# Architecture Overview

We propose a **microservices-based architecture** for an AI agent platform. At a high level, the system consists of a **React frontend** and a **Golang backend** (API server), supported by a **workflow execution engine**, **agent runtime sandbox**, a **persistent database (PostgreSQL)**, and a **task queue/broker**. Users log in through the frontend, configure **workflows** composed of agent steps (like “browse web”, “run shell”, “edit code”, or “call LLM”), and monitor execution. The Golang backend exposes REST/gRPC APIs (and websockets for live updates) to manage workflows, agents, and executions. An orchestration service drives multi-step workflows by enqueueing tasks into a job queue. Worker processes pull tasks, spawn sandboxed containers for execution, and report results back to the backend. Throughout, strict security (sandboxing, secrets management, auth) and scalability (container orchestration, load balancing) are enforced to support a multi-tenant SaaS deployment.

In more detail, the key components are:

* **Frontend (React)**: A modern Single-Page App for workflow creation, editing, and monitoring. It communicates with backend APIs (HTTP/REST or GraphQL) and subscribes to updates (via WebSockets or Server-Sent Events) to display live logs and agent outputs.
* **API Gateway / Backend (Golang)**: Implements business logic and the public API. It handles user requests, performs authentication/authorization, reads/writes to the database, enqueues tasks, and coordinates execution state. It also integrates with LLM APIs (OpenAI, Anthropic, Ollama) and other external services.
* **Workflow Orchestrator**: A component (which may be part of the backend or a separate service) that interprets user-defined workflows (likely stored as JSON/DAG in the database) and enqueues each step as a job. It tracks progress and handles branching or conditional logic.
* **Task Queue / Job Broker**: A distributed queue (e.g. Redis-backed [Asynq]) that holds pending agent tasks. Multiple worker nodes subscribe to the queue and execute tasks concurrently, enabling high availability and horizontal scaling[[1]](https://github.com/hibiken/asynq" \l ":~:text=,processed%20concurrently%20by%20multiple%20workers).
* **Worker / Agent Runtime**: Each worker pulls tasks and executes them in an isolated **sandbox**. We propose using container-based isolation (e.g. Docker or Kubernetes pods) to run arbitrary code and commands safely[[2]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=The%20OpenHands%20Docker%20Runtime%20is,without%20risking%20the%20host%20system). The worker loads the relevant agent tools (e.g. a headless browser for web, a shell, Git access, or LLM client libraries) and performs the action, then returns observations/results.
* **Sandbox Runtime Environment**: A reusable Docker image or runtime environment that includes necessary language runtimes (Python/Node), libraries, and the “agent client”. This image is used to spin up ephemeral containers for each agent’s action. As in OpenHands, the runtime client inside the container handles commands, file I/O, etc., while the backend communicates with it over a REST or gRPC interface[[3]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=components%20like%20a%20bash%20shell,the%20OpenHands%20backend%20as%20observations).
* **Persistent Database (PostgreSQL)**: Central store for users, workflows, agent configurations, execution logs, and metadata. Workflows and agent definitions may be stored as relational tables or JSONB fields, enabling versioning and queries. Execution logs (stdout, stderr, tool outputs) are also persisted for auditing and monitoring.
* **Authentication/Authorization**: A service or module to manage user accounts, roles, and tenant isolation. Likely using JWT/OAuth2. Each request checks user permissions, and all data/models are tagged by tenant ID to enforce multi-tenancy[[4]](https://articles.wesionary.team/building-a-multi-tenant-architecture-in-golang-a-practical-guide-8ee066436678?gi=9a3f54dc9af1" \l ":~:text=The%20key%20to%20multi,That%E2%80%99s%20the%20core%20concept).
* **LLM Integration Service**: A helper module (or library code) for calling LLMs. It abstracts differences between providers: OpenAI and Anthropic (Claude) via their HTTP APIs (requiring API keys), and local models (via Ollama) via a local HTTP endpoint or subprocess[[5]](https://dshills.medium.com/go-ollama-simple-local-ai-3a89be4bfbaf" \l ":~:text=The%20API%E2%80%99s%20URL%20is%20typically,http%3A%2F%2Flocalhost%3A11434%2Fapi%2Fchat).
* **Monitoring & Logging**: Centralized logging (e.g. ELK, or cloud logging) collects all agent execution logs and platform events. Metrics (Prometheus, Grafana) track system health, queue lengths, container usage, and LLM API quotas. Tracing (OpenTelemetry/Jaeger) can be used for request tracing across services.
* **DevOps/Infrastructure**: All components are containerized (Docker) and deployed on Kubernetes (or similar) for scalability. CI/CD pipelines (e.g. GitHub Actions) build and push container images. In production, use managed services: e.g. AWS EKS/GKE for Kubernetes, AWS RDS (Postgres), Redis cluster for queue, Vault or AWS Secrets Manager for secrets.

Below we detail each aspect of the architecture.

## Component Breakdown

* **React Frontend**: Serves the web UI. Manages user interactions:
* **Workflow Builder**: A drag-and-drop or form interface where users define agents and steps. Could use a visual workflow library (e.g. React Flow).
* **Execution Dashboard**: Shows running workflows, status (running/succeeded/failed), and logs. Uses WebSocket/SSE to update in real time.
* **Agent Configuration UI**: For defining agent personas, selecting LLM models (OpenAI/Claude/Ollama), and uploading code repos or documents.
* The frontend talks to the Golang API (via HTTPS/REST) and may use WebSockets for live logs.
* **API Gateway / Backend (Golang)**: Exposes endpoints like /api/workflows, /api/agents, /api/executions. Responsibilities:
* **User Management**: Register/login, issue JWT tokens (or integrate with SSO/OAuth).
* **Tenant Routing**: Identify user’s tenant (multi-tenant ID) and scope queries.
* **Workflow CRUD**: Create/read/update/delete workflows stored in Postgres. Each workflow is a sequence or DAG of steps.
* **Execution Control**: Start a workflow execution; enqueue initial tasks to the queue. Track execution state in the DB.
* **Agent Tools Management**: Maintain a registry of available tools (browser, shell, file editor, API caller). Step definitions reference these tools.
* **LLM API Adapters**: Provide functions to call OpenAI/Claude/Ollama. These might be background calls from workers (see below).
* **WebSocket Server**: For pushing execution logs and status changes to subscribed frontend clients. Implemented via gorilla/websocket or similar.
* **Security**: Validate all requests, sanitize inputs. Enforce rate limits and usage quotas per tenant.
* **Workflow Orchestrator / Execution Engine**: May be a separate microservice or embedded in the backend.
* **Task Generator**: Takes a workflow definition (possibly a JSON or directed graph) and generates sequential tasks. For example, step 1 might be “Query LLM to plan web browsing steps,” then step 2 “Worker runs headless browser” etc.
* **State Management**: Keeps track of which steps are done, which are pending, and any conditionals or loops. If workflows support conditional logic (e.g., if results contain X then do Y), the orchestrator evaluates those conditions (potentially using LLM or custom logic).
* **Enqueueing**: Creates tasks for workers by serializing step information (tool type, parameters, context) and pushing into the queue. Could use Redis Streams or a library like [Asynq](https://github.com/hibiken/asynq)[[1]](https://github.com/hibiken/asynq" \l ":~:text=,processed%20concurrently%20by%20multiple%20workers).
* **Coordination**: Listens for completion events (via database updates or pub/sub) and moves to the next step or finishes the workflow.
* **Task Queue / Broker**: We suggest a Redis-backed queue (e.g. Asynq or Bull). Features:
* **Distributed Workers**: Multiple worker processes (in Kubernetes pods) can pull tasks in parallel. This ensures horizontal scaling and redundancy.
* **Scheduling & Retries**: The queue supports retries, timeouts, and delayed tasks (e.g. wait 5 minutes).
* **High Availability**: Redis or message broker in a cluster mode to avoid single point of failure.
* **Task Definition**: A task message includes tenant ID, workflow ID, step ID, tool name (e.g. “BrowserAction”), parameters (e.g. URL to fetch, shell command to run), and a reference to the LLM prompt if needed.
* **Worker Nodes / Agent Runtime**: Each worker is a Go process that:
* **Pulls a Task**: It reads tasks from the queue, deserializes them.
* **Spawns a Sandbox**: For security, each task runs in an isolated container or sandbox. For example, the worker might invoke the Docker API (or containerd) to run a new container from a pre-built runtime image.
* **Executes Tools**: Inside the container, the agent client program executes the requested action:
  + **Web Browsing**: Launch a headless browser (e.g. Puppeteer or Chrome via Go bindings) and return HTML/text.
  + **Shell Commands**: Run a bash/zsh process on the file system, capturing stdout/stderr.
  + **File Modifications**: Perform git operations or text edits. The container may mount a volume with the user’s codebase or code can be pulled from a repo.
  + **API Calls**: Use HTTP client libraries within the container.
  + **LLM Calls**: The worker either calls external LLM APIs directly (OpenAI/Claude via HTTPS) or calls the local Ollama service (e.g. http://localhost:11434/api/chat)[[5]](https://dshills.medium.com/go-ollama-simple-local-ai-3a89be4bfbaf" \l ":~:text=The%20API%E2%80%99s%20URL%20is%20typically,http%3A%2F%2Flocalhost%3A11434%2Fapi%2Fchat).
* **Report Results**: Collect output from each tool and send observations back to the backend. This may be done by calling a backend API endpoint or writing to the database. The backend then marks the step as completed.
* **Stateful Context**: The worker/container may maintain a working directory, allowing later steps to access files created or modified in earlier steps (but use fresh containers for full isolation).
* **Sandbox Runtime (Container Images)**: We maintain a Docker image for the agent runtime, e.g. openhands/runtime-style. This image includes:
* Common language runtimes (Python, Node, Java, etc.) for running tools/scripts.
* A minimal Linux environment with git, bash, etc.
* The “runtime client” code (a Go or Python binary that listens for commands via REST or a UNIX socket).
* Browser engine (headless Chrome) if web browsing is needed.
* Required CLI tools (curl, wget, docker CLI if needed, etc.)
* Plugins for code editing (e.g. common IDE connectors) if needed.

We tag runtime images by version and lock hash for reproducibility (similar to OpenHands’ tagging scheme[[6]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=Image%20Tagging%20System)). For example, agent-runtime:v1.2\_python3.12-nodejs16 might be a tag.

* **Persistent Database (PostgreSQL)**: Stores:
* **Users/Tenants**: tables for users and organizations (tenants). Each user has a foreign key to tenant. Could use JWT with tenant claim.
* **Workflows**: workflows table (id, tenant\_id, name, description, created\_at) and workflow\_steps table (id, workflow\_id, step\_index, step\_type, parameters JSONB). The steps might include references to agent ID or tool ID.
* **Agents**: agents table defining agent settings (LLM model, temperature, etc.) and associated tools/plugins.
* **Executions**: workflow\_executions (id, workflow\_id, status, start\_time, end\_time) and step\_executions (id, execution\_id, step\_id, status, start, end, output JSONB). We store logs/outputs in a field (e.g. JSONB or TEXT) for viewing later.
* **Logs**: Optionally, a logs table for streaming log entries if we need fine-grained auditing.
* **External Services**: llm\_credentials table for storing API keys or tokens (encrypted).

All tables include tenant\_id for multi-tenancy. We can use Postgres Row-Level Security (RLS) policies to ensure users only see their own data.

* **Authentication/Authorization**: Use JWT tokens signed by a private key or integrate an identity provider (Auth0, Keycloak). The frontend sends a JWT with each request. The backend validates and checks user roles. Every API call verifies tenant\_id to isolate data between customers. For example, a user’s JWT might include {user:123, tenant:orgA}, and every DB query uses WHERE tenant\_id = 'orgA'.
* **Integration Layer (LLMs and APIs)**:
* **OpenAI GPT**: Use OpenAI’s official Go SDK or call https://api.openai.com/v1/chat/completions with an API key. Parameterize model (e.g. gpt-4o, gpt-3.5-turbo) and prompt. Handle rate limits.
* **Anthropic Claude**: Similarly call Anthropic’s HTTP API (/v1/completions or /v1/chat/completions), as described in their docs. Provide API keys and version header. The backend can wrap these calls in a Go client.
* **Ollama (Local LLM)**: We can run Ollama server locally (supports ollama run <model> which opens a local API on port 11434)[[5]](https://dshills.medium.com/go-ollama-simple-local-ai-3a89be4bfbaf" \l ":~:text=The%20API%E2%80%99s%20URL%20is%20typically,http%3A%2F%2Flocalhost%3A11434%2Fapi%2Fchat). The backend (or worker) can POST JSON to http://localhost:11434/api/chat to get responses. Alternatively, spawn an ollama subprocess from Go (using os/exec), but the HTTP approach is cleaner.

In all cases, calls to LLMs should be done asynchronously (as part of tasks) and support streaming if needed. We should cache or rate-limit as needed, and handle failures gracefully (retries, fallback to smaller model).

* **Logging and Monitoring**:
* **Logs**: Each service logs to stdout. Use structured logging (JSON) with fields like tenant\_id, workflow\_id, step\_id. Centralize with ELK or a logging service.
* **Metrics**: Instrument Golang code and workers with Prometheus metrics (request latency, error rates, queue lengths, LLM API usage). Deploy Prometheus + Grafana for dashboards.
* **Tracing**: Use OpenTelemetry to trace requests across services (e.g. from API call through worker to container and LLM call) for debugging slowdowns.

## Database Schema Suggestions

Below is an illustrative Postgres schema outline (pseudocode). One can expand as needed.

-- Users and Tenants  
CREATE TABLE organizations (  
 id UUID PRIMARY KEY,  
 name TEXT NOT NULL UNIQUE  
);  
CREATE TABLE users (  
 id UUID PRIMARY KEY,  
 org\_id UUID REFERENCES organizations(id),  
 username TEXT UNIQUE NOT NULL,  
 password\_hash TEXT NOT NULL,  
 role TEXT NOT NULL, -- e.g. 'admin', 'user'  
 created\_at TIMESTAMP NOT NULL DEFAULT now()  
);  
  
-- Workflows  
CREATE TABLE workflows (  
 id UUID PRIMARY KEY,  
 org\_id UUID REFERENCES organizations(id),  
 name TEXT NOT NULL,  
 description TEXT,  
 created\_by UUID REFERENCES users(id),  
 created\_at TIMESTAMP NOT NULL DEFAULT now()  
);  
CREATE TABLE workflow\_steps (  
 id UUID PRIMARY KEY,  
 workflow\_id UUID REFERENCES workflows(id) ON DELETE CASCADE,  
 step\_index INTEGER NOT NULL, -- order of step  
 tool TEXT NOT NULL, -- e.g. 'browser', 'shell', 'llm'  
 parameters JSONB NOT NULL, -- tool-specific params  
 created\_at TIMESTAMP NOT NULL DEFAULT now()  
);  
  
-- Agents/Tools (optional separate config)  
CREATE TABLE agents (  
 id UUID PRIMARY KEY,  
 org\_id UUID REFERENCES organizations(id),  
 name TEXT,  
 model\_provider TEXT NOT NULL, -- e.g. 'openai', 'anthropic', 'ollama'  
 model\_name TEXT NOT NULL, -- e.g. 'gpt-4o', 'claude-2', 'llama2'  
 config JSONB, -- temperature, max\_tokens, etc.  
 created\_at TIMESTAMP NOT NULL DEFAULT now()  
);  
  
-- Workflow Executions and Step Logs  
CREATE TABLE workflow\_executions (  
 id UUID PRIMARY KEY,  
 workflow\_id UUID REFERENCES workflows(id),  
 org\_id UUID REFERENCES organizations(id),  
 status TEXT NOT NULL, -- 'pending', 'running', 'failed', 'completed'  
 started\_at TIMESTAMP,  
 finished\_at TIMESTAMP,  
 created\_at TIMESTAMP NOT NULL DEFAULT now()  
);  
CREATE TABLE step\_executions (  
 id UUID PRIMARY KEY,  
 execution\_id UUID REFERENCES workflow\_executions(id) ON DELETE CASCADE,  
 step\_id UUID REFERENCES workflow\_steps(id),  
 status TEXT NOT NULL, -- 'pending', 'running', 'failed', 'succeeded'  
 started\_at TIMESTAMP,  
 finished\_at TIMESTAMP,  
 output JSONB, -- action output or logs  
 created\_at TIMESTAMP NOT NULL DEFAULT now()  
);  
  
-- LLM/API Credentials (encrypted)  
CREATE TABLE service\_tokens (  
 id SERIAL PRIMARY KEY,  
 org\_id UUID REFERENCES organizations(id),  
 provider TEXT NOT NULL, -- e.g. 'openai', 'anthropic'  
 token TEXT NOT NULL, -- encrypted API key  
 created\_at TIMESTAMP NOT NULL DEFAULT now()  
);

This schema stores workflows and execution history. **Multitenancy** is enforced via org\_id foreign keys, and we should add row-level-security (RLS) so a user can only see rows matching their org\_id[[4]](https://articles.wesionary.team/building-a-multi-tenant-architecture-in-golang-a-practical-guide-8ee066436678?gi=9a3f54dc9af1" \l ":~:text=The%20key%20to%20multi,That%E2%80%99s%20the%20core%20concept). The parameters and output fields as JSONB allow flexible storage of step-specific data (e.g. shell command text, JSON response from LLM, etc.).

## Communication Mechanisms

* **Frontend ↔ Backend**: Use REST/HTTPS (e.g. JSON over HTTP) for most operations (create/edit workflows, fetch status). For real-time updates (log streaming, step status changes), use WebSockets or Server-Sent Events. In Go, the gorilla/websocket package can handle bidirectional updates to the React UI. This lets the UI push a new log line as soon as it’s available (for live monitoring).
* **Backend ↔ Worker**:
* If using a **task queue**, the primary communication is indirect: the backend enqueues a task (JSON payload) in Redis, and workers pop tasks. Alternatively, one could use gRPC between the orchestrator and worker, but using a broker decouples them and improves fault tolerance.
* For quick RPCs (e.g. validating credentials, small lookups), gRPC can be used internally between microservices written in Go. gRPC uses HTTP/2 and protobuf which is faster and more efficient than plain REST/JSON[[7]](https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis" \l ":~:text=Characteristic%20gRPC%20REST%20API%20HTTP,Time%2045%20Minutes%2010%20Minutes). For example, if we split the orchestrator and worker as distinct services, gRPC is ideal.
* **Backend ↔ LLM APIs**: All LLM calls are HTTP/REST. We must handle API keys (via environment or secret store) and follow each provider’s SDK/HTTP spec. For example, to call OpenAI:
* resp, err := openaiClient.CreateChatCompletion(ctx, openai.ChatCompletionRequest{  
   Model: "gpt-4o",  
   Messages: []openai.ChatCompletionMessage{{Role: "user", Content: prompt}},  
  })
* For Anthropic, a similar REST POST to api.anthropic.com/v1/messages with Authorization:Bearer.
* **Backend ↔ Database**: Use a Go ORM or database library (e.g. sqlx or gorm) to query Postgres. Connection pooling and prepared statements should be configured for performance.
* **Secrets Management**: Backend services should not store raw API keys in code or config. Use a secrets manager (AWS Secrets Manager, HashiCorp Vault) and fetch tokens at startup or via environment injection[[8]](https://blog.gitguardian.com/how-to-handle-secrets-in-go/" \l ":~:text=However%2C%20we%20can%27t%20hard,secrets%20in%20the%20Docker%20image)[[9]](https://blog.gitguardian.com/how-to-handle-secrets-in-go/" \l ":~:text=). The service\_tokens table in DB can store encrypted tokens, decrypted only in memory.

## Job Queue / Agent Scheduling

We recommend a Redis-backed task queue like **Asynq** (for Go)[[1]](https://github.com/hibiken/asynq" \l ":~:text=,processed%20concurrently%20by%20multiple%20workers). Key points:

* **Task Creation**: When a workflow execution starts, for each step, the orchestrator pushes a job to the queue with the step details. For multi-step workflows, the orchestrator can either enqueue all steps at once or enqueue the first step and then enqueue the next step when the previous one completes (to handle dependencies).
* **Worker Pools**: Multiple worker instances run (e.g. in Kubernetes Deployment). Each worker listens on the queue. Upon receiving a task, the worker executes it (in a sandbox) and then acknowledges or requeues if failed.
* **Concurrency & Scaling**: Asynq workers use goroutines to handle tasks concurrently. We can scale by running many worker pods. Redis cluster or Sentinels can provide a highly available broker.
* **Retries and Scheduling**: Configure Asynq (or alternative) to retry transient failures (e.g. network errors) and to delay tasks (e.g. retry after 1 minute). This ensures robustness.
* **Workflow Continuation**: When a worker finishes a step, it updates the DB and possibly sends an event (through a pub/sub channel or via the orchestrator) that triggers the next step. For simplicity, the orchestrator can poll or listen on a Redis pub/sub channel for completion messages.

Using a queue separates concerns: the orchestrator focuses on logic, workers on execution. This matches production-grade architectures (task queues “give way to high availability and horizontal scaling”[[1]](https://github.com/hibiken/asynq" \l ":~:text=,processed%20concurrently%20by%20multiple%20workers)).

## Security and Sandboxing

**Sandboxing** is critical: agents run *arbitrary code*, so we must isolate them fully. We propose:

* **Container Isolation**: Each agent action runs in its own Docker container (or Kubernetes pod). This container is ephemeral and destroyed after use. It runs under a non-root user, with minimal privileges.
* **Network Restrictions**: By default, containers have no outgoing internet access except through controlled channels (e.g. allow LLM API calls via host, or only allow browsing limited hosts). Use Kubernetes NetworkPolicies or Docker --network rules. Disable access to AWS metadata endpoints.
* **Filesystem Isolation**: Mount only a workspace directory (read/write). Do not mount host filesystem. Limit disk usage quotas. Use an empty working directory or temp volume.
* **Resource Limits**: Enforce CPU and memory limits on containers to prevent runaway. For example, --cpus=2 --memory=4g.
* **Docker Socket**: If containers need Docker (for building code), use a build service instead of direct host docker access. Avoid mounting /var/run/docker.sock from host into agents, as that breaks isolation.
* **User Privileges**: The container process should not be root. Use a low-privilege user inside the image.
* **Sandbox Runtime**: Similar to OpenHands, use Docker to sandbox untrusted code[[2]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=The%20OpenHands%20Docker%20Runtime%20is,without%20risking%20the%20host%20system). The OpenHands docs note that sandboxing is needed “to prevent malicious code from accessing or modifying the host’s resources”[[10]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=1,from%20affecting%20the%20host%20system).
* **Audit and Logging**: Log all commands executed inside the sandbox and inspect outputs. This helps detect misuse.
* **Secrets Management**: Sensitive information (API keys, credentials) must be injected securely. Don’t bake them into images. Use environment variables at container launch (populated from a secret store or K8s secrets). For example, pass the user’s OpenAI key via an environment variable into the container process; ensure the container image does not have permission to read from other users’ secrets.
* **Multi-tenancy Isolation**: Enforce that containers/agents belonging to Tenant A cannot communicate with Tenant B’s data. This is mainly a software/authorization issue, but also network isolation between tenants can be enforced (separate network namespaces, or labels for Kubernetes network policies).

For secrets specifically, we avoid storing keys in code or database plaintext. We use a secret manager. As noted by GitGuardian, “reading secrets from a file doesn’t meet the needs of a cloud-native environment”[[8]](https://blog.gitguardian.com/how-to-handle-secrets-in-go/" \l ":~:text=However%2C%20we%20can%27t%20hard,secrets%20in%20the%20Docker%20image). Instead, credentials are retrieved at runtime from a manager or injected as Kubernetes Secrets (and not checked into version control). The backend then fetches tokens from Vault or AWS SM, keeping them in memory only.

## LLM and API Integrations

* **OpenAI GPT Integration**: Use OpenAI’s official API (over HTTPS). In Go, the [openai](https://pkg.go.dev/github.com/sashabaranov/go-openai) client or openai-go can be used. Example:
* client := openai.NewClient(os.Getenv("OPENAI\_API\_KEY"))  
  resp, err := client.CreateChatCompletion(ctx, openai.ChatCompletionRequest{  
   Model: "gpt-4o",  
   Messages: []openai.ChatCompletionMessage{  
   {Role: "user", Content: "Fix the bug in function X."},  
   },  
  })
* This returns the agent’s reasoning or code suggestion. The Golang backend or the worker handles this call and parses the JSON response.
* **Anthropic Claude Integration**: Similar to OpenAI, but use Anthropic’s API endpoints. For example, POST to https://api.anthropic.com/v1/chat/completions with headers x-api-key and anthropic-version. Anthropic provides an official Go SDK or we can use standard HTTP clients. No special concerns beyond handling keys.
* **Ollama (Local LLM)**: Ollama runs as a local service after you execute ollama run <model>. It listens on http://localhost:11434 by default. The platform should run Ollama on GPU-enabled nodes (if using llama/groq etc.) and ensure the API is reachable by the worker service. The worker can then do:
* // Example from [12]  
  ollamaReq := OllamaRequest{Model: "llama3.1", Messages: []Message{...}, Stream: false}  
  resp, err := talkToOllama("http://localhost:11434/api/chat", ollamaReq)
* This local call avoids external network and keeps data in-house, as [12] describes in detail. If multiple models are needed, run separate Ollama containers or multiple instances.
* **External APIs (User Tools)**: If agents need to call arbitrary external APIs (like GitHub or other SaaS), we provide a generic API caller tool where the workflow specifies endpoint and parameters. The worker then makes HTTP calls. We must manage secrets for those API tokens via a secure vault and only supply the relevant token to the container.

All external calls (LLM or APIs) should be logged for auditing and limited (rate limiting, per-tenant usage caps) to control cost.

## DevOps and Deployment

* **Containerization**: Each component (API server, workers, database, queue, frontend) is containerized. Use Dockerfiles to build images. For example, the Golang API server image (based on golang:1.XX-alpine) and the React app (built and served by nginx or a static file server).
* **Kubernetes**: Deploy on Kubernetes for scalability:
* **API Server**: Deployment with 2+ replicas behind a Service/Ingress. Horizontal Pod Autoscaler (HPA) based on CPU or request latency.
* **Workers**: Deployment of worker pods (e.g. 4 replicas), HPA based on queue length (scale out when many tasks pending).
* **Database**: Use managed Postgres or a StatefulSet with persistent volumes. Enable auto-scaling read replicas if needed for heavy queries.
* **Redis**: Clustered Redis for broker, possibly via a StatefulSet or a cloud Redis service with HA.
* **Ollama/GPU Nodes**: If running large local models, have GPU node pools. Use node selectors or taints/tolerations to schedule heavy LLM workloads onto GPU nodes only.
* **Observability**: Run a monitoring stack:
* Prometheus + Grafana for metrics (K8s metrics, custom app metrics).
* ELK/EFK (Elasticsearch/Fluentd/Kibana) or Loki/Tempo for logs and traces.
* Alerting (PagerDuty, Slack) for errors or resource anomalies.
* **CI/CD**: On commit to main:
* Run tests, then build Docker images, push to registry.
* Update Helm charts or Kubernetes manifests.
* Use Blue/Green or Rolling deployments for zero-downtime.
* **Scaling**:
* Stateless services (API, frontend) scale horizontally behind a load balancer.
* Database scaling via read replicas and sharding if needed.
* Ensure sticky sessions or JWT so any pod can service a user session.
* **Backup & Recovery**:
* Regular backups of Postgres.
* Use a message queue with persistence to avoid losing tasks.
* Design the system so failed executions can be resumed/retried.

Overall, the architecture should follow Cloud Native best practices (12-factor apps, metrics, health checks). For example, use Kubernetes readiness probes on the API, and liveness probes on workers.

## Libraries and Technologies

* **Golang**:
* Web framework: Gin or chi for REST API. gRPC (built-in grpc pkg) for internal RPCs.
* Database: sqlx or GORM for Postgres ORM.
* Queues: [Asynq](https://github.com/hibiken/asynq) for Redis-based tasks[[1]](https://github.com/hibiken/asynq" \l ":~:text=,processed%20concurrently%20by%20multiple%20workers).
* WebSockets: [gorilla/websocket](https://github.com/gorilla/websocket) or use a library like nhooyr/websocket for simplicity.
* Cloud SDKs: AWS SDK for secrets, Kubernetes client-go for any K8s interactions (if needed).
* LLM Clients: openai-go, or custom HTTP calls for OpenAI/Anthropic; custom code or ollama clients for local LLMs.
* **React**:
* UI libraries: Material-UI or Chakra for components.
* State management: Context API or Redux.
* Data fetching: axios or fetch for REST, Socket.IO or native WebSocket API for real-time.
* Workflow editor: A library like react-flow for DAG editing (optional).
* Authentication: Use JWT stored in memory or secure cookie. Integrate with OAuth if single sign-on needed.
* **Container Runtime**:
* Use Docker (for dev) and containerd/CRI-O on Kubernetes.
* For secure sandbox, consider gVisor or Firecracker if extra isolation is needed.
* **DevOps Tools**:
* Docker Compose (for local dev), Helm (for k8s).
* GitHub Actions / GitLab CI for pipelines.
* Terraform or CloudFormation for infra.
* Monitoring: Prometheus operator, Grafana, Loki, etc.

## Summary

In summary, the proposed architecture uses a **microservices approach** with a Golang backend and React frontend. A **workflow engine** interprets user-defined steps and enqueues tasks. **Workers** run each task in isolated containers, using tools for web browsing, shell execution, code editing, and LLM reasoning. Postgres stores all definitions and logs, and Redis-based queues (e.g. Asynq) manage task scheduling with horizontal scaling[[1]](https://github.com/hibiken/asynq" \l ":~:text=,processed%20concurrently%20by%20multiple%20workers). Communication is via REST/JSON and gRPC internally, with WebSockets for live updates. Security is enforced through container sandboxing[[2]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=The%20OpenHands%20Docker%20Runtime%20is,without%20risking%20the%20host%20system) and robust secrets handling (avoiding file-based secrets as recommended[[8]](https://blog.gitguardian.com/how-to-handle-secrets-in-go/" \l ":~:text=However%2C%20we%20can%27t%20hard,secrets%20in%20the%20Docker%20image)). The platform leverages cloud-native practices (containerization, Kubernetes orchestration, CI/CD, monitoring) to achieve a **scalable, multi-tenant SaaS** solution. This design supports any agent that can “brows[e] the web, run commands, call APIs, and reason using LLMs,” mirroring OpenHands’ vision[[11]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=How%20does%20the%20Runtime%20work%3F)[[4]](https://articles.wesionary.team/building-a-multi-tenant-architecture-in-golang-a-practical-guide-8ee066436678?gi=9a3f54dc9af1" \l ":~:text=The%20key%20to%20multi,That%E2%80%99s%20the%20core%20concept), but implemented in Go/React with enterprise-grade production patterns.

[[1]](https://github.com/hibiken/asynq" \l ":~:text=,processed%20concurrently%20by%20multiple%20workers) GitHub - hibiken/asynq: Simple, reliable, and efficient distributed task queue in Go

<https://github.com/hibiken/asynq>

[[2]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=The%20OpenHands%20Docker%20Runtime%20is,without%20risking%20the%20host%20system) [[3]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=components%20like%20a%20bash%20shell,the%20OpenHands%20backend%20as%20observations) [[6]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=Image%20Tagging%20System) [[10]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=1,from%20affecting%20the%20host%20system) [[11]](https://docs.all-hands.dev/usage/architecture/runtime" \l ":~:text=How%20does%20the%20Runtime%20work%3F) Runtime Architecture - All Hands Docs

<https://docs.all-hands.dev/usage/architecture/runtime>

[[4]](https://articles.wesionary.team/building-a-multi-tenant-architecture-in-golang-a-practical-guide-8ee066436678?gi=9a3f54dc9af1" \l ":~:text=The%20key%20to%20multi,That%E2%80%99s%20the%20core%20concept) Building a Multi-Tenant Architecture in Golang: A Practical Guide | by Mukesh Kumar Chaudhary | wesionaryTEAM

<https://articles.wesionary.team/building-a-multi-tenant-architecture-in-golang-a-practical-guide-8ee066436678?gi=9a3f54dc9af1>

[[5]](https://dshills.medium.com/go-ollama-simple-local-ai-3a89be4bfbaf" \l ":~:text=The%20API%E2%80%99s%20URL%20is%20typically,http%3A%2F%2Flocalhost%3A11434%2Fapi%2Fchat) Go + Ollama = Simple Local AI. Are you a Go developer interested in… | by Davin Hills | Medium

<https://dshills.medium.com/go-ollama-simple-local-ai-3a89be4bfbaf>

[[7]](https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis" \l ":~:text=Characteristic%20gRPC%20REST%20API%20HTTP,Time%2045%20Minutes%2010%20Minutes) gRPC vs. REST: Key Similarities and Differences

<https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis>

[[8]](https://blog.gitguardian.com/how-to-handle-secrets-in-go/" \l ":~:text=However%2C%20we%20can%27t%20hard,secrets%20in%20the%20Docker%20image) [[9]](https://blog.gitguardian.com/how-to-handle-secrets-in-go/" \l ":~:text=) How to Handle Secrets in Go

<https://blog.gitguardian.com/how-to-handle-secrets-in-go/>

--------------------------------------------------------------------------------------------------------------------

# OpenHands-Like AI Agent Platform: Development Guide

## 1. High-Level Development Roadmap

**Phase 1 – MVP (Minimum Viable Product)**

* Core features:
  + Workflow creation (simple DAG of steps)
  + Agent execution (run shell commands, basic API calls)
  + LLM integration (OpenAI / Ollama)
  + Logging and execution tracking
* Stack:
  + Backend: Go (Gin or Chi)
  + Frontend: React + Material-UI
  + Database: PostgreSQL
  + Queue: Redis + Asynq
* Deliverables:
  + Users can create workflows
  + Workers execute tasks safely
  + Live logs displayed in UI

**Phase 2 – Advanced Agent Tools**

* Add capabilities:
  + Web browsing / scraping (headless Chrome in sandbox)
  + Code modification / file management
  + Multi-agent coordination (parallel steps, subgoals)
  + Conditional workflow branching
* Security:
  + Docker-based sandboxing for all agents
  + Resource limits and network restrictions
* Deliverables:
  + Agents can autonomously fetch data, modify code, and call APIs
  + Full audit logs

**Phase 3 – Multi-Tenant SaaS & Scalability**

* Multi-tenancy: isolate users/orgs
* Horizontal scaling: multiple worker pods, orchestrator scaling
* Frontend enhancements: real-time dashboards, workflow visualization
* Integrate marketplace for agent templates
* Observability: Prometheus, Grafana, centralized logging
* Deliverables:
  + Platform can handle multiple enterprises
  + Scalable architecture with CI/CD

**Phase 4 – Ecosystem & Marketplace**

* Agent templates and extensions
* Customizable LLM prompts
* Plugin marketplace for 3rd-party integrations
* Training and onboarding modules
* Deliverables:
* Self-service platform for developers to extend the system

# or

### Phase 1 – MVP (Minimum Viable Product)

* **Core features:** Workflow creation, agent execution (shell commands, basic API calls), LLM integration, logging.
* **Stack:** Go (Gin/Chi), React + Material-UI, PostgreSQL, Redis + Asynq.
* **Deliverables:** Users can create workflows, workers execute tasks safely, live logs displayed in UI.

### Phase 2 – Advanced Agent Tools

* **Capabilities:** Web browsing, code modification, multi-agent coordination, conditional branching.
* **Security:** Docker-based sandboxing, resource limits.
* **Deliverables:** Autonomous agents with full audit logs.

### Phase 3 – Multi-Tenant SaaS & Scalability

* Multi-tenancy, horizontal scaling, real-time dashboards, workflow visualization.
* Observability: Prometheus, Grafana, centralized logging.

### Phase 4 – Ecosystem & Marketplace

* Agent templates, customizable LLM prompts, plugin marketplace, training modules.

## 2. Tech Stack

| Layer | Technology / Library |
| --- | --- |
| Frontend | React, Material-UI / Chakra, React Flow, WebSockets |
| Backend API | Go, Gin / Chi, GORM / SQLx |
| Worker / Agent Runtime | Go + Docker SDK, Headless Chrome, Python scripts |
| LLM Integrations | OpenAI Go SDK, Ollama API, Anthropic Claude API |
| Database | PostgreSQL, JSONB for workflow steps and outputs |
| Queue / Scheduler | Redis + Asynq |
| Authentication | JWT, OAuth2 (Keycloak or Auth0) |
| Monitoring / Logging | Prometheus, Grafana, Loki, OpenTelemetry |
| DevOps / Infra | Docker, Kubernetes, Helm, Terraform |

## 3. Backend Architecture (Go)

* **API Layer:** CRUD for workflows, start/stop executions, manage agents, WebSocket logs.
* **Worker / Agent Runtime:** Pull tasks from Redis, run sandboxed Docker containers, execute steps, capture outputs.
* **Orchestrator:** Reads workflow DAG from DB, enqueues steps, manages dependencies.
* **LLM Integration Module:** Abstraction for multiple providers (OpenAI, Ollama, Claude).
* **Database Models:** Users, tenants, workflows, steps, executions, agents, service tokens.

## 4. Frontend Structure (React)

* Pages/Components: Login/Register, Dashboard, Workflow Builder (drag-and-drop DAG), Agent Configuration.
* Real-Time Updates: WebSockets or SSE.
* State Management: Redux or Context API.
* Visualization: React Flow, Chart.js or D3.js.

## 5. Security & Sandboxing

* Each agent step runs in a Docker container.
* Non-root users, filesystem/network restrictions.
* Secrets via environment variables or secret manager.
* CPU/memory limits per container.
* Multi-tenant isolation with tenant IDs.

## 6. CI/CD & Deployment

* **Local dev:** Docker Compose for backend, frontend, Redis, Postgres.
* **Production:** Kubernetes + Helm charts.
* **CI/CD:** Build Docker images, run tests, deploy.
* **Monitoring:** Prometheus + Grafana + Loki.

## 7. Development Plan (Week-wise)

| Week | Tasks |
| --- | --- |
| 1-2 | Setup Go backend, PostgreSQL, Redis, basic API CRUD |
| 3-4 | React frontend, login/auth, workflow CRUD UI |
| 5-6 | Worker setup, queue integration, execute simple shell/LLM steps |
| 7-8 | Docker sandbox, logging, real-time frontend updates |
| 9-10 | Web scraping, code editing tools, multi-step workflows |
| 11-12 | LLM provider abstraction, integrate OpenAI/Ollama/Claude |
| 13+ | Multi-tenancy, scaling, CI/CD, monitoring, marketplace |

This document serves as a **starter blueprint** for building a production-ready OpenHands-like AI agent platform using Go and React.