DevOps Orchestra: A Multi-Agent AI-Powered DevOps Automation System

Purpose:
Automate the **entire DevOps lifecycle** using **AI agents** that collaborate like a team to handle code validation, infrastructure provisioning, testing, deployment, monitoring, and recovery—without requiring manual scripts or human intervention.

How It Works

1. **User pushes code to GitHub** along with a `devops_orchestra.yaml` file that defines deployment preferences (e.g., cloud, environment, strategy).

2. The system's **agents** kick in:

    - **GitOps Agent** detects the push via webhook and triggers the workflow.
    - **Code Analysis Agent** inspects code quality and completeness.
    - **Testing Agent** auto-generates and runs unit tests, integration tests using AI.
    - **Build Agent** checks for `Dockerfile` (generates if missing). It builds the code and pushes the Docker image to Docker Hub/ Amazon ECR.
    - **IaC Agent** provisions cloud infra using Terraform (generates if missing).
    - **Deployment Agent** deploys using strategies like blue-green or canary.
    - **Observability Agent** monitors app health and auto-rolls back if needed.
    - **ChatOps Agent** communicates with users via Slack.

3. Agents communicate and reassign tasks among themselves via the shared event bus (Kafka).

What Makes It Special

- Uses **open-source LLMs** (e.g., LLaMA2, Mistral) to generate missing DevOps artifacts.
- Works even when the user only provides **partial code** (no Docker/Terraform).
- Smart enough to **auto-heal**, **self-correct**, or **ask for user intervention** when needed.
- Supports **dynamic coordination** between agents for efficiency.
- Ideal for a **cloud-native** app.

Use-Cases:

1. Push-to-Deploy: Full Automation on Code Push

Actors: Developer, GitOps Agent, Code Analysis Agent, Testing Agent, Build Agent, IaC Agent, Deployment Agent, Observability Agent.
Scenario: A developer pushes new backend code to GitHub. DevOps Orchestra autonomously builds, tests, provisions, deploys, and monitors the application.
Flow:
1. GitOps Agent receives the GitHub webhook on code push.
    - Parses the devops_orchestra.yaml file (if present) or uses the default configuration.
    - Publishes a code_push event to the event bus.
2. Code Analysis Agent listens for the code_push event.
    - Lints and analyzes the code for errors or missing modules.
    - Publishes code_analysis_result event (with pass/fail + warnings).
3. Testing Agent listens for code_analysis_result.
    - If valid, generates and runs unit/integration tests.
    - Publishes test_results event (pass/fail, coverage report).
4. Build Agent listens for test_results.
    - If tests passed, builds Docker image (or generates Dockerfile if missing) and pushes it to Docker Hub/ AWS ECR.
    - Publishes build_ready event with image URL or artifact.
5. IaC Agent listens for build_ready.
    - Checks for existing Terraform/IaC files.
    - Generates and applies infrastructure code (EC2, RDS, etc.).
    - Publishes iac_ready event with provisioned resources info.
6. Deployment Agent listens for iac_ready.
    - Deploys the new version using blue-green strategy (or as specified).
    - Publishes deployment_triggered or deployment_failed.
7. Observability Agent listens for deployment_triggered.
    - Monitors health, latency, error rates.
    - If failure is detected, triggers auto rollback.
    - Publishes observability_alert or rollback_event.
8. ChatOps Agent listens to all events.
    - Notifies the developer in Slack about:
        - Linting/test results
        - Build status
        - Infra provisioning
        - Deployment success/failure
        - Rollback or alerts

2. Code Quality Gate Before Build

Actors: Developer, Code Analysis Agent, ChatOps Agent
Scenario: Code is pushed but contains anti-patterns or incomplete modules.
Flow:

- GitOps Agent triggers the workflow.
- Code Analysis Agent reviews syntax, structure, and style.
- Flags issues like missing modules or inconsistent structure.
- ChatOps Agent sends feedback via Slack with suggestions.
- Deployment halts until issues are fixed.

3. Missing Dockerfile – Build Agent Intervention

Actors: Build Agent, ChatOps Agent
Scenario: The Project lacks a Dockerfile.
Flow:

- Build Agent inspects the codebase.
- Based on language/framework (e.g., Node.js, Django), generates a Dockerfile using AI.
- Sends it to the global state (Kafka) for other agents to consume.
- Alerts user via ChatOps Agent.

4. Missing Terraform Files – IaC Agent Bootstrapping

Actors: IaC Agent, ChatOps Agent
Scenario: The repository doesn't contain `main.tf` or infrastructure config.
Flow:

- IaC Agent generates Terraform scripts from `devops_orchestra.yaml`.
- Provisions cloud infra (e.g., S3, EC2, Lambda, VPC) based on configuration.
- Stores generated files and alerts via ChatOps Agent.

5. Intelligent Test Generation for Legacy Code

Actors: Code Analysis Agent, Testing Agent, ChatOps Agent
Scenario: An old codebase lacks test coverage.
Flow:

- Code Analysis Agent maps out the code structure.
- Testing Agent uses AI to generate unit/integration tests.

- Runs tests in an isolated container.
- Sends test reports to Slack via ChatOps Agent.

6. Automated Canary Release With Auto Rollback

Actors: Deployment Agent, Observability Agent, ChatOps Agent
Scenario: A New version of the app is released.
Flow:

- Deployment Agent performs canary release (small % of traffic gets new version).
- Observability Agent monitors metrics (latency, error rates).
- If issues detected, rollback is triggered automatically.
- ChatOps Agent alerts team.

7. Chat-Based Deployment Command

Actors: ChatOps Agent, Deployment Agent
Scenario: Dev wants to deploy a hotfix from Slack.
Flow:

- Developer types `/deploy frontend hotfix-v2` in Slack.
- ChatOps Agent parses the intent.
- Sends deployment instruction to Deployment Agent.
- Agent fetches latest Git revision, builds, and deploys.
- Reports back in the same Slack thread.

8. Real-Time Issue Detection and Healing

Actors: Observability Agent, Deployment Agent, ChatOps Agent
Scenario: A frontend microservice crashes due to memory leaks.
Flow:

- Observability Agent detects anomalous CPU/memory usage.
- Notifies global event bus.
- Deployment Agent rolls back to the last healthy version.
- ChatOps Agent summarizes crash logs and notifies team.

9. Pre-Merge Deployment Validation

Actors: GitOps Agent, Code Analysis Agent, Testing Agent, ChatOps Agent
Scenario: A pull request is raised for staging deployment.
Flow:

- GitOps Agent triggers on PR.
- Code Analysis Agent inspects code for risks.
- Testing Agent generates and runs unit/ integration tests.
- If all tests passes, ChatOps Agent posts a PR comment: “Safe to merge and deploy.”

10. Frontend and Backend Coordinated Deployments

Actors: GitOps Agent, Build Agent, Deployment Agent
Scenario: Frontend and backend are in the same repo.
Flow:

- GitOps Agent watches for commits in both `/frontend` and `/backend`.
- Build Agent creates two separate images and pushes them to Docker Hub/ Amazon ECR.
- Deployment Agent coordinates rollout such that frontend and backend are deployed together using `depends_on` logic in `devops_orchestra.yaml`.

`devops_orchestra.yaml` Template:

```
# Required: Project metadata
project:
  name: my-app
  language: python
  framework: flask
  repo: https://github.com/user/my-app

# Required: Build settings
build:
  tool: docker         # options: docker, maven, npm, poetry, custom
  context: .
  dockerfile: Dockerfile

# Required: Test configuration
testing:
  enabled: true
  framework: pytest     # options: pytest, unittest, jest, mocha, etc.
  command: pytest tests/

# Required: Deployment
deployment:
  type: cloud           # options: cloud, on-premise, hybrid
  provider: aws         # options: aws, gcp, azure, custom
  strategy: blue-green  # options: blue-green, canary, rolling
  region: us-west-2
  services:
    - name: api-server
      runtime: ec2       # options: ec2, lambda, kubernetes, etc.
      port: 5000
      env:
        - key: ENV
          value: production
        - key: DB_URL
          value: postgres://...

# Required: Infrastructure as Code (IaC)
infrastructure:
  tool: terraform        # options: terraform, cloudformation, ansible
  path: infra/           # path to IaC scripts

# Optional: Secrets (recommend using secret manager)
secrets:
  manager: aws_secrets_manager
```

```
  keys:
    - DB_PASSWORD
    - API_KEY

# Optional: Rollback policy
rollback:
  enabled: true
  threshold:
    cpu: 90
    errors: 5
    duration: 5m

# Optional: Observability
observability:
  enabled: true
  tools:
    - prometheus
    - grafana
    - cloudwatch
  alerts:
    - type: latency
      threshold_ms: 500
    - type: error_rate
      threshold: 2%

# Optional: ChatOps settings
chatops:
  enabled: true
  platform: slack
  channel: "#devops-orchestra"
  notify_on:
    - deployment_success
    - deployment_failure
    - rollback_triggered
```

Here are the key **assumptions** made in the **DevOps Orchestra** system design:

System-Level Assumptions

1. **GitHub as Source of Truth**
   All project codebases are stored in GitHub repositories. No support for other VCS like GitLab/Bitbucket unless extended.

2. **Presence of `devops_orchestra.yaml`**
   Users must include a configuration file in the root of their repository specifying:
   - Environment (cloud/on-prem)
   - Infrastructure preferences
   - Deployment strategy
     If missing, default behavior or agent intervention kicks in.

3. **LLMs Available Locally**
   The system uses open-source LLMs (like LLaMA 2, Code LLaMA, or Mistral) instead of commercial APIs like GPT-4 due to cost constraints.

4. **Kafka or Event Bus Available**
   A working event bus (e.g., Apache Kafka or Redis Streams) is assumed to be set up to facilitate inter-agent communication.

5. **Microservice-Friendly Codebase**
   The system assumes the code is modular (e.g., frontend, backend) so that agents can independently build and deploy parts of the system.

Agent-Level Assumptions

1. **Build Agent Assumptions**

   - Can correctly identify the language/framework (Node, Django, Flask, etc.).
   - Has access to a template library for Dockerfile generation.

2. **IaC Agent Assumptions**

   - The YAML contains sufficient information (cloud provider, region, service type) for Terraform file generation.
   - Users are okay with basic default infrastructure if configuration is minimal.

3. **Code Analysis Agent Assumptions**

    - Code is at least syntactically correct and the project structure is identifiable.
    - Errors can be detected using static analysis + LLM heuristics.

4. **Testing Agent Assumptions**
    - Can generate tests using AI even without user-provided ones.
    - Enough function/method structure is available in the code for test generation to be meaningful.

5. **Deployment Agent Assumptions**

    - Code has been containerized or can be containerized before deployment.
    - Infra provisioning is already done successfully before it starts.

6. **Observability Agent Assumptions**

    - Application logs, metrics, and health endpoints (like `/health`, `/metrics`) are available or inferred.
    - Sufficient permissions to perform rollback or restart actions.

7. **ChatOps Agent Assumptions**

    - User has integrated Slack via webhook or bot.
    - Users can be identified and authenticated within the chat platform.

User Assumptions

1. **Basic YAML Proficiency**
   Users can define the required fields correctly in `devops_orchestra.yaml`. Defaults are used if some fields are missing.

2. **Push Triggers Deployment**
   Users are aware that pushing to certain branches (like `main`, `release`) automatically triggers the DevOps pipeline.

3. **Cloud Credentials Available**
   If deploying to cloud, credentials (via GitHub Secrets or environment variables) are accessible to the IaC and Deployment Agents.

4. **Failure Handling Is Acceptable**
   The system may skip or rollback if:

   - Infrastructure provisioning fails
   - Tests fail
   - App health degrades after deployment

LLM Behavior Assumptions

1. **Sufficient Context Window**
   LLMs can process a representative portion of the codebase, but may not handle extremely large monorepos in one go.

2. **Trust in AI-Generated Artifacts**
   The system assumes the AI-generated Dockerfile, Terraform scripts, and test cases are safe defaults unless overridden.

Full Project Structure for Peer-to-Peer DevOps Orchestra:

```
devops-orchestra/
├── agents/
│   ├── gitops_agent/
│   │   ├── main.py
│   │   ├── event_handler.py
│   │   ├── config_parser.py
│   │   └── README.md
│   ├── build_agent/
│   │   ├── main.py
│   │   ├── dockerfile_generator.py
│   │   ├── event_handler.py
│   │   └── README.md
│   ├── iac_agent/
│   │   ├── main.py
│   │   ├── terraform_generator.py
│   │   ├── event_handler.py
│   │   └── README.md
│   ├── code_analysis_agent/
│   │   ├── main.py
│   │   ├── static_analyzer.py
│   │   ├── event_handler.py
│   │   └── README.md
│   ├── testing_agent/
│   │   ├── main.py
```

```
│   │   ├── test_generator.py
│   │   ├── test_runner.py
│   │   ├── event_handler.py
│   │   └── README.md
│   ├── deployment_agent/
│   │   ├── main.py
│   │   ├── deployer.py
│   │   ├── event_handler.py
│   │   └── README.md
│   ├── observability_agent/
│   │   ├── main.py
│   │   ├── monitor.py
│   │   ├── rollback_handler.py
│   │   └── README.md
│   └── chatops_agent/
│       ├── main.py
│       ├── slack_bot.py
│       ├── command_parser.py
│       └── README.md
│
├── shared/
│   ├── event_bus/
│   │   ├── kafka_producer.py
│   │   ├── kafka_consumer.py
│   │   ├── topics.py
│   │   └── event_schema.py
│   ├── utils/
│   │   ├── logger.py
│   │   ├── config_loader.py
│   │   └── file_utils.py
│   └── llm/
│       ├── model_wrapper.py
│       ├── prompts/
│       │   ├── docker_prompt.txt
│       │   ├── terraform_prompt.txt
│       │   ├── test_prompt.txt
│       │   └── analysis_prompt.txt
│
├── examples/
│   └── fullstack-project/
│       ├── frontend/
│       ├── backend/
│       └── devops_orchestra.yaml
```

```
├── infrastructure/
│   ├── generated/
│   │   ├── main.tf
│   │   ├── ec2.tf
│   │   └── outputs.tf
│   └── base/
│       ├── provider.tf
│       └── variables.tf

├── tests/
│   ├── test_gitops_agent.py
│   ├── test_build_agent.py
│   ├── test_testing_agent.py
│   ├── test_event_bus.py
│   └── test_utils.py

├── docker-compose.yml
├── requirements.txt
├── devops_orchestra.yaml   # Optional default config
└── README.md
```

Root Directory (`devops-orchestra/`)

`README.md`

- Project overview and setup guide
- Contains: agent roles, how to run Kafka + agents, usage examples
- **Essential for onboarding and documentation**

`requirements.txt`

Lists common Python dependencies.

`docker-compose.yml`

- Defines and spins up:
    - Kafka

    - Zookeeper
    - Agents (e.g., GitOps Agent, Build Agent)

- Used for **local development and testing**

`devops_orchestra.yaml`

- Sample orchestration config (if user repo doesn't provide one)
- Used to simulate deployments or provide fallback defaults

`agents/` (Contains One Folder per Agent)

Example: `gitops_agent/`

`main.py`

- Entry point for the GitOps Agent
- Starts the webhook server or event listener
- Subscribes to push events and publishes `code_push` events to Kafka

`event_handler.py`

- Handles incoming GitHub push events
- Parses branch, repo, changed files

`config_parser.py`

- Reads and validates the `devops_orchestra.yaml` file
- Sends its parsed output to the event bus

`README.md`

- Explains GitOps Agent's responsibilities and usage

All other agent folders (e.g., `build_agent/`, `iac_agent/`, etc.) follow the same structure:

`main.py`

- Initializes Kafka consumer for that agent's topic
- Subscribes to relevant events like `code_push`, `build_ready`, `infra_ready`

`*_generator.py`, `*_handler.py`, etc.

- Agent-specific logic:
    - **Build Agent:** `dockerfile_generator.py` creates Dockerfiles using LLM
    - **IaC Agent:** `terraform_generator.py` creates Terraform files
    - **Testing Agent:** `test_generator.py` creates tests, `test_runner.py` executes them
    - **Deployment Agent:** `deployer.py` performs blue-green/canary deployments
    - **Observability Agent:** `monitor.py` reads logs/metrics, `rollback_handler.py` performs rollback
    - **ChatOps Agent:** `slack_bot.py` connects to Slack, `command_parser.py` maps commands to actions

---

`shared/`

`event_bus/`

`kafka_producer.py`

- Publishes events to Kafka topics like:
    - `build_ready`
    - `deployment_triggered`

    - `rollback_event`

`kafka_consumer.py`

- Subscribes to one or more Kafka topics
- Dispatches incoming events to appropriate handlers

`topics.py`

- Defines standard Kafka topic names to ensure consistency across agents

`event_schema.py`

- Defines and validates event structure (e.g., JSON schema) to ensure clean inter-agent communication

---

`utils/`

`logger.py`

- Standard logging setup for all agents
- Formats logs with timestamp, agent name, and log level

`config_loader.py`

- Loads and validates YAML config files (`devops_orchestra.yaml`)

`file_utils.py`

- Utilities to check if a file exists, read/write files, copy templates

---

`llm/`

`model_wrapper.py`

- Abstraction layer for calling LLaMA, Mistral, etc.
- Supports both local inference and Hugging Face API
- Used by agents like Build, IaC, and Testing

`prompts/`

- Stores reusable prompt templates

Examples:

- `docker_prompt.txt`:
  "Generate a Dockerfile for a Flask app with port 5000..."

- `terraform_prompt.txt`:
  "Generate Terraform code to deploy an EC2 instance..."

- `test_prompt.txt`:
  "Generate unit tests for this Python function..."

- `analysis_prompt.txt`:
  "Perform code quality analysis for this React component..."

`examples/`

`fullstack-project/`

`frontend/` and `backend/`

- Sample app code (React + Flask, etc.)

`devops_orchestra.yaml`

- Real config used to trigger the full DevOps pipeline.

`infrastructure/`

`generated/`

- Contains Terraform files **generated by the IaC Agent** on the fly

Files:

- `main.tf`, `ec2.tf`, `outputs.tf`: auto-generated based on YAML config

`base/`

- Contains reusable modules or manually written Terraform templates
- Used as fallback or as scaffolding for generation

Files:

- `provider.tf`: AWS provider config
- `variables.tf`: shared variables for all infra

`tests/`

Contains **unit and integration tests** for agents and shared code.

Files:

- `test_gitops_agent.py`: Tests for GitOps event handling
- `test_build_agent.py`: Dockerfile generation, edge cases
- `test_testing_agent.py`: AI test generation, runner logic
- `test_event_bus.py`: Kafka message flow
- `test_utils.py`: Utility module test.

  Run with: `pytest tests/`