get(

"http://localhost:3500/v1.0/secrets/kubernetes-secrets/openai-api-key"

)

api\_key = response.json()["openai-api-key"]

# Complete Dapr Architecture

┌──────────────────────────────────────────────────────────────────────────────────────┐

│ KUBERNETES CLUSTER │

│ │

│ ┌─────────────────────┐ ┌─────────────────────┐ ┌─────────────────────┐ │

│ │ Frontend Pod │ │ Backend Pod │ │ Notification Pod │ │

│ │ ┌───────┐ ┌───────┐ │ │ ┌───────┐ ┌───────┐ │ │ ┌───────┐ ┌───────┐ │ │

│ │ │ Next │ │ Dapr │ │ │ │FastAPI│ │ Dapr │ │ │ │Notif │ │ Dapr │ │ │

│ │ │ App │◀┼▶Sidecar│ │ │ │+ MCP │◀┼▶Sidecar│ │ │ │Service│◀┼▶Sidecar│ │ │

│ │ └───────┘ └───────┘ │ │ └───────┘ └───────┘ │ │ └───────┘ └───────┘ │ │

│ └──────────┬──────────┘ └──────────┬──────────┘ └──────────┬──────────┘ │

│ │ │ │ │

│ └─────────────────────────┼─────────────────────────┘ │

│ │ │

│ ┌────────────▼────────────┐ │

│ │ DAPR COMPONENTS │ │

│ │ ┌──────────────────┐ │ │

│ │ │ pubsub.kafka │───┼────▶ Cluster Kafka │

│ │ ├──────────────────┤ │ │

│ │ │ state.postgresql │───┼────▶ Neon DB │

│ │ ├──────────────────┤ │ │

│ │ │ scheduler │ │ (Scheduled triggers) │

│ │ ├──────────────────┤ │ │

│ │ │ secretstores.k8s │ │ (API keys, credentials) │

│ │ └──────────────────┘ │ │

│ └─────────────────────────┘ │

└──────────────────────────────────────────────────────────────────────────────────────┘

# Dapr Components Summary

| **Component** | **Type** | **Purpose** |
| --- | --- | --- |
| **kafka-pubsub** | pubsub.kafka | Event streaming (task-events, reminders) |
| **statestore** | state.postgresql | Conversation state, task cache |
| **dapr-jobs** | Jobs API | Trigger reminder checks |
| **kubernetes-secrets** | secretstores.kubernetes | API keys, DB credentials |

# Why Use Dapr?

| **Without Dapr** | **With Dapr** |
| --- | --- |
| Import Kafka, Redis, Postgres libraries | Single HTTP API for all |
| Connection strings in code | Dapr components (YAML config) |
| Manual retry logic | Built-in retries, circuit breakers |
| Service URLs hardcoded | Automatic service discovery |
| Secrets in env vars | Secure secret store integration |
| Vendor lock-in | Swap Kafka for RabbitMQ with config change |

# Local vs Cloud Dapr Usage

| **Phase** | **Dapr Usage** |
| --- | --- |
| **Local (Minikube)** | Install Dapr, use Pub/Sub for Kafka, basic state management |
| **Cloud (DigitalOcean)** | Full Dapr: Pub/Sub, State, Bindings (cron), Secrets, Service Invocation |

# Getting Started with Dapr

# Install Dapr CLI

curl -fsSL https://raw.githubusercontent.com/dapr/cli/master/install/install.sh | bash

# Initialize Dapr on Kubernetes

dapr init -k

# Deploy components

kubectl apply -f dapr-components/

# Run app with Dapr sidecar

dapr run --app-id backend --app-port 8000 -- uvicorn main:app

# Bottom Line

Dapr abstracts infrastructure (Kafka, DB, Secrets) behind simple HTTP APIs. Your app code stays clean, and you can swap backends (e.g., Kafka → RabbitMQ) by changing YAML config, not code.

# Submission Requirements

## Required Submissions

1. Public GitHub Repository containing:

* All source code for all completed phases
* /specs folder with all specification files
* CLAUDE.md with Claude Code instructions
* README.md with comprehensive documentation
* Clear folder structure for each phase

1. Deployed Application Links:

* Phase II: Vercel/frontend URL + Backend API URL
* Phase III-V: Chatbot URL
* Phase IV: Instructions for local Minikube setup
* Phase V: DigitalOcean deployment URL

1. Demo Video (maximum 90 seconds):

* Demonstrate all implemented features
* Show spec-driven development workflow
* Judges will only watch the first 90 seconds

1. WhatsApp Number for presentation invitation

# Resources

## Core Tools

| **Tool** | **Link** | **Description** |
| --- | --- | --- |
| Claude Code | claude.com/product/claude-code | AI coding assistant |
| GitHub Spec-Kit | github.com/panaversity/spec-kit-plus | Specification management |
| OpenAI ChatKit | platform.openai.com/docs/guides/chatkit | Chatbot UI framework |
| MCP | github.com/modelcontextprotocol/python-sdk | MCP server framework |

## Infrastructure

| **Service** | **Link** | **Notes** |
| --- | --- | --- |
| Neon DB | neon.tech | Free tier available |
| Vercel | vercel.com | Free frontend hosting |
| DigitalOcean | digitalocean.com | $200 credit for 60 days |
| Minikube | minikube.sigs.k8s.io | Local Kubernetes |

# Frequently Asked Questions

**Q: Can I skip phases?**

A: No, each phase builds on the previous. You must complete them in order.

**Q: Can I use different technologies?**

A: The core stack must remain as specified. You can add additional tools/libraries.

**Q: Do I need a DigitalOcean account from the start?**

A: No, only for Phase V. Use the $200 free credit for new accounts.

**Q: Can I work in a team?**

A: This is an individual hackathon. Each participant submits separately.

**Q: What if I don't complete all the phases?**

A: Submit what you complete. Partial submissions are evaluated proportionally.

# The Agentic Dev Stack: AGENTS.md + Spec-KitPlus + Claude Code

This is a powerful integration. By combining the **declarative** nature of AGENTS.md, the **structured workflow** of Panaversity Spec-KitPlus, and the **agentic execution** of Claude Code, you move from "vibe-coding" to a professional, spec-driven engineering pipeline.

This section outlines a workflow where AGENTS.md acts as the **Constitution**, Spec-KitPlus acts as the **Architect**, and Claude Code acts as the **Builder**.

## 1. The Mental Model: Who Does What?

| **Component** | **Role** | **Responsibility** |
| --- | --- | --- |
| **AGENTS.md** | **The Brain** | Cross-agent truth. Defines *how* agents should behave, what tools to use, and coding standards. |
| **Spec-KitPlus** | **The Architect** | Manages spec artifacts (.specify, .plan, .tasks). Ensures technical rigor before coding starts. |
| **Claude Code** | **The Executor** | The agentic environment. Reads the project memory and executes Spec-Kit tools via MCP. |

**Key Idea:** Claude reads AGENTS.md via a tiny CLAUDE.md shim and interacts with Spec-KitPlus. For development setup an MCP Server and upgrade specifyplus commands to be available as Prompts in MCP. SpecKitPlus MCP server ensures every line of code maps back to a validated task.

## 2. Step 1: Initialize Spec-KitPlus

First, scaffold the spec-driven structure in your project root. This ensures the agent has the necessary templates to create structured plans.

uv specifyplus init <project\_name>

**This enables the core pipeline:**

* /specify -> Captures requirements in speckit.specify.
* /plan -> Generates the technical approach in speckit.plan.
* /tasks -> Breaks the plan into actionable speckit.tasks.
* /implement -> Executes the code changes.

## 3. Step 2: Create a Spec-Aware AGENTS.md

Create AGENTS.md in your root. This file teaches all AI agents (Claude, Copilot, Gemini) how to use your specific Spec-Kit workflow.

```markdown

# AGENTS.md  
Here is a \*\*significantly improved, clearer, more actionable, more valuable\*\* version of your \*\*AGENTS.md\*\*.

I kept the spirit but made it \*practical\*, \*strict\*, and \*agent-compatible\*, so Claude/Gemini/Copilot can actually follow it in real workflows.

---

# \*\*AGENTS.md\*\*

## \*\*Purpose\*\*

This project uses \*\*Spec-Driven Development (SDD)\*\* — a workflow where \*\*no agent is allowed to write code until the specification is complete and approved\*\*.

All AI agents (Claude, Copilot, Gemini, local LLMs, etc.) must follow the \*\*Spec-Kit lifecycle\*\*:

> \*\*Specify → Plan → Tasks → Implement\*\*

This prevents “vibe coding,” ensures alignment across agents, and guarantees that every implementation step maps back to an explicit requirement.

---

## \*\*How Agents Must Work\*\*

Every agent in this project MUST obey these rules:

1. \*\*Never generate code without a referenced Task ID.\*\*

2. \*\*Never modify architecture without updating `speckit.plan`.\*\*

3. \*\*Never propose features without updating `speckit.specify` (WHAT).\*\*

4. \*\*Never change approach without updating `speckit.constitution` (Principles).\*\*

5. \*\*Every code file must contain a comment linking it to the Task and Spec sections.\*\*

If an agent cannot find the required spec, it must \*\*stop and request it\*\*, not improvise.

---

## \*\*Spec-Kit Workflow (Source of Truth)\*\*

### \*\*1. Constitution (WHY — Principles & Constraints)\*\*

File: `speckit.constitution`

Defines the project’s non-negotiables: architecture values, security rules, tech stack constraints, performance expectations, and patterns allowed.

Agents must check this before proposing solutions.

---

### \*\*2. Specify (WHAT — Requirements, Journeys & Acceptance Criteria)\*\*

File: `speckit.specify`

Contains:

\* User journeys

\* Requirements

\* Acceptance criteria

\* Domain rules

\* Business constraints

Agents must not infer missing requirements — they must request clarification or propose specification updates.

---

### \*\*3. Plan (HOW — Architecture, Components, Interfaces)\*\*

File: `speckit.plan`

Includes:

\* Component breakdown

\* APIs & schema diagrams

\* Service boundaries

\* System responsibilities

\* High-level sequencing

All architectural output MUST be generated from the Specify file.

---

### \*\*4. Tasks (BREAKDOWN — Atomic, Testable Work Units)\*\*

File: `speckit.tasks`

Each Task must contain:

\* Task ID

\* Clear description

\* Preconditions

\* Expected outputs

\* Artifacts to modify

\* Links back to Specify + Plan sections

Agents \*\*implement only what these tasks define\*\*.

---

### \*\*5. Implement (CODE — Write Only What the Tasks Authorize)\*\*

Agents now write code, but must:

\* Reference Task IDs

\* Follow the Plan exactly

\* Not invent new features or flows

\* Stop and request clarification if anything is underspecified

> The golden rule: \*\*No task = No code.\*\*

---

## \*\*Agent Behavior in This Project\*\*

### \*\*When generating code:\*\*

Agents must reference:

```

[Task]: T-001

[From]: speckit.specify §2.1, speckit.plan §3.4

```

### \*\*When proposing architecture:\*\*

Agents must reference:

```

Update required in speckit.plan → add component X

```

### \*\*When proposing new behavior or a new feature:\*\*

Agents must reference:

```

Requires update in speckit.specify (WHAT)

```

### \*\*When changing principles:\*\*

Agents must reference:

```

Modify constitution.md → Principle #X

```

---

## \*\*Agent Failure Modes (What Agents MUST Avoid)\*\*

Agents are NOT allowed to:

\* Freestyle code or architecture

\* Generate missing requirements

\* Create tasks on their own

\* Alter stack choices without justification

\* Add endpoints, fields, or flows that aren’t in the spec

\* Ignore acceptance criteria

\* Produce “creative” implementations that violate the plan

If a conflict arises between spec files, the \*\*Constitution > Specify > Plan > Tasks\*\* hierarchy applies.

---

## \*\*Developer–Agent Alignment\*\*

Humans and agents collaborate, but the \*\*spec is the single source of truth\*\*.

Before every session, agents should re-read:

1. `.memory/constitution.md`

This ensures predictable, deterministic development.

```

## 4. Step 3: Wire Spec-KitPlus into Claude via MCP

To let Claude Code actually *run* Spec-KitPlus commands, you will set up an MCP server with prompts present in .claude/commands. Each command here will become a prompt in the MCP server.

### 4.1 Install SpecKitPlus, Create an MCP Server

1. uv init specifyplus <project\_name>
2. Create your Consitution
3. Add Anthropic's official MCP Builder Skill
4. Using SDD Loop (Specify, Plan, Tasks, Implement) you will set up an MCP server with prompts present in .claude/commands
5. Use these as part of your prompt instructions in specify: `We have specifyplus commands on @.claude/commands/\*\* Each command takes user input and updates its prompt variable before sending it to the agent. Now you will use your mcp builder skill and create an mcp server where these commands are available as prompts. Goal: Now we can run this MCP server and connect with any agent and IDE.
6. Test the MCP server

### 4.2 Register with Claude Code

Add the server to your Claude Code config (usually .mcp.json at your project root):

{  
 "mcpServers": {  
 "spec-kit": {  
 "command": "spec-kitplus-mcp",  
 "args": [],  
 "env": {}  
 }  
 }  
}

**Success:**

* After running MCP Server and connecting it with Claude Code now you can have the same commands available as MCP prompts.

## 

## 5. Step 4: Connect Claude Code via the "Shim"

Copy the default [CLAUDE.md](http://claude.md) file and integrate the content within AGENTS.md . Claude Code automatically looks for CLAUDE.md. To keep a single source of truth, use a redirection pattern.

**Create CLAUDE.md in your root:**

**```markdown**

@AGENTS.md  
**```**

*This "forwarding" ensures Claude Code loads your comprehensive agent instructions into its context window immediately upon startup.*

## 

## 6. Step 5: The Day-to-Day Workflow

Once configured, your interaction with Claude Code looks like this:

* **Context Loading:** You start Claude Code. It reads CLAUDE.md -> AGENTS.md and realizes it must use Spec-Kit.
* **Spec Generation:**
  + *User:* "I need a project dashboard."
  + *Claude:* Calls speckit\_specify and speckit\_plan using the MCP.
* **Task Breakdown:**
  + *Claude:* Calls speckit\_tasks to create a checklist in speckit.tasks.
* **Implementation:**
  + *User:* "Execute the first two tasks."
  + *Claude:* Calls speckit\_implement, writes the code, and checks it against the speckit.constitution.

## 7. Constitution vs. AGENTS.md: The Difference

It is important not to duplicate information.

* **AGENTS.md (The "How"):** Focuses on the **interaction**. "Use these tools, follow this order, use these CLI commands."
* **speckit.constitution (The "What"):** Focuses on **standards**. "We prioritize performance over brevity, we use async/await, we require 90% test coverage."

## Summary of Integration

1. **Initialize:** specify init creates the structure.
2. **Instruct:** AGENTS.md defines the rules.
3. **Bridge:** CLAUDE.md (@AGENTS.md) connects the agent.
4. **Empower:** MCP gives the agent the tools to execute.

**Good luck, and may your specs be clear and your code be clean! 🚀**

*— The Panaversity, PIAIC, and GIAIC Teams*