Hackathon III

Reusable Intelligence and Cloud-Native Mastery

Building Agentic Infrastructure with Skills, MCP Code Execution, Claude Code, and Goose

🚀 **Quick Start Checklist**

- Watch AAIF Announcement: https://www.youtube.com/watch?v=8WdO7U3KASo
- Watch Video Don't Build Agents, Build Skills Instead: https://www.youtube.com/watch?v=CEvIs9y1uog
- Read: MCP Code Execution Pattern - https://www.anthropic.com/engineering/code-execution-with-mcp
- Install: Minikube, Docker, Claude Code, Goose
- Create: skills with MCP code execution and learnflow-app repositories
- Windows: On Windows we will do all development using WSL
- Standards: Adopt Agentic AI Foundation (AAIF) Standards
- Read: This entire document before starting

Part 1: Introduction

Welcome to the Reusable Intelligence and Cloud-Native Mastery Hackathon! This hackathon represents a paradigm shift in software development. Instead of writing code manually, you'll learn to **teach AI coding agents** how to build sophisticated cloud-native agentic applications autonomously.

What You'll Learn

By the end of this hackathon, you will:

- Understand and implement the MCP Code Execution Implementation with Skills
- Create reusable skills that work with both Claude Code and Goose
- Deploy containerized applications on Kubernetes
- Build event-driven microservices using Kafka and Dapr
- Understand Agentic AI Foundation (AAIF) Standards
- Create a complete AI-powered learning platform (LearnFlow)

The Big Picture

Think of this hackathon as learning to be a **teacher for AI**. Rather than writing every line of code yourself, you'll create "Measurable Skills" that teach Claude Code and Goose how to architect, containerize, and orchestrate distributed systems. These skills become reusable knowledge that AI agents can apply to build complex applications autonomously.

💡 **Key Concept: From Coder to Teacher**

Traditional Development: You write code → Code runs → Application works

Agentic Development: You write Skills → AI learns patterns → AI writes code → Application works

The difference: Your skills can be reused to build many applications, not just one.

Critical: The Skills are the Product

The Skills are the product, not just documentation or the LearnFlow apps you create.

Judges will evaluate both the development process behind your Skills AND test the Skills with Claude Code and Goose. Your goal: make your skills work autonomously to get in the winners queue.

🛠️ Your Action Plan

Step 1: The "Unification" (Understand Standards)

- **The Challenge:** Take a standard task (e.g., "Design a Next.js Frontend").
- **The Action:** Look at how you did it in a Claude Skill with MCP
- **The Deliverable:** Recreate the Skill where MCP is Code executable instead of agent integration

Step 2: The "Build" (Construct LearnFlow)

- The Challenge: Build the LearnFlow platform (Microservices, Kafka, Dapr).
- The Action: DO NOT write Skills directly or integrate MCP servers with Coding Agentss. DO NOT write application code manually.
- The Deliverable: Write Skills with MCP executed in Code for every component (e.g., `kafka-setup.md`, `tutor-api.md`). Let Claude Code and Goose build the actual app.

Hackathon 2: You wrote Skills for Claude.
Hackathon 3: You write Skills with MCP Code Execution

Part 2A: Glossary of Terms

Before diving in, familiarize yourself with these key terms. Refer back to this section whenever you encounter unfamiliar terminology.

Term	Definition
Claude Code	Anthropic's agentic CLI tool. It can write, execute, and debug code autonomously. Uses "Skills" (SKILL.md files) to learn new capabilities.
Goose	Open-source local AI agent from the Agentic AI Foundation. Uses "Recipes" (recipe.yaml files) to learn new capabilities.
Skill	The emerging industry-standard format for teaching AI agents capabilities. A SKILL.md file with YAML frontmatter containing instructions and supporting scripts. Works on Claude Code, OpenAI Codex, and Goose.
Recipe	Goose's format for learning capabilities. A recipe.yaml file with title, description, instructions, activities, and extensions.
MCP	Model Context Protocol. A standard for giving AI agents real-time access to external data sources like databases, APIs, or your Kubernetes cluster.
Dapr	Distributed Application Runtime. A sidecar that handles state management, pub/sub messaging, and service invocation for microservices.
Kafka	A distributed event streaming platform. Services publish events to "topics" and other services subscribe to receive them asynchronously.

Term	Definition
Kubernetes (K8s)	Container orchestration platform. Manages deployment, scaling, and operations of containerized applications across clusters of machines.
Minikube	A tool that runs a single-node Kubernetes cluster locally on your machine for development and testing.
Helm	Package manager for Kubernetes. Helm "charts" are pre-configured templates for deploying applications like Kafka or PostgreSQL.
Spec-Kit Plus	A framework for spec-driven development. You define application behavior in specification files, then AI agents generate the implementation.
AGENTS.md	A markdown file that describes a repository's structure, conventions, and guidelines so AI agents can understand how to work with the codebase.

Part 2B: Goose vs Claude Code

Goose and Claude Code are both powerful AI agents for software development, but they have fundamental differences in **architecture, flexibility, and focus**.

In short: **Goose** is an open-source AAIF Standard, local-first, LLM-agnostic *agent framework* for full workflow automation. **Claude Code** is a proprietary, cloud-first, Claude-model-specific *tool* that focuses on code-centric tasks.

Here is a detailed comparison:

Feature	Goose (AAIF Standard)	Claude Code (by Anthropic)
Architecture	**Local-first**, extensible, open-source AAIF Standard AI agent. Runs on your machine (CLI/Desktop app).	**Cloud-first** agentic development tool. Leverages Anthropic's cloud services.
LLM Flexibility	**LLM-Agnostic**. Works with *any* LLM that supports tool calling, including local models (like open-source models) and commercial ones (like Claude, GPT, Gemini).	**Claude-Specific**. Tied exclusively to Anthropic's Claude models (e.g., Claude 3.5 Sonnet, Claude 4 Opus). But can use other models by using Claude Code Router.

Core Function	**Full Autonomous Workflow Orchestration**. Automates complex, end-to-end engineering tasks (scaffolding, installing deps, running tests, interacting with APIs, etc.).	**Code-centric Operations**. Excels at searching, explaining, editing code, performing PR reviews, and managing issues.
Code Execution	**Direct System Execution** (on the local machine) or via MCP servers. Can directly write, execute, debug, and test code.	**Direct Action** via integration with the development environment (reading/editing files, running commands).
Control & Security	**Maximum Local Control/Security**. Code and data primarily remain on your machine. Excellent for sensitive or air-gapped environments.	**Cloud-based Security/Control**. Relies on Anthropic's security measures and cloud infrastructure.
Pricing	**Model Dependent**. You only pay for the LLM you choose to use (API costs). The Goose framework itself is open-source.	**Subscription/API Model**. You pay Anthropic for the service and the underlying Claude model usage.
Best For	Developers who need **full local control, multi-model flexibility, and end-to-end automation** across their entire engineering pipeline.	Developers who want **top-tier code intelligence** and are already invested in the **Claude ecosystem** for code-specific tasks and refactoring.

🔑 Key Takeaways

- **Flexibility and Control:** Goose wins on **flexibility** as you can plug in the best-performing or most cost-effective LLM for a given task (even Claude!). It also provides **more security** and **local control** since it runs on your machine

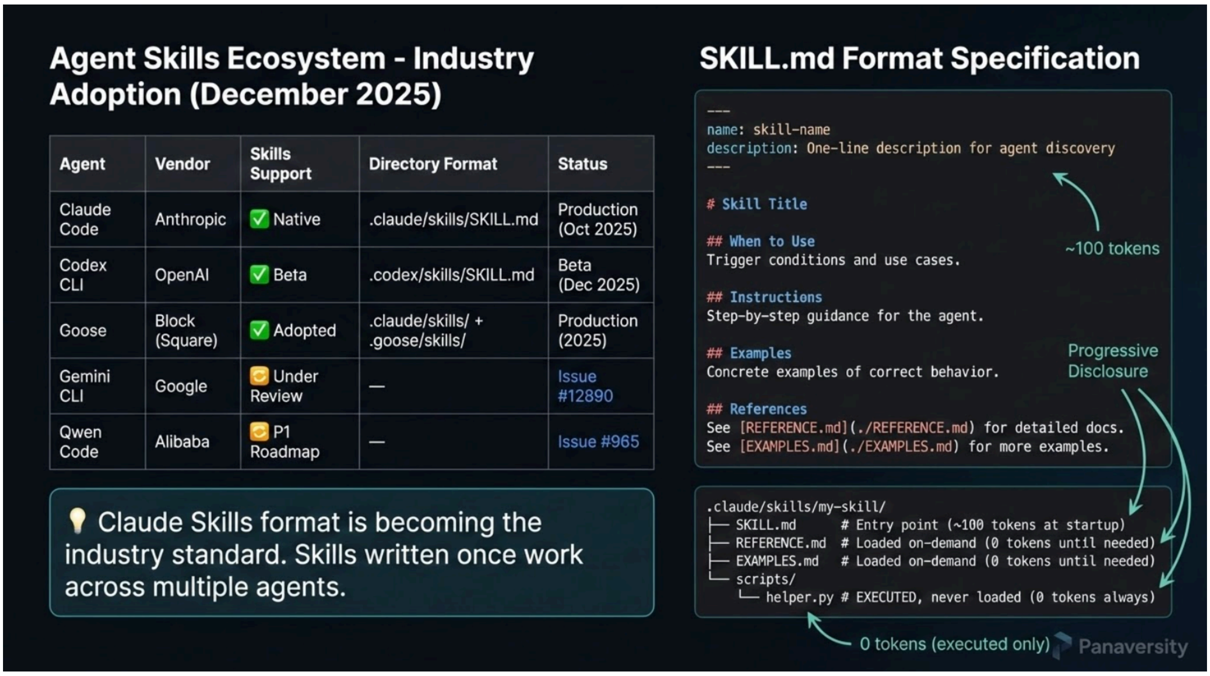
- **Breadth of Automation:** Goose is designed to be a general-purpose *engineering assistant* that can orchestrate entire workflows, not just coding tasks Claude Code is a more *specialized* tool focused primarily on code and repository management
- **Code Quality:** Claude models (Opus, Sonnet) are often cited on benchmarks as being among the best for raw **code generation accuracy and quality**, particularly for complex reasoning and adhering to multi-step instructions Goose's performance in this area will depend entirely on the specific LLM you configure it to use.

In summary, if you value **open-source, local execution, and the ability to swap models**, Goose is the stronger choice. If you prioritize having **Anthropic's SOTA code LLMs** directly integrated into your terminal environment for code-specific tasks, Claude Code is a powerful contender.

Part 3: Understanding Skills with MCP Code Execution

The Industry Convergence

Skills are the emerging standard for teaching AI coding agents. The industry has converged on Claude's Skills format:



💡 Skills written once work across Claude Code, Codex, and Goose. No transpilation needed.

The Token Problem: MCP Bloat

When you connect MCP servers directly to an agent, every tool definition loads into context at startup:

MCP Servers Connected	Token Cost BEFORE Conversation
1 server (5 tools)	~10,000 tokens
3 servers (15 tools)	~30,000 tokens
5 servers (25 tools)	~50,000+ tokens

⚠️ With 5 MCP servers, you've consumed 25% of your context window before typing a single prompt.

But it gets worse. Every intermediate result also flows through context:

Direct MCP call - transcript flows through context TWICE

TOOL CALL: gdrive.getDocument(documentId: "abc123")

→ returns full transcript (25,000 tokens into context)

TOOL CALL: salesforce.updateRecord(data: { Notes: [full transcript] })

→ model writes transcript again (25,000 more tokens)

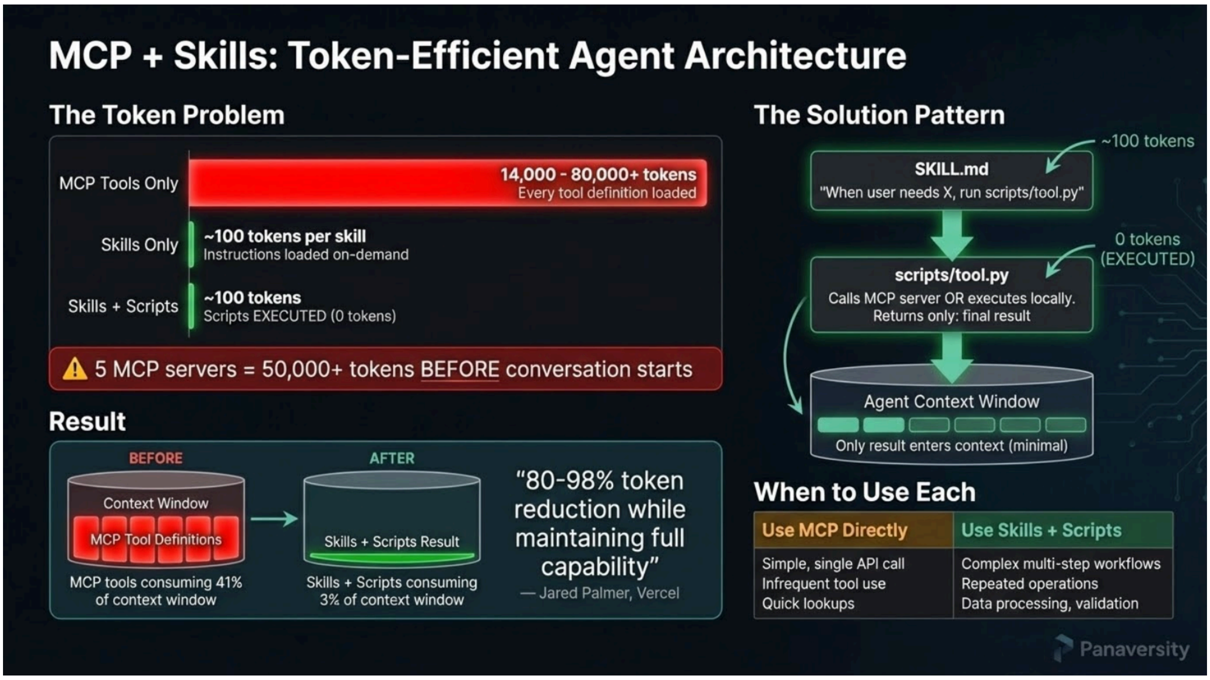
Total: 50,000 tokens for a simple copy operation

The Solution: Skills + Code Execution

Instead of loading MCP tools directly, wrap them in Skills that execute scripts:

The Pattern:

1. SKILL.md tells the agent WHAT to do (~100 tokens)
2. scripts/*.py does the actual work (0 tokens - executed, not loaded)
3. Only the final result enters context (minimal tokens)



Context Window Optimization: From Direct MCP to Skills + Scripts

The diagram illustrates a significant improvement in the agent's context window efficiency, moving from a "Before" state using Direct MCP (Managed Context Pool) to an "After" state leveraging Skills and Scripts.

Before (Direct MCP): High Context Consumption

- **Agent Context Window Contents:**
  - MCP Tool Definitions (50k tokens)
  - Intermediate Results (50k tokens)
  - Your Conversation (variable tokens)
- **Consumption Rate:** 41% of the context window is consumed *before* the agent even starts processing the task.

After (Skills + Scripts): Minimal Context Consumption

- **Mechanism:**
  - A compact `SKILL.md` file ($\sim$100 tokens) references an external script (`scripts/tool.py`).
  - The external script executes outside of the main context window.
  - The only output returned to the context window is the minimal result, e.g., "✓ Done."
- **Consumption Rate:** Only 3% of the context window is consumed, leaving 97% free for work.

This can result in 80-98% token reduction while maintaining full capability

How It Works: Code Execution with MCP?

From Anthropic's engineering blog (Nov 2025):

Strategy: Treat MCP Servers as Code APIs for Efficient Agent Interaction

The core idea is to shift from direct, high-volume tool calls to having the agent write code that interacts with the MCP servers via APIs. This allows for client-side processing, dramatically reducing the data volume the agent's context needs to handle.

The Problem with Direct MCP Tool Calls (Inefficient):

- **Action:** A direct tool call is made.
- **Result:** The entire dataset flows into the agent's context.

Example (Inefficient): All 10,000 rows flow through context
TOOL CALL: gdrive.getSheet(sheetId: 'abc123')
→ returns 10,000 rows in context

The Solution: Code Execution for Data Filtering (Efficient):

- **Action:** The agent executes a script using the server's functionality as an API.
- **Result:** Data filtering and processing occur within the script, and only the necessary, filtered output is returned to the agent's context.

Example (Efficient): Filter happens in script, only 5 rows reach context

```
const allRows = await gdrive.getSheet({ sheetId: 'abc123' });
const pending = allRows.filter(row => row.Status === 'pending');
console.log(pending.slice(0, 5)); // Only log first 5
```

Outcome: The agent's context sees only 5 relevant rows instead of 10,000, leading to significant efficiency gains.

Building Skills with MCP Code Execution

Here's the pattern you'll implement in this hackathon:

Directory Structure:

```
.claude/skills/kafka-k8s-setup/
├── SKILL.md            # Instructions (~100 tokens)
├── REFERENCE.md        # Deep docs (loaded on-demand)
└── scripts/
    ├── deploy.sh       # Executes Helm commands
    ├── verify.py       # Calls kubectl, returns status
    └── mcp_client.py   # Wraps MCP calls (optional)
```

SKILL.md (What the agent loads):

```
---
name: kafka-k8s-setup
description: Deploy Apache Kafka on Kubernetes
---

# Kafka Kubernetes Setup

## When to Use
- User asks to deploy Kafka
- Setting up event-driven microservices

## Instructions
1. Run deployment: `./scripts/deploy.sh`
2. Verify status: `python scripts/verify.py`
3. Confirm all pods Running before proceeding.

## Validation
- [ ] All pods in Running state
- [ ] Can create test topic

See [REFERENCE.md](./REFERENCE.md) for configuration options.
```

scripts/deploy.sh (What actually executes):

```
#!/bin/bash
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
kubectl create namespace kafka --dry-run=client -o yaml | kubectl apply -f -
helm install kafka bitnami/kafka --namespace kafka \
  --set replicaCount=1 \
  --set zookeeper.replicaCount=1

# Only this output enters agent context:
echo "✓ Kafka deployed to namespace 'kafka'"
```

```
scripts/verify.py (Returns minimal result):
#!/usr/bin/env python3
```

```
import subprocess, json, sys

result = subprocess.run(
    ["kubectl", "get", "pods", "-n", "kafka", "-o", "json"],
    capture_output=True, text=True
)
pods = json.loads(result.stdout)["items"]

running = sum(1 for p in pods if p["status"]["phase"] == "Running")
total = len(pods)

# Only this enters context - not the full pod JSON
if running == total:
    print(f"✓ All {total} pods running")
    sys.exit(0)
else:
    print(f"✗ {running}/{total} pods running")
    sys.exit(1)
```

Component	Tokens	Notes
SKILL.md	~100	Loaded when triggered
REFERENCE.md	0	Loaded only if needed
deploy.sh	0	Executed, never loaded
verify.py	0	Executed, never loaded
Final output	~10	"✓ All 3 pods running"

Total: ~110 tokens vs 50,000+ with direct MCP

Advanced Pattern: MCP Server as Skill

For MCP servers you use frequently, convert them to Skills:

Before (MCP Server loaded at startup): ~/.claude/mcp.json

```
{
  "servers": {
    "kubernetes": { "command": "mcp-k8s-server" }
  }
}
```

Cost: ~15,000 tokens at startup, every session

After (Skill + Script): .claude/skills/k8s-ops/SKILL.md

```
---
name: k8s-ops
description: Kubernetes operations via kubectl
---
```

Instructions
Use scripts in this directory for K8s operations:
- `scripts/get_pods.py <namespace>` - List pods
- `scripts/get_logs.py <pod> <namespace>` - Get logs
- `scripts/apply.py <file>` - Apply manifest

Cost: ~100 tokens when triggered, 0 otherwise

Your Hackathon Challenge

Hackathon 2: You wrote Skills for Claude.
Hackathon 3: You write Skills with MCP Code Execution.

For each skill you create:

1. SKILL.md — Minimal instructions (~100 tokens)
2. scripts/ — Code that does the heavy lifting (0 tokens)
3. REFERENCE.md — Deep docs loaded only when needed

The goal: Single prompt → Agent loads skill → Script executes → Minimal result → Task complete.

Part 4: Environment Setup

Follow these step-by-step instructions to set up your development environment. Complete each section before moving to the next. On Windows please do all development with WSL.

4.1 Install Prerequisites

Docker

```
# macOS
brew install --cask docker

# Ubuntu/Debian
sudo apt-get update
sudo apt-get install docker.io docker-compose
sudo usermod -aG docker $USER

# Verify installation
docker --version
```

Minikube (Local Kubernetes)

```
# macOS
brew install minikube

# Ubuntu/Debian
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube

# Start cluster with recommended resources
minikube start --cpus=4 --memory=8192 --driver=docker

# Verify
kubectl cluster-info
```

Helm (Kubernetes Package Manager)

```
# macOS
brew install helm

# Ubuntu/Debian
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash

# Verify
helm version
```

Claude Code

```
# macOS
brew install --cask claude-code

# Ubuntu/Debian
curl -fsSL https://claude.ai/install.sh | bash

# Authenticate
claude auth login



# Verify
claude --version
```

Goose

```
# macOS
brew install --cask block-goose

# Ubuntu/Debian
curl -fsSL https://github.com/block/goose/releases/download/stable/download_cli.sh | bash

# Verify
goose --version
```

4.2 Create Repositories

```
# 1. Create skills-library repository
mkdir skills-library && cd skills-library
git init
mkdir -p .claude/skills

# 2. Create learnflow-app repository
cd ..
mkdir learnflow-app && cd learnflow-app
git init
```

4.3 Verify Everything Works

Run this verification script to ensure your environment is ready:

```
#!/bin/bash
echo "Checking prerequisites..."
```

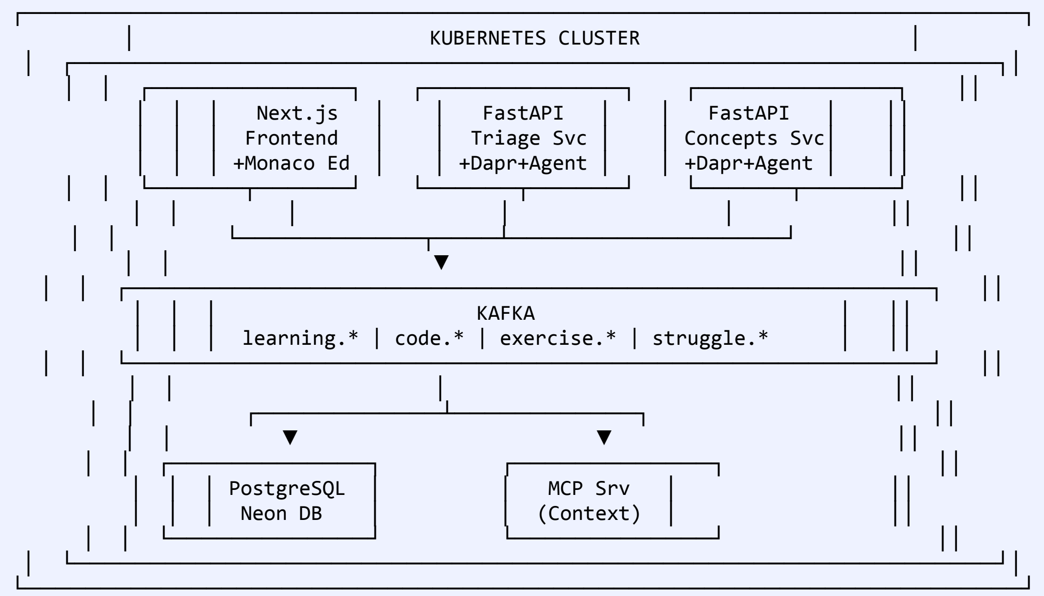
```
docker --version && echo "✓ Docker OK"
minikube status && echo "✓ Minikube OK"
kubectl cluster-info && echo "✓ Kubectl OK"
helm version && echo "✓ Helm OK"
claude --version && echo "✓ Claude Code OK"
goose --version && echo "✓ Goose OK"

echo "All checks passed! Ready for hackathon."
```

Part 5: Technical Stack Overview

This section provides context for each technology you'll be working with.

LearnFlow Architecture



Technology Summary

Layer	Technology	Purpose
AI Coding Agents	Claude Code, Goose (Claude Code Router)	Execute your Skills to build the application
Frontend	Next.js + Monaco	User interface with embedded code editor
Backend	FastAPI + OpenAI SDK	AI-powered tutoring agents as microservices
Auth	Better Auth	Authentication framework
Service Mesh	Dapr	State management, pub/sub, service invocation
Messaging	Kafka on Kubernetes	Asynchronous event-driven communication
Database	Neon PostgreSQL	User data, progress, code submissions
API Gateway	Kong API Gateway on Kubernetes	Routes traffic and handles JWT authentication
AI Context	MCP Servers	Give AI agents real-time access to data
Orchestration	Kubernetes	Deploy and manage all containerized services
Continuous Delivery	Argo CD + GitHub Actions	Argo CD is tool for Continuous Delivery (CD) on Kubernetes using the GitOps approach, and it works exceptionally well with Helm and GitHub Actions
Documentation	Docusaurus	Auto-generated documentation site

Part 6: Your Deliverables

You will create and submit two repositories. Here's exactly what each should contain.

Form to Submit: https://forms.gle/Mrhf9XZsuXN4rWJf7

Repository 1: Skills Library (skills-library)

Structure:

```
skills-library/
├── README.md
├── .claude/skills/          # Works on Claude + Goose + Codex
│   ├── agents-md-gen/
│   │   ├── SKILL.md
│   │   └── scripts/
│   ├── kafka-k8s-setup/
│   │   ├── SKILL.md
│   │   ├── REFERENCE.md
│   │   └── scripts/
│   ├── postgres-k8s-setup/
│   ├── fastapi-dapr-agent/
│   ├── mcp-code-execution/
│   ├── nextjs-k8s-deploy/
│   └── docusaurus-deploy/
└── docs/
    └── skill-development-guide.md
```

Skill Name	Purpose	Must Include	Potential New Skill Ideas
agents-md-gen	Generate AGENT…	SKILL.md + script	**agent-testing-framework**: Automated testing for agent interactions
kafka-k8s-setup	Deploy Kafka on K…	SKILL.md + deploy scri…	**kafka-stream-processor**: Deploying and managing Kafka Stream processing applications
postgres-k8s-setup	Deploy PostgreSQL	SKILL.md + migration s…	**pg-data-backup-restore**: Implementing automated backup and recovery for PostgreSQL on K8s
fastapi-dapr-agent	FastAPI + Dapr se…	SKILL.md + templates	**dapr-pubsub-binding**: Implementing Dapr Pub/Sub and Bindings for microservices

mcp-code-execution	MCP with code ex…	SKILL.md + Python scri…	**mcp-state-management**: Implementing durable state management in the MCP pattern
nextjs-k8s-deploy	Deploy Next.js apps	SKILL.md + Dockerfile t…	**nextjs-perf-optimize**: Techniques for optimizing Next.js application performance and bundling
docusaurus-deploy	Deploy document…	SKILL.md + deploy script	**docusaurus-search-config**: Configuring and fine-tuning search functionality in Docusaurus
-	-	-	**prometheus-grafana-setup**: Monitoring setup for K8s applications (Prometheus/Grafana)
-	-	-	**argocd-app-deployment**: Implementing GitOps for application deployment using ArgoCD

Repository 2: LearnFlow Application (learnflow-app)

The complete application built using your Skills:
- Built using Claude Code and/or Goose with your custom Skills
- Both agents can use the same .claude/skills/ directory

Commit Message Tip:
Your commit history should reflect an agentic workflow. Use messages like:
- "Claude: implemented Kafka consumer using kafka-k8s-setup skill"
- "Goose: deployed PostgreSQL using postgres-k8s-setup skill"

Part 7: Development Roadmap

Follow these phases to build your solution incrementally. Each phase builds on the previous one.

Number	Phase	Deliverables
1	Setup	Environment ready, repos created, Minikube running
1-2	Foundation Skills	agents-md-gen, k8s-foundation skills working
2-3	Infrastructure	Kafka + PostgreSQL deployed via Skills
3-4	Backend Services	FastAPI + Dapr + Agent microservices
4-5	Frontend	Next.js with Monaco editor deployed
5-6	Integration	MCP servers + Docusaurus documentation
6-7	LearnFlow Build	Complete application via Claude + Goose
8	Polish & Demo	Documentation complete, demo ready, submitted
9	Cloud Deployment	Deploy on Azure, Google, or Oracle Cloud
10	Continues Deployment	Use Argo CD with Github Actions

Phase Details

Phase 1: Setup

Goal: Development environment ready

1. Install all prerequisites (see Part 4)
2. Start Minikube: minikube start --cpus=4 --memory=8192
3. Create skills-library and learnflow-app repositories
4. Run verification script to confirm setup

✓ **Success Criteria:** kubectl cluster-info returns cluster information

Phase 2: Foundation SKILLS

Goal: Basic Skills for project setup

- **agents-md-gen:** Teaches AI agents how to create AGENTS.md files
- **k8s-foundation:** Check cluster health and apply basic Helm charts

✓ **Success Criteria:** AI agents generate valid AGENTS.md from a single prompt

Phase 3: Infrastructure SKILLs

Goal: Skills for stateful infrastructure

- **kafka-k8s-setup:** Deploy Kafka, create topics, verify connectivity
- **postgres-k8s-setup:** Deploy PostgreSQL, run migrations, verify schemas

✓ **Success Criteria:** AI agents autonomously deploy and verify Kafka/PostgreSQL

Part 8: LearnFlow Application Specification

LearnFlow is an AI-powered Python tutoring platform. This section provides the complete business requirements.

Product Overview

LearnFlow helps students learn Python programming through conversational AI agents. Students can chat with tutors, write and run code, take quizzes, and track their progress. Teachers can monitor class performance and generate custom exercises.

Role	Features
Student	Chat with Python tutor, write & run code, take coding quizzes, view progress
Teacher	View class progress, receive struggle alerts, generate coding exercises

Python Curriculum

Module	Topics Covered
1. Basics	Variables, Data Types, Input/Output, Operators, Type Conversion
2. Control Flow	Conditionals (if/elif/else), For Loops, While Loops, Break/Continue
3. Data Structures	Lists, Tuples, Dictionaries, Sets
4. Functions	Defining Functions, Parameters, Return Values, Scope
5. OOP	Classes & Objects, Attributes & Methods, Inheritance, Encapsulation
6. Files	Reading/Writing Files, CSV Processing, JSON Handling
7. Errors	Try/Except, Exception Types, Custom Exceptions, Debugging
8. Libraries	Installing Packages, Working with APIs, Virtual Environments

AI Agent System

LearnFlow uses a multi-agent architecture where specialized agents handle different aspects of tutoring.

Agent	Purpose & Capabilities
Triage Agent	Routes queries to specialists: "explain" → Concepts, "error" → Debug
Concepts Agent	Explains Python concepts with examples, adapts to student level
Code Review Agent	Analyzes code for correctness, style (PEP 8), efficiency, readability
Debug Agent	Parses errors, identifies root causes, provides hints before solutions
Exercise Agent	Generates and auto-grades coding challenges
Progress Agent	Tracks mastery scores and provides progress summaries

Business Rules

Mastery Calculation

Topic Mastery = weighted average of:

- Exercise completion: 40%
- Quiz scores: 30%
- Code quality ratings: 20%
- Consistency (streak): 10%

Mastery Levels:

- 0-40% → Beginner (Red) | 41-70% → Learning (Yellow)

- 71-90% → Proficient (Green) | 91-100% → Mastered (Blue)

Struggle Detection Triggers

- Same error type 3+ times
- Stuck on exercise > 10 minutes
- Quiz score < 50%
- Student says "I don't understand" or "I'm stuck"
- 5+ failed code executions in a row

Code Execution Sandbox

- Timeout: 5 seconds | Memory: 50MB
- No file system access (except temp) | No network access
- Allowed imports: standard library only (MVP)

Demo Scenario

This scenario demonstrates the key features of LearnFlow:

1. **Student Maya** logs in → Dashboard shows: "Module 2: Loops - 60% complete"
2. Maya asks: *"How do for loops work in Python?"*
3. Concepts Agent explains with code examples and visualizations
4. Maya writes a for loop in the Monaco editor, runs it successfully
5. Agent offers a quiz → Maya gets 4/5 → Mastery updates to 68%
6. **Student James** struggles with list comprehensions → Gets 3 wrong answers
7. Struggle alert sent to teacher Mr. Rodriguez
8. Teacher views James's code attempts, types: "Create easy exercises on list comprehensions"
9. Exercise Agent generates exercises → Teacher assigns with one click
10. James receives notification → Completes exercises → Confidence restored

Part 9: Evaluation Criteria

Your submission will be scored on these criteria. Understand what "Gold" standard means for each.

Criterion	Weight	Gold Standard
Skills Autonomy	15%	AI goes from single prompt to running K8s deployment, zero manual intervention
Token Efficiency	10%	Skills use scripts for execution, MCP calls wrapped efficiently
Cross-Agent Compatibility	5%	Same skill works on Claude Code AND Goose
Architecture	20%	Correct Dapr patterns, Kafka pub/sub, stateless microservice principles
MCP Integration	10%	MCP server provides rich context enabling AI to debug and expand system
Documentation	10%	Comprehensive Docusaurus site deployed via Skills playbook
Spec-Kit Plus Usage	15%	High-level specs translate cleanly to agentic instructions
LearnFlow Completion	15%	Application built entirely via skills

Remember: The Skill is the Product and how it was developed is the process.

Judges will evaluate both the development process behind your Skills AND test them with Claude Code and Goose. Your goal: make your skills work autonomously to get in the winners queue.

Part 10: FAQ & Troubleshooting

Frequently Asked Questions

Q: Do I need to build both Claude Code and Goose versions?
A: Yes, both are required. This demonstrates that your Skills are truly portable. Since Goose reads .claude/skills/ directly, the same skills work on both agents.

Q: What if Claude Code or Goose generates incorrect code?
A: This is expected! The goal is to refine your Skills until the AI generates correct code consistently. Document what changes you made to improve the skills.

Q: Can I use other AI models besides Claude and Goose?
A: Yes, you can use Claude Code Router to integrate Gemini or other APIs. However, Claude Code and Goose are required as the primary agents.

Q: How much should the AI do vs. manual coding?
A: Aim for maximum autonomy. Your evaluation score increases when AI agents can complete tasks with minimal manual intervention. The gold standard is single-prompt-to-deployment.

Common Issues & Solutions

⚠️ **Issue: Minikube won't start**
Symptoms: "Exiting due to DRV_NOT_HEALTHY" or Docker errors
Solution:

```
1. Ensure Docker Desktop is running
2. minikube delete && minikube start --driver=docker
3. If on Mac M1/M2: minikube start --driver=docker --alsologtostderr
```

⚠️ **Issue: Helm chart installation fails**
Symptoms: "no matches for kind" or version errors
Solution:

```
helm repo update
helm search repo bitnami/kafka --versions  # Find compatible version
helm install kafka bitnami/kafka --version X.Y.Z
```

⚠️ **Issue: Claude Code not recognizing Skills**
Symptoms: Skill doesn't appear or isn't used
Solution:

```
1. Verify SKILL.md is in .claude/skills/<name>/SKILL.md
2. Check YAML frontmatter syntax (--- at start and end)
3. Run: claude --debug to see skill loading
4. Ensure allowed-tools are valid tool names
```

⚠️ **Issue: Pods stuck in Pending state**
Symptoms: kubectl get pods shows Pending for >5 minutes
Solution:

```
kubectl describe pod <pod-name>  # Check Events section
# Common causes:
# - Insufficient resources: Increase Minikube memory
# - PVC issues: Check storage class exists
# - Image pull: Verify image name and registry access
```

Part 11: Resources

Official Documentation

- **Agentic AI Foundation (AAIF):** https://aaif.io/
- **Watch AAIF Announcement:** https://www.youtube.com/watch?v=8WdO7U3KASo
- **Claude Code Skills:** code.claude.com/docs/en/skills
- **Goose Documentation:** block.github.io/goose/
- **Model Context Protocol:** modelcontextprotocol.io
- **Dapr:** dapr.io
- **OpenAI Agents SDK:** github.com/openai/openai-agents-python
- **Kubernetes:** kubernetes.io/docs/
- **Minikube:** minikube.sigs.k8s.io/docs/
- **Helm:** helm.sh/docs/
  MCP Code Execution:
  https://www.anthropic.com/engineering/code-execution-with-mcp
- **Goose Skills Guide:**
  https://block.github.io/goose/docs/guides/context-engineering/using-skills
- **OpenAI Codex Skills**: https://github.com/openai/codex/blob/main/docs/skills.md

Good luck, Engineers!

It's time to stop writing code and start teaching machines how to build systems.

— End of Document —