GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

DSPy-backed agentic system to orchestrate Kubernetes,

Sadaf Shafi

MatrNr: 26350674

Supervisor: Mojtaba Akbari

Georg-August-Universität Göttingen
Institute of Computer Science

March 27, 2026

Abstract

Kubernetes is powerful, but it is often hard to use because commands and deployment steps can be complex. Many users need strong `kubectl` and YAML knowledge, and small mistakes can cause failures.

This report presents `k8sCMD`, a natural-language command-line system that lets users describe goals in plain English. The system uses a four-agent DSPy pipeline (Planner, Executor, Validator, and Summarizer) to break goals into tasks, run Kubernetes actions, check results, and return clear explanations.

A key feature is the validation and retry loop: when a task fails, the system generates corrective actions and tries again in a controlled way. Tests on a Docker-based Kubernetes mock environment show that this approach improves reliability and makes Kubernetes operations easier to understand, while still keeping transparency through confirmations and logs.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☑ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

CLI Command-Line Interface

DSPy Declarative Self-improving Python for DSPY

FPSR First Pass Success Rate

GCR Goal Completion Rate

ISFR Initial Step Failure Rate

LLM Large Language Model

REPL Read-Eval-Print Loop

RR Recovery Rate

1 Introduction

1.1 Background of the Study

The management of Kubernetes clusters traditionally requires deep technical expertise and strong familiarity with the `kubectl` command-line tool. Platform engineers and developers often need to write complex YAML deployment files or execute long-chained terminal commands to orchestrate containerized applications. The `k8sCMD` system aims to simplify this interaction by introducing a natural language interface, allowing users to express their goals in plain English instead of relying strictly on complex Kubernetes syntax.

1.2 Problem Statement

Despite the power of Kubernetes, its steep learning curve and verbose command structures present a significant barrier to entry. Errors in command execution or YAML formatting can cause critical deployment failures. There is a need for an intelligent intermediary that not only translates human language into valid Kubernetes commands but also validates the success of those commands in real time.

1.3 Objectives of the Project

The project is guided by three objectives:

- **Natural Language Processing:** Translate plain English operational goals into executable Kubernetes commands.
- **Autonomous Orchestration:** Design a multi-agent AI system that can plan, execute, validate, and summarize administrative tasks in a Kubernetes environment.
- **Safety and Verification:** Implement a robust validation and corrective feedback loop to ensure commands achieve the user's intended state without causing system instability.

1.4 Scope and Limitations

The scope of this project is building `k8sCMD`, a command-line interface (CLI) backed by a 4-agent DSPy pipeline using Large Language Models (LLMs). It interacts with a Kubernetes API endpoint. Current limitations include personalisation of the system to a particular user and contextual constraints inherited from the underlying LLM.

2 Methodology and System Architecture

2.1 Proposed System

The project utilizes an agentic AI methodology via the DSPy (Declarative Self-Improving Language Programs) framework developed by Stanford NLP [Kha+23; DSP26].

DSPy is chosen because it shifts development from brittle prompt engineering toward programmable, modular agent pipelines [Kha+23]. This enables explicit role separation, reliable retries, and model-agnostic deployment through LiteLLM.

Unlike standard LLM chatbots that only generate text, `k8sCMD` behaves as an autonomous agentic system. It perceives environment state, reasons over goals, executes actions through tools, and adapts based on feedback.

The architecture exhibits three core agentic characteristics:

- **Goal-oriented autonomy:** A high-level user goal (e.g., "Deploy Redis") is converted into executable Kubernetes sub-steps without requiring low-level manual instructions.
- **Environment interaction (tool use):** The Executor agent actively invokes the Kubernetes REST API and mutates or queries cluster state.
- **Dynamic feedback loops:** The Validator agent inspects results, explains failures, and triggers corrective actions, creating a self-healing ReAct-style reasoning-and-acting loop.

2.2 System Design and Architecture

As shown in Figure 1, the overall system acts as a bridge between the user interface, the DSPy orchestrator, and the Kubernetes target environment.

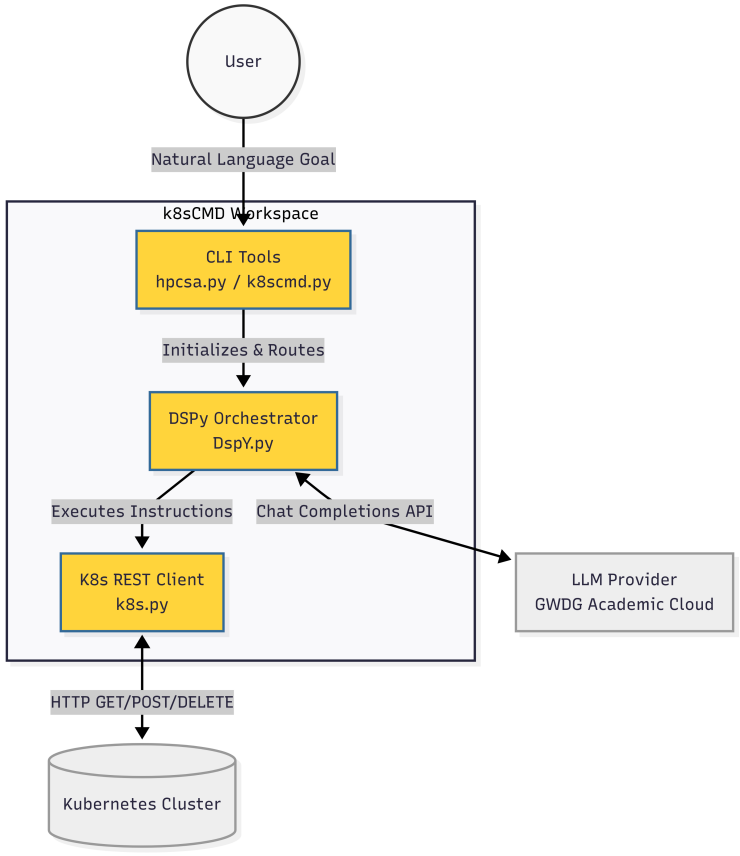

Figure 1: High-level system architecture of `k8sCMD`, connecting user input, DSPy orchestration, and the Kubernetes mock environment.

The internal 4-agent execution structure is illustrated in Figure 2. This diagram captures the linear data flow with corrective feedback:

1. **PlannerAgent:** Deconstructs plain-English goals into ordered atomic sub-tasks using chain-of-thought reasoning.
2. **ExecutorAgent:** Translates tasks into concrete `kubectl`-style operations and executes them via `K8sClient` REST calls.
3. **ValidatorAgent:** Verifies whether the intended state was achieved and, if necessary, emits corrective actions.
4. **SummarizerAgent:** Produces a concise and readable plain-English result for the user.

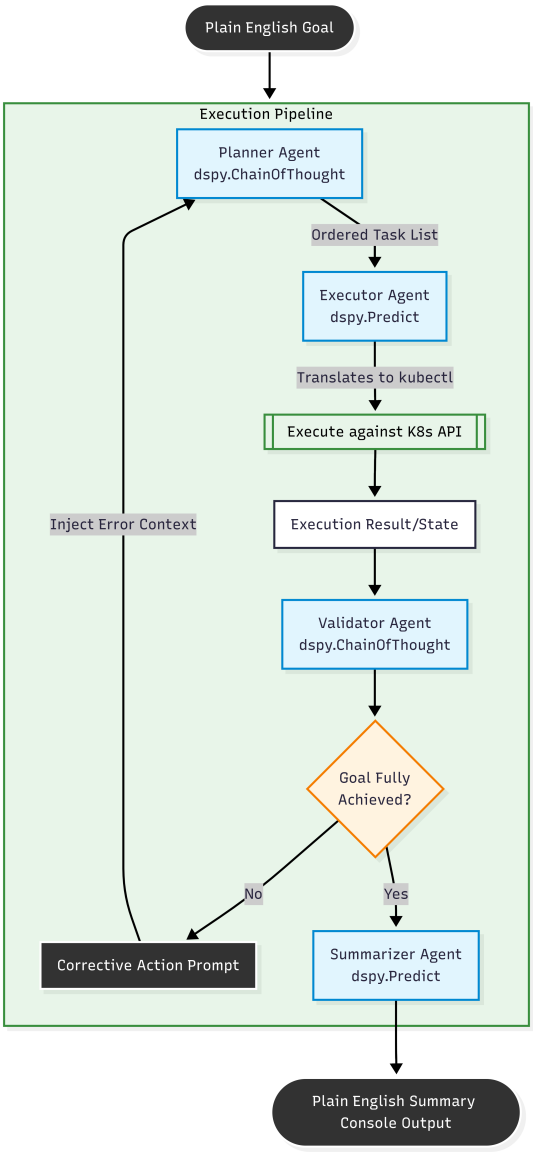

Figure 2: 4-agent DSPy execution pipeline with validation feedback loop in `k8sCMD`.

2.3 Tools and Technologies Used

- **Programming language:** Python 3
- **AI framework:** DSPy (Stanford NLP), integrated with LiteLLM for model routing
- **Large language model endpoint:** OpenAI-compatible GWDG Academic Cloud endpoint using `openai-gpt-oss-120b`
- **Infrastructure / target environment:** Dockerized Kubernetes Cluster Mock (`ezequielmr94/kubernetes-cluster-mock`)

We selected `gpt-oss-120b` because the model family is explicitly optimized for reasoning and agentic tool use. OpenAI reports strong performance in tool-calling and agentic evaluations, support for configurable reasoning effort (low/medium/high), and compatibility with structured outputs and agentic workflow patterns [Ope25]. These properties align directly with the requirements of `k8sCMD`, where the pipeline must plan, execute, validate, and recover through iterative tool interactions.

3 Implementation and Results

3.1 Implementation Details

The core orchestration logic is implemented in `DspY.py`, where the four agents are defined and coordinated. The module `k8s.py` serves as the REST abstraction layer that maps generated cluster operations to HTTP requests (GET/POST/DELETE) against endpoints such as `/api/v1/namespaces/{ns}/pods`.

The system additionally includes robust session logging to preserve every sub-task, generated command, validation result, and retry event. This supports both auditability and debugging while keeping the user-facing terminal output concise.

3.2 Outputs and Interface

Users interact through either an interactive `k8scmd` CLI command. For example, a prompt such as:

> "Deploy a Redis pod named redis-cache in production and verify it is running."

triggers the complete planning, execution, validation, and summarization lifecycle.

The interface exposes execution transparency through command previews and safety confirmations (`[y/n]` prompts), followed by a plain-language final summary. The feedback-oriented flow visible in Figure 2 reflects this runtime behavior.

3.3 Performance Evaluation and Testing

Performance and reliability are evaluated through repeated scenario execution against the Docker mock cluster, using the provided `tests/llm_test.py` infrastructure. Since `k8sCMD` is an agentic pipeline rather than a single-shot command generator, evaluation must capture not only immediate success but also recovery behavior after intermediate failures.

3.3.1 Evaluation Metrics

To evaluate `k8sCMD` appropriately, we track both first-attempt efficiency and final-goal reliability:

- **First Pass Success Rate (FPSR):** Percentage of experiments that succeed on the first execution pass without requiring Validator-driven correction.
- **Goal Completion Rate (GCR):** Percentage of experiments that reach the intended final state after all allowed retries.
- **Recovery Rate:** Percentage of initially failed experiments that are later corrected by the Validator-guided retry loop.

Based on the 12 experiments reported in Table 4, the quantitative results are shown in Table 1.

Metric	Result	Breakdown
First Pass Success Rate (FPSR)	58.3%	7 of 12 experiments succeeded on the first attempt.
Initial Step Failure Rate	41.7%	5 of 12 experiments required retry or pivot logic.
Recovery Rate	100%	All 5 initially failed experiments were corrected by the retry loop.
Final Goal Completion (GCR)	100%	All 12 experiments reached the intended final state.

Table 1: Evaluation metric summary for `k8sCMD` over 12 appendix experiments.

3.3.2 Interpretation of Intermediate Failures

The entries marked as failed-first-step in the appendix should be interpreted as robustness checks, not terminal system failures. In several scenarios, failure was intentionally triggered (e.g., deleting non-existent resources or invoking unsupported commands) to test whether the Validator can detect the issue and trigger a valid corrective plan. Test 7 additionally reflects environmental friction in the mock API behavior.

In an agentic orchestration setting, these intermediate failures are expected and informative: they demonstrate perception of environment errors and successful adaptation through a self-healing feedback loop.

3.4 Module Architecture and Responsibilities

3.4.1 Module Descriptions

1. `DspY` — The Core Pipeline. This module is the central brain of the system and implements a four-agent DSPy pipeline. The `PlannerAgent` (ChainOfThought) decomposes high-level goals into an ordered list of atomic Kubernetes tasks. The `ExecutorAgent` (Predict) converts each task into a kubectl-style command and executes it via `K8sClient`. The `ValidatorAgent` (ChainOfThought) checks whether the original goal is achieved; if not, it emits a corrective action and triggers a retry loop up to the configured maximum

retries. The `SummarizerAgent` (Predict) converts raw outputs into a clear plain-English response. The module also handles session logging and model configuration through `configure_lm()`.

2. `k8s` — **Kubernetes REST Client.** This module wraps the `kubernetes-cluster-mock` Docker service on `localhost:9988` and mirrors common kubectl operations, including listing/creating/deleting pods, deployments, and ingresses, listing nodes, and special operations such as pod phase changes and cluster resizing. Its `execute(command)` function parses kubectl-style command strings and dispatches them to the correct REST handler.

3. `LLM` — **Standalone LLM Wrapper.** This is a raw HTTP wrapper for the GWDG Academic Cloud OpenAI-compatible endpoint. It supports lightweight interactions independent of DSPy, including connection tests and retry logic (default three attempts).

4. `k8scmd` — **Interactive REPL Entry Point.** This module provides an interactive loop where users repeatedly enter natural-language goals. On startup, it initializes the model and logger, checks Kubernetes cluster connectivity, and forwards each goal to `run_pipeline()` in `DspY`.

5. `agents` / `orchestration`**.** These modules are currently placeholders reserved for future extension: agent-level additions and higher-level orchestration logic, respectively.

Component	Purpose
LLM orchestration	Plans, executes, validates, and summarizes tasks
Kubernetes client	Parses kubectl-style commands and calls mock cluster APIs
Standalone LLM access	Handles direct API calls with retries
Interactive interface	Runs the REPL loop for repeated goals

Table 2: Summary of major modules in the `k8sCMD` system.

3.4.2 Detailed Agentic Flow

The end-to-end orchestration is coordinated by `run_pipeline()` in `DspY`. All four agents are implemented as DSPy `Module` subclasses, which enables later optimization of prompts and behavior within DSPy without changing core orchestration code.

Step 1: PlannerAgent (ChainOfThought). The planner receives the user goal (or goal plus corrective guidance during retries) and produces an ordered list of atomic tasks. Chain-of-thought reasoning is used to model dependencies and prevent invalid sequencing (for example, trying to configure ingress before creating a target workload).

Step 2: ExecutorAgent (Predict). The executor processes one task at a time and emits exactly one kubectl-style command, then executes it via `K8sClient`. Predict is used instead of chain-of-thought to keep the output clean and executable. Execution context grows incrementally: outputs from earlier tasks are appended so later tasks can adapt. During normal execution, the interface requests `[y/n]` confirmation before running each command (and this execution can be skipped in `-dry-run` mode).

Step 3: ValidatorAgent (ChainOfThought). The validator consumes the original goal plus all tasks and execution outputs, then returns three fields: `achieved` (boolean), `reason` (short rationale), and `corrective_action`. Chain-of-thought helps it reason over

aggregate evidence before producing a binary verdict, reducing false positives and false negatives.

Step 4: SummarizerAgent (Predict). If validation succeeds, the summarizer converts commands and outputs into a concise 1–3 sentence user-facing response that directly answers the original goal.

Retry Loop. If `achieved = false` and retry budget remains, the validator-proposed `corrective_action` is appended to the original goal and fed back to the planner. This creates a self-correcting loop where the system replans with explicit remediation guidance.

Agent	DSPy Module	Rationale
PlannerAgent	ChainOfThought	Reasons about task ordering and inter-task dependencies before committing to a plan
ExecutorAgent	Predict	Must output a precise, single command string without reasoning text pollution
ValidatorAgent	ChainOfThought	Reasons over multi-step outcomes before producing a binary success/failure verdict
SummarizerAgent	Predict	Produces concise user-facing natural language rather than structured reasoning traces

Table 3: Why each agent in the pipeline uses its corresponding DSPy module type.

4 Conclusion

This project demonstrates that a multi-agent LLM framework can significantly reduce the operational friction of Kubernetes management. By decomposing responsibilities into Planning, Executing, Validating, and Summarizing, `k8sCMD` mitigates typical hallucination and reliability problems of monolithic LLM usage.

The resulting CLI enables safer, more accessible infrastructure operation by aligning human intention with verifiable cluster-state transitions through a transparent natural-language interface.

5 Future Work

- **Comparison without DSPy:** Conduct a controlled comparison between the current DSPy-backed workflow and an equivalent system that does not use DSPy, measuring reliability, correction quality, and task completion rate.
- **Cross-model agent benchmarking:** Evaluate multiple LLMs designed for agentic tool use (beyond OpenAI OSS), and compare first-pass success, recovery quality, latency, and cost across identical Kubernetes task suites.
- **Expand orchestration intelligence:** Extend orchestration capabilities with richer multi-step planning, dependency-aware execution, and improved handling of complex operational intents.

- **Enhanced safety intelligence:** Add stronger safety intelligence through policy-aware validation, stricter guardrails, and context-sensitive risk checks before command execution.

References

[DSP26] DSPy. *DSPy Documentation*. https://dspy.ai/. Accessed: 2026-03-10. 2026. URL: https://dspy.ai/.

[Kha+23] Omar Khattab et al. "DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines". In: (2023). arXiv: 2310.03714 [cs.CL]. URL: https://arxiv.org/abs/2310.03714.

[Ope25] OpenAI. *Introducing gpt-oss*. https://openai.com/index/introducing-gpt-oss/. Accessed: 2026-03-27. 2025. URL: https://openai.com/index/introducing-gpt-oss/.

A Code samples

Repository: `https://github.com/SadafShafi/k8sCMD`

B Demo Screenshot

The following screenshot demonstrates the `k8sCMD` system in action.

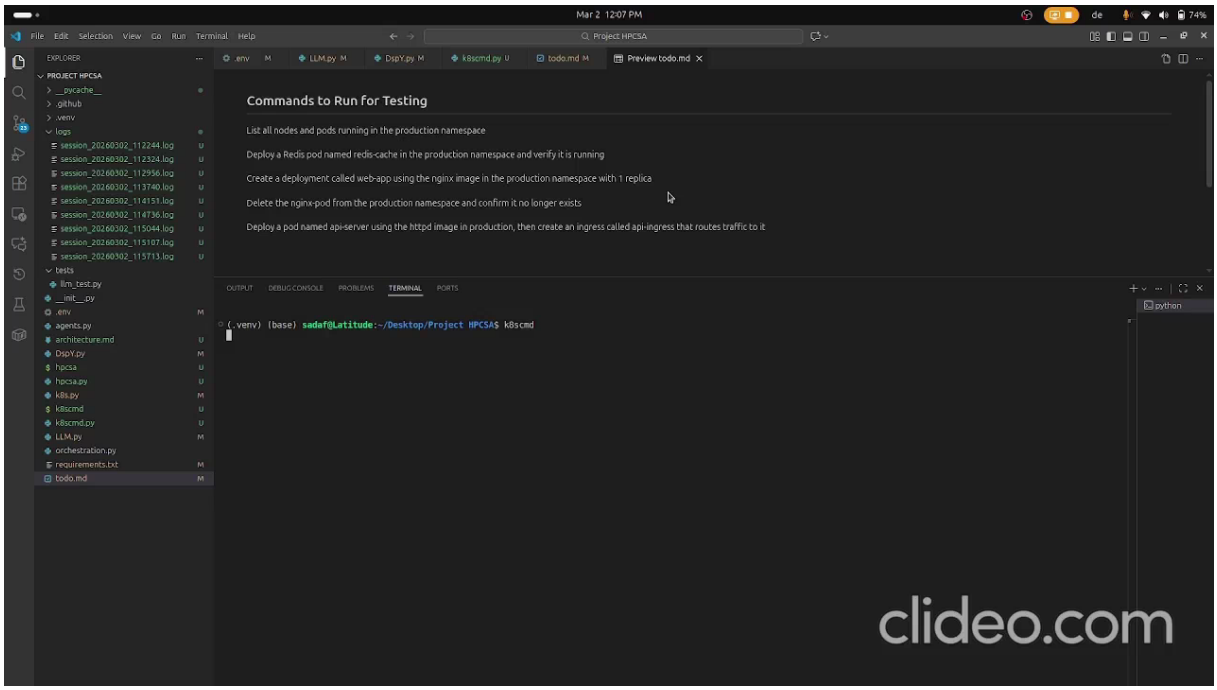

Figure 3: Demo screenshot of the `k8sCMD` workflow.

Demo repository link: `https://github.com/SadafShafi/k8sCMD`

C Evaluation Run Summary

Table 4 summarizes prompts, expected behavior, actual outcomes, and recovery actions across the `k8sCMD` evaluation runs.

Table 4: k8sCMD evaluation run summary.

Experiment	Prompt	Expected Output	Actual Output	Status	System Suggestion (if failed)	Retries
1. List Resources	`List all nodes and pods running in the production namespace`	Returns nodes and pods in the production namespace.	Listed nodes and pods exactly as expected.	Success	N/A	1

Continued on next page

Experiment	Prompt	Expected Output	Actual Output	Status	System Suggestion (if failed)	Retries
2. Deploy & Verify Pod	`Deploy a Redis pod named redis-cache in the production namespace and verify it is running`	Creates `redis-cache`, then verifies it via `get pods`.	Verified creation and Running phase of `redis-cache`.	Success	N/A	1
3. Create Deployment	`Create a deployment called web-app using the nginx image in the production namespace with 1 replica`	Triggers deployment creation for `web-app`.	Successfully created deployment `web-app` using `nginx`.	Success	N/A	1
4. Delete Resource	`Delete the nginx-pod from the production namespace and confirm it no longer exists`	Deletes `nginx-pod` and verifies removal via `get pods`.	Confirmed `nginx-pod` is not found in production.	Success/Retried	`kubectl delete pod nginx-pod -n production -ignore-not-found`	2+
5. Deploy & Ingress	`Deploy a pod named api-server using the httpd image in production, then create an ingress called api-ingress that routes traffic to it`	Creates `api-server` and maps `api-ingress`.	Deployed `api-server` and created `api-ingress` successfully.	Success	N/A	1
6. Cross-Namespace Awareness	`List all pods in the 'development' namespace and then list all pods in the 'staging' namespace`	Queries both namespaces and summarizes their contents.	Confirmed both namespaces have no pods.	Success	N/A	1
7. Sequential State Mutation	`Delete the deployment named 'v1-app' in the 'production' namespace, and replace it by creating a new deployment named 'v2-app' using the 'nginx:alpine' image in the same namespace`	Delete may fail (not found), then `v2-app` is created.	Delete failed first; mock API could not ignore not-found, but `v2-app` creation succeeded.	Failed	`kubectl delete deployment v1-app -n production -ignore-not-found=true`	3
8. Multi-Component Interdependency	`Deploy a backend pod named 'auth-service' using the 'redis' image in the 'secure' namespace. Once running, create an ingress named 'auth-ingress' that routes traffic to it`	Pod runs in `secure`; ingress is created.	Successfully deployed pod and created mapped ingress in `secure`.	Success	N/A	1

Continued on next page

Experiment	Prompt	Expected Output	Actual Output	Status	System Suggestion (if failed)	Retries
9. Full Lifecycle Pipeline	`Create a deployment called 'temp-worker' with the 'busybox' image in the 'batch' namespace. Verify it exists by listing the deployments in that namespace, and then immediately delete it`	Deployment is created, listed, and deleted in one run.	Successfully created, verified, and deleted `temp-worker` in `batch`.	Success	N/A	1
10. Basic Self-Healing Loop	`Try to delete a pod named 'missing-data-pod' in the 'production' namespace. When it fails, seamlessly recover by deploying a new pod named 'fallback-pod' using the 'nginx' image instead`	Delete fails (404), then fallback creates `fallback-pod`.	Initial delete failed; system correctly created `fallback-pod`.	Failed first step (Expected)	Execute fallback deployment since pod was not found.	2
11. The "Clean-up Failure"	`Try to delete a pod named 'ghost-pod' in the 'production' namespace. When that operation fails because the pod doesn't exist, recover your process by skipping the deletion and just listing the nodes in the cluster instead`	Delete fails, then pipeline pivots to `kubectl get nodes`.	Intercepted `ghost-pod` deletion failure and pivoted to getting nodes.	Failed first step (Expected)	Skip deletion, pivot to query nodes.	2
12. The "Unsupported Command" Pivot	`Scale the deployment 'web-server' to 5 replicas. Since scaling isn't a supported command in our mock API, the system will throw an 'Unsupported' error. When you catch that error, pivot to creating a new deployment named 'web-server-v2' using the 'httpd' image`	`ExecutorAgent` catches unsupported command; `ValidatorAgent` pivots to creating deployment.	Recognized `scale` was unsupported, then generated `web-server-v2`.	Failed first step (Expected)	Skip scale operation, execute creation fallback.	2