**High-Level Components and Architecture**

Frontend (CLI Microservice Client): GoLang (using github.com/urfave/cli for CLI, github.com/rabbitmq/amqp091-go for RabbitMQ producer/consumer).

Backend Microservice: Python 3.10+ with:

Orchestration: LangGraph for execution graphs.

Agents: AutoGen for multi-agent collaboration within graph nodes.

LLMIntegration: Configurable via environment variables or YAML config:

* + Commercial: LangChain/OpenAI/Grok wrappers.
  + Local: Ollama (via langchain\_ollama) or Hugging Face Transformers for on-prem models.
  + Toggle: Use a config flag (e.g., LLM\_PROVIDER=groq vs. LLM\_PROVIDER=ollama).

Message Broker: RabbitMQ (for decoupling frontend and backend; queues for requests, results, and status updates).

Containerization for Offensive Tools: Docker (via docker-py in Python) to spin up ephemeral containers (e.g., Kali Linux image) for tools like nmap (python-nmap) and Metasploit (pymetasploit3). No offensive code runs outside containers.

Additional Libs: pika (RabbitMQ), fastapi (optional internal API for health checks), pydantic (state validation), docker (client).

Configuration Management: YAML files (e.g., config.yaml) will be used for LLM keys, RabbitMQ creds, and deployment modes. Environment vars for secrets.

Microservice Components:

* + CLI Client: User-facing; sends recon requests (e.g., IP) to RabbitMQ queue (recon\_requests). Listens to recon\_results for reports/shell info. Can run locally or as a containerized binary.
  + Backend Service: Consumes from recon\_requests, executes LangGraph workflow (with AutoGen agents), containerizes tool runs, and publishes results to recon\_results. Scalable horizontally (multiple instances consuming from the queue).

CLI publishes JSON message to RabbitMQ (e.g., {"ip": "192.168.1.1"). Backend consumes, initializes LangGraph state, and uses configured LLM for agents. Spin up Docker container (e.g., kalilinux/kali-rolling), exec tool (e.g., container.exec\_run('nmap -A ' + ip)), capture output, destroy container. AutoGen agents analyze in graph nodes; publish a report or shell status back.

**Phase 1: Core Setup and Local Development (2-3 weeks)**

* Set up repo structure: /cli (Go), /backend (Python), /docker (images/scripts), /config (YAML).
* Implement basic RabbitMQ integration: CLI publisher/consumer; backend consumer that echoes messages.
* Build the LangGraph skeleton: Define the state and add a planning node using AutoGen (initially use a dummy LLM).
* Integrate LLM toggle: Test with Ollama locally and a commercial API.
* Containerize offensive mock: Add a node that runs echo 'scan' in a Docker container.
* Test: Run CLI locally, verify message flow, and container exec.

**Phase 2: Recon Logic and Tool Integration (3-4 weeks)**

* Expand LangGraph: Add nodes for scanning, vuln assessment, and exploitation (all containerized).
* Integrate tools: Nmap/Metasploit in Kali containers. Handle shell attempts (e.g., Metasploit reverse shell listener in container; proxy output if successful).
* Enhance AutoGen: Agents for planning, analysis.
* Add fallback: If the shell fails, generate a report from the state.
* Config for LLMs: Ensure seamless switch (e.g., commercial for speed, local for privacy).

**Phase 3: Microservice Polish and Security (2-3 weeks)**

* Add RabbitMQ features: Queues, acks, and dead-letter queues for errors.
* Implement logging/monitoring: Backend exposes /health endpoint; log container outputs.
* Security audit: Ensure no host exec of tools; add rate limiting on queues.
* CLI enhancements: Progress updates via status queue; handle timeouts.

**Phase 4: Deployment and Testing (1-2 weeks)**

* See Deployment Strategies below.
* End-to-end testing: Safe IPs, mock containers.
* Performance: Stress test with multiple requests.

**Deployment Strategies**

Design for dual-mode: Cloud (scalable, managed) and On-Premise (self-hosted, private).

Shared foundation for both deployment strategies:

* Containerize everything: Dockerfile for CLI (as binary), backend (Python app), RabbitMQ (official image), Ollama.
* Docker Compose for local/dev: Single docker-compose.yml to spin up all services.
* Kubernetes manifests for prod: Deploy as pods (e.g., backend deployment, RabbitMQ statefulset).

On-Premise Deployment:

* Hardware: Local machine or VM with Docker/K8s.
* Setup: Docker Compose for simplicity: docker-compose up starts RabbitMQ, backend, and Ollama.
* LLM: Local models via Ollama container (e.g., pull Llama3).
* Access: CLI runs on user’s machine, connects to local RabbitMQ.
* Scaling: Manual replicas of backend containers.

Cloud Deployment:

* Platform: Azure (VMs/AKS).
* Managed Services: Amazon MQ (for RabbitMQ).
* LLM: Commercial APIs (config points to endpoints).
* Orchestration: Kubernetes (AKS) for auto-scaling backend pods based on queue length.
* CI/CD: Not considered as of right now, will be addressed later in the development.
* Monitoring: Not considered as of right now, will be addressed later in the development.