Step 1:

File: cmd/dockerd/docker.go

```
func main() {
    if reexec.Init() {
        return
    }

    // initial log formatting; this setting is updated after the daemon
configuration is loaded.
    logrus.SetFormatter(&logrus.TextFormatter{
        TimestampFormat: jsonmessage.RFC3339NanoFixed,
        FullTimestamp:   true,
    })

    // Set terminal emulation based on platform as required.
    _, stdout, stderr := term.StdStreams()

    initLogging(stdout, stderr)
    configureGRPCLog()

    onError := func(err error) {
        fmt.Fprintf(stderr, "%s\n", err)
        os.Exit(1)
    }

    cmd, err := newDaemonCommand()
    if err != nil {
        onError(err)
    }
    cmd.SetOut(stdout)
    if err := cmd.Execute(); err != nil {
        onError(err)
    }
}
```

which part? `cmd, err := newDaemonCommand()`

What Does it do?

```
1. Check if the process is a re-executed process.
   - It uses `reexec.Init()` to determine if the process is a re-executed
one.

2. Set the log format.
   - It configures the logrus logger with a specific timestamp format and
full timestamp display.

3. Set the terminal emulation.
```

```
   - It determines the standard input, output, and error streams of the
terminal.

4. Initialize the logging.
   - It initializes the logging with the configured standard output and
error streams.

5. Configure the GRPC log.
   - It sets up configuration for GRPC (Google Remote Procedure Call)
logging.

6. Create a new daemon command.
   - It creates a new daemon command instance, which will handle various
operations.

7. Execute the command.
   - It executes the previously created daemon command, which likely
performs Docker-related operations.
```

Step 2:

File: cmd/dockerd/docker.go

```
func newDaemonCommand() (*cobra.Command, error) {
    cfg, err := config.New()
    if err != nil {
        return nil, err
    }
    opts := newDaemonOptions(cfg)

    cmd := &cobra.Command{
        Use:           "dockerd [OPTIONS]",
        Short:         "A self-sufficient runtime for containers.",
        SilenceUsage:  true,
        SilenceErrors: true,
        Args:          cli.NoArgs,
        RunE: func(cmd *cobra.Command, args []string) error {
            opts.flags = cmd.Flags()
            return runDaemon(opts)
        },
        DisableFlagsInUseLine: true,
        Version:               fmt.Sprintf("%s, build %s",
dockerversion.Version, dockerversion.GitCommit),
    }
    cli.SetupRootCommand(cmd)

    flags := cmd.Flags()
    flags.BoolP("version", "v", false, "Print version information and
quit")
    defaultDaemonConfigFile, err := getDefaultDaemonConfigFile()
    if err != nil {
        return nil, err
    }
    flags.StringVar(&opts.configFile, "config-file",
defaultDaemonConfigFile, "Daemon configuration file")
    configureCertsDir()
    opts.installFlags(flags)
    if err := installConfigFlags(opts.daemonConfig, flags); err != nil {
        return nil, err
    }
    installServiceFlags(flags)

    return cmd, nil
}
```

What Does it do?

```
1. Create a new configuration instance.
2. Initialize daemon options based on the configuration.
3. Define a new Cobra command for the 'dockerd' command.
4. Set up the root command for the CLI.
5. Add a flag to print version information.
6. Get the default daemon configuration file path.
7. Add a flag to specify the daemon configuration file.
8. Configure certificates directory.
9. Install flags for daemon options.
10. Install configuration flags.
11. Install service flags.
```

Step 3:

File: cmd/dockerd/docker_unix.go

```
func runDaemon(opts *daemonOptions) error {
    daemonCli := NewDaemonCli()
    return daemonCli.start(opts)
}
```

What Does it do?

```
- Create a new Daemon CLI instance.
- Start the Daemon CLI using the provided options.
```

Step 4:

File: cmd/dockerd/daemon.go

```go
func (cli *DaemonCli) start(opts *daemonOptions) (err error) {
	if cli.Config, err = loadDaemonCliConfig(opts); err != nil {
		return err
	}
	if err := checkDeprecatedOptions(cli.Config); err != nil {
		return err
	}

	serverConfig, err := newAPIServerConfig(cli.Config)
	if err != nil {
		return err
	}

	if opts.Validate {
		// If config wasn't OK we wouldn't have made it this far.
		_, _ = fmt.Fprintln(os.Stderr, "configuration OK")
		return nil
	}

	configureProxyEnv(cli.Config)
	configureDaemonLogs(cli.Config)

	logrus.Info("Starting up")

	cli.configFile = &opts.configFile
	cli.flags = opts.flags

	if cli.Config.Debug {
		debug.Enable()
	}

	if cli.Config.Experimental {
		logrus.Warn("Running experimental build")
	}

	if cli.Config.IsRootless() {
		logrus.Warn("Running in rootless mode. This mode has feature
limitations.")
	}
	if rootless.RunningWithRootlessKit() {
		logrus.Info("Running with RootlessKit integration")
		if !cli.Config.IsRootless() {
			return fmt.Errorf("rootless mode needs to be enabled for
running with RootlessKit")
		}
	}

	// return human-friendly error before creating files
	if runtime.GOOS == "linux" && os.Geteuid() != 0 {
		return fmt.Errorf("dockerd needs to be started with root
privileges. To run dockerd in rootless mode as an unprivileged user, see
https://docs.docker.com/go/rootless/")
    }

    if err := setDefaultUmask(); err != nil {
        return err
    }

    // Create the daemon root before we create ANY other files (PID, or
migrate keys)
    // to ensure the appropriate ACL is set (particularly relevant on
Windows)
    if err := daemon.CreateDaemonRoot(cli.Config); err != nil {
        return err
    }

    if err := system.MkdirAll(cli.Config.ExecRoot, 0700); err != nil {
        return err
    }

    potentiallyUnderRuntimeDir := []string{cli.Config.ExecRoot}

    if cli.Pidfile != "" {
        pf, err := pidfile.New(cli.Pidfile)
        if err != nil {
            return errors.Wrap(err, "failed to start daemon")
        }
        potentiallyUnderRuntimeDir = append(potentiallyUnderRuntimeDir,
cli.Pidfile)
        defer func() {
            if err := pf.Remove(); err != nil {
                logrus.Error(err)
            }
        }()
    }

    if cli.Config.IsRootless() {
        // Set sticky bit if XDG_RUNTIME_DIR is set && the file is actually
under XDG_RUNTIME_DIR
        if _, err :=
homedir.StickRuntimeDirContents(potentiallyUnderRuntimeDir); err != nil {
            // StickRuntimeDirContents returns nil error if XDG_RUNTIME_DIR
is just unset
            logrus.WithError(err).Warn("cannot set sticky bit on files
under XDG_RUNTIME_DIR")
        }
    }

    cli.api = apiserver.New(serverConfig)

    hosts, err := loadListeners(cli, serverConfig)
    if err != nil {
        return errors.Wrap(err, "failed to load listeners")
    }
```

```
	ctx, cancel := context.WithCancel(context.Background())
	waitForContainerDShutdown, err := cli.initContainerD(ctx)
	if waitForContainerDShutdown != nil {
		defer waitForContainerDShutdown(10 * time.Second)
	}
	if err != nil {
		cancel()
		return err
	}
	defer cancel()

	stopc := make(chan bool)
	defer close(stopc)

	trap.Trap(func() {
		cli.stop()
		<-stopc // wait for daemonCli.start() to return
	}, logrus.StandardLogger())

	// Notify that the API is active, but before daemon is set up.
	preNotifyReady()

	pluginStore := plugin.NewStore()

	if err := cli.initMiddlewares(cli.api, serverConfig, pluginStore); err
!= nil {
		logrus.Fatalf("Error creating middlewares: %v", err)
	}

	d, err := daemon.NewDaemon(ctx, cli.Config, pluginStore)
	if err != nil {
		return errors.Wrap(err, "failed to start daemon")
	}

	d.StoreHosts(hosts)

	// validate after NewDaemon has restored enabled plugins. Don't change
order.
	if err := validateAuthzPlugins(cli.Config.AuthorizationPlugins,
pluginStore); err != nil {
		return errors.Wrap(err, "failed to validate authorization plugin")
	}

	cli.d = d

	if err := startMetricsServer(cli.Config.MetricsAddress); err != nil {
		return errors.Wrap(err, "failed to start metrics server")
	}

	c, err := createAndStartCluster(cli, d)
	if err != nil {
		logrus.Fatalf("Error starting cluster component: %v", err)
	}
	// Restart all autostart containers which has a swarm endpoint
	// and is not yet running now that we have successfully
	// initialized the cluster.
	d.RestartSwarmContainers()

	logrus.Info("Daemon has completed initialization")

	routerOptions, err := newRouterOptions(cli.Config, d)
	if err != nil {
		return err
	}
	routerOptions.api = cli.api
	routerOptions.cluster = c

	initRouter(routerOptions)

	go d.ProcessClusterNotifications(ctx, c.GetWatchStream())

	cli.setupConfigReloadTrap()

	// The serve API routine never exits unless an error occurs
	// We need to start it as a goroutine and wait on it so
	// daemon doesn't exit
	serveAPIWait := make(chan error)
	go cli.api.Wait(serveAPIWait)

	// after the daemon is done setting up we can notify systemd api
	notifyReady()

	// Daemon is fully initialized and handling API traffic
	// Wait for serve API to complete
	errAPI := <-serveAPIWait
	c.Cleanup()

	// notify systemd that we're shutting down
	notifyStopping()
	shutdownDaemon(d)

	// Stop notification processing and any background processes
	cancel()

	if errAPI != nil {
		return errors.Wrap(errAPI, "shutting down due to ServeAPI error")
	}

	logrus.Info("Daemon shutdown complete")
	return nil
}
```

What Does it do?

```
1. Load the daemon CLI configuration.
2. Check for deprecated options.
3. Create the API server configuration.
4. If validation is requested, print a configuration confirmation message
and return.
5. Configure proxy environment and daemon logs.
6. Log the startup process.
7. Handle debug and experimental modes.
8. Handle rootless mode with or without RootlessKit integration.
9. Check for root privileges on Linux.
10. Set the default umask.
11. Create the daemon root directory.
12. Create the daemon execution root directory.
13. Handle the creation of PID files.
14. Set sticky bit for files under XDG_RUNTIME_DIR in rootless mode.
15. Initialize the API server.
16. Load listeners for Docker services.
17. Initialize ContainerD.
18. Handle daemon shutdown and traps.
19. Notify API readiness.
20. Create and initialize the plugin store.
21. Initialize middlewares.
22. Create and configure the Docker daemon.
23. Start the metrics server.
24. Create and start the cluster component.
25. Restart autostart containers in swarm mode.
26. Complete daemon initialization.
27. Configure the router and set up routing.
28. Start processing cluster notifications.
29. Set up trap for configuration reload.
30. Start the API server in a goroutine.
31. Notify systemd about readiness.
32. Wait for the API server to complete.
33. Cleanup and shut down the cluster.
34. Notify systemd about stopping.
35. Shutdown the Docker daemon.
36. Stop notification processing and background processes.
37. Handle API server errors.
38. Log shutdown completion.
```

Step 5:

File: cmd/dockerd/daemon.go

```go
func initRouter(opts routerOptions) {
    decoder := runconfig.ContainerDecoder{
        GetSysInfo: func() *sysinfo.SysInfo {
            return opts.daemon.RawSysInfo()
        },
    }

    routers := []router.Router{
        // we need to add the checkpoint router before the container router
or the DELETE gets masked
        checkpointrouter.NewRouter(opts.daemon, decoder),
        container.NewRouter(opts.daemon, decoder,
opts.daemon.RawSysInfo().CgroupUnified),
        image.NewRouter(
            opts.daemon.ImageService(),
            opts.daemon.ReferenceStore,
            opts.daemon.ImageService().DistributionServices().ImageStore,
            opts.daemon.ImageService().DistributionServices().LayerStore,
        ),
        systemrouter.NewRouter(opts.daemon, opts.cluster, opts.buildkit,
opts.features),
        volume.NewRouter(opts.daemon.VolumesService(), opts.cluster),
        build.NewRouter(opts.buildBackend, opts.daemon, opts.features),
        sessionrouter.NewRouter(opts.sessionManager),
        swarmrouter.NewRouter(opts.cluster),
        pluginrouter.NewRouter(opts.daemon.PluginManager()),
        distributionrouter.NewRouter(opts.daemon.ImageService()),
    }

    grpcBackends := []grpcrouter.Backend{}
    for _, b := range []interface{}{opts.daemon, opts.buildBackend} {
        if b, ok := b.(grpcrouter.Backend); ok {
            grpcBackends = append(grpcBackends, b)
        }
    }
    if len(grpcBackends) > 0 {
        routers = append(routers, grpcrouter.NewRouter(grpcBackends...))
    }

    if opts.daemon.NetworkControllerEnabled() {
        routers = append(routers, network.NewRouter(opts.daemon,
opts.cluster))
    }

    if opts.daemon.HasExperimental() {
        for _, r := range routers {
            for _, route := range r.Routes() {
                if experimental, ok := route.(router.ExperimentalRoute); ok
{
                    experimental.Enable()
                }
            }
        }
    }

    opts.api.InitRouter(routers...)
}
```

What Does it do?

```
The `initRouter` function is responsible for initializing the Docker
daemon's routing mechanism, which defines how incoming requests are
handled. It sets up various routers and routes for different Docker
functionalities.

1. Initialize a container decoder for decoding container configurations.
2. Create an array of routers to handle different aspects of Docker:
   - Checkpoint Router: Handles checkpoints for containers.
   - Container Router: Manages container-related operations.
   - Image Router: Handles Docker image-related operations.
   - System Router: Manages system-level operations.
   - Volume Router: Manages Docker volumes.
   - Build Router: Handles build-related operations.
   - Session Router: Manages Docker sessions.
   - Swarm Router: Handles Docker Swarm operations.
   - Plugin Router: Manages Docker plugins.
   - Distribution Router: Handles image distribution operations.
3. Create an array of gRPC backends for gRPC-based routes.
4. If gRPC backends are available, create a gRPC router.
5. If the network controller is enabled, add a network router.
6. If Docker experimental features are enabled, enable experimental routes
in routers.
7. Initialize the Docker API router with the configured routers.
```

Step 6:

File: api/server/router/container/container.go

```go
// NewRouter initializes a new container router
func NewRouter(b Backend, decoder httputils.ContainerDecoder, cgroup2 bool)
router.Router {
    r := &containerRouter{
        backend: b,
        decoder: decoder,
        cgroup2: cgroup2,
    }
    r.initRoutes()
    return r
}
```

What Does it do?

```
- `NewRouter` initializes a new container router.
- It takes a Backend, ContainerDecoder, and a flag indicating whether
Cgroup V2 (cgroup2) is enabled.
- The function creates a containerRouter instance and initializes its
routes.
- The initialized router is returned.
```

Step 7:

File: api/server/router/container/container.go

```
func (r *containerRouter) initRoutes() {
    r.routes = []router.Route{
        // HEAD
        router.NewHeadRoute("/containers/{name:.*}/archive",
r.headContainersArchive),
        // GET
        router.NewGetRoute("/containers/json", r.getContainersJSON),
        router.NewGetRoute("/containers/{name:.*}/export",
r.getContainersExport),
        router.NewGetRoute("/containers/{name:.*}/changes",
r.getContainersChanges),
        router.NewGetRoute("/containers/{name:.*}/json",
r.getContainersByName),
        router.NewGetRoute("/containers/{name:.*}/top",
r.getContainersTop),
        router.NewGetRoute("/containers/{name:.*}/logs",
r.getContainersLogs),
        router.NewGetRoute("/containers/{name:.*}/stats",
r.getContainersStats),
        router.NewGetRoute("/containers/{name:.*}/attach/ws",
r.wsContainersAttach),
        router.NewGetRoute("/exec/{id:.*}/json", r.getExecByID),
        router.NewGetRoute("/containers/{name:.*}/archive",
r.getContainersArchive),
        // POST
        router.NewPostRoute("/containers/create", r.postContainersCreate),
        router.NewPostRoute("/containers/{name:.*}/kill",
r.postContainersKill),
        router.NewPostRoute("/containers/{name:.*}/pause",
r.postContainersPause),
        router.NewPostRoute("/containers/{name:.*}/unpause",
r.postContainersUnpause),
        router.NewPostRoute("/containers/{name:.*}/restart",
r.postContainersRestart),
        router.NewPostRoute("/containers/{name:.*}/start",
r.postContainersStart),
        router.NewPostRoute("/containers/{name:.*}/stop",
r.postContainersStop),
        router.NewPostRoute("/containers/{name:.*}/wait",
r.postContainersWait),
        router.NewPostRoute("/containers/{name:.*}/resize",
r.postContainersResize),
        router.NewPostRoute("/containers/{name:.*}/attach",
r.postContainersAttach),
        router.NewPostRoute("/containers/{name:.*}/copy",
r.postContainersCopy), // Deprecated since 1.8 (API v1.20), errors out
since 1.12 (API v1.24)
        router.NewPostRoute("/containers/{name:.*}/exec",
r.postContainerExecCreate),
        router.NewPostRoute("/exec/{name:.*}/start",
r.postContainerExecStart),
        router.NewPostRoute("/exec/{name:.*}/resize",
r.postContainerExecResize),
        router.NewPostRoute("/containers/{name:.*}/rename",
r.postContainerRename),
        router.NewPostRoute("/containers/{name:.*}/update",
r.postContainerUpdate),
        router.NewPostRoute("/containers/prune", r.postContainersPrune),
        router.NewPostRoute("/commit", r.postCommit),
        // PUT
        router.NewPutRoute("/containers/{name:.*}/archive",
r.putContainersArchive),
        // DELETE
        router.NewDeleteRoute("/containers/{name:.*}", r.deleteContainers),
    }
}
```

What Does it do?

```
- The `initRoutes` function initializes routes for handling various HTTP
methods (HEAD, GET, POST, PUT, DELETE) related to containers in Docker.
- It sets up routes for actions such as container creation, retrieval,
manipulation, and removal.
- The routes are defined using the `router.NewXxxRoute` functions, where
"Xxx" represents the HTTP method (e.g., `router.NewGetRoute`,
`router.NewPostRoute`).
- Each route is associated with a specific handler function for handling
the corresponding HTTP request.
```

Step 8:

File: api/server/router/container/container_routes.go

```
func (s *containerRouter) postContainersCreate(ctx context.Context, w
http.ResponseWriter, r *http.Request, vars map[string]string) error {
    if err := httputils.ParseForm(r); err != nil {
        return err
    }
    if err := httputils.CheckForJSON(r); err != nil {
        return err
    }

    name := r.Form.Get("name")

    config, hostConfig, networkingConfig, err :=
s.decoder.DecodeConfig(r.Body)
    if err != nil {
        if errors.Is(err, io.EOF) {
            return errdefs.InvalidParameter(errors.New("invalid JSON: got
EOF while reading request body"))
        }
        return err
    }
    version := httputils.VersionFromContext(ctx)
    adjustCPUShares := versions.LessThan(version, "1.19")

    // When using API 1.24 and under, the client is responsible for
removing the container
    if hostConfig != nil && versions.LessThan(version, "1.25") {
        hostConfig.AutoRemove = false
    }

    if hostConfig != nil && versions.LessThan(version, "1.40") {
        // Ignore BindOptions.NonRecursive because it was added in API
1.40.
        for _, m := range hostConfig.Mounts {
            if bo := m.BindOptions; bo != nil {
                bo.NonRecursive = false
            }
        }
        // Ignore KernelMemoryTCP because it was added in API 1.40.
        hostConfig.KernelMemoryTCP = 0

        // Older clients (API < 1.40) expects the default to be shareable,
make them happy
        if hostConfig.IpcMode.IsEmpty() {
            hostConfig.IpcMode = container.IPCModeShareable
        }
    }
    if hostConfig != nil && versions.LessThan(version, "1.41") &&
!s.cgroup2 {
        // Older clients expect the default to be "host" on cgroup v1 hosts
        if hostConfig.CgroupnsMode.IsEmpty() {
            hostConfig.CgroupnsMode = container.CgroupnsModeHost
        }
    }

    if hostConfig != nil && versions.LessThan(version, "1.42") {
        for _, m := range hostConfig.Mounts {
            // Ignore BindOptions.CreateMountpoint because it was added in
```

```
API 1.42.
            if bo := m.BindOptions; bo != nil {
                bo.CreateMountpoint = false
            }

            // These combinations are invalid, but weren't validated in API
< 1.42.
            // We reset them here, so that validation doesn't produce an
error.
            if o := m.VolumeOptions; o != nil && m.Type != mount.TypeVolume
{
                m.VolumeOptions = nil
            }
            if o := m.TmpfsOptions; o != nil && m.Type != mount.TypeTmpfs {
                m.TmpfsOptions = nil
            }
            if bo := m.BindOptions; bo != nil {
                // Ignore BindOptions.CreateMountpoint because it was added
in API 1.42.
                bo.CreateMountpoint = false
            }
        }
    }

    if hostConfig != nil && versions.GreaterThanOrEqualTo(version, "1.42")
{
        // Ignore KernelMemory removed in API 1.42.
        hostConfig.KernelMemory = 0
        for _, m := range hostConfig.Mounts {
            if o := m.VolumeOptions; o != nil && m.Type != mount.TypeVolume
{
                return errdefs.InvalidParameter(fmt.Errorf("VolumeOptions
must not be specified on mount type %q", m.Type))
            }
            if o := m.BindOptions; o != nil && m.Type != mount.TypeBind {
                return errdefs.InvalidParameter(fmt.Errorf("BindOptions
must not be specified on mount type %q", m.Type))
            }
            if o := m.TmpfsOptions; o != nil && m.Type != mount.TypeTmpfs {
                return errdefs.InvalidParameter(fmt.Errorf("TmpfsOptions
must not be specified on mount type %q", m.Type))
            }
        }
    }

    if hostConfig != nil && runtime.GOOS == "linux" &&
versions.LessThan(version, "1.42") {
        // ConsoleSize is not respected by Linux daemon before API 1.42
        hostConfig.ConsoleSize = [2]uint{0, 0}
    }

    var platform *specs.Platform
    if versions.GreaterThanOrEqualTo(version, "1.41") {
        if v := r.Form.Get("platform"); v != "" {
            p, err := platforms.Parse(v)
            if err != nil {
                return errdefs.InvalidParameter(err)
            }
            platform = &p
        }
    }

    if hostConfig != nil && hostConfig.PidsLimit != nil &&
*hostConfig.PidsLimit <= 0 {
        // Don't set a limit if either no limit was specified, or
"unlimited" was
        // explicitly set.
        // Both `0` and `-1` are accepted as "unlimited", and historically
any
        // negative value was accepted, so treat those as "unlimited" as
well.
        hostConfig.PidsLimit = nil
    }

    ccr, err := s.backend.ContainerCreate(types.ContainerCreateConfig{
        Name:             name,
        Config:           config,
        HostConfig:       hostConfig,
        NetworkingConfig: networkingConfig,
        AdjustCPUShares:  adjustCPUShares,
        Platform:         platform,
    })
    if err != nil {
        return err
    }

    return httputils.WriteJSON(w, http.StatusCreated, ccr)
}
```

What Does it do?

```
- The `postContainersCreate` function handles the HTTP POST request for
creating a new Docker container.
- It parses and validates the request parameters and payload.
- Extracts container configuration, host configuration, and networking
configuration from the request.
- Adjusts CPU shares based on the Docker API version.
- Modifies host configuration based on Docker API version for backward
compatibility.
- Handles various validation and parameter adjustments based on the Docker
API version.
- Creates a new container using the provided configurations.
- Returns the container creation response as JSON.
```

Step 9:

File: daemon/create.go

```
func (daemon *Daemon) ContainerCreate(params types.ContainerCreateConfig)
(containertypes.CreateResponse, error) {
    return daemon.containerCreate(createOpts{
        params:                  params,
        managed:                 false,
        ignoreImagesArgsEscaped: false})
}
```

What Does it do?

```
- The `ContainerCreate` function is part of the Daemon and is responsible
for creating a new container based on the provided
`types.ContainerCreateConfig`.
- It calls the `containerCreate` function with specific options, including
the creation parameters, managed flag, and image argument handling.
- The `containerCreate` function performs the actual container creation,
and this function acts as a higher-level wrapper.
- The function returns a `containertypes.CreateResponse` containing
information about the created container or an error if the creation fails.
```

Step 10:

File: daemon/create.go

```
func (daemon *Daemon) containerCreate(opts createOpts)
(containertypes.CreateResponse, error) {
    start := time.Now()
    if opts.params.Config == nil {
        return containertypes.CreateResponse{},
errdefs.InvalidParameter(errors.New("Config cannot be empty in order to
create a container"))
    }

    warnings, err := daemon.verifyContainerSettings(opts.params.HostConfig,
opts.params.Config, false)
    if err != nil {
        return containertypes.CreateResponse{Warnings: warnings},
errdefs.InvalidParameter(err)
    }

    if opts.params.Platform == nil && opts.params.Config.Image != "" {
        if img, _ := daemon.imageService.GetImage(opts.params.Config.Image,
opts.params.Platform); img != nil {
            p := maximumSpec()
            imgPlat := v1.Platform{
                OS:           img.OS,
                Architecture: img.Architecture,
                Variant:      img.Variant,
            }

            if !images.OnlyPlatformWithFallback(p).Match(imgPlat) {
                warnings = append(warnings, fmt.Sprintf("The requested
image's platform (%s) does not match the detected host platform (%s) and no
specific platform was requested", platforms.Format(imgPlat),
platforms.Format(p)))
            }
        }
    }

    err = verifyNetworkingConfig(opts.params.NetworkingConfig)
    if err != nil {
        return containertypes.CreateResponse{Warnings: warnings},
errdefs.InvalidParameter(err)
    }

    if opts.params.HostConfig == nil {
        opts.params.HostConfig = &containertypes.HostConfig{}
    }
    err = daemon.adaptContainerSettings(opts.params.HostConfig,
opts.params.AdjustCPUShares)
    if err != nil {
        return containertypes.CreateResponse{Warnings: warnings},
errdefs.InvalidParameter(err)
    }

    ctr, err := daemon.create(opts)
    if err != nil {
        return containertypes.CreateResponse{Warnings: warnings}, err
    }
    containerActions.WithValues("create").UpdateSince(start)

    if warnings == nil {
        warnings = make([]string, 0) // Create an empty slice to avoid
https://github.com/moby/moby/issues/38222
    }

    return containertypes.CreateResponse{ID: ctr.ID, Warnings: warnings},
nil
}
```

What Does it do?

```
- The `containerCreate` function is part of the Daemon and is responsible
for creating a new container with the provided options.
```

```
- It performs various checks and validations before creating the container.
- Checks if the provided configuration is valid and not empty.
- Verifies container settings, including host configuration and networking
configuration.
- Validates the platform of the requested image if not specified
explicitly.
- Adapts container settings, such as CPU shares, if needed.
- Calls the `create` function to actually create the container.
- Records metrics related to container creation.
- Returns a `containertypes.CreateResponse` with the container ID and any
warnings, or an error if the creation fails.
```

Step 11:

File: daemon/create.go

```go
func (daemon *Daemon) verifyContainerSettings(hostConfig
*containertypes.HostConfig, config *containertypes.Config, update bool)
(warnings []string, err error) {
    // First perform verification of settings common across all platforms.
    if err = validateContainerConfig(config); err != nil {
        return warnings, err
    }
    if err := validateHostConfig(hostConfig); err != nil {
        return warnings, err
    }

    // Now do platform-specific verification
    warnings, err = verifyPlatformContainerSettings(daemon, hostConfig,
update)
    for _, w := range warnings {
        logrus.Warn(w)
    }
    return warnings, err
}
```

What Does it do?

```
- The `verifyContainerSettings` function is part of the Daemon and is
responsible for verifying container settings before creating or updating a
container.
- It first performs verification of settings that are common across all
platforms.
  - It validates the container configuration using
`validateContainerConfig`.
```

```
  - It validates the host configuration using `validateHostConfig`.
- After common verifications, it proceeds to perform platform-specific
verification using `verifyPlatformContainerSettings`.
- The function collects any warnings generated during the verification
process and logs them.
- It returns the collected warnings and an error if any validation fails.
```