The emergence of AI agents as primary operators within an engineering organization presents both a profound opportunity and a significant challenge. This document outlines a comprehensive platform system design tailored for an agent-augmented environment, with a focus on resilience, observability, and scalability, specifically leveraging AWS services.

Overview: Designing for an Agent-Driven Platform

Our goal is to create a robust and forward-looking platform system where AI agents are the primary users and operators. This necessitates a paradigm shift in how we design our CI/CD pipelines, infrastructure orchestration, and agent lifecycle management. The proposed architecture emphasizes autonomous operation while maintaining human oversight and intervention capabilities.

Part 1: System Architecture Design

The core of our agent-driven platform system will be a highly automated, self-healing environment.

Core Components and Responsibilities:

- **Agent Orchestration Service (AOS):** The central nervous system for agents.
    - **Agent Registry:** Stores agent identities, capabilities, current state, and assigned tasks.
    - **Task Scheduler:** Distributes tasks to available and capable agents based on predefined policies and dynamic system state.
    - **Workflow Engine:** Manages complex multi-agent workflows, ensuring dependencies are met and progress is tracked.
    - **Communication Hub:** Facilitates inter-agent communication and human-agent interaction (e.g., via a message bus).
    - **Policy Enforcement Module:** Ensures agents adhere to operational policies, security constraints, and resource limits.
    - **AWS Services:** Leveraged using AWS Step Functions for workflows, AWS Lambda for individual task execution by agents, and Amazon SQS/SNS for communication.
- **Infrastructure Orchestration Engine (IOE):** Manages the underlying infrastructure.

    - **IaC Repository (AWS CodeCommit/GitHub):** Stores all infrastructure definitions (Terraform/CloudFormation).
    - **Provisioning Module:** Provisions and de-provisions infrastructure based on agent requests and policy.
    - **Configuration Management Module:** Configures and maintains infrastructure (e.g., AWS Systems Manager, AWS OpsWorks).
    - **Drift Detection:** Continuously monitors infrastructure for deviations from desired state.
    - **AWS Services:** AWS CloudFormation/Terraform (managed via AWS CodePipeline), AWS Systems Manager.
- **CI/CD Pipeline as a Service (PaaS):** Enables agents to define and execute delivery pipelines.
    - **Pipeline Definition Store (AWS CodeCommit/S3):** Stores pipeline definitions (e.g., YAML files).
    - **Execution Engine:** Executes pipeline stages, integrating with code repositories, build services, and deployment targets.
    - **Artifact Repository (Amazon S3/ECR):** Stores build artifacts and container images.
    - **Security Scanning Module:** Integrates with security tools (e.g., Amazon Inspector, third-party SAST/DAST) for automated vulnerability detection.
    - **AWS Services:** AWS CodePipeline, AWS CodeBuild, AWS CodeDeploy, Amazon ECR, Amazon S3.
- **Observability and Analytics Platform (OAP):** Provides deep insights into agent and system behavior.
    - **Log Aggregation (Amazon CloudWatch Logs, OpenSearch Service):** Centralizes all system and agent logs.
    - **Metrics Collection (Amazon CloudWatch Metrics, Prometheus):** Gathers performance and operational metrics.
    - **Distributed Tracing (AWS X-Ray, OpenTelemetry):** Tracks requests and operations across agents and services.
    - **Anomaly Detection (Amazon CloudWatch Anomaly Detection, Amazon Lookout for Metrics):** Identifies deviations from baseline behavior.
    - **Reporting and Visualization (Amazon QuickSight, Grafana):** Provides dashboards and reports for human operators.
    - **AWS Services:** Amazon CloudWatch, AWS X-Ray, Amazon OpenSearch Service, Amazon QuickSight, Amazon Lookout for Metrics.
- **Agent Runtime Environment (ARE):** The execution environment for individual agents.

    - **Containerization (Amazon ECS/EKS):** Provides isolated, reproducible environments for agents.
    - **Resource Management:** Allocates CPU, memory, and network resources to agents.
    - **Security Contexts:** Enforces least-privilege access for agents.
    - **AWS Services:** Amazon ECS/EKS, AWS IAM.
- **Human-in-the-Loop (HITL) Interface:** A dedicated interface for human oversight and intervention.
    - **Alerting and Notification System (Amazon SNS, PagerDuty):** Notifies humans of critical events or anomalies.
    - **Approval Workflows (AWS Step Functions):** Manages human approval steps for sensitive operations.
    - **Audit Trail and Replay:** Provides a complete historical record of agent actions for auditing and debugging.
    - **Command and Control Dashboard:** Allows humans to issue commands, adjust policies, and override agent decisions.
    - **AWS Services:** Amazon SNS, AWS Step Functions, Amazon CloudWatch Events, custom UI leveraging AWS Amplify/AppSync.

Agent Sensors: Measurements and Telemetry for Agents:

Agents require rich, real-time telemetry to make informed decisions.

- **System Metrics:** CPU utilization, memory usage, network I/O, disk space for all infrastructure components.
- **Application Metrics:** Latency, error rates, throughput, saturation of services and APIs.
- **CI/CD Pipeline Status:** Build success/failure rates, deployment progress, test coverage, artifact availability.
- **Security Posture:** Vulnerability scan results, compliance status, network flow logs, unusual access patterns.
- **Resource Availability:** Current capacity of compute, storage, and network resources; available IP addresses.
- **Cost Metrics:** Real-time cost consumption of provisioned resources.
- **Agent-Specific Metrics:** Task completion rates, decision-making latency, resource consumption per agent, communication success rates.
- **External Service Health:** Health checks and status of integrated third-party services.

- **Feedback Loops:** Human feedback and approval statuses.
- **AWS Services:** Amazon CloudWatch, AWS Config, AWS Security Hub, AWS Cost Explorer, custom metrics pushed to CloudWatch.

Automatic Control and Human-in-the-Loop:

- **Full AI Agent Control:**
    - **Routine Infrastructure Provisioning/De-provisioning:** Spinning up/down ephemeral environments for demo teams.
    - **Automated Scaling:** Adjusting resource allocation based on observed load.
    - **Self-Healing:** Remediation of common infrastructure failures (e.g., restarting services, replacing unhealthy instances).
    - **Automated Testing and Validation:** Execution of unit, integration, and performance tests.
    - **Minor Configuration Updates:** Applying predefined patches or configuration changes.
    - **CI/CD Pipeline Execution:** Triggering builds, running tests, and deploying to non-production environments.
- **Human-in-the-Loop Required:**
    - **Production Deployments:** Approval for critical production changes, especially for Long-Term Platform Teams.
    - **Major Infrastructure Changes:** Significant architecture shifts or resource re-allocations that could impact multiple services.
    - **Policy Adjustments:** Changes to security policies, resource allocation policies, or agent behavioral policies.
    - **Anomaly Remediation for Unknown Issues:** When agents identify an anomaly but cannot determine the root cause or a safe remediation.
    - **New Agent Onboarding and Training:** Initial setup and training data provision for new agent capabilities.
    - **Disaster Recovery Scenarios:** High-impact events requiring strategic human decision-making.
    - **Auditing and Compliance Reporting:** Periodic human review of agent actions for regulatory compliance.

Communication and Orchestration Patterns:

- **Message Buses (Amazon SQS/SNS):** Decoupled, asynchronous communication between agents and services. Agents publish events (e.g., "build completed," "resource requested"), and interested parties subscribe.

- **Agent Registries (AOS Component):** Agents register their capabilities and availability. Other agents or the AOS can query the registry to discover and invoke suitable agents.
- **Coordination Protocols:** Standardized APIs and data models for inter-agent communication. This ensures agents understand each other's requests and responses. GraphQL or RESTful APIs exposed via AWS API Gateway could serve this purpose.
- **Event-Driven Architecture:** Events are the primary mechanism for triggering actions. For example, a "code commit" event triggers a CI agent, which then publishes a "build success" event, triggering a deployment agent. AWS EventBridge can be used to route events.
- **Shared State via Centralized Data Stores (Amazon DynamoDB/Aurora):** Agents can access and update shared information about system state, task progress, and resource utilization.

Isolation and Safety Mechanisms:

- **Least Privilege IAM Roles:** Each agent operates with the absolute minimum necessary permissions on AWS. Policies are granular and frequently audited.
- **Containerization (Amazon ECS/EKS):** Isolates agent execution environments, preventing a misbehaving agent from impacting others. Resource limits are enforced at the container level.
- **Network Segmentation (AWS VPC, Security Groups, NACLs):** Agents are placed in segmented networks, limiting their ability to access unauthorized resources.
- **Circuit Breakers and Bulkheads:** In multi-agent workflows, these patterns prevent failures in one agent or service from cascading and bringing down the entire system. AWS Step Functions states can incorporate retry and catch mechanisms.
- **Immutable Infrastructure:** Infrastructure is replaced rather than modified in place, reducing configuration drift and making rollbacks simpler.
- **Policy-as-Code (AWS Config, AWS Organizations SCPs):** Security and operational policies are defined as code and automatically enforced.
- **Canary Deployments/Blue-Green Deployments:** For sensitive deployments, changes are rolled out gradually, allowing for quick rollback if issues arise. Agents would manage these strategies.

Observability: Monitoring Agent Performance, Anomalies, and Emergent Behavior:

- **Comprehensive Logging:** Every agent action, decision, and system interaction is logged with contextual information. Logs are centralized, indexed, and searchable (Amazon CloudWatch Logs, OpenSearch Service).
- **Detailed Metrics:** Track agent-specific metrics (e.g., task success/failure rates, decision-making latency, resource consumption), and system-wide metrics (e.g., infrastructure stability, deployment frequency, mean time to recovery).
- **Distributed Tracing (AWS X-Ray):** Trace the entire lifecycle of a request or task across multiple agents and services, providing a holistic view of execution flow and identifying bottlenecks.
- **Anomaly Detection (Amazon CloudWatch Anomaly Detection, Amazon Lookout for Metrics):** Machine learning models continuously analyze metrics and logs to detect deviations from normal behavior, flagging potential misbehavior or emerging issues.
- **Behavioral Auditing:** A dedicated audit trail of all agent actions, decisions, and policy evaluations for compliance and post-mortem analysis.
- **Emergent Behavior Detection:** Beyond simple anomalies, look for patterns across multiple agents or system components that indicate unintended emergent behaviors. This may involve using AWS SageMaker for custom ML models to analyze complex interaction patterns.
- **Human-Readable Dashboards and Alerts:** Utilize Amazon QuickSight or Grafana for visualizing key metrics and trends. Configure Amazon SNS for alerts on critical thresholds or detected anomalies.
- **Agent Self-Reporting:** Agents are designed to report their internal state, confidence levels in decisions, and any perceived ambiguities or conflicts.

Key Tradeoffs and Assumptions:

- **Tradeoff: Autonomy vs. Control:** Increasing agent autonomy reduces human operational burden but increases the risk surface. Our approach seeks a balance by implementing robust HITL mechanisms for critical operations.
- **Tradeoff: Performance vs. Resilience:** Highly resilient systems often have overhead (e.g., redundant components, extensive logging). We prioritize resilience, accepting potential minor performance impacts for critical platform stability.
- **Assumption: Agents are Deterministic (within bounds):** While agents use AI, we assume their core decision-making processes, when given the same inputs and policies, will produce consistent, predictable outputs within a defined operational envelope. Non-determinism is assumed to be within acceptable variance or flagged as an anomaly.

- **Assumption: Data Quality for Training:** The effectiveness of AI agents heavily relies on high-quality, representative training data. Poor data will lead to poor agent performance.
- **Assumption: Clear Task Definition and Goal Alignment:** Agents need clearly defined goals and tasks to operate effectively. Ambiguity can lead to undesirable emergent behavior.
- **Assumption: Secure AI Model Deployment:** The underlying AI models for agents are assumed to be securely deployed and protected against adversarial attacks.

Part 2: Resilience Strategy

Building fault tolerance and rollback mechanisms for AI agents is paramount to ensuring system stability.

Fault Tolerance and Rollback Mechanisms:

- **Immutable Infrastructure & Rollback:** When an agent attempts an infrastructure change (e.g., deploying a new version of a service), it should be performed using immutable deployments (e.g., launching new instances/containers and shifting traffic). If the new deployment exhibits misbehavior (detected via monitoring metrics, anomaly detection), traffic can be instantly rolled back to the previous stable version. This is managed by the IOE and PaaS.
- **Transactional Agent Workflows (AWS Step Functions):** Complex, multi-step agent workflows are designed as transactions. If a step fails or an agent misbehaves mid-workflow, the entire transaction can be rolled back to a known good state or compensated for. Step Functions' error handling and retry mechanisms are crucial here.
- **State Checkpoints:** For long-running agent tasks, periodic checkpoints of their internal state are saved. If an agent crashes or misbehaves, it can resume from the last good checkpoint. This can be stored in Amazon DynamoDB.
- **Automated Remediation Playbooks:** For known failure modes or misbehaviors, pre-defined automated remediation playbooks are triggered by monitoring and anomaly detection. These playbooks can restart services, roll back deployments, or scale resources.
- **Agent Sandboxing and Quarantining:** If an agent consistently misbehaves or enters an unrecoverable state, it can be automatically quarantined or isolated from the main system, preventing it from causing further damage. This involves terminating its execution environment and alerting human operators.

- **Versioned Agent Models and Policies:** AI agent models and their operational policies are versioned. If a new version of an agent or policy leads to misbehavior, the system can automatically revert to a previous, stable version. This can be managed through AWS CodeCommit/S3 and AWS SageMaker Model Registry.
- **Human-Initiated Emergency Stop:** A "panic button" or emergency stop mechanism allows human operators to immediately halt all agent operations or specific agent types in critical situations.

Dangerous Assumptions About Agent Behavior:

- **Agents are Infinitely Rational/Optimal:** Agents are designed to be rational within their programming and training data, but they can still encounter unforeseen edge cases, make suboptimal decisions in novel situations, or be influenced by biases in their training data.
- **Agents will Always Follow Instructions Perfectly:** While agents are programmed to follow instructions, errors in logic, environmental changes, or misinterpretations of complex commands can lead to deviations.
- **Agents Understand Human Intent (without explicit definition):** Agents operate based on explicit instructions and learned patterns. They don't inherently understand implicit human intent or common sense unless explicitly programmed or trained to do so.
- **Agents Will Never Collude/Develop Emergent Malicious Behavior:** While unlikely in well-designed systems, the possibility of unintended emergent "collusion" or self-reinforcing negative feedback loops between agents, leading to undesirable outcomes, should be considered and monitored.
- **Agents are Immune to Adversarial Attacks:** AI models are susceptible to adversarial attacks that can manipulate their inputs to produce incorrect or malicious outputs. Robust security measures for input validation and model integrity are crucial.
- **Agents Can Handle All Unforeseen Circumstances:** Agents excel at handling anticipated scenarios. Novel, truly unforeseen circumstances can lead to unpredictable or frozen behavior.

Ensuring Usability and Safety for Human Takeover/Audit:

- **Comprehensive Audit Trails:** Every action, decision, and communication by an agent is meticulously logged and easily accessible. This provides a clear,

immutable record for human review and auditing (Amazon CloudWatch Logs, OpenSearch Service).

- **Clear State Representation:** The current state of the system, including all ongoing agent tasks, resource allocations, and deployed versions, is presented in a human-understandable format through the HITL Interface.
- **Read-Only Access for Audit:** Human auditors can be granted read-only access to all system components, logs, and metrics without the ability to inadvertently alter the system.
- **Pause and Resume Capabilities:** Humans can pause ongoing agent operations at specific points in workflows to inspect the state, make manual adjustments, and then resume the workflow.
- **Granular Control and Override:** The HITL Interface allows humans to override specific agent decisions, cancel tasks, or issue direct commands that bypass agent autonomy for critical interventions.
- **Semantic Logging:** Agent logs are not just technical data but include high-level descriptions of their reasoning and intent behind actions, making it easier for humans to understand "why" an agent did something. This requires agents to explicitly log their decision-making process.
- **Interactive Debugging Tools:** Tools that allow humans to simulate agent actions, "step through" agent decision processes, or replay past agent behaviors to understand their logic.

Part 3: Multiple Environments

Perle's diverse product teams (Short-Term Demo, Medium-Term Customer, Long-Term Platform) necessitate a flexible yet robust environment provisioning system. Our design uses a tiered approach for infrastructure, CI/CD, and agent behavior.

High-Level Design for Three Requirements:

- **Shared Core Platform, Differentiated Configuration:** The underlying platform components (AOS, IOE, PaaS, OAP) remain largely consistent. The differentiation comes in the **configuration, policies, and agent behaviors** applied to each environment type.
- **Infrastructure as Code (IaC) Templates:** Standardized IaC templates (CloudFormation/Terraform) will define the baseline infrastructure for each environment type.

- **Environment Profiles:** Define the specific configurations, security policies, resource quotas, and agent operational parameters for each team type. These profiles are versioned and managed.
- **Automated Provisioning and De-provisioning:** Agents, controlled by the AOS, will automate the lifecycle management of these environments based on the defined profiles.
- **Dedicated Agent Pools (Optional but Recommended):** For long-term platform teams, consider dedicated agent pools or agent specializations to ensure dedicated resources and expertise for critical shared services.

Environment Specifics:

1. **Short-Term Demo Teams:**
    a. **Infrastructure:** Highly ephemeral, cost-optimized, and disposable. Utilizes serverless (AWS Lambda, Fargate) and auto-scaling groups for quick spin-up/spin-down. Minimal persistence.
    b. **CI/CD:** Rapid iteration, minimal human gates. Automated builds, deployments to a dedicated demo environment. Focus on speed over exhaustive testing.
    c. **Agent Behavior:** Prioritizes speed and low cost. Aggressive de-provisioning after a short time. Agents are optimized for rapid iteration and resource cleanup.
    d. **Isolation:** Isolated VPCs or subnets per demo, but resource sharing can be acceptable for cost efficiency within a controlled tenant.
2. **Medium-Term Customer Teams:**
    a. **Infrastructure:** More stable, persistent environments. Utilizes services like Amazon EC2, RDS, and EKS/ECS for longer-term operation. Scalability and reliability are more important.
    b. **CI/CD:** More rigorous testing (integration, performance). Staging environments are mandatory. Human approval for production deployments is often required. Blue/Green or Canary deployments.
    c. **Agent Behavior:** Balances speed with reliability. Agents are more cautious with deployments, incorporating more health checks and rollback strategies. Agents monitor customer-facing metrics closely.
    d. **Isolation:** Stronger isolation between customer environments (e.g., dedicated VPCs for each customer or strong tenancy models within shared infrastructure).
3. **Long-Term Platform Teams:**

a. **Infrastructure:** Highly durable, scalable, and resilient. Multi-AZ, multi-region deployments. Extensive use of managed services (DynamoDB Global Tables, Aurora Global Database). Strict security and compliance.
b. **CI/CD:** Most rigorous testing, including security audits, compliance checks, and extensive performance testing. Multiple human approval gates for critical changes. Dark launches and advanced deployment strategies.
c. **Agent Behavior:** Prioritizes stability, security, and long-term maintainability. Agents operate with extreme caution, extensive pre-checks, and multi-stage rollout processes. Extensive monitoring for any deviation.
d. **Isolation:** Maximum isolation, often dedicated accounts or highly segmented VPCs. Strict network controls and access policies.

CLI Tool Implementation (Python)

This basic CLI tool demonstrates the core logic for spinning up environments based on the track type. It doesn't provision real AWS infrastructure but simulates the process.

Python

```
import argparse
import os
import json
from datetime import datetime

class EnvironmentProvisioner:
    def __init__(self, output_dir="env_provisioning_logs"):
        self.output_dir = output_dir
        os.makedirs(self.output_dir, exist_ok=True)
        self.env_configs = {
            "demo": {
                "description": "Short-Term Demo Environment - ephemeral, cost-optimized,
rapid.",
                "infrastructure_template": "aws_demo_template.json",
                "ci_cd_profile": "demo_ci_cd_profile.json",
                "agent_policy": "demo_agent_policy.json",
                "resources": ["Lambda", "DynamoDB (on-demand)", "API Gateway"]
            },
            "customer": {
                "description": "Medium-Term Customer Environment - stable, scalable,
```

```
persistent.",
            "infrastructure_template": "aws_customer_template.json",
            "ci_cd_profile": "customer_ci_cd_profile.json",
            "agent_policy": "customer_agent_policy.json",
            "resources": ["EC2 (Auto Scaling)", "RDS", "EKS", "S3"]
        },
        "platform": {
            "description": "Long-Term Platform Environment - durable, highly available,
secure.",
            "infrastructure_template": "aws_platform_template.json",
            "ci_cd_profile": "platform_ci_cd_profile.json",
            "agent_policy": "platform_agent_policy.json",
            "resources": ["EC2 (Dedicated Instances)", "Aurora Global DB", "EKS (Multi-AZ)",
"VPC Endpoints", "GuardDuty"]
        }
    }

  def _generate_env_id(self, track):
    timestamp = datetime.now().strftime("%Y%m%d%H%M%S")
    return f"{track}-{timestamp}-{os.urandom(4).hex()}"

  def _simulate_aws_provisioning(self, env_id, config):
    """Simulates calling AWS services for provisioning."""
    print(f"  [SIMULATION] Initiating AWS CloudFormation/Terraform for {env_id} with
template: {config['infrastructure_template']}")
    # In a real scenario, this would call AWS SDK (boto3)
    # e.g., boto3.client('cloudformation').create_stack(...)
    print(f"  [SIMULATION] Configuring CI/CD pipeline for {env_id} using profile:
{config['ci_cd_profile']}")
    # e.g., boto3.client('codepipeline').create_pipeline(...)
    print(f"  [SIMULATION] Deploying AI agents configured with policy:
{config['agent_policy']}")
    # This would involve updating agent orchestration service policies or deploying agent
containers
    print(f"  [SIMULATION] Provisioning core resources: {', '.join(config['resources'])}")

  def new_environment(self, track: str, name: str = None):
    """
```

```
    Spins up a new environment based on the specified track type.
    """
    if track not in self.env_configs:
        print(f"Error: Invalid track type '{track}'. Available tracks are: {',
'.join(self.env_configs.keys())}")
        return

    env_config = self.env_configs[track]
    env_id = name if name else self._generate_env_id(track)
    log_file_path = os.path.join(self.output_dir, f"{env_id}.log")

    print(f"\n--- Initiating Environment Provisioning for '{track}' track ---")
    print(f"Environment ID: {env_id}")
    print(f"Description: {env_config['description']}")

    provisioning_details = {
        "env_id": env_id,
        "track_type": track,
        "description": env_config['description'],
        "infrastructure_template": env_config['infrastructure_template'],
        "ci_cd_profile": env_config['ci_cd_profile'],
        "agent_policy": env_config['agent_policy'],
        "provisioned_resources_simulation": env_config['resources'],
        "status": "IN_PROGRESS",
        "start_time": str(datetime.now())
    }

    with open(log_file_path, "w") as f:
        json.dump(provisioning_details, f, indent=4)
        f.write("\n\n--- Simulation Log ---\n")
        f.write(f"[{datetime.now()}] Starting provisioning for {env_id}\n")

        print(f"  [INFO] Writing provisioning details to {log_file_path}")

        # Simulate interaction with internal platform services and AWS
        self._simulate_aws_provisioning(env_id, env_config)

        provisioning_details["status"] = "COMPLETED"
```

```
        provisioning_details["end_time"] = str(datetime.now())

        f.write(f"[{datetime.now()}] Provisioning completed for {env_id}\n")
        json.dump(provisioning_details, f, indent=4)


    print(f"--- Environment '{env_id}' Provisioning Initiated (simulated) ---")
    print(f"Details logged to: {log_file_path}")
    print(f"Next steps for a real system:\n"
          f"  1. An Agent Orchestration Service (AOS) agent would pick up this request.\n"
          f"  2. It would trigger the Infrastructure Orchestration Engine (IOE) to provision
resources.\n"
          f"  3. CI/CD pipelines would be configured via the PaaS.\n"
          f"  4. Agent-specific configurations would be applied.\n")

def main():
  parser = argparse.ArgumentParser(
    description="Perle Environment Provisioning CLI Tool",
    formatter_class=argparse.RawTextHelpFormatter
  )
  subparsers = parser.add_subparsers(dest="command", required=True, help="Available
commands")

  # `perle-env new` command
  new_parser = subparsers.add_parser("new", help="Spin up a new environment.")
  new_parser.add_argument(
    "--track",
    choices=["demo", "customer", "platform"],
    required=True,
    help="Type of environment to provision: 'demo', 'customer', or 'platform'."
  )
  new_parser.add_argument(
    "--name",
    type=str,
    help="Optional: A specific name for the environment. If not provided, a unique ID will
be generated."
  )
```

```
    args = parser.parse_args()
    provisioner = EnvironmentProvisioner()

    if args.command == "new":
        provisioner.new_environment(args.track, args.name)

if __name__ == "__main__":
    main()
```

How to Run the CLI Tool:

1. **Save the code:** Save the code above as perle_env.py.
2. **Run from terminal:**
    a. To provision a demo environment:

Bash
python perle_env.py new --track=demo

    b. To provision a customer environment with a specific name:

Bash
python perle_env.py new --track=customer --name=my-alpha-customer-env

    c. To provision a platform environment:

Bash
python perle_env.py new --track=platform

Output Example:

--- Initiating Environment Provisioning for 'demo' track ---
Environment ID: demo-20250710123456-a1b2c3d4
Description: Short-Term Demo Environment - ephemeral, cost-optimized, rapid.
 [INFO] Writing provisioning details to env_provisioning_logs/demo-20250710123456-a1b2c3d4.log
 [SIMULATION] Initiating AWS CloudFormation/Terraform for demo-20250710123456-

a1b2c3d4 with template: aws_demo_template.json
  [SIMULATION] Configuring CI/CD pipeline for demo-20250710123456-a1b2c3d4 using profile: demo_ci_cd_profile.json
  [SIMULATION] Deploying AI agents configured with policy: demo_agent_policy.json
  [SIMULATION] Provisioning core resources: Lambda, DynamoDB (on-demand), API Gateway
--- Environment 'demo-20250710123456-a1b2c3d4' Provisioning Initiated (simulated) ---
Details logged to: env_provisioning_logs/demo-20250710123456-a1b2c3d4.log
Next steps for a real system:
  1. An Agent Orchestration Service (AOS) agent would pick up this request.
  2. It would trigger the Infrastructure Orchestration Engine (IOE) to provision resources.
  3. CI/CD pipelines would be configured via the PaaS.
  4. Agent-specific configurations would be applied.

This CLI tool serves as the initial human interface for requesting environment provisioning. In a real-world scenario, this request would be picked up by the Agent Orchestration Service (AOS), which would then coordinate with other agents and platform components (IOE, PaaS) to fully provision and configure the requested environment on AWS, adhering to the specific policies and resource types defined for each "track." The use of AWS Bedrock and SageMaker would come into play in training and deploying the AI agents that would execute these provisioning, monitoring, and remediation tasks autonomously.