What are Container Exit Codes

Exit codes are used by container engines, when a container terminates, to report why it was terminated.

If you are a Kubernetes user, container failures are one of the most common causes of pod exceptions, and understanding container exit codes can help you get to the root cause of pod failures when troubleshooting.

The most common exit codes used by containers are:

CODE #	NAME	WHAT IT MEANS
Exit Code 0	Purposely stopped	Used by developers to indicate that the container was automatically stopped
Exit Code 1	Application error	Container was stopped due to application error or incorrect reference in the image specification
Exit Code 125	Container failed to run error	The docker run command did not execute successfully
Exit Code 126	Command invoke error	A command specified in the image specification could not be invoked
Exit Code 127	File or directory not found	File or directory specified in the image specification was not found
Exit Code 128	Invalid argument used on exit	Exit was triggered with an invalid exit code (valid codes are integers between 0-255)
Exit Code 134	Abnormal termination (SIGABRT)	The container aborted itself using the abort() function.
Exit Code 137	Immediate termination (SIGKILL)	Container was immediately terminated by the operating system via SIGKILL signal
Exit Code 139	Segmentation fault (SIGSEGV)	Container attempted to access memory that was not assigned to it and was terminated
Exit Code 143	Graceful termination (SIGTERM)	Container received warning that it was about to be terminated, then terminated
Exit Code 255	Exit Status Out Of Range	Container exited, returning an exit code outside the acceptable range, meaning the cause of the error is not known

Below we'll explain how to troubleshoot failed containers on a self-managed host and in Kubernetes, and provide more details on all of the exit codes listed above.

This is part of an extensive series of guides about Observability.

The Container Lifecycle

To better understand the causes of container failure, let's discuss the lifecycle of a container first. Taking Docker as an example – at any given time, a Docker container can be in one of several states:

- **Created** – the Docker container is created but not started yet (this is the status after running docker create, but before actually running the container)
- **Up** – the Docker container is currently running. This means the operating system process managed by the container is running. This happens when you use the commands docker start or docker run can happen using docker start or docker run.
- **Paused** – the container process was running, but Docker purposely paused the container. Typically this happens when you run the Docker pause command
- **Exited** – the Docker container has been terminated, usually because the container's process was killed

When a container reaches the **Exited** status, Docker will report an exit code in the logs, to inform you what happened to the container that caused it to shut down.

Understanding Container Exit Codes

Below we cover each of the exit codes in more detail.

Exit Code 0: Purposely Stopped

Exit Code 0 is triggered by developers when they purposely stop their container after a task completes. Technically, Exit Code 0 means that the foreground process is not attached to a specific container.

What to do if a container terminated with Exit Code 0?

1. Check the container logs to identify which library caused the container to exit
2. Review the code of the existing library and identify why it triggered Exit Code 0, and whether it is functioning correctly

Exit Code 1: Application Error

Exit Code 1 indicates that the container was stopped due to one of the following:

- **An application error** – this could be a simple programming error in code run by the container, such as "divide by zero", or advanced errors related to the runtime environment, such as Java, Python, etc
- **An invalid reference** – this means the image specification refers to a file that does not exist in the container image

What to do if a container terminated with Exit Code 1?

1. Check the container log to see if one of the files listed in the image specification could not be found. If this is the issue, correct the image specification to point to the correct path and filename.
2. If you cannot find an incorrect file reference, check the container logs for an application error, and debug the library that caused the error.

Exit Code 125: Container Failed to Run

Exit Code 125 means that the command is used to run the container. For example `docker run` was invoked in the system shell but did not execute successfully. Here are common reasons this might happen:

- An undefined flag was used in the command, for example `docker run --abcd`
- The user-defined in the image specification does not have sufficient permissions on the machine
- Incompatibility between the container engine and the host operating system or hardware

What to do if a container terminated with Exit Code 125?

1. Check if the command used to run the container uses the proper syntax
2. Check if the user running the container, or the context in which the command is executed in the image specification, has sufficient permissions to create containers on the host
3. If your container engine provides other options for running a container, try them. For example, in Docker,

try `docker start` instead of `docker run`

4. Test if you are able to run other containers on the host using the same username or context. If not, reinstall the container engine, or resolve the underlying compatibility issue between the container engine and the host setup

Exit Code 126: Command Invoke Error

Exit Code 126 means that a command used in the container specification could not be invoked. This is often the cause of a missing dependency or an error in a continuous integration script used to run the container.

What to do if a container terminated with Exit Code 126?

1. Check the container logs to see which command could not be invoked
2. Try running the container specification without the command to ensure you isolate the problem
3. Troubleshoot the command to ensure you are using the correct syntax and all dependencies are available
4. Correct the container specification and retry running the container

Exit Code 127: File or Directory Not Found

Exit Code 127 means a command specified in the container specification refers to a non-existent file or directory.

What to do if a container terminated with Exit Code 127?

Same as Exit Code 126, identify the failing command and make sure you reference a valid filename and file path available within the container image.

guide to exit code 127.

Exit Code 128: Invalid Argument Used on Exit

Exit Code 128 means that code within the container triggered an exit command, but did not provide a valid exit code. The Linux `exit` command only allows integers between 0-255, so if the process was exited with, for example, `exit code 3.5`, the logs will report Exit Code 128.

What to do if a container terminated with Exit Code 128?

1. Check the container logs to identify which library caused the container to exit.
2. Identify where the offending library uses the `exit` command, and correct it to provide a valid exit code.

Exit Code 134: Abnormal Termination (SIGABRT)

Exit Code 134 means that the container abnormally terminated itself, closed the process and flushed open streams. This operation is irreversible, like SIGKILL (see Exit Code 137 below). A process can trigger SIGABRT by doing one of the following:

Calling the `abort()` function in the `libc` library
Calling the `assert()` macro, used for debugging. The process is then aborted if the assertion is false.

What to do if a container terminated with Exit Code 134?

1. Check container logs to see which library triggered the SIGABRT signal
2. Check if process abortion was planned (for example because the library was in debug mode), and if not, troubleshoot the library and modify it to avoid aborting the container.

Exit Code 137: Immediate Termination (SIGKILL)

Exit Code 137 means that the container has received a SIGKILL signal from the host operating system. This signal instructs a process to terminate immediately, with no grace period. This can be either:

- Triggered when a container is killed via the container engine, for example when using the `docker kill` command
- Triggered by a Linux user sending a `kill -9` command to the process
- Triggered by Kubernetes after attempting to terminate a container and waiting for a grace period of 30 seconds (by default)
- Triggered automatically by the host, usually due to running out of memory. In this case, the `docker inspect` command will indicate an `OOMKilled` error.

guide to exit code 137.

What to do if a container terminated with Exit Code 137?

1. Check logs on the host to see what happened prior to the container terminating, and whether it previously received a SIGTERM signal (graceful termination) before receiving SIGKILL
2. If there was a prior SIGTERM signal, check if your container process handles SIGTERM and is able to gracefully terminate
3. If there was no SIGTERM and the container reported an `OOMKilled` error, troubleshoot memory issues on the host

Learn more in our detailed guide to the SIGKILL signal >>

Exit Code 139: Segmentation Fault (SIGSEGV)

Exit Code 139 means that the container received a SIGSEGV signal from the operating system. This indicates a segmentation error – a memory violation, caused by a container trying to access a memory location to which it does not have access. There are three common causes of SIGSEGV errors:

1. **Coding error**—container process did not initialize properly, or it tried to access memory through a pointer to previously freed memory
2. **Incompatibility between binaries and libraries**—container process runs a binary file that is not compatible with a shared library, and thus may try to access inappropriate memory addresses
3. **Hardware incompatibility or misconfiguration**—if you see multiple segmentation errors across multiple libraries, there may be a problem with memory subsystems on the host or a system configuration issue

What to do if a container terminated with Exit Code 139?

1. Check if the container process handles SIGSEGV. On both Linux and Windows, you can handle a container's response to segmentation violations. For example, the container can collect and report a stack trace
2. If you need to further troubleshoot SIGSEGV, you may need to set the operating system to allow programs to run even after a segmentation fault occurs, to allow for investigation and debugging. Then, try to intentionally

cause a segmentation violation and debug the library causing the issue

3. If you cannot replicate the issue, check memory subsystems on the host and troubleshoot memory configuration

Learn more in our detailed guide to the SIGSEGV signal >>

Exit Code 143: Graceful Termination (SIGTERM)

Exit Code 143 means that the container received a SIGTERM signal from the operating system, which asks the container to gracefully terminate, and the container succeeded in gracefully terminating (otherwise you will see Exit Code 137). This exit code can be:

- Triggered by the container engine stopping the container, for example when using the `docker stop` or `docker-compose` down commands
- Triggered by Kubernetes setting a pod to **Terminating** status, and giving containers a 30 second period to gracefully shut down

What to do if a container terminated with Exit Code 143?

Check host logs to see the context in which the operating system sent the SIGTERM signal. If you are using Kubernetes, check the kubelet logs to see if and when the pod was shut down.

In general, Exit Code 143 does not require troubleshooting. It means the container was properly shut down after being instructed to do so by the host.

Learn more in our detailed guide to the SIGTERM signal >>

Exit Code 1: Application Error

Exit Code 1 indicates that the container was stopped due to one of the following:

- **An application error** – this could be a simple programming error in code run by the container, such as “divide by zero”, or advanced errors related to the runtime environment, such as Java, Python, etc

- **An invalid reference** – this means the image specification refers to a file that does not exist in the container image

What to do if a container terminated with Exit Code 1?

1. Check the container log to see if one of the files listed in the image specification could not be found. If this is the issue, correct the image specification to point to the correct path and filename.
2. If you cannot find an incorrect file reference, check the container logs for an application error, and debug the library that caused the error.

Exit Code 125

Exit Code 125: Container Failed to Run

Exit Code 125 means that the command is used to run the container. For example `docker run` was invoked in the system shell but did not execute successfully. Here are common reasons this might happen:

- An undefined flag was used in the command, for example `docker run --abcd`
- The user-defined in the image specification does not have sufficient permissions on the machine
- Incompatibility between the container engine and the host operating system or hardware

What to do if a container terminated with Exit Code 125?

1. Check if the command used to run the container uses the proper syntax
2. Check if the user running the container, or the context in which the command is executed in the image specification, has sufficient permissions to create containers on the host
3. If your container engine provides other options for running a container, try them. For example, in Docker, try `docker start` instead of `docker run`
4. Test if you are able to run other containers on the host using the same username or context. If not, reinstall the container engine, or resolve the underlying compatibility issue between the container engine and the host

setup

Exit Code 126: Command Invoke Error

Exit Code 126 means that a command used in the container specification could not be invoked. This is often the cause of a missing dependency or an error in a continuous integration script used to run the container.

What to do if a container terminated with Exit Code 126?

1. Check the container logs to see which command could not be invoked
2. Try running the container specification without the command to ensure you isolate the problem
3. Troubleshoot the command to ensure you are using the correct syntax and all dependencies are available
4. Correct the container specification and retry running the container

Exit Code 127: File or Directory Not Found

Exit Code 127 means a command specified in the container specification refers to a non-existent file or directory.

What to do if a container terminated with Exit Code 127?

Same as Exit Code 126 above, identify the failing command and make sure you reference a valid filename and file path available within the container image.

Exit Code 128: Invalid Argument Used on Exit

Exit Code 128 means that code within the container triggered an exit command, but did not provide a valid exit code. The Linux `exit` command only allows integers between 0-255, so if the process was exited with, for example, `exit code 3.5`, the logs will report Exit Code 128.

What to do if a container terminated with Exit Code 128?

1. Check the container logs to identify which library caused the container to exit.

2. Identify where the offending library uses the `exit` command, and correct it to provide a valid exit code.

Exit Code 134: Abnormal Termination (SIGABRT)

Exit Code 134 means that the container abnormally terminated itself, closed the process and flushed open streams. This operation is irreversible, like SIGKILL (see Exit Code 137 below). A process can trigger SIGABRT by doing one of the following:

Calling the `abort()` function in the `libc` library
Calling the `assert()` macro, used for debugging. The process is then aborted if the assertion is false.

What to do if a container terminated with Exit Code 134?

1. Check container logs to see which library triggered the SIGABRT signal
2. Check if process abortion was planned (for example because the library was in debug mode), and if not, troubleshoot the library and modify it to avoid aborting the container.

Exit Code 137: Immediate Termination (SIGKILL)

Exit Code 137 means that the container has received a SIGKILL signal from the host operating system. This signal instructs a process to terminate immediately, with no grace period. This can be either:

- Triggered when a container is killed via the container engine, for example when using the `docker kill` command
- Triggered by a Linux user sending a `kill -9` command to the process
- Triggered by Kubernetes after attempting to terminate a container and waiting for a grace period of 30 seconds (by default)
- Triggered automatically by the host, usually due to running out of memory. In this case, the `docker inspect` command will indicate an `OOMKilled` error.

What to do if a container terminated with Exit Code 137?

1. Check logs on the host to see what happened prior to the container terminating, and whether it previously received a SIGTERM signal (graceful termination) before receiving SIGKILL
2. If there was a prior SIGTERM signal, check if your container process handles SIGTERM and is able to gracefully terminate
3. If there was no SIGTERM and the container reported an `OOMKilled` error, troubleshoot memory issues on the host

Learn more in our detailed guide to the SIGKILL signal >>

Exit Code 139: Segmentation Fault (SIGSEGV)

Exit Code 139 means that the container received a SIGSEGV signal from the operating system. This indicates a segmentation error – a memory violation, caused by a container trying to access a memory location to which it does not have access. There are three common causes of SIGSEGV errors:

1. **Coding error**—container process did not initialize properly, or it tried to access memory through a pointer to previously freed memory
2. **Incompatibility between binaries and libraries**—container process runs a binary file that is not compatible with a shared library, and thus may try to access inappropriate memory addresses
3. **Hardware incompatibility or misconfiguration**—if you see multiple segmentation errors across multiple libraries, there may be a problem with memory subsystems on the host or a system configuration issue

What to do if a container terminated with Exit Code 139?

1. Check if the container process handles SIGSEGV. On both Linux and Windows, you can handle a container's response to segmentation violations. For example, the container can collect and report a stack trace
2. If you need to further troubleshoot SIGSEGV, you may need to set the operating system to allow programs to run even after a segmentation fault occurs, to allow for investigation and debugging. Then, try to intentionally cause a segmentation violation and debug the library causing the issue
3. If you cannot replicate the issue, check memory subsystems on the host and troubleshoot memory

configuration

Learn more in our detailed guide to the SIGSEGV signal >>

Exit Code 143: Graceful Termination (SIGTERM)

Exit Code 143 means that the container received a SIGTERM signal from the operating system, which asks the container to gracefully terminate, and the container succeeded in gracefully terminating (otherwise you will see Exit Code 137). This exit code can be:

- Triggered by the container engine stopping the container, for example when using the `docker stop` or `docker-compose` down commands
- Triggered by Kubernetes setting a pod to **Terminating** status, and giving containers a 30 second period to gracefully shut down

What to do if a container terminated with Exit Code 143?

Check host logs to see the context in which the operating system sent the SIGTERM signal. If you are using Kubernetes, check the kubelet logs to see if and when the pod was shut down.

In general, Exit Code 143 does not require troubleshooting. It means the container was properly shut down after being instructed to do so by the host.

Learn more in our detailed guide to the SIGTERM signal >>

Exit Code 255: Exit Status Out Of Range

When you see exit code 255, it implies the main entrypoint of a container stopped with that status. It means that the container stopped, but it is not known for what reason.

What to do if a container terminated with Exit Code 255?

1. If the container is running in a virtual machine, first try removing overlay networks configured on the virtual machine and recreating them.

2. If this does not solve the problem, try deleting and recreating the virtual machine, then rerunning the container on it.

3. Failing the above, bash into the container and examine logs or other clues about the entrypoint process and why it is failing.

Which Kubernetes Errors are Related to Container Exit Codes?

Whenever containers fail within a pod, or Kubernetes instructs a pod to terminate for any reason, containers will shut down with exit codes. Identifying the exit code can help you understand the underlying cause of a pod exception.

You can use the following command to view pod errors: `kubectl describe pod [name]`

The result will look something like this:

```
Containers:
kubedns:
Container ID: ...
Image: ...
Image ID: ...
Ports: ...
Host Ports: ...
Args: ...
State: Running
   Started: Fri, 15 Oct 2021 12:06:01 +0800
Last State: Terminated
   Reason: Error
   Exit Code: 255
   Started: Fri, 15 Oct 2021 11:43:42 +0800
   Finished: Fri, 15 Oct 2021 12:05:17 +0800
Ready: True
Restart Count: 1
```

Use the Exit Code provided by `kubectl` to troubleshoot the issue:

- **If the Exit Code is 0** – the container exited normally, no troubleshooting is required
- **If the Exit Code is between1-128** – the container terminated due to an internal error, such as a missing or invalid command in the image specification

- **If the Exit Code is between 129-255** – the container was stopped as the result of an operating signal, such as SIGKILL or SIGINT
- **If the Exit Code was** `exit(-1)` or another value outside the 0-255 range, `kubectl` translates it to a value within the 0-255 range.

Refer to the relevant section above to see how to troubleshoot the container for each exit code.

Troubleshooting Kubernetes Pod Termination with Komodor

As a Kubernetes administrator or user, pods or containers terminating unexpectedly can be a pain and can result in severe production issues. The troubleshooting process in Kubernetes is complex and, without the right tools, can be stressful, ineffective, and time-consuming.

Some best practices can help minimize the chances of container failure affecting your applications, but eventually, something will go wrong—simply because it can.

This is the reason why we created Komodor, a tool that helps dev and ops teams stop wasting their precious time looking for needles in (hay)stacks every time things go wrong.

Acting as a single source of truth (SSOT) for all of your k8s troubleshooting needs, Komodor offers:

- **Change intelligence:** Every issue is a result of a change. Within seconds we can help you understand exactly who did what and when.
- **In-depth visibility:** A complete activity timeline, showing all code and config changes, deployments, alerts, code diffs, pod logs and etc. All within one pane of glass with easy drill-down options.
- **Insights into service dependencies:** An easy way to understand cross-service changes and visualize their ripple effects across your entire system.
- **Seamless notifications:** Direct integration with your existing communication channels (e.g., Slack) so you'll have all the information you need, when you need it.