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: 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)

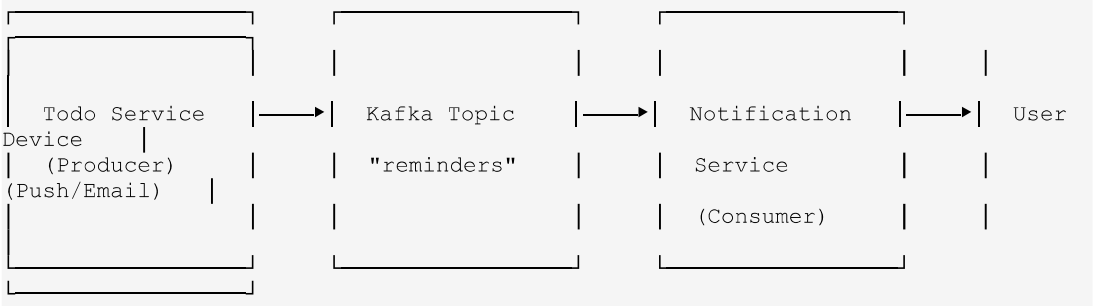
US$300 credits, usable for 90 days for new customers:

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

Kafka Use Cases in Phase

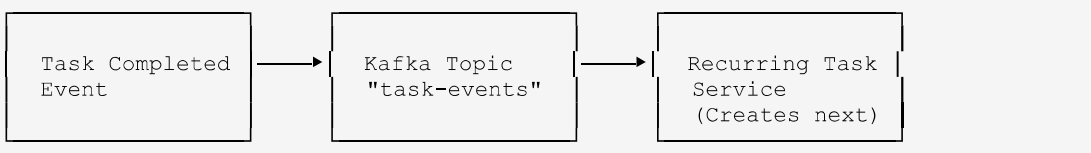
Event-Driven Architecture for Todo Chatbot

1. Reminder/Notification System



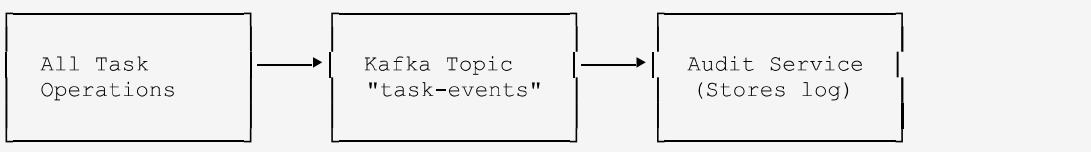
When a task with a due date is created, publish a reminder event. A separate notification service consumes and sends reminders at the right time.

2. Recurring Task Engine



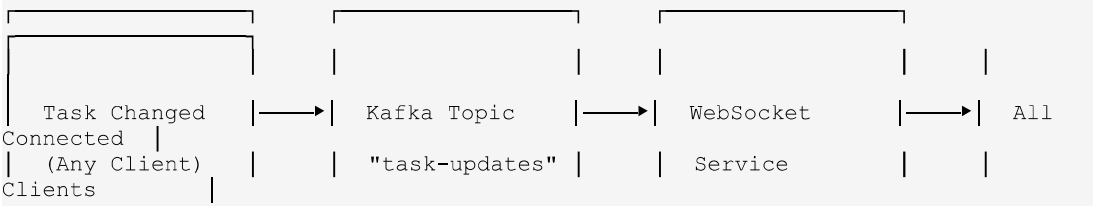
When a recurring task is marked complete, publish an event. A separate service consumes it and auto-creates the next occurrence.

3. Activity/Audit Log



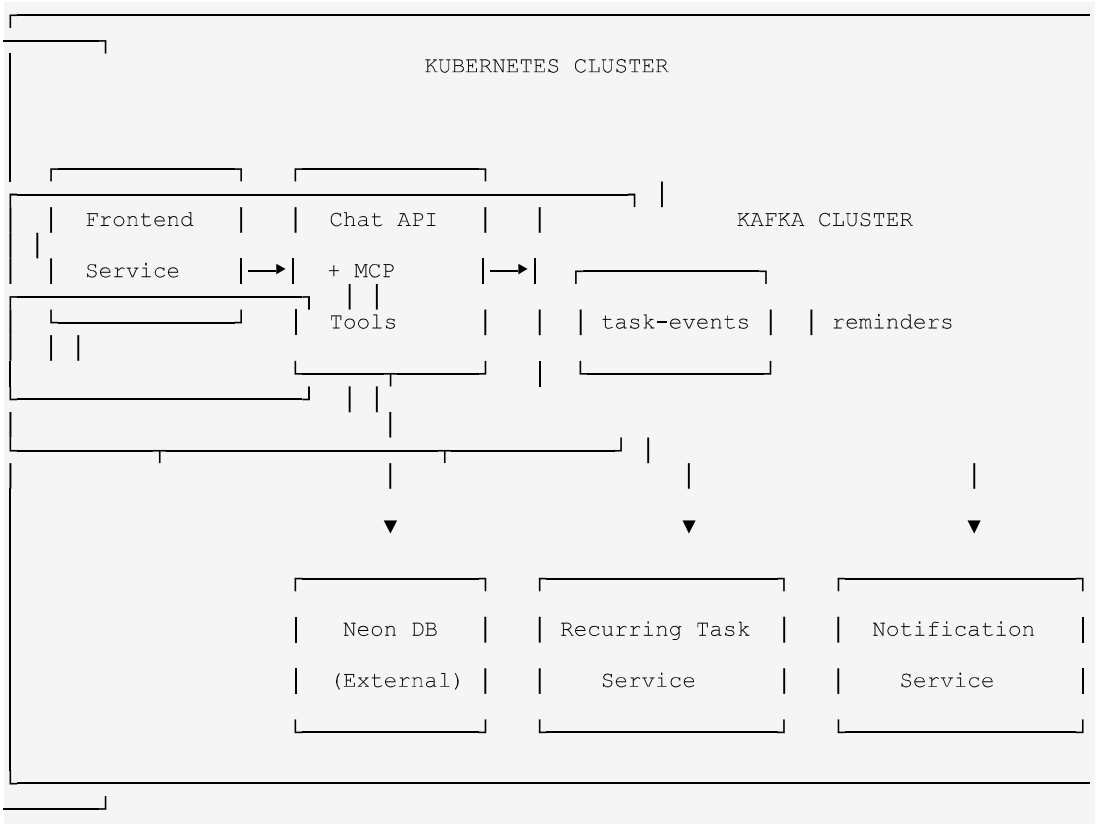
Every task operation (create, update, delete, complete) publishes to Kafka. An audit service consumes and maintains a complete history.

4. Real-time Sync Across Clients



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

Recommended Architecture



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

Field	Type	Description
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?

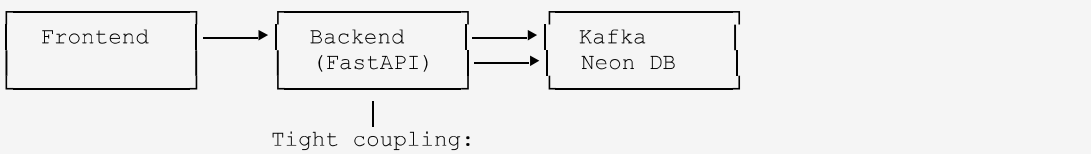
Dapr (Distributed Application Runtime) is a portable, event-driven runtime that simplifies building microservices. It runs as a **sidecar** next to your application and provides building blocks via HTTP/gRPC APIs.

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

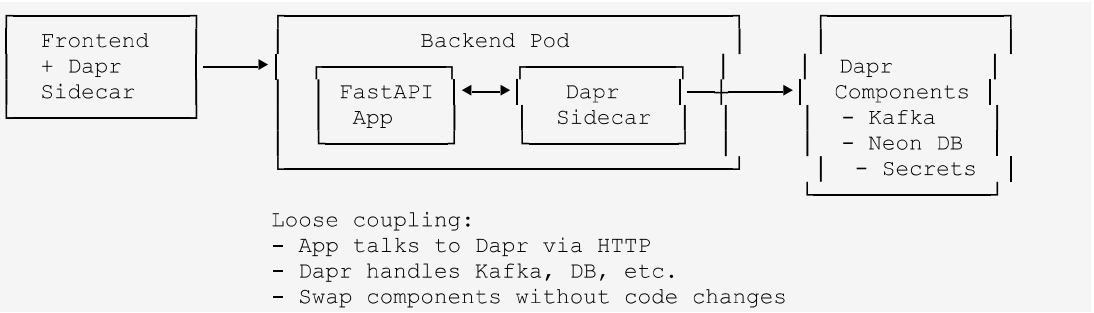
Architecture: Without Dapr vs With Dapr

Without Dapr (Direct Dependencies)



```
                    - kafka-python library
                    - psycopg2/sqlmodel
                    - Direct connection strings
```

With Dapr (Abstracted Dependencies)




```
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):

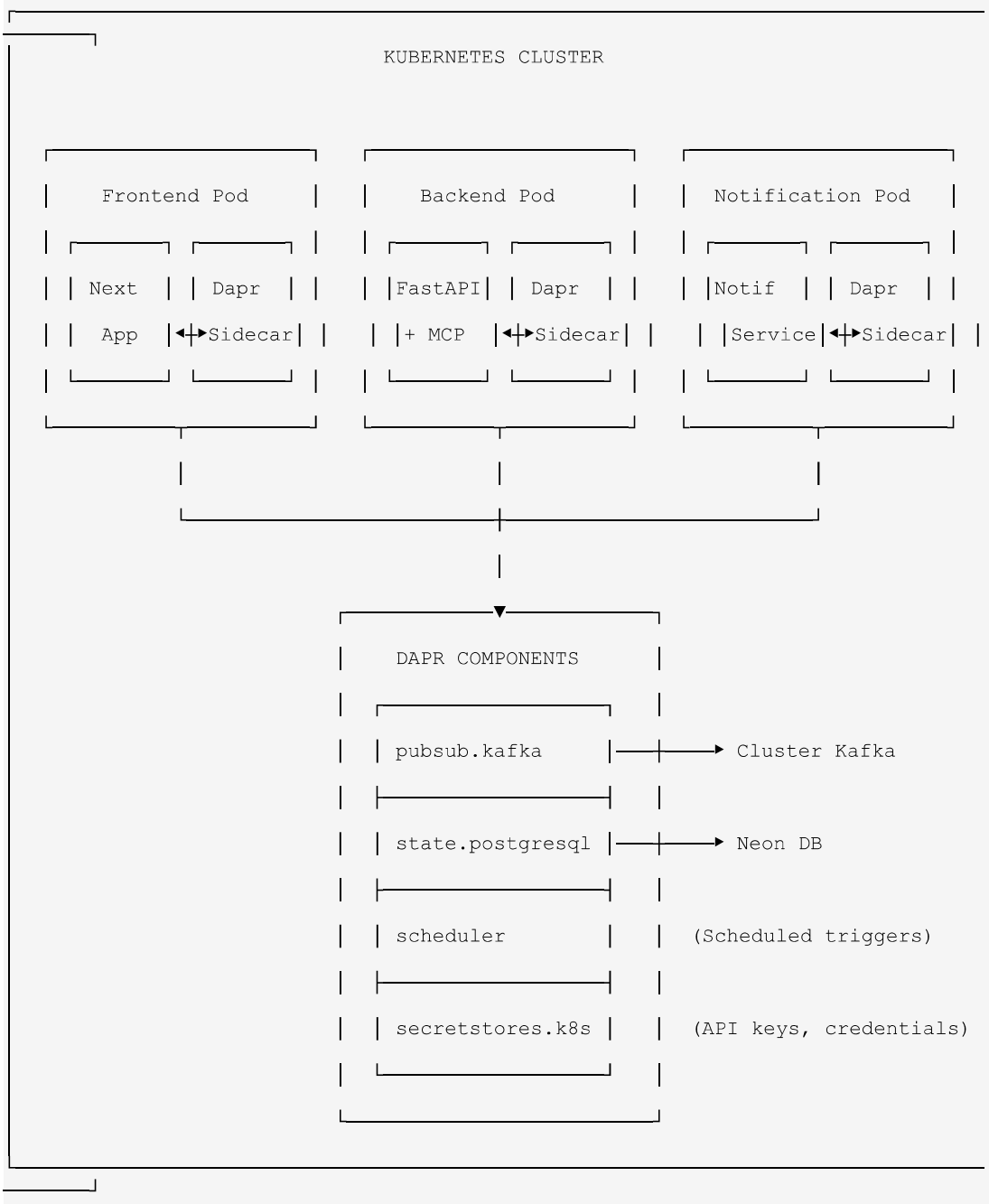
```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: kubernetes-secrets
spec:
  type: secretstores.kubernetes
  version: v1
```

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



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

2. 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

3. Demo Video (maximum 90 seconds):
- Demonstrate all implemented features
- Show spec-driven development workflow
- Judges will only watch the first 90 seconds
4. 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 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

3. **Initialize:** specify init creates the structure.
4. **Instruct:** AGENTS.md defines the rules.

5. **Bridge:** CLAUDE.md (@AGENTS.md) connects the agent.
6. **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