Comprehensive Guide to Docker: From Fundamentals to Advanced Applications

This guide provides a comprehensive overview of Docker, covering fundamental concepts like images, containers, and networking, progressing to advanced topics such as debugging distributed systems, scaling strategies, and integration with specific frameworks like Eclipse Velocitas. Whether you are new to Docker or looking to deepen your understanding, this document aims to equip you with the knowledge and best practices needed to effectively leverage containerization in your projects.

Part 1: Docker Fundamentals

1. Introduction to Docker

Docker has emerged as a transformative technology in the realm of software development and deployment, offering an open-source platform designed for containerization. At its core, Docker enables developers to package applications along with all their necessary dependencies—libraries, system tools, code, and runtime—into a standardized unit known as a container. This approach addresses a common and often frustrating challenge faced by developers: the "it works on my machine" problem. Before containerization solutions like Docker became widespread, inconsistencies between development, testing, and production environments frequently led to application failures upon deployment. Differences in operating systems, library versions, or configuration settings could prevent code that ran perfectly on a developer's local machine from functioning correctly on a server. Docker solves this by encapsulating the application and its environment, ensuring that it runs uniformly regardless of where the container is deployed.

The benefits of adopting Docker are manifold and contribute significantly to streamlining the software development lifecycle. One of the primary advantages is **portability**. Docker containers bundle everything an application needs, allowing them to be moved seamlessly across different environments—from a developer's laptop to on-premises servers or various cloud platforms—without modification. This consistency eliminates environment-specific bugs and simplifies testing. **Isolation** is another key benefit; each container runs as an isolated process, preventing interference with other containers or the host system. This enhances security and stability, particularly in multi-tenant environments. Furthermore, Docker containers are remarkably **resource-**

efficient compared to traditional virtual machines (VMs). While VMs virtualize hardware and require a full guest operating system, containers share the host system's OS kernel. This results in significantly lower overhead, reduced disk space and RAM consumption, faster startup times, and the ability to run more applications on the same hardware infrastructure. Docker also facilitates **scalability**, allowing developers to easily create multiple instances of a container to handle increased load, often managed through orchestration tools like Kubernetes or Docker Swarm. Finally, Docker accelerates the **development and deployment** process. It enables developers to work in local environments that mirror production, reducing compatibility issues. The deployment process becomes more straightforward and automatable, leading to faster release cycles and fewer deployment errors.

Understanding Docker's architecture provides insight into how it achieves these benefits. Docker utilizes a client-server architecture. The **Docker Client** is the primary interface through which users interact with Docker, issuing commands like `docker run` or `docker build`. The **Docker Daemon** (dockerd) is a background service running on the host machine that listens for Docker API requests and manages Docker objects. It handles the heavy lifting of building, running, and distributing containers. The **Docker Registry** is a system for storing and distributing Docker images. Docker Hub is the default public registry, but organizations can also host private registries. Key **Docker Objects** include **Images**, which are read-only templates containing instructions for creating a container; **Containers**, which are runnable instances of images; **Networks**, which provide communication channels between containers and the outside world; and **Volumes**, which manage persistent data used by containers.

(Source: dimensiona.com, geeksforgeeks.org)

2. Docker Images

Docker Images are the blueprints for Docker Containers. They are essentially read-only templates that contain a set of instructions for creating a container, including the application code, libraries, dependencies, tools, and other files needed for an application to run. Images are built from a set of instructions defined in a special text file called a Dockerfile. When you run an image, it becomes one or more instances of a container.

The power and efficiency of Docker Images stem largely from their layered architecture. Each instruction in a Dockerfile creates a new layer in the image. These layers are stacked on top of each other, and each layer represents a set of filesystem changes (additions, modifications, or deletions) compared to the layer below it. This layered approach, often implemented using Union File Systems (like Aufs, OverlayFS), has

several significant advantages. Firstly, layers are immutable; once created, they cannot be changed. If you modify a Dockerfile instruction, only the layer corresponding to that instruction and subsequent layers are rebuilt. Secondly, layers are shareable and reusable. If multiple images share the same base layers (e.g., an operating system layer or a runtime layer like Python or Node.js), Docker only stores those layers once on the host system. When you pull an image, Docker only downloads the layers that don't already exist locally. This significantly reduces disk space usage, speeds up image builds and downloads, and minimizes network bandwidth consumption. When a container is started from an image, Docker adds a thin, writable container layer on top of the immutable image layers. Any changes made within the running container, such as writing new files, modifying existing files, or deleting files, are written to this container layer. The underlying image layers remain untouched.

A Dockerfile is the cornerstone of building Docker Images. It's a text file containing a sequence of commands or instructions that Docker uses to assemble an image automatically. The Docker daemon reads these instructions line by line, executing them in order to create the final image. Key Dockerfile instructions include:

- **`FROM`**: Specifies the base image to start the build process from. Every Dockerfile must start with a `FROM` instruction (or `ARG` followed by `FROM`).
- **`RUN`**: Executes commands in a new layer on top of the current image. Used for installing packages, creating directories, etc.
- **`COPY`**: Copies files or directories from the build context (your local machine) into the image's filesystem.
- **`ADD`**: Similar to `COPY`, but with additional features like local-only tar extraction and URL support (though `COPY` is generally preferred for its transparency).
- **`CMD`**: Provides defaults for an executing container. There can only be one `CMD` instruction in a Dockerfile. If you specify a command when running the container (`docker run <image> <command>`), it overrides the `CMD`.
- **`ENTRYPOINT`**: Configures a container that will run as an executable. Similar to `CMD`, but it's harder to override when running the container. Often used in conjunction with `CMD` to specify default parameters.
- **`WORKDIR`**: Sets the working directory for any subsequent `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, and `ADD` instructions.
- **`EXPOSE`**: Informs Docker that the container listens on the specified network ports at runtime. It doesn't actually publish the port; it functions as documentation between the image builder and the person running the container.
- **`ENV`**: Sets environment variables within the image.
- **`ARG`**: Defines a build-time variable that users can pass at build time using the `--build-arg` flag with the `docker build` command.

- `VOLUME` : Creates a mount point with the specified name and marks it as holding externally mounted volumes from the native host or other containers.

Writing efficient and maintainable Dockerfiles involves several best practices. **Use official base images** whenever possible, as they are regularly updated and scanned for vulnerabilities. **Keep images small** by including only necessary components, using multi-stage builds to discard build-time dependencies, and minimizing the number of layers by chaining `RUN` commands (e.g., `RUN apt-get update && apt-get install -y --no-install-recommends package1 package2 && rm -rf /var/lib/apt/lists/*`). **Leverage build cache** effectively by ordering instructions from least frequently changing (like OS updates) to most frequently changing (like copying application code). Use a `.dockerignore` file to exclude unnecessary files and directories from the build context, reducing build time and image size. **Run containers as a non-root user** using the `USER` instruction for enhanced security. Avoid storing sensitive information directly in the Dockerfile; use build arguments, environment variables passed at runtime, or secrets management tools instead. Finally, **clearly document** your Dockerfile with comments explaining the purpose of each step.

Managing images involves commands like `docker pull <image>` to download images from a registry, `docker images` or `docker image ls` to list local images, `docker rmi <image>` or `docker image rm <image>` to remove images, and `docker build -t <tag> .` to build an image from a Dockerfile in the current directory. Images are typically stored and shared via registries like Docker Hub (the default public registry) or private registries (like AWS ECR, Google GCR, or self-hosted ones).

(Source: docs.docker.com, kodekloud.com, spacelift.io, sysdig.com, geeksforgeeks.org)

3. Docker Containers

If Docker Images are the blueprints, then Docker Containers are the actual running instances built from those blueprints. A container packages an application's code along with all its dependencies, libraries, and configuration files, ensuring it runs consistently and reliably across different computing environments. Each container is a lightweight, standalone, executable package that runs in isolation from the host system and other containers, sharing the host OS kernel but having its own isolated process space, network interface, and filesystem view. This isolation is a key feature, preventing conflicts between applications and enhancing security.

Understanding the container lifecycle is crucial for effective management. A container typically progresses through several states: **Created** (the container has been created but

not started, using `docker create`), **Running** (the container is actively executing, started with `docker run` or `docker start`), **Paused** (all processes within the container are suspended, using `docker pause`), **Stopped** (the container's main process has exited, either gracefully or forcefully using `docker stop`), and finally **Removed** (the container is permanently deleted from the host system using `docker rm`). Commands like `docker start`, `docker stop`, `docker pause`, and `docker unpause` allow control over these states.

The primary command for launching containers is `docker run`. This versatile command combines creating and starting a container in one step and offers numerous options to configure its behavior, such as mapping ports (`-p host_port:container_port`), attaching volumes (`-v volume_name:/path/in/container` or `-v /host/path:/path/in/container`), setting environment variables (`-e KEY=VALUE`), running in detached mode (`-d`), and assigning a name (`--name container_name`). Once a container is running, interaction is possible through several commands. `docker exec -it <container_name_or_id> <command>` allows you to run a command inside an already running container, often used to get a shell (`bash` or `sh`) for debugging or inspection. `docker logs <container_name_or_id>` fetches the standard output and standard error streams from the container's main process, essential for monitoring and troubleshooting. `docker attach <container_name_or_id>` connects your terminal's standard input, output, and error streams to a running container's main process.

Managing containers involves keeping track of their status and resources. `docker ps` lists currently running containers, while `docker ps -a` shows all containers, including stopped ones. `docker inspect <container_name_or_id>` provides detailed low-level information about a container, including its configuration, network settings, and volume mounts. As mentioned, `docker stop` sends a SIGTERM signal (followed by SIGKILL after a grace period) to the main process, allowing for a graceful shutdown, while `docker kill` sends a SIGKILL signal immediately. `docker rm` removes stopped containers; adding the `-f` flag forces removal of a running container.

One critical aspect of working with containers is data persistence. By default, any data written inside a container's writable layer is ephemeral; it is lost when the container is removed. For applications like databases, logging systems, or user-generated content platforms, this is unacceptable. Docker provides two main mechanisms for persisting data beyond the container's lifecycle: **Volumes** and **Bind Mounts**. **Volumes** are the preferred method. They are managed by Docker, stored in a dedicated area on the host filesystem (often `/var/lib/docker/volumes/`), and are decoupled from the container's lifecycle. Volumes can be created explicitly (`docker volume create`) or

automatically when a container starts. They offer benefits like easier backup, migration, and sharing between containers. **Bind Mounts**, on the other hand, mount a file or directory from the host machine directly into the container. While useful for development (e.g., mounting source code into a container for live updates), they are less flexible than volumes, depend on the host's directory structure, and can potentially expose sensitive host files or create permission issues. Both volumes and bind mounts are specified using the `-v` or `--mount` flag with `docker run`.

(Source: docs.docker.com, dev.to, medium.com, geeksforgeeks.org, daily.dev)

4. Docker Networking

Networking is a fundamental aspect of Docker, enabling communication between containers, between containers and the host machine, and between containers and the outside world. By default, containers are network-enabled and can make outgoing connections, but Docker provides a flexible system with various drivers and configurations to tailor network behavior to specific application needs. Understanding these concepts is crucial for building robust, scalable, and secure containerized applications.

Docker's networking subsystem is pluggable, utilizing different **network drivers** to provide distinct networking capabilities. Several drivers come built-in. The **`bridge`** driver is the default for standalone containers. When Docker starts, it creates a default bridge network (often named `docker0`), and containers launched without a specific network assignment are attached to it. Containers on the same default bridge network can communicate using their internal IP addresses, but automatic DNS resolution by container name is not supported on the default bridge. The **`host`** driver removes network isolation between the container and the Docker host. The container shares the host's network stack directly, meaning services running inside the container appear as if they were running directly on the host (e.g., binding to port 80 in the container binds to port 80 on the host). This offers performance benefits but sacrifices the isolation advantage. The **`none`** driver completely disables networking for a container, isolating it entirely from any network communication; it only retains a loopback interface. The **`overlay`** driver is designed for multi-host networking, connecting multiple Docker daemons together and enabling containers running on different hosts (often managed by an orchestrator like Docker Swarm or Kubernetes) to communicate securely over an overlay network. **`macvlan`** allows assigning a MAC address to a container, making it appear as a physical device on the network, directly connected to the physical network infrastructure. **`ipvlan`** offers similar capabilities to `macvlan` but provides more control over IPv4 and IPv6 addressing.

While the default bridge network is convenient for simple use cases, **user-defined networks** are strongly recommended for most applications, especially multi-container setups. User-defined networks, typically created using the `bridge` driver (`docker network create --driver bridge my-custom-network`), offer significant advantages over the default bridge. Firstly, they provide better **isolation**; containers on different user-defined networks cannot communicate unless explicitly connected to both, whereas all containers on the default bridge can potentially reach each other. Secondly, user-defined networks provide automatic **DNS resolution** between containers using their container names or network aliases. This means a web container can reach a database container simply by using the database container's name (e.g., `db`) as the hostname, without needing to know its internal IP address, which can change. Containers can be connected to multiple networks simultaneously, allowing for complex network topologies (e.g., a frontend container connected to both an external-facing network and an internal backend network).

Exposing container services to the outside world or the host machine is typically done via **port publishing**. Using the `-p` or `--publish` flag with `docker run` (e.g., `-p 8080:80`) maps a port on the Docker host (8080) to a port inside the container (80). By default, this makes the container port accessible from any interface on the host, including external ones. To restrict access to only the host machine, you can specify the host IP address (e.g., `-p 127.0.0.1:8080:80`).

For multi-container applications, managing individual container networks and links can become cumbersome. **Docker Compose** simplifies this significantly. By defining services, networks, and volumes in a `docker-compose.yml` file, Compose automatically creates a default user-defined bridge network for the application. All services (containers) defined in the file are attached to this network, enabling easy communication between them using service names as hostnames, thanks to the built-in DNS resolution. Compose allows defining custom networks, connecting services to multiple networks, and specifying network aliases, providing a declarative way to manage complex application networking setups.

(Source: docs.docker.com, medium.com, spacelift.io, warp.dev)

Part 2: Debugging Dockerized Distributed Systems

Debugging applications deployed within Docker containers, especially when they form part of a larger distributed system, presents unique challenges compared to traditional monolithic applications running on a single host. The inherent complexity arises from multiple interacting services, potentially spread across different containers and even hosts, communicating over networks. Issues can stem from individual container

misconfigurations, application bugs, network connectivity problems between containers, resource contention, or failures in the underlying infrastructure. The ephemeral nature of containers adds another layer of difficulty, as logs and state might be lost if not managed correctly. Effectively debugging these systems requires a combination of Docker-specific tools, robust observability practices (logging, monitoring, tracing), and systematic troubleshooting methodologies.

Fundamental Docker commands provide the first line of defense when investigating issues within individual containers. The `docker logs <container_id>` command is indispensable for retrieving the standard output and standard error streams generated by the main process running inside a container. This often reveals application errors, stack traces, or status messages crucial for diagnosis. For real-time observation, `docker attach <container_id>` streams the container's output directly to your terminal. When deeper inspection is needed, `docker exec -it <container_id> /bin/sh` (or `bash`) grants interactive shell access inside a running container. This allows you to examine the container's filesystem, check running processes (`ps aux`), test network connectivity from within the container (`ping`, `curl`), inspect configuration files, or even install temporary debugging tools (`apt-get update && apt-get install -y net-tools`). The `docker inspect <container_id>` command provides a wealth of low-level configuration details about a container, including its IP address, network settings, mounted volumes, environment variables, and state. For performance issues, `docker stats` offers a live stream of resource usage metrics (CPU, memory, network I/O, block I/O) for running containers, helping to identify resource bottlenecks.

While these commands are useful for individual containers, effective debugging in a distributed system necessitates a more holistic approach centered around observability. Comprehensive **logging** is paramount. Simply relying on `docker logs` is often insufficient for complex systems. Implementing centralized logging, where logs from all containers are aggregated into a single, searchable platform (like the ELK stack - Elasticsearch, Logstash, Kibana, or alternatives like Fluentd, Graylog, Splunk), is crucial. Adopting structured logging formats (e.g., JSON) makes logs easier to parse, filter, and analyze. Logs should include contextual information like timestamps, service names, request IDs (correlation IDs), and user IDs to enable tracing requests across different services.

Complementing logging, **monitoring** provides real-time insights into the health and performance of the system. This involves collecting metrics from containers (CPU, memory, network usage via `docker stats` or cAdvisor), the Docker daemon itself, and the applications running within the containers (e.g., request latency, error rates, queue lengths). Tools like Prometheus for metrics collection, Grafana for visualization, and Alertmanager for notifications form a popular open-source monitoring stack.

Commercial solutions like Datadog, Dynatrace, and New Relic also offer comprehensive container monitoring capabilities. Monitoring helps detect anomalies, identify performance bottlenecks, and understand resource utilization patterns.

Understanding the flow of requests as they traverse multiple services in a distributed system is often critical for pinpointing latency issues or failures. This is where **distributed tracing** comes in. By instrumenting application code to propagate unique trace IDs across service calls, tracing tools (like Jaeger, Zipkin, Tempo) can reconstruct the entire journey of a request, visualizing the sequence of operations and the time spent in each service (spans). This end-to-end visibility is invaluable for identifying bottlenecks, understanding service dependencies, and diagnosing errors that only manifest during inter-service communication.

Network issues are a common source of problems in containerized distributed systems. Beyond using `ping` or `curl` inside containers via `docker exec`, `docker network inspect <network_name>` provides details about a specific Docker network, including connected containers and their IP addresses. Ensuring containers are connected to the correct user-defined networks and checking for port conflicts or firewall rules (on the host or external) that might block traffic are essential steps. Tools like `tcpdump` or Wireshark can be run on the host or even within containers (if installed) for deep network packet analysis.

For complex application bugs, **remote debugging** might be necessary. This involves attaching a language-specific debugger (like GDB for C/C++, PDB for Python, Java's JDWP debuggers) to a process running inside a container. This typically requires exposing a debug port from the container (using the `-p` flag) and configuring the application to start in debug mode. IDEs often have integrations to facilitate attaching to remote processes running in Docker containers, allowing developers to set breakpoints, inspect variables, and step through code execution as if it were running locally.

Finally, adopting certain best practices can significantly simplify debugging. Running only **one primary process per container** makes it easier to isolate failures and interpret logs. Implementing **health checks** within Dockerfiles (`HEALTHCHECK` instruction) or orchestration systems allows Docker or the orchestrator to automatically detect and restart unhealthy containers. **Simulating failures** using chaos engineering principles can proactively uncover weaknesses in the system's resilience and error handling. Consistent use of correlation IDs across logs and traces is vital for correlating events related to a single request or transaction.

(Source: lumigo.io, geeksforgeeks.org, docker.com, medium.com)

Part 3: Scaling with Docker: From Single Container to Multi-Container Networks

As applications grow in complexity and demand, the initial approach of running everything within a single Docker container quickly becomes inefficient and difficult to manage. While containerizing a simple script or a monolithic web application is straightforward, modern applications often consist of multiple specialized services working together – a web frontend, backend APIs, databases, caching layers, message queues, etc. Attempting to bundle all these components into one container violates the best practice of "one process per container," leading to bloated images, complex entrypoint scripts, difficulties in independent scaling and updates, and challenges in debugging. The transition to a multi-container architecture, orchestrated effectively, is key to building scalable, resilient, and maintainable systems with Docker.

The journey from a single container typically involves decomposing the application into logical services, each running in its own dedicated container. For instance, a web application might be split into a frontend container (e.g., running Nginx or a Node.js server for static assets), one or more backend API containers (e.g., running Python/Flask, Java/Spring Boot, or Go), and a database container (e.g., PostgreSQL, MySQL, MongoDB). This modular approach offers numerous advantages: services can be developed, deployed, updated, and scaled independently; technology stacks can be chosen appropriately for each service; and failures are better isolated, preventing a crash in one service from bringing down the entire application.

However, managing multiple containers individually using separate `docker run` commands introduces significant operational overhead. Manually configuring networks for inter-container communication, managing dependencies and startup order, handling persistent data for stateful services, and ensuring consistent configurations across environments becomes increasingly complex and error-prone. This is precisely where **Docker Compose** becomes an essential tool. Docker Compose allows you to define and manage your entire multi-container application using a single declarative YAML file (`compose.yml`). In this file, you define each component as a `service`, specifying its build instructions (or pre-built image), ports, volumes, environment variables, dependencies (`depends_on`), and network connections.

Docker Compose significantly simplifies the networking aspect of multi-container applications. By default, when you run `docker compose up`, Compose creates a dedicated user-defined bridge network for your application stack. All services defined in the `compose.yml` file are automatically attached to this network. Crucially, Compose provides built-in **service discovery** via DNS within this network. This means one service container (e.g., the backend API) can reach another service container (e.g., the database)

simply by using the service name defined in the `compose.yml` file (e.g., `db`) as its hostname. Compose handles the underlying IP address resolution, abstracting away the complexities of container IP management. You can also define custom networks in the Compose file for more complex scenarios, such as creating separate frontend and backend networks for enhanced security.

Scaling individual services within your application stack is another area where Docker Compose excels. Instead of manually running multiple `docker run` commands for a service, you can simply use the `--scale` option with `docker compose up`. For example, `docker compose up --scale backend=3` would start three instances (containers) of the `backend` service defined in your `compose.yml`. When scaling services like web servers or API backends that need to handle incoming traffic, you typically need a **load balancer** or **reverse proxy** (like Nginx or Traefik) running in its own container within the same Compose network. This proxy acts as the single entry point for external traffic, distributing requests across the available instances of the scaled service (e.g., using round-robin or other strategies). Docker Compose makes setting up such a proxy straightforward, as the proxy can discover and route traffic to the backend service instances using their service names.

Transitioning to a multi-container architecture requires careful consideration of data persistence for stateful services like databases. Docker Compose integrates seamlessly with Docker Volumes. You can define named volumes directly within the `compose.yml` file and mount them into the appropriate service containers, ensuring that data survives container restarts, updates, or scaling operations. Managing configuration consistently across services and environments is also simplified using environment variables defined within the Compose file or external `.env` files.

In summary, scaling from a single Docker container involves embracing a microservices-oriented approach by decomposing the application into multiple, specialized containers. Docker Compose provides the essential orchestration capabilities to define, run, network, and scale these multi-container applications efficiently, simplifying development workflows and enabling the creation of robust, scalable distributed systems.

(Source: docs.docker.com, docker77.hashnode.dev, medium.com, stackoverflow.com)

Part 4: Docker and Eclipse Velocitas

Eclipse Velocitas is an open-source project focused on providing a standardized development toolchain and runtime environment for creating modern, containerized in-

vehicle applications, often referred to as Vehicle Apps. A core principle of Velocitas is leveraging containerization technology, specifically Docker, to ensure consistency, portability, and reproducibility across the entire development lifecycle – from local development workstations to CI/CD pipelines and potentially target vehicle hardware (often via platforms like Eclipse Leda).

One of the key ways Velocitas integrates Docker is through the use of **VS Code DevContainers**. The Velocitas project provides template repositories (available for languages like Python and C++) that come pre-configured with a `.devcontainer` definition. This definition specifies a Docker image containing all the necessary tools, SDKs (like the Velocitas Vehicle App SDK), dependencies, and even VS Code extensions required for Vehicle App development. When a developer clones a Velocitas template repository and opens it in VS Code with the Remote-Containers extension installed, VS Code automatically prompts to reopen the project inside this defined DevContainer. This action builds (if not already cached) and starts the Docker container, effectively providing a fully provisioned, isolated, and consistent development environment regardless of the developer's host operating system or local configuration. This significantly simplifies onboarding and eliminates common "works on my machine" issues.

Beyond the development environment setup, Velocitas utilizes Docker to manage and run essential **runtime services** needed for testing and debugging Vehicle Apps locally. These services often include components like the KUKSA Databroker (for vehicle signal abstraction), MQTT brokers for communication, and potentially simulated vehicle services. The Velocitas toolchain includes commands (often wrapped in VS Code tasks like `Local Runtime - Up`) that use Docker or Docker Compose under the hood to start these dependent services in their own containers, connecting them via a Docker network. This allows developers to run and interact with their Vehicle App within the DevContainer while communicating with these necessary backend services running locally in separate containers, simulating a realistic runtime environment. Debugging is also facilitated within this setup; developers can set breakpoints in their Vehicle App code running inside the DevContainer and trigger them by interacting with the runtime services (e.g., sending MQTT messages or querying the Databroker).

Furthermore, the **CI/CD workflows** provided in the Velocitas template repositories heavily rely on Docker. These workflows, typically defined for platforms like GitHub Actions, automate the process of building the Vehicle App, running unit and integration tests, and ultimately packaging the application as a container image. The build process usually involves a Dockerfile (often leveraging multi-stage builds for optimization) that takes the application code and dependencies and creates a production-ready Docker image. This image is then often pushed to a container registry (like GitHub Container

Registry) upon successful testing and release tagging. This containerized artifact is the intended deployment unit, designed to be run on compatible target platforms like Eclipse Leda, which itself manages container deployments on automotive-grade hardware.

In essence, Docker is deeply integrated into the Eclipse Velocitas workflow. It provides the foundation for consistent development environments (DevContainers), enables local simulation of runtime dependencies, and serves as the core technology for building, testing, and packaging Vehicle Apps for deployment. By leveraging Docker, Velocitas aims to streamline the development and deployment of software-defined vehicle applications.

(Source: eclipse.dev/velocitas, github.com/eclipse-velocitas, eclipse-leda.github.io)