Integrating Holochain DNA with ROS2 Humble for Distributed Robotics and XR

1. Architectural Overview: Merging ROS2 with Holochain

The integration of Holochain's agent-centric, distributed ledger technology (DLT) with the Robot Operating System 2 (ROS2) presents a paradigm shift for building resilient, scalable, and secure robotic and extended reality (XR) applications. This architecture moves beyond traditional centralized or client-server models, where a single point of failure can compromise the entire system. Instead, it proposes a model where each device—be it a robot, an XR/MR headset, or a compute node—operates as an autonomous agent within a peer-to-peer network. The core of this architecture is the Holochain DNA (Distributed Network Application), which serves as the foundational blueprint for the network's rules, data structures, and validation logic. By embedding this DNA into each agent, the system achieves a shared, verifiable state without requiring global consensus, a process that is often computationally expensive and slow in traditional blockchains. This approach is particularly well-suited for resource-constrained IoT devices, as highlighted in a review of DLTs for IoT security, which emphasizes Holochain's ability to function effectively on devices with limited computational power, storage, and energy . The architecture aims to provide a unified framework for distributed compute, inference, storage, and memory, enabling complex, multi-agent scenarios such as decentralized robotics swarms and agentic DevOps managed through voice and vision inputs from XR/MR glasses.

1.1. Core Concept: Using Holochain DNA as a Decentralized Layer for ROS2

The fundamental concept behind this integration is to use Holochain as a decentralized, trust-enabling layer that underpins and enhances ROS2's communication infrastructure. While ROS2 excels at providing a robust middleware framework for robotics, its default discovery and communication mechanisms, often based on DDS (Data Distribution Service), are primarily designed for local network environments and can face challenges in large-scale, dynamic, or security-sensitive deployments. Holochain addresses these limitations by introducing a cryptographically secure, agent-centric data layer. Each agent maintains its own source chain, an immutable log of its actions, which provides data provenance and auditability. When an agent commits data, it is not just stored locally; it is published to a distributed hash table (DHT), where it is validated by a subset of peers before being integrated into the shared reality of the

network. This process ensures data integrity and prevents malicious actors from corrupting the system. The Holochain RSM (Refactored State Model) migration guide emphasizes this shift, highlighting that commits now represent the "act of saying something," with each action being a discrete event that transforms both local and shared state . This event-sourcing pattern is a powerful fit for robotics, where a sequence of commands, sensor readings, and state changes can be reliably tracked and audited across a distributed network of agents.

1.1.1. Mapping ROS2 Communication Patterns to Holochain

A critical aspect of the integration is mapping ROS2's core communication patterns—publish/subscribe (topics), request/response (services), and asynchronous goal-oriented tasks (actions)—onto Holochain's architecture. The publish/subscribe model, which is the most common in ROS2, can be elegantly replicated using Holochain's DHT. A ROS2 node acting as a publisher would commit a new entry to the DHT for each message it wishes to broadcast. This entry would contain the topic name, the serialized message data, and a timestamp. Subscribers, in turn, would query the DHT for entries corresponding to the topics they are interested in. This can be achieved by creating links from an "anchor" entry (e.g., representing the topic) to each new message entry, allowing for efficient retrieval of a topic's message history . For services and actions, the interaction is more direct. A service request can be modeled as a committed entry, and the response can be another entry linked back to the request. The requesting agent would monitor its source chain or the DHT for the linked response entry. This approach, while potentially introducing higher latency than native DDS, provides the significant benefits of data persistence, auditability, and cross-network accessibility. Research into integrating ROS2 with other distributed ledger technologies (DLTs) like Hyperledger Fabric and IOTA has demonstrated the viability of this bridging concept, often using a dedicated application layer to translate between ROS2 and the DLT's native protocols . These existing frameworks provide valuable architectural blueprints for a Holochain-based solution.

1.1.2. Leveraging the DHT for Distributed Storage and Data Integrity

The Distributed Hash Table (DHT) is the cornerstone of Holochain's shared data layer and a critical component for enabling distributed storage and ensuring data integrity across the robotic network. Unlike a traditional blockchain where every node stores a complete copy of the ledger, the Holochain DHT is a monotonic, validating DHT where each node is responsible for storing and validating only a small, specific portion of the data . This design is a significant advantage for resource-constrained devices like

robots and XR glasses, as it avoids the massive storage and bandwidth requirements of a full ledger. When a ROS2 node (acting as a Holochain agent) publishes data—such as a robot's pose, a camera frame's metadata, or a system health log—it commits an entry to its local source chain. This entry is then gossiped to other peers in the network who are responsible for that entry's hash space. These peers, known as validation authorities, check the entry against the validation rules defined in the DNA. If the entry is valid, it is stored in the DHT, making it discoverable and retrievable by any other agent. This process creates a resilient and tamper-evident record of all system events. The "HoloSec" framework, proposed for IoT security, leverages this very architecture, combining Holochain's agent-centric model and advanced cryptography to secure distributed networks . This ensures that even if individual robots fail or are compromised, the collective memory and integrity of the system are preserved within the DHT.

1.1.3. Agent-Centric Model for Decentralized Robotics and DevOps

Holochain's agent-centric architecture is a natural fit for decentralized robotics and agentic DevOps, as it aligns with the autonomous nature of robots and the goal of creating self-managing systems. In this model, each robot, XR headset, or server is an agent with its own unique identity (a cryptographic key pair) and its own source chain. This means that each agent has full authority over its own state and actions. For robotics, this allows for a high degree of local autonomy; a robot can make decisions and execute tasks based on its local state and sensory input without needing constant permission from a central server. When coordination is required, agents can interact directly through the DHT. For example, a fleet of delivery robots could use the DHT to negotiate routes, share map updates, and coordinate handoffs, all without a central dispatch. This agent-centric approach is also powerful for agentic DevOps. System configurations, software versions, and operational logs can be managed as entries on each agent's source chain. An agent could autonomously decide to update its software based on a validated update entry it discovers on the DHT, or it could log a hardware failure, which would then be visible to the entire network for maintenance scheduling. The Holochain RSM migration guide notes that this model better reflects the multi-perspective nature of the real world, allowing Alice and Bob to both create a "Buy milk" to-do item and mark their own as done without affecting the other's . This translates directly to a multi-robot scenario where each robot can maintain its own task list and state, while still contributing to a shared understanding of the overall mission.

1.2. System Components and Their Roles

The integrated system is composed of several key components, each with a distinct role in enabling the distributed functionality. These components work in concert to bridge the gap between the ROS2 world of topics, services, and actions, and the Holochain world of agents, DHTs, and source chains. The primary components are the Holochain DNA, which defines the application's logic; the Holochain Conductor, which manages the execution environment for the DNA; and the ROS2 nodes themselves, which are adapted to act as agents within the Holochain network. Understanding the interplay between these components is crucial for designing and implementing a successful integration. The DNA provides the "law of the land," the Conductor acts as the local interpreter and enforcer of that law, and the ROS2 nodes are the citizens of the land, interacting with each other and the environment according to the established rules. This modular design allows for flexibility and evolution; the DNA can be updated to introduce new features or fix bugs, and new ROS2 nodes can be added to the network as long as they adhere to the DNA's protocols.

1.2.1. The Holochain DNA: Defining Network Rules and Data Structures

The Holochain DNA (Distributed Network Application) is the foundational blueprint for the entire distributed application. It is a bundle of configuration files and compiled WebAssembly (WASM) code that defines the rules, data structures, and validation logic for the network. Think of it as the constitution and legal code for the robotic collective. The DNA specifies what kinds of data (entries) can be created, what functions (zomes) can be called, and who is allowed to do what. For a ROS2 integration, the DNA would contain zomes for managing ROS2 topics, handling distributed compute tasks, and coordinating agentic DevOps. For example, a "topic_manager" zome might define entry types for `TopicAdvertisement`, `TopicSubscription`, and `MessagePublication`. The validation rules within this zome would ensure that only authorized agents can publish to a specific topic or that message data conforms to a predefined schema. The DNA is packaged into a `.dna` file and is installed into a Holochain Conductor. Every agent in the network runs the same DNA, which ensures that they all agree on the rules of the game. This shared, immutable set of rules is what allows a diverse set of devices to collaborate in a trustless environment, as they can independently verify that all actions are valid according to the agreed-upon DNA.

1.2.2. The Holochain Conductor: Managing Agent Instances and Access

The Holochain Conductor is the central runtime component that orchestrates the execution of Holochain applications (hApps) on a participant's device . It functions as a local application server, sandboxing the hApp code within a WebAssembly (WASM)

environment and mediating all access to the device's resources, including networking and storage . This architecture ensures that each participant runs their own instance of the application logic, operating from their unique perspective and with their own data. The Conductor is responsible for several critical functions: it manages the participant's cryptographic identities (agent IDs) stored in a keyring, handles data flow and persistence, and facilitates both local and remote communication . When a client application, such as a user interface or another process like a ROS2 node, makes a function call, the Conductor routes this request to the appropriate function within the correct hApp . This client–server model, where the Conductor acts as the server, is fundamental to how external systems interact with the Holochain network .

The Conductor exposes two primary WebSocket–based Remote Procedure Call (RPC) APIs for external communication: the **Admin API** and the **Application (App) API** . The Admin API is designed for application managers or setup scripts to control the Conductor itself. It provides functions to install new hApps, create and manage agent IDs, activate or deactivate applications, and query the Conductor's state, such as listing installed DNAs or active cells . The Application API, on the other hand, is used by client applications to interact with the running hApps. This API allows clients to call the public functions (zome functions) exposed by a hApp's DNA and to receive real–time event notifications, known as signals, broadcast by the cells . For a ROS2 integration, the **Application API is the primary interface** through which ROS2 nodes would commit data to the Distributed Hash Table (DHT) or query data from it. The Conductor's configuration is typically managed through a YAML file, which specifies settings like network parameters, storage paths, and the interfaces to be exposed . This configuration allows for fine–grained control over how the Conductor operates within its environment, making it adaptable to various deployment scenarios, from development to production .

1.2.3. ROS2 Nodes as Holochain Agents: Interfacing with the DNA

In this integrated architecture, each ROS2 node (or a group of nodes representing a single robot) is treated as a Holochain agent. This means that each node has its own unique agent key and its own instance of the DNA running within a Conductor. This agent–centric model allows each robot to maintain its own state and history while participating in the larger distributed network. The ROS2 nodes interface with the Holochain DNA through the Conductor's API. For example, a robot's localization node might publish its current pose as a ROS2 topic and simultaneously commit a corresponding entry to its source chain via a call to a "pose_publisher" zome. This

action would make the robot's location a part of the shared, verifiable history of the network. Conversely, a navigation node could query the DHT for the poses of other robots by calling a function in a "pose_subscriber" zome. This allows the robot to build a real-time, trusted map of its environment based on data from its peers. The key is to create a "bridge" or "shim" layer within the ROS2 node that translates between ROS2 message types and Holochain entry types, and that manages the asynchronous calls to the Conductor's API. This approach allows existing ROS2 applications to be gradually adapted to work within the Holochain framework, leveraging the benefits of decentralization without requiring a complete rewrite.

1.3. High-Level Integration Strategy

The high-level integration strategy focuses on creating a seamless bridge between the ROS2 and Holochain ecosystems, allowing them to work together as a single, cohesive distributed system. This involves not only technical integration at the code level but also a unified approach to packaging, deployment, and networking. The goal is to create a development and deployment environment where developers can build and run ROS2 applications that are inherently decentralized and secure, without needing to be experts in both ROS2 and Holochain. The strategy is built on three main pillars: creating a robust bridge between ROS2 and the Holochain Conductor, packaging both systems into a unified containerized environment, and establishing a networking model that supports both ROS2's DDS communication and Holochain's peer-to-peer protocols. This holistic approach ensures that the integrated system is not just a collection of disparate parts but a well-architected platform for building the next generation of distributed robotic applications.

1.3.1. Creating a ROS2-Holochain Bridge via the Conductor API

The primary strategy for integrating ROS2 with Holochain involves creating a bridge that allows ROS2 nodes to communicate with the Holochain Conductor via its Application API . This bridge acts as an intermediary, translating ROS2 messages and service calls into Holochain zome function invocations and vice versa. The Conductor's Application API is exposed over a local WebSocket, providing a standard interface for any client process to interact with the running hApp cells . A dedicated ROS2 node, or a set of nodes, would be developed to serve as this bridge. This "Holochain Bridge Node" would be responsible for establishing and maintaining a connection to the local Conductor's App API. It would subscribe to specific ROS2 topics, and upon receiving a message, it would serialize the message data and call a corresponding zome function to commit that data as an entry to the Holochain DHT. Conversely, it could listen for

signals from the Conductor, which represent real-time events from the Holochain network (e.g., new data committed by another agent), and publish this information as a ROS2 message for other nodes in the local ROS2 graph to consume .

This approach leverages the client-server architecture inherent in Holochain, where the Conductor is the server and the ROS2 bridge node is the client . The Holochain project provides official client libraries in JavaScript and Rust to facilitate this interaction, abstracting away the low-level details of the WebSocket communication and MessagePack serialization . For a ROS2 Humble environment, which is primarily C++ and Python, a C++ client library would be ideal. While an official C++ client is not mentioned in the provided documentation, the existence of a community-maintained C# client suggests that creating a C++ wrapper around the WebSocket API is a feasible path . Alternatively, a Python-based bridge node could be developed using a Python WebSocket client and a MessagePack library, which might be a faster route for prototyping. The bridge node would need to handle the asynchronous nature of both ROS2 (using callbacks) and Holochain (using signals and promises), ensuring that data flows smoothly between the two systems without blocking.

1.3.2. Packaging Holochain and ROS2 within a Unified Environment

To simplify development and deployment, it is essential to package both ROS2 and Holochain within a unified environment. The user's specified `ros2-humble-env` repository, which uses NixOS and Docker, provides an excellent foundation for this. The strategy is to extend this environment to include all the necessary components for running a Holochain Conductor. This includes installing the `holochain` binary and its dependencies, as well as the tools needed to build and package Holochain DNAs (e.g., `hc`, the Holochain CLI). The Docker configuration will need to be updated to ensure that the Holochain Conductor can run correctly within the container. This may involve setting up the necessary network permissions, file system mounts, and environment variables. By creating a single, self-contained Docker image that includes both ROS2 Humble and a fully configured Holochain environment, developers can easily spin up new instances of the system on any machine that supports Docker. This approach also facilitates the use of more advanced containerization technologies like Firecracker and Kata, which can provide enhanced security and isolation for the distributed agents. The goal is to create a "batteries-included" development environment where everything needed to build and run a decentralized ROS2 application is pre-installed and pre-configured.

1.3.3. Networking Model for Cross-Device Communication

A critical aspect of the integration is the networking model, which must support both ROS2's DDS-based communication and Holochain's peer-to-peer protocols. ROS2's default discovery mechanism relies on multicast, which can be problematic in complex network environments or when communicating across different subnets. To address this, the integration will leverage DDS's support for discovery servers, as detailed in documentation from Clearpath Robotics and Husarion . In this model, one or more devices in the network act as discovery servers, and all other ROS2 nodes are configured to connect to these servers. This allows for more controlled and reliable discovery, especially in large-scale deployments. The Holochain networking layer, on the other hand, is designed to be more flexible and can work over a variety of transport protocols. The key is to ensure that the Docker containers are configured with the correct network settings to allow for both types of communication. This will likely involve using Docker's host networking mode (`--net=host`) or creating a custom Docker network that allows the containers to communicate with each other and with the host machine. The networking configuration will also need to account for potential issues like NAT traversal and firewall rules, which are common in real-world deployments. The goal is to create a robust and flexible networking model that allows the distributed agents to communicate seamlessly, regardless of their physical location or network configuration.

2. Environment Setup: The `ros2-humble-env` Foundation

The successful integration of Holochain and ROS2 hinges on a robust and reproducible development environment. The user has specified the use of the `ros2-humble-env` repository, which provides a pre-configured environment based on NixOS, Pixi, Nushell, and Docker. This section details the steps required to set up this foundation and extend it to support Holochain. The process involves initializing the NixOS development shell, configuring Docker for containerized deployment, and considering the use of advanced containerization technologies like Firecracker and Kata for enhanced security and performance. By following these steps, developers can create a consistent and reliable environment for building and testing their decentralized robotic applications. The emphasis is on creating a "batteries-included" setup where all necessary tools and dependencies are pre-installed and configured, allowing developers to focus on their application logic rather than on environment setup.

2.1. Initializing the NixOS Development Environment

The `ros2-humble-env` repository is built around NixOS, a Linux distribution that uses the Nix package manager to provide a declarative and reproducible way to manage software. This is a significant advantage for a complex project like this, as it ensures that all developers are working with the exact same versions of all dependencies, from the ROS2 libraries to the Holochain binaries. The initialization process involves cloning the repository and using the provided Nix configuration to enter a development shell that contains all the necessary tools. This shell will provide a consistent environment across different machines, eliminating the "it works on my machine" problem. The use of Pixi and Nushell further enhances the development experience by providing a fast and user-friendly way to manage the environment and run commands.

2.1.1. Utilizing the `ros2-humble-env` Repository Structure

The first step in setting up the development environment is to clone the `ros2-humble-env` repository from GitHub. This repository contains all the necessary configuration files for setting up the NixOS-based development environment. The structure of the repository is designed to be modular and easy to understand. It will likely contain a `flake.nix` file, which is the main entry point for the Nix configuration, as well as directories for Docker configurations, ROS2 packages, and other project-specific files. By examining the repository structure, developers can get a clear understanding of how the environment is organized and where they need to make changes to add Holochain support. The repository's README file should provide detailed instructions on how to get started, including how to clone the repository and how to enter the development shell. It is important to follow these instructions carefully to ensure that the environment is set up correctly.

2.1.2. Integrating Nix Package Management for ROS2 and Dependencies

Nix's declarative package management is a key feature of the `ros2-humble-env`. The `flake.nix` file will specify all the dependencies required for the project, including the ROS2 Humble distribution, various ROS2 packages, and any other system-level dependencies. To add Holochain support, we will need to modify this file to include the necessary Holochain packages. This may involve adding the `holochain` package from the official Nixpkgs repository, or it may require creating a custom Nix derivation to build Holochain from source. The Nix configuration will also need to specify the correct versions of all dependencies to ensure compatibility. Once the `flake.nix` file has been updated, developers can use the `nix develop` command to enter a shell with all the specified packages installed. This provides a clean and isolated environment where

developers can work without worrying about conflicts with other software installed on their system.

2.1.3. Configuring the Development Shell with Pixi and Nushell

The `ros2-humble-env` also utilizes Pixi and Nushell to enhance the development experience. Pixi is a fast and user-friendly package manager that can be used to manage project-specific dependencies, while Nushell is a modern shell that provides a more powerful and intuitive command-line interface. The Nix development shell can be configured to use Pixi and Nushell by default, providing a seamless and efficient workflow. For example, Pixi can be used to install Python packages that are needed for a specific ROS2 node, while Nushell can be used to run complex build and test commands with ease. The configuration for Pixi and Nushell will likely be included in the `flake.nix` file or in a separate configuration file that is sourced by the development shell. By taking advantage of these tools, developers can create a highly customized and productive development environment that is tailored to the specific needs of their project.

2.2. Docker Configuration for Containerized Deployment

Docker is a key component of the `ros2-humble-env`, providing a way to package the entire development environment into a self-contained container. This is essential for ensuring that the application can be deployed consistently across different machines, from a developer's laptop to a production server. The Docker configuration will need to be updated to include the Holochain components, as well as any other dependencies that are required for the application. This section details the steps for building the ROS2 Humble Docker image, configuring Docker networking for inter-container and cross-host communication, and enabling GUI applications with X11 forwarding.

2.2.1. Building the ROS2 Humble Docker Image

The `ros2-humble-env` repository will contain a `Dockerfile` that defines the ROS2 Humble Docker image. This file will specify the base image to use (e.g., `ubuntu:22.04`), as well as all the commands needed to install ROS2 and its dependencies. To add Holochain support, we will need to modify this `Dockerfile` to include the installation of the Holochain binary and any other required tools. This may involve adding a new `RUN` command to download and install the Holochain package, or it may require copying a pre-built Holochain binary into the image. Once the `Dockerfile` has been updated, we can use the `docker build` command to create a

new Docker image that includes both ROS2 and Holochain. This image can then be used to run containers that are ready to execute our decentralized robotic applications. It is a good practice to tag the new image with a descriptive name and version number to make it easy to manage and deploy.

2.2.2. Configuring Docker Networking for Inter-Container and Cross-Host Communication

Configuring Docker networking is a critical step to ensure that ROS2 nodes running in separate containers can communicate with each other and with the host system. ROS2's default middleware, DDS, relies on network discovery mechanisms that can be disrupted by Docker's default network isolation. The most common and effective solution for enabling seamless ROS2 communication within a single host is to use the host's network stack by running containers with the **`--network=host`** flag . This mode gives the container direct access to the host's network interfaces, eliminating the need for port mapping and ensuring that DDS can use its standard multicast discovery protocols. When a container is run with `--network=host` , it shares the same IP address as the host, which simplifies network configuration and avoids conflicts . This is particularly important for GUI applications like RViz or Gazebo, which also require access to the host's X11 server, a configuration that is greatly simplified when the container is on the same network .

However, using the host network mode has its trade-offs. It reduces the level of network isolation between the container and the host, which could be a security concern in some environments. For multi-host deployments, where ROS2 nodes need to communicate across different machines, a more complex networking setup is required. While `--network=host` works well on a single machine, it does not inherently solve cross-host communication. In such scenarios, one might need to configure an overlay network, such as Docker's built-in `overlay` driver or a third-party solution like Weave Net or Flannel. These networks create a virtual network that spans multiple Docker hosts, allowing containers on different machines to communicate as if they were on the same local network. This is essential for a distributed robotics system where robots and user devices (like XR glasses) are on different physical machines but need to share data via the Holochain DHT and ROS2 topics.

2.2.3. Enabling GUI Applications (e.g., RViz, Gazebo) with X11 Forwarding

Running GUI applications like RViz or Gazebo from within a Docker container requires special configuration to forward the graphical display from the container to the host's

X11 server. This process, known as X11 forwarding, is essential for visualizing robot models, sensor data, and simulations. The most common method for achieving this on a Linux host involves sharing the host's X11 Unix socket with the container and setting the `DISPLAY` environment variable correctly . The X11 Unix socket is typically located at `/tmp/.X11-unix` on the host. By mounting this directory as a volume into the container using the `-v /tmp/.X11-unix:/tmp/.X11-unix:rw` flag, the container can communicate with the host's X server. Additionally, the `DISPLAY` environment variable inside the container must be set to match the one on the host, which is usually `:0` or `:1` . This can be passed to the container using the `-e DISPLAY=$DISPLAY` flag .

However, simply sharing the socket and setting the `DISPLAY` variable is often not enough due to X11's security mechanisms. The X server needs to be configured to allow the container to connect. This is typically done using the `xhost` utility. The command **`xhost +local:docker`** (or `xhost +local:<your_username>` if running the container as a non-root user) allows any local process, including those in a Docker container, to connect to the X server . While this is convenient for development, it is a security risk as it allows any local user to access your display. A more secure approach is to mount the host user's `.Xauthority` file into the container. The `.Xauthority` file contains a "magic cookie" that is used to authenticate connections to the X server. By mounting this file with `-v $HOME/.Xauthority:/root/.Xauthority:ro` (adjusting the path and user as necessary), the container can present the correct credentials to the X server, eliminating the need for `xhost` .

2.3. Advanced Containerization with Firecracker and Kata

While Docker is the primary containerization technology for this project, the user's specification of Firecracker and Kata containers suggests an interest in exploring more advanced and secure forms of containerization. These technologies represent a middle ground between traditional containers and full virtual machines (VMs), offering enhanced isolation and security without the significant performance overhead of a full VM. Understanding their role and potential benefits is important for designing a robust and future-proof distributed system. Firecracker is a micro-VM technology developed by AWS, designed for running serverless workloads like AWS Lambda. It provides a lightweight, secure, and fast-booting virtual machine that can be used to run containers. Kata Containers, on the other hand, is an open-source project that aims to make running containers in VMs as seamless as running them in traditional containers.

It uses a hypervisor to create a lightweight VM for each container, providing the security and isolation of a VM with the user experience of a container.

2.3.1. Rationale for Using Micro-VMs in a Distributed Robotics Context

The use of micro-VMs like Firecracker and Kata in a distributed robotics context offers several compelling advantages, particularly in terms of security and multi-tenancy. In a decentralized system where multiple users' devices contribute to a shared pool of compute, storage, and memory, ensuring the isolation of workloads is of paramount importance. Traditional Docker containers share the host kernel, which means that a vulnerability in the kernel or a misconfigured container could potentially compromise the entire system. Micro-VMs, by contrast, provide a stronger security boundary by running each container in its own isolated virtual machine with its own kernel. This makes them an attractive option for running untrusted or third-party code on a user's device, as it significantly reduces the risk of a security breach. For example, if a distributed inference task is offloaded to a user's device, running it in a micro-VM would prevent it from accessing or interfering with other processes on the device.

Another key benefit of micro-VMs is their ability to support multi-tenancy more effectively. In a distributed robotics system, it may be desirable to run multiple, independent applications on the same hardware. For instance, a robot might be running a navigation stack, a perception pipeline, and a Holochain conductor simultaneously. Using micro-VMs would allow each of these applications to be isolated in its own secure environment, preventing them from interfering with each other and making it easier to manage resource allocation. Furthermore, the fast boot times of micro-VMs like Firecracker make them well-suited for dynamic and ephemeral workloads, which are common in serverless and edge computing scenarios. While the performance overhead of micro-VMs is slightly higher than that of traditional containers, it is significantly lower than that of full VMs, making them a viable option for performance-sensitive applications like robotics. The choice to include Firecracker and Kata in the development environment indicates a forward-looking approach to building a secure and resilient distributed system.

2.3.2. Initial Setup and Configuration Considerations

Integrating micro-VMs like Firecracker and Kata into the `ros2-humble-env` requires careful consideration of the setup and configuration process. Unlike Docker, which has a mature and well-established ecosystem, the tooling for micro-VMs is still evolving. The initial setup will likely involve installing the necessary hypervisor software and

configuring the system to support nested virtualization, which is a prerequisite for running VMs inside a container. For Kata Containers, this involves installing the Kata runtime and configuring the Docker daemon to use it as an alternative to the default runc runtime. This can be done by adding a new runtime configuration to the Docker daemon's configuration file (`/etc/docker/daemon.json`) and then specifying the runtime when running a container with the `--runtime=kata-runtime` flag.

For Firecracker, the setup process is more involved, as it is a lower-level technology that requires more manual configuration. It typically involves creating a JSON configuration file that specifies the VM's resources (CPU, memory), kernel image, and root filesystem. The Firecracker binary is then used to start the micro-VM based on this configuration. While this process is more complex than using Docker, it provides a high degree of control over the VM's configuration. In the context of the `ros2-humble-env` , it may be possible to use a tool like Weave Ignite, which provides a more user-friendly interface for running Firecracker VMs. Another consideration is the management of VM images. Just as Docker uses images to create containers, micro-VMs require images for their root filesystems. These images can be created from scratch or by converting existing Docker images. The process of building and managing these images will be a key part of the setup and configuration workflow. As the ecosystem for micro-VMs matures, it is likely that more user-friendly tools and workflows will emerge, making it easier to leverage their benefits in complex distributed systems like the one envisioned for this project.

3. Designing the Holochain DNA for Distributed Robotics

The design of the Holochain DNA is the most critical step in building a decentralized robotics application. The DNA is the blueprint for the entire system, defining the data structures, the rules of interaction, and the public API that ROS2 nodes will use to communicate. A well-designed DNA will be modular, secure, and efficient, enabling the development of complex and scalable applications. The process of designing the DNA involves defining its overall structure, implementing the core zomes in Rust, and packaging the final product for deployment. This section will provide a detailed guide to each of these steps, drawing on best practices from the Holochain developer community and adapting them for the specific needs of a distributed robotics system.

3.1. Defining DNA Structure and Zomes

The first step in designing the DNA is to define its high-level structure. This involves creating a `dna.yaml` file that specifies the properties of the DNA, such as its name,

description, and version, and lists the zomes that it contains. A zome is a module of executable code that defines a specific piece of functionality. It is the smallest unit of modularity in a Holochain application . For a distributed robotics system, it is advisable to adopt a modular approach, with separate zomes for different aspects of the system. This makes the DNA easier to develop, test, and maintain. For example, you might have a `perception` zome for handling sensor data, a `navigation` zome for managing robot movement, a `task` zome for coordinating multi-robot activities, and a `devops` zome for system configuration and management. Each zome would be responsible for its own data types and validation rules, and they could interact with each other through inter-zome calls . This modular design promotes code reuse and allows for a clear separation of concerns.

3.1.1. Creating a `dna.yaml` Configuration File

The `dna.yaml` file is the manifest for the DNA. It is a YAML file that provides metadata about the DNA and defines its composition. A typical `dna.yaml` file for a distributed robotics application might look like this:

yaml 复制

```yaml
---
manifest_version: "1"
name: ros2_holochain_bridge
description: A Holochain DNA for bridging ROS2 with a decentralized
network.
version: "0.1.0"
integrity:
  network_seed: "ros2_holochain_bridge_seed"
  properties: {}
  zomes:
    - name: perception
      hash: ~
      bundled: "target/wasm32-unknown-
unknown/release/perception.wasm"
    - name: navigation
      hash: ~
      bundled: "target/wasm32-unknown-
unknown/release/navigation.wasm"
    - name: task
      hash: ~
      bundled: "target/wasm32-unknown-unknown/release/task.wasm"
    - name: devops
```

```
      hash: ~
      bundled: "target/wasm32-unknown-unknown/release/devops.wasm"
coordinator:
  zomes:
    - name: perception_coordinator
      hash: ~
      bundled: "target/wasm32-unknown-
unknown/release/perception_coordinator.wasm"
      dependencies:
        - name: perception
    - name: navigation_coordinator
      hash: ~
      bundled: "target/wasm32-unknown-
unknown/release/navigation_coordinator.wasm"
      dependencies:
        - name: navigation
    - name: task_coordinator
      hash: ~
      bundled: "target/wasm32-unknown-
unknown/release/task_coordinator.wasm"
      dependencies:
        - name: task
    - name: devops_coordinator
      hash: ~
      bundled: "target/wasm32-unknown-
unknown/release/devops_coordinator.wasm"
      dependencies:
        - name: devops
```

This file defines a DNA with four integrity zomes and four corresponding coordinator zomes. The integrity zomes define the data schema and validation rules, while the coordinator zomes provide the public API that ROS2 nodes will call. The `bundled` field specifies the path to the compiled WASM file for each zome. The `network_seed` is a crucial parameter that determines the unique identity of the network. All agents that want to participate in the same network must use the same DNA, which means they must have the same `network_seed` .

3.1.2. Designing Zomes for Core Functionality

The design of the zomes is where the core logic of the application is implemented. As mentioned earlier, a modular approach is recommended. Each zome should have a clear and focused responsibility. For example, the `perception` zome would be responsible for handling all sensor data. It would define entry types for different types

of sensor readings (e.g., `Image`, `PointCloud`, `Pose`) and provide functions for creating and retrieving these entries. The `navigation` zome would handle robot movement, with entries for waypoints, paths, and navigation goals. The `task` zome would be responsible for coordinating multi-robot activities, with entries for tasks, assignments, and status updates. The `devops` zome would handle system configuration and management, with entries for configuration parameters, software versions, and deployment commands. For each zome, it is good practice to create a pair of zomes: an integrity zome and a coordinator zome . The integrity zome defines the data types and validation rules, while the coordinator zome provides the public functions that interact with the data. This separation of concerns makes the code more organized and easier to maintain.

3.1.3. Specifying Entry Types for ROS2 Messages and System State

Entry types are the fundamental building blocks of a Holochain application. They define the structure of the data that is stored on the DHT. For a ROS2 integration, it is necessary to define entry types that can represent the various ROS2 messages and system states. For example, you might define an entry type for a `Twist` message (used to represent velocity commands) like this:

rust 复制

```rust
use hdk::prelude::*;

#[hdk_entry(id = "twist")]
#[derive(Clone)]
pub struct Twist {
    pub linear_x: f64,
    pub linear_y: f64,
    pub linear_z: f64,
    pub angular_x: f64,
    pub angular_y: f64,
    pub angular_z: f64,
}
```

This Rust struct defines the fields of the `Twist` message. The `#[hdk_entry(id = "twist")]` attribute is used to register this struct as a valid entry type for the zome. The `id` parameter specifies the unique identifier for the entry type. The `#[derive(Clone)]` attribute is used to automatically implement the `Clone` trait for the struct, which is required by the HDK. By defining entry types for all the relevant ROS2

messages and system states, you can create a rich and expressive data model for your distributed robotics application.

3.2. Implementing Core Zomes in Rust

Once the DNA structure and entry types have been defined, the next step is to implement the core zomes in Rust. This involves writing the zome functions that will be exposed to the outside world, as well as the validation logic that will ensure the integrity of the data on the DHT. The Holochain Development Kit (HDK) provides a set of Rust macros and functions that make it easy to write zomes. The HDK handles the low-level details of interacting with the Holochain Conductor, allowing developers to focus on the application logic.

3.2.1. Zome for ROS2 Topic Management (Publish/Subscribe)

The `ros2_topics` zome is responsible for implementing the publish/subscribe communication pattern of ROS2. It would need to provide zome functions for publishing messages to a topic and for subscribing to a topic to receive messages. The `publish` function would take a serialized ROS2 message as input, create a new entry on the agent's source chain, and then publish it to the DHT. The `subscribe` function would query the DHT for all entries of a specific type (i.e., all messages on a specific topic) and return them to the caller. The following is an example of a `publish_pose` function for the `ros2_topics` zome:

rust 复制

```rust
use hdk::prelude::*;

#[hdk_extern]
pub fn publish_pose(pose: PoseStamped) -> ExternResult<HeaderHash> {
    let entry = EntryTypes::PoseStamped(pose);
    let header_hash = create_entry(entry)?;
    Ok(header_hash)
}

#[hdk_extern]
pub fn get_poses(_: ()) -> ExternResult<Vec<PoseStamped>> {
    let links =
get_links(AnyLinkableHash::from(agent_info()?.agent_initial_pubkey),
LinkTypes::Pose)?;
    let poses: Vec<PoseStamped> = links
        .into_iter()
```

```
        .filter_map(|link| {
            let entry_hash = link.target.into_entry_hash()?;
            let element = get(entry_hash,
GetOptions::default()).ok()??;
            element.entry().to_app_option::<PoseStamped>().ok()?
        })
        .collect();
    Ok(poses)
}
```

This code defines two zome functions: `publish_pose` and `get_poses`. The `publish_pose` function takes a `PoseStamped` struct as input, creates a new entry from it, and commits it to the agent's source chain. The `get_poses` function queries the DHT for all `PoseStamped` entries linked to the agent's public key and returns them as a vector.

3.2.2. Zome for Distributed Inference and Compute Tasks

The `distributed_compute` zome would handle the distribution of computational tasks, such as machine learning inference, across the network. It would define entry types for compute tasks and results, and provide zome functions for submitting tasks and retrieving results. A compute task could be represented by an entry containing the task description, the input data, and the ID of the agent that submitted the task. The zome would also provide a function for agents to claim tasks from a queue and another function for them to submit the results of their computation. The validation rules for this zome would ensure that only authorized agents can submit tasks and that the results are valid.

3.2.3. Zome for Agentic DevOps and System Configuration

The `agentic_devops` zome would manage the system configuration and software deployment. It would define entry types for configuration parameters and software packages, and provide zome functions for updating the configuration and deploying new software. For example, a configuration update could be represented by an entry containing the new configuration parameters and a signature from an authorized agent. The zome would provide a function for agents to retrieve the latest configuration and another function for them to apply it to their own systems. The validation rules for this zome would ensure that only authorized agents can make configuration changes and that the new configuration is valid.

3.3. Compiling and Packaging the DNA

Once the zomes have been implemented, the final step is to compile them into WebAssembly (WASM) and package them into a DNA bundle. This process is handled by the Holochain CLI tool (`hc`) and the Rust compiler (`cargo`).

3.3.1. Compiling Zomes to WebAssembly (WASM) with `cargo`

The first step is to compile each zome into a WASM module. This is done using the `cargo` command with the `wasm32-unknown-unknown` target. The following command will compile a zome located in the current directory:

bash 复制

```bash
cargo build --release --target wasm32-unknown-unknown
```

This will create a `.wasm` file in the `target/wasm32-unknown-unknown/release/` directory. This file contains the compiled code for the zome, which can then be loaded into the Holochain Conductor.

3.3.2. Packaging WASM Files into a DNA Bundle with `hc dna pack`

Once all the zomes have been compiled, they can be packaged into a DNA bundle using the `hc dna pack` command. This command takes a `dna.yaml` file as input and creates a `.dna` file that contains all the WASM files and the DNA configuration. The following command will create a DNA bundle from a `dna.yaml` file in the current directory:

bash 复制

```bash
hc dna pack
```

This will create a `.dna` file in the current directory. This file is the final product of the DNA development process and can be installed into the Holochain Conductor.

3.3.3. Versioning and Managing DNA Releases

It is important to version and manage DNA releases carefully, as any change to the DNA will result in a new network. This is because the DNA's hash is used as the unique identifier for the network. If the DNA changes, the hash will change, and the new DNA will not be compatible with the old network. Therefore, it is a good practice to use a version control system like Git to track changes to the DNA and to tag each release

with a version number. This will make it easier to manage different versions of the DNA and to deploy them to different networks.

4. Integrating ROS2 Nodes with the Holochain DNA

The integration of ROS2 nodes with the Holochain DNA is the final step in building a decentralized robotics application. This involves creating a bridge between the ROS2 and Holochain ecosystems, developing a ROS2 package that uses this bridge, and managing the dependencies and build configuration for the package. This section will provide a detailed guide to each of these steps, with a focus on the practical aspects of implementing the integration.

4.1. The ROS2–Holochain Bridge: Implementation Approaches

The ROS2–Holochain bridge is the key component that enables communication between the two systems. There are several ways to implement this bridge, each with its own trade–offs in terms of performance, modularity, and ease of use.

4.1.1. Creating a Dedicated ROS2 Node as a Holochain Client

One approach is to create a dedicated ROS2 node that acts as a Holochain client. This node would be responsible for all communication with the Holochain Conductor, and it would expose a set of ROS2 topics and services that other nodes can use to interact with the Holochain network. This approach has the advantage of being modular and easy to understand, as all the Holochain–specific code is contained in a single node. However, it can also become a bottleneck for performance, as all communication with the Holochain network must go through this single node.

4.1.2. Utilizing the Holochain Conductor's Application API

The bridge will interact with the Holochain Conductor through its Application API, which is exposed over a WebSocket connection. The bridge will need to use a client library to connect to this API and to call the zome functions defined in the DNA. The Holochain project provides official client libraries in JavaScript and Rust, but for a ROS2 Humble environment, a C++ or Python client library would be more appropriate. While an official C++ client is not currently available, it is possible to create a custom client library using a WebSocket library and a MessagePack library.

4.1.3. Handling Asynchronous Communication and Signals

Both ROS2 and Holochain are asynchronous systems, so the bridge will need to be designed to handle asynchronous communication. This can be done using callbacks, futures, or other asynchronous programming patterns. The Holochain Conductor also supports a signaling mechanism, which allows it to push notifications to the client when new data is available. The bridge can use this mechanism to receive real-time updates from the Holochain network, which can then be published as ROS2 messages.

4.2. Developing the ROS2 Integration Package

Once the bridge has been implemented, the next step is to develop a ROS2 package that uses the bridge to interact with the Holochain network. This package will contain the ROS2 nodes that perform the actual work of the application, such as controlling a robot or processing sensor data.

4.2.1. Creating Custom ROS2 Message Types for Holochain Data

It is often necessary to create custom ROS2 message types to represent the data that is stored in the Holochain DHT. This is because the data structures in Holochain may not map directly to the standard ROS2 message types. By creating custom message types, you can ensure that the data is represented in a way that is both efficient and easy to use in your ROS2 application.

4.2.2. Implementing Publishers that Commit Data to the DHT

The ROS2 nodes that produce data, such as sensor nodes, will need to be modified to commit their data to the Holochain DHT. This can be done by adding a call to the bridge's `publish` function in the node's code. The `publish` function will take the data as input, serialize it, and then call the appropriate zome function to commit it to the DHT.

4.2.3. Implementing Subscribers that Query Data from the DHT

The ROS2 nodes that consume data, such as navigation nodes, will need to be modified to query the Holochain DHT for the data they need. This can be done by adding a call to the bridge's `subscribe` function in the node's code. The `subscribe` function will take the topic name as input, query the DHT for all entries related to that topic, and then publish them as ROS2 messages.

4.3. Managing Dependencies and Build Configuration

The final step is to manage the dependencies and build configuration for the ROS2 integration package. This involves adding the Holochain client library to the package's dependencies and configuring the `CMakeLists.txt` and `package.xml` files to build the package correctly.

4.3.1. Adding Holochain Client Libraries to the ROS2 Package

The Holochain client library will need to be added to the ROS2 package's dependencies. This can be done by adding the library to the `package.xml` file and by installing the library in the development environment. If a custom client library is being used, it will need to be built and installed as part of the package's build process.

4.3.2. Configuring `CMakeLists.txt` and `package.xml` for Integration

The `CMakeLists.txt` and `package.xml` files will need to be configured to build the ROS2 integration package correctly. This will involve adding the necessary dependencies, setting up the build targets, and installing the package's files. The specific configuration will depend on the structure of the package and the dependencies that it uses.

4.3.3. Integrating with the Nix and Docker Build Process

The ROS2 integration package will need to be integrated with the Nix and Docker build process. This will involve adding the package to the Nix configuration and to the Docker image. This will ensure that the package is built and deployed correctly as part of the overall system.

5. Application-Specific Implementation: XR/MR, Robotics, and DevOps

With the foundational architecture and environment in place, this section explores the application-specific implementations for the three core use cases: decentralized robotics with XR/MR glasses, distributed inference and compute, and agentic DevOps. These examples demonstrate how the integrated Holochain-ROS2 system can be used to build powerful and resilient applications.

5.1. Decentralized Robotics with XR/MR Glasses

This use case focuses on using XR/MR glasses as a natural interface for controlling and collaborating with a team of robots. The glasses act as a Holochain agent, processing user input and sharing perception data with the robotic fleet.

5.1.1. Processing Voice and Vision Data on Edge Devices

The XR/MR glasses, equipped with cameras and microphones, would run a ROS2 node that captures and processes voice and vision data locally on the device. This edge processing is crucial for reducing latency and preserving user privacy. For voice commands, a speech-to-text engine would convert the user's speech into text strings. For vision, computer vision algorithms would analyze the camera feed to identify objects, gestures, or gaze direction. The results of this local processing—such as a parsed command like "move to the red box" or a 3D point indicating where the user is looking—would then be formatted into a standard ROS2 message.

5.1.2. Sharing Perception Data and Robot States via the DNA

The processed voice and vision data, now in the form of ROS2 messages, would be passed to the Holochain bridge node on the XR glasses. This node would commit the data as entries to the DHT, making it available to all other agents in the network. For example, a `UserCommand` entry could contain the parsed voice command, while a `GazeTarget` entry could contain the 3D coordinates of the user's point of interest. Simultaneously, each robot in the fleet would be publishing its own state (e.g., pose, battery level, current task) to the DHT via their own bridge nodes. This creates a shared, decentralized "world model" that is accessible to all agents, including the XR glasses and other robots.

5.1.3. Coordinating Multi-Robot Tasks through the DHT

With a shared world model, complex multi-robot tasks can be coordinated through the DHT. For example, a user could look at a specific robot and say, "you, pick up that object." The XR glasses would publish this command to the DHT, referencing the specific robot's ID and the object's location. The targeted robot's navigation and manipulation nodes would be subscribed to the DHT and would receive this command, allowing it to autonomously plan and execute the task. Other robots would also be aware of this command and could adjust their own behavior accordingly, for example, by yielding right-of-way or by offering assistance. This creates a fluid and natural collaboration between the human user and the robotic team, all mediated by the secure and resilient Holochain network.

5.2. Distributed Inference and Compute

This use case leverages the collective computational power of all devices in the network to perform large-scale inference tasks, such as training a machine learning

model or running a complex simulation.

5.2.1. Defining Compute Tasks as Holochain Entries

A compute task would be defined as a specific entry type in the DNA, for example, `InferenceJob`. This entry would contain all the necessary information for a device to perform the computation, such as the model to be used, the input data (or a link to it), and the specific operation to be performed. A user or a robot could submit a new job by committing a `InferenceJob` entry to the DHT.

5.2.2. Distributing Inference Workloads Across User Devices

Devices in the network that have available computational resources (e.g., idle CPU/GPU cycles) would monitor the DHT for new `InferenceJob` entries. When a device finds a job it can perform, it would "claim" the job, perhaps by creating a link from the job entry to its own agent ID. This prevents other devices from working on the same job simultaneously. The device would then download the model and input data, perform the inference, and prepare the results.

5.2.3. Aggregating and Validating Results on the DHT

Once a device completes an inference task, it would commit the results as a new entry, for example, `InferenceResult`, and link it back to the original `InferenceJob` entry. The DNA's validation rules could be used to ensure the quality of the results. For example, the same job could be sent to multiple devices, and the results would only be considered valid if a certain number of devices return the same answer (a form of Byzantine fault tolerance). This allows the system to leverage the power of the crowd while maintaining a high degree of trust in the final output.

5.3. Agentic DevOps

This use case applies the principles of decentralization and agent-centricity to the management of the robotic system itself, creating a self-healing and self-organizing infrastructure.

5.3.1. Using the DNA for Decentralized Configuration Management

System-wide configuration parameters (e.g., network settings, safety limits, feature flags) would be stored as entries in the DHT. A dedicated `devops` zome would manage these configurations, with validation rules to ensure that only authorized agents (e.g., a lead developer's key) can propose changes. Other agents in the network

would monitor the DHT for configuration updates and apply them to their local systems autonomously.

5.3.2. Logging System Events and Health Data to the Source Chain

Each robot and device would act as an agent responsible for its own health monitoring. Critical system events, such as software crashes, hardware failures, or performance degradation, would be logged as entries on the agent's source chain. This creates an immutable and auditable record of the system's health history. This data can be aggregated from the DHT to provide a real-time overview of the entire fleet's status.

5.3.3. Enabling Autonomous Software Updates and Deployments

Software updates can be distributed through the DHT. A new software version could be packaged and committed to the DHT by a DevOps agent. Other agents would discover this update, validate its signature, and autonomously decide whether to install it based on rules defined in the DNA (e.g., only install updates that have been signed by a trusted key and have passed a set of automated tests reported by other agents). This creates a secure, decentralized, and automated deployment pipeline that is resilient to network partitions and does not rely on a central server.

6. Deployment and Network Orchestration

The final stage of the integration process is deploying the system to the target environment and orchestrating the network of agents. This involves configuring the Holochain Conductor, setting up networking for peer discovery, and establishing monitoring and debugging procedures.

6.1. Running the Holochain Conductor in the Target Environment

The Holochain Conductor must be properly configured and running on each device in the network. This section details the steps for setting up the Conductor in the `ros2-humble-env`.

6.1.1. Configuring the Conductor for the `ros2-humble-env`

The Conductor is configured using a YAML file. This file specifies the network settings, storage paths, and the APIs to be exposed. For the `ros2-humble-env`, the configuration file would need to be included in the Docker image and would need to be tailored to the specific requirements of the containerized environment. For example, it

would need to specify the correct paths for the Conductor's data directory and for the DNA file.

6.1.2. Installing and Running the DNA within the Conductor

Once the Conductor is running, the DNA must be installed into it. This can be done using the Conductor's Admin API. A setup script could be created that uses the Admin API to install the DNA and to create a new agent key for the device. This script would be run when the container is first started, ensuring that the device is ready to join the network.

6.1.3. Managing Agent Keys and Identities Across Devices

Each device in the network must have a unique agent key. These keys are used to sign all actions taken by the agent, providing cryptographic provenance for all data published to the DHT. The keys should be securely stored on the device, for example, in a hardware security module or in an encrypted file system. A key management strategy should be developed to handle the lifecycle of agent keys, including key generation, rotation, and revocation.

6.2. Networking and Peer Discovery

A critical aspect of deploying a distributed system is ensuring that the agents can discover and communicate with each other. This involves configuring both the Holochain peer-to-peer network and the ROS2 DDS network.

6.2.1. Configuring Peer-to-Peer Networking for Holochain Agents

Holochain agents discover each other using a combination of bootstrap nodes and a gossip protocol. The Conductor's configuration file must specify the addresses of one or more bootstrap nodes. These are well-known, always-on agents that new agents can connect to in order to discover the rest of the network. The configuration file can also specify the use of proxy servers to help with NAT traversal.

6.2.2. Ensuring ROS2 DDS Discovery Functions Across the Network

As discussed in the environment setup section, ROS2's DDS discovery can be challenging in complex network environments. The use of a discovery server is a recommended approach for large-scale deployments. The DDS configuration for each ROS2 node must be updated to point to the discovery server, ensuring that all nodes can find and communicate with each other.

6.2.3. Strategies for NAT Traversal and Firewall Configuration

In many real-world deployments, the devices in the network will be behind NATs and firewalls. This can make it difficult for them to establish direct peer-to-peer connections. Holochain's use of proxy servers can help with this, but it may also be necessary to configure the firewalls to allow traffic on the ports used by the Conductor and the DDS implementation. A clear set of networking requirements should be documented to guide the deployment process.

6.3. Monitoring and Performance Considerations

Once the system is deployed, it is important to monitor its performance and to have tools in place for debugging any issues that may arise.

6.3.1. Analyzing Latency and Throughput for Robotics Control Loops

For many robotics applications, low latency is critical. The latency of the Holochain-ROS2 integration should be carefully measured and analyzed to ensure that it meets the requirements of the application. This can be done by instrumenting the code with timestamps and by using tools like `ros2 topic delay` to measure the end-to-end latency of message delivery.

6.3.2. Monitoring DHT Storage Growth and Network Overhead

The DHT will grow over time as more data is published to it. It is important to monitor the storage usage of the DHT on each device to ensure that it does not consume all available disk space. The network overhead of the gossip protocol should also be monitored to ensure that it does not saturate the network bandwidth.

6.3.3. Tools for Debugging the Integrated System

A variety of tools can be used to debug the integrated system. The `ros2` command-line tools can be used to inspect the ROS2 network, while the Holochain Conductor's logs can be used to debug issues with the Holochain layer. A centralized logging system, such as the ELK stack, could be used to collect and analyze logs from all the devices in the network, providing a holistic view of the system's health.