



Agent based distributed system demonstrator

DEMO: Supply Chain
v 1.2 - 14 Jun 2025



KATLAS
Technology

Table of Contents

Table of Contents.....	2
Overview.....	6
 Access to the interactive demonstrator.....	7
 Setup steps for Microsoft Windows.....	7
Install PuTTY.....	7
Configure connections.....	7
Launching Sessions and Connecting to Wallets.....	8
 Setup steps for Linux with GNOME Terminal.....	9
 Design: Background information.....	10
 The Daemon.....	10
 Command-Line Interface (CLI).....	10
 Graphical and Web Interfaces.....	10
 Secure Communications.....	10
 Efficient Data Transport.....	10
 Network.....	10
 Deployment Architecture and Component Layers.....	10
Wallet daemon process.....	11
 Concepts.....	13
 Identity.....	13
 Personality.....	13
 p9y.....	13
 Agent.....	13
 Agent View and Connection Status Indicators.....	14
 Connection Status Icons and Meanings.....	14
 Cryptographic Server Authentication Without DNS.....	14
 Endpoint view.....	15
 Certificates.....	15
 What It Contains.....	16
 Alignment with Verifiable Credentials.....	16
 Wallet CLI.....	17
 User: custom wrapper for wallet CLI.....	19
 function: user restart: Restart the system.....	19
 function: user bm: Bookmarks.....	20
 function:user st: actor dashboard.....	21
 function:user watch: actor dashboard (Live Mode).....	22
 Per-Role Screen Breakdown.....	23
 Role: Ship. Actors: ship1.....	23
 Note for Windows / PuTTY Users.....	23
 Ship Dashboard quick overview.....	24
 Me – Personality.....	24
 Agents – Connections with Other Actors.....	24
 Location Map.....	24
 Navigation Status.....	24
 Itinerary.....	24
 Activity Status.....	24
 Cargo Summary.....	24
 Ship Dashboard Detailed Overview.....	25
 Header.....	25
Blue Header Bar.....	25
 Me – Personality (p9y).....	25
 Personality Storage.....	25
 Agents View.....	25
 Shared Trade Context via tid.....	26
On ship1: wallet trade ChdEx3gcuiT84Aqyxre1wcMWCuu show data↵.....	26
 Interpreting the Trade Data.....	27
 Location map.....	28
 Activity status.....	28
 Cargo.....	29

 watch.....	29
 Screenshot.....	29
 Role: Port. Actors: port1, port2.....	30
 Header.....	30
 Agents.....	30
 Handling Pipeline.....	31
Container queues initial status (Port1):.....	31
 Handling & Operations Pipeline State Machine.....	32
 Handling Pipeline (Yellow States).....	32
♦ stock.....	32
♦ available.....	32
♦ reserved.....	32
♦ opened.....	32
 Operations Pipeline (Blue States).....	33
♦ departures.....	33
♦ arrivals.....	33
 Container Readiness & Bootstrapping.....	33
 Operations Pipeline.....	33
 Departures.....	33
 Arrivals.....	33
 Interactions.....	34
1. Ship Sets Port as Manager.....	35
2. Unloading Cargo.....	35
3. Loading Cargo.....	35
 watch.....	35
 Screenshot.....	35
 Role: Container. Actors: container1, container2, container3.....	36
 Header.....	36
 Agents.....	36
 Section stakeholders.....	36
 Initial (Orphan) State.....	37
 Manager Authority.....	37
 Section Activity.....	37
 Section State.....	37
 State Machine Overview.....	38
 Available.....	38
 Opened.....	38
 Sealed.....	38
 Section Payload.....	38
 Screenshots for other statuses.....	39
 Interactions.....	40
 Summary of Key Phases.....	40
1. Assignment & Sealing.....	40
2. Transit Notification.....	40
3. Certificate Exchange.....	40
4. Arrival & Unsealing.....	41
 Highlights.....	41
 watch.....	41
 Screenshot.....	41
 Role: User. Actors: alice, bob.....	42
 Header.....	42
 Agents.....	42
 User Activities.....	42
 Container Manager (CM).....	43
 Outbound Workflow Logic.....	43
 Timing Behavior.....	43
 Order Manager (OM).....	44
 Order status.....	45
 Interactions.....	45
 Users agreement - Dialogue Flow.....	45

✉ Messages.....	45
📦 Hire a Container – Interaction Flow.....	46
✉ Messages.....	46
✅ Contextual Role.....	46
🔒 Open/Seal Lifecycle – Role User Message Flow.....	46
✉ Messages.....	47
✅ Contextual Notes.....	47
📄 Access to certificates.....	47
✉ Messages.....	48
✅ Honest Actors (Green Zone):.....	48
❌ Malicious Actor (Blue Zone):.....	48
👤 Issuing our first order - Step-by-Step Example.....	48
🚀 Step-by-Step Example.....	48
📦 Entering the Payload.....	49
🥚 Easter Egg Warning.....	49
👤 Recipient Selection.....	49
📄 Send Order Widget (Alice's Dashboard).....	51
👁 Receive Order Widget (Bob's Dashboard).....	51
🔍 Observations.....	51
📦 Let's Check on container1.....	52
🏗 Let's Peek Into port1.....	53
📺 Prepare to Track the Delivery Journey.....	54
👤 Delivery Walkthrough.....	55
👁1. Alice: New order was being agreed with Bob: dialing.....	55
🔍 Key Observations:.....	55
👁2. Alice: New order was being agreed with Bob: chatting.....	56
🟡 Highlights:.....	56
👁3. Alice: New order was being agreed with Bob: agreed.....	57
🟢 Key Points:.....	57
👁4. Alice: Container sealed and in transit.....	58
🟢 Key Indicators:.....	58
👁5. Port1: container in departures queue.....	58
📦 Key Observations:.....	58
👁6. Container1: sealed. With Payload and recipient information.....	59
🔒 Key Highlights:.....	59
👁7. Ship1: arriving at port1.....	60
🚢 Key Observations:.....	60
👁8. Ship1: sailing towards port2. Missed capture of container to see it managed by ship1.....	61
🌊 Key Details:.....	61
👁9. Port2: Container in arrivals queue.....	62
📖 Key Observations:.....	62
👁10. Bob: missed capture while unsealing, finally delivered and goods in Bob's possession :) 🚀.....	63
📦 Key Takeaways:.....	63
👁11. Port2: container in available queue after bob finished unloading the container and released it..	64
✅ Key Details:.....	64
👁12. Container1: available status managed by port2.....	64
🟢 Key Observations:.....	64
📺 watch.....	65
📺 Screencast.....	65
📈 Scaling Up Workload: Parallel Operations and Throughput.....	65
🔊 R2R API Overview.....	66
👤 Symmetric R2R protocol: User-to-User.....	66
👤🏗 Asymmetric R2R protocol: User-to-Port.....	66
👤🏗 User side.....	66
👤🏗 Port side.....	66
👤📦 Asymmetric R2R protocol: User-to-Container.....	66
👤📦 User side.....	66
👤📦 Container side.....	66
Functions Implemented in library libtrader-u2c-c.so.....	66
🏗 Asymmetric R2R protocol: Container-to-Port.....	67

  	Container side.....	67
  	Port side.....	67
 	Asymmetric R2R protocol: Port-to-Ship.....	67
 	Port side.....	67
 	Ship side.....	67
 	Asymmetric R2R protocol: Container-to-Ship.....	67
 	Container side.....	67
 	Ship side.....	67
	AI integration Exercise - Chat with Port.....	68
	Let's Try It.....	68
	Useful Commands.....	68
 	New actors.....	69
	Adding a New Actor: Eve. Linux account, wallet daemon, p9y.....	69
	Bookmarks: Establishing Initial Trust and Visibility.....	70
	Surface attack scenario: Attempted Unauthorized Certificate Access.....	71
	Attack Simulation Brief.....	71
	step 1.....	71
	Attack warm up.....	71
	inspect our bookmarks.....	71
	Attempting to Seal with a Crafted Payload.....	73
	Payload Accepted –  Action Rejected.....	74
	This attempted attack is properly blocked by the backend.....	74
	Code Insight: Seal Authorization Logic.....	75
	Attempted Unseal: Blocked by Authorization Logic.....	75
	Step-by-Step Plan.....	76
	User: Advanced/developer/insight functions.....	80
	functions: user down and user up.....	80
	function: user logtree.....	81
	Codebase Metrics (via SLOCCount).....	83
	Development Effort Estimation (COCOMO Model).....	83
	Conclusions: Alignment with Evaluation Objectives.....	84
	Use Case 1: Blockchain for Supply Chain Data Visibility.....	84
	Use Case 2: Confidential, AI-Powered Knowledge Agent.....	84
	Critical Considerations Addressed.....	84
	Desired Outcomes.....	85
	Platform Assessment Summary.....	85

Overview

This demonstration presents a foundational scenario built around key actors, their roles, and the peer-to-peer interactions that define their workflows—powered by our proprietary P2P node technology.

Our platform has been engineered from first principles to support high-performance, decentralized systems. Developed entirely in modern C++20, it offers exceptional efficiency, scalability, and concurrency—critical for real-world deployments.

Beyond its technical strength, the software stack provides a robust foundation for building domain-specific ecosystems. Through modular design and well-defined architectural patterns, it enables rapid adaptation to industry-specific needs.

In this instance, we showcase how the technology models and facilitates fundamental processes in the supply chain—demonstrating both its technical capabilities and its relevance to real-world business applications.



Screencasts:



https://www.youtube.com/playlist?list=PLxosfK0Tg_rWlddsiz4Nad40EJmMgIM5G



Access to the interactive demonstrator

To interact with the wallets assigned to each actor in the scenario, secure access is facilitated via SSH clients. This ensures authenticated, encrypted communication with the node infrastructure, aligning with best practices for operational security and access control in decentralized systems.

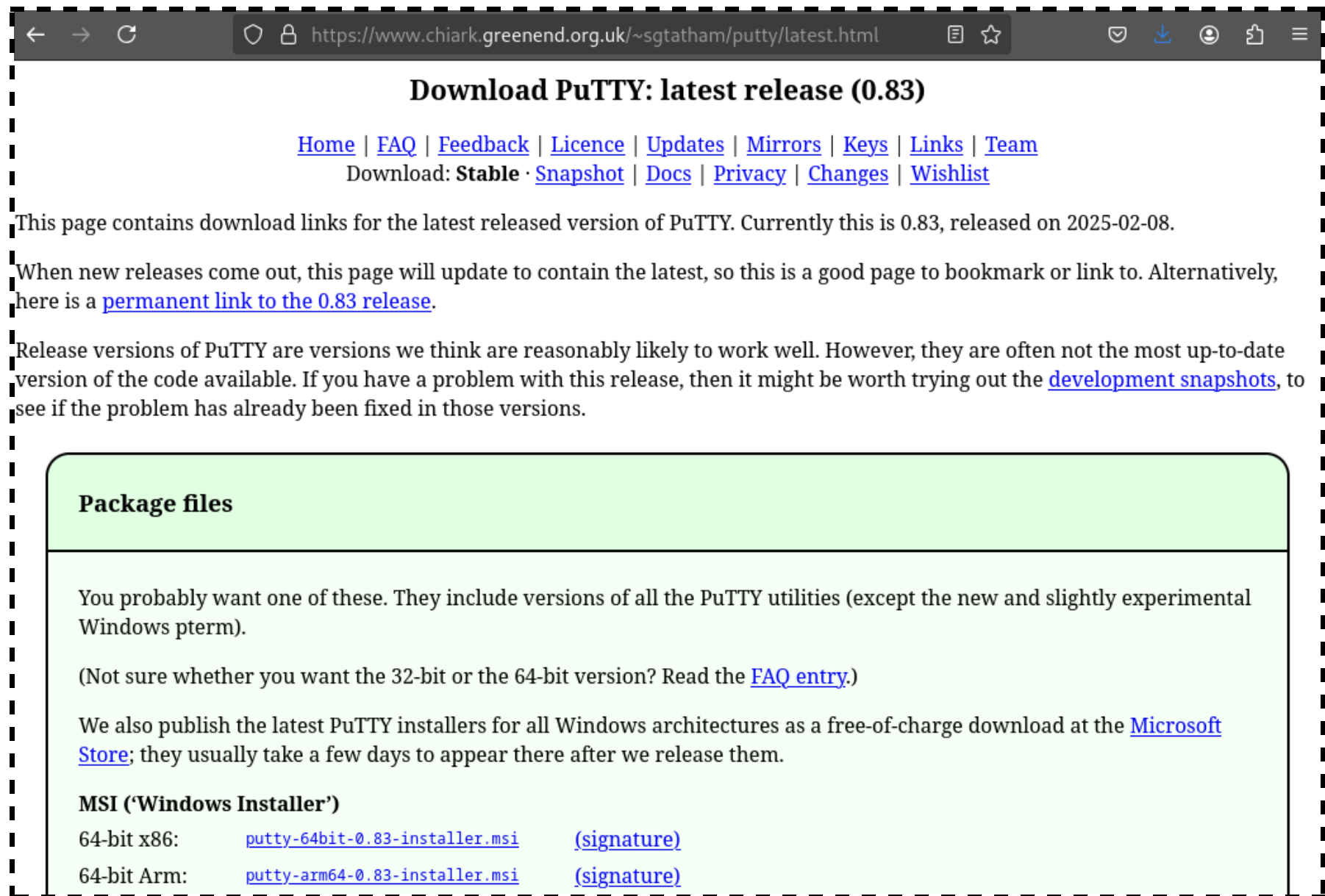
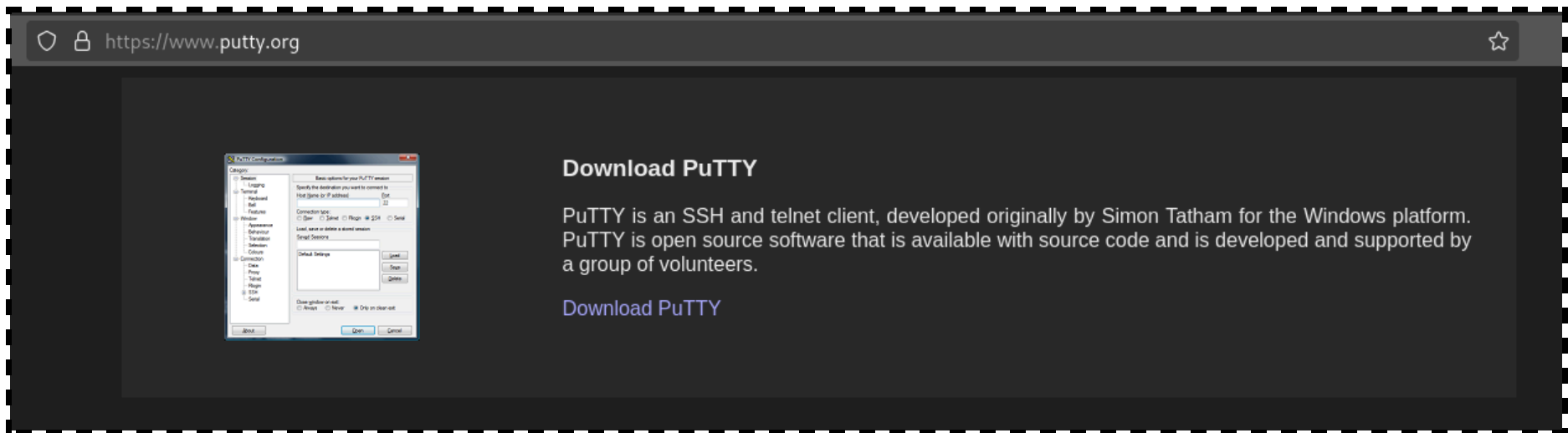


Setup steps for Microsoft Windows



Install PuTTY.

PuTTY is a widely used SSH client for Windows, providing secure access to remote systems. You can download and install it from the official website: <https://www.putty.org>



Configure connections

After installing PuTTY, launch the application to begin configuring SSH connections for each actor.

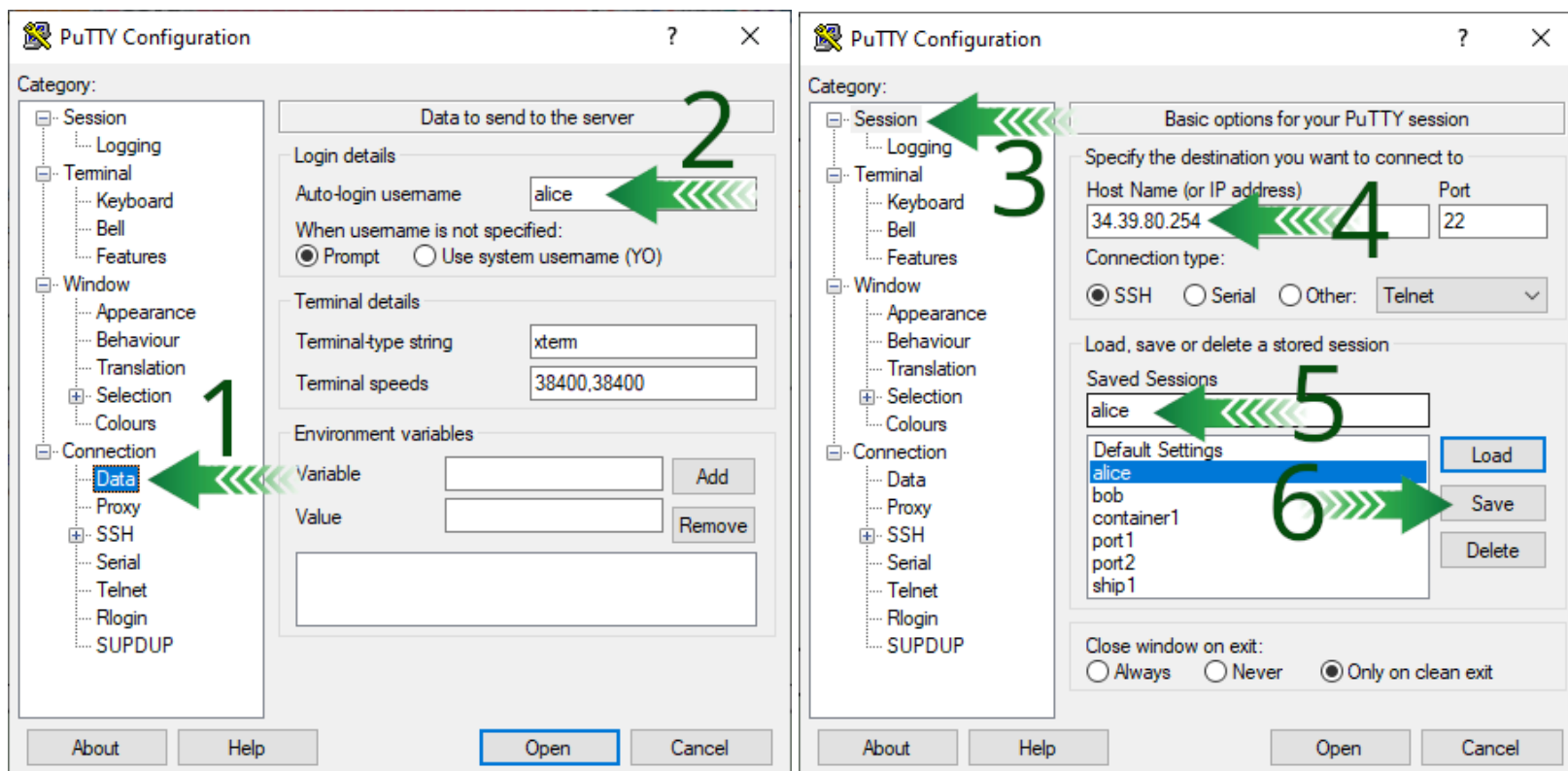
Each actor is associated with a unique Linux user account on the host `34.39.80.254`. The usernames are all lowercase and must be entered accurately to ensure successful authentication.

Follow the 6-step configuration process described below for each of the following users:

- alice
- bob
- port1
- port2
- ship1
- container1

Optionally, you may also configure additional connections for **container2** and **container3**.

⚠ Note: Ensure the correct username (e.g., **alice**) is entered in **Step 2**, as SSH login will fail if it is misspelled or incorrectly capitalized.



Launching Sessions and Connecting to Wallets

With the initial configuration complete, you can open multiple PuTTY windows—each representing a different actor. To begin a session:

1. Select the desired actor from your list of saved sessions.
2. Click **“Load”** to apply the configuration.
3. Click **“Open”** to initiate the connection.

When prompted, enter the password that was provided to you through a separate, secure channel. The first time you connect to the server, a security prompt will appear to confirm the host's identity—accept this to proceed.

To connect all actors, repeat this process for the following users:

- **alice**
- **port1**
- **container1**
- **ship1**
- **port2**
- **bob**

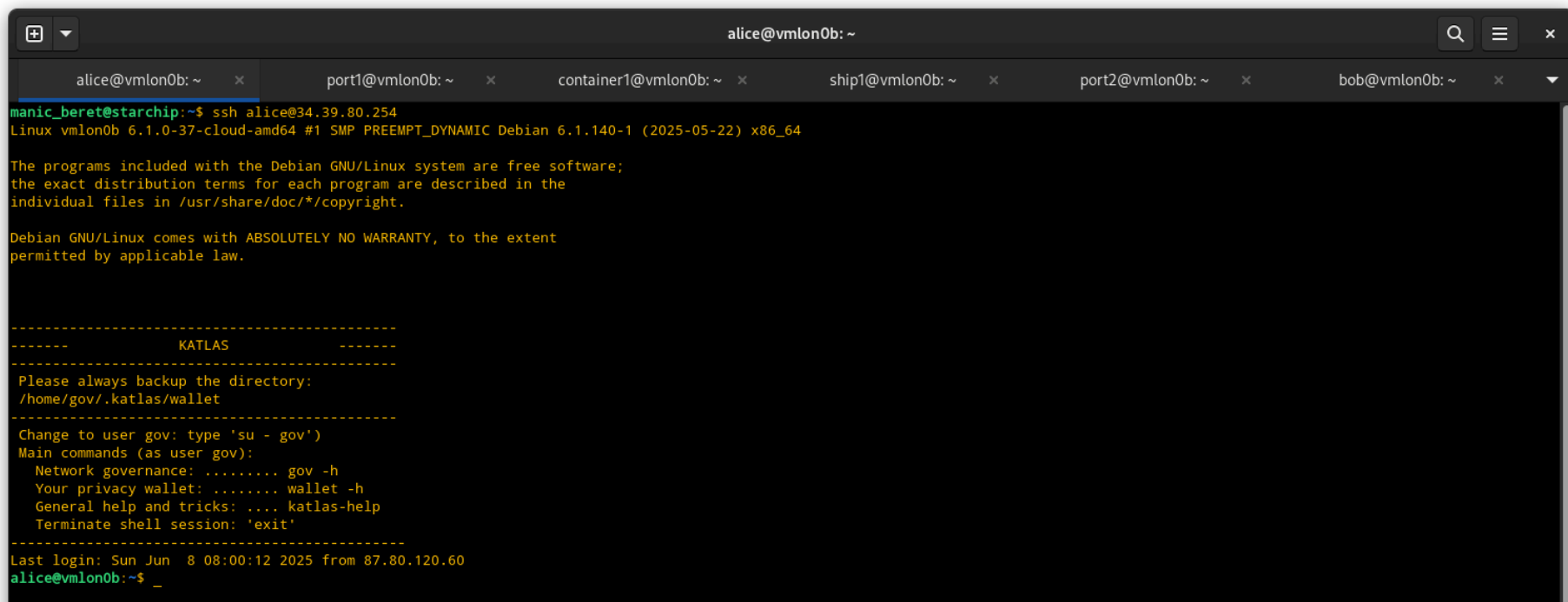
You should then have six active terminal windows, each representing a connected actor wallet.

Optional: For easier management of multiple windows, consider using a tabbed PuTTY utility such as [ttyplus Multi-Tabbed Putty](https://ttyplus.com/multi-tabbed-putty) at <https://ttyplus.com/multi-tabbed-putty>



Setup steps for Linux with GNOME Terminal

On Linux systems, a multi-tab terminal emulator such as **GNOME Terminal**, **Konsole**, or **Tilix** can be used to manage multiple sessions more efficiently. Each actor can be connected in a separate tab using standard SSH commands.



```
alice@vm1on0b: ~
manic_beret@starchip:~$ ssh alice@34.39.80.254
Linux vm1on0b 6.1.0-37-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.140-1 (2025-05-22) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

-----
-----          KATLAS          -----
-----
Please always backup the directory:
/home/gov/.katlas/wallet
-----
Change to user gov: type 'su - gov')
Main commands (as user gov):
  Network governance: ..... gov -h
  Your privacy wallet: ..... wallet -h
  General help and tricks: .... katlas-help
  Terminate shell session: 'exit'
-----
Last login: Sun Jun  8 08:00:12 2025 from 87.80.120.60
alice@vm1on0b:~$
```

Connect to all wallets using this correspondence table:

```
Tab1: ssh alice@34.39.80.254
Tab2: ssh port1@34.39.80.254
Tab3: ssh container1@34.39.80.254
Tab4: ssh ship1@34.39.80.254
Tab5: ssh port2@34.39.80.254
Tab6: ssh bob@34.39.80.254
```

This order intentionally aligns with the container journey, which visits alice, port1, ship1, port2 in sequence.

Design: Background information

The Daemon

The daemon is the backbone of the wallet system, designed to run continuously as a high-availability service. Its key responsibilities include:

- **Custody of Private Data** – Secure management of each user's cryptographic keys and sensitive information.
- **Peer-to-Peer Trading** – Execution of transactions through autonomous, role-based agents acting on behalf of user interests.
- **Backend Services** – Exposure of APIs that support front-end applications such as CLIs, mobile apps, and GUIs.

Command-Line Interface (CLI)

The CLI acts as a lightweight front-end for direct human interaction (Human-Machine Interface – HMI). It can be executed on any machine—local or remote—relative to its corresponding daemon. It offers essential capabilities for controlling, querying, and interacting with the wallet system.

Graphical and Web Interfaces

Graphical user interfaces (GUIs), mobile applications, and web-based front ends can be seamlessly integrated, providing user-friendly and purpose-specific extensions to the platform.

Secure Communications

All communications—whether between wallets or between a CLI/GUI and its daemon—are secured using **secp256k1**, an elliptic curve cryptographic standard renowned for its robustness and efficiency. This cipher, also used in Bitcoin, secures billions of dollars daily and is one of the most heavily vetted cryptographic standards in existence.

Efficient Data Transport

Unlike conventional solutions that rely on HTTP/HTTPS with verbose formats like XML or JSON, our platform communicates using compact binary packets and datagrams. This design eliminates the overhead associated with format translation and reduces both network bandwidth and CPU consumption—delivering a lean, high-performance messaging layer built for decentralized applications.

Network

Deployment Architecture and Component Layers

Our software pipeline produces a set of **relocatable components** designed for deployment flexibility. These components can run as multiple processes on a single machine—as demonstrated in this demo—or be distributed across multiple machines in production environments to support scalability, isolation, and performance.

The architecture is structured into **three conceptual layers**, each represented with distinct color codes for clarity:

1. **Public Blockchain Node Layer** (*light yellow*)

These components form the foundation of the public blockchain layer. They are responsible for:

- **Consensus verification**, ensuring agreement across the distributed network.
- **On-chain data interaction**, including state queries and transaction submission.
- **Cryptographic validation** of transactions and evidence.

In our design abstraction, **transactions are treated as evidence**—verifiable records applied to financial settlements. This perspective aligns with the broader goal of building trust-minimized systems where every action is backed by cryptographic proof.

2. **Private Trading Network Node Layer** (*intense yellow*)

This layer encompasses the core components of our proprietary peer-to-peer trading infrastructure and privacy-preserving wallets. It operates entirely off-chain, ensuring fast, secure, and confidential interactions.

Key elements include:

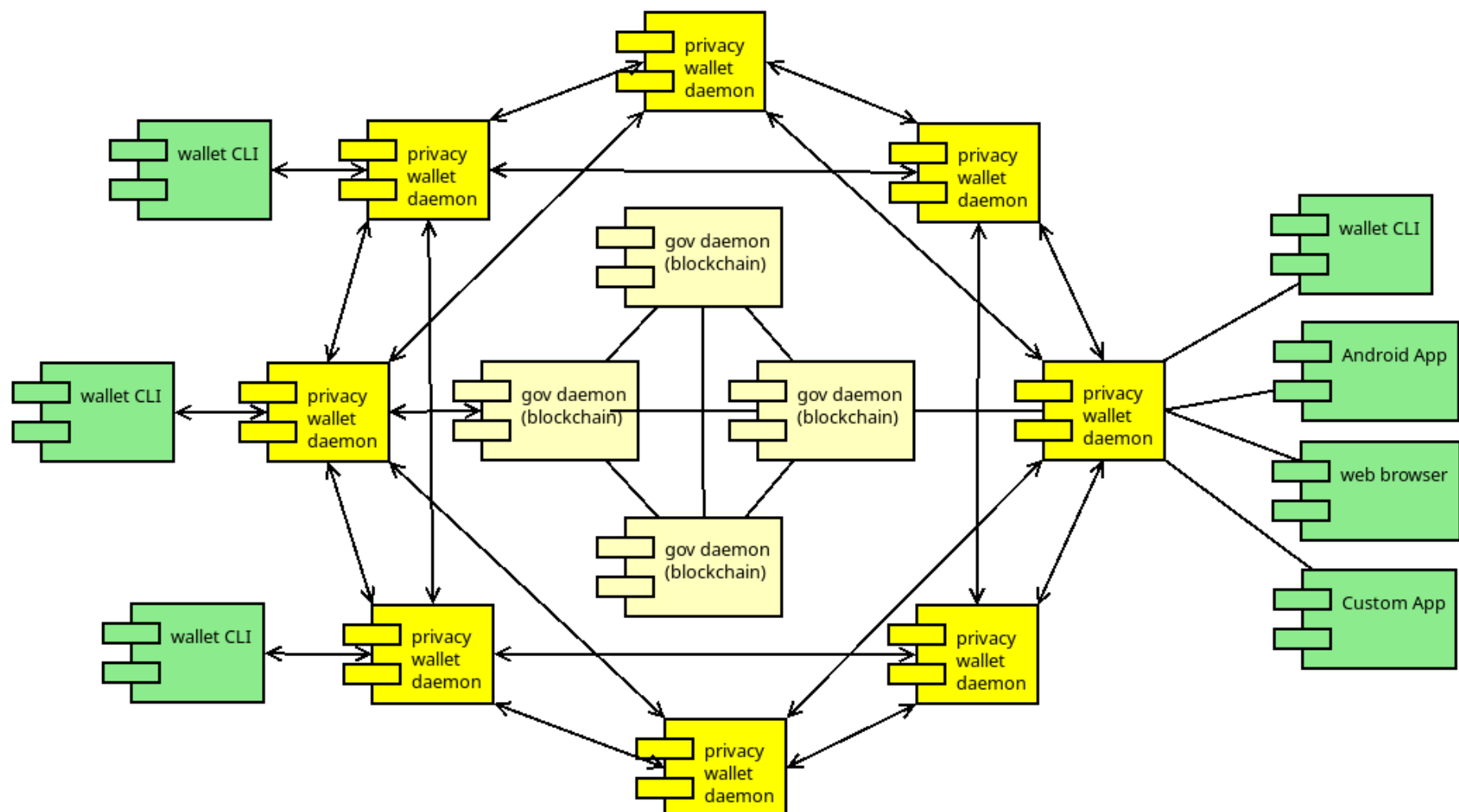
- **Daemons** that provide persistent services and maintain wallet state.
- **Agent infrastructure** responsible for executing role-based logic and negotiating trades on behalf of users.

- **Wallet coordination mechanisms** that manage identities, sessions, and secure data exchanges between peers.

Together, these components enable decentralized trading workflows while preserving user autonomy and privacy.

3. Human-Machine Interfaces (HMI) (green)

User-facing components such as the **Command-Line Interface (CLI)** and **Graphical User Interfaces (GUI)**. These allow human actors to interact with their wallets, manage roles, and trigger actions through intuitive interfaces.



Wallet daemon process

This component is responsible for **safeguarding private user data** and for executing **negotiators**, also referred to as *agents*, which act on behalf of specific **roles and actors** in the system.

Secure Execution and Plugin-Based Extensibility

It features a **plugin-based architecture** that allows seamless extensibility. New behaviors, strategies, or domain-specific logic can be introduced by installing plugins as shared libraries.

For this demonstration, a custom set of plugins has been developed and deployed to power the participating actors. These are installed in the following directory:

```
/home/gov/.katlas/wallet/trader/lib/
```

```
root@vmlon0b:~# ls -la /home/gov/.katlas/wallet/trader/lib/
-rwxr-xr-x 1 gov gov 194688 Jun  8 15:57 librbz-c.so
-rwxr-xr-x 1 gov gov 120800 Jun  8 15:57 librbz-p.so
-rwxr-xr-x 1 gov gov 109680 Jun  8 15:57 librbz-s.so
-rwxr-xr-x 1 gov gov 307912 Jun  8 15:57 librbz-u.so
-rwxr-xr-x 1 gov gov 154480 Jun  8 15:57 libtrader-c2p-c.so
-rwxr-xr-x 1 gov gov 153768 Jun  8 15:57 libtrader-c2p-p.so
-rwxr-xr-x 1 gov gov 153800 Jun  8 15:57 libtrader-c2s-c.so
-rwxr-xr-x 1 gov gov 144472 Jun  8 15:57 libtrader-c2s-s.so
-rwxr-xr-x 1 gov gov 156312 Jun  8 15:57 libtrader-p2s-p.so
-rwxr-xr-x 1 gov gov 156312 Jun  8 15:57 libtrader-p2s-s.so
-rwxr-xr-x 1 gov gov 240720 Jun  8 15:57 libtrader-u2c-c.so
-rwxr-xr-x 1 gov gov 245080 Jun  8 15:57 libtrader-u2c-u.so
-rwxr-xr-x 1 gov gov 154904 Jun  8 15:57 libtrader-u2p-p.so
-rwxr-xr-x 1 gov gov 149584 Jun  8 15:57 libtrader-u2p-u.so
-rwxr-xr-x 1 gov gov 160056 Jun  8 15:57 libtrader-u2u-u.so
-rwxr-xr-x 1 gov gov 292440 Jun  8 15:57 libworkflow-logistics.so
```

Plugin Types and Their Responsibilities

The system supports three distinct types of plugins, each serving a specific function within the architecture:

1. **librbz-** Plugins

These define the **business logic** associated with each role. The naming convention reflects the role they implement:

- **c** – Container
- **p** – Port
- **s** – Ship
- **u** – User

Each plugin encapsulates the autonomous behavior and decision-making logic required for agents acting under that role.

2. **libtrader-** Plugins

These implement the **protocols** used by P2P agents during interactions with other roles. Protocols may be:

- **Symmetric** – Both peers follow the same protocol logic
 - Example: **libtrader-u2u-u.so** (User-to-User negotiation)
- **Asymmetric** – Each peer follows a role-specific part of the protocol
 - Example: **libtrader-u2p-u.so** (User initiating a protocol with Port)

This structure supports flexible, role-aware communication patterns between agents.

3. **libworkflow-** Plugins

These define the **domain-specific objects** exchanged between roles. In the context of logistics, this includes:

- Definitions for **container payloads**
- **Certificate templates** used for validation and authorization
- Other structured artifacts relevant to the domain

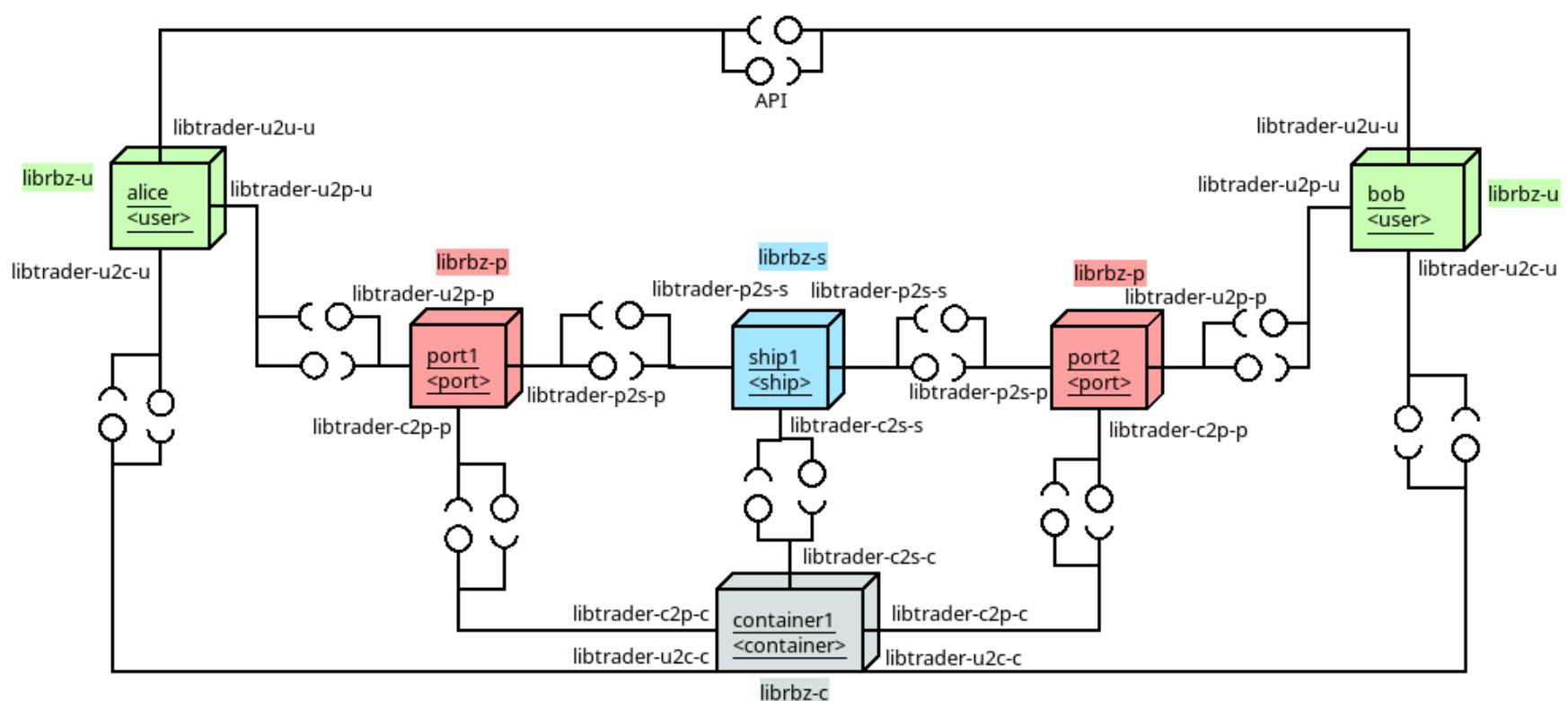
These plugins abstract the application layer semantics, enabling clear separation between protocol mechanics and domain data.

Plugin Placement in the Ecosystem

The following illustration provides a visual overview of where each type of plugin operates within the ecosystem. It highlights how the libraries are integrated across the system's layers and interactions:

- **librbz-** plugins drive **role-specific logic** within each agent. (labels with coloured background)
- **libtrader-** plugins govern the **protocols of interaction** between agents. (labels with white background)
- **libworkflow-** plugins define the **structured data** exchanged during domain operations. (objects transmitted (lines))

This visual mapping clarifies how each library contributes to the overall coordination, negotiation, and data flow within the decentralized network.



Concepts

Identity

In traditional systems, **identity** is a mechanism used by governments and organizations to uniquely identify and track individuals or users. Such systems exhibit the following characteristics:

- **Uniqueness:** Each person is assigned a single identifier (e.g., a national ID number) linked to sensitive data such as name, address, date of birth, and photograph. This identifier is issued and recorded in a centralized ledger controlled by the organization.
- **Weak Privacy:** Centralized identity systems pose significant privacy risks. If the ledger is compromised—whether by poor security practices or external attacks—large volumes of sensitive information may be exposed. These systems represent high-value targets and a single point of failure, often requiring expensive cyber-defense measures.
- **Custodial / Non-Sovereign:** Identity credentials, including digital identities based on **Public Key Infrastructure (PKI)**, are typically issued and controlled by the organization. This means public/private key pairs are generated by the issuing authority, not the individual. As a result, users lack sovereignty over their own identities from the outset, with private keys often stored by third parties.

This traditional identity scheme is **incompatible with our technology stack**, which was designed after critically examining legacy systems in search of innovations capable of advancing digital autonomy and societal infrastructure.

In contrast, **sovereign wallets** are built to operate **exclusively in the interest of the holder**, without surrendering control to third parties. Ownership, access, and authority remain entirely with the wallet holder.

Personality

To distinguish our concept from conventional identity systems, we refer to it as **personality**—an evolved construct grounded in digital sovereignty and privacy. It embodies the following traits:

- **Multiplicity**
Individuals can generate an **unlimited number of personalities**, similar to how Bitcoin addresses are created. Each personality is derived by generating a large random secret (a private key), never disclosed to anyone, and computing a corresponding public key via cryptographic means (PKI). This public key acts as the personality's public identifier.
- **Seamless Integration**
This model does not disrupt existing institutions. Organizations or governments may continue to operate their ledgers, provided they can record or acknowledge the personalities that individuals choose to disclose in online or real-world activities.
- **Sovereign by Design**
The model upholds the fundamental right of **autonomy and reinvention**. Individuals can start anew, maintain separate spheres of activity, and develop multiple, independent public profiles, histories, or reputations—all without centralized oversight.
- **Trade-offs and Considerations**
The support for multiplicity introduces **challenges for systems** that rely on a one-to-one mapping between identifiers and individuals, such as law enforcement databases or Sybil-attack-sensitive applications. However, these systems should be redesigned to adapt—**enhancing individual freedoms and privacy** rather than sacrificing them in the name of systemic convenience or safety.

p9y

The term **p9y** is a **numeromin**—a shorthand notation formed by combining the first and last letters of a word with the number of characters omitted in between. In this case, “**p9y**” represents the word “**personality**”.

Numeromins are a convenient way to abbreviate long terms while preserving their recognizability, commonly used in technical contexts (e.g., “i18n” for “internationalization”).

Agent

An **agent** is an autonomous actor responsible for executing a **Role-to-Role (R2R) protocol** in negotiation with a remote peer. These agents operate on behalf of specific roles within the system, driving peer-to-peer interactions.

- Each negotiation instance is called a **trade** and is uniquely identified by a **trade ID (tid)**, which is **shared by both parties** involved in the negotiation.
- **Trades may be short-lived or persistent.** In the case of longer engagements, the trade state is preserved and can be **resumed by either party** at any time—even after disconnection, whether due to intentional logout or unexpected network failure.
- Because the **tid** is a shared identifier, **both parties can independently query or audit the state** of the trade from their local perspectives, enabling robust synchronization and traceability.

This architecture ensures continuity, reliability, and transparency in decentralized negotiations.

Agent View and Connection Status Indicators

The **agent view** is presented on screen through a dedicated widget that displays the current state of active or archived trades. Each trade is identified by its **trade ID** ( **tid**), accompanied by an icon that visually represents the **connection status** with the remote peer.



Connection Status Icons and Meanings

The icon besides tid represents the connection state with the following translation codes:

Icon	Status Code	Description
	off	Trade is archived
	offline	Peer is not connected
	req_online	In process or getting online
	connect_failed	Network error
	connected	Connection established but peer not yet authenticated
	online	R2R protocols can securely use the connection
	req_offline	Going offline on our request
	gone_offline	Gone offline for other causes
	dying	In process of being archived

When a trade reaches the **online**  state, it indicates that the **counterparty's identity has been cryptographically verified**. At this stage, a secure handshake has occurred involving a **Zero-Knowledge Proof (ZKP)**, which confirms that the remote party possesses the **secret key corresponding to the expected p9y** (personality), without revealing the key itself.

This ensures mutual authentication and establishes a trusted channel for executing Role-to-Role (R2R) protocols securely.

Cryptographic Server Authentication Without DNS

To protect against **phishing and impersonation** in environments where connections are made to a **raw IP address**—and traditional hostname-based TLS validation is not applicable—we employ a cryptographic challenge-response mechanism that eliminates the need for third-party trust anchors.

Authentication Flow:

1. **Challenge Issuance:**
Upon connection, the client generates a unique challenge containing a **random nonce** and a **timestamp** to prevent replay attacks.
2. **Server Signature:**
The server signs this challenge using its **private key**, associated with its known **p9y** (personality).

3. Client Verification:

The client verifies the returned signature using the server's **public p9y key**.

- If the signature is **valid**, the server is authenticated and the session proceeds.
- If the signature is **invalid**, the connection is terminated immediately.

Security Benefits:

- Prevents impersonation **even if the IP address is spoofed or DNS is bypassed**.
- Ensures **peer authenticity** without relying on third-party certificate authorities or traditional identity infrastructure.
- Reinforces the sovereignty of the system by using **self-issued p9y keys** instead of external identity proofs.

This approach offers a **robust, decentralized alternative** to mainstream TLS/DNS identity models, preserving the core principle of autonomy inherent in the personality-based architecture.

Endpoint view

In addition to the **trade ID (tid)**, the endpoint view displays information about the **remote actor's endpoint**. This data is typically obtained from a **QR code** and is used to identify the actor **before** any connection is established or trade is initiated.

A key element in this view is the **p9y field**, which represents the **personality** of the remote actor. As described earlier, this personality serves as a **cryptographic identifier** and is expected to be **validated upon connection**.

This allows secure, pre-authenticated interactions, ensuring the remote party can later prove ownership of their declared identity through cryptographic means.

```
QR: 4VrH1C6JxGctXaGoqbr8jYKY9ZFp p9y: 3vK6C38Th6BA5C5msvPnVr6p5pBp (container1)
```

Certificates

In the context of this system, a **certificate** is a **digitally signed document** that describes the contents of a shipment—essentially, a formal **manifest** of the goods included in a delivery order.

Each certificate is:

- **Created by the sender**
- **Digitally signed** using their cryptographic private key
- **Stored** in the wallet's certificate authority (CA) directory
- **Attached** to the container before sealing

All issued certificates are securely stored in the wallet's local **Certificate Authority (CA) directory**, organized by hash-based subdirectories for scalability and performance.

To view the list of certificate files, use the following command: `find .katlas/wallet/trader/ca/ -type f`

```
alice@vmlon0b:~$ find .katlas/wallet/trader/ca/ -type f
.katlas/wallet/trader/ca/cert/hE/cj/jw/1B/FC/D5/mF/pk/o2/2j/r1/zY/2/hEcjjw1BFCD5mFpko22jr1zY23g
.katlas/wallet/trader/ca/cert/2R/Bo/78/ic/7x/9t/Mh/49/5J/hq/Jo/SR/dK/2RBo78ic7x9tMh495JhqJoSRdKqk
.katlas/wallet/trader/ca/cert/mk/Bf/3W/Zc/zZ/FK/VZ/ZK/JJ/Eg/tb/c3/s/mkBf3WZczZFKVZZKJJEgtbc3sMb
.katlas/wallet/trader/ca/cert/2K/cq/2M/P3/fK/LE/JY/Zk/jH/t1/sh/V7/J/2Kcq2MP3fKLEJYZkjHt1shV7JYx
.katlas/wallet/trader/ca/cert/mr/cL/t2/5y/iP/wb/Mm/78/FQ/ZB/NC/46/U/mrcLt25yiPwbMm78FQZBNC46UWC
```

or, using the wallet-cli:

```
alice@vmlon0b:~$ wallet cert list
2RBo78ic7x9tMh495JhqJoSRdKqk cert
mrcLt25yiPwbMm78FQZBNC46UWC cert
hEcjjw1BFCD5mFpko22jr1zY23g cert
mkBf3WZczZFKVZZKJJEgtbc3sMb cert
2Kcq2MP3fKLEJYZkjHt1shV7JYx cert
```

Filenames are derived from the certificate's **hash-based ID**, which appears in summaries and ledger views.

 The structure is designed to prevent file system bottlenecks and support high volumes of certificates efficiently.

Certs and be consulted, check wallet help, section Certificates for more functions.

```
alice@vmlon0b:~$ wallet cert get hEcjjw1BFCD5mFpko22jr1zY23g
-- BEGIN OF DOCUMENT -----
Certificate
Document date: ~Mon May 26 2025 21:41:24 UTC
Expires: ~Sat Aug 16 2059 21:41:24 UTC

Parameters:
  CA: SOA Start of Authority
  lang: en
  version: 1

Content:
  Tesla Cybertruck
  Honda Accord
  Lamborghini Countach

Signed by: 3TaDdWk9s1GP8MyoCtK2NsEceabk
Public key: 22RwGaoQLAmviq4VjUujVbbjf23ieUgum4XyWx51JMcqt
Signature: 6421BZuUFBq1nuVE1JiobH6zrhg91UTRmdYrVADwPaarhebEEHjqMZRUtBMavujqyLs5uAfq3yTWX1rMFMiB1Rik
[INFO] Cryptographic verification succeeded. Data is legitimate.
-- END OF DOCUMENT -----
```

 What It Contains

A typical certificate includes:

- Document metadata (timestamp, expiry date, issuer)
- Parameters (language, version, issuing authority)
- **Payload content:** list of goods being shipped
- **Cryptographic signature** of the issuer
- Public key of the issuer for independent verification

 Alignment with Verifiable Credentials

This certificate design is conceptually and technically aligned with the **W3C Verifiable Credentials (VC) standard**, in the following ways:

VC Principle	How This Certificate Aligns
Issuer Identity	The sender signs the certificate using their private key, verifiable by the recipient via the sender’s public key.
Tamper-Proof Claims	The content (item list, metadata) is cryptographically bound to the signature, ensuring integrity.
Decentralized Verification	Any party with the public key (e.g., recipient or container) can verify the authenticity of the certificate without relying on a central authority.
Selective Disclosure	Certificates are only released to authorized parties (e.g., sender, recipient), and requests from unauthorized actors (like Eve) are denied.
Extensible Schema	Metadata fields (e.g., language, expiration, CA, parameters) allow domain-specific adaptation and future expansion.

The official reference for the **Verifiable Credentials Data Model** standard is published by the W3C:

 **Verifiable Credentials Data Model 1.1 (W3C Recommendation)**
<https://www.w3.org/TR/vc-data-model/>

This specification defines the data model and associated terminology for expressing, exchanging, and verifying credentials on the web in a secure and privacy-preserving way



Wallet CLI

The **Wallet CLI** is a fast and efficient RPC client written in **modern C++**, providing direct control over the **core wallet functionality** and its **extensible components**. It features a front-end **plugin-based architecture**, enabling the seamless addition of new **Role-to-Role (R2R) protocols** as the ecosystem evolves, in parallel with the backend plugins running in the daemon.

To access the list of available commands and usage information, simply type:

wallet

For this **supply chain demo**, we will focus primarily on a subset of commands—especially those in the **roles** and **trade** groups. However, to highlight the full potential of this smart wallet, the **entire help screen output** is included in this document. This provides a broader view of the wallet’s extensibility and protocol capabilities.

```
alice@vmlon0b:~$ wallet
katlas-wallet (rpc client). Channel 0. Copyright (C)
    This program comes with LIMITED WARRANTY. For details type 'show w'.
    Obtain a license from KATLAS Technology.. Type 'show c' for details.
    version: katlas-dj6e03fc71_main_90077c.f09438 90077caeaf9cec53dbb76263b290d712864e5f92 2025-06-07_09-14-25
    monotonic versions:
    deploy: 12
    api_v gov: 9
    api_v wallet: 13
    binary serialization: 11
    component brandcodes:
    deployment blobs:11
    rpc-client blobs:
    android Zerrq9u56
    console Zerrq9u56
    local time: 1749328800452323032 ns since 1/1/1970
    tx time shift: 15 seconds.
    Build configuration: [with logs] [logdir /var/log/katlas] [optimized build]
    Parameters:
    channel: 0
    home: /home/alice/.katlas
    this is an optimized build.
    warning: logs are ON. Writing files in /var/log/katlas
    run rpc shell
    wallet daemon at: localhost:16573
    connect_for_recv 1
    stop_on_disconnection 1
    remote wallet is non-custodial.
Usage: katlas-wallet [options] [command]
Options:
    General parameters:
    -b
    -home <homedir>
    -d
    -e <edges>
    -v <edges>
    -w <workers>
    -lp <port>
    -pp <port>
    -pin <PIN number>
    -c <channel>
    -B <connection_blob>
    --custodial <sString>
    -log
    -logd <dir>
    -nolog
    -dd <downloads_dir>
    -n
    -nb
    -bhost <address>
    -bp <port>
    -whost <address>
    -wp <port>
    -t <secs>
    -wr <n>
    .
    Basic mode.
    homedir. [/home/alice/.katlas]
    Runs daemon (backend) instead of an rpc-client.. [16673]
    Max num walletd neighbours. [8]
    Max num device neighbours. [4]
    Threadpool size. [4]
    Listening Port. (-d). [16673]
    Published listening port.. [16673]
    Connect using PIN.. [0]
    System channel. [0]
    User the given connection blob to connect to a wallet backend.
    (rpc) Select custodial wallet, or '-' for non-custodial..'-'
    Logs in screen..
    Directory for writting log files..
    Disable logs..
    Directory containing software updates blobs. [/home/gov/downloads]
    Omit field names in output. [true]
    Don't show the banner..
    us-gov IP address. [localhost]
    TCP port. [16672]
    walletd address. [localhost]
    walletd port. [16573]
    RPC timeout in seconds. [5]
    wait for n responses (push messages) before exiting. [0]

Commands:
    misc:
    id
    version
    shell

    cash: Query
    balance <0|1|2>

    recv_address

    cash: Functions for transfering to <dest account>
    transfer <dest account> <amount> <gas|coin>
    transfer_from <source account> <dest account> <amount> <gas|coin>

    cash: Two-step transactions
    invoice <recv account> <amount> <gas|coin> [-reward <award amount> <gas|coin>]
    pay <tx>

    cash: Coin/token creation
    set_supply <address> <amount>
    mint_seal <address>

    Transaction craft
    tx new
    tx add_section <tx> <token>
    tx add_input <tx> <section> <address> <amount>
    tx add_output <tx> <section> <address> <amount>
    tx make_sigcode exec_time <open|close> [s <section> [i <input>]* [o <output>]* ]
    [-]
    tx decode_sigcode <sigcode b58>
    tx ts <tx> <secs in future>
    tx sign_input <tx> <section> <input> <sigcode> <secret key>
    tx check <tx>
    tx send <tx>
    tx sign <tx> <sigcodeb58>
    tx decode <evidenceB58>
    tx decode -f <evidence_blob_file>

    keys:
    address new

    address add <privkey>
    list [show_priv_keys=0|1]
    gen_keys
    mine_public_key <string>
    priv_key <private key>
    pub_key <public key>
    digest <filename>
    sign -f <filename> <private key>
    sign "<message>" <private key>
    verify -f <filename> <pubkey> <signature>
    verify "<message>" <pubkey> <signature>
    encrypt -f <filename> <sender_private_key> <recipient_public_key>
    encrypt "<message>" <sender_private_key> <recipient_public_key>
    decrypt "<b58>" <sender_public_key> <recipient_private_key>
    hash add <hash1> <hash2>
    encode <file>

    Device pairing:
    device_id
    pair <pubkey> [<subhome>|-] [<name>]

    -----
    Print wallet id and exit..
    Print software version..
    Enters interactive shell. default command if no args..

    -----
    If detail is 0 it displays a collapsed view of accounts. 1 or 2 shows the breakd
    own by account with the given level of detail.
    Shows a suitable address for receiving transfers.

    -----
    Source account/s will be choosen automatically.
    Selectable source account.

    -----
    Creates an invoice (incomplete transaction).
    Pay invoice. Complete the transaction and send it over.

    -----
    Set or reset the coin mint balance.
    Seal the given mint making a zero-inflation coin..

    -----
    Creates an empty transaction.
    Creates a coin section in the given transaction.
    Creates an input in the specified section of the given transaction.
    Creates an output in the specified section of the given transaction.

    .
    Sets a new exec-time for the given transaction.
    Signs the given tx input.
    .
    .
    .
    Decode transaction.
    Decode transaction.

    -----
    Generates a new key-pair, adds the private key to the wallet and prints its asoc
    iated address.
    Imports a given private key in the wallet.
    Lists all keys. Default value for show_priv_keys is 0.
    Generates a key pair without adding them to the wallet..
    Creates a vanity address containing the specified string.
    Gives information about the given private key.
    Gives information about the given public key.
    Computes RIPEMD160+base58 to the content of the file..
    Sign file..
    Sign message.
    Verify signed file..
    Verify signed message.
    Encrypts file..
    Encrypts message.
    Decrypts message.
    Results in RIPEMD160(hash1, hash2)..
    Prints the content of a file in Base58 encoding..

    -----
    Show this device Id..
    Authorize a device identified by its public key.
```

```
unpair <pubkey>
prepair <pin|auto> [<subhome>|-|auto] [<name>]
```

```
unprepair <pin>
list_devices
```

```
net_info
```

```
Certificates:
create_personality_file <privkey> <moniker> <file>
print_personality_file <file>
cert <cmd>
"   create [-f msg_file] [-m b58]
"   import [-f cert] [-enc cert_b58]
"   list <0|1>
"   get <id> [-o file] [-enc]
```

```
Runtime wallet configuration files:
wallet connections:
make_wallet_connection <ip4addr> <port> <channel> <name> <subhome>
decode_wallet_connection <blob_b58>
wallet_connections_add <wallet_connections_file> <wallet_connection-blob-b58>
decode_wallet_connections [-f <wallet_connections_file> | <b58>]
default_wallet_connections
protocols:
create_protocols <prot> <role> <file>
protocols_add <protocols_file> <protocol> <role>
decode_protocols [-f <protocols_file> | <b58>]
roles:
create_roles <role> <file>
roles_add <roles_file> <role>
decode_roles [-f <roles_file> | <b58>]
bookmarks:
create_bookmark <title> <icofile|-> "[<channel>] <address>[.subhome]" <protocol>
-> <role|-> [<output-file>]
decode_bookmark [-f <filename> | <blob_b58>]
r2r:
r2r_index [-f <file>] [-e]
r2r_index_hdr
```

```
Public storage (on-chain):
key-value:
store <address> -kv <key> <value>
rm <address> -kv <key>
search <address> <word1|word2...>
file:
store <address> -f <file> [path]
rm <address> -f <hash> <path>
file <hash>
list <address> <path>
```

```
Private storage: (in-wallet)
timeseries list
timeseries <address> new
timeseries <address> add [-f <filename>|-b58 <b58>|"text line"]
timeseries <address> show [<timemark>]
```

```
Trader:
trade <command>
Trading help.
Commands:
```

```
[Agents / R2R Trades]
qr
save_gr_bookmark <name> <file>
list_protocols
startT <node address> <protocol> <role>
start_bm <bookmark index>
list
kill <trade#>
<trade#> [<subcommand>]
reload <full-path-doc>
wallet
```

```
[Service API]
exec [<cmd>]
```

```
[World]
world
bookmarks [file]
bookmarks_append <dstfile> <srcfile>
bookmarks_rename <file> <old-name> <new-name>
save_bookmark <name> <file>
load_bookmarks <file> <i>
```

```
Daemon control/monitoring:
```

```
s
gw
allw
regw <ip address>
sync
reload_doc <full-path-doc>
conf <key> [<value>]
```

```
Net-dev control:
patch_os <script file> <skey>
(Account for updates: 4NwEEwnQbnwB7p8yCBNkx9uj7lru)
```

```
Connectivity:
ping
```

```
Software:
version
h|-h|help|-help|--help
```

Revoke authorization to the specified device.
Pre-Authorize an unknown device identified by a pin number [1-65535]. (auto selects a random PIN). If subhome is 'auto' a new custodial wallet is created for the device. If subhome is '-' it pairs the main wallet..

Revoke Pre-authorization to the specified pin.
Prints the list of recognized devices, together with the list of recently unauthorized attempts log.
Prints contextual information and wallet type (root/guest).

-----.
.
.
Certificate/NFT functions..
Create certificate/NFT from payload (file or b58 encoded string)..
Import certificate/NFT from cert (file or b58 encoded string)..
Lists certificates/NFTs. 0-all; 1-issued..
Retrieve certificate/NFT by its id. Out put goes to file or screen, either encoded or human readable..

-----.
Build a connection blob..
Decode the given connection blob.
Append wc into container..
Decode the given connections file or encoded blob.
Print default wallet connections.

Create protocols file with the one entry given..
Append protocol selection to given file..
Decode the given protocols file or encoded blob.
Create roles file with the one entry given..
Append role to given file..
Decode the given roles file or encoded blob.
Display bookmark blob on screen, or to specified file..

Decode bookmark_t blob (from file or encoded) and shows its content on screen..

Obtain interesting bookmarks. Optionally save it to file or print encoded..
Print protocol_selections available in the r2r_index..

-----.
Store key-value pair in a funded account. cost: 1 per cycle.
Removes the key-value pair.
Search key-value table.

Store file in a funded account. cost: 1 per kibibyte per cycle.
Removes a file. <path> example: /.
Retrieves file content.
List files starting with path.

-----.
List accounts holding timeseries..
Creates a new time series in the given address..
Appends a new entry..
List all events registered. Or obtain its content if a timemark is given..

-----.
Access to trading functions.

-----.
Display my QRs.
Save to file (or stdout if no file) a bookmark of me by name.
Lists available trading protocols.
Initiates a new P2P private trade using endpoint.
Initiates a new P2P trade using bookmark entry. See command bookmarks below..
Lists all active trades.
Terminates the specified trade.
Enter a shell for the specified trade.
Tell all active trades the specified file changed on disk.
Exit trading shell going back to wallet shell.

-----.
Exec command. Empty cmd for help.

-----.
Lists remote wallets.
Print my bookmarks (or those in input file)..
Append bookmarks in srcfile into bookmarks file dstfile.
Rename entry in bookmarks file.
Append to file (or stdout) a bookmark by name.
Load bookmark from file. Select bookmark #i in in file or -1 for all.

-----.
Show socket connections.
Show neighbours.
Show all wallets.
Register this wallet reachable at the specified address..
Reload files.
Invoke reload on all active trades in all wallets. Only via root wallet.
get/[change] config by key..

-----.
System upgrade/maintenance. Requires governance key.

-----.
Endpoint check.

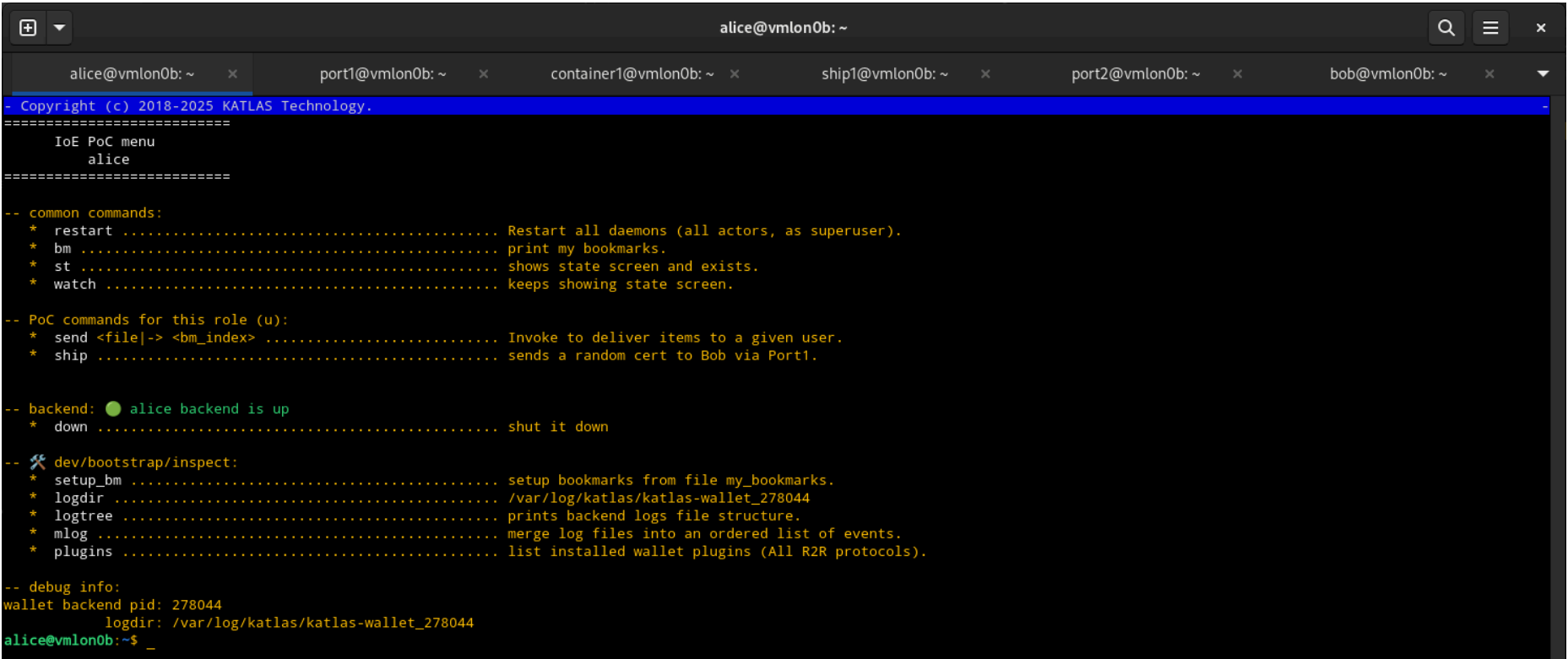
-----.
Print version..
This help screen.

User: custom wrapper for wallet CLI

The **User CLI** is a specialized utility designed specifically for this use case. It acts as a **wrapper** around the core **wallet** CLI, simplifying interactions by encapsulating common command patterns.

To view available commands or usage instructions, simply invoke it without arguments from any connected terminal:

Type **user**



```
alice@vmlon0b: ~  
- Copyright (c) 2018-2025 KATLAS Technology.  
=====
```

IoE PoC menu
alice

```
=====
```

-- common commands:

- * restart Restart all daemons (all actors, as superuser).
- * bm print my bookmarks.
- * st shows state screen and exists.
- * watch keeps showing state screen.

-- PoC commands for this role (u):

- * send <file|-> <bm_index> Invoke to deliver items to a given user.
- * ship sends a random cert to Bob via Port1.

-- backend: ● alice backend is up

- * down shut it down

-- dev/bootstrap/inspect:

- * setup_bm setup bookmarks from file my_bookmarks.
- * logdir /var/log/katlas/katlas-wallet_278044
- * logtree prints backend logs file structure.
- * mlog merge log files into an ordered list of events.
- * plugins list installed wallet plugins (All R2R protocols).

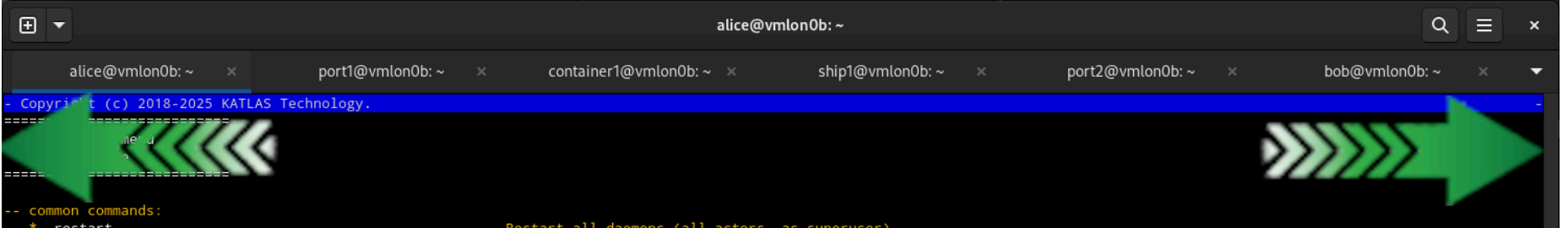
-- debug info:
wallet backend pid: 278044
logdir: /var/log/katlas/katlas-wallet_278044
alice@vmlon0b:~\$ _

The **user** utility provides both **common commands** shared across all actors—for general system interaction and status overviews—and **custom action commands** tailored to each actor’s **specific role** within the ecosystem.

This design enables a unified interface while supporting role-specific workflows in a streamlined and intuitive manner.

Adjusting Screen Size for Optimal Display

For the best experience, ensure your terminal window is wide enough to display content properly. Use the **blue reference line** as a guide: Adjust the window size so that the entire line fits on a **single row** without wrapping or splitting across lines.



Increase the window height to display the **maximum amount of information** without triggering scrollbars or causing flicker. Insufficient vertical space may result in scrolling and visual disruptions that degrade usability.

function: **user restart**: Restart the system

The **user restart** command is used to **reset the entire system**. This is particularly useful in demo environments where the system is configured **without data persistence** due to limited disk space. As a result, all processes start from a **clean initial state** after each restart—ideal for repeatable demonstrations.

To execute the restart, use any terminal connection (e.g., from the **alice** session) and enter: **user restart**

This command will **shut down all active daemons** (services) and then restart them.

 **Note:** This is a **super-user command** that affects **all actors**, not just the one from which it was invoked.

It can be run from **any actor's session**, not only **alice**.

After execution, the system displays a **list of running daemons**, confirming that all components have been successfully restarted.

```
Process running:
* blockchain node:
  268852 ?      S<sl  1:21 /usr/local/bin/katlas-gov -ds -e 12 -v 100 -c 0 -p 16672 -pp 16672 -status 0 -E 0
* wallets:
  278044 ?      S1    0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 16573 -pp 16573 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  278121 ?      S1    0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 16873 -pp 16873 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  278198 ?      S1    0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 17073 -pp 17073 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  278274 ?      S1    0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 16773 -pp 16773 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  278348 ?      S1    0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 16973 -pp 16973 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  278422 ?      S1    0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 17173 -pp 17173 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  278505 ?      S1    0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 17273 -pp 17273 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  278581 ?      S1    0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 17373 -pp 17373 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
alice@vmlon0b:~$
```

The presence of **eight wallet processes**—including **three distinct container actors**, not just one—confirms that the system has restarted successfully.

In addition, the list includes a **ninth process**, corresponding to the **blockchain node**, which operates as part of the public network layer.

This complete set of processes ensures that all actors and core infrastructure are active and ready for interaction following the restart.

 function: user bm: Bookmarks

 Bookmarks: Preconfigured Actor Discovery

Each actor starts with a **pre-configured set of bookmarks**, which serve as the mechanism through which wallets become aware of other participants in the ecosystem.

A **bookmark** is essentially a **QR-encoded reference** containing endpoint and personality (p9y) information for a known actor. It allows the wallet to recognize and initiate interactions with peers—even before any direct contact or trade has occurred.

Bookmarks are stored in the following file, relative to each actor’s home directory:

```
~/katlas/wallet/trader/bookmarks
```

 **user bm** is an alias for the command `wallet trade bookmarks`

user bm outputs:

 Example: Alice’s Bookmarks

When executing the **user bm** command from Alice’s terminal, the following output appears:

```
alice@vmlon0b:~$ user bm
1: 'bob': qr: "2zbZkCCvRrET6KtLVd9SgkquYkUo 3FRNp6rMLL4wVjAeCCtgCjZ2VFip "bob" u2u u " label: "" icon: [0 bytes]
2: 'port1': qr: "3kdzg2ymfo3nbCHrpkyfqY1hXHG 9mhtEQL86w9w5hfaDR5dAidZux4 "port1" u2p p " label: "p (u2p)" icon: [0 bytes]
```

Interpretation:

- **Alice knows Bob and Port1**, each identified by a bookmark that includes:
 - A **wallet address** (network endpoint).
 - A **public p9y key** (personality identity).
 - A **name** (e.g., "bob", "port1").
 - A **protocol identifier** (e.g., **u2u** for user-to-user, **u2p** for user-to-port).
 - A **role** played in that protocol (**u** for user, **p** for port).
- In Bob's case:
 - The **u2u** protocol is **symmetric**, meaning both parties are users.
 - Alice sees Bob as a peer she can securely contact via this protocol.
- Port1's bookmark includes:
 - The **u2p** protocol, which is **asymmetric** (user-to-port).
 - Alice plays the role **u** (user), and Port1 plays the role **p** (port).
- **Optional fields:**
 - **label**: A user-defined description (e.g., "**p (u2p)**" for Port1).
 - **icon**: A placeholder for graphical representation (currently **[0 bytes]**).

These bookmarks form the foundation for secure peer discovery and interaction in the wallet ecosystem.

The illustration highlights the **portion of the bookmark used to generate the QR code**—specifically the compact string containing the **wallet address, public p9y key, actor name, protocol, and role**.

This encoded string serves as the complete payload for the **graphical QR representation**, enabling fast and secure sharing of actor metadata for establishing trusted connections.



The following outputs display the **preconfigured bookmarks for the remaining actors**, each representing known peers along with their associated protocol and role metadata. These entries enable secure peer discovery and interaction across the network.

Bob:

```
bob@vmlon0b:~$ user bm
1: 'alice': qr: "zuyUMFMJapXyr8TaTWQzrhUNPfh 3TaDdWk9s1GP8MyoCtK2NsEceabk "alice" u2u u " label: "" icon: [0 bytes]
2: 'port2': qr: "33nCR3WseaXRxyJeGYquYhju1RQ4 2rf47mCQupUfr45B3nAJeL6RV919 "port2" u2p p " label: "p (u2p)" icon: [0 bytes]
```

Similarly Bob knows about his friend Alice and his nearest port port2

Port1 and Port2:

```
port1@vmlon0b:~$ user bm
1: 'container1': qr: "4VrH1C6JxGCtXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1" c2p c " label: "c (c2p)" icon: [0 bytes]
```

```
port2@vmlon0b:~$ user bm
1: 'container2': qr: "3awX71NqZrwZ1BhSiDr8oRqmfcDY 241SzCZfUAx6JkYDdu3vHxcaQLGr "container2" c2p c " label: "c (c2p)" icon: [0 bytes]
2: 'container3': qr: "4HoUpiQNoQBSX3T4GYEULcjMnmNr 3gR3FREtgVbWzDp4f48MMucC97m3 "container3" c2p c " label: "c (c2p)" icon: [0 bytes]
```

Ship1:

```
ship1@vmlon0b:~$ user bm
1: 'port1': qr: "3kdzg2ymfo3nbCHrpkyjqY1hXHG 9mhtEQL86w9w5hfaDR5dAidZux4 "port1" p2s p " label: "p (p2s)" icon: [0 bytes]
2: 'port2': qr: "33nCR3WseaXRxyJeGYquYhju1RQ4 2rf47mCQupUfr45B3nAJeL6RV919 "port2" p2s p " label: "p (p2s)" icon: [0 bytes]
```

Initially, **ports are only aware of their associated containers** through preconfigured bookmarks.

In this simplified model, **ports serve a dual function**:

- 1. **Logistical Role**: Managing **load and unload operations** between containers and ships (i.e., port–ship–container relationships).
- 2. **Organizational Role**: Acting as the **owning entity** or **company**, managing the relationship between **containers and end-users** (i.e., container–user associations).

This dual role allows ports to coordinate both physical and administrative aspects of the supply chain within the demo environment.

Finally, we observe that the **ship is only aware of the two ports**, each represented by a bookmark configured to initiate the **asymmetric Role-to-Role (R2R) protocol p2s** (port-to-ship).

This reflects the intended design where **ships interact exclusively with ports**, relying on ports to coordinate container operations and manage broader logistical workflows.

 function:user st: actor dashboard

The **user st** command displays a **dashboard-style summary** of the actor’s current state. It prints a structured frame on screen showing key information such as active trades, known peers, connection statuses, and role-specific metrics—providing a quick overview of the actor’s operational context.

function: **user watch**: actor dashboard (Live Mode)

The `user watch` command provides a **live, continuously updating view** of the actor's current state. It displays successive frames on screen as new information becomes available.

Unlike the `user st` command—which performs repeated full cycles of connecting to the backend, issuing a query, and exiting (a resource-intensive polling method)—`user watch` leverages an **event-driven push model**.

Upon invocation, the CLI sends a **one-time instruction** to the backend to stream updates as they occur. The CLI then remains active, **listening and displaying real-time changes**, resulting in significantly more efficient resource usage.

Press **CTRL-C** to exit watch mode.



Per-Role Screen Breakdown

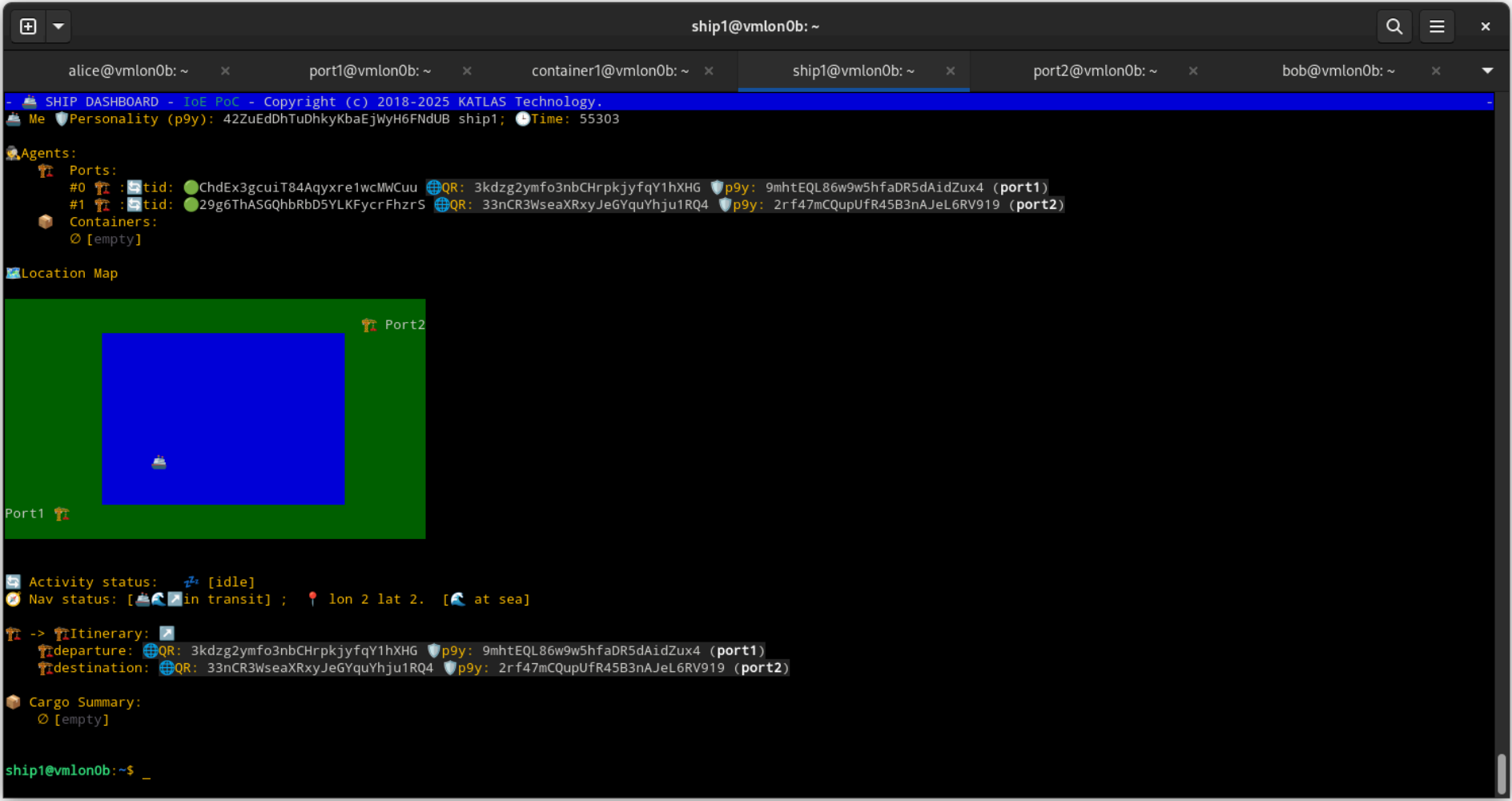
We now proceed to describe the **individual elements that compose the dashboard view** (`user st` / `user watch`) for each actor, categorized by **role**. Each role-specific screen presents tailored information relevant to its function within the ecosystem—whether user, port, container, or ship—providing insight into ongoing trades, known peers, and operational state.



Role: Ship. Actors: ship1

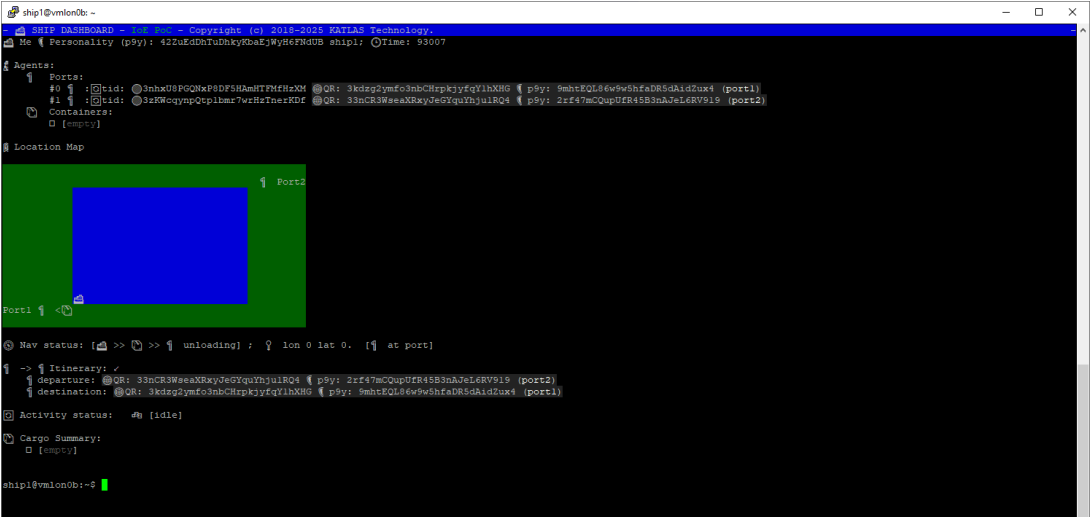
To view the current state of the ship actor, **switch to the terminal window or tab running `ship1`**, and enter the following command: `user st`

This will display the ship’s **dashboard**, presenting a summary snapshot of its operational context—including known ports, active trades, connection statuses, and any role-specific metrics related to its interaction in the supply chain.



Note for Windows / PuTTY Users

Users accessing the system via **PuTTY on Windows** may experience a **slightly degraded look-and-feel**, primarily due to **limited Unicode support** in the default font settings.



To improve rendering—especially for icons and layout symbols—consider installing a **font with full Unicode coverage** (e.g., *DejaVu Sans Mono*, *Fira Code*, or *Noto Mono*) and configuring PuTTY to use it.

This adjustment can significantly enhance the visual quality and alignment of the terminal interface.

Ship Dashboard quick overview

Command: `user st` (run from the `ship1` terminal)

This dashboard provides a live snapshot of the ship actor's status, connections, and planned itinerary. Below is a description of each section:

Me – Personality

Displays the ship's **unique personality (p9y)** and its human-readable alias (`ship1`).

-  **Personality (p9y):** `42ZuEdDhTuDhkyKbaEjWyH6FNdUB`
-  **Time:** `47035` – a system time marker used for simulation consistency.

Agents – Connections with Other Actors

This section lists all known and active trades with other peers, grouped by role:

-  **Ports:**
 - **#0:** Connected to `port1`, trade is active ( online), trade ID: `2YKN...AgSc`.
 - **#1:** Connected to `port2`, trade is active ( online), trade ID: `3wNG...a1i`.
 - Each entry includes:
 - **tid:** Trade ID shared between peers.
 -  **QR:** Remote actor's endpoint address.
 -  **p9y:** The personality (public key) of the connected port.
-  **Containers:**
 - Currently **empty** () – no containers are onboard or associated.

Location Map

A **text-based geographic visualization** of the ship's current position and surroundings:

- Ship  is shown **at Port2**.
- Port1 and Port2 are rendered spatially to suggest position and movement across the map.

Navigation Status

Summarizes the ship's current **operational and navigational context**:

-  <<  <<  **loading:** Ship is at port, currently in loading phase.
-  **lon 9 lat 9:** Simulated geographic coordinates.
-  **at port]:** Docked status confirmed.

Itinerary

Details the planned travel route:

- **departure:** From Port2 (including QR and p9y).
- **destination:** To Port1 (including QR and p9y).
-  indicates the direction of travel on the simulated map.

Activity Status

-  **idle]:** Ship is not currently executing any active protocol or loading/unloading action.

Cargo Summary

-  **[empty]:** No containers onboard at the moment.

🚢 Ship Dashboard Detailed Overview

🛡️ Header

```
- 🚢 SHIP DASHBOARD - IoE PoC - Copyright (c) 2018-2025 KATLAS Technology.  
🚢 Me 🛡️ Personality (p9y): 42ZuEdDhTuDhkyKbaEjWyH6FNdUB ship1; 🕒 Time: 55303
```

Blue Header Bar

The **blue header** at the top of the screen serves multiple purposes:

- **Identifies the current dashboard** — in this case, the **Ship Dashboard**.
- **Provides a visual guide for adjusting screen width**: ensure the header fits on a single line without wrapping for optimal display.
- **Credits ownership** — it includes the **copyright notice** for the UI.

🕒 Time (Emulation Clock)

The **Time** field reflects the **current simulation time (tick)**. It advances at a rate of **one unit per second** by default. Although this rate is configurable at the system level, it is **not user-accessible through the interface**—changing it requires source-level modification and recompilation.

🛡️ Me – Personality (p9y)

The **p9y** field displays the personality of the currently active actor—in this case, **ship1**.

- **Example Output**: **42ZuEdDhTuDhkyKbaEjWyH6FNdUB ship1**
 - The first string (**42Zu...**) is a **Base58-encoded public key**.
 - The second part (**ship1**) is a **moniker**, a human-readable label assigned to the key by the actor.

This identity represents "me" within the current wallet context, as the CLI is running under **ship1**.

📁 Personality Storage

Each actor's default personality is stored in: `~/.katlas/wallet/trader/personality`

While this value is used by default in most interactions, it can be **overridden dynamically** by:

- The **trading agent**
- The **R2R protocol in use**
- Other **programmable or contextual conditions**

This design allows for **identity flexibility** in multi-role or multi-context scenarios.

👤 Agents View

```
👤 Agents:  
🚢 Ports:  
#0 🚢 : 🛡️ tid: ● ChdEx3gcuiT84Aqyxre1wcMwCuu 🌐 QR: 3kdzg2ymfo3nbCHrpkjyfqY1hXHG 🛡️ p9y: 9mhtEQL86w9w5hfaDR5dAidZux4 (port1)  
#1 🚢 : 🛡️ tid: ● 29g6ThASGQhbRbD5YLFKfycrFhzrS 🌐 QR: 33nCR3WseaXRxyJeGYquYhju1RQ4 🛡️ p9y: 2rf47mCQupUfR45B3nAJeL6RV919 (port2)  
📦 Containers:  
📦 [empty]
```

The **Agents section** of the dashboard presents a categorized list of ongoing or potential peer connections. For the **ship1** actor, only two categories are relevant:

- 🚢 **Ports** – with which the ship negotiates loading/unloading operations.
- 📦 **Containers** – which may be onboarded or offloaded (empty in this case, indicating no active container trades).

Each agent entry includes the **shared trade ID (tid)**, QR-derived endpoint information, and the peer's personality (**p9y**). The connection status (e.g., ● online) gives an at-a-glance view of whether secure negotiation is currently possible.

Shared Trade Context via **tid**

Each **Role-to-Role (R2R) trade** is uniquely identified by a **tid**, which is **shared between both parties**. This allows for **symmetric access** to trade state and logs, enabling full traceability from either side.

For example, the ship **ship1** and port **port1** share the trade ID: ChdEx3gcuiT84Aqyxre1wcMWCuu

This trade can be inspected on **both sides** by executing the same command:

On **ship1**: `wallet trade ChdEx3gcuiT84Aqyxre1wcMWCuu show data`

```
ship1@vmlon0b:~$ wallet trade ChdEx3gcuiT84Aqyxre1wcMWCuu show data
Data:
'activity': 1749393641250318901
'bootstrapped_by': 4M4ciabTzKREsvXdDXfpFnzfY8Kz
'chat_script': 0
'created': 1749325044898835784
'data_generated': 1749393641250463211
'datadir': /home/ship1/.katlas/wallet/trader/trade_data/ChdEx3gcuiT84Aqyxre1wcMWCuu
'datasubdir': ChdEx3gcuiT84Aqyxre1wcMWCuu
'homemode': me/peer
'ico': N
'initiator': Y
'lang': en
'local__doctypes_consumer':
'local__doctypes_producer':
'local_endpoint': 4M4ciabTzKREsvXdDXfpFnzfY8Kz anonymous
'logfile': /home/ship1/.katlas/wallet/trader/log/trade_1749325044898835784_ChdEx3gcuiT84Aqyxre1wcMWCuu
'logo': N
'my_moniker': ship1
'my_personality': 42ZuEdDhTuDhkyKbaEjWyH6FNdUB
'online_age': 19:03:16
'parent_tid': ChdEx3gcuiT84Aqyxre1wcMWCuu
'peer_moniker': port1
'peer_personality': 9mhtEQL86w9w5hfaDR5dAidZux4
'peer_protocols': 0
'pphome': /home/ship1/.katlas/wallet/trader/p2s/s/42ZuEdDhTuDhkyKbaEjWyH6FNdUB/9mhtEQL86w9w5hfaDR5dAidZux4
'protocol': p2s s
'qrs': N
'remote__doctypes_consumer':
'remote__doctypes_producer':
'remote_endpoint': 3kdzg2ymfo3nbCHrpkyfqY1hXHG 9mhtEQL86w9w5hfaDR5dAidZux4 "port1"
'remote_ip4': 34.39.80.254:16773
'roles': N
'state': online
'trade_state': 1
```

On **port1**: `wallet trade ChdEx3gcuiT84Aqyxre1wcMWCuu show data`

```
port1@vmlon0b:~$ wallet trade ChdEx3gcuiT84Aqyxre1wcMWCuu show data
Data:
'activity': 1749393941302111855
'bootstrapped_by': 4M4ciabTzKREsvXdDXfpFnzfY8Kz
'chat_script': 0
'created': 1749325044950217104
'data_generated': 1749393941302161505
'datadir': /home/port1/.katlas/wallet/trader/trade_data/ChdEx3gcuiT84Aqyxre1wcMWCuu
'datasubdir': ChdEx3gcuiT84Aqyxre1wcMWCuu
'homemode': me/peer
'ico': N
'initiator': N
'lang': en
'local__doctypes_consumer':
'local__doctypes_producer':
'local_endpoint': 3kdzg2ymfo3nbCHrpkyfqY1hXHG anonymous
'logfile': /home/port1/.katlas/wallet/trader/log/trade_1749325044950217104_ChdEx3gcuiT84Aqyxre1wcMWCuu
'logo': N
'my_moniker': port1
'my_personality': 9mhtEQL86w9w5hfaDR5dAidZux4
'online_age': 19:08:16
'parent_tid': ChdEx3gcuiT84Aqyxre1wcMWCuu
'peer_moniker': ship1
'peer_personality': 42ZuEdDhTuDhkyKbaEjWyH6FNdUB
'peer_protocols': 0
'pphome': /home/port1/.katlas/wallet/trader/p2s/p/9mhtEQL86w9w5hfaDR5dAidZux4/42ZuEdDhTuDhkyKbaEjWyH6FNdUB
'protocol': p2s p
'qrs': N
'remote__doctypes_consumer':
'remote__doctypes_producer':
'remote_endpoint': 4M4ciabTzKREsvXdDXfpFnzfY8Kz 42ZuEdDhTuDhkyKbaEjWyH6FNdUB "ship1"
'remote_ip4': 0.0.0.0:0
'roles': N
'state': online
'trade_state': 1
```

Interpreting the Trade Data

The output of the `show data` command reveals detailed information about the trade session:

- **Protocol:**
 - On `ship1: p2s s` (ship role in port-to-ship protocol)
 - On `port1: p2s p` (port role in same protocol)
- **State:**
 - `online` — mutual cryptographic authentication has succeeded; agents are ready for protocol execution.
- **Endpoints:**
 - `local_endpoint`: Address of the wallet issuing the command.
 - `remote_endpoint`: Address of the peer wallet.
- **Personalities:**
 - `my_personality` / `peer_personality`: Each actor's `p9y` (public key identity).
 - `my_moniker` / `peer_moniker`: Human-readable names.
- **Timestamps:**
 - `created`: Trade session creation time.
 - `activity` / `data_generated`: Latest activity timestamps.
- **Connection Metadata:**
 - `online_age`: Duration the trade has been in the online state.
 - `logfile`: Path to the session log.
 - `datadir`: Location of trade-related storage.
- **Role-based home paths (`pptime`):**
 - Structured directory paths segregated by role pairing (`p2s/s/...` on ship, `p2s/p/...` on port).

This shared access to trade metadata empowers both parties with **verifiable, decentralized state synchronization**, while maintaining full control over their respective records and logs.

The table below provides a side-by-side comparison.

Field	ship1	port1
actor	ship1	port1
protocol	p2s s	p2s p
state	online	online
initiator	Y	N
my_moniker	ship1	port1
my_personality	42ZuEdDhTuDhkyKbaEjWyH6FNdUB	9mhtEQL86w9w5hfaDR5dAidZux4
peer_moniker	port1	ship1
peer_personality	9mhtEQL86w9w5hfaDR5dAidZux4	42ZuEdDhTuDhkyKbaEjWyH6FNdUB
local_endpoint	4M4ciabTzKREsvXdDXfpFnzfY8Kz anonymous	3kdzg2ymfo3nbCHrpkyfqY1hXHG anonymous
remote_endpoint	3kdzg2ymfo3nbCHrpkyfqY1hXHG "port1"	4M4ciabTzKREsvXdDXfpFnzfY8Kz "ship1"
remote_ip4	34.39.80.254:16773	0.0.0.0:0
pptime	/home/ship1/.katlas/.../p2s/s/42Zu.../9mht...	/home/port1/.katlas/.../p2s/p/9mht.../42Zu...
logfile	/home/ship1/.katlas/.../log/trade_..._ChdEx...	/home/port1/.katlas/.../log/trade_..._ChdEx...
datadir	/home/ship1/.katlas/.../trade_data/ChdEx...	/home/port1/.katlas/.../trade_data/ChdEx...
online_age	19:03:16	19:08:16

Note: The reason the `remote_ip4` field appears as `0.0.0.0:0` on the **non-initiator** side (also known as the *follower*) is because only the **initiator** is responsible for resolving the IP address of the remote party. The follower simply accepts incoming connections and does not perform active resolution.

🚢 Location map



The **Location Map** provides a visual and textual representation of the ship's current **geographic position**, its **planned route**, and its **operational status**.

- Displays the ship's **coordinates** (latitude and longitude).
- Highlights whether the ship is **in transit** or **moored at a port**.
- When docked, it shows whether the ship is currently **loading** or **unloading containers**.
- The route plan includes the **departure** and **destination ports**, offering context on the ship's role within the current logistical flow.

This map helps situational awareness and enables operators or observers to track the ship's movement and state at a glance.

🚢 Activity status



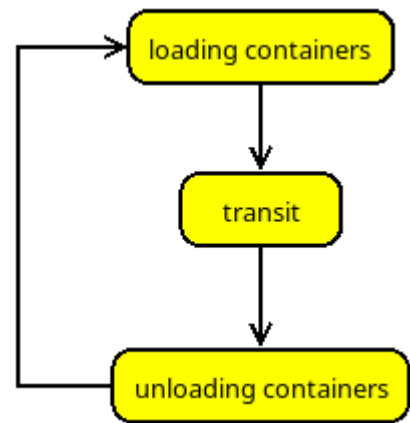
The **Activity Status** indicates what the **ship's manager (captain)** is currently doing within the system. It reflects real-time operational behavior beyond simple movement.

Possible States:

Icon	Status	Description
🛑	idle	The ship is not performing any special operation beyond sailing or waiting.
📞🚚	contacting_departure_port	Initial handshake with the departure port . Occurs only once at system startup to bootstrap trades.
📞🚚	contacting_destination_port	Initial handshake with the destination port . Also part of trade bootstrapping.
❌	error	An error occurred—manual or automated intervention may be required.

🔄 State Descriptions

This diagram represents a **cyclical state machine** modeling the operational lifecycle of a **cargo ship** or similar logistics entity handling containers. Here's a breakdown of the states and transitions:



1. **loading containers**
 - The initial and recurring state.
 - The ship is actively onboarding containers.
 - This occurs at the **origin port** before departure.

2. transit
 - Indicates the entity is in motion—i.e., the ship is sailing between ports.
 - No load/unload actions occur during this phase.
3. unloading containers
 - The system is offloading containers at the destination port.
 - Marks the completion of a delivery segment.
4. Return Transition to loading containers
 - After unloading, the system loops back to begin the cycle again, ready to take on new cargo.

 Cargo

The **Cargo** section is the final element in the ship's dashboard. It lists all **containers currently onboard** the vessel.

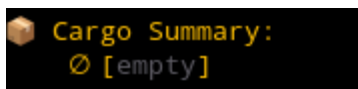
Each entry typically includes:

- The **container identifier**
- Its associated metadata (e.g., sender, recipient, contents) if available

If no containers are currently loaded, this section will display an **empty symbol** (∅), indicating the ship is either awaiting cargo or has completed unloading.

This view provides a quick overview of the ship's load status and is essential for tracking logistics across the supply chain.

With no containers on board the view is:



or, with e.g. one container cargo, the view would look like:



 watch

You can now type **user watch** to observe the **live activity of the ship** as it begins its **patrolling routine**. Although it is not yet carrying any cargo, containers will begin to be dispatched shortly, and their appearance will be reflected in real time.

To exit watch mode at any time, press **CTRL-C**.

 Screencast

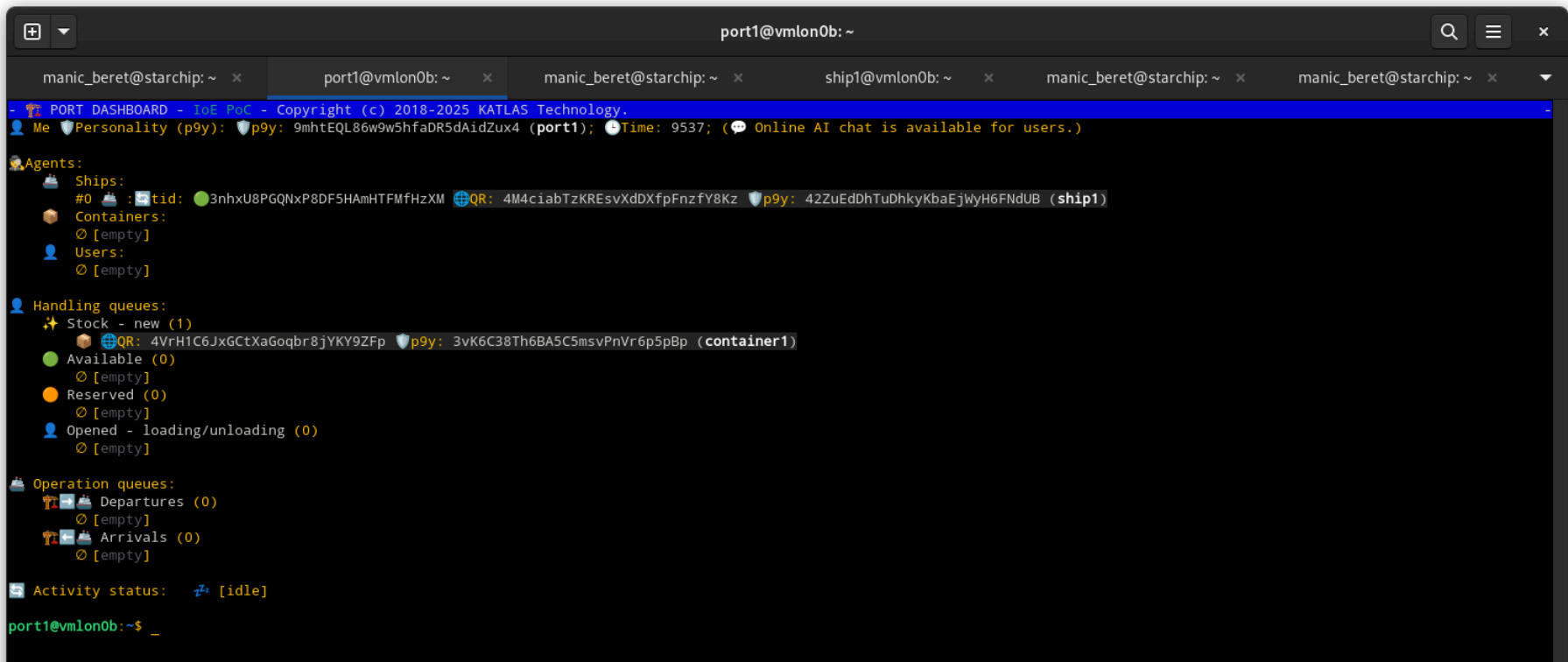
You can watch a **screencast recording** showing the ship eventually **picking up a container from Port2** and **delivering it to Port1**.

		Ship dashboard KATLAS-PoC-SupplyChain-DEMO__Ship1 https://www.youtube.com/watch?v=MtBgG4zU0z8&list=PLxosfK0Tg_rWlddsiz4Nad40EJmMgIM5G&index=4
---	---	--

🏗️ Role: Port. Actors: port1, port2

To begin exploring the functionality of a port actor, **open the terminal window for either port1 or port2** and enter the following command:

```
user st
```



This displays the **Port Dashboard**, offering a structured view of the port's current operational state. Here's a breakdown of the key sections:

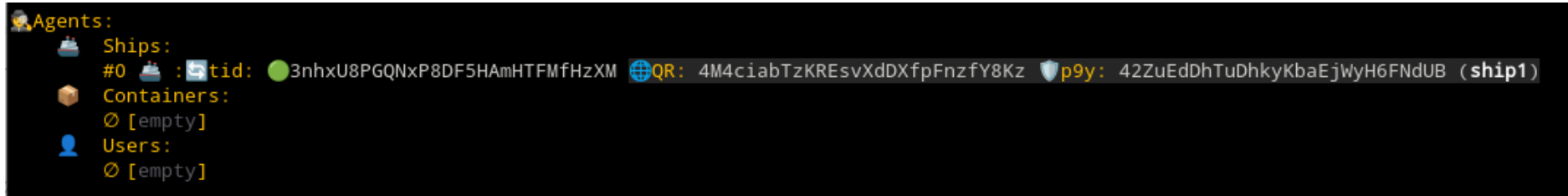
🛡️ Header

The **header** and **Agents** sections are similar to those found in the Ship Dashboard, with one notable addition: Online AI chat is available for users.

🤖 All agents are equipped with an **online chat interface** connected to automation hooks. For **port roles**, this feature includes **ChatGPT integration**, allowing the port to respond intelligently to user queries.

We'll explore this interactive capability later in the demonstration.

👤 Agents



This section reflects all **active and known interactions** between the port and other actors—specifically **ships**, **containers**, and **users**. Each interaction is managed by a dedicated agent executing a **Role-to-Role (R2R) protocol**, with the current status displayed here. All interactions with ships, containers and users, which are managed by the respective agent running the corresponding R2R protocol, are reflected in this section.

⚙️ Port Role Responsibilities

Ports are programmed to carry out two distinct categories of activity:

1. 📦 **Container Handling**
2. 🚢 **Ship Operations**

Ports have been coded to perform two separate activities: **Container Handling** and **Ship operations**. The section in the dashboard representing que Handling pipeline is named Handling queues. It has 4 sections where containers are allocated according to their handling state.

🏗️ Handling Pipeline

Container queues initial status (Port1):

```
👤 Handling queues:
  ✨ Stock - new (1)
    🏠 🌐QR: 4VrH1C6JxGCtXaGoqbr8jYKY9ZFp 🛡️p9y: 3vK6C38Th6BA5C5msvPnVr6p5pBp (container1)
  ● Available (0)
    ⚪ [empty]
  ● Reserved (0)
    ⚪ [empty]
  👤 Opened - loading/unloading (0)
    ⚪ [empty]
```

Container queues initial status (Port2):

```
👤 Handling queues:
  ✨ Stock - new (2)
    🏠 🌐QR: 3awX71NqZrwZ1BhSiDr8oRqmfcDY 🛡️p9y: 241SzCZfUAx6JkYDdu3vHxcaQLGr (container2)
    🏠 🌐QR: 4HoUpiQNoQBSX3T4GYEULcjMnmNr 🛡️p9y: 3gR3FREtgVbwzDp4f48MMucC97m3 (container3)
  ● Available (0)
    ⚪ [empty]
  ● Reserved (0)
    ⚪ [empty]
  👤 Opened - loading/unloading (0)
    ⚪ [empty]
```

The **handling pipeline** is composed of **four container queues**, each representing a stage in the **container management lifecycle** within the port. These queues support the port’s responsibility of tracking, scheduling, and coordinating container activity.

Queue	Purpose
✨ Stock – new	New unregistered containers awaiting until demand needs them
● Available	Containers ready for dispatch or assignment
● Reserved	Containers reserved for specific users
👤 Opened	Containers currently in the process of loading or unloading

🧱 1. Stock Queue – Initial Registration

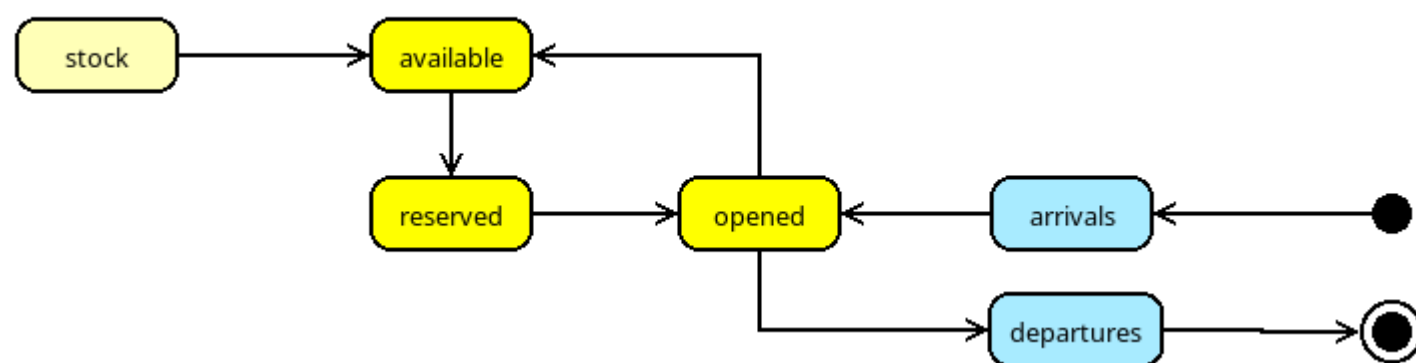
- This queue is **preloaded at startup** with the list of containers the port is aware of.
- The source of this information is the port’s **bookmarks file**, which includes references to known containers.

Example configuration:

- **port1** starts with **one container**: **container1**
- **port2** starts with **two containers**: **container2, container3**

These containers are visible in the ✨ **Stock – new** section until further handling actions are taken (e.g., reservation, assignment, loading).

Handling & Operations Pipeline State Machine



This state machine illustrates the lifecycle of **containers managed by a port**. It distinguishes two main pipeline stages:

- **Handling pipeline** (in **yellow**) – covers all stages **prior to sealing** a container.
- **Operations pipeline** (in **blue**) – involves containers that are **sealed** and **scheduled for movement** (arrival or departure).

Handling Pipeline (Yellow States)

This section manages **non-sealed containers**, handling their preparation and readiness for operations.

♦ stock

- **Initial state** for all containers defined in a port's bookmarks.
- Containers in this state have never been contacted by the port.
- Considered **orphan**: no owner, no assigned user, not ready for operation.
- Accessible from the wallet but **inactive**.

Transition:

When a user request arrives and no container is available, the port **bootstraps** a container from stock by contacting it. This **first contact establishes ownership**, moving it to **available**.

♦ available

- Containers in this state are **ready for assignment**.
- Awaiting user demand.
- Represents containers the port has **successfully contacted** and can operate.

Transition:

- On user reservation: moves to **reserved**.
- If loading/unloading is cancelled: may return here from **opened**.

♦ reserved

- Containers have been **allocated** for a specific operation (i.e., user request).
- Not yet engaged in active transfer.

Transition:

- When loading/unloading begins (i.e. user accepted the container): moves to **opened**.

♦ opened

- Container is **actively being handled**: either loading or unloading.
- Considered under active physical or protocol-level manipulation.

Transition:

- On completion and sealing: moves to **departures** or **arrivals** (operations pipeline).
- If operation is aborted: may return to **available**.

📦 Operations Pipeline (Blue States)

These states are used for **sealed containers**, i.e., those that are finalized for shipping.

♦ **departures**

- Container is **sealed** and **awaiting departure** on a ship.
- This is a **terminal preparation state** before actual transit.

♦ **arrivals**

- Container has **arrived** and is queued for **post-arrival handling** (unsealing process).

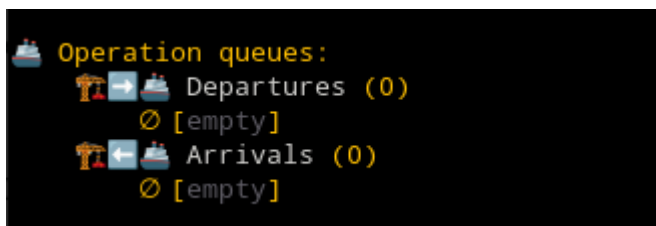
Transition:

- On unsealing: moves to **opened** (operations pipeline).

⚙️ Container Readiness & Bootstrapping

- If **available** is **empty** when a user request arrives, a new container is **bootstrapped from stock**.
- Bootstrapping is performed by initiating contact; the first port to do so becomes its **registered owner** and current **manager**.
- This process **activates the container**, promoting it to **available**.

🏗️ Operations Pipeline



The **Operations Pipeline** manages **sealed containers**—those that are finalized and ready for physical movement. It consists of two queues:

🚢 Departures

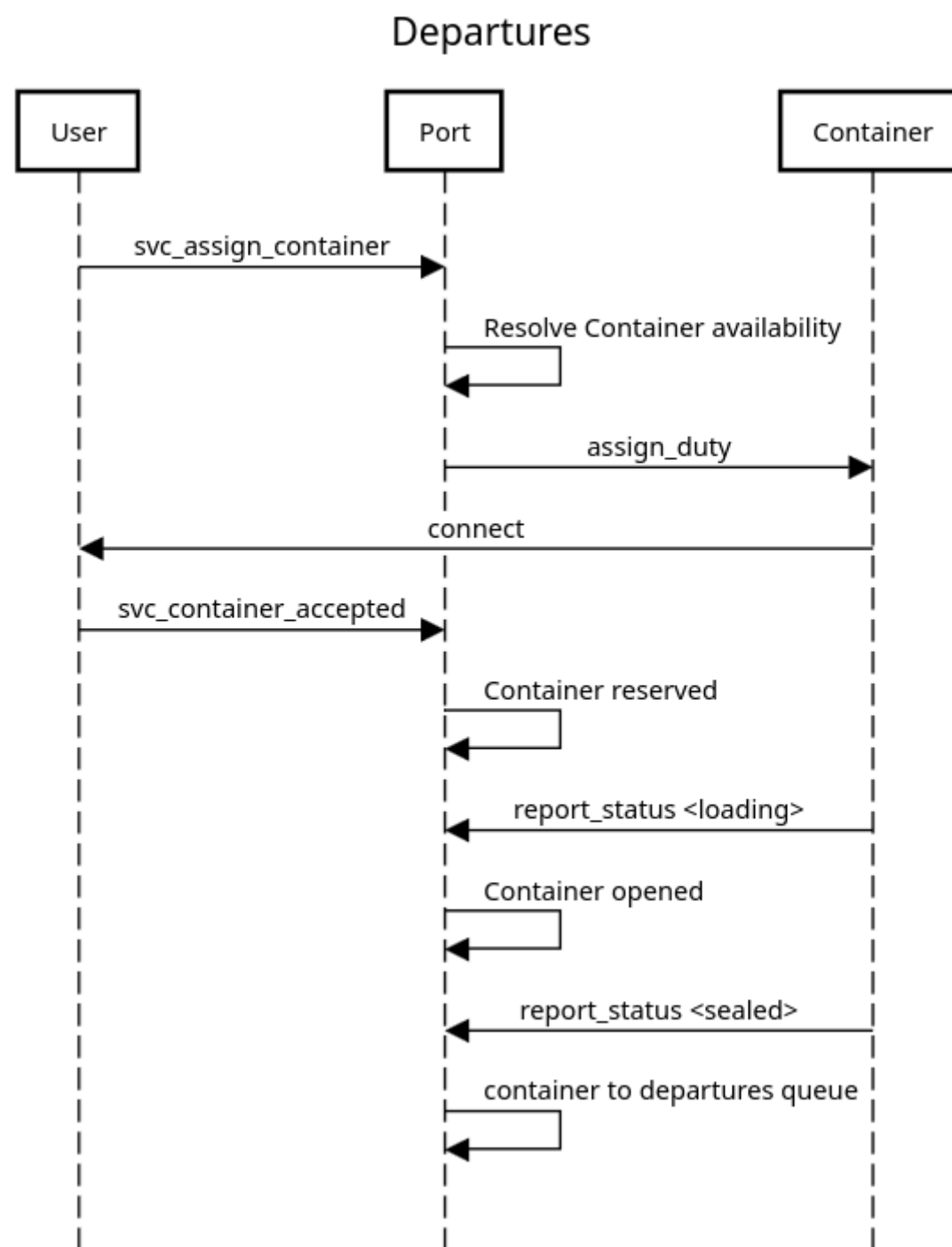
- Fed directly from the **Handling Pipeline** once a container has been **sealed** during the loading process.
- Represents containers that are **ready to leave the port**, awaiting pickup by a scheduled ship.
- These containers have completed all preparatory steps and are in a finalized state.

🚢 Arrivals

- Fed by **incoming ships** delivering containers to the port.
- Contains sealed containers that have been offloaded and are now awaiting post-arrival processing (e.g., unpacking, reassignment, or inspection), in our basic model, just waiting to be unsealed.
- These containers are ready to be transitioned back into the handling pipeline once unsealed.

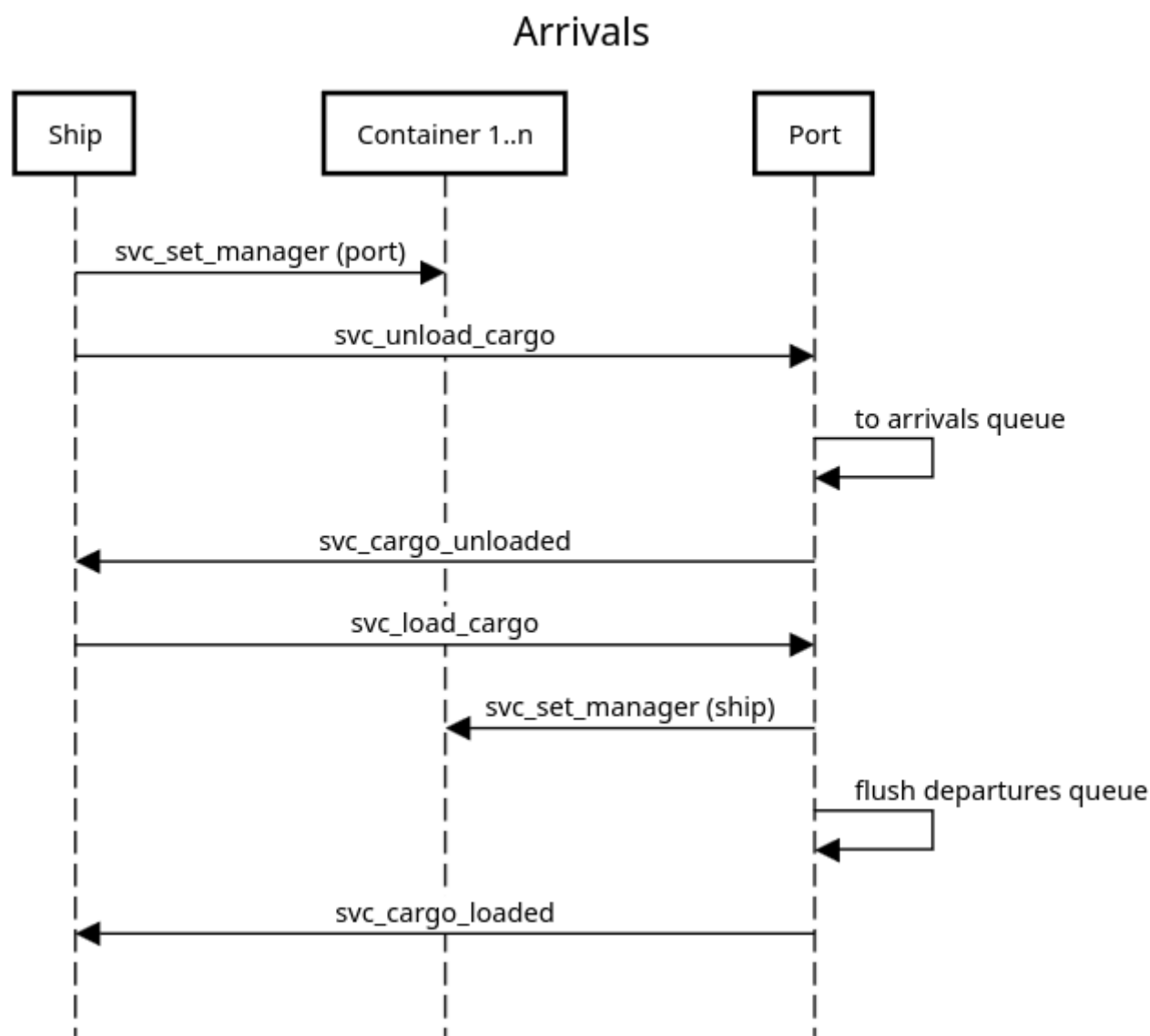
Interactions

This section presents a set of **sequence diagrams** that illustrate the **message flows** and corresponding **actions triggered by agents** during container and trade operations.



This **sequence diagram** illustrates the process of assigning a container for **departure operations**, initiated by a user and coordinated by the port.

1. **User → Port:** `svc_assign_container`
The user requests the port to assign a container for outbound use.
2. **Port:**
 - Resolves container availability (checks the `available` queue, bootstraps from stock if needed).
 - Assigns the task to a specific container using `assign_duty`.
3. **Container → User:** `connect`
The Container establishes a secure connection with the User.
4. **Port → User:** `svc_container_accepted`
The user is notified that the container has been successfully assigned.
5. **Port:**
 - Marks the container as **reserved**.
 - Waits for status updates from the container:
 - `report_status <loading>` → loading has begun
 - `report_status <sealed>` → container is sealed and ready
6. **Port:**
 - Once sealed, the container is moved into the **departures queue**, completing the pre-departure process.



This **sequence diagram** models the **Arrivals process**, showing how containers are offloaded and new ones are loaded between a ship and a port.

1. Ship Sets Port as Manager

- **svc_set_manager (port)**: The ship tells each onboard container that the port is now their new manager, preparing them for disembarkation.

2. Unloading Cargo

- **svc_unload_cargo**: The container initiates the unloading process to the port.
- The port receives the cargo and **places the container into the arrivals queue**.
- **svc_cargo_unloaded**: The container confirms that the unloading process has completed.

3. Loading Cargo

- **svc_load_cargo**: A new container is loaded from the port onto the ship.
- **svc_set_manager (ship)**: The port sets the ship as containers new manager for the journey.
- The port then **flushes the departures queue**, finalizing the handoff of the container.
- **svc_cargo_loaded**: The port confirms the cargo has been successfully loaded onto the ship.

watch

You can type at any time **user watch** to observe the **live activity of the port**. To exit watch mode, press **CTRL-C**.

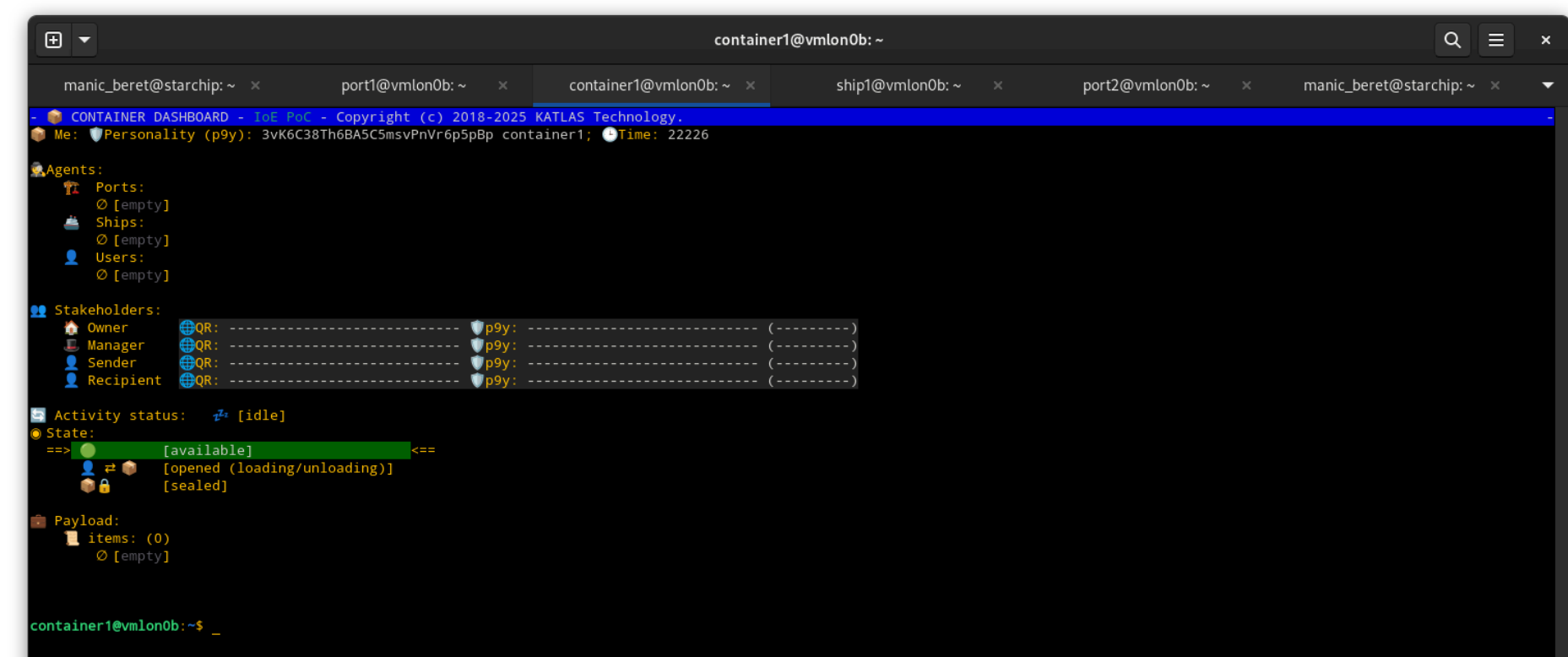
Screencast

You can watch a **screencast recording** showing the port1 activity during a single delivery from Alice to Bob.

		port dashboard KATLAS-PoC-SupplyChain-DEMO_Port1 https://www.youtube.com/watch?v=NwZUI1UQI8g&list=PLxosfK0Tg_rWlddsiz4Nad40EJmMgIM5G&index=2
--	--	---

 Role: Container. Actors: container1, container2, container3

To begin exploring the behavior of a container actor, **log in to the terminal session** of `container1`, `container2`, or `container3`, and run:
`user st`



This dashboard provides detailed information about the container's identity, current handlers, operational state, and cargo.

 Header

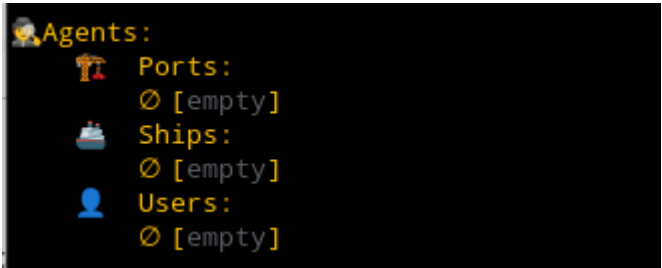


Shows the container's **personality (p9y)** and its human-readable **moniker**.
Example: `3vK6C38Th6BA5C5msvPnVr6p5pBp container1`

 **Time: 12813** indicates the simulation time tick.

 Agents

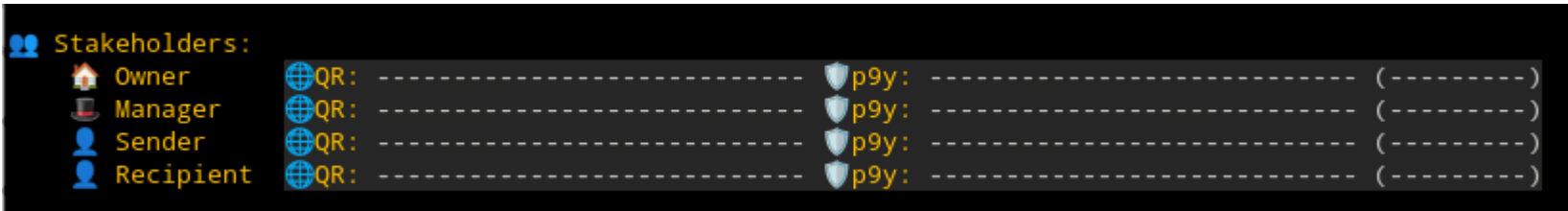
The first two sections, header and agents, have analogous descriptions.



We can observe that **container1 has not yet established contact with any other actor**—neither ports, ships, nor users appear in its agents list. This is expected and consistent with its current state: it resides in the **stock queue** of `port1`, meaning it has not yet been bootstrapped or assigned.

At this stage, the container is in a “**just manufactured**” **condition**—recognized by the system, cryptographically initialized, and ready for activation, but still **unclaimed and inactive** within the logistics workflow.

 Section stakeholders



Stakeholders Section

This section lists **external actors** that play key roles in the container's lifecycle. These include:

-  **Owner** – The entity that claims long-term custody.
-  **Manager** – The entity currently authorized to issue commands.
-  **Sender** – The customer initiating a shipment.
-  **Recipient** – The final destination party, revealed at sealing time.

Initial (Orphan) State

In the current **orphan state**, the container has not been contacted by any actor and thus shows no stakeholders. This is consistent with its presence in the **stock queue** at **port1**.

Upon **first contact by a port**, the container will:

- Permanently assign that port as its **owner**—similar to imprinting, like a newborn recognizing its guardian.
- Temporarily assign the same entity as its **manager**.

 In this prototype model, the **owner never changes** once set, but the **manager may change** over time—particularly when the container travels between ports and ships.

Manager Authority

The container will **only respond to instructions from its current manager**, and will **ignore commands from any other actor**. This enforces a strict trust model and helps prevent unauthorized manipulation.

The most critical instruction a manager can issue is to **assign a duty** to the container—triggering the start of a transaction with a new customer. When this occurs:

- The **sender** is registered as part of the container's stakeholder list.
- The **recipient** is later revealed **by the sender** at the moment the container is **sealed**.

Section Activity

 Activity status:  [idle]

In this implementation, containers do not engage in significant decision-making. All state transitions are event-driven, so the activity status remains consistently **idle** through.

Section State

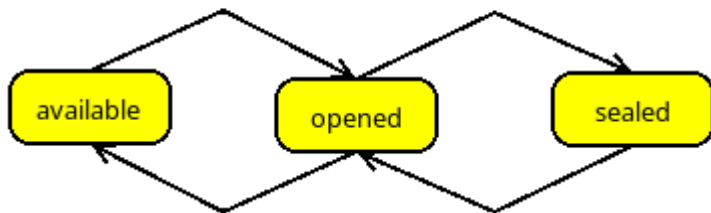
 State:
==>  [available] <==
   [opened (loading/unloading)]
  [sealed]

This section displays the container's current **operational state**, selected from a predefined set of **three lifecycle stages**:

- available
- opened
- sealed

The container's **current state** is visually indicated by a **green highlight bar**.

State Machine Overview



The diagram illustrates valid **transitions** between states, forming a simple yet expressive lifecycle:

Available

- The container is **off-duty**, in a **waiting state**.
- It is not yet assigned to a sender or recipient.
- Can be selected by a port to fulfill a new duty.

Opened

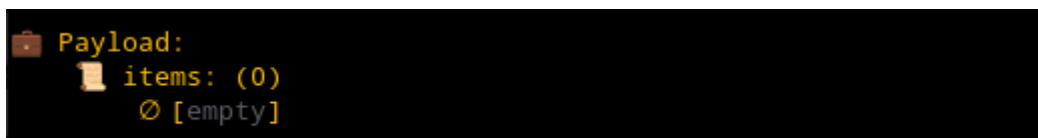
- The container has been **assigned a duty by its manager**.
- Two possible scenarios depending on direction:
 - If **sending**: the sender is currently **filling** the container with goods.
 - If **receiving**: the recipient has been granted access to **unseal and unload**.
- This state ends when the sender seals it, or the recipient finishes unloading.

Sealed

- The container is **sealed and protecting its contents**.
- It can **only be unsealed** by the **registered recipient**, who was assigned by the sender at the moment of sealing.
- Any other actor attempting to unseal will be ignored.

This event-driven state machine emphasizes security, clarity of authority, and data integrity, while keeping the container's internal logic simple and deterministic.

Section Payload



The **Payload** section displays the list of **items currently loaded into the container**. These items are added by the **sender** while the container is in the **opened** state.

Each item is represented by a **digital certificate**, conforming to the **Verifiable Credentials** standard. These certificates are:

- **Signed by the sender**, serving as a declaration of the contents.
- Intended to describe the **physical goods** that have been loaded.
- **Tamper-proof and cryptographically verifiable**.

In future, more advanced implementations—such as those including a **regulator role**—these certificates may be issued or co-signed by **recognized authorities**, ensuring legal or regulatory compliance.

This design ensures that the payload is both **digitally accountable** and **secure**, aligning with decentralized trust models and future-ready supply chain standards.

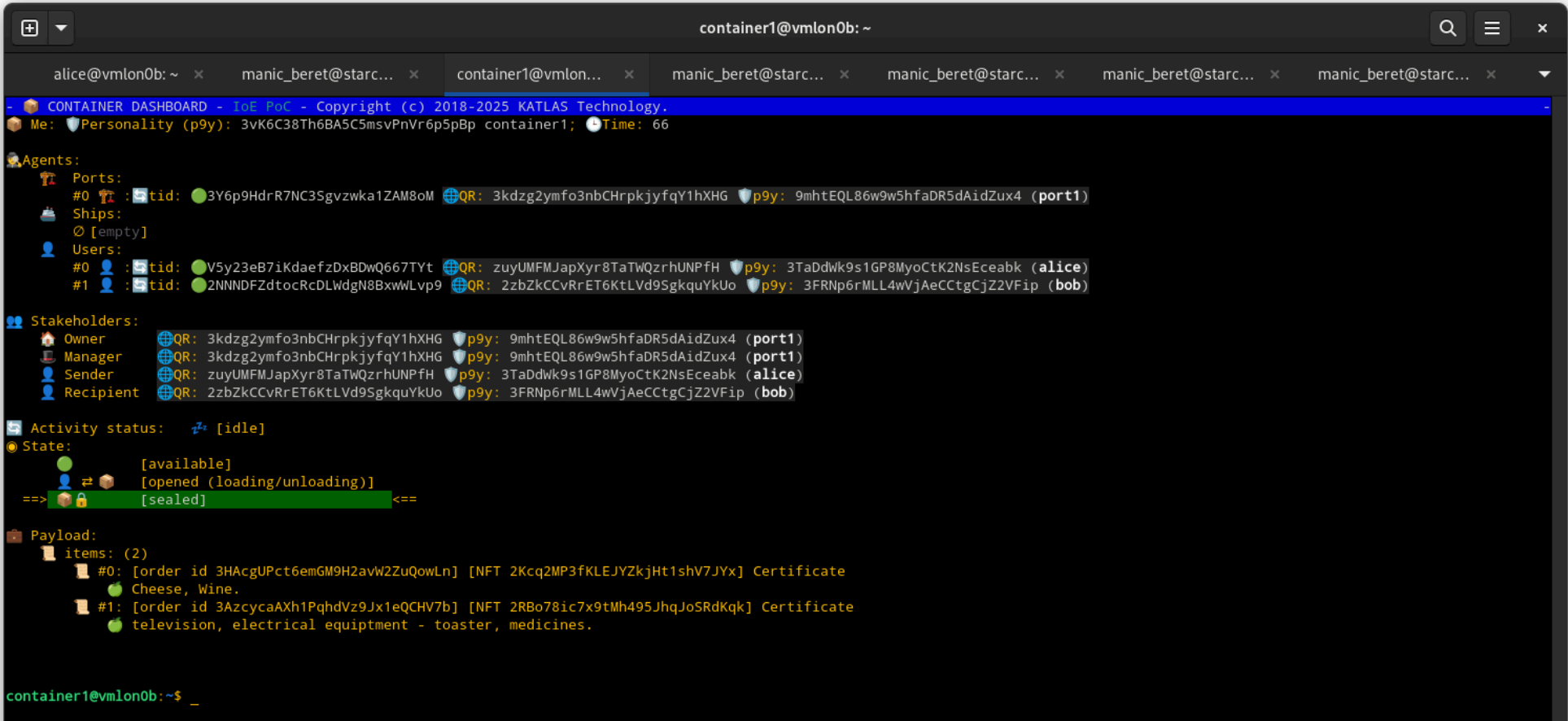
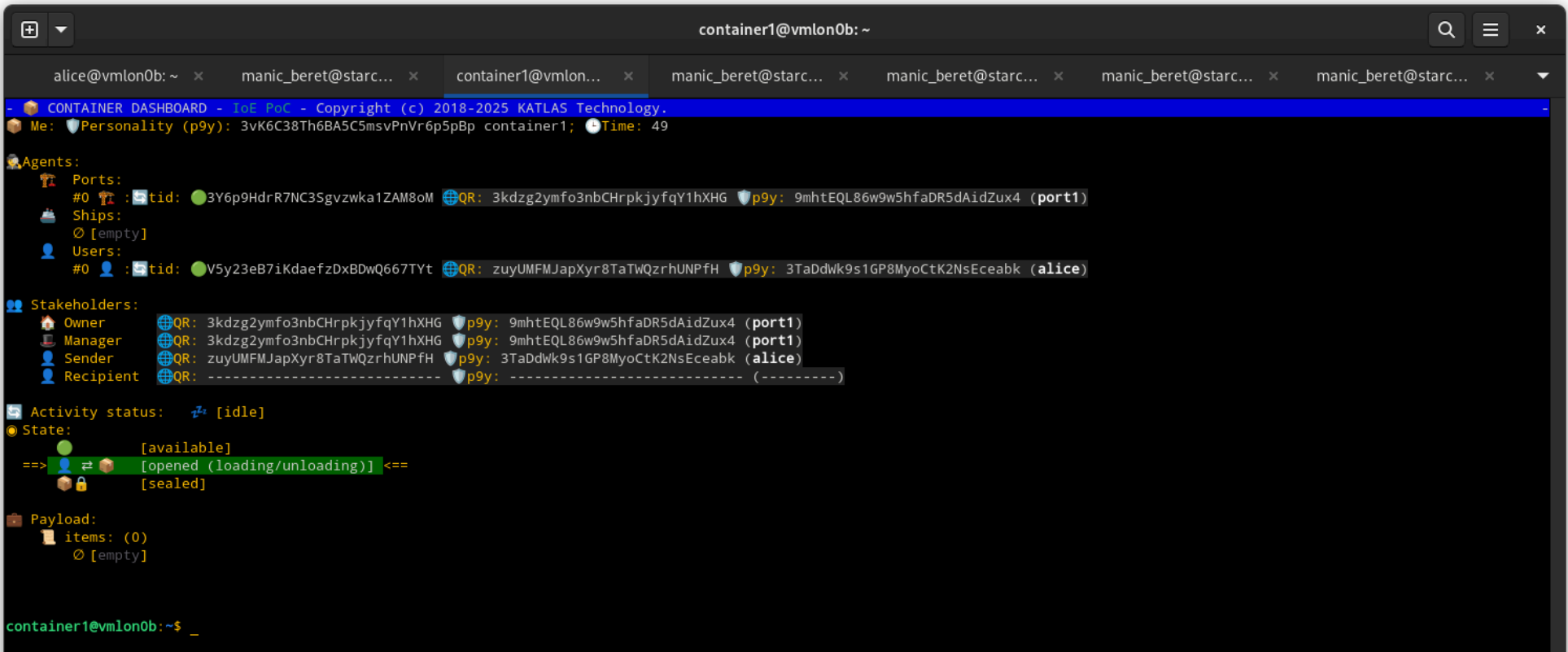
 Screenshots for other statuses

The following screenshots illustrate how the **container dashboard** appears when the container transitions through the **opened** and **sealed** states.

These views highlight changes in:

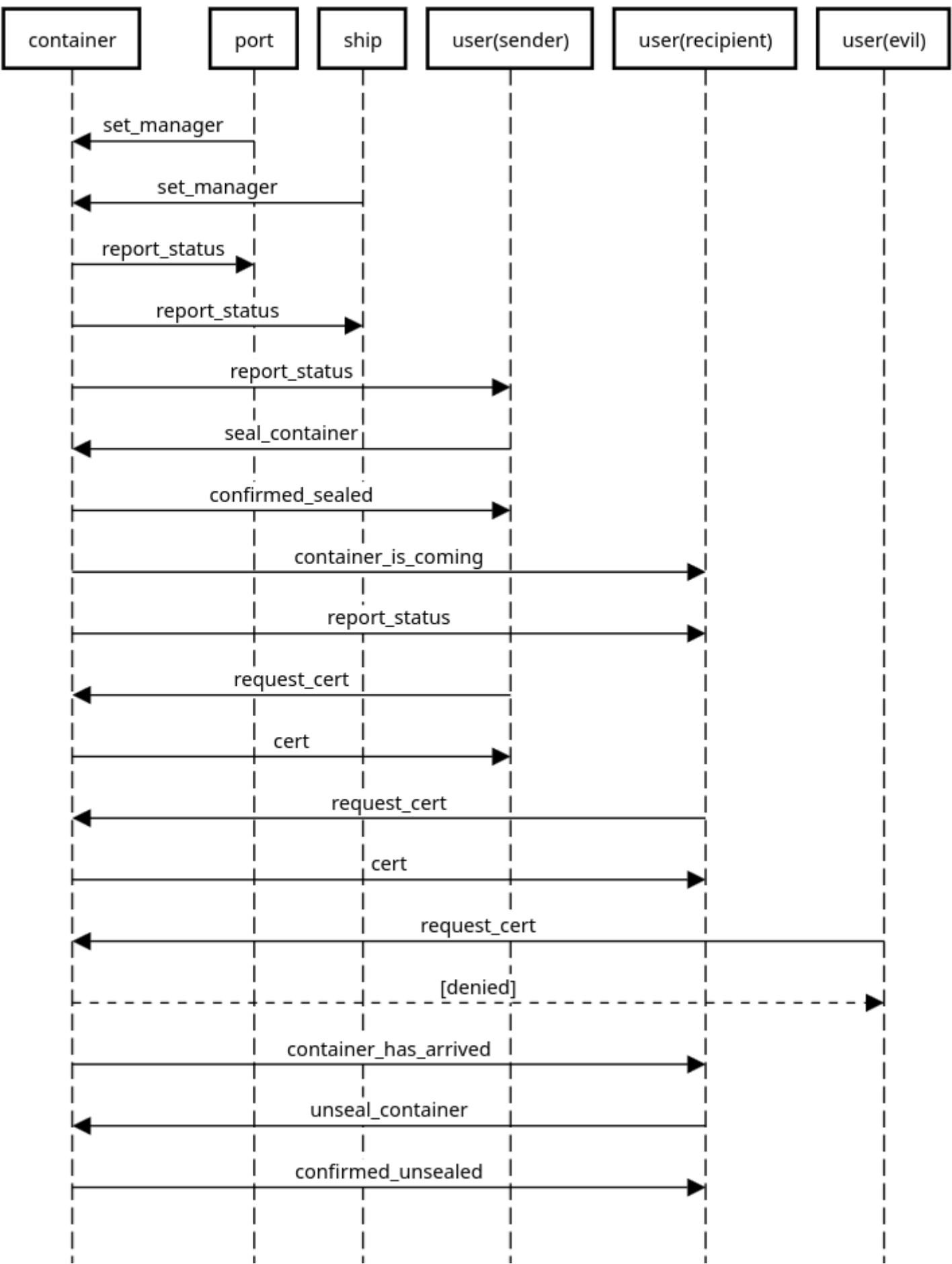
- The **state indicator**, reflecting the current lifecycle stage
- The **payload section**, which evolves as items are added and the container is sealed
- The **stakeholders**, showing updates such as the assignment of sender and recipient

These snapshots help visualize the container's behavior as it moves from active handling to secured transit.



 Interactions

Sequence diagram illustrating the exchange of messages



This **sequence diagram** illustrates the complete lifecycle of a container transaction, from assignment and sealing to delivery and unsealing, involving all key actors: container, port, ship, sender, and recipient.

 Summary of Key Phases

1. Assignment & Sealing

- A **port** or a Ship resets container's new **manager** at convenient times (unload/load ship)
- The **sender** initiates **seal_container**, locking in the payload and recipient identity.
- The container confirms it is **sealed**.

2. Transit Notification

- The container notifies **recipient** that a **sealed container is coming**.
- Status updates are propagated to maintain state awareness.

3. Certificate Exchange

- Several actors (including recipient) **request certificates** for verification (e.g., delivery proof).

- Valid requests are granted; invalid ones are explicitly **denied**.

4. Arrival & Unsealing

- Once arrived, the container receives **unseal_container** from the **authorized recipient**.
- Upon successful validation, it confirms with **confirmed_unsealed**.

✔ Highlights

- Strict access control: only the assigned **recipient** can unseal.
- **Verifiable credentials** exchanged to assert authenticity and permissions.
- Shows **privacy-preserving design**, where sensitive roles (like recipient) are container-managed and not port-visible.

📺 watch

You can type at any time **user watch** to observe the **live activity of the container**. To exit watch mode, press **CTRL-C**.

🎥 Screencast

You can watch a **screencast recording** showing the container1 activity during a single delivery from Alice to Bob.

		Container dashboard  KATLAS-PoC-SupplyChain-DEMO__Container1 https://www.youtube.com/watch?v=Vby933RDlx4&list=PLxosfK0Tg_rWlddsiz4Nad40EJmMglM5G&index=1
---	--	--

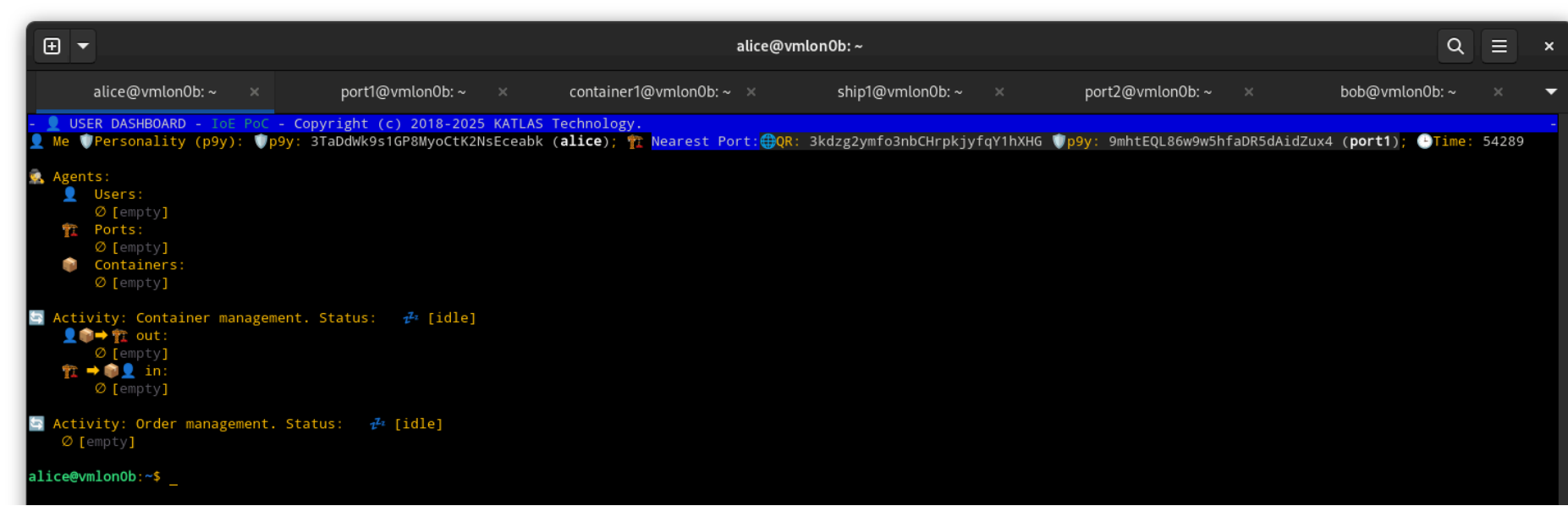
Role: User. Actors: alice, bob

For this walkthrough, we'll **impersonate** **alice**, but the behavior and dashboard layout apply identically to **bob** or any actor running under the **user** role.

 Note: The term **user** is used both to describe the **role** (e.g. actor type: user) and the **CLI wrapper utility** named **user**, which interfaces with the wallet. Despite the name overlap, context should make their usage clear.

To access the user's dashboard, run the following command from **alice's** terminal:

```
user st
```



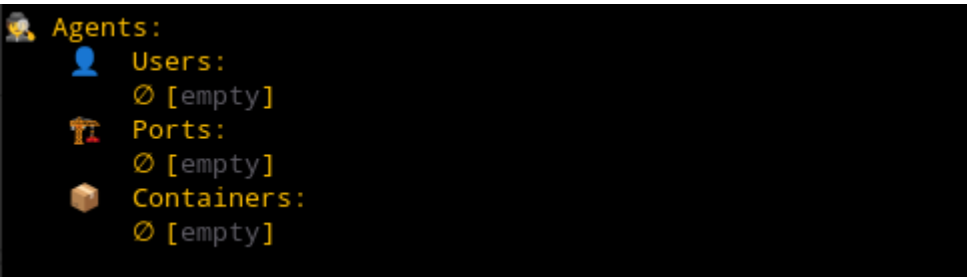
Header

```
Nearest Port: QR: 3kdzg2ymfo3nbCHrpkyfqY1hXHG p9y: 9mhtEQL86w9w5hfaDR5dAidZux4 (port1);
```

As with other roles, the **header section** displays the user's **personality (p9y)** and the current **simulation time tick**.

In addition, for the **user role**, it also shows the **endpoint of the nearest port**, which is determined from the user's **bookmarks file**. This helps the user identify the primary logistics node they are expected to interact with when initiating or receiving container operations.

Agents



This section displays the **active connections and known peers** the user is interacting with. These include:

- **Other users** (e.g. recipients or senders in peer-to-peer exchanges)
- The **nearest port**
- Any **containers** currently being sent or received

These connections are managed automatically by wallet agents and reflect the live state of role-to-role protocols.

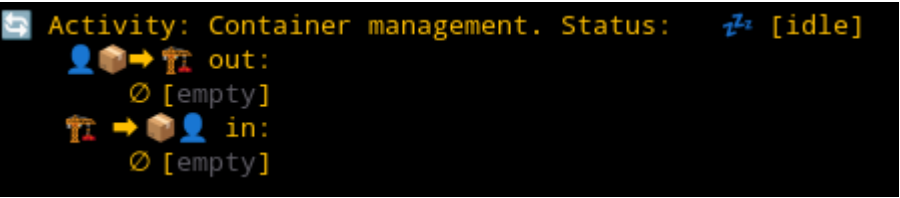
User Activities

Users are responsible for two parallel management workflows:

1.  **Container Management**
 - Outgoing containers prepared for shipment
 - Incoming containers arriving for unsealing and receipt
2.  **Order Management**
 - Creating new shipment requests
 - Tracking the fulfillment and delivery of their orders

These activities are reflected in the dashboard through agent status lines and summarized under their respective sections.

 Container Manager (CM)



This widget displays two internal queues used to track **container flow** from the user’s perspective:

- **Outbound queue:** Containers being prepared for shipment
- **Inbound queue:** Containers arriving from other users, awaiting unsealing

 Outbound Workflow Logic

As orders accumulate for a given recipient:

1. When **at least one order** is registered for a recipient, the Container Manager **engages the port** to **procure a container**.
2. The container, once assigned and opened, will appear in the **outbound queue**, marked with state **opened**.
3. When **two or more orders** are accumulated for the same recipient (a fixed threshold for this demo), the container will automatically be **sealed**.
4. Upon sealing, the container leaves the user’s scope and transitions to the **departure queue of the nearest port**, beginning its physical journey.

The **status field** in the Container Manager widget reflects the current step within the user’s container-related activity. Each state models a distinct stage in the lifecycle of outbound or inbound container interaction.

Icon	State Name	Description
	idle	No current task; awaiting new activity
	allocating_container	Ensuring every send order has a corresponding container in the outbound queue
	contacting_port	Requesting a new container from the nearest port
	sealing_container	Sealing a container and handing it over to the port
	unsealing_container	Unsealing a received container from the inbound queue
	collect_items	Retrieving the payload from an open inbound container

 Timing Behavior

Each status is **artificially extended across several ticks** to simulate operational delays and enhance observability for humans monitoring the system.

 Note: These delays are **simulated only**—the system is idle during this time and not consuming CPU cycles. This same behavior applies to **Order Manager** statuses as well, ensuring consistent user experience across modules.

 Order Manager (OM)

```
Activity: Order management. Status: 🧘 [idle]
🌀 [empty]
```

Send Order: Represents the intention to dispatch goods to a specific recipient.
Receive Order: Represents the expectation of receiving goods from a specific sender.
Status of the Order Manager can be any of:

Icon	Status Name	Description
🧘	idle	No current activity.
📞👤	contacting recipient	Initiating communication with the intended recipient.
💬👤	chat with recipient	Engaged in a conversation to agree on delivery details.
📦🕒	waiting for container	Waiting for the Container Manager to assign a container for this order.
📦←🍏	loading container	Transferring the items listed in the order into the assigned container.

 Order Ledger

The **Order Manager (OM)** maintains a structured ledger that records all orders initiated or expected by the user. Each ledger entry represents a **send** or **receive** order and includes its associated **order status**, which reflects the current progress in the delivery workflow.

Below is an example of the ledger containing a **send** order entry:

```
order id: 49twQfBZPszMwio7fgw1n8fyJTzD 👤->✉️ send; status: ⌚ [waiting]
👤 Recipient 🌐QR: 2zbZkCCvRrET6KtLVd9SgkquYkUo 🛡️p9y: 3FRNp6rMLL4wVjAeCCtgCjZ2VFip (bob)
🚢 Port arrival 🌐QR: ----- 🛡️p9y: ----- (-----)
📦 Container 🌐QR: ----- 🛡️p9y: ----- (-----)
📜 Certificate 2RBo78ic7x9tMh495JhqJoSRdKqk 🍏 television, electrical equiptment - toaster, medicines.
```

This entry represents Alice’s intention to send a container of goods to Bob. At this stage, the container and port of arrival have not yet been assigned, and the order remains in the **waiting** state.
The following ledger entry represents a **receive** order. It tracks the expectation of receiving goods from another user. Once the sender and recipient have finalized the delivery agreement, the order advances to the **agreed with sender** state:

```
order id: 49twQfBZPszMwio7fgw1n8fyJTzD ✉️->👤 recv; status: 👤👤👤 [agreed with sender]
👤 Sender 🌐QR: zuyUMFMJapXyr8TaTWQzrhUNPfh 🛡️p9y: 3TaDdwk9s1GP8MyoCtK2NsEceabk (alice)
🚢 Collect at 🌐QR: 33nCR3WseaXRxyJeGYquYhju1RQ4 🛡️p9y: 2rf47mCQupUfR45B3nAJeL6RV919 (port2)
📦 Container 🌐QR: ----- 🛡️p9y: ----- (-----)
```

In this case, Bob is expecting a container from Alice. The pickup is arranged at **port2**, and the container assignment is still pending.

 Order status

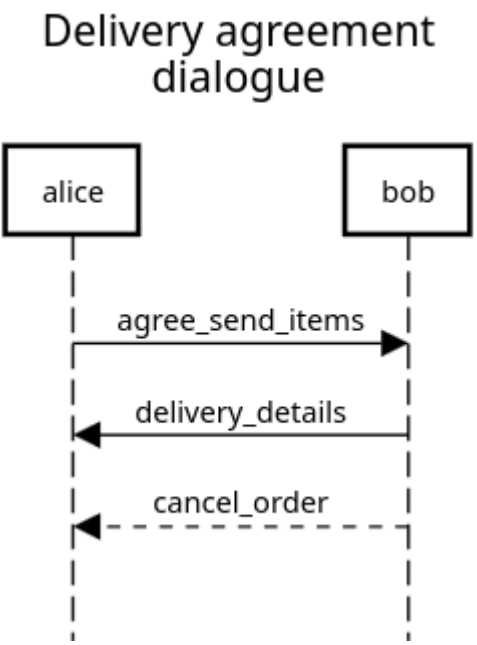
Each order in the ledger, whether a **send** or **receive** order, progresses through a defined set of statuses managed by the **Order Manager (OM)**. Below is a comprehensive list of possible states:

 Symbol	Status	Applies to	Description
	waiting	send	Order has been queued; OM will begin processing shortly.
	agent dealing with recipient	send	OM is initiating a connection with the recipient.
	waiting chat	send	Waiting for the recipient to join the chat.
	chat with recipient	send	Negotiation in progress via chat session.
	agreed with recipient	send	Recipient has accepted the delivery terms.
	agreed with sender	recv	Sender has committed to sending the goods.
	waiting for container	send	OM is waiting for a container to be allocated by CM.
	ready to load	send	Container is assigned and ready to receive goods.
	loading	send	OM is busy loading items into the container.
	payload in container	send	Goods have been successfully placed in the container.
	container sealed and in transit	send/recv	Container has been dispatched and is en route.
	ready for collection	recv	Container has arrived at destination port, ready to be opened.
	Opening sealed container	recv	OM is processing the unseal and unloading steps.
	delivered	send/recv	Final state: goods delivered and order marked as complete.
	error	send/recv	Order failed or was cancelled due to an error or peer disagreement.

 Interactions

This section presents a series of **sequence diagrams** that illustrate the **messaging protocols and role-to-role negotiations** that take place as part of the delivery workflow.

 Users agreement - Dialogue Flow



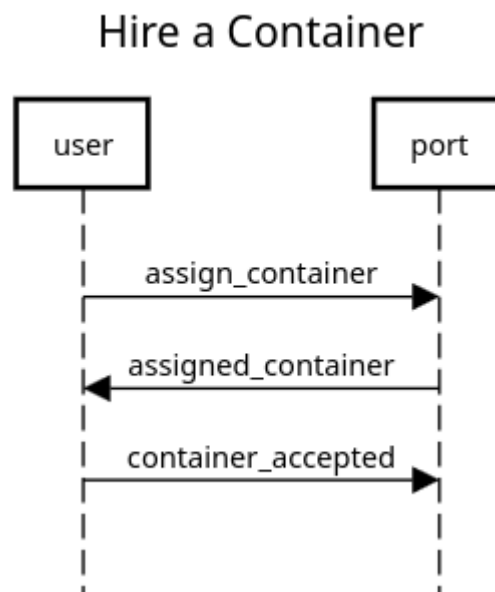
This **sequence diagram** illustrates the **delivery agreement dialogue** between two users—**Alice** (sender) and **Bob** (recipient). It models the peer-to-peer negotiation process involved in setting up a send order.

 Messages

- agree_send_items**
 - Alice initiates the delivery request, offering to send items to Bob.
- delivery_details**

- Bob responds with delivery details, confirming his availability and preferences (e.g., preferred port, delivery terms).
3. **cancel_order** (*optional, dashed*)
- If Bob decides to reject the delivery, he can cancel the order. This is represented as an optional termination path.

Hire a Container – Interaction Flow



This **sequence diagram** illustrates the interaction between a **user** and a **port** when requesting to hire a container. It represents part of the outbound logistics process managed by the **Container Manager (CM)**.

Messages

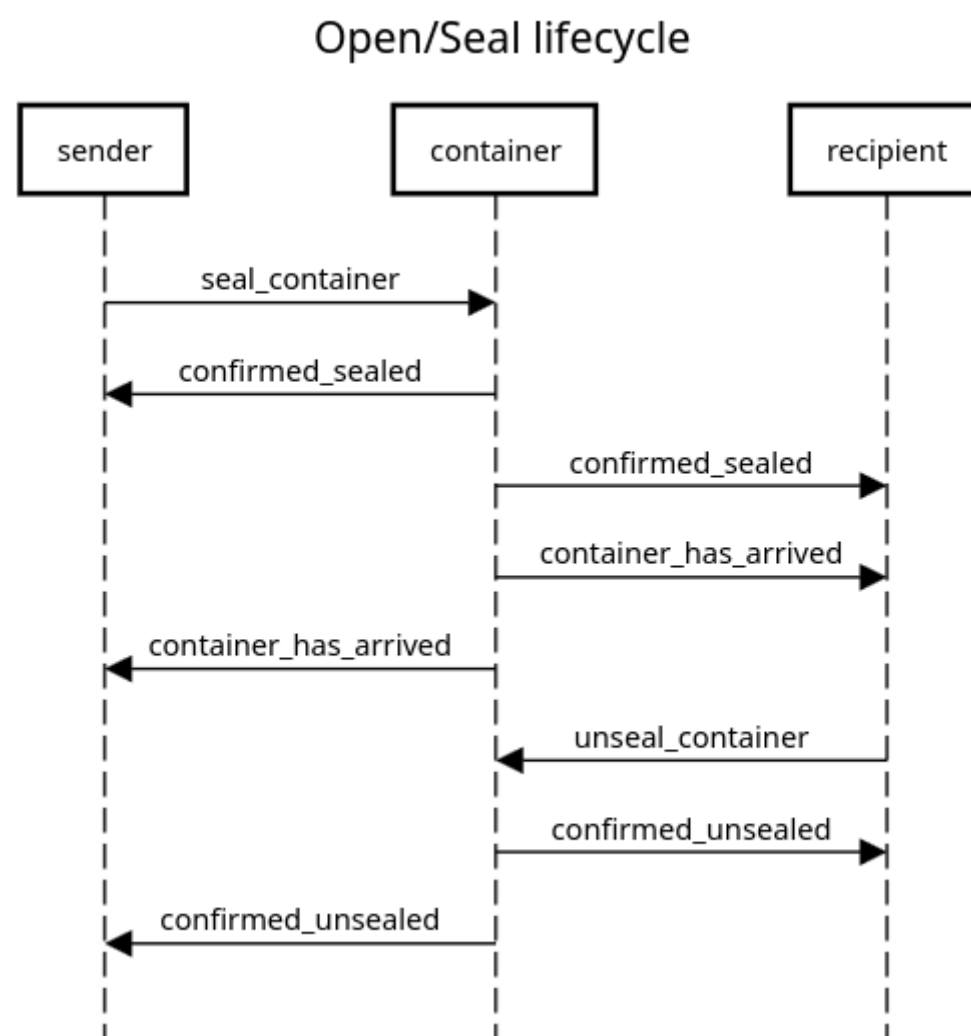
1. **assign_container**
The user requests the port to provide a container suitable for fulfilling one or more send orders.
2. **assigned_container**
The port selects and assigns an available container, responding with its identity and connection details.
3. **container_accepted**
The user acknowledges receipt of the container assignment, confirming readiness to proceed with loading.

Contextual Role

This exchange is triggered automatically when:

- A **user has a pending send order**, and
- No container is yet associated with that recipient

Open/Seal Lifecycle – Role User Message Flow



This **sequence diagram** illustrates the full **Open/Seal Lifecycle** of a container, highlighting the message flow between the **sender**, **container**, and **recipient**.

 Messages

seal_container

- Initiated by the **sender** to finalize the container contents and lock it.
- Marks the transition from **opened** to **sealed**.

confirmed_sealed

- The container acknowledges sealing and notifies both **sender** and **recipient** that it is now sealed and ready for dispatch.

container_has_arrived

- Once the container reaches its destination port, it announces its arrival to both the **recipient** and the **sender**.

unseal_container

- The **recipient** initiates the unsealing process to retrieve the goods.

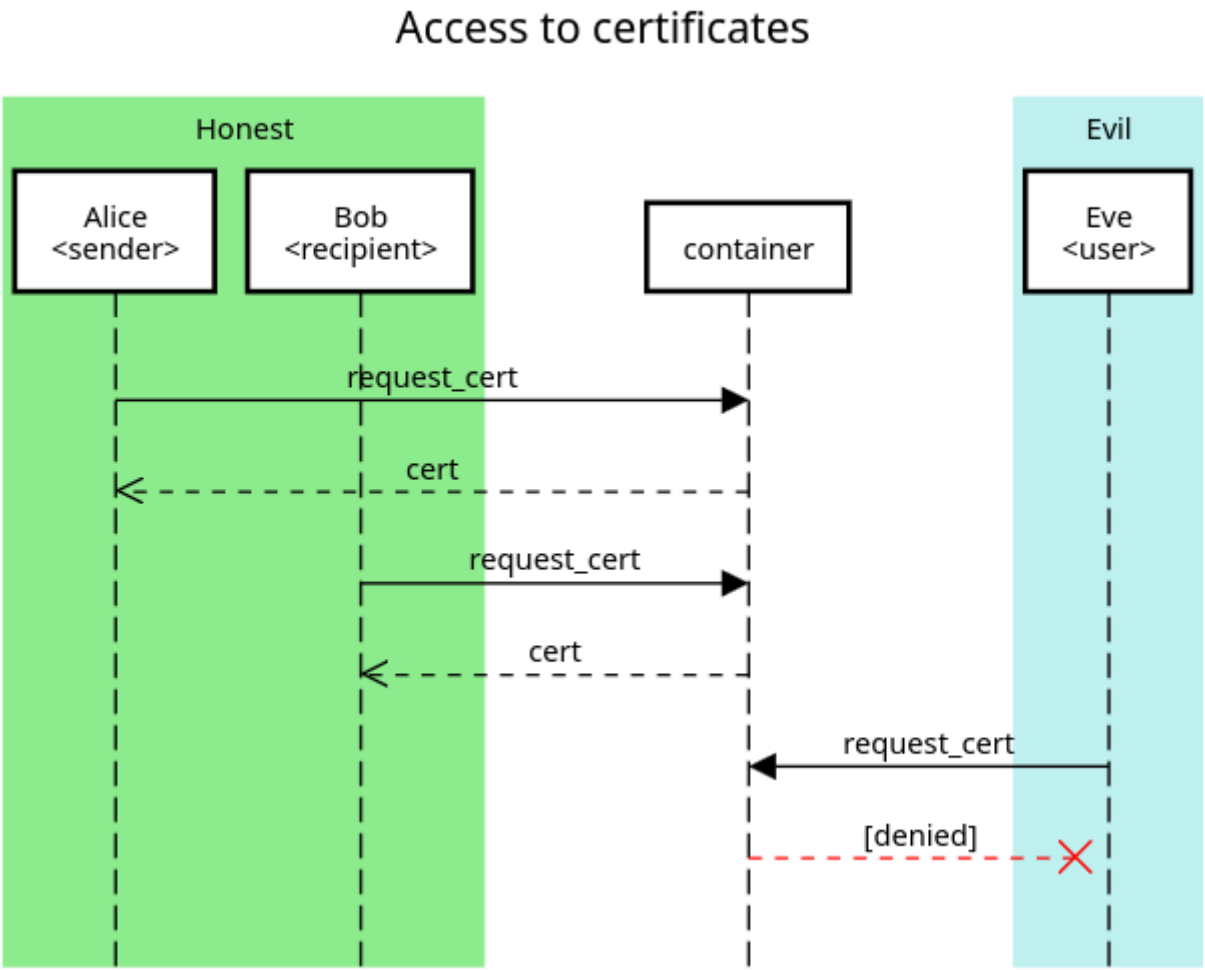
confirmed_unsealed

- The container confirms successful unsealing and informs both the **recipient** and **sender**, completing the delivery lifecycle.

 Contextual Notes

- The container acts as a **trust anchor**: it manages confidentiality of the payload and enforces access control by only responding to the authorized recipient.
- The **sender is never aware of the recipient's internal actions**, preserving privacy-by-design principles.
- This lifecycle reflects the transition through container states: **opened** → **sealed** → **arrived** → **unsealed**.

 Access to certificates



This **sequence diagram** demonstrates the **selective access control** enforced by a container when handling certificate requests. The container ensures that **only authorized parties**—specifically the **sender (Alice)** and the **recipient (Bob)**—can retrieve certificates associated with a delivery. Unauthorized users, like **Eve**, are explicitly denied access.

 Messages

 Honest Actors (Green Zone):

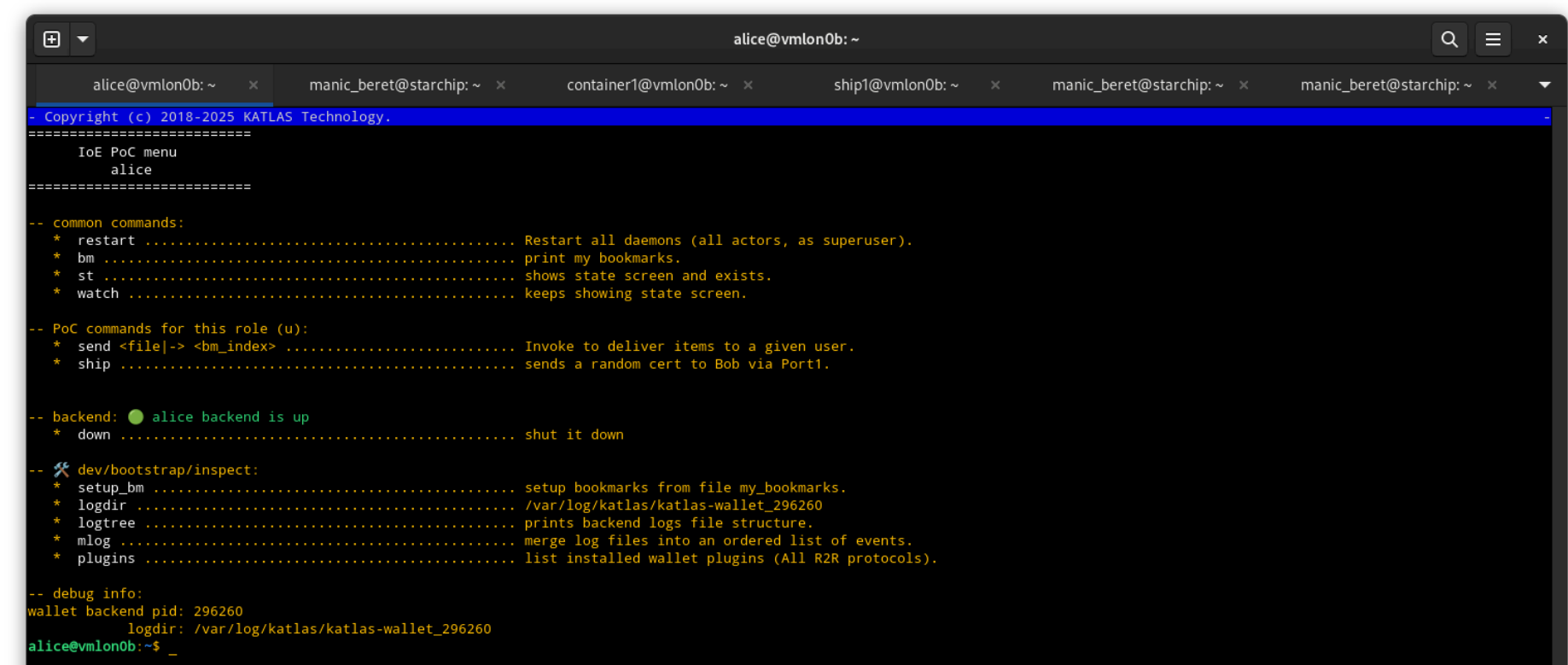
- **Alice (sender)** sends a `request_cert` to the container
→ The container returns the corresponding `cert`
- **Bob (recipient)** sends a `request_cert` as well
→ The container grants access and returns the `cert`

 Malicious Actor (Blue Zone):

- **Eve** (an unauthorized user) attempts a `request_cert`
→ The container denies the request, responding with `[denied]`

 Issuing our first order - Step-by-Step Example

To create your first delivery order, **switch to Alice's terminal** and run the `user` utility:
type `user`



```
alice@vmlon0b: ~  
-----  
Copyright (c) 2018-2025 KATLAS Technology.  
-----  
IoE PoC menu  
alice  
-----  
-- common commands:  
* restart ..... Restart all daemons (all actors, as superuser).  
* bm ..... print my bookmarks.  
* st ..... shows state screen and exists.  
* watch ..... keeps showing state screen.  
-- PoC commands for this role (u):  
* send <file|-> <bm_index> ..... Invoke to deliver items to a given user.  
* ship ..... sends a random cert to Bob via Port1.  
-- backend: ● alice backend is up  
* down ..... shut it down  
-- dev/bootstrap/inspect:  
* setup_bm ..... setup bookmarks from file my_bookmarks.  
* logdir ..... /var/log/katlas/katlas-wallet_296260  
* logtree ..... prints backend logs file structure.  
* mlog ..... merge log files into an ordered list of events.  
* plugins ..... list installed wallet plugins (All R2R protocols).  
-- debug info:  
wallet backend pid: 296260  
logdir: /var/log/katlas/katlas-wallet_296260  
alice@vmlon0b:~$
```

 `send` Command Syntax

```
* send <file|-> <bm_index> ..... Invoke to deliver items to a given user.
```

Arguments

1. - (hyphen) indicates that you want to input the list of items manually via the keyboard.
2. The optional bookmark index points to the recipient's entry in Alice's bookmarks. If omitted, the program will prompt you to choose from the list interactively.

 Step-by-Step Example

1. Run the command: `user send -`
2. Enter your list of items when prompted, one per line (e.g., `television`, `medicines`, `toaster`).
3. When prompted, select the recipient (e.g., `bob`) from the bookmark list.

```
alice@vmlon0b:~$ user send -  
Enter lines with items to send: (press ctrl+d when finished, or ctrl+c to abort)  
_
```

(Don't forget the hyphen after `send` — it tells the program to accept items from standard input.)

The order will be submitted and appear in your order ledger, ready to be processed by the Order Manager.

📦 Entering the Payload

Type in your items (one per line), then press **CTRL-D** to finish.
Example:

```
alice@vm1on0b:~$ user send -
Enter lines with items to send: (press ctrl+d when finished, or ctrl+c to abort)
Defibrillator
Stethoscope
Infusion pump
-
```

🥚 Easter Egg Warning

For demonstration purposes, the system includes a special edge case:

🚫 **Never include "muffins"** in your item list — doing so will cause the recipient to reject the order!

This allows you to test how the system handles order refusals gracefully.

✅ Certificate Summary

After item input, the system will:

- 1. Save the list to a file in the `items/` directory.
- 2. Use the wallet backend to generate a **digital certificate**, signing the payload with Alice’s private key.
- 3. Certs are kept in wallet's custody

📧 Recipient Selection

The program will now prompt you to choose a recipient from your **bookmarks**:

```
alice@vm1on0b: ~
----- summary -----
* payload: cert id mkBf3wZczZFKVZZKJJegtbc3sMb
-- BEGIN OF DOCUMENT -----
Certificate
Document date: ~Mon Jun 09 2025 16:00:17 UTC
Expires: ~Sat Aug 30 2059 16:00:17 UTC

Parameters:
  CA: SOA Start of Authority
  lang: en
  version: 1

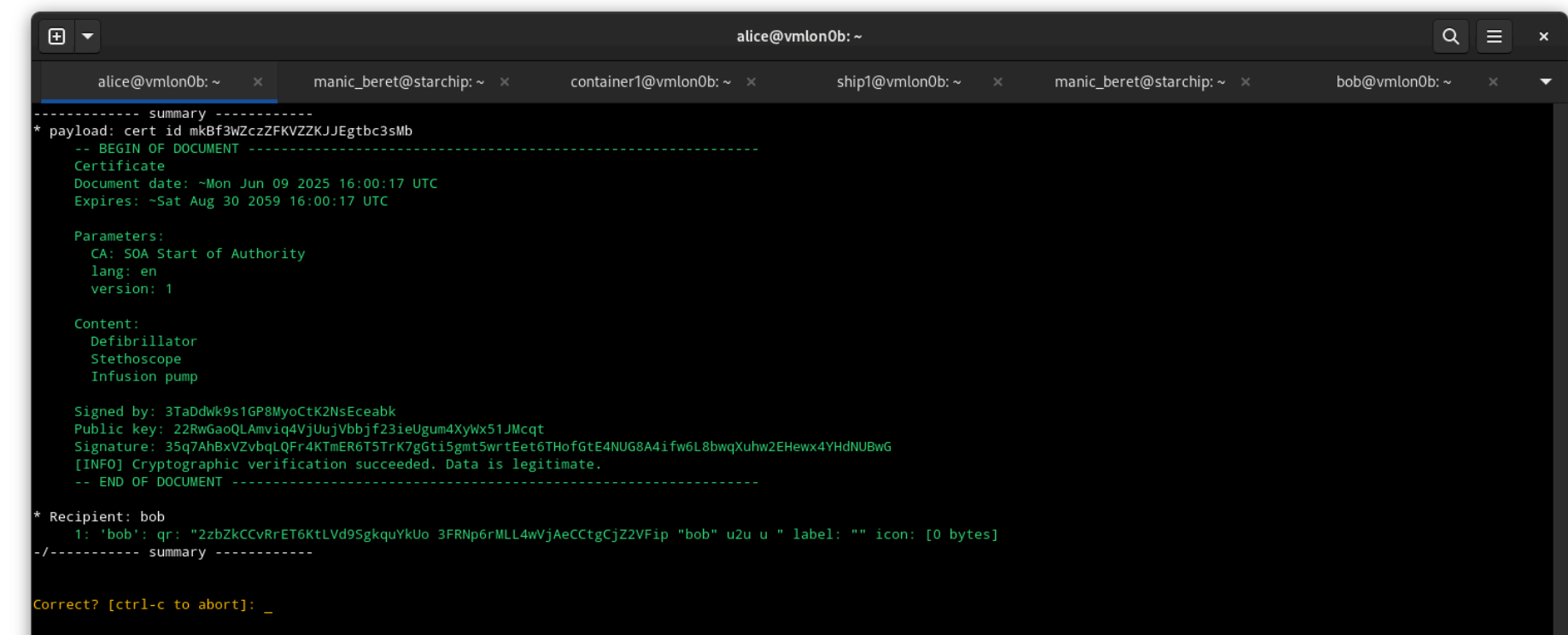
Content:
  Defibrillator
  Stethoscope
  Infusion pump

Signed by: 3TaDdWk9s1GP8MyoCtK2NsEceabk
Public key: 22RwGaoQLAmviq4VjUujVbbjf23ieUgum4Xywx51JMcqt
Signature: 35q7AhBxVZvbqLQFr4KTmER6T5TrK7gGti5gmt5wrtEet6THofGtE4NUG8A4ifw6L8bwqXuhw2EHewx4YHdNUBwG
[INFO] Cryptographic verification succeeded. Data is legitimate.
-- END OF DOCUMENT -----

-/----- summary -----

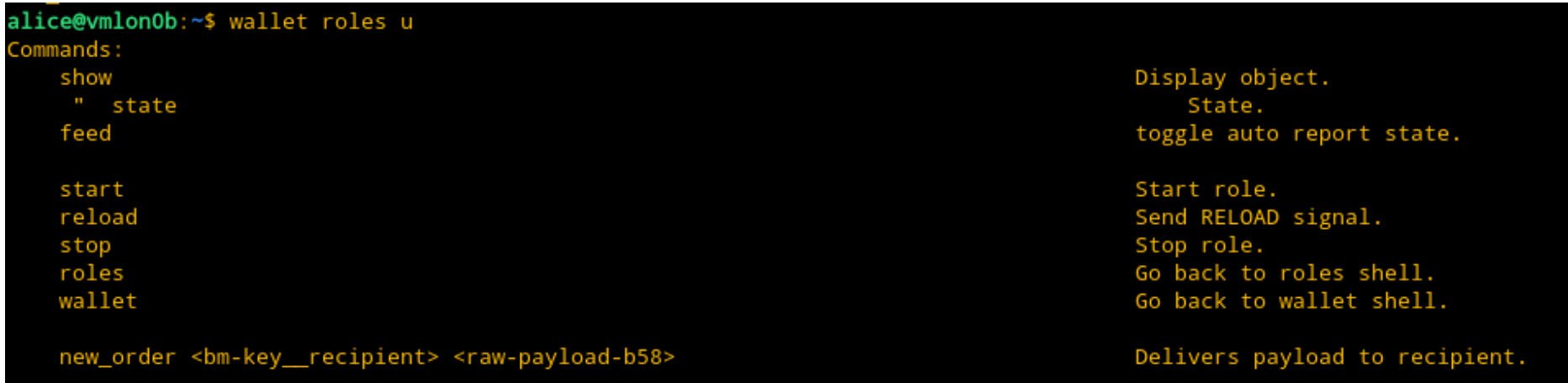
Select recipient user:
1: 'bob': qr: "2zbZkCCvRrET6KtLVd9SgkquYkUo 3FRNp6rMLL4wVjAeCCtgCjZ2VFip "bob" u2u u " label: "" icon: [0 bytes]
Enter bookmark number: _
```

just type 1↵ and hit Enter to complete the process.



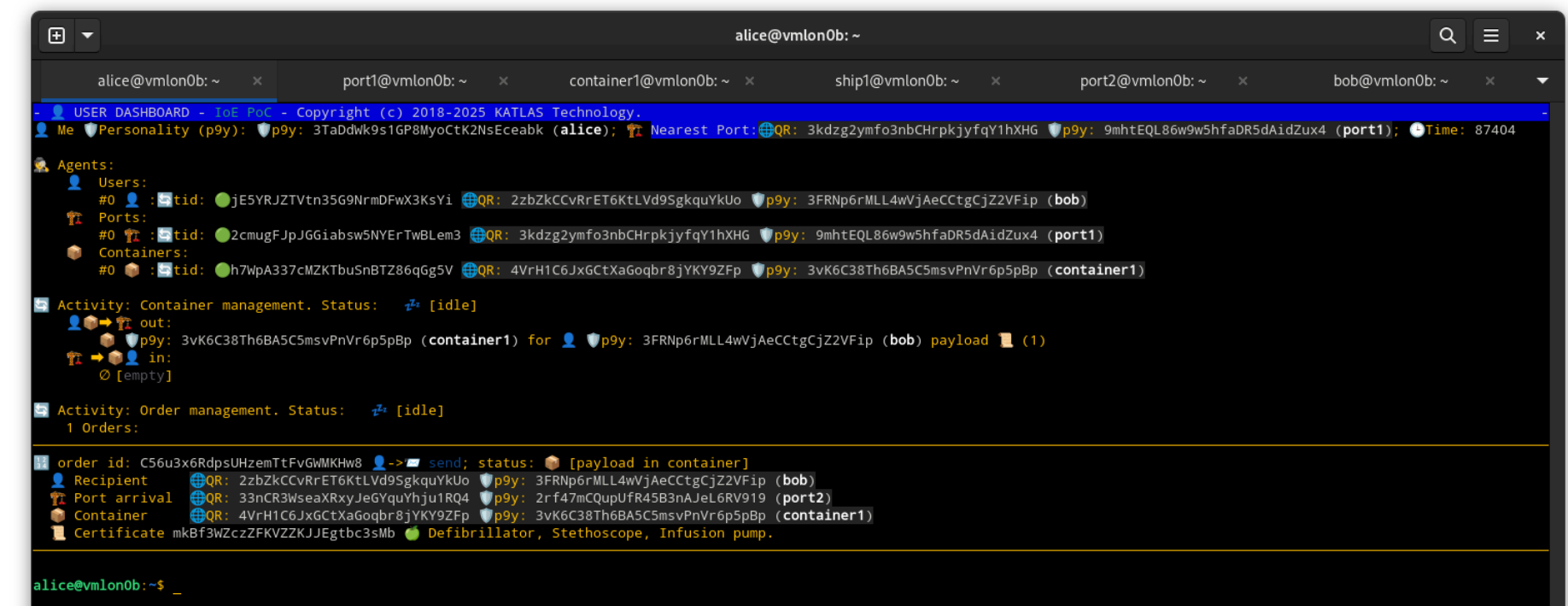
and confirm with [ENTER]

i Behind the scenes, the program triggered a wallet function associated with the user (u) role API, specifically the new_order function.



📋 Check the Dashboard

Type User st to display the updated dashboard.



After cycling through a few transient states handled by our two managers—**Container Manager** and **Order Manager**—we reach a stable situation within a few seconds:

A container now appears in the **outbound queue**, marked as **opened** and ready to accept more payload.

(Recall that the Container Manager is configured to wait until **two orders** are loaded before proceeding with sealing and dispatch.)

 Send Order Widget (Alice's Dashboard)

```
order id: C56u3x6RdpsUHzemTtFvGWMKHw8 -> send; status: [payload in container]
Recipient QR: 2zbZkCCvRrET6KtLVd9SgkquYkUo p9y: 3FRNp6rMLL4wVjAeCCtgCjZ2VFip (bob)
Port arrival QR: 33nCR3WseaXRxyJeGYquYhju1RQ4 p9y: 2rf47mCQupUfR45B3nAJeL6RV919 (port2)
Container QR: 4VrH1C6JxGCtXaGoqbr8jYKY9ZFp p9y: 3vK6C38Th6BA5C5msvPnVr6p5pBp (container1)
Certificate mkBf3WZczZFKVZZKJJegtbc3sMb Defibrillator, Stethoscope, Infusion pump.
```

This widget represents an active **send order**. Let's break it down:

- **ID Order ID:** A unique identifier generated by the sender (Alice) to track the order.
- **Recipient:** Bob, selected from Alice's bookmarks. His identity is displayed along with his QR and p9y.
- **Port Arrival:** Port2 — this is the endpoint Bob provided when asked where to receive the delivery.
- **Container:** `container1` — this was allocated to Alice by the port when her Container Manager triggered a `assign_container` request.
- **Certificate:** Shows the ID of the cryptographic certificate representing the payload (e.g., *Defibrillator, Stethoscope, Infusion pump*).
- **Status:** `payload in container` — meaning the container is loaded but **not yet sealed**. The system is waiting for more orders before dispatch.

This order is now actively visible in the **outbound queue** under Alice's Container Manager, as per her configuration that waits until two orders are loaded before sealing.

 Receive Order Widget (Bob's Dashboard)

```
order id: C56u3x6RdpsUHzemTtFvGWMKHw8 -> recv; status: [agreed with sender]
Sender QR: zuyUMFMJapXyr8TaTWQzrhUNPfh p9y: 3TaDdwk9s1GP8MyoCtK2NsEceabk (alice)
Collect at QR: 33nCR3WseaXRxyJeGYquYhju1RQ4 p9y: 2rf47mCQupUfR45B3nAJeL6RV919 (port2)
Container QR: ----- p9y: ----- (-----)
```

Bob's view of the same order reflects the **recipient-side perspective**:

- **Sender:** Alice, with matching QR and p9y to authenticate the origin.
- **Collect at:** Port2 — the port Bob specified in response to Alice's query.
- **Container:** Still **unassigned** (placeholder shown), because the container has not been sealed yet.
- **Status:** `agreed with sender` — Bob has confirmed he is expecting the delivery, but final delivery details (like the container ID) are still pending.

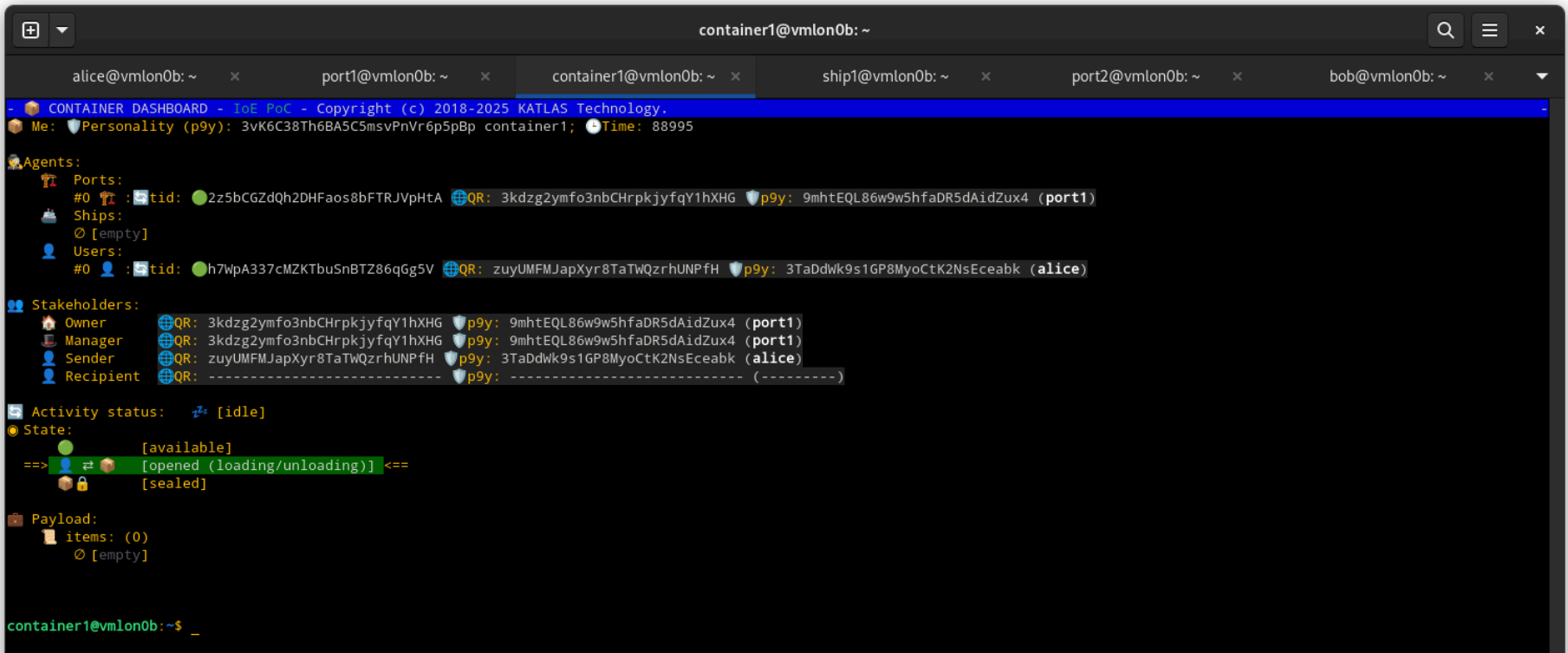
 Observations

- **Order symmetry:** Both sides (Alice and Bob) see the **same order ID**, ensuring consistency.
- **Progress transparency:** Bob doesn't yet see the container details because the shipment is still **open and modifiable**.
- **Privacy:** Only once the container is sealed will its identity become visible to the recipient. This maintains operational flexibility and privacy until the shipment is finalized.

📦 Let's Check on **container1**

Now that the first order has been loaded and the container is waiting for more payload, let's **peek into container1's wallet** to see how it perceives its current state.

Switch to the terminal where you're logged in as **container1**, and type: `user st`



As expected, we find **container1** in the **opened state**.

- It is currently **owned by port1** (which initially assigned it).
- It is also **managed by port1**, reflecting the chain of command during this stage.
- The container is actively **servicing Alice**, the sender who is loading goods into it.

At this point:

- We **do not yet see the recipient** in the dashboard.
- We also **don't see the payload contents**, even though the goods are already *physically loaded*.

📌 This is by design: the container **only discloses recipient identity and payload details once it is sealed**. This approach ensures:

- **Privacy** for the recipient until delivery is finalized
- **Immutability** of contents only after the sealing step
- Better observability of container lifecycle milestones

🚧 Let's Peek Into port1

Now that the container is actively being used for an outbound delivery, let's take a look at how things appear from the perspective of the assigning port.

Switch to **port1**'s terminal and type `user st`

```
port1@vm1on0b: ~  
- PORT DASHBOARD - 10E PoC - Copyright (c) 2018-2025 KATLAS Technology.  
Me Personality (p9y): p9y: 9mhtEQl86w9w5hfaDR5dAidZux4 (port1); Time: 89677; (Online AI chat is available for users.)  
Agents:  
Ships:  
#0 tid: 3nhxU8PGQnxP8DF5HAMHTFMfHzXM QR: 4M4ciabTzKREsvXdDXfpFnzfY8Kz p9y: 42ZuEdDhTuDhkyKbaEjWyH6FNdUB (ship1)  
Containers:  
#0 tid: 2z5bCGZdQh2DHFaos8bFTRJVpHtA QR: 4VrH1C6JxGctXaGoqbr8jYKY9ZFp p9y: 3vK6C38Th6BA5C5msvPnVr6p5pBp (container1)  
Users:  
#0 tid: 2cmugFJpJGGiabsW5NYErTwBLem3 QR: zuyUMFMJapXyr8TaTWQzrhUNPfh p9y: 3TaDdwk9s1GP8MyoCtK2NsEceabk (alice)  
Handling queues:  
Stock - new (0)  
[empty]  
Available (0)  
[empty]  
Reserved (0)  
[empty]  
Opened - loading/unloading (1)  
QR: 4VrH1C6JxGctXaGoqbr8jYKY9ZFp p9y: 3vK6C38Th6BA5C5msvPnVr6p5pBp (container1) QR: zuyUMFMJapXyr8TaTWQzrhUNPfh p9y: 3TaDdwk9s1GP8MyoCtK2NsEceabk (alice)  
Operation queues:  
Departures (0)  
[empty]  
Arrivals (0)  
[empty]  
Activity status: [idle]  
port1@vm1on0b:~$
```

We can observe that **container1** has successfully moved from its initial position in the **stock** queue to its current state in the **opened** queue.

Although we didn't catch it in the intermediate steps, the container **automatically transitioned through** the following states:

- **available** – when it was ready for assignment
- **reserved** – once it was selected for a specific user and order
- **opened** – after being contacted and assigned a duty, now accepting payload

These transitions occurred quickly and automatically behind the scenes as part of the **container manager's workflow**.

 *Tip:* You can repeat the process by issuing new orders to observe each stage more closely. Each run can help focus on a different aspect—such as allocation, sealing, or dispatch—to build a clearer mental model of the logistics lifecycle.

Let's return to **Alice's wallet** to check the current status of her operations.

📦 **Switch back to Alice's terminal** and type `user st`

```
alice@vm1on0b: ~  
- USER DASHBOARD - 10E PoC - Copyright (c) 2018-2025 KATLAS Technology.  
Me Personality (p9y): p9y: 3TaDdwk9s1GP8MyoCtK2NsEceabk (alice); Nearest Port: QR: 3kdzg2ymfo3nbCHrpkyfY1hXHg p9y: 9mhtEQl86w9w5hfaDR5dAidZux4 (port1); Time: 89994  
Agents:  
Users:  
#0 tid: jE5YRJTvtN35G9NrmDFwX3KsYi QR: 2zbZkCCvRrET6KtLVd9SgkquYkUo p9y: 3FRNp6rMLL4wVjAeCCtgCjZ2VFip (bob)  
Ports:  
#0 tid: 2cmugFJpJGGiabsW5NYErTwBLem3 QR: 3kdzg2ymfo3nbCHrpkyfY1hXHg p9y: 9mhtEQl86w9w5hfaDR5dAidZux4 (port1)  
Containers:  
#0 tid: h7WpA337CMZKTbuSnBT86qGg5V QR: 4VrH1C6JxGctXaGoqbr8jYKY9ZFp p9y: 3vK6C38Th6BA5C5msvPnVr6p5pBp (container1)  
Activity: Container management. Status: [idle]  
out:  
p9y: 3vK6C38Th6BA5C5msvPnVr6p5pBp (container1) for p9y: 3FRNp6rMLL4wVjAeCCtgCjZ2VFip (bob) payload (1)  
in:  
[empty]  
Activity: Order management. Status: [idle]  
1 Orders:  
order id: C56u3x6RdpsUHzemTtFvGWMKhw8 --> send; status: [payload in container]  
Recipient QR: 2zbZkCCvRrET6KtLVd9SgkquYkUo p9y: 3FRNp6rMLL4wVjAeCCtgCjZ2VFip (bob)  
Port arrival QR: 33nCR3WseaXRxyJeGYquYhju1RQ4 p9y: 2rf47mCQupUfR45B3nAJeL6RV919 (port2)  
Container QR: 4VrH1C6JxGctXaGoqbr8jYKY9ZFp p9y: 3vK6C38Th6BA5C5msvPnVr6p5pBp (container1)  
Certificate mkBf3WzcZFKVZZKJJegtbc3sMb Defibrillator, Stethoscope, Infusion pump.  
alice@vm1on0b:~$
```

This will bring up the latest **user dashboard**, where we can observe how the **outbound queue** has progressed, and whether the **container is ready to be sealed**, depending on how many orders have been accumulated. The state of the **Order Manager (OM)** and **Container Manager (CM)** should now reflect the most recent actions.

Let’s now issue a **second order** to Bob so the container can be sealed and dispatched.

```
-- PoC commands for this role (u):
* send <file|-> <bm_index> ..... Invoke to deliver items to a given user.
* ship ..... sends a random cert to Bob via Port1.
```

You can repeat the familiar command: **user send -** as many times as you'd like—each execution creates a new send order. However, now that you've seen it once, it's more efficient to let the system’s **automation take over**.

Instead, use the command: **user ship**

 *Note: this is the verb **to ship**, not to be confused with the actor role **ship**.*

This command picks a **random cert** and uses the **first recipient bookmark** (which in our case is Bob) to automatically issue a new order using **user send**.

 Prepare to Track the Delivery Journey

Before issuing this next triggering order—which will cause the container to be sealed, handed over to the port, loaded onto a ship, and transported to Bob—you may want to observe how each actor updates their dashboard in real time.

Open a terminal or tab for each of the following wallets and enter **watch mode**:

port1@vmlon0b:~\$ user watch
container1@vmlon0b:~\$ user watch
ship1@vmlon0b:~\$ user watch
port2@vmlon0b:~\$ user watch
bob@vmlon0b:~\$ user watch

This way, you’ll be able to **visually track** the container’s full journey through the system, observing each actor’s state transitions as they occur.

When you're ready, return to **Alice’s wallet** and trigger the next step:

user ship; user watch

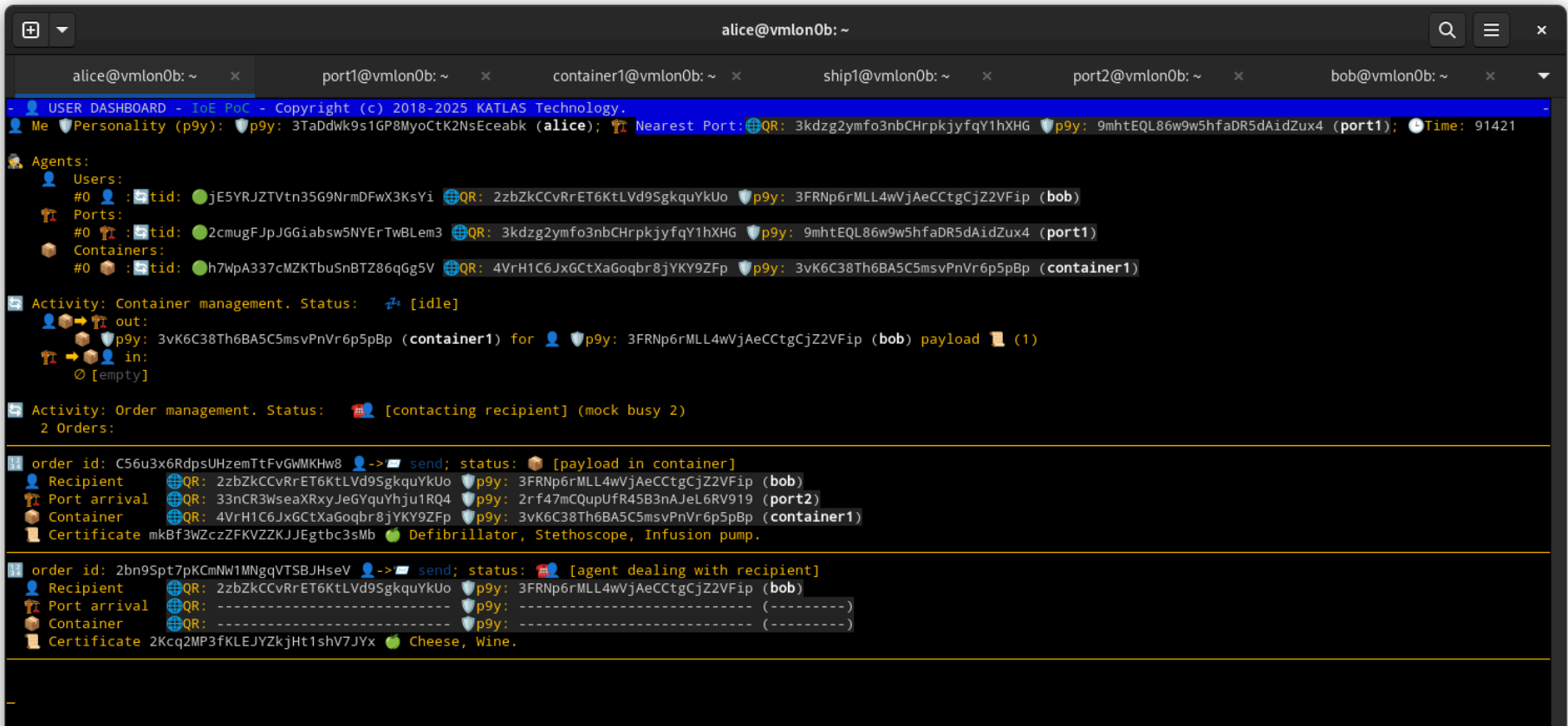
Then sit back and enjoy the orchestration unfold in real time.

Delivery Walkthrough

The following sequence of **10 screenshots** was captured across the involved wallets, illustrating the full journey of **container1** as it traveled from **Alice** (the sender) to **Bob** (the recipient).

Each screenshot reflects a key stage in the lifecycle of the container, as observed from the perspective of the different actors participating in the process.

1. Alice: New order was being agreed with Bob: dialing



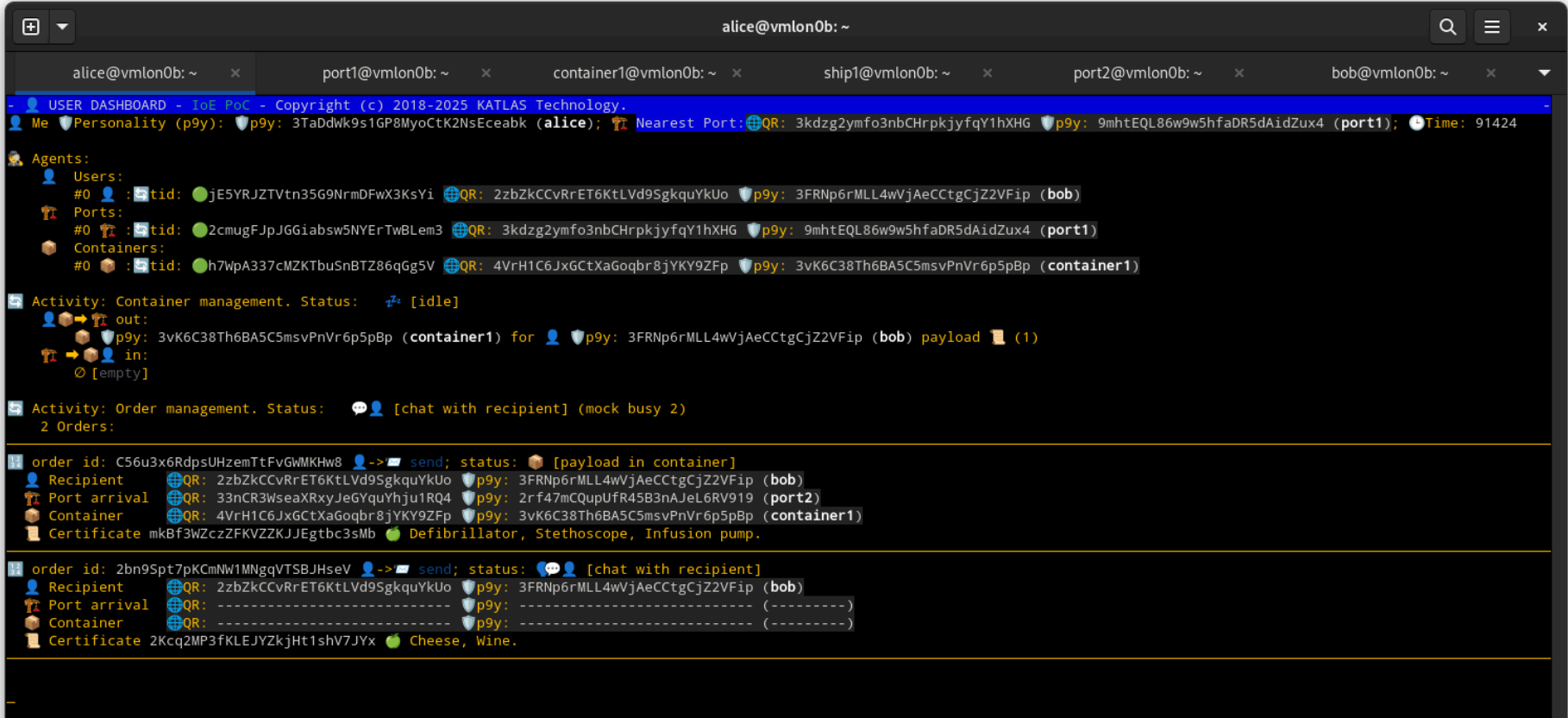
This screenshot shows the **Alice user dashboard** during the moment when she has just sent **two orders** to **Bob**, and her wallet is actively engaged in the **container management and order processing** workflow.

Key Observations:

- **Personality & Nearest Port:** Alice's wallet personality (**p9y**) is visible at the top, and her nearest port is correctly shown as **port1**.
- **Agents:**
 - Bob is listed under "Users."
 - **port1** is her local port.
 - **container1** is the active container assigned for her deliveries.
- **Container Manager:**
 - Status is **idle**, but it shows that **container1** is assigned to Bob and holds **1** payload.
- **Order Manager:**
 - Status: **contacting recipient (mock busy 2)** — the system is handling recipient-side confirmation or final steps.
 - Two **send orders** are listed:
 1. The first one is already **payload in container** (loaded).
 2. The second one is still in **agent dealing with recipient**, awaiting completion of the agreement protocol.
 - Certificates list payload items: e.g., **Defibrillator, Stethoscope, and Cheese, Wine**.

 This snapshot captures the moment right **before the container is sealed**, once the payload threshold (2 orders) is reached.

👁️2. Alice: New order was being agreed with Bob: chatting



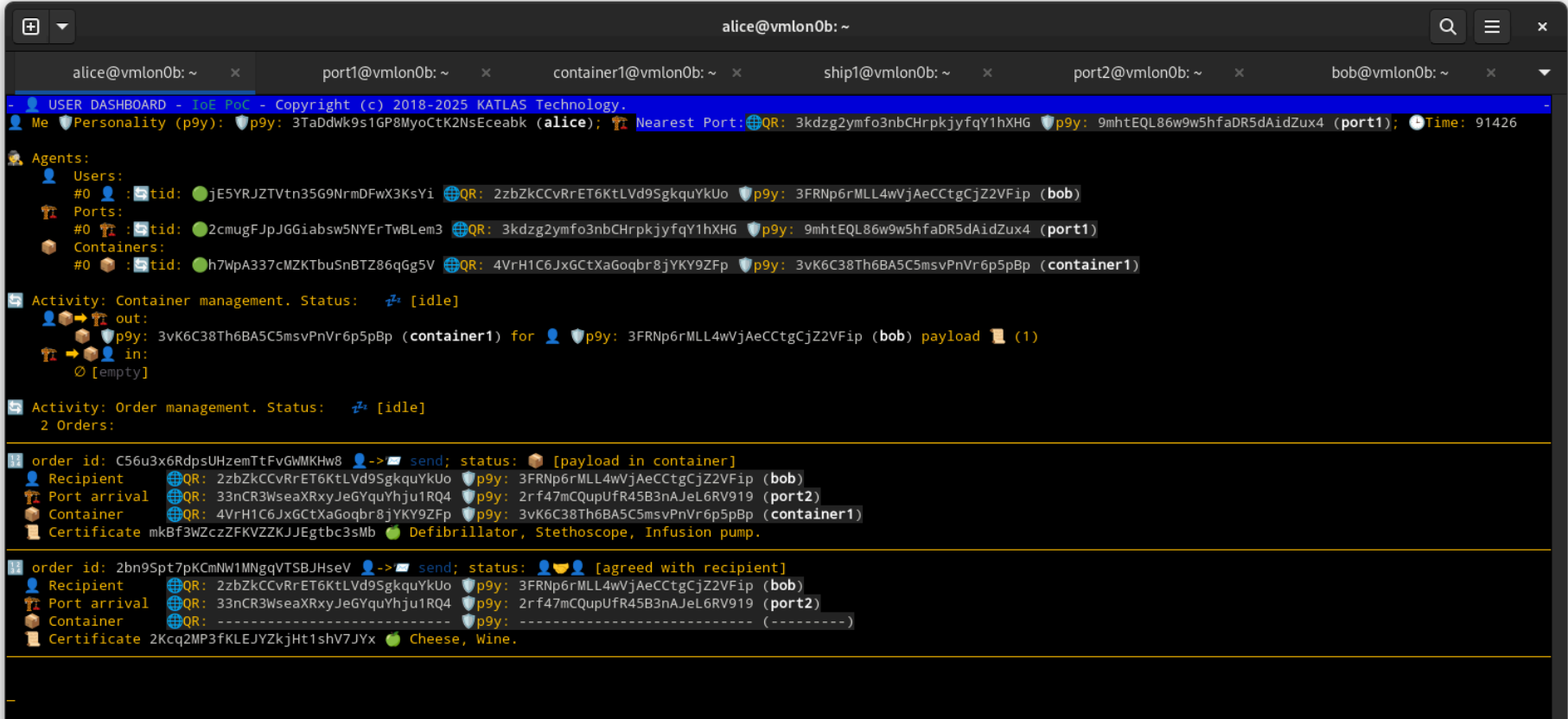
This screenshot shows Alice's dashboard just moments **after both send orders have been accepted by Bob**, and the system is now in the **chat phase** with the recipient for each order.

🟡 **Highlights:**

- **Container Manager** remains **idle**, with **container1** still opened and holding **1** payload (about to reach sealing threshold).
- **Order Manager Status:** **chat with recipient** (mock busy 2) — both orders are actively in the interactive phase, finalizing agreement details.
- Both orders still show **Bob** as the recipient and are in **send** mode:
 - First order: already loaded into **container1**.
 - Second order: chat in progress, container assignment pending.

🕒 This is the brief window before sealing — the system is finalizing user-level confirmation prior to locking the shipment.

👁️3. Alice: New order was being agreed with Bob: agreed



This screenshot shows Alice's dashboard at a **steady state** right before container sealing.

🟢 Key Points:

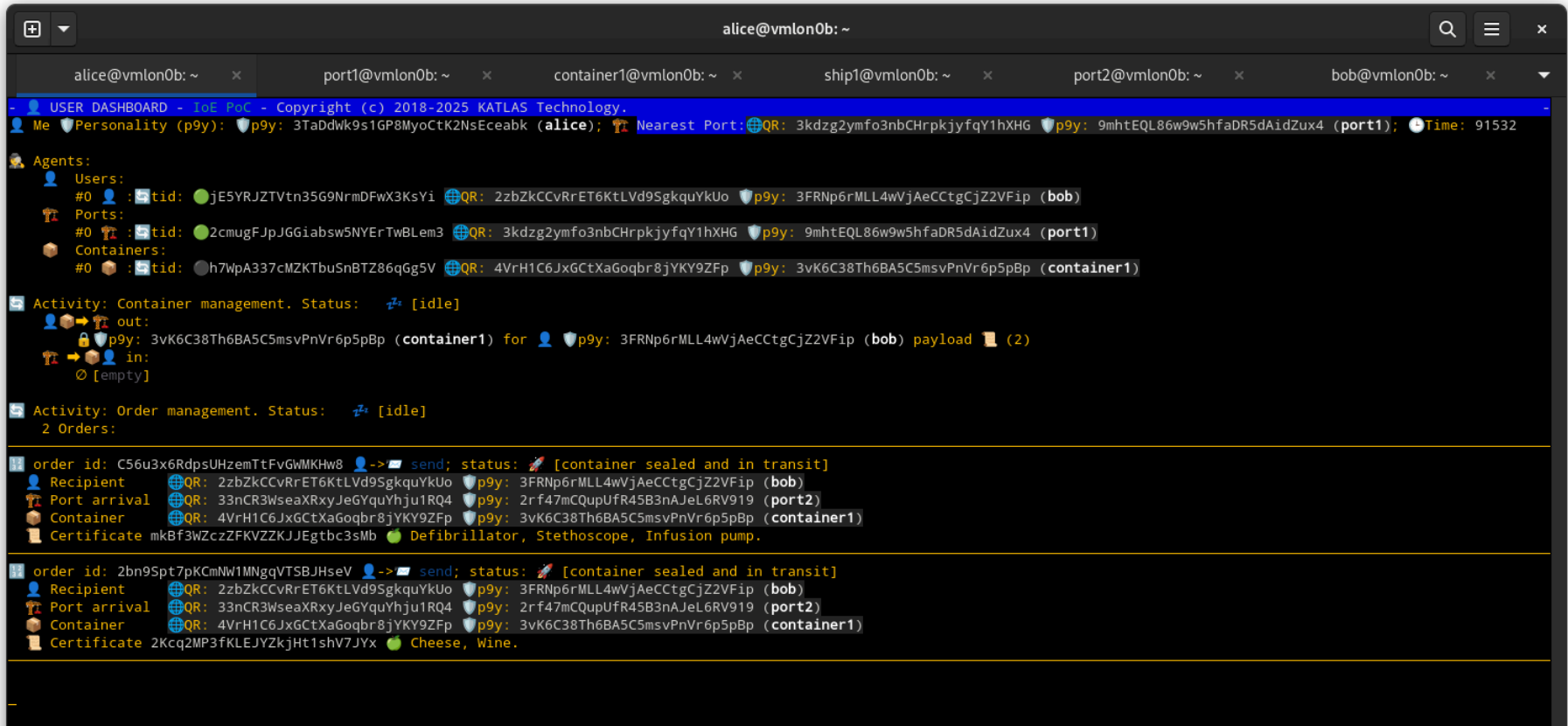
- **Container Manager** and **Order Manager** are both **idle**, indicating all tasks (container assignment and order negotiations) have been completed.
- **Order 1** is fully loaded in **container1**, with payload visible.
- **Order 2** has successfully transitioned to the **agreed with recipient** status, awaiting inclusion in the container.
- **Container1** remains in **opened** state, still accepting payload, with 1 confirmed loaded order.



Once the system detects that the **payload threshold (2 orders)** has been reached, the container will automatically proceed to **seal and dispatch**.

This is the last calm moment before the logistics chain becomes active.

👁️👁️4. Alice: Container sealed and in transit



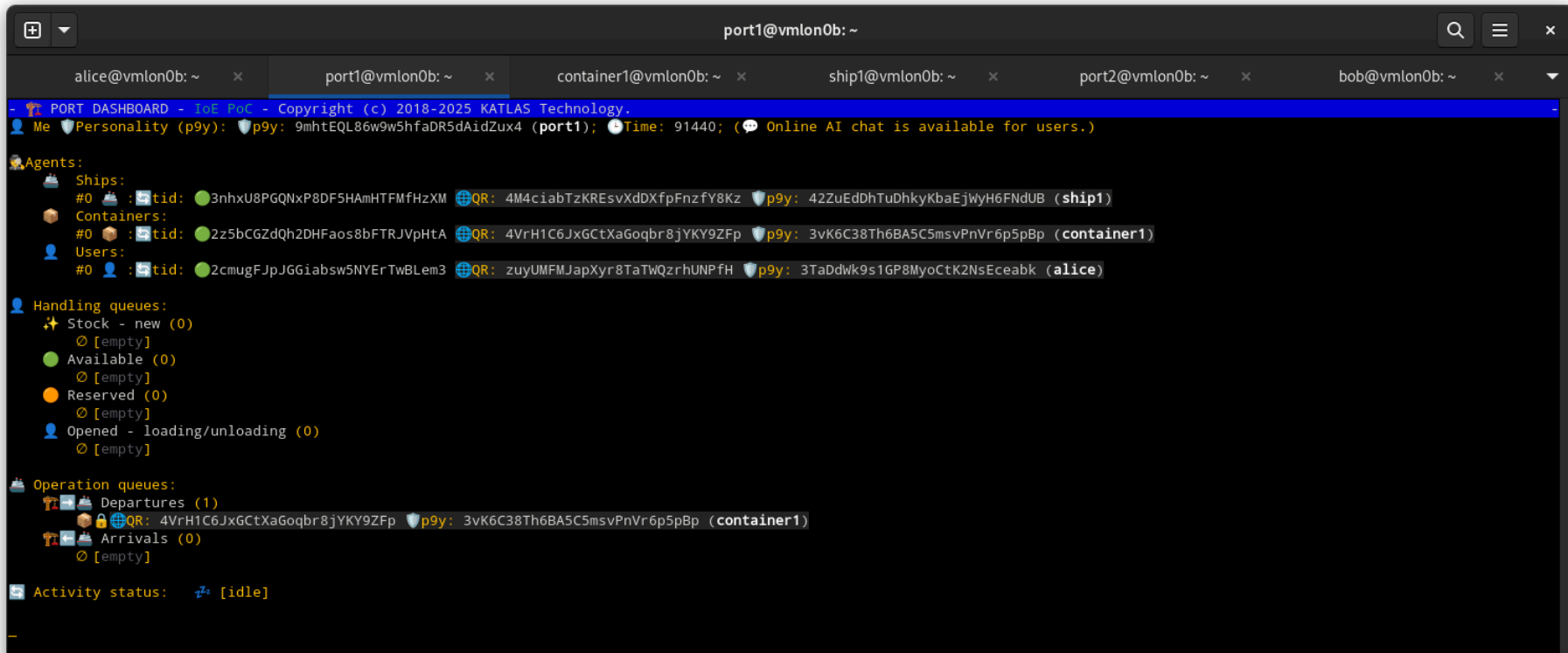
This screenshot shows **Alice's dashboard** immediately after the **container has been sealed and dispatched**.

🟢 **Key Indicators:**

- Both send orders now have status: **container sealed and in transit**
- **Container1** has successfully transitioned from **opened** to **sealed**, (lock icon) indicating no further changes to its contents are possible.
- The **payload count** is now **2**, confirming both orders were packed into the same shipment.
- The **Container Manager** and **Order Manager** are again **idle**, having completed their tasks.

✅ This marks the transition from preparation to execution: the shipment has officially left Alice's hands and is en route to Bob.

👁️👁️5. Port1: container in departures queue



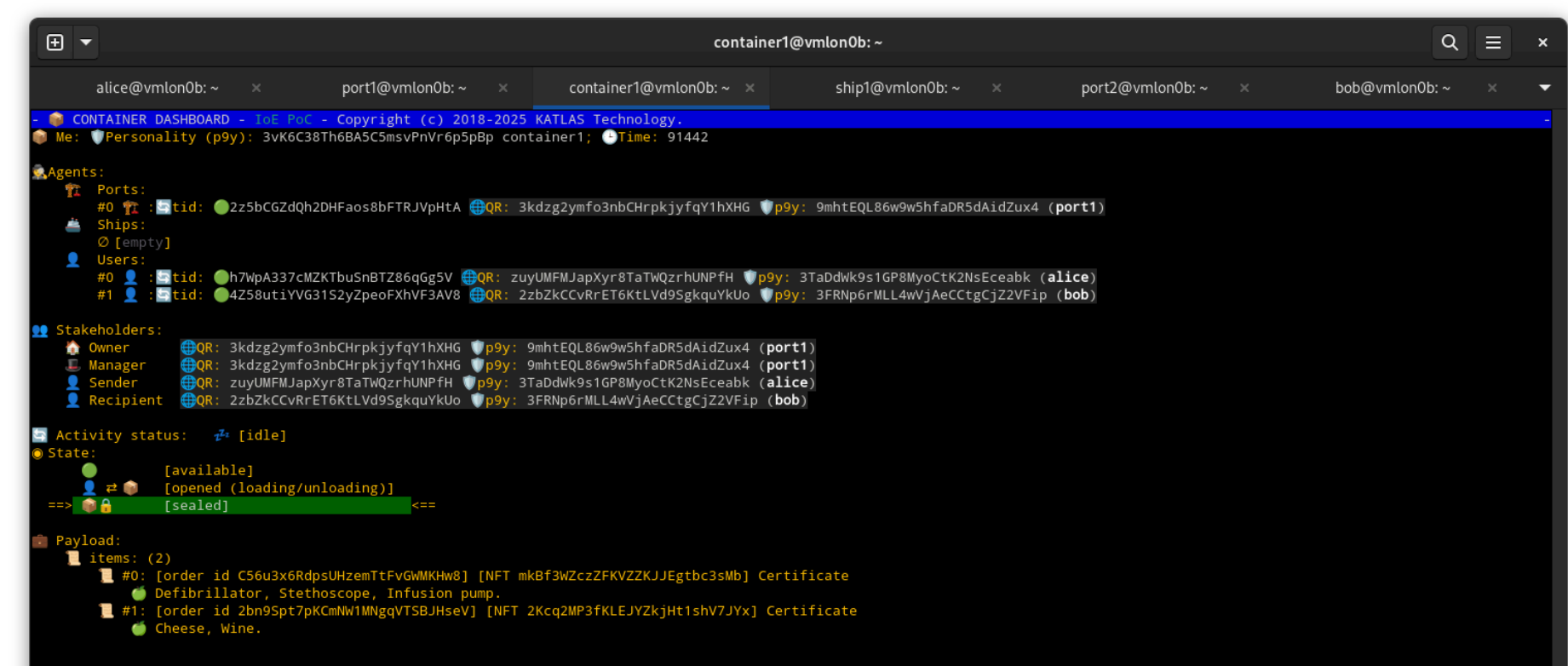
This screenshot shows **Port1's dashboard** immediately after the container has been **sealed and handed over** by Alice.

📦 **Key Observations:**

- **container1** has been **moved from the handling queues** to the **Departures queue**, ready for pickup by a ship.
- The container now appears in the **Operation queues → Departures (1)** section, indicating it is sealed and awaiting transport.
- Alice is listed under "Users," reflecting her involvement in dispatch.
- The port's **activity status** is **idle**, as it awaits ship engagement to continue the journey.

✅ This confirms that Port1 has completed its job preparing the shipment and is now standing by for a ship to collect the outbound container.

6. Container1: sealed. With Payload and recipient information.



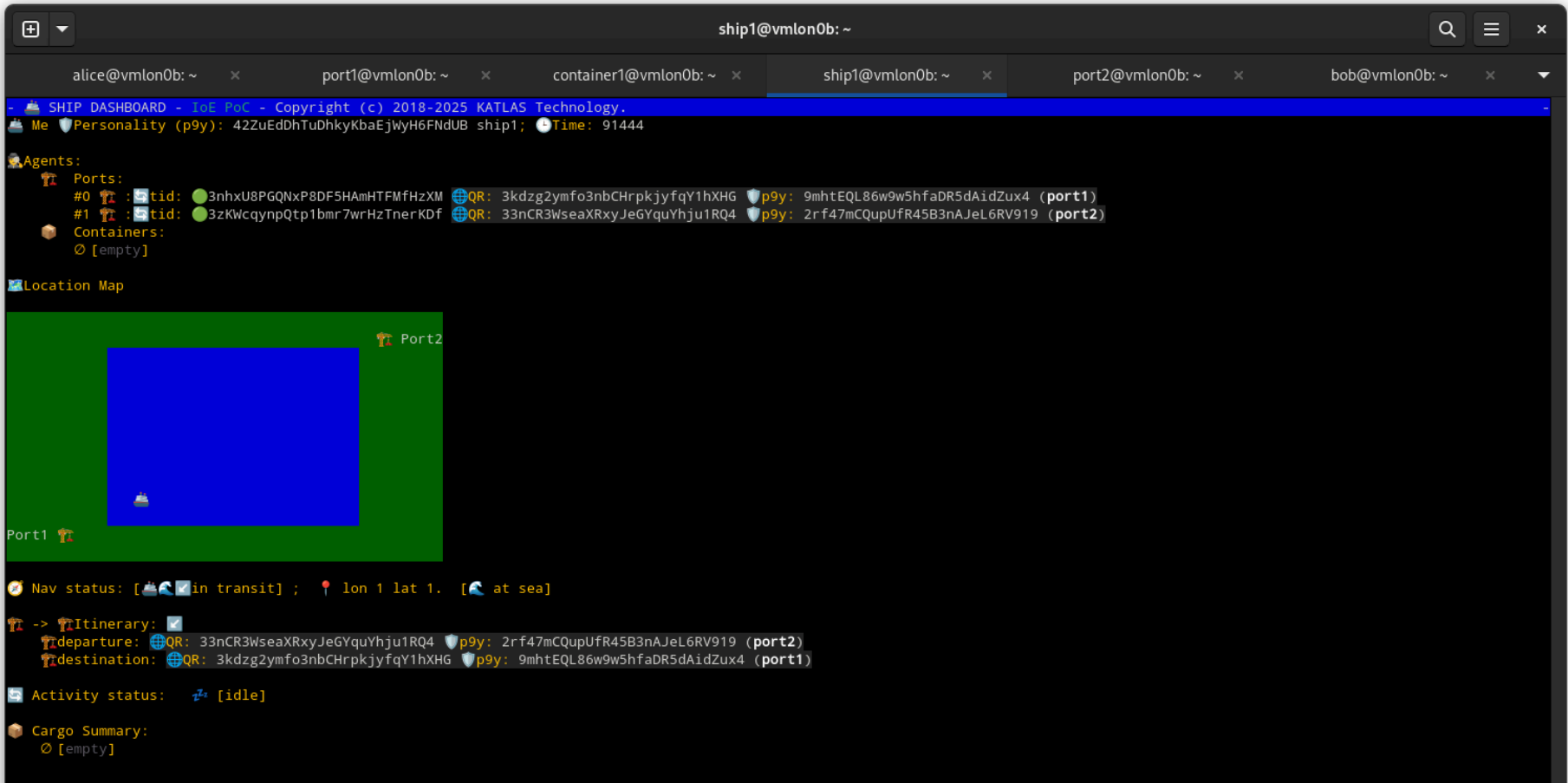
This screenshot captures **Container1’s dashboard** immediately after it has been **sealed** and is ready for transit.

 Key Highlights:

- **State:** The container is now in the **[sealed]** state, meaning it can no longer accept new orders and is locked for transport.
- **Stakeholders** are fully populated:
 - **Owner** and **Manager:** **port1**
 - **Sender:** **alice**
 - **Recipient:** **bob** — now visible because the container is sealed.
- **Payload:** The payload section lists both orders:
 - Order 0: Defibrillator, Stethoscope, Infusion pump
 - Order 1: Cheese, Wine
- **Activity status** is **idle**, indicating no new events until handoff.

✓ This confirms that the container has transitioned into a final, immutable state, with full metadata and cryptographic certificates attached. Ready for pickup by the ship.

7. Ship1: arriving at port1



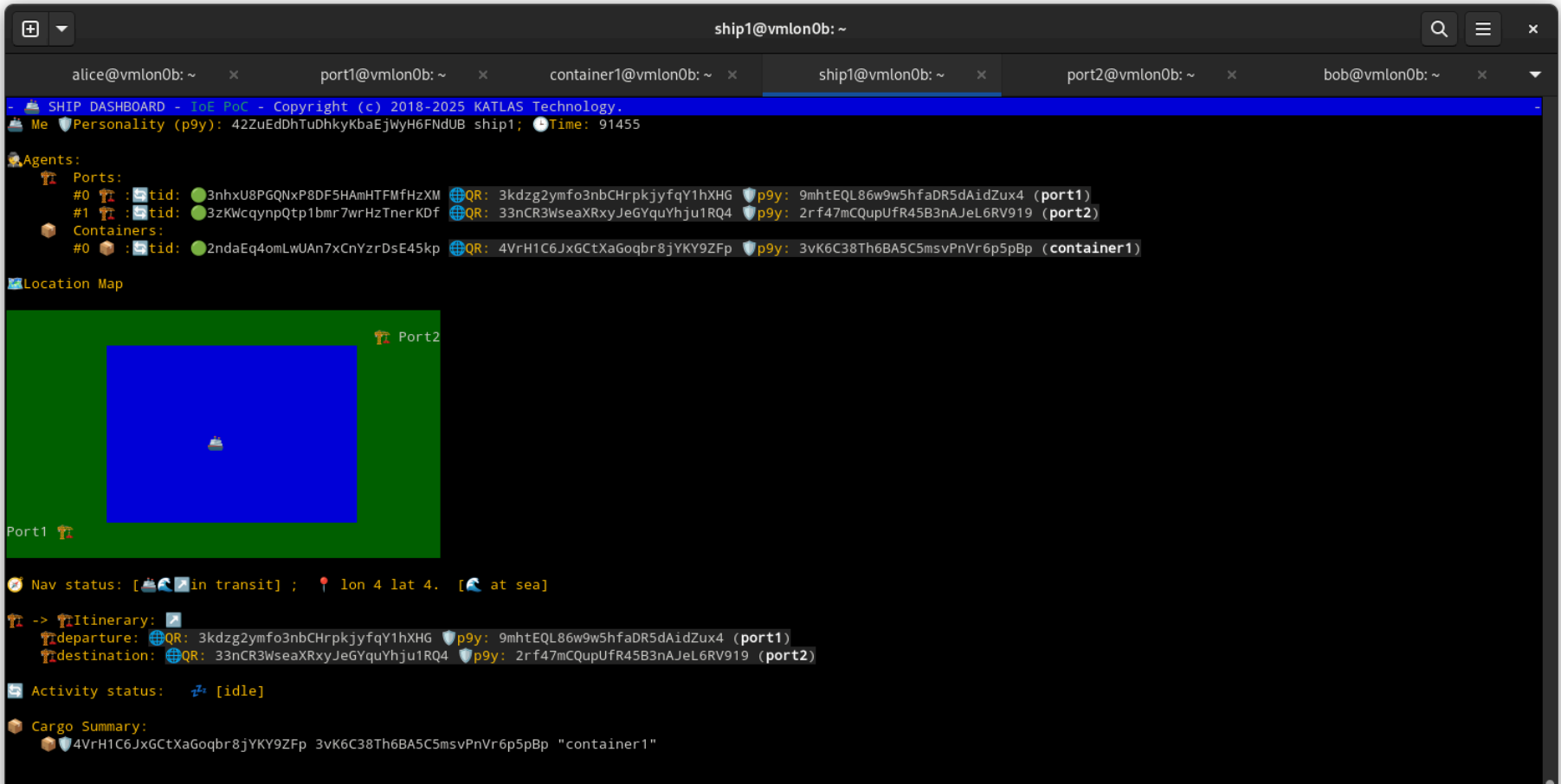
This screenshot shows **Ship1's dashboard** as it approaches **Port1** to collect sealed containers.

 Key Observations:

- **Location Map:** Ship1 is visually approaching **Port1** on the map.
- **Navigation Status:** **in transit**, currently at coordinates **lon 1 lat 1**, confirming it's **at sea**, en route to port.
- **Itinerary:**
 - **Departure:** Port2
 - **Destination:** Port1 — the port where it is about to dock and collect container1.
- **Agents:** Two trades are listed with both ports.
- **Cargo Summary:** Still **empty**, as pickup has not yet occurred.

 This is the precise moment before container loading takes place — Ship1 is on schedule and will soon add container1 to its manifest.

👁️8. Ship1: sailing towards port2. Missed capture of container to see it managed by ship1.



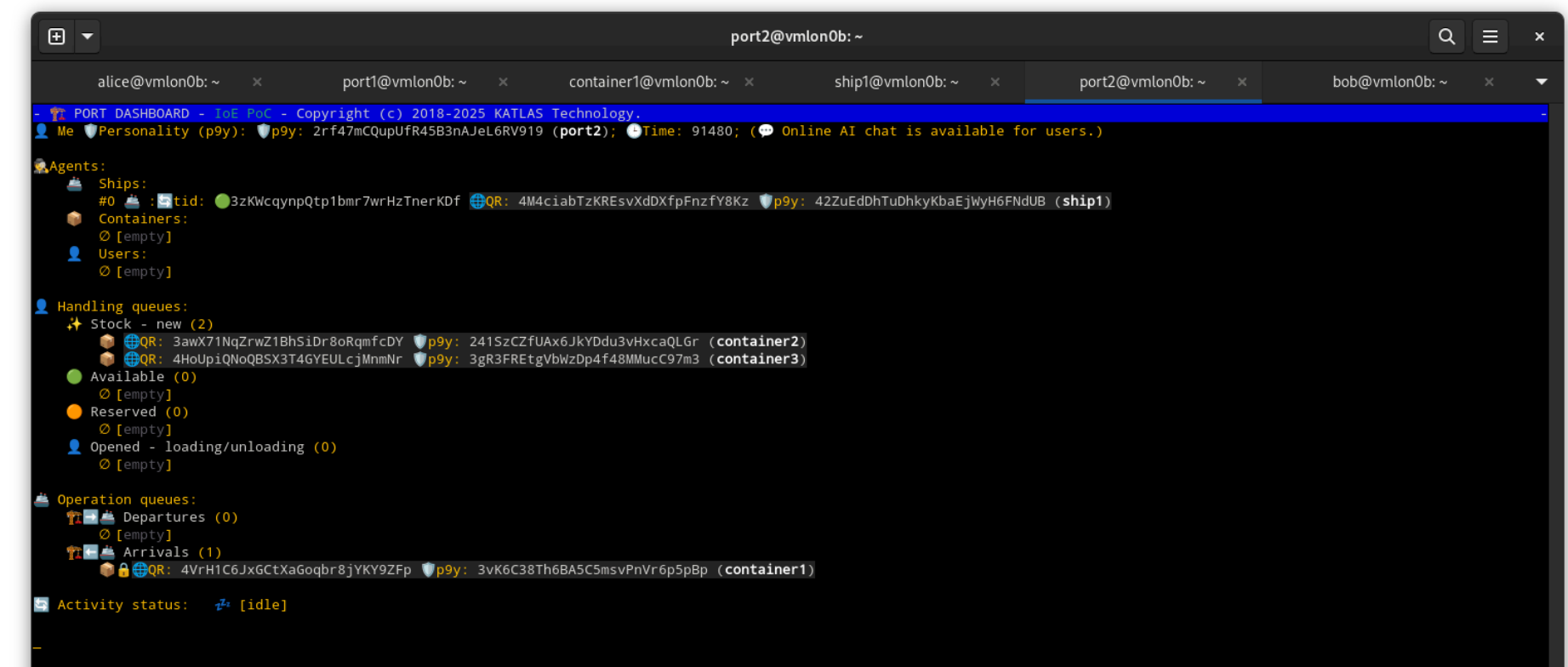
This screenshot shows **Ship1's dashboard** during its voyage **from Port1 to Port2**, now carrying the sealed container.

🌊 Key Details:

- **Navigation Status:** **in transit**, currently at **lon 1 lat 4**, still at sea en route to **Port2**.
- **Cargo Summary:** **container1** is now listed as onboard — this confirms the pickup occurred at Port1 (even though the moment of acquisition wasn't captured).
- **Itinerary:**
 - **Departure:** Port1 (the pickup port)
 - **Destination:** Port2 (Bob's nearest port)

🧭 The ship is now fulfilling its delivery route, and the presence of the container in the cargo confirms that handoff from the port has been successful.

9. Port2: Container in arrivals queue



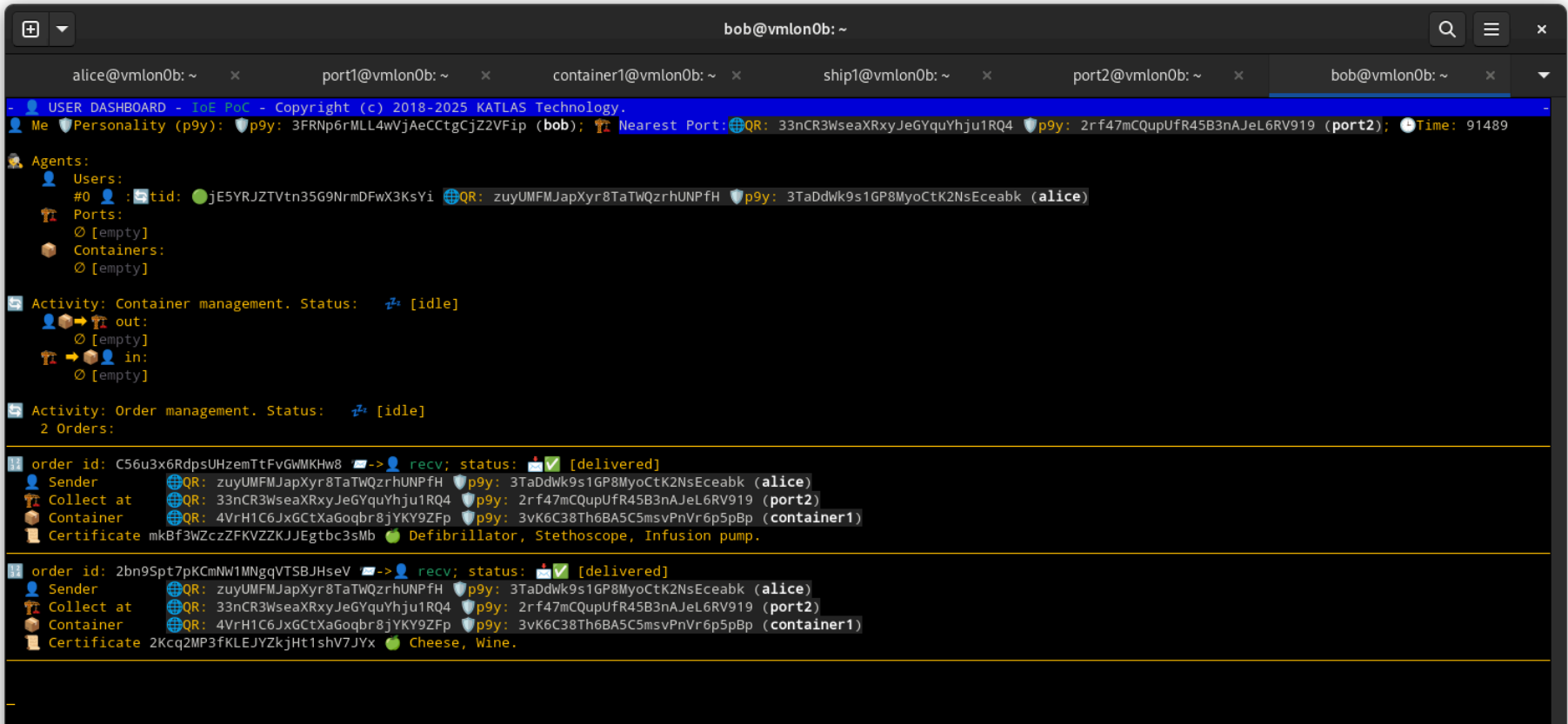
This screenshot shows **Port2's dashboard** after receiving **container1**, now placed in the **Arrivals queue**.

 Key Observations:

- **container1** appears in **Operation queues** → **Arrivals (1)**, indicating it has been successfully unloaded by Ship1.
- The **Handling queues** show Port2's own containers (container2, container3) in stock and available, but **container1 is not yet integrated**, as it's pending user interaction.
- **Ship1** is visible in the "Ships" section, confirming its docking at Port2.
- **Activity status** is **idle**, showing the port is standing by for the next action (likely user retrieval or container processing).

✅ This is the arrival handoff moment, where the container has transitioned from **in transit** to **awaiting collection** at the destination port.

👁️10. Bob: missed capture while unsealing, finally delivered and goods in Bob’s possession :) 🚀



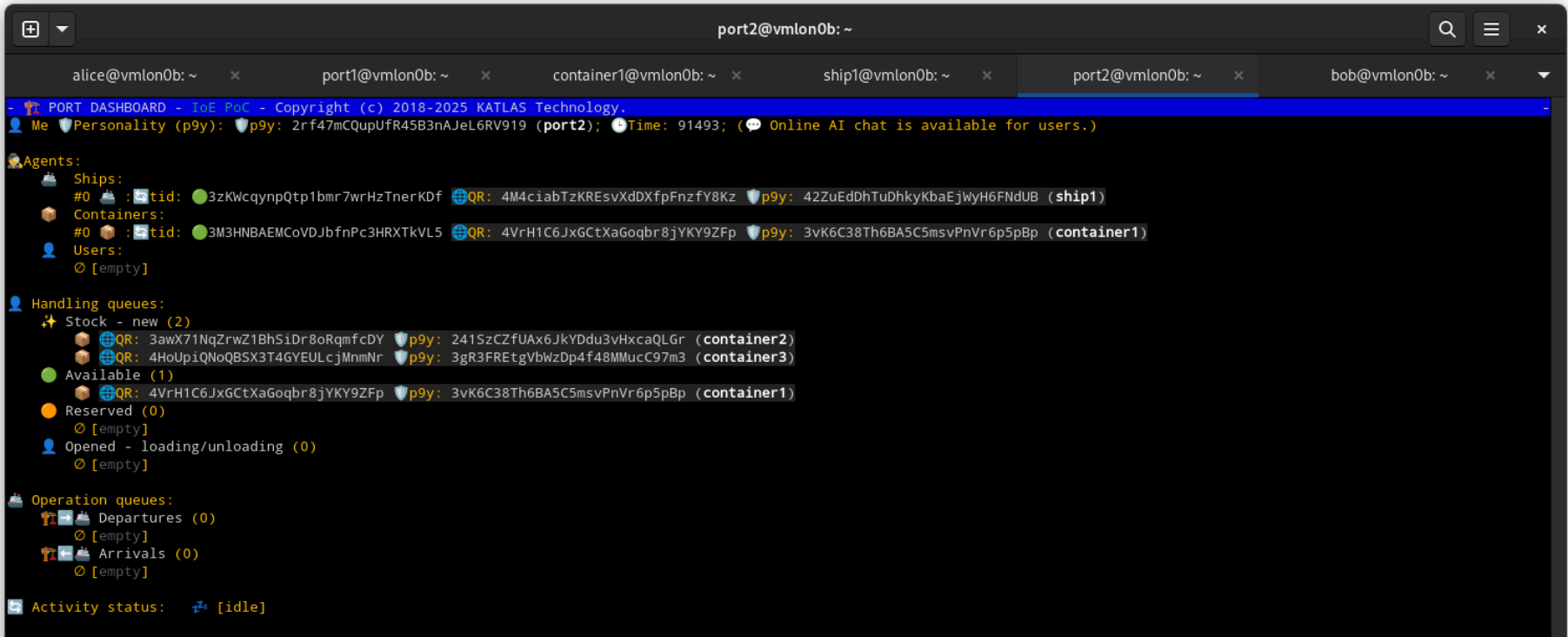
This screenshot shows **Bob’s dashboard** at the final stage of the delivery process—all **goods successfully received** and orders marked as complete.

📦 Key Takeaways:

- Both **receive orders** are marked with status: **[delivered]**, confirming successful unsealing and transfer of goods from **container1**.
- Order Manager** is **idle**, indicating all workflows have concluded.
- The **container manager queues** are empty—no more inbound containers remain to be processed.
- Despite missing the exact unsealing moment, we see the final state: **Bob is now in possession of both payloads**:
 - Order 1: Defibrillator, Stethoscope, Infusion pump
 - Order 2: Cheese, Wine

🎉 Mission accomplished: goods traveled from Alice to Bob via two ports, one ship, and a sealed smart container — all autonomously coordinated.

👁️👁️11. Port2: container in available queue after bob finished unloading the container and released it.



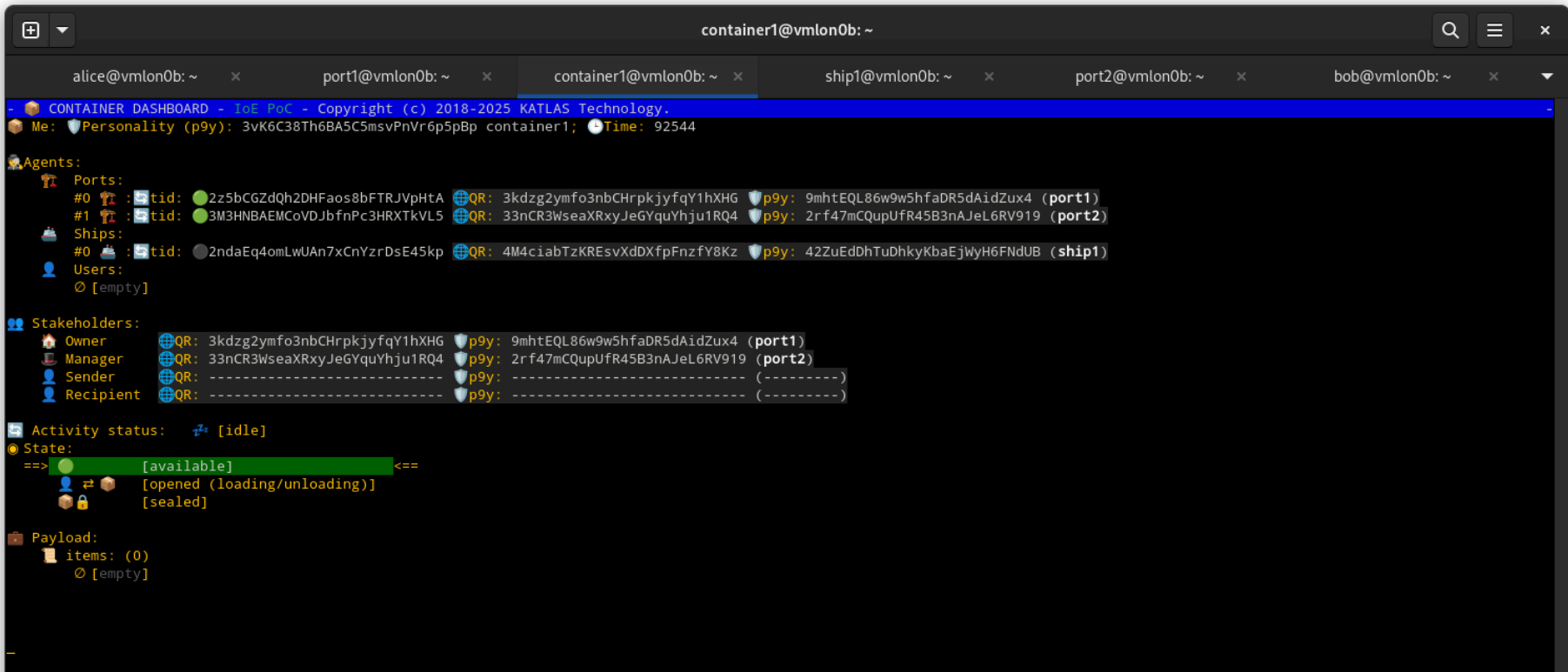
This screenshot shows **Port2's dashboard** after **Bob** has finished unloading **container1**, which is now **returned and available** for reuse.

✓ **Key Details:**

- **container1** has transitioned into the **Available queue**, ready for reassignment.
- It previously appeared in **Arrivals**, then was unsealed and unloaded by Bob, completing its delivery cycle.
- The **Handling Queues** reflect:
 - 2 containers in stock (new and unused)
 - 1 container (container1) now idle but functional
- No containers are in **Reserved** or **Opened** states — indicating a clean handoff.

🔄 This illustrates full cycle reuse: from stock → assigned → sealed → delivered → available again, completing a modular logistics loop.

👁️👁️12. Container1: available status managed by port2



This screenshot shows **Container1's dashboard** after completing its delivery and being returned to the **available** state under the control of **Port2**.

🟢 **Key Observations:**

- **State:** Clearly marked as **[available]**, confirming it is idle and ready for new assignment.
- **Manager** has changed to **Port2**, reflecting the container's current location and operational jurisdiction.
- **Sender** and **Recipient** fields are cleared — the previous mission is complete.
- **Payload** is empty — goods were successfully unloaded.

 This marks the full reset of the container lifecycle. It’s now clean, idle, and managed by its new port — ready for a new shipment chain.

 watch

You can type at any time `user watch` to observe the **live activity of the container**. To exit watch mode, press **CTRL-C**.

 Screencast

You can watch a **screencast recording** showing the container1 activity during a single delivery from Alice to Bob.

		 User dashboard  KATLAS-PoC-SupplyChain-DEMO__Alice https://www.youtube.com/watch?v=9HPup55NuvM&list=PLxosfK0Tg_rWiddsjz4Nad40EJmMgIM5G&index=3
---	---	--

 Scaling Up Workload: Parallel Operations and Throughput

At this point **Bob can initiate a shipment back to Alice**, using the same autonomous workflow—but in reverse. Since **container1 is already available and managed by Port2**, and Bob has Alice in his bookmarks, he can begin sending new orders just as Alice did before. Meanwhile, **Alice remains active and can continue sending orders as well**, meaning the system supports **bi-directional activity**. Both users can operate **independently and simultaneously**, with each wallet managing its own agents, container requests, and shipment negotiations in parallel—demonstrating the flexibility, concurrency, and autonomy of the platform.

R2R API Overview

This section outlines the Role-to-Role (R2R) interaction protocols between ecosystem actors. Each protocol is implemented via dedicated shared libraries and supports secure, asynchronous message exchange over the P2P wallet infrastructure.

Symmetric R2R protocol: User-to-User

Functions Implemented in library `libtrader-u2u-u.so`

<code>agree_send_items <itemsb58> <orderid></code>	Initiates a delivery proposal with the remote user.
<code>delivery_details <delivery_details_blob_b58></code>	Send sender delivery details (destination port, ..).
<code>cancel_order <is-hash b58> <reason></code>	Cancels an order with a given reason.

Asymmetric R2R protocol: User-to-Port

- User side

Functions Implemented in library `libtrader-u2p-u.so`

<code>assign_container</code>	Requests a container from the port.
<code>container_accepted <container endpoint></code>	Confirms acceptance of assigned container and proceeds to load.

- Port side

Functions Implemented in library `libtrader-u2p-p.so`

<code>assigned_container <container_endpoint></code>	Informs the user of the allocated container.
--	--

Asymmetric R2R protocol: User-to-Container

- User side

Functions Implemented in library `libtrader-u2c-u.so`

<code>seal_container <payload></code>	Seals the container with loaded content.
<code>unseal_container</code>	Attempts to open a sealed container.
<code>request_cert <index></code>	Requests a certificate stored in the container (access controlled).

- Container side

Functions Implemented in library `libtrader-u2c-c.so`

<code>report_status <uint8_t></code>	Sends container status to sender/recipient.
<code>confirmed_sealed</code>	Acknowledges successful sealing.
<code>container_is_coming <payload></code>	Notifies recipient that delivery is underway.
<code>container_has_arrived</code>	Notifies recipient that container has reached the port.
<code>confirmed_unsealed</code>	Confirms to sender that delivery is complete.

 Asymmetric R2R protocol: Container-to-Port

 Container side

Functions Implemented in library `libtrader-c2p-c.so`

report_status <uint8_t>	Sends state updates to managing port.
-------------------------	---------------------------------------

 Port side

Functions Implemented in library `libtrader-c2p-p.so`

assign_duty <container_duty>	Assigns container a user and destination.
set_manager <ship_endpoint_b58>	Indicates transfer of container custody to a ship.

 Asymmetric R2R protocol: Port-to-Ship

 Port side

Functions Implemented in library `libtrader-p2s-p.so`

cargo_unloaded	Confirms cargo was unloaded from the ship.
cargo_loaded <cargo_b58>	Confirms cargo was successfully loaded.

 Ship side

Functions Implemented in library `libtrader-p2s-s.so`

unload_cargo <cargo_b58>	Requests port to unload specified cargo.
load_cargo	Requests cargo from port.

 Asymmetric R2R protocol: Container-to-Ship

 Container side

Functions Implemented in library `libtrader-c2s-c.so`

report_status <uint8_t>	Reports current status to managing ship.
-------------------------	--

 Ship side

Functions Implemented in library `libtrader-c2s-s.so`

set_manager <port_endpoint_b58>	INotifies container of new manager after drop-off.
---------------------------------	--



AI integration Exercise - Chat with Port

To demonstrate our ecosystem’s seamless support for AI integration, we’ve augmented the **chat interface** of the **Port** role with a lightweight AI module powered by ChatGPT.

This module is programmed to return a **short rhymed paragraph** that incorporates **keywords from the user's prompt**, showcasing both the system’s extensibility and a touch of playful interaction.



Let’s Try It

From Alice’s wallet, first retrieve an active **trade id (tid)** by typing **user st**

```
Agents:
Users:
#0 : tid: 2GVoD9Ks2ns4FTSKe8L3izqZYXxh QR: 2zbZkCCvRrET6KtLVd9SgkquYkUo p9y: 3FRNp6rMLL4wVjAeCCtgCjZ2VFip (bob)
Ports:
#0 : tid: KpgBfKbstZzCJwtCRwXzNpCNoSC QR: 3kdzg2ymfo3nbCHrpkyfqY1hXHG p9y: 9mhtEQL86w9w5hfaDR5dAidZux4 (port1)
Containers:
[empty]
```

Look for an open connection with **port1** and **copy the tid** — e.g.: KpgBfKbstZzCJwtCRwXzNpCNoSC

Send a chat message to **port1** using:

```
wallet -wr 1 trade KpgBfKbstZzCJwtCRwXzNpCNoSC msg How many containers have you available?
```

The **-wr 1** flag tells the wallet to **wait for one push response** (in this case, the chat reply) before exiting.

Almost instantly, you’ll receive a response like:

```
Tue Jun 10 2025 10:02:50 me:
How many containers have you available?

Tue Jun 10 2025 10:02:51 peer:
How many containers do you have available? In the kitchen, they're all stacked and stable.
```

🎨 *Adorably poetic!* — and a fun proof that AI interaction is tightly coupled with our R2R messaging fabric.



Useful Commands

To view the full chat history of a trade:

```
wallet trade KpgBfKbstZzCJwtCRwXzNpCNoSC show chat
```

To view available trade commands:

```
wallet trade KpgBfKbstZzCJwtCRwXzNpCNoSC
```

+ 👤 New actors

A scalable-by-design ecosystem must support seamless growth—and a key requirement is the **ability to integrate new actors easily and reliably**. In our architecture, this is not only possible—it’s automated.

Our approach to producing complex software is grounded in **automation and encapsulation**. The ecosystem is built from **specification input files** that define roles, actors, and their behavioral contracts. From these inputs, we generate:

- Derived artifacts
- R2R (Role-to-Role) protocol code templates
- API stubs
- Configuration files and connection metadata

These outputs integrate directly with the **core codebase**, freeing developers to focus on the **model’s core logic**—the part that demands craftsmanship and domain expertise.

We’ve extended this philosophy to our **DevOps pipeline** as well, enabling controlled, repeatable **continuous integration and deployment**. Even this live deployment reflects that principle.

🧩 Adding new actors is reduced to a single function call, requiring only minimal specification.

To demonstrate this in practice, we’ll now create a **new user actor: Eve**—showcasing just how effortless actor onboarding is in this ecosystem.

🐧 Adding a New Actor: Eve. Linux account, wallet daemon, p9y

To demonstrate the **scalability and extensibility** of our actor model, we introduce a new user: **eve**. By appending a new line to the **actors** spec file:

```
root@vmlon0b:/etc/katlas# cat actors
#linuxuser  #walletd port #role
alice       165          u
bob         168          u
container1  170          c
port1       167          p
ship1       169          s
port2       171          p
container2  172          c
container3  173          c
eve         174          u
```

By modifying this file we instruct the automation pipeline to:

- Create a Linux user account (**eve**)
- Enabling SSH access
- Assign a wallet service bound to the **174XX** port range
- Enable user role (**u**) plugins
- Generate a new digital personality (**p9y**) uniquely identifying Eve in the system

Running **bin/all linux_users** parses this spec and automatically bootstraps the necessary environment:

```
root@vmlon0b:/etc/katlas# bin/all linux_users
/home/alice already exists. skipping.
/home/bob already exists. skipping.
/home/container1 already exists. skipping.
/home/port1 already exists. skipping.
/home/ship1 already exists. skipping.
/home/port2 already exists. skipping.
/home/container2 already exists. skipping.
/home/container3 already exists. skipping.
adding linux account for actor eve
actor eve role u wallet_tcp_port 17473
gen personality
producing file personality_key
2bbtfZZYXUbRpoCnKWJ9jutoN9FT eve
allow demo restart
```

This automation confirms our infrastructure is **ready to scale**, requiring only declarative specs to onboard new actors, making system growth operationally effortless.

Bookmarks: Establishing Initial Trust and Visibility

Bookmarks act as each actor’s **address book**, defining which peers they can discover and interact with upon system startup. These mappings are declared in the `actors_bookmarks` file and regenerated across wallets using the `bin/all write_bookmarks` automation.

In Eve’s case, she is intentionally configured to **know all user, port, and container actors** (except `ship1`, since there's no user-to-ship protocol), simulating a surveillance or adversarial role. Her bookmark list grants visibility into the wider ecosystem, showcasing how trust domains can be flexibly preconfigured. This also illustrates the system's ability to **model asymmetric knowledge**, where actors may or may not be aware of one another by design.

In the `actors_bookmarks` file, each line defines the **initial visibility scope** for a particular actor in the ecosystem:

Let's add eve to this PoC master bookmarks file:

```
root@vmlon0b:/etc/katlas# cat actors_bookmarks
#actor $knows-actor-list
port1 container1
alice bob port1
bob alice port2
ship1 port1 port2
port2 container2 container3
eve alice bob port1 port2 container1 container2 container3
```

The **first column** (e.g. `eve`) refers to the *actor* whose wallet will be configured.

The **subsequent columns** (`bob`, `port1`, ...) list the *other actors* that this actor knows at startup—effectively forming that actor’s **bookmark list**.

This line means that **Eve’s** wallet will start with bookmarks for *all* other users, ports, and containers—allowing her to interact with any of them directly from the beginning. This setup is particularly useful to model a privileged or adversarial actor with broad visibility.

This format provides a simple yet powerful way to define **trust topologies** and **peer awareness** across all roles in the system.

After updating the `actors_bookmarks` file to include Eve's relationships, running the `bin/all write_bookmarks` command regenerates each actor’s local bookmark list accordingly.

```
root@vmlon0b:/etc/katlas# bin/all write_bookmarks
.....
====port1=====
/home/port1/.katlas/wallet/trader/bookmarks:
1: 'container1': qr: "4VrH1C6JxGctXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1" c2p c " label: "c (c2p)" icon: [0 bytes]

====alice=====
/home/alice/.katlas/wallet/trader/bookmarks:
1: 'bob': qr: "2zbZkCCvRrET6KtLVd9SgkquYkUo 3FRNp6rMLL4wVjAeCCtgCjZ2VFip "bob" u2u u " label: "" icon: [0 bytes]
2: 'port1': qr: "3kdzg2ymfo3nbCHrpkyfY1hXHG 9mhtEQL86w9w5hfaDR5dAidZux4 "port1" u2p p " label: "p (u2p)" icon: [0 bytes]

====bob=====
/home/bob/.katlas/wallet/trader/bookmarks:
1: 'alice': qr: "zuyUMFMJapXyr8TaTWQzrhUNPfh 3TaDdWk9s1GP8MyoCtK2NsEceabk "alice" u2u u " label: "" icon: [0 bytes]
2: 'port2': qr: "33nCR3WseaXRxyJeGYquYhju1RQ4 2rf47mCQupUfr45B3nAJeL6RV919 "port2" u2p p " label: "p (u2p)" icon: [0 bytes]

====ship1=====
/home/ship1/.katlas/wallet/trader/bookmarks:
1: 'port1': qr: "3kdzg2ymfo3nbCHrpkyfY1hXHG 9mhtEQL86w9w5hfaDR5dAidZux4 "port1" p2s p " label: "p (p2s)" icon: [0 bytes]
2: 'port2': qr: "33nCR3WseaXRxyJeGYquYhju1RQ4 2rf47mCQupUfr45B3nAJeL6RV919 "port2" p2s p " label: "p (p2s)" icon: [0 bytes]

====port2=====
/home/port2/.katlas/wallet/trader/bookmarks:
1: 'container2': qr: "3awX71NqZrwZ1BhSiDr8oRqmfcDY 241SzCZfUAx6JkYDdu3vHxcaQLGr "container2" c2p c " label: "c (c2p)" icon: [0 bytes]
2: 'container3': qr: "4HoUpiQNoQBSX3T4GYEULcjMnmNr 3gR3FREtgVbWzDp4f48MMucC97m3 "container3" c2p c " label: "c (c2p)" icon: [0 bytes]

====eve=====
/home/eve/.katlas/wallet/trader/bookmarks:
1: 'alice': qr: "zuyUMFMJapXyr8TaTWQzrhUNPfh 3TaDdWk9s1GP8MyoCtK2NsEceabk "alice" u2u u " label: "" icon: [0 bytes]
2: 'bob': qr: "2zbZkCCvRrET6KtLVd9SgkquYkUo 3FRNp6rMLL4wVjAeCCtgCjZ2VFip "bob" u2u u " label: "" icon: [0 bytes]
3: 'container1': qr: "4VrH1C6JxGctXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1" u2c c " label: "c (u2c)" icon: [0 bytes]
4: 'container2': qr: "3awX71NqZrwZ1BhSiDr8oRqmfcDY 241SzCZfUAx6JkYDdu3vHxcaQLGr "container2" u2c c " label: "c (u2c)" icon: [0 bytes]
5: 'container3': qr: "4HoUpiQNoQBSX3T4GYEULcjMnmNr 3gR3FREtgVbWzDp4f48MMucC97m3 "container3" u2c c " label: "c (u2c)" icon: [0 bytes]
6: 'port1': qr: "3kdzg2ymfo3nbCHrpkyfY1hXHG 9mhtEQL86w9w5hfaDR5dAidZux4 "port1" u2p p " label: "p (u2p)" icon: [0 bytes]
7: 'port2': qr: "33nCR3WseaXRxyJeGYquYhju1RQ4 2rf47mCQupUfr45B3nAJeL6RV919 "port2" u2p p " label: "p (u2p)" icon: [0 bytes]
```

This process results in a `.katlas/wallet/trader/bookmarks` file under each actor’s home directory, pre-populated with QR codes and metadata for their known peers. The output you see confirms:

- Each actor now has up-to-date visibility on others, based on the configuration.
- For **Eve**, seven bookmarks were generated, granting her broad access to other users, ports, and containers—excluding `ship1`, as no direct R2R protocol exists between users and ships.

This automatic regeneration ensures that every actor begins the session with a correct and consistent view of the ecosystem, simplifying bootstrapping and simulating realistic knowledge topologies.

While the initial set of bookmarks is preconfigured through the `actors_bookmarks` file and deployed at startup, **bookmarks are not static**. The wallet daemon (`walletd`) is designed to dynamically **update each actor's bookmarks in real time** as they **discover or interact with new actors** during the course of operation.

This means that even if two actors were not initially aware of each other, a successful protocol exchange (e.g. `user-to-user`, `user-to-port`, or `user-to-container`) can result in the **automatic addition of the counterpart's contact details** to the initiating actor's bookmarks. This ensures the network of known peers evolves naturally over time, reflecting **actual interaction patterns** rather than only static configuration.

Surface attack scenario: Attempted Unauthorized Certificate Access

This exercise demonstrates the security guarantees in place during a container's sealed transit and verifies **proper access control enforcement** over sensitive data like payload certificates.

Attack Simulation Brief

Objective: Validate that **only authorized actors**—the sender and recipient—can access a container's certificates while it's sealed, and that **unauthorized actors** (like Eve) are denied.

Step-by-step Plan

This is the briefing of our plan:

1. Connect to alice's wallet and restart the ecosystem so we start from scratch
2. From Eve's wallet start a trade with container1
3. Send a container to bob

During the time frame between the container is sealed by the Alice and unsealed by Bob...

4. From either Alice's or Bob's wallet successfully obtain a certificate from the container, as they are allowed to see them.
5. Verify Eve is unable to obtain any certificate.

 To run this scenario hands-on, be sure Eve is listed in your terminal client (e.g., PuTTY) bookmarks for quick access and testing.

step 1

To begin the demonstration from a clean slate, connect to **Alice's wallet** and issue the restart command. This ensures that all actors, containers, and trades are reset to their initial state, eliminating any side effects from previous runs.

```
alice@vmlon0b:~$ user restart
...
Processes running:
* blockchain node:
  345458 ?      S<sI  0:23 /usr/local/bin/katlas-gov -ds -e 12 -v 100 -c 0 -p 16672 -pp 16672 -status 0 -E 0
* wallets:
  351480 ?      SI      0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 16573 -pp 16573 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  351558 ?      SI      0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 16873 -pp 16873 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  351636 ?      SI      0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 17073 -pp 17073 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  351713 ?      SI      0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 16773 -pp 16773 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  351788 ?      SI      0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 16973 -pp 16973 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  351863 ?      SI      0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 17173 -pp 17173 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  351947 ?      SI      0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 17273 -pp 17273 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  352024 ?      SI      0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 17373 -pp 17373 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
  352101 ?      SI      0:00 /usr/local/bin/katlas-wallet -d -c 0 -lp 17473 -pp 17473 -bp 16672 -e 20 -v 6 -bhost 34.39.80.254 -whost 34.39.80.254
```

Attack warm up

To simulate the early stages of an attack, connect to **Eve's wallet** and initiate a trade with `container1`. This mimics an unauthorized actor attempting to engage with the container—an action that is **technically possible** but **does not grant access to protected data**.

inspect our bookmarks

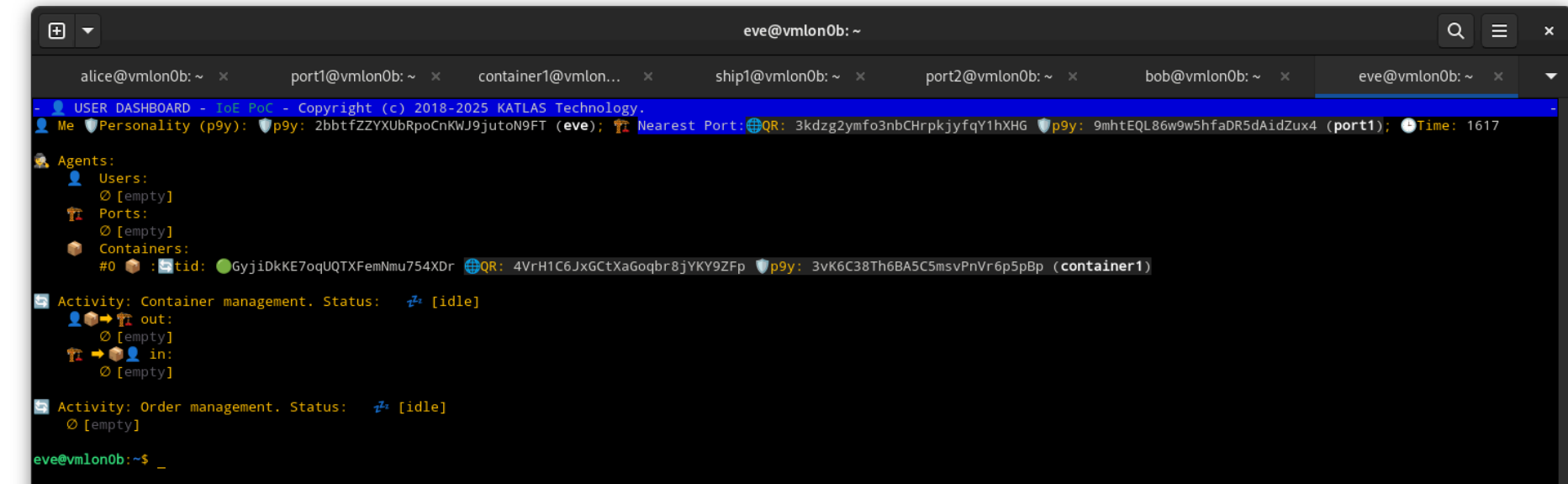
```
eve@vmlon0b:~$ wallet trade bookmarks
1: 'alice': qr: "zuyUMFMJapXyr8TaTWQzrhUNPfH 3TaDdWk9s1GP8MyoCtK2NsEceabk "alice" u2u u " label: "" icon: [0 bytes]
2: 'bob': qr: "2zbZkCCvRrET6KtLVd9SgkquYkUo 3FRNp6rMLL4wVjAeCCtgCjZ2VFip "bob" u2u u " label: "" icon: [0 bytes]
3: 'container1': qr: "4VrH1C6JxGctXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1" u2c c " label: "c (u2c)" icon: [0 bytes]
4: 'container2': qr: "3awX71NqZrwZ1BhSiDr8oRqmFcDY 241SzcZfUAx6JkYDdu3vHxcaQLGr "container2" u2c c " label: "c (u2c)" icon: [0 bytes]
5: 'container3': qr: "4HoUpiQNoQBSX3T4GYEULcjMnmNr 3gR3FREtgVbWzDp4f48MMucC97m3 "container3" u2c c " label: "c (u2c)" icon: [0 bytes]
```

```
6: 'port1': qr: "3kdzg2ymfo3nbCHrpkyfY1hXHG 9mhtEQL86w9w5hfaDR5dAidZux4 "port1" u2p p " label: "p (u2p)" icon: [0 bytes]
7: 'port2': qr: "33nCR3WseaXRxyJeGYquYhju1RQ4 2rf47mCQupUfR45B3nAJeL6RV919 "port2" u2p p " label: "p (u2p)" icon: [0 bytes]
```

We are interested in the entry at index **3**, key **‘container1’**
start a new trade using QR at bookmark index 3

```
eve@vmlon0b:~$ wallet trade start_bm container1
New trade. GyjiDkKE7oqUQTxFemNmu754XDr
GyjiDkKE7oqUQTxFemNmu754XDr
Data:
  'activity': 1749660966263265665
  'bootstrapped_by': 3f5aHDKq97H5xHp6BD6PMEZRHZxt
  'created': 1749660966257997385
  'data_generated': 1749660966263308175
  'datasubdir': GyjiDkKE7oqUQTxFemNmu754XDr
  'initiator': Y
  'lang': en
  'local_endpoint': 3f5aHDKq97H5xHp6BD6PMEZRHZxt  anonymous
  'logfile': /home/eve/.katlas/wallet/trader/log/trade_1749660966257997385_GyjiDkKE7oqUQTxFemNmu754XDr
  'my_moniker': eve
  'my_personality': 2bbtfZZYXUbRpoCnKWJ9jutoN9FT
  'parent_tid': GyjiDkKE7oqUQTxFemNmu754XDr
  'peer_moniker': anonymous
  'peer_personality': 11111111111111111111
  'peer_protocols': 0
  'protocol': not set
  'qrs': N
  'remote_endpoint': 4VrH1C6JxGCtXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1"
  'remote_ip4': 34.39.80.254:17073
  'roles': N
  'state': connected
```

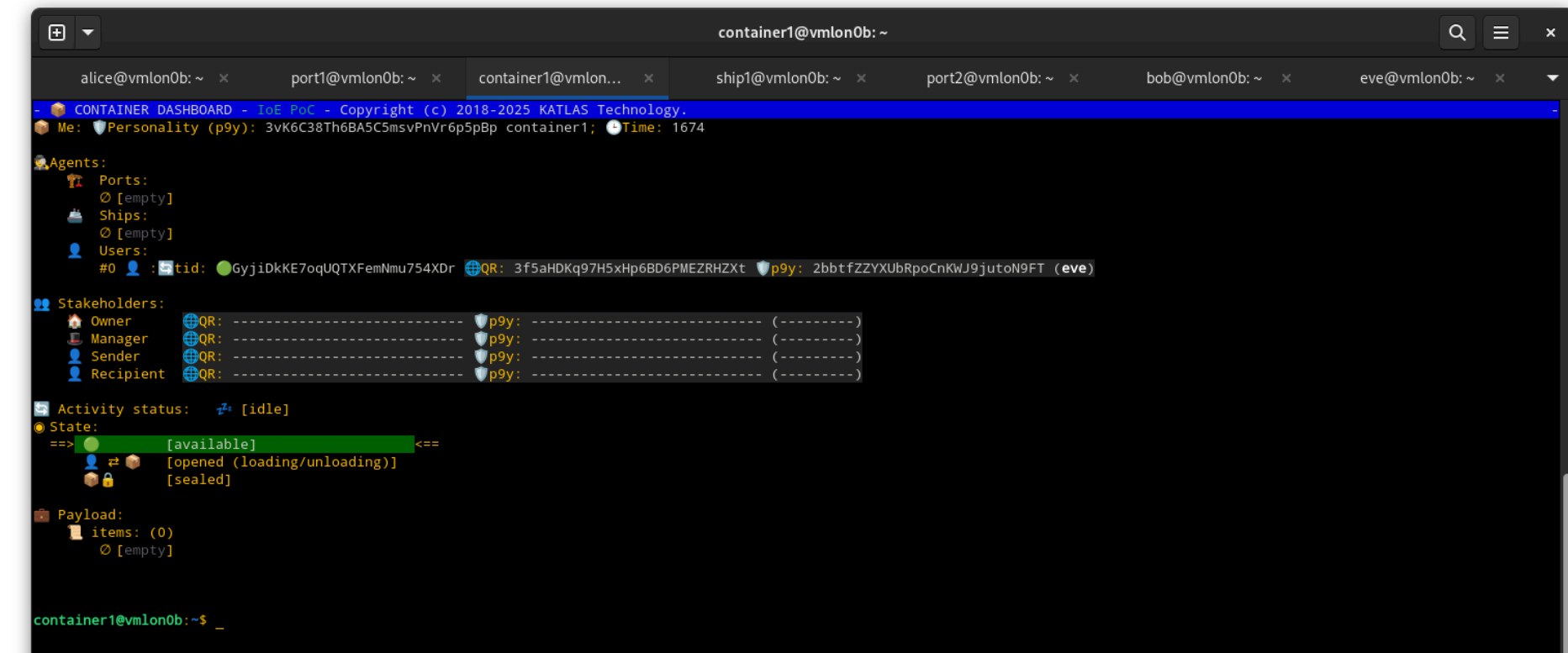
With the given trade ID **GyjiDkKE7oqUQTxFemNmu754XDr** now established between **Eve** and **container1**, we can verify the session is active by inspecting the dashboard:



Confirmed:

```
Containers:
#0 : tid: GyjiDkKE7oqUQTxFemNmu754XDr QR: 4VrH1C6JxGCtXaGoqbr8jYKY9ZFp p9y: 3vK6C38Th6BA5C5msvPnVr6p5pBp (container1)
```

In the same manner, inspecting container1’s dashboard confirms it has registered Eve as a peer:



This verification aligns, as the same trade ID (tid) appears on both ends:

```
👤 Users:
#0 👤 : 📄tid: 🟢GyjiDkKE7oqUQTXFemNmu754XDr 🌐QR: 3f5aHDKq97H5xHp6BD6PMEZRHZxt 💎p9y: 2bbtfZZYXUbRpoCnKWJ9jutoN9FT (eve)
```

From Eve’s wallet, you can display the list of available APIs by invoking the help screen with:

```
eve@vmlon0b:~$ wallet trade GyjiDkKE7oqUQTXFemNmu754XDr
Trade id: GyjiDkKE7oqUQTXFemNmu754XDr
Personality: 2bbtfZZYXUbRpoCnKWJ9jutoN9FT eve
Trader commands:
  disconnect      Go offline.
  ping            Ping remote trader..
  request         Request remote object.
  " roles        Request roles they want you to play.
  " qrs          Request remote shared r2r QRs..
  " rf           Request remote functions list.
  show           Display object.
  " roles        Roles I can initiate.
  " roles mine   Roles I can play, by protocol.
  " roles peer   Roles peer can play.
  " qrs mine     My shared r2r endpoints. QRs..
  " qrs peer     Their shared r2r endpoints. QRs..
  " data         Available data.
  " log          Log content.
  " chat         Dialogue history.
  " bookmarks    Endpoints worth to remember..
  makebm <local_name> <ico_file|-> <label> Create a bookmark with this endpoint..
  copybm <#ordinal> Copy bookmark from peer. (see 'qrs peer' command)..
  change personality <key> <moniker> Change my personality..eve
  change moniker <moniker> Change moniker of current personality..eve
  send personality [mute] Send over personality proof [without feedback]..eve
  send cert <nft> Send cert (aka nft) using its id..
  start <protocol> <role> Start protocol playing the specified role..
  reset          Send RESET signal.
  reload         Send RELOAD signal.
  kill           Archive this trade.
  msg <text>      Send message to peer [chat].
  trade         Go back to trading shell.
  wallet        Go back to wallet shell.

R2R Protocol: u2c. Role: u
  Home: /home/eve/.katlas/wallet/trader/u2c/u/2bbtfZZYXUbRpoCnKWJ9jutoN9FT/3vK6C38Th6BA5C5msvPnVr6p5pBp
Protocol offline commands:
  show           Display object.
  " params       Show params..
  " ico          peer's icon.
  " logo         peer's logo.
  show redirects Show redirects suggested by peer..
  copyredirect <#ordinal> Copy bookmark from redirects. (see 'show redirects' command)..
  followredirect <#ordinal> Create a child trade on the same working dir. (see 'show redirects' command)..
Protocol Online commands:
  end            Finishes current protocol.
  request logo   Request peer's logo.
  request ico    Request peer's icon.
  send <file>     Send over a file.
  send shared_params [mute] Send shared parameters. [without feedback].
  seal_container <payload> Seals the container loaded with the given payload.
  unseal_container Opens a sealed container.
  request_cert <index> Requests cert held in the container.
```

To begin exploring potential unauthorized use of container APIs, we’ll attempt to call the `seal_container` function from Eve's wallet without providing any arguments. This simulates a malicious attempt to force the container into a sealed state without legitimate payload data.

Since `seal_container` expects a `<payload>` argument, omitting it should either produce an error or trigger input prompts depending on the container's implementation and input validation.

This step helps:

- Confirm that Eve has technical access to the interface.
- Reveal how the system handles malformed or incomplete invocations.
- Test whether unauthorized sealing attempts are rejected based on trade context and roles.

⚠️ If the system is correctly secured, the call should **fail gracefully**, refusing to proceed because Eve is not the legitimate sender of the container.

Run the following from Eve’s wallet terminal:

```
eve@vmlon0b:~$ wallet trade GyjiDkKE7oqUQTXFemNmu754XDr seal_container
KO 60292 Invalid payload
```

As expected, Eve's attempt to invoke `seal_container` without providing a valid payload results in a rejection: This confirms that the container enforces input validation at the protocol level. While Eve has established a connection and can technically invoke the API, the system correctly prevents misuse by rejecting malformed or unauthorized calls. This is a fundamental safeguard in the container’s R2R implementation.

🔥 Attempting to Seal with a Crafted Payload

In this step, we are simulating an attack scenario in which Eve, an unauthorized actor, attempts to invoke the `seal_container` function on a container using a **crafted payload**. This payload is a block of syntactically valid data — a base-58 encoded binary blob that mimics a certificate — but Eve has no authority to perform sealing.

In a **production-grade system**, sealing a container would typically involve verifying not only the actor’s permissions (e.g., only the manager can seal) but also the **validity of the payload itself**. This would include checks such as:

- Ensuring the certificate is **digitally signed by a trusted authority**,

- Verifying that the signature is cryptographically sound and not expired,
- Confirming the payload matches expected content (e.g., schema, issuer, subject).

However, in this **proof-of-concept design**, the container **does not validate** the contents or signature of the payload. Instead, it trusts the **identity of the caller** (i.e., the manager) and assumes that a trusted actor will only provide valid certificates. This simplifies implementation and focuses on testing **role-based trust boundaries**. The omission of signature verification is a **deliberate design decision** made to prioritize clarity and visibility of protocol behavior, not a security oversight.

This step demonstrates that **unauthorized actors like Eve cannot bypass the system by submitting well-formed but unverified data**, as the permission to seal is enforced at the **actor level**, not just the data level.

The following base-58 encoded string is a **crafted payload** used in the `seal_container` call:

```
11t5gUyzT3MrGQDmRiorcdxp3ks9GnqRmHGfXER5XtboNmXwfpvFd3bjUSPMq1TXzNiLejis
WpeCzxy5FMDExdjJnEsK91Sbrv7VRm96WzCcemyEeKxK4r3N98jFggZMncTaaqZyyVmUyFb
tBQpKQfkNwLsYzMUdN1XvmevbeGjPoMJAU8LUPoBXmavAYKf7ET6Wyb2ca1uTpuzXm87
is5q5HzAnWs6mMSG2rfqbgMPTMNREk3J18dX9rw6SCn51kpQqTsXeNq5nSsEfDz23mpNg
sGdX7WWPqPQwkGfUb5b2nZxcRMAyH4AwVYbPrNrJ5ULL5r8WMUu3KYGyF3NbkaX6F2tu
3ex7PypZhNBF85HRAXRUcoCDJjxTigMzagG7c6HX2tQbkETEouc6Ra5zMAj7d4vZ4cPbtsyh
JxdyGQpNA76HZUQBHCkRkb7cTHxcbWEjmhjS2xynCrz1X9rLRbSckr614421bL4PEvfj9sgU
55BrTtKwVEoUfm47hwJNQ5yZpJzG6afD2xuf13p6NT7KjHS5RLgVLBoJxcUs4skgpgwz8iwjv
GaraMsutTcHHSJAEV6VFrLMmPbdUSRxJpGoQ449vYvo5cgJC8hDMcupHWgZFB91d5H3o
uaYDdbi1nHBQU1gXaXLB4FH2Fg9CGkRdKT25oAdERrYZ672TXV1nugvy6NhM2WASrovHzd
kwQDxLtXYMqQ4eKQt3J8jzQSdEn6bzQe74WAWaRUfQ9nuAZ
```

This payload contains a **valid digital certificate** that was **genuinely signed by Alice**, a legitimate user in the system. It was previously issued through a proper order flow and captured from her wallet’s certificate store.

However, the context here is critical: this payload is now being **replayed by Eve**, an unauthorized actor who is **not the container’s current manager**. While the certificate structure and signature are technically valid, Eve is not permitted to instruct the container to seal itself — regardless of the data’s legitimacy.

This test demonstrates how the system prioritizes **who** is issuing the command over **what** is being submitted. Even a valid certificate, when supplied by an illegitimate actor, **must be rejected** — and indeed, as we’ll see, it is.

This aligns with the principle of **identity-bound authority**, and highlights the design’s robustness in enforcing actor-based trust, not just data validation.

⛔ Attempting to Seal with a Forged Payload

We now attempt to invoke the `seal_container` function using the crafted payload described earlier.

So we try this payload out:

```
eve@vmlon0b:~$ wallet trade GyjiDkKE7oQUOTXFemNmu754XDr seal_container 11t5gUyzT3MrGQDmRiorcdxp3ks9GnqRmHGfXER5XtboNmXwfpvFd3bjUSPMq1TXzNiLejiswpeCzxy5FMDExdjJnEsK91Sbrv7VRm96WzCcemyEeKxK4r3N98jFggZMncTaaqZyyVmUyFbtBQpKQfkNwLsY
zMUdN1XvmevbeGjPoMJAU8LUPoBXmavAYKf7ET6Wyb2ca1uTpuzXm87is5q5HzAnWs6mMSG2rfqbgMPTMNREk3J18dX9rw6SCn51kpQqTsXeNq5nSsEfDz23mpNgsGdX7WWPqPQwkGfUb5b2nZxcRMAyH4AwVYbPrNrJ5ULL5r8WMUu3KYGyF3NbkaX6F2tu3ex7PypZhNBF85HRAXRUcoCDJjxTigMza
gG7c6HX2tQbkETEouc6Ra5zMAj7d4vZ4cPbtsyhJxdyGQpNA76HZUQBHCkRkb7cTHxcbWEjmhjS2xynCrz1X9rLRbSckr614421bL4PEvfj9sgU55BrTtKwVEoUfm47hwJNQ5yZpJzG6afD2xuf13p6NT7KjHS5RLgVLBoJxcUs4skgpgwz8iwjvGaraMsutTcHHSJAEV6VFrLMmPbdUSRxJpGoQ449vYV
o5cgJC8hDMcupHWgZFB91d5H3ouaYDdbi1nHBQU1gXaXLB4FH2Fg9CGkRdKT25oAdERrYZ672TXV1nugvy6NhM2WASrovHzdkwQDxLtXYMqQ4eKQt3J8jzQ5dEn6bzQe74WAWaRUfQ9nuAZ
KO 76003 Unauthorised user or trade.
```

Zoom: `KO 76003 Unauthorised user or trade.`

✅ Payload Accepted – ❌ Action Rejected

While the payload **was accepted** in terms of formatting and deserialization — confirming its validity — the container **refused to process the seal operation**.

This is by design. The container enforces a hardcoded rule: **only the actor currently recognized as the sender** in the trade may initiate the sealing process.

This experiment proves that **format-level validation alone is not enough** — **actor identity is enforced at execution time**, providing an additional layer of protection against forged or replayed inputs.

✅ This attempted attack is properly blocked by the backend.

🔗 Code Insight: Seal Authorization Logic

To reinforce the behavior we observed during the attempted attack, let's inspect the relevant logic from the container's source code (C++20). The sealing operation is handled by the following function:

```
ko c::seal(user_trader_t& user_trader, payload_t&& payload) {
    log("seal");

    if (auto r = validate_payload(payload); is_ko(r)) return r;

    {
        auto [r, user_ref] = agents.users.resolve(user_trader);
        if (is_ko(r)) return r;
        if (user_ref != agents.sender) {
            auto r = "KO 76003 Unauthorised user or trade.";
            log(r);
            return r;
        }
    }

    {
        auto [r, recipient_ref] = agents.users.resolve(payload.recipient_endpoint);
        if (is_ko(r)) return r;
        agents.recipient = move(recipient_ref);
    }

    {
        lock_guard lock(mx);
        if (use_status != use_status_t::opened) {
            auto r = "KO 76758 I am not opened.";
            log(r);
            return r;
        }
        log("payload set with this many certs", payload.items.size());
        this->payload = move(payload);
        use_status = use_status_t::sealed;
    }

    report_manager();

    log("inform user");
    if (agents.sender) {
        agents.sender->trader().schedule_exec_p9y("confirmed_sealed");
    }

    log("inform recipient");
    if (agents.recipient) {
        agents.recipient->trader().schedule_exec_p9y("container_is_coming " + this->payload.encode());
    }
    return ok;
}
```

The highlighted section enforces a **strict identity check**:

Only the agent previously assigned as the **sender** is authorized to call `seal`. Even if the payload is **syntactically valid** and **passes deserialization**, the container still performs this runtime verification to ensure trustworthiness.

This demonstrates a **layered security model**:

- The payload must be **well-formed and verifiable**.
- The actor attempting the operation must be **authorized by trade context**.

In our demo, signature validation is intentionally omitted during payload parsing to simplify experiments, but in a production deployment, additional cryptographic checks (e.g., certificate signature verification) would be enforced before or during payload acceptance.

This design ensures that unauthorized actors, like Eve in our example, **cannot seal** the container — even if they reverse-engineer a legitimate-looking payload.

🚫 Attempted Unseal: Blocked by Authorization Logic

As expected, attempting to invoke the `unseal_container` function from Eve's wallet results in the same rejection:

```
eve@vmlon0b:~$ wallet trade GyjiDkKE7oqUQTXFemNmu754XDr unseal_container
KO 76103 Unauthorised user or trade.
```

The Attack. Accessing certificates in a sealed container.

This exercise aims to demonstrate how the CLI offers fine-grained control over the container lifecycle, allowing us to replicate step-by-step what the `user send` wrapper command abstracts away. By using direct wallet-CLI calls, we gain visibility into each stage of the process—and also attempt to simulate an adversarial scenario.

Step-by-Step Plan

1. Restart the system

Ensure a clean slate by restarting the system. This guarantees that port1 has a fresh container in the `available` queue and the actors begin with a known state.

2. Start a trade with port1 from all three users:

- Alice
- Bob
- Eve

Alice:

```
alice@vmlon0b:~$ wallet trade start_bm port1
New trade. 31Y3a4tY1KPkucaGinRSrMvqCGgR
31Y3a4tY1KPkucaGinRSrMvqCGgR
Data:
  'activity': 1749679701335575670
  'bootstrapped_by': zuyUMFMJapXyr8TaTWQzrhUNPfh
  'created': 1749679701329968780
  'data_generated': 1749679701335646600
  'datasubdir': 31Y3a4tY1KPkucaGinRSrMvqCGgR
  'initiator': Y
  'lang': en
  'local_endpoint': zuyUMFMJapXyr8TaTWQzrhUNPfh anonymous
  'logfile': /home/alice/.katlas/wallet/trader/log/trade_1749679701329968780_31Y3a4tY1KPkucaGinRSrMvqCGgR
  'my_moniker': alice
  'my_personality': 3TaDdwk9s1GP8MyoCtK2NsEceabk
  'parent_tid': 31Y3a4tY1KPkucaGinRSrMvqCGgR
  'peer_moniker': anonymous
  'peer_personality': 11111111111111111111
  'peer_protocols': 0
  'protocol': not set
  'qrs': N
  'remote_endpoint': 3kdzg2ymfo3nbCHrpkyfY1hXHG 9mhtEQL86w9w5hfaDR5dAidZux4 "port1"
  'remote_ip4': 34.39.80.254:16773
  'roles': N
  'state': connected
```

Bob:

```
bob@vmlon0b:~$ wallet trade start 4VrH1C6JxGctXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1" u2c c
New trade. SAq1i6MjaqwBkJGu7HgfYa22vnJ
SAq1i6MjaqwBkJGu7HgfYa22vnJ
Data:
  'activity': 1749679875283893621
  'bootstrapped_by': 2zbZkCCvRrET6KtLVd9SgkquYkUo
  'created': 1749679875277905681
  'data_generated': 1749679875283941950
  'datasubdir': SAq1i6MjaqwBkJGu7HgfYa22vnJ
  'initiator': Y
  'lang': en
  'local_endpoint': 2zbZkCCvRrET6KtLVd9SgkquYkUo anonymous
  'logfile': /home/bob/.katlas/wallet/trader/log/trade_1749679875277905681_SAq1i6MjaqwBkJGu7HgfYa22vnJ
  'my_moniker': bob
  'my_personality': 3FRNp6rMLL4wVjAeCctgCjZ2VFip
  'parent_tid': SAq1i6MjaqwBkJGu7HgfYa22vnJ
  'peer_moniker': anonymous
  'peer_personality': 11111111111111111111
  'peer_protocols': 0
  'protocol': not set
  'qrs': N
  'remote_endpoint': 4VrH1C6JxGctXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1"
  'remote_ip4': 34.39.80.254:17073
  'roles': N
  'state': connected
```

Eve:

```
eve@vmlon0b:~$ wallet trade start 4VrH1C6JxGCtXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1" u2c c
New trade. Cxfe35bAr6tKPcZfwpUFSw57HQD
Cxfe35bAr6tKPcZfwpUFSw57HQD
Data:
  'activity': 1749679886170025163
  'bootstrapped_by': 3f5aHDKq97H5xHp6BD6PMEZRHZXt
  'created': 1749679886161885033
  'data_generated': 1749679886170066713
  'datasubdir': Cxfe35bAr6tKPcZfwpUFSw57HQD
  'initiator': Y
  'lang': en
  'local_endpoint': 3f5aHDKq97H5xHp6BD6PMEZRHZXt anonymous
  'logfile': /home/eve/.katlas/wallet/trader/log/trade_1749679886161885033_Cxfe35bAr6tKPcZfwpUFSw57HQD
  'my_moniker': eve
  'my_personality': 2bbtfZZYXUbRpoCnKWJ9jutoN9FT
  'parent_tid': Cxfe35bAr6tKPcZfwpUFSw57HQD
  'peer_moniker': anonymous
  'peer_personality': 1111111111111111111
  'peer_protocols': 0
  'protocol': not set
  'qrs': N
  'remote_endpoint': 4VrH1C6JxGCtXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1"
  'remote_ip4': 34.39.80.254:17073
  'roles': N
  'state': connected
```

Observation on Trade Initiation Methods

In the screenshots, we illustrate two equivalent—but stylistically different—methods of initiating a trade:

-  **Alice** used `start_bm` with a bookmark index (`port1`) to initiate a trade. This is a convenient approach when working within an environment where actors are pre-bookmarked.
-  **Bob** and **Eve** directly specified the container's full QR and personality in the `start` command. This is a lower-level method that bypasses bookmark resolution and gives full control—ideal for scripting or testing scenarios where direct addressing is preferred.

This flexibility reflects the CLI's dual nature: it's both user-friendly for guided workflows and powerful enough for precision operations and automation.

Alice Reserves and Confirms Her Container

Alice actions:

```
alice@vmlon0b:~$ wallet trade 31Y3a4tY1KPkucaGinRSrMvqCGgR assign_container
Chat:
  Wed Jun 11 2025 22:08:58      peer:
                               Your assigned container will contact you.

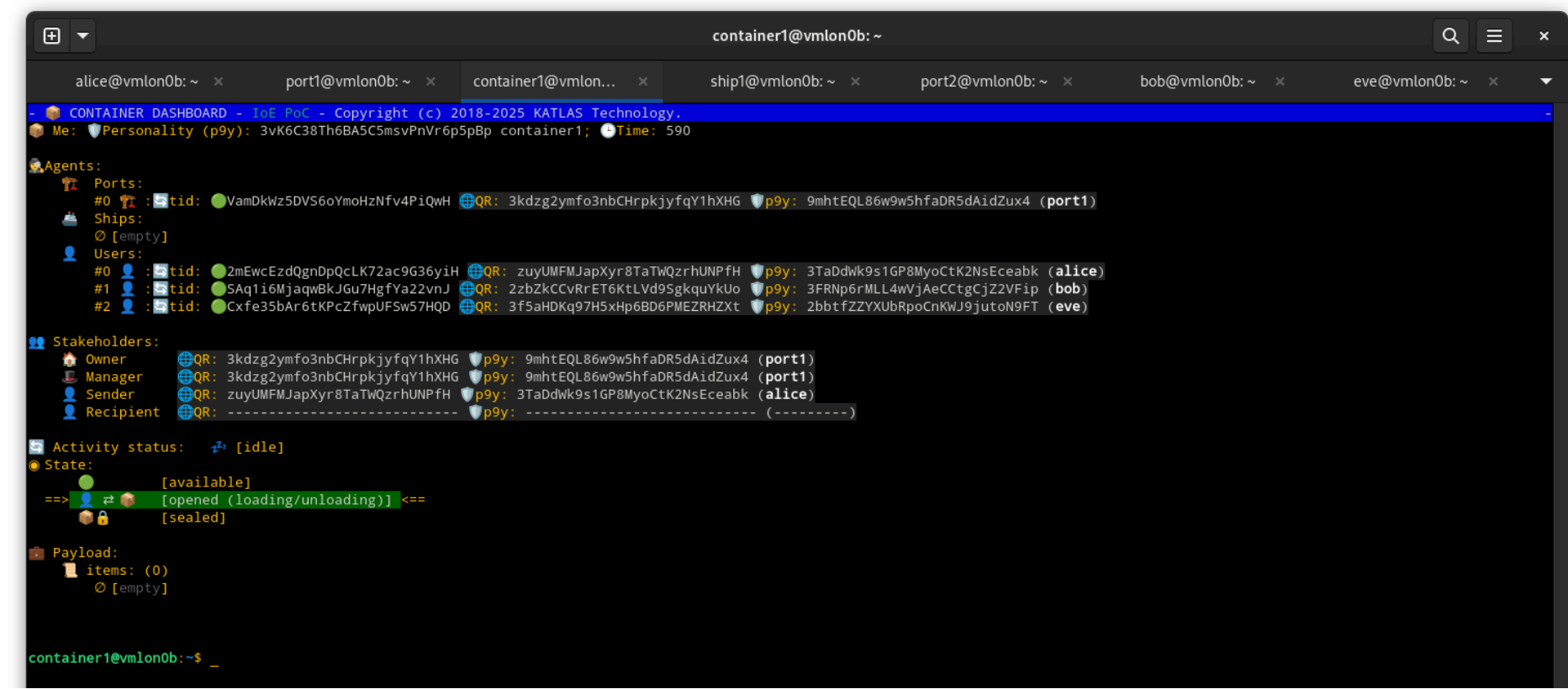
alice@vmlon0b:~$ wallet trade 31Y3a4tY1KPkucaGinRSrMvqCGgR container_accepted -t 4VrH1C6JxGCtXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1"
Thanks for confirming the reservation. Next: Load and seal.
```

This screenshots documents the initial phase of Alice's container booking process with the port:

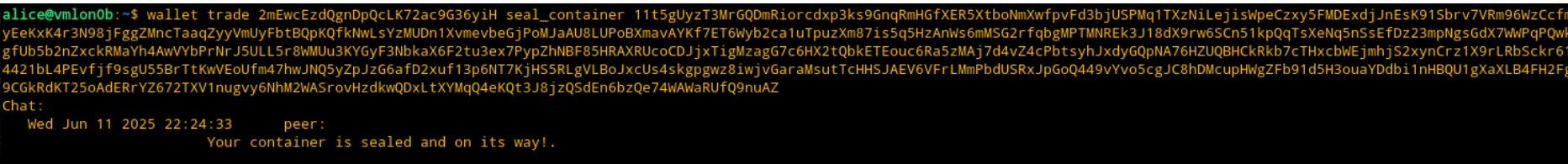
1. **Assign Request** – Alice issues `assign_container` to her active trade with `port1`. The port responds that a container will reach out to her—implicitly confirming the reservation is underway.
2. **Confirmation** – Alice follows up with `container_accepted`, explicitly acknowledging the assigned container (`container1`). This step locks the reservation and transitions the process to the loading and sealing phase.

 Container Status Overview

The container dashboard now reflects 3 active trades — one with each user: Alice, Bob, and Eve. Among them, Alice is correctly recognized and assigned as the sender, indicating she holds legitimate control over the container’s lifecycle operations.



 Sealing the Container

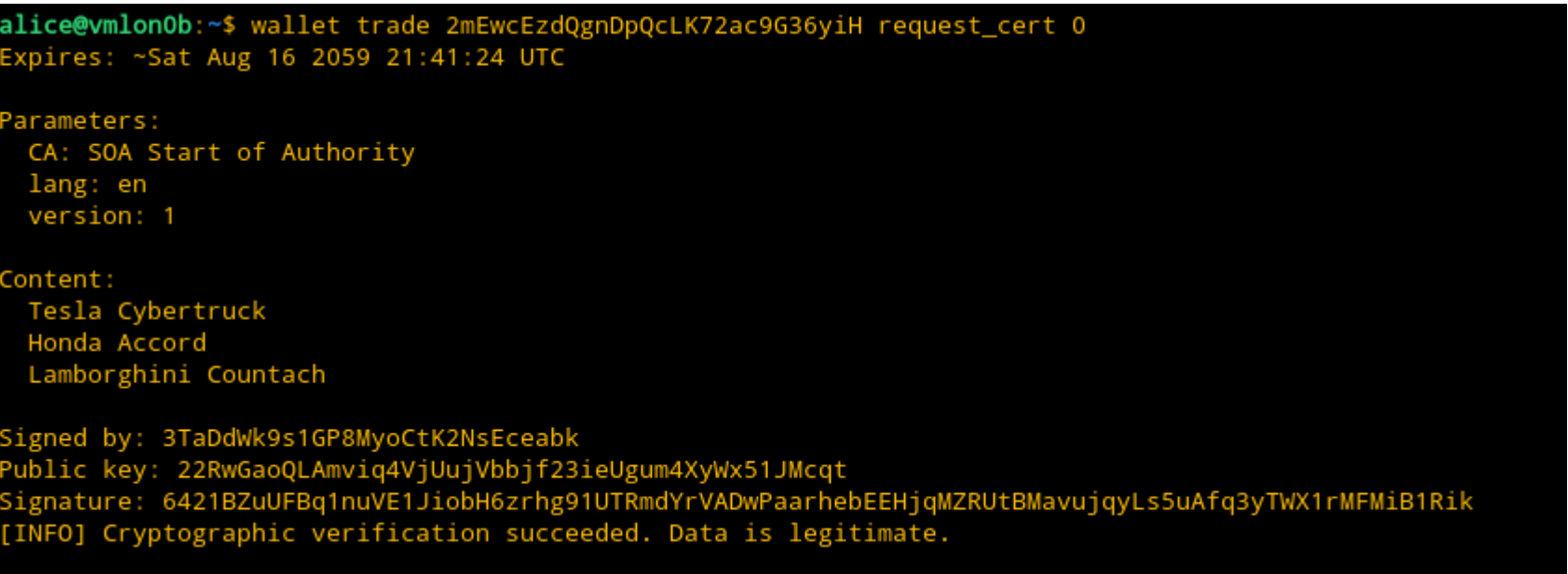


Alice has successfully sealed the container using a crafted payload. The container is now marked as sealed and has entered the outbound phase, under the control of the port, en route to Bob as its recipient.

 Attempting Certificate Access During Transit

While the container is sealed and in transit, each of the three actors—**Alice**, **Bob**, and **Eve**—attempts to retrieve one of the certificates stored inside:

Alice:



Bob:

```
bob@vmlon0b:~$ wallet trade list
SAq1i6MjaqwBkJGu7HgfYa22vnJ 1749679875277905681 1749679875336633401 / 4VrH1C6JxGCtXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1" u2c u

bob@vmlon0b:~$ wallet trade SAq1i6MjaqwBkJGu7HgfYa22vnJ request_cert 0
Expires: ~Sat Aug 16 2059 21:41:24 UTC

Parameters:
  CA: SOA Start of Authority
  lang: en
  version: 1

Content:
  Tesla Cybertruck
  Honda Accord
  Lamborghini Countach

Signed by: 3TaDdWk9s1GP8MyoCtK2NsEceabk
Public key: 22RwGaoQLAmviq4VjUujVbbjf23ieUgum4XyWx51JMcqt
Signature: 6421BZuUFBq1nuVE1JiobH6zrhg91UTRmdYrVADwPaarhebEEHjqMZRUtBMavujqyLs5uAfq3yTWX1rMFMiB1Rik
[INFO] Cryptographic verification succeeded. Data is legitimate.
```

Eve:

```
eve@vmlon0b:~$ wallet trade list
Cxfe35bAr6tKPcZfwpUFSw57HQD 1749679886161885033 1749679886220506684 / 4VrH1C6JxGCtXaGoqbr8jYKY9ZFp 3vK6C38Th6BA5C5msvPnVr6p5pBp "container1" u2c u

eve@vmlon0b:~$ wallet trade Cxfe35bAr6tKPcZfwpUFSw57HQD request_cert 0
KO 76039 Unauthorised user or trade.

KO 76039 Unauthorised user or trade.
```

-  **Alice** (the sender): succeeds in accessing the certificate, as she's an authorized stakeholder.
-  **Bob** (the recipient): also succeeds, having been granted visibility by design upon sealing.
-  **Eve** (an unauthorized user): is denied access, as the container enforces strict validation, only allowing certificate access to the sender, the recipient, or optionally a recognized authority.

 This reinforces the **privacy-preserving access control** design, which ensures that sensitive shipping data is only exposed to the intended parties and validates the PoC's security-by-design model where **sealed containers act autonomously**, upholding strict data confidentiality throughout transit.



User: Advanced/developer/insight functions

Last but not least—from a developer’s perspective, and in alignment with business goals of maintaining a codebase that is both extensible and debuggable—the CLI wrapper **user** includes a set of **advanced insight functions**. These are available across all actor accounts and provide introspection, diagnostics, and low-level control useful during development, testing, and system validation.

user st

```

eve@vmlon0b: ~
- Copyright (c) 2018-2025 KATLAS Technology.
=====
IoE PoC menu
eve
=====

-- common commands:
* restart ..... Restart all daemons (all actors, as superuser).
* bm ..... print my bookmarks.
* st ..... shows state screen and exists.
* watch ..... keeps showing state screen.

-- PoC commands for this role (u):
* send <file|-> <bm_index> ..... Invoke to deliver items to a given user.
* ship ..... sends a random cert to Bob via Port1.

-- backend: ● eve backend is up
* down ..... shut it down

-- dev/bootstrap/inspect:
* setup_bm ..... setup bookmarks from file my_bookmarks.
* logdir ..... /var/log/katlas/katlas-wallet_413834
* logtree ..... prints backend logs file structure.
* mlog ..... merge log files into an ordered list of events.
* plugins ..... list installed wallet plugins (All R2R protocols).

-- debug info:
wallet backend pid: 413834
logdir: /var/log/katlas/katlas-wallet_413834
eve@vmlon0b:~$ _
```

Focusing on the bottom half we find the dev section:

```

-- backend: ● eve backend is up
* down ..... shut it down

-- dev/bootstrap/inspect:
* setup_bm ..... setup bookmarks from file my_bookmarks.
* logdir ..... /var/log/katlas/katlas-wallet_413834
* logtree ..... prints backend logs file structure.
* mlog ..... merge log files into an ordered list of events.
* plugins ..... list installed wallet plugins (All R2R protocols).

-- debug info:
wallet backend pid: 413834
logdir: /var/log/katlas/katlas-wallet_413834
```



functions: user down and user up

These commands allow fine-grained control over the wallet daemon process **on a per-user basis**:

- **user down** — gracefully **shuts down** the wallet backend for the current user, simulating an outage or performing maintenance.
- **user up** — **restarts** the wallet daemon for the user, restoring normal operation.

These are particularly useful for testing, debugging or simulating actor unavailability in a multi-agent environment.

🔧 function: **user logtree**

```
eve@vmlon0b:~$ user logtree
/var/log/katlas/katlas-wallet_413834
├── daemon
│   ├── gov_rpc_daemon
│   │   ├── inbound
│   │   │   ├── traffic-inbound-single_139760685266624-0
│   │   │   └── outbound
│   │   │       ├── traffic-outbound_139760693659328-0
│   │   └── housekeeping_139760668481216-0
│   ├── inbound
│   │   ├── thpool
│   │   │   ├── worker_139760718837440-0
│   │   │   ├── worker_139760801814208-0
│   │   │   ├── worker_139760810206912-0
│   │   │   └── worker_139760818599616-0
│   │   └── traffic-inbound-multipeer-port_17473_139760826992320-0
│   ├── outbound
│   │   └── traffic-outbound_139760835385024-0
│   ├── peer_daemon_139760702052032-0
│   ├── peers
│   │   └── gc_139760710444736-0
│   ├── pushman_139760676873920-0
│   ├── main_139760860497792-0
│   ├── rbz
│   │   └── role_business_139760843777728-0
│   └── trades
│       └── trade_Cxfe35bAr6tKPcZfwpUFSw57HQD_139759913531072-0
```

This command reveals the **log structure** of the wallet daemon process for the current user. It provides:

- The **root log directory** used by the current wallet instance.
- A **tree view** showing active components (like the RPC daemon, peer manager, and trading engine) and their associated threads.
- Insight into how logs are **organized by function and execution context**, making it easier to trace events, debug workflows, and isolate concurrency issues.

This is especially valuable for developers and operators seeking visibility into the system's runtime behavior and performance.

function: **user mlog**

The mlog function (short for merge log) is a powerful diagnostic tool provided in the CLI-wrapper for wallet accounts. Its main purpose is to aggregate all log outputs from the various threads and components of the wallet backend into a unified, time-ordered view.

 How it works:

- Each wallet daemon spawns multiple threads (e.g. networking, worker pools, peer communication, R2R protocol handlers). Each thread typically writes logs to its own separate file under a structured directory tree.
- mlog reads all these files, parses the timestamps, and merges the logs line by line into a single chronological sequence.
- This provides a clear, global timeline of what happened in the wallet — invaluable during debugging, auditing, or tracing an incident.

 Benefits:

- Consolidation: Eliminates the need to manually jump between files to understand interactions between concurrent threads.
- Filtering: Standard UNIX tools like grep, less, or awk can be used on top of mlog to focus on specific events (e.g. user mlog | grep cert).
- Traceability: Each log line includes the filename and line number from the source code, aiding pinpoint analysis.

```
container1@vmlon0b: ~      x      ship1@vmlon0b: ~      x      port2@vmlon0b: ~      x      bob@vmlon0b: ~      x      eve@vmlon0b: ~
_139760718837440-0: worker gov/socket/outbound send_queue.cpp 62 overwriting service field with pri 7
_139760718837440-0: worker gov/socket/outbound send_queue.cpp 353 new ql. dgram sz 109 pri 7 cli 0x7f1c68006680 sendref 1
_139760718837440-0: worker gov bgtask.cpp 114 wakeup
_139760718837440-0: worker gov bgtask.cpp 120 reset resume
_139760718837440-0: worker gov/socket/outbound send_queue.cpp 27 onwakeup
_139760718837440-0: worker gov/id peer_t.cpp 234 turn_on_encryption?
_139760718837440-0: worker gov/id peer_t.cpp 229 get_keys
bund_139760835385024-0: traffic-outbound gov bgtask.cpp 106 reset_wait 1
_139760718837440-0: worker gov/crypto symmetric_encryption.cpp 27 init. pub_other: pmxSuv3ojpBzBa1AFzYDDKUp3ZXarjmH2pgvN9kYW1GA
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 223 awaken
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 217 take next
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 184 dgam cli 0x7f1c68006680 pri 7 progress 0 / 109
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 228 dgram pri 7 public remaining 109 dgram progress 0 / 109
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 247 ::send fd 21 sz 109 returned 109
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 249 sent 109 to 34.39.80.254:17373 progress 109 / 109 [ 100 %]
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 217 take next
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 178 finished. comm completed send. dend= 109 svc= 7 hash= 45wJj8ngvGZow64za8xSRYXe8hbz
bund_139760835385024-0: traffic-outbound gov/socket datagram.cpp 176 destructor
bund_139760835385024-0: traffic-outbound gov/socket datagram.cpp 220 deleted datagram 0x7f1c80007ad0
_139760718837440-0: worker gov/id peer_t.cpp 254 ===== ENCRYPTION TURNED ON =====
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 172 empty
_139760718837440-0: worker gov/id peer_t.cpp 609 call verification_completed 17373 0
bund_139760835385024-0: traffic-outbound gov bgtask.cpp 109 reset_wait 0
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 220 sleep
_139760718837440-0: worker wallet/engine peer_t.cpp 144 verification_completed. endpoint 34.39.80.254:17373 fd 21 rpport 17373 pin 0
bund_139760835385024-0: traffic-outbound gov bgtask.cpp 109 reset_wait 0
_139760718837440-0: worker gov/auth peer_t.cpp 20 verification_completed 34.39.80.254:17373 17373
bund_139760835385024-0: traffic-outbound gov/socket/outbound send_queue.cpp 316 wait till signal
_139760718837440-0: worker gov/id peer_t.cpp 101 peer has the same sw version
_139760718837440-0: worker wallet/engine peer_t.cpp 105 authorize pmxSuv3ojpBzBa1AFzYDDKUp3ZXarjmH2pgvN9kYW1GA 0
_139760718837440-0: worker wallet/engine peer_t.cpp 108 role peer or sysop authorized 1 0
_139760718837440-0: worker gov/auth peer_t.cpp 39 stage authorized
_139760718837440-0: worker wallet/engine peer_t.cpp 159 role_peer or role_sysop
_139760718837440-0: worker wallet/engine peer_t.cpp 167 role_peer
_139760718837440-0: worker gov/peer clique_t.cpp 27 clique add. peerPubKey lsb 137 dimension mask 0 advance it 0 clique sz 1 grid ptr 0x55d710324d30 unique 0
_139760718837440-0: worker gov/peer grid_t.cpp 177 add_to_grid 0x7f1c68006680 34.39.80.254:17373
_139760718837440-0: worker gov/peer grid_t.cpp 197 grid-added peer_t 0x7f1c68006680 34.39.80.254:17373 4HoUpiQNoQBSX3T4GYEULcjMnmNr
_139760718837440-0: worker wallet/engine peer_t.cpp 174 post_auth TRACE 8c
_139760718837440-0: worker wallet/engine daemon_t.cpp 180 post-auth announcing peer in correspondence 34.39.80.254:17373 reachable at port 17373
_139760718837440-0: worker wallet/engine peer_t.cpp 139 announce 17373
_139760718837440-0: worker wallet/engine daemon_t.cpp 389 on peer wallet 4HoUpiQNoQBSX3T4GYEULcjMnmNr 4266665762 17373
```

 Example: these lines are the result of filtering the merged log with word 'cert', showing events related to our last query from Eve tracing to the source code.

```
eve@vmlon0b: ~$ user mlog | grep cert
0x3b47e244674ca /var/log/katlas/katlas-wallet_413834/main_139760860497792-0: main wallet/trader traders_t.cpp 67 cert_authority_t::init /home/eve/.katlas/wallet/trader/ca
0x3b5783c2c6c61 /var/log/katlas/katlas-wallet_413834/daemon/inbound/thpool/worker_139760810206912-0: worker wallet/trader traders_t.cpp 675 exec Cxfe35bAr6tKPCzfwpUFSw57HQD request_cert 0
0x3b5783c2cb2e3 /var/log/katlas/katlas-wallet_413834/daemon/inbound/thpool/worker_139760810206912-0: worker wallet/trader conman.cpp 169 schedule_exec request_cert 0
0x3b5783c2e2f97 /var/log/katlas/katlas-wallet_413834/trades/trade_Cxfe35bAr6tKPCzfwpUFSw57HQD_139759913531072-0: trade_Cxfe35bAr6tKPCzfwpUFSw57HQD wallet/trader conman.cpp 217 cmdq sz 1 next cmd is request_cert 0
0x3b5783c2e4a09 /var/log/katlas/katlas-wallet_413834/trades/trade_Cxfe35bAr6tKPCzfwpUFSw57HQD_139759913531072-0: trade_Cxfe35bAr6tKPCzfwpUFSw57HQD wallet/trader conman.cpp 225 flushing command request_cert 0
0x3b5783c2eac15 /var/log/katlas/katlas-wallet_413834/trades/trade_Cxfe35bAr6tKPCzfwpUFSw57HQD_139759913531072-0: trade_Cxfe35bAr6tKPCzfwpUFSw57HQD wallet/trader conman.cpp 248 calling exec_online request_cert 0
0x3b5783c2ec6cd /var/log/katlas/katlas-wallet_413834/trades/trade_Cxfe35bAr6tKPCzfwpUFSw57HQD_139759913531072-0: trade_Cxfe35bAr6tKPCzfwpUFSw57HQD wallet/trader tid Cxfe35bAr6tKPCzfwpUFSw57HQD trader_t.cpp 139 request_cert 0
0x3b5783c2f11c3 /var/log/katlas/katlas-wallet_413834/trades/trade_Cxfe35bAr6tKPCzfwpUFSw57HQD_139759913531072-0: trade_Cxfe35bAr6tKPCzfwpUFSw57HQD wallet/trader tid Cxfe35bAr6tKPCzfwpUFSw57HQD trader_t.cpp 139 request_cert 0
0x3b5783c2f53e5 /var/log/katlas/katlas-wallet_413834/trades/trade_Cxfe35bAr6tKPCzfwpUFSw57HQD_139759913531072-0: trade_Cxfe35bAr6tKPCzfwpUFSw57HQD trader/r2r/u2c/u protocol.cpp 323 exec_online request_cert 0
0x3b5783c2f7e01 /var/log/katlas/katlas-wallet_413834/trades/trade_Cxfe35bAr6tKPCzfwpUFSw57HQD_139759913531072-0: trade_Cxfe35bAr6tKPCzfwpUFSw57HQD wallet/trader/workflow trader_protocol.cpp 190 exec_online request_cert 0
0x3b5783c2fc131 /var/log/katlas/katlas-wallet_413834/trades/trade_Cxfe35bAr6tKPCzfwpUFSw57HQD_139759913531072-0: trade_Cxfe35bAr6tKPCzfwpUFSw57HQD wallet/trader/workflow workflow_t.cpp 326 exec_online request_cert 0
0x3b5783c301a5f /var/log/katlas/katlas-wallet_413834/trades/trade_Cxfe35bAr6tKPCzfwpUFSw57HQD_139759913531072-0: trade_Cxfe35bAr6tKPCzfwpUFSw57HQD trader/r2r/u2c/u protocol.cpp 328 exec_online here request_cert 0
```



Codebase Metrics (via SLOCCount)

The `katlasnet` project encompasses:

- **Total Source Lines of Code (SLOC): 192,064**
- **Primary Languages:**
 - C++: 102,797 lines (53.5%)
 - Java: 61,534 lines (32.0%) (Android app)
 - Shell Scripts: 14,685 lines (7.6%)
 - XML, ANSI C, C#, Objective-C, JavaScript: ~3,000 lines combined



Development Effort Estimation (COCOMO Model)

- **Effort: ~50 person-years** (or ~600 person-months)
- **Schedule: ~2.4 years**
- **Team Size Estimate: ~21 full-time developers**
- **Estimated Cost: ~\$6.75 million USD**
(based on \$56,286/year salary + 2.4x overhead multiplier)

Note: These estimates are generated using the *Basic COCOMO* model and should be treated as ballpark figures. Real-world cost and productivity may vary.

- Tool used: `SLOCCount` by David A. Wheeler License: GNU GPL

✔ Conclusions: Alignment with Evaluation Objectives

This Proof of Concept successfully validates the Katlas platform against both primary use cases under review — **Supply Chain Data Visibility** and **Confidential, Role-Based AI Interaction** — while also addressing the **critical considerations and desired outcomes** laid out in the evaluation brief.

📦 Use Case 1: Blockchain for Supply Chain Data Visibility

✔ Role-specific access to consignment data

Each actor (user, port, container, ship) is instantiated with a cryptographically unique identity and wallet. This identity is used to control data flow — a container, for instance, will **only reveal contents to recognized agents** (e.g. sender, recipient, port authorities). Unauthorized agents (e.g., Eve) are denied access, as demonstrated in live attack scenarios.

✔ Earlier awareness for ports

Port dashboards show structured queues (stock, available, reserved, opened, sealed, arrivals, departures) in real-time, automatically updating as containers are sealed and dispatched. A port officer logging in can instantly see incoming containers, view only what their role permits, and trace origins with cryptographic integrity.

✔ Auditability & non-repudiation

All transactions (seal, unseal, load, cert access) are logged both in human-readable dashboards and in internal structured logs. These are traceable per actor, per certificate, and per transaction ID. Tools like `user logtree` and `user mlog` allow post-facto forensic analysis of every step.

✔ Fine-grained permissioning

Only the sender can seal; only the recipient can unseal; only authorized roles can access certificates. Attempts to circumvent these policies — even with correctly structured payloads — are logged and rejected. This is enforced both by protocol and C++ code-level logic.

🤖 Use Case 2: Confidential, AI-Powered Knowledge Agent

✔ Secure, role-bound AI dialogue

An AI-powered dialogue mechanism is embedded within the port agent, showcasing contextual and role-sensitive responses. Though simple in scope (rhymed responses based on prompt content), it proves the ability to **route chat interactions through AI logic constrained by wallet identity**.

✔ Foundation for role-differentiated interpretation

While all users may be able to submit queries, responses are **deliberately shaped by the role context**, mimicking how answers to a customs officer might differ from those given to a freight handler. This opens a clear path to implement more complex inference constraints on knowledge agents.

✔ No leakage of sensitive content

Knowledge exposure is bounded strictly by wallet authority. The PoC demonstrates this principle not just in AI chat, but also through data disclosure paths (e.g., certificates, ports, container manifest visibility), creating a consistent and enforceable trust model.

🔍 Critical Considerations Addressed

Consideration	PoC Response
Wallet/agent model scalability	Actors are instantiated automatically via declarative configuration. Bookmarks, identities, and keys are derived without manual coding. Bookmarks update dynamically as actors discover each other.
Authoring & validation of content	Certificates are generated and cryptographically signed before being loaded. Signature validation is enforced at retrieval time.
Governance & audit structures	Role-based permissions enforced at runtime. All interactions are traceable. Unauthorized attempts are denied and logged with cause codes.
Integration effort	All components (wallets, logs, agents, chat, and dashboards) operate independently and communicate via standard RPC or file-based contracts, simplifying integration. The system is developer-friendly, with introspection tools and CLI scripting support.

🎯 Desired Outcomes

Outcome	Validated
Platform maturity	✓ Demonstrated in full system deployment, cryptographic signing, security model, CLI introspection, and extensibility to new actors (e.g. Eve).
Real-world pilot readiness	✓ Use cases mimicking international shipping and customs interactions executed successfully. Dynamic actor orchestration and AI-in-the-loop tested.
Integration effort & extensibility	✓ Demonstrated automatic actor provisioning, plugin-based role binding, low-friction AI agent inclusion, and full logging infrastructure.

📌 Platform Assessment Summary

The Katlas platform, as evidenced by this Proof of Concept, offers a **robust, secure, and extensible framework** for addressing the dual challenges of **role-based supply chain transparency** and **confidential AI interaction**.

This exercise has:

-  **Proven role-based security** through live enforcement of access boundaries, including successful denials of unauthorized operations even under simulated attack conditions.
-  **Demonstrated seamless integration and interoperability** with external systems – notably, incorporating ChatGPT for AI-driven responses constrained by actor roles.
-  **Illustrated the scalability and modularity** of the platform, showing that actors and interactions can grow organically in both number and complexity without architectural friction.
-  **Provided clear proof-of-capability** by building, configuring, and executing a realistic, multi-actor ecosystem over our legacy platform – all with transparent automation, logging, and cryptographic enforcement.

This PoC confirms that Katlas is **technically mature enough to support structured pilots** across logistics, compliance, and intelligent advisory workflows. The system’s introspective developer tooling, modular wallet-agent design, and cryptographically anchored access controls make it a strong candidate for real-world deployments.

 We also hope that this demonstration – part technical exercise, part curated artwork – can **inspire and unlock new, challenging collaborations** aimed at advancing the platform’s maturity toward fully operational production environments.

This concludes this DEMO – demonstrating a scalable, actor-driven ecosystem in action.



"I love it when a plan comes together!"



KATLAS

Technology

KATLAS Technology are:

CEO: Edward Cole

CTO: (author of this demo): Marcos Mayorga

<https://katlastechnology.com>

