Kill Switch Design Pattern for Microservice Architectures on Internet of Things Devices

by

David Lennick

A thesis submitted to the
School of Graduate and Postdoctoral Studies in
partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Faculty of Engineering and Applied Science

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

December 2020

Thesis Examination Information

Submitted by: David Lennick
Master of Science in Computer Science
Thesis Title: Kill Switch Design Pattern for Microservice Architectures on Internet of Things Devices

An oral defense of this thesis took place on December 11, 2020 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Faisal Quereshi
Research Supervisor	Dr. Ramiro Liscano
Research Co-supervisor	Dr. Akramul Azim
Examining Committee Member	Dr. Jeremy Bradbury
Thesis Examiner	Dr. Patrick Hung

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

Containers and virtual machines are being adopted to develop embedded Linux Internet-of-Things applications. Consumer Internet-of-Things devices have been notoriously insecure due to loss of continued software support. To help prevent this, we propose the 'kill switch' pattern. By defining operation levels for microservice-based virtualized application components and their respective communication paths, application functionality can be dynamically modified to an essential state. This thesis contributes: a formalized definition of the proposed design pattern for virtualized microservice applications; and an algorithm for handling the operation level mode change. We illustrate with three example realizations: a generic microservice-based model-view-controller application, an example system utilizing the Suricata intrusion detection system to generate events, and a modified Docker Engine implementation. Use cases, scenarios, and general application design processes are discussed, with suggested areas of future work.

Keywords: microservice architecture; internet of things; design patterns; containers

Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the University of Ontario Institute of Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

Statement of Contributions

The initial concept of this work was presented at the 2020 IEEE International Symposium on Real-Time Computing, and published in the conference proceedings as follows:

D. Lennick, A. Azim and R. Liscano, "Container-Based Internet-of-Things Architecture Pattern: Kill Switch," 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), Nashville, TN, USA, 2020, pp. 74-78, doi: 10.1109/ISORC49007.2020.00020.

Dedication

For my family,
without whom I would not be who I am

For my friends,
without whom I would not have become myself

For those who came before,
to whom we all owe

And for those who follow,
to whom we owe all

Acknowledgements

I acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

I would like to thank my supervisors Dr. Ramiro Liscano and Dr. Akramul Azim for their patience, support, and guidance.

Contents

List of Figures

Listings

Acronyms

API Application Programming Interface.

HTTP Hypertext Transfer Protocol.

HV Hypervisor.

IDS Intrusion Detection System.

IoT Internet of Things.

MSA Microservice Architecture.

MVC Model View Controller.

OS Operating System.

REST Representative State Transfer.

SOA Service-Oriented Architecture.

VM Virtual Machine.

1. Introduction

For some given system, there should be a way to stop the system from causing harm in an emergency situation. For example, treadmills have a rope that attaches the user to a kill switch. If the rope is pulled out, such as when the user falls, the treadmill stops and prevents the user from further injury. This principle is also widely used in machining tools, such as with lathes or milling machines. A kill switch is designed to prevent such harm, or stop unintended effects of operation of some system.

Traditionally, embedded systems were designed for secured environments like on-site industrial automation systems. With the recent increase of popularity of automation systems with consumers, embedded systems have been tasked with higher amounts of processing, while maintaining persistent connections with remote cloud systems. This has increased the demand for edge systems, and solutions revolving around the Internet of Things (IoT). As such, the rigor and standards required in an industrial automation system have been cast aside in favour of a less rigorous 'consumer-focused' approach.

Unfortunately, there have been several instances of these Internet-connected devices causing harm. Examples include:

- Nest Thermostats losing functionality in winter [1],
- devices infected with the Mirai botnet attacking large DNS providers which disabled a large part of the Internet on the east coast [2],
- Jeep vehicles being taken offline remotely [3],
- and smart home hubs allowing remote code execution [4]

Should the trend of automating consumer-based products via this method continue, the safety of end users is at serious risk. Furthermore, a major secu-

rity concern is that once a device manufacturer no longer updates the software for a system, it will eventually be rendered useless or dangerous as exploits are discovered. The ability to rapidly deploy updates to fleets of systems, or entire consumer lines of products, has been a source of both interest and contention for IoT adoption.

1.1 Motivation

Containers are an increasingly popular method of adding significant isolation capabilities to various types of systems. Containers, or operating-system-level virtualization, differ from 'regular' virtual machines (VM) and hypervisors (HV). In contrast to HVs, which provide abstractions for subdivision of 'raw' base system resources to VMs running full operating systems, containers leverage isolation techniques provided by operating system facilities. This allows for 'lighter' virtualization, especially when considering the reuse of base operating system kernel resources by the containers themselves. As a result, containers can be optimized to require less system resources in comparison to a VM. For example, virtualization of an entire guest operating system or guest filesystem is not required with containers. Furthermore, techniques such as copy-on-write filesystems can be implemented to further reduce their resource requirements.

Containers and associated platforms are rapidly being adopted for enterprise cloud workloads and application development. Some examples of current container platform implementations include Linux Containers (LXC[1]), and Docker[2]. According to a Jan 2018 RightScale survey [5], Docker container adoption has increased from 35% in 2017 to 49% in 2018. As a result of this adoption, a robust ecosystem has developed around Docker.

The rapid adoption of containers can be attributed to the value proposition of microservice architecture (MSA). MSA, as described in [6] and [7], is an adaptation of service-oriented architecture (SOA). In comparison to SOA, and in contrast to 'monolithic' architecture, MSA divides an application into "a suite of small services, each running in its own process and communicating with

[1] `https://linuxcontainers.org/`

[2] `https://www.docker.com/resources/what-container`

lightweight mechanisms, often an hypertext transfer protocol (HTTP) resource application programming interface (API). These services are built around business capabilities and independently deployable by fully automated deployment machinery" [6].

Combined IoT markets are estimated to grow from $235 billion in 2017 to $520 billion in 2021 [8], with top concerns of adoption being security at 45%, and difficulties integrating IT with operational technology at 34% [9]. The combination of resource isolation, security isolation, and evolving development operations practices make containers particularly well-suited for Internet-of-Things (IoT) devices. For example, an IoT device can deploy multiple containers on the same device – this allows for higher utilization of bare-metal resources, as well as edge or fog computing application scenarios. balenaEngine[3], a fork of the Docker engine, is an example of an IoT-focused container engine. HypriotOS[4] and balenaOS[5] are examples of Linux operating systems focused on running container engines for embedded and IoT devices. Also, Docker has recently added multi-architecture build support directly into their image pipeline. This includes support for ARM devices, amongst others. [10]

Rancher Labs have developed a distribution of Kubernetes[6] called k3s targeting resource-constrained environments. Like Kubernetes, there are many methods and architecture styles for deploying k3s.

The advantages of leveraging virtualization in IoT systems present new possibilities. Due to how applications are developed when using "proper" container application architecture patterns, such as microservice architecture (MSA), subsets of critical functionality can be kept isolated in separate containers. This can allow for all non-essential containers to be shut down or killed. This results in serious considerations for MSA pattern development. As evidenced by adoption of containers, the infrastructure of systems are becoming synonymous with application architectures. By considering the software packaging systems as part of the definition of the application architecture itself, we have identified a pattern – the kill switch pattern. When a developer is creating a MSA application, functionality can be separated out into discrete services. By

[3] https://www.balena.io/engine/
[4] https://blog.hypriot.com/
[5] https://www.balena.io/os/
[6] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

identifying and isolating essential application functionality, and the services associated with this required functionality, an application can be developed such that it removes unnecessary application functionality. For example, this is useful in a security scenario; surface area of an application can be reduced in a compromised state, while retaining critical functionality.

We specifically target this design pattern at IoT systems for a few reasons. Given the operating environment of IoT systems, compute resources are limited. An IoT system does not have the luxury of scaling out functionality, or the processing power to analyze large amounts of data. In the situation of an external threat, it is much easier to lock the system down than it is to prevent attacks. Another reason is if an IoT system is battery operated, one would expect that the system would reduce 'nice-to-have' functionality for the sake of battery life. Also, IoT systems are notoriously difficult to keep updated, especially if a vendor discontinues support. This approach would allow development of a 'fail-safe' mode, preserving baseline functionality, or allow for the deployment of general-purpose containers. Alternatively, this approach also lends itself to the deployment of general-purpose services that are much easier to keep updated.

By implementing the kill switch pattern, device manufacturers would gain several advantages. First, devices would be easier to maintain due to the decomposition of services as defined by microservice architecture. Second, clear dependency tracing would be achieved, which is a serious limitation of microservice architecture. Third, the device would be able to utilize the kill switch architecture to reduce application functionality to a desired state - that is, should the application architect design the application to have several levels, desired service of IoT devices would be able to continue in a reduced state should the application require it.

Likewise, end-users would gain several advantages via implementation of the kill switch pattern. Should an event cause 'smart' functionality of the device to fail, the device could be developed such that base functionality continues. Furthermore, in conjunction with a mode change mechanism,

This approach in not limited to containers alone. Any system utilizing MSA can benefit from this pattern regardless of deployment method, such as in IoT

projects leveraging on-device virtual machines such as Project ACRN[7].

1.2 Contributions

This thesis contributes:

- a formalized definition of the design pattern for virtualized microservice applications,
- an algorithm for handling a mode change, utilizing the pattern,
- an analysis of the pattern,
- example architectures:
  - a generic microservice-based model-view-controller application
  - an example system utilizing the Suricata intrusion detection system to generate event
  - an example implementation of the mode switch in the Docker engine API

1.3 Outline

The remainder of the thesis is organized as follows. Chapter 2 provides background information and related work on virtualization, containers, container patterns, cloud patterns, microservice architecture, IoT application architectures, and safety-critical systems patterns. Chapter 3 defines the pattern using set and graph definitions, defines an algorithm for a mode change, and analyses the pattern. Chapter 4 demonstrates realizations of the pattern, and is supplemented by discussion. Chapter 5 provides a brief conclusion, with suggested areas of future work.

[7] https://projectacrn.org/

2. Background and Related Work

In this section, we review required background information, and works related to the kill switch pattern. We begin with a background on hypervisors and virtual machines, a background on containers and related work of their patterns, a background of microservice architecture and related work in the area of cloud design patterns, a background of IoT, and background on safety-critical systems. We then discuss related works more explicitly, and the relations they have to the kill switch pattern.

2.1 Hypervisors and Virtual Machines

In order to discuss virtualization methods, some concept overviews are required.

- Bare-Metal: Bare-metal (or native) refers to a set of dedicated hardware that an application, operating system (OS), or other software, may run on — i.e., there is no method of virtualization. [11]
- Virtual Machine (VM): A VM is an isolated system that runs in the same way a non-virtualized machine would, except it can only access and utilize 'virtualized' resources provisioned to it by some hardware abstraction layer. These resources appear normally to the VM.
- Hypervisor (HV): A hypervisor is a hardware abstraction layer platform that manages and distributes resources to virtualized guest components. Resources include processor, memory, hard-drive space, or peripherals. A hypervisor may run directly on bare-metal (Type 1), or in an OS (Type 2) [11]

- Host: A host is a system or platform for a virtualized component to run on. The term is often used interchangeably to refer to the bare-metal instance a hypervisor or other platform runs on, the OS a virtualized component runs on, or other similar concepts. For example, a hypervisor is a host for guest virtualized components.
- Guest: A guest is a virtualized component which is managed by or run on a host, and is provided some abstracted host resources.

Maximum utilization of bare-metal compute resources is a significant concern from a cost and efficiency viewpoint. By using virtualization concepts, bare-metal resources can be divided and utilized by a set of virtualized components [12] [13]. Typical use cases include isolated development environments on a developer's workstation, or deployment of multiple virtualized application servers across clustered bare-metal resources managed by a hypervisor.

Virtualization also allows for easier and cost-effective implementation of high-availability, fault-tolerant, scalable, systems. Furthermore, local area networks (LANs) and bare-metal network appliances such as switches, routers, etc., can be virtualized as well. This results in implementations like virtual LANS (VLANs) and software-defined networks (SDNs), respectively. These techniques can be used to further isolate and manage communications between virtualized components [12].

2.2 Containers

The current idiomatic definition of a container refers to the concept put forward by Soltesz et al. [13], which combine two separate ideas: resource containers, a method of subdividing system resources; and security containers, a method of isolating processes from the base operating system. In Linux, the usual way of implementing security container functionality is with namespaces, and resource containers with control groups.

Containers are a variation of the sandbox concept, as described by Goldberg, Wagner, Thome, and Brewer in 1996 [14], and have been implemented in a variety of ways. Some notable instances include chroot jails, FreeBSD jails as described by Kamp and Watson in 2000 [15], Security-Enhanced Linux's

(SELinux) implementation as described by Loscocco and Smalley in 2001 [16], Solaris Zones as described by Price and Tucker in 2004 [17], amongst others.

A namespace is, according to the Linux Programmer's Manual [18], an abstractions of a global system resource that only allow namespace process members isolated access to the global resource instance. Of particular interest are user namespaces, which were introduced to the Linux kernel in 2008. According to the Linux Programmer's Manual [19], user namespaces allow for different 'perspectives' relating to security-related identifiers and attributes (user IDs, group IDs, root directory, capabilities, etc.).

Container resource management of host Linux operating system resources are performed using control groups (cgroups). Cgroups are hierarchies of processes [20]. These hierarchies are then able to have resources limited or altered based on each type using associated subsystems/resource controllers. Example resources controlled by these subsystems include CPU, memory, number of processes, and process suspension/resumption.

2.2.1 Container Design Patterns

Burns and Oppenheimer describe and formalize a set of emergent design patterns for container-based systems in [21]. They assert there are three types of patterns:

- single-container management patterns
- single-node, multi-container application patterns
- multi-node application patterns

Burns and Oppenheimer assert [21] that containers provide many of the benefits of object-oriented programming. The authors also assert that using containers as units have the following advantages: resource accounting and allocation, packaging, reuse, inherent failure containment boundary, and ease of deployment.

Single container patterns revolve around one container on one node. The patterns involve the following components: a management interface for the

container, the boundary of a container, and lifecycle of a container. The management interface can be simple (running, stopping, pausing), or extended using an exposed API internal to the container. Burns and Oppenheimer assert that similar to objects in object-oriented programming, containers can expose as much information as a developer would like. Furthermore, lifecycle operations can be as simple as immediately killing containers, to complex series of events such as allowing in-flight transactions to complete, closing connections, and gracefully stopping. [21]

Single-node, multi-container application patterns involve one system hosting multiple containers. The authors identify three patterns: sidecar, ambassador, and adaptor. [21]

The sidecar pattern involves a main container, and a sidecar container that extends or augments the functionality of the main container. The authors provide an example web server serving static content, and a sidecar that periodically updates the static content from a Git repository.

The ambassador pattern involves a main container and a container used to proxy communication to the main container.

The adaptor pattern involves taking heterogeneous interfaces for different containers or services, and providing some method or strategy for homogenizing the interface to the application as a whole.

Multi-node application patterns involve multiple nodes running $1..n$ containers. Often, these reflect typical distributed system patterns. However, their implementation vary based on the characteristics of containers. [21]

The leader election pattern involves using a set of application containers requiring leader election. In distributed systems, a replica set may elect a 'leader' to handle functionality, and are traditionally implemented at the application layer via a library. However, Burns and Oppenheimer assert that the implementation is difficult. Instead, leader-election containers present a simplified HTTP API to one another for use in leader election. [21]

The work queue pattern allows an application developer to treat a specialized container as a job, to allow for tasks to execute in a language-agnostic fashion. Using this approach, the queue itself can be implemented in any generic language or framework. [21]

The scatter/gather pattern involves a root node, which fans a request out

to a set of leaf processing nodes. Given a generic interface, a merge container that manages client interaction, can send the request to 'leaf' containers in parallel. The merge container then gathers and merges the results from the leaf containers. [21]

The hierarchy and grouping of the patterns helps to put ours in a frame of reference, and the pattern definitions themselves are of importance as they provide context to each pattern type grouping. In other words, we are able to categorize realization styles of our pattern using these types.

2.3 Microservice Architecture Pattern

In MSA, applications components are separated out into individual services. [6] These services are developed around 'business capabilities'. This allows teams to develop microservices independently, using tools, languages, and software suites of their choosing. This also allows for evolution of the capability of the system. MSA enables new services to be added or removed in a variety of ways. Services communicate over dumb pipes, avoiding the complexity of traditional middleware solutions. Usually, services are responsible for their own data management, avoiding a single point of data failure as in the case of traditional database systems. These systems are designed for fault tolerance and failure; MSA tries to account for individual service failures. This is usually done via scaling, fail over, or auto-recovery. Services are deigned to be deployed using automated systems and pipelines. [6]

Major advantages of MSA include enforced separation of application concerns (data, logic, and presentation, for example), auto-scaling, high availability, and continuous delivery and deployment. Additionally, services are designed to be immutable; that is, the services are not treated as special deployments that require special configuration for each instance. They are aimed at being redeployed on a regular basis with little to no manual intervention. Containers are a primary method of developing and deploying services using this pattern. [22]

In a particular study from the academic literature, [23], Akbulut and Perros conduct a performance analysis of microservice design patterns. However, each of their implementations varies significantly. They developed three case

studies using three different patterns. The first used an API gateway pattern, which includes a single entrypoint API gateway that implements routing and aggregation of all services. Akbulut and Perros state the major advantage of this pattern is gateway offloading and circuit breaking.The next case study implemented the chain of responsibility pattern. In short, output from one service becomes input for another. The third pattern implemented the asynchronous messaging pattern, utilizing a messaging bus for asynchronous communication between services. The authors state that state that similar applications implemented using different patterns have trade-offs between overhead, performance, and cost. Akbulut and Perros assert that the API gateway is most flexible. They conclude that no one pattern is 'the best', and that each pattern is better suited for different scenarios.

The MSA pattern has several key associated patterns aimed at cloud deployments. These are discussed in Section 2.3.1, in the context of our kill switch pattern. IoT specific MSA topics from academic literature are outlined in Section 2.4.1.

2.3.1 Cloud Design Patterns

Cloud design patterns are extremely compatible with MSA. Resources provided by Microsoft offer a fantastic description of these patterns, most of which are directly compatible with MSA, and include significant overlap with 'traditional' distributed system patterns. However, some patterns are preferable in microservice environments. We have selected useful examples for this thesis in this section. These are referenced in the following chapters to further explain our pattern.

As mentioned in Section 2.2.1, the *sidecar pattern* involves a secondary service that performs peripheral or accenting tasks to a main service or application. This service is independent of the main service. Examples of functionality include platform abstraction, proxying connections, logging, and configuration. [24]

The *retry pattern* is a a method of preventing failures of requests in the case of transient errors of services. Requests are wrapped to perform a retry, retry after delay, or cancel in order to handle the transient errors gracefully. For

example, if service A has transient issues, service B can retry communicating with service A until it is successful.

The *circuit breaker pattern* is similar to the retry pattern. It utilizes the retry pattern, but prevents repeated retries in the case of a likely-to-fail service request. In short, it assumes that the request will fail, where the retry pattern expects it to succeed. The circuit breaker determines if the error is transient or not, and either backs off retry requests or cancels the attempts all together. A proxy utilizing a state machine with closed, half-open, and open states is an example implementation. [25]

The *ambassador pattern* involves an 'ambassador' to a main service, either co-located with a service or in a separate sidecar service. It is used to send network requests on behalf of another service, and is often used with legacy applications to offload client tasks such as monitoring, logging, and security. Some example of ambassador functionality include connecting to a service registry to find actual endpoint location, measure request latency, or for building a common set of client connectivity functionality. [26]

The *bulkhead pattern* involves partitioning services into groups based on load and availability. If a consumer of multiple services tries to repeatedly connect to a failing service and alive services, it can cause cascading failures to all services due to resource exhaustion waiting for response from the failed service. By grouping services, it allows for preservation of some functionality in the event of one service failure. An example implementation is replicating services so that consumers have multiple service instances to connect to. [27]

While not necessarily a cloud pattern, feature flags are a useful pattern in this context. While developing an application or service, certain functionality can be wrapped in a toggle switch. This allows for a specific feature, or set of features, to be enabled or disabled. It also allows for new functionality to be switched in, in place of existing functionality. This is helpful for a canary release, where new services or software are introduced to a system gradually. [28] [29]

2.4 Internet of Things

As mentioned in Section 1, a main inhibitor of IoT adoption is security. While not a complete security solution [7], containerization has inherent security iso-

lation which raises the cost of attacks and reduces surface area. As also mentioned in Section 1, IT operation integration is another major inhibitor of IoT development [9]. Existing and in-development tooling around containers and MSA, such as orchestration platforms, will likely help ease the development and deployment process.

2.4.1 IoT and Microservices

In [30], Santana, Alencar, and Prazeres conducted a systematic mapping study of 18 sources from academic literature to identify research on microservices and the IoT. They identify several contributions to the area, and state their future work includes applying architectural properties of microservices to address the challenges of IoT. The shows an interest in the area, which helps validate our research.

In [31], Butzin, Golatowski, and Timmermann discuss best practices of microservices, and compare them to IoT principles. They compare across self-containment, monitoring and fault cascading, choreography and orchestration, containers, and versioning of services. The authors note the similarities between IoT SOA and microservices, but state that they both approach similar results from differing directions.

In [32], Al-Masri explores the topic of industrial IoT and quality of service for microservices. The author measures the response time, throughput, availability, reliability, and cost of each microservice to create a measure of quality of service, and developed a middleware solution to centralize logging of this information. They assert that this framework "measures the overall quality of microservices for possible integration in IIoT applications". In their future work, they state they plan on including a ranking mechanism to track service quality over time. Our pattern could be considered an example of their proposed ranking mechanism that is abstracted to be general purpose.

2.4.2 Performance Overhead of Containers on IoT Devices

Due to the nature of IoT devices, compute resources are limited. In several studies targeting typical IoT devices, container performance overhead was demon-

strated to be objectively minimal. This suggests that they are ideal for IoT devices.

In [33], Morabito performed an in-depth analysis of the effects of Docker containerization on several types of single-board computers (SBC), and empirically demonstrated that there was negligible impact on CPU, memory, disk I/O, and mixed workload performance. Network performance was shown to be impacted negatively when using the default NAT method, but near native performance when using host networking.

In [34], Noronha, Riegel, Lang, and Bauschert implement a similar performance testing methodology as [33], but investigate using LXC as the container technology, and target devices with various chipsets. Devices include routers, and IoT development board, and a SBC. Noronha, Riegel, Lang, and Bauschert assert there is negligible impact on CPU and memory performance, and comparable network throughput and round-trip times when using Macvlan for the container network interface.

Mendki [35], in an analysis of Docker containers usage in IoT edge video analytics, demonstrated that the overhead of containerization was objectively minimal. The authors also measured the average load time of machine learning models, and found negligible difference between bare-metal and containerized performance.

Considering these studies, it is likely that containers will be further developed, specialized, and adopted for a wide variety of IoT use cases. Of note, enterprise cloud adoption of containers is likely to lead to cross-development of patterns and solutions.

2.4.3 Example IoT Container Architectures

In [36], Ismail et al. analyze Docker in the context of an edge computing platform. The authors evaluated based on the following criteria: deployment and termination, resource and service management, fault tolerance, and caching. To demonstrate, they define a set of use cases, developed an application satisfying the use cases. While Docker has developed significantly since the paper was published,Ismail et al. assert that Docker is a viable candidate especially when compared to VMs.

Morabito et al. [33] describe a platform architecture for smart cars. To demonstrate, they analyze the feasibility of a Docker-enabled SBC system connected to the vehicle via the OBD interface. They implement the application using a variety of containers assigned one of four priority types: critical, high, moderate, and low.

Garcia et al. [37] demonstrate a container-based architecture for industrial robot control. Using Docker and the architectural model specified by the IEC-61499 standard, the authors developed and implemented an architecture called 'flexible container architecture' for a KUKA youBot robot. They utilized several containers: a control manager container for other container orchestration, a robot operating system control container, and a set of application containers used for specific functionality. Garcia et al. assert this architecture addresses legacy emulation and deployment of flexible functionality.

Alam et al. [38] built an example architecture by splitting cloud, fog, and edge tiers using microservices comprised of Docker containers. The different layers communicated using a message hub, with edge and gateway devices acting as Docker 'workers', and the cloud maintaining responsibility for orchestration.

In [39], Noor et al. proposed IoTDoc, which is a Docker-container based architecture for IoT-enabled cloud systems. Noor et al. assert that this architecture creates a 'mobile cloud' comprised of containers running on distributed IoT devices. At a very high level, the architecture is comprised of three primary components: a cloud manager responsible for analysis of existing resources, as well as state management of swarm managers; a swarm manager responsible for collections of nodes, analyzing node resources such as CPU, memory, and sensors, sends images of tasks to the node manager, and tracks work done by the node manager; and a node manager responsible for lifecycle operations of task containers, and resource allocation for containers.

2.5 Safety-Critical Systems and Patterns

Hobbs asserts [40] that major safety-critical system architectural and design patterns can be categorized as follows:

- error detection, including anomaly detection

- error handling
- replication and diversification

Hobbs asserts that [40] by selecting from these patterns acknowledges a 'balancing act' between 'tensions'. For example, a system can be extremely safe by not doing anything; however, the system would not be useful. These 'tensions' are outlined as follows:

- availability and reliability
- usefulness and safety
- security, performance, and safety
- performance and reliability
- implementation concerns, such as readability versus performance

Our approach is most in line with error handling set of design patterns. In particular, [40] outlines the "design safe state pattern". The *design safe state* is a mitigation of known or unknown system factors that cause unwanted behaviour in the system. The design safe state is a state which the system can 'fall back' to in the case of some unknown situation. Returning to the treadmill example from Chapter 1, the design safe state is where the treadmill is no longer moving or providing power to actuators. Hobbs states that it "must be externally visible in an unambiguous manner, [...] must be entered within a predefined, and published, time of the condition being detected, [...] [and] must be documented in the safety manual so that a designer using the component can incorporate it into a larger design" [40].

2.6 Patterns and Methods Similar to Kill Switch

With these elements in mind, our approach synthesizes and adapts trends approaches from three key areas: cloud design patterns applicable to microservice architectures, safety-critical patterns, and IoT container architectures. We outline approaches and related work most applicable to our pattern in this section.

Utilizing the categorization of container patterns outlined in [21], we assert that our pattern is applicable to containers systems using single-node, multi-container deployments. While the categorization is helpful for definition purposes, our pattern is not singularly applicable to containers. Due to how MSA is defined in [6], and by using containers as an expression of services, we assert that our pattern is an addition to MSA architecture patterns as listed in 2.3.1. In particular, our pattern is similar to the bulkhead pattern. However, the bulkhead pattern only considers how to mitigate instances of failing services, and does not consider the dependency of how services interact.

From the safety-critical systems domain, our pattern is similar to the "safe state pattern" as outlined by [40]. However, we assert that we define multiple 'safe-states'; as such, our pattern is quite different. It is also less rigorous in terms of the required qualities outlined by Hobbs. This is explored in more detail in Section 3.6.4.

In terms of IoT architectures, our approach focuses on the general implementation of on-device services. The architectures in 2.4.3 focus on specific implementation solutions to common IoT problem spaces. Of particular interest is work outlined by Morabito et al. in [33] and in Section 2.4.3. In this work, the authors discuss the usage of containers as an enabling technology for smart cars. The authors developed a test architecture for 'accentuating' services, such as infotainment, cameras, and insurance. The authors split containers based on their priority into the following types:

- critical: firmware
- high: cameras
- moderate: insurance
- low: multimedia

To our knowledge, this is the only instance of 'ranking' services based on their priority in the container-based IoT domain. Our pattern, while developed before knowledge of this architecture, takes a similar approach. However, the authors do not extend it as a reusable approach to the concerns of IoT devices.

3. Kill Switch Design Pattern

This chapter provides context for the pattern, the problem addressed, assumptions about the system the pattern is realized in, the definition of the pattern, an overview of the mode change, and analysis of the pattern. Throughout these sections, we highlight key concepts of the definition. To reduce ambiguity, set notation and graph notation are used to define and describe the pattern in detail.

3.1 Context

The concept for the pattern arose from several trends happening in the IT industry and IoT device architecture. There has been a clear push toward widespread adoption of virtualization technologies for at least the past two decades. One of the more recent additions – containers – has led to the ability for deep software stacks to be started and stopped very quickly, and treated similarly to objects as opposed to stand-alone servers. IoT devices have begun adopting virtualization for a variety of use cases, including containers.

From a software architecture standpoint, the use of virtualized systems have inherent boundaries – the network interface of the device, or shared storage of the virtualized systems. By utilizing these inherent boundaries, the application can be structured into units. This leads to microservice architecture, as outlined in Section 2.3.

3.2 Problem

IoT devices have suffered from various security breaches, and this is largely due to systems no longer being updated. Should the system no longer be connected to the Internet, for example, the baseline functionality of the device should be retained.

For example, consider a consumer smart oven developed by the fictional Banana Corporation. The smart oven is built using a set of microservices: a hardware interface service, a scheduler service, a cloud connection service, an updater service, an internet gateway service, and a telemetry service. This smart oven is given away with new home developments, and as such is extremely popular. However, a few days after Banana Corporation goes out of business due to lawsuits, a security vulnerability is discovered that allows an attacker to remotely control the oven via the cloud connection service. This is a nightmare scenario. One would expect that the device should continue to operate as a basic oven, and that the smart oven would stop attempting to connect to the cloud after the company goes out of business. However, now an attacker has access to a stove that could burn down a house. If the developers had decided to add a kill switch to stop all unessential services, and triggered it just before the company went out of business, this scenario would be much less concerning.

3.3 Assumptions

We make a few assumptions about the pattern and deployment methods. The deployment method of resulting architectures utilizing this pattern are built from individually deployable services. This includes runtimes required to run components of an application, and are isolated from other processes that may be running on the same system. In short, this means containers, microkernel-based operating systems, virtual machines, and unikernels.

Because services are isolated units, and because they have inherent boundaries, there should be some method of altering services states to modify application functionality. Then, given some monitor listening for events, there should be some way to trigger the change in application functionality.

Concept 1 *Given some external event, the MSA-based application should be able to adapt to operate using a predefined set of critical functionality, without significant alteration or modification to source code. That is, super sets and subsets of application functionality should be able to operate and exist, in multiple modes of operation, based on developer-defined levels of critical functionality.*

While this pattern is technically applicable to any type of MSA system, the target of this pattern is a resource-constrained environment. The results of the pattern is that at any given time, the system can be reduced so only the bare essential services are running at any given time.

3.3.1 Features, Functionality, and Communication

For clarity we make some assumptions about the distinction between feature and function.

Concept 2 *For the purposes of this pattern, we define features as a holistic measure based on requirements of the consumer of system output. Functionality is the expected outputs of the system, such that it is codified, consumable, and may have side effects on other parts of a system.*

Functionality is provided either passively as in a subscription model, or triggered some external input as in an API call. While features are an important part of any system, the pattern is only concerned with functionality. As an example, a function of a desktop computer is that it can turn on and off, and a feature is that it has a power button to turn it on and off.

Concept 3 *We assert that systems operate by combining the results of separate internal, and optionally external, communication 'sub-functionality' outputs together to provide some set of resultant functionality. In order to access the results of a system, and thereby its functionality, at least one method of external communication is required.*

If the system is 'closed' and does not provide external functionality, while it may be valid, it is useless.

As such, for systems that require external communication, we need to define system boundaries. This boundary starts and ends with the available methods of invoking functionality of the services. This would be ports, for example.

MSA relies on lightweight methods of interprocess communication. In order to reduce complexity, we do not consider the properties of protocols used to deliver data or functionality between services. For example, we do not consider a message being posted to a message bus with TCP as a bi-directional set of communication due to the SYN/SYN-ACK/ACK handshake.

3.3.2 Services, Microservices, and Containers

The definition of services and microservices is quite broad in the literature. We define them here for clarity and to avoid ambiguity in the following sections.

A virtual machine is a system that operates using subdivided 'bare-metal' resources. They virtualize the entire operating system or appliance, without reusing resources from other parts of the system the 'guests' run on.

A container, as defined in Section 2.2, is a method of isolating a process and constraining available resources to it. Containers are typically implemented in Linux, and utilize the base Linux kernel to run the process. This allows for various distributions or stand-alone runtimes to be used as the 'base' of the container. It is a method of 'light' virtualization, as resources from the 'host' operating system can be reused in the 'guest' container.

Services, in the typical view of a SOA system, are monolithic service components. Several services are combined to provide application functionality. Services typically use shared storage and a shared communication bus. As such, traditional services are harder to maintain as the code-base for a service is tightly-coupled, and highly dependent on multiple single points of failure.

A microservice, as defined in Section 2.3, are a variation of SOA. Microservices are individually deployable services. Microservices are developed around 'business capabilities', and have full autonomy over data storage, tools, and languages. The communicate over dumb pipes, or simple communication methods such as REST APIs. Microservices should be implemented such that the communication path makes sense; via HTTP requests, or communication over message busses. Also, data should be stored per microservice; that is, each

service is responsible for its own data. This allows for easier replication, scaling, and resiliency.

A microservice can be implemented in a variety of ways. Specifically, using containers as an example, a microservice can be implemented either via one container for the entire microservice, or multiple tightly-coupled containers for an individual microservice. As such, there is not necessarily a one-to-one mapping of one container to one microservice. Similarly, virtual machines can be used to create microservices in the same way; one-to-one, or one-to-many. It is preferable to separate microservices out to several components to enable easier deployment of the microservice. However, this comes with the drawback of increased resource utilization and interprocess communication.

In the kill switch, we are concerned with the component-level organization of the system. Specifically, we consider the *components* of a microservice architecture. We define these components as *services* throughout the following sections. A microservice architecture is comprised of one or many microservices. For our definition, we use the term *service* to define a 'component' of the *microservice architecture*, such that it provides some functionality to the system. In other words, *services* provide a service to the application itself. Services can be viewed as either the individual components that create several microservices, *or* the individual microservices that communicate with one another. In either view, and the definition and pattern still holds. With the introduction of the mode change mechanism, we kill either microservices, or components of microservices.

We assume that the overall system architecture follows good microservice design, such that functionality of microservices are loosely coupled, and organized by 'higher level' functionality. We also assume that at least one 'microservice' exists on the system.

3.4 Definition

The pattern itself is straight-forward: we have a single device running application architecture using services as the internal building blocks. Services are defined as microservices, as described in Section 2.3, where each service is able to deployed as an individual unit. Service can combine to become a microser-

vice, or the entire device could be viewed as a microservice.

Each service has a potential method of communicating with other services. Likewise, each service can also communicate with external systems or services. Because the internal services are standalone units, they can be removed from the system at any time. As such, the application can be built into tiers of functionality, as any service can become an external communication point, and any service can perform myriads of functionality in a self-contained way. By killing separate tiers, the system gains the ability to reduce its attack surface area, and energy consumption.

Concept 4 *Via some switch event event, a set of services providing some set of functionality, and their individual states, shall be alterable at run time to provide some reduced, minimum, or altered set of services and functionality. The application architecture resulting from accounting for this pattern consideration should have the ability to enable multiple modes of overall application operation. For example, in an application where both critical and non-critical containers are deployed, a mode change would ensure critical functionality is preserved.*

For example, suppose you have an IoT software system that has a series of containers that each individually manage a hardware interface, core application logic API, an Internet proxy, and a local network proxy. In a security event, the application could kill all the containers except for the hardware interface and core application logic, preserving baseline functionality of the system. Section 3.4.5 provides an overview of this scenario.

3.4.1 Internal System

To begin, we define n to be positive integers:

$$\text{Let } n \in \mathbb{Z}^+$$

There is a set called $Services$ comprised of n individually deployable services s. This implies that each s is treated as a unique element. Even if a service is 'replicated', each unit is treated as unique:

$$\text{Let } Services = \{\ s_1,\ \ldots,\ s_n\ \}$$

Each s is mapped via an arbitrary 'decision' function $nodeLeveler$ that decides an $opLevel$ for each s. $opLevel$ belongs to a set $OpLevels$, denoted by a set of positive integers and constrained by the number of s (i.e, n) in the system. In other words, there is at least one operation level, and there can be at most one operation level for each service in the system. For clarity, 1 can be thought of as the essential operation level.

$$\text{Let } nodeLeveler \text{ be a 'decision' function}$$
$$nodeLeveler : Services \rightarrow OpLevels$$
$$\forall opLevel \in OpLevels,\ \exists s \in Services \mid opLevel = nodeLeveler(\ s)$$
$$OpLevels \subseteq n$$

3.4.2 Internal Communication

Each s has an optional number of outward-directional communication methods to other s, as defined by an application developer. This is represented as ordered pairs in the relation set $Comms$, and simplified in the notation as c_x:

$$Comms = \{(\ s_a, s_b)\ , (\ s_a, s_c)\ \dots)\ \}$$
$$Comms = \{c_1\ , c_2\ \dots)\ \}$$

Items in $Comms$ are considered edges in the system, leading to a directed graph. As such, this leads to each s being a node. As a further constraint, we do not consider self-referential communication from s_a to s_a as an edge (loops). This leads to a simple directed graph. Therefore, each s can have a maximum of exactly one communication path to any other s in $Services$; as such, for each s has outdegree of at most $n - 1$:

$$0 \leq \mid Comms \mid \leq\ n * (n - 1)$$

To formalize, the $System$ graph is defined as follows, with $Services$ as nodes and $Comms$ as edges:

$$System = (\ Services,\ Comms)$$
$$Comms \subseteq \{(\ s_a, s_b)\ \mid (\ s_a, s_b)\ \in Services^2 \wedge s_a \neq s_b\}$$

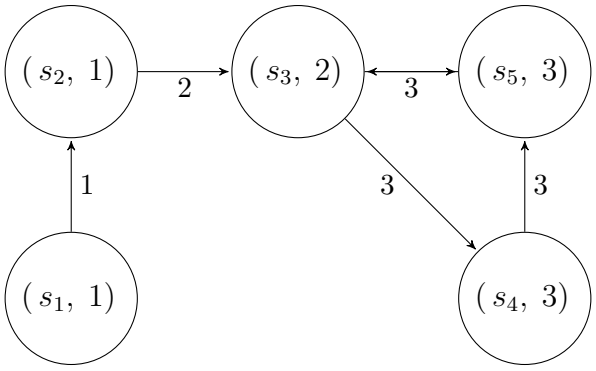


Figure 3.1: A graph representation of an example $System$. Each node is labeled with the results of $nodeLeveler$ function, and each edge is 'weighted' with the results of $commLeveler$.

In order to find the operation level of the communication path, the function $commLeveler$ is applied $Comms$. It takes arguments c from $Comms$, and finds the maximum of the results of applying the $nodeLeveler$ function to each node in the ordered pair of c. As a result, each c receives an $opLevel$, and the maximum is selected. For example, if a communication endpoint connects from a level 1 node to a level 2 node, that level of communication is set as 2. The reasoning for this is that once a higher level node is destroyed, the lower level node will not be able to communicate with it. Because there is no guarantee that each c will map to all $opLevel$, the function is not surjective. The function maps $Comms$ to $OpLevels$ as follows:

$$commLeveler = max(\, nodeLeveler(\, s_a)\, , (\, nodeLeveler(\, s_b)\,)$$
$$commLeveler \;:\; Comms \rightarrow OpLevels$$
$$\forall c \in Comms,\; \exists! opLevel \in OpLevels \mid opLevel = commLeveler(\, c)$$

$$commLeveler = \{(\, (\, s_a,\; s_b)\, ,\; max(nodeLeveler(\, s_a)\, ,\; nodeLeveler(\, s_b)\,)\, , \dots \}$$

The results of this allows for each edge to be 'weighted', making $System$ a simple directed weighted graph. While edge weighting is not strictly required, the weighting allows for design and structuring of application communication paths.

3.4.3 External Communication

As mentioned in 3.3.1, the pattern assumes that all systems require some method of interacting with external systems. As such, there is at least one method of communicating with an external system. For example, this could be a console session, a web interface, or an API.

We define m to be positive integers:

$$\text{Let } m \in \mathbb{Z}^+$$

We define the external systems x, as part of set Ext. There are m external systems. We treat each external system endpoint as a unique node. As x is outside the controllable domain of $System$, it receives an $opLevel$ of 0 by default:

$$\text{Let } Ext = \{\ x_1,\ \ldots,\ x_m\ \}$$

External communication can occur from an internal node to the external node, and vice versa. This is captured in the relation set $ExtComms$, and are treated as edges in the $System$ graph. Each edge operation level is decided by the $commLeveler$ function. Again, there is no guarantee that each edge will map to all $opLevel$, so the function is not surjective.:

$$ExtComms = \{(\ x_a, s_a)\ ,(\ s_n, x_b)\ \ldots)\ \}$$
$$commLeveler\ :\ ExtComms \rightarrow OpLevels$$
$$\forall(\ x_a, x_b) \in ExtComms,\ \exists! opLevel \in OpLevels \mid opLevel = commLeveler(\ (\ x_a, x_b))$$

We combine the external nodes and internal nodes, and add the internal and external edges to create $FullSystem$.

$$Nodes = Services \cup Ext$$
$$Edges = Comms \cup ExtComms$$
$$FullSystem = (\ Nodes,\ Edges)$$

3.4.4 Functionality

We treat functionality from each software component and collections of software components as 'black boxes'. That is, functionality of each node on the graph is only available via some communication path..

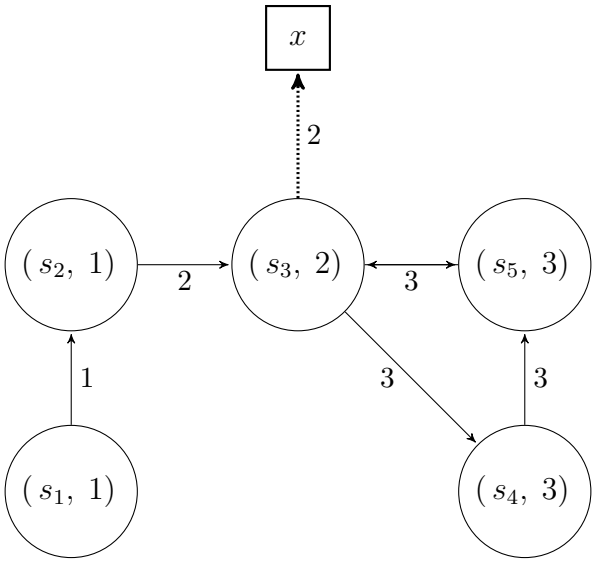


Figure 3.2: A graph representation of an example $FullSystem$, with square nodes representing Ext, and the dashed line representing $ExtComms$ edges

Functionality is provided over external communication edge x. Each item of functionality, then, is subject to the $opLevel$ of the edge.

This functionality may be exposed via some intermediary service at a higher $opLevel$, meaning that in the case of a mode change, while the external functionality may be removed at the externally-accessible node, internal functionality could remain in place. This, however, is a design decision.

3.4.5 Example

While the definitions are necessary, it is helpful to solidify using a simple example. In Figure 3.3, there is a microservice-based application on an IoT device with three levels.

The application consists of four services:

- a hardware interaction service at level 0, that would handle communications with the underlying hardware such as sensors, actuators, or other device hardware
- a web API service at level 1, responsible for communicating with the hardware service

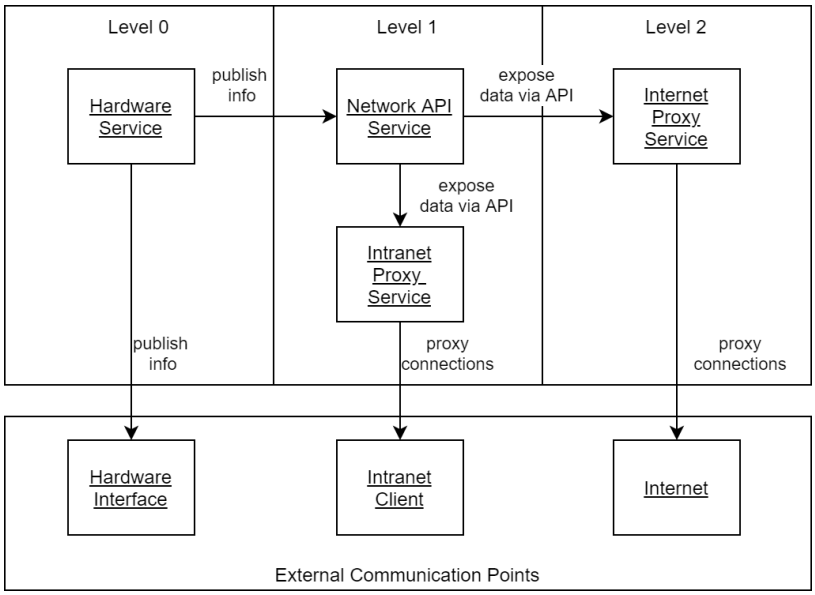


Figure 3.3: An example of services and their communication paths

- an intranet proxy sidecar service at level 1, responsible for communicating with the web API and any intranet clients
- an internet proxy sidecar service at level 2, responsible for communicating with the web API and any Internet clients

In terms of communication, as shown, the hardware service directly communicates with the hardware user interface to change application state. This creates a dependency graph, which we explore the implications of in Section 3.6.3. The intranet proxy deals with client endpoints on the internal network, and the internet proxy deals with client endpoints on the external network. While the proxy communication paths would usually be accomplished via an on device firewall, by utilizing the proxy, connection to functionality can be dynamically updated without affecting the functionality of the internal clients. This is useful in the case of a secured configuration: should the end user of the IoT choose to disable all internet functionality, all that is required is stopping the level 2 service. Should the user no longer require network functionality, they can disable level 1. This retains hardware functionality.

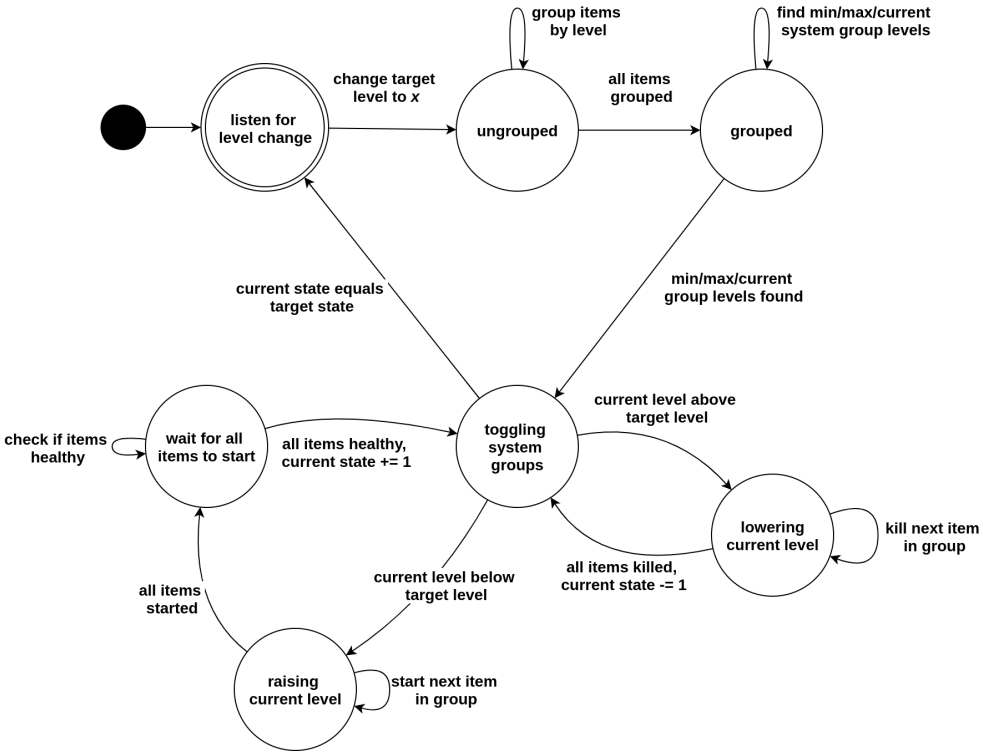


Figure 3.4: State machine diagram outlining the general process of altering system state. The solid circle represents the initial state, the circles represent a state, and the double outlined circle represents the start state.

3.5 Mode Change Mechanism

The pattern is essentially an operational level organization system for services, internal communication paths, external communication paths, and system functionality. This in itself is useful for structuring the application. By explicitly abstracting or placing key parts of the application further away from interaction points, there is a natural 'barrier' of lower-value application components. While this is a fairly subjective measure of usefulness, the concrete value comes with the mode change. As such, there needs to be a way to implement the mode change.

An example algorithm for the mode change outlined in Figure 3.4 is extremely basic. The mode change service 'mode-changer' listens for a mode change condition. Once 'mode-changer' hears a mode change, it reads the meta-data of the services, and groups them all by level. It then finds the mini-

mum, maximum, and current level of the services. If the target level is different than the current level, 'mode-changer' checks if the level is higher or lower. If the target level is higher – that is, the current level needs to be raised – each service in the level is started. 'mode-changer' then checks if all services have started. If so, then 'mode-changer' updates the current level, and repeats this process until all services in each group are started up to the target level. If the target level is lower, each service in the current level is killed, the current level is updated, and the process repeats. Using the state machine as a basis, we have added an algorithm using pseudocode in Listing 3.1.

This state machine does not account for the time required to start or stop the services, or how quickly events should be processed. However, as a rule of thumb:

- in the case of an $opLevel$ elevation (1 to 2), any subsequent elevation event should wait an equal or longer length of time than the longest s takes to start in that level
- in the case of an $opLevel$ reduction (2 to 1), subsequent reduction events should be processed immediately

The mode change listener can be implemented in a variety of ways. The most straight-forward way would be with an external script that calls the engine or system responsible for maintaining state of the services. The next would be with a service internal to the system, but with access to APIs that control the state of services on the target system. The third would be modifying the engine or state maintainer of the services to include a kill switch API, which was implemented in our previous work [41].

3.5.1 Events

The mode change should be triggered by events. For example, a complex event processor may trigger the mode change. For examples of what may generate events:

- 'metadata' rankings of regular layer 7 (application) requests to the system as defined by some criteria

```
def do_level_change(target_level, services_list):
  groups = do_grouping(services_list)
  min, max, current = find_min_max_current(services_list)

  if current_level > target_level:
    for i = current_level; i <= target_level; i++:
      for service in group[i]:
        service.kill()

  elif current_level < target_level:
    for i = current_level; i >= target_level; i--:
      for service in groups[i]:
        service.start()
    wait_for_services(groups[i])

def wait_for_services(group):
  [...]

def do_grouping(services_list):
  groups = {}
  for service in services_list:
    if service.level not in groups:
      groups.add(service.level)
    groups[service_level].append(service)
  return groups

def find_min_max_current(services):
  min_level = Infinity
  max_level = -Infinity
  current_level = -Infinity

  for service in services:
    if service.level > max_level:
      max_level = service.level

    if service.level < min_level:
      min_level = service.level

    if service.level > current_level and service.is_alive:
      current_level = service.level
  return min_level, max_level, current_level
```

Listing 3.1: Mode change pseudocode

- an external cloud system that triggers a mode change, such as an orchestration layer
- an on-device or off-device security system that monitors for security events
- an internal resource monitor that triggers a mode change based on some criteria; for example, a low battery usage, or high resource utilization in a particular layer

Given that this is a pattern, and not a framework or architecture, how to define events is not described here. Concrete examples of events are provided in Chapter 4. Regardless of how events are defined, how they eventually interact with the system is of importance. An example of mapping events follows.

For any arbitrary event e in set $Events$, map each e to an $opLevel$:

$$eventLeveler \ : \ Events \rightarrow OpLevels$$
$$\forall e \in Events, \ \exists! opLevel \in OpLevels \mid opLevel = eventLeveler(\ e)$$

3.6 Analysis

The definition of the pattern is aimed at reducing ambiguity, and serves as a reference point for the pattern. However, there are various ways to implement a pattern, and kill switch is no different. This section is aimed at analyzing the pattern, in order to reduce the ambiguity of implementing the pattern and to understand its benefits and drawbacks.

We discuss level definitions, the trade-offs of communication and dependency, methods of interacting with other design patterns, specific considerations for IoT devices, and limitations and challenges of the pattern.

3.6.1 Level Definition

Levels are the key component of this pattern, as they define how services are deployed and communicate. However, levels are just the method of organization; developers must decide what they represent. It is important to remember that each of these levels are running concurrently; that is, level 0 is running when level 1 is running, and levels 0 and 1 are running when level 2 is running.

Implementation of levels should be accomplished via metadata; that is, the configuration of levels should not be embedded directly as part of application code, but as part of the deployment definition of the services. The goal of the kill switch pattern is not to introduce new functionality to an application based on levels; the goal is to remove existing functionality of the application in a controlled way based on end-user requirements. Just like a physical kill switch, the kill switch pattern is meant to protect end users from unintended consequences of using the system or device. This is an important distinction: end users are the ones who benefit from graceful service degradation in an emergency event.

Determining how an application will communicate before all services are in place is challenging. But, separating services into levels reduces the amount of information flowing to one or all services. Should most or all services be known, by developing a dependency graph similar to Figure 3.3 and those outlined in Chapter 3, service interaction can be more easily categorized into levels. Should a service be interconnected to the point where all other services require communication to one service, this goes against the core of MSA. MSA should allow for independent deployment of services. In a cloud scenario, tools such as Chaos Monkey [1] randomly kill instances of services to encourage resiliency. However, in a resource-constrained system, services do not have the luxury of using replication to handle resiliency. By limiting communication paths, the higher level layers compensate or mitigate issues such as problems with end-client communication. Should communications happen over a message bus, or a centralized bus, things become easier to implement. However, the bus itself becomes a critical service, and should be treated as such.

Should a service be essential, it is helpful to create sidecars to limit access to the service. These sidecars can then be placed at whichever level is required for dependent services, and be killed to cleanly break communication paths. Similarly, by using multiple lightweight API gateways, different levels of the application can be exposed based on levels. As a drawback, this introduces overhead.

Consider that not all parts of the application are required to be in a level; however, any service outside of scope will either be always running, be killed

[1] `https://netflix.github.io/chaosmonkey/`

should the kill policy of the application deem it necessary, or be running intermittently despite any scenario requiring essential services only.

3.6.2 Internet-of-Things Devices

While we have described the kill switch in terms of multiple levels, it is aimed at an MSA architecture with few interdependencies and a mission-critical set of functionality. In an oversimplification, IoT devices have two duties: perform some real-life function, such as sensing input from the real world or operating as a normal item in everyday life, and provide functionality over a network interface. In its most basic form, the kill switch should be implemented in two levels to reflect this. The level 0 would be the core hardware interaction level, and level 1 would be for all other 'smart' functionality. From an end-users' perspective, if the device is under active attack, it should still have functionality without being connected to the cloud.

In short, this pattern is aimed at single-node, multi-container environments as defined in Section 2.2.1. This does not preclude it from MSA. MSA gains a significant amount of influence because deployed services are designed to be more transient. That is, MSA services should be designed to be killed, restarted, and scaled. Because IoT devices do not have access to a large amount of resources, the kill switch pattern acts as a mitigation for redundancy, while still realizing many of the benefits of MSA. By controlling application functionality by level state, the application is more predictable should higher level services become unusable.

3.6.3 Communication and Dependency

While not a hard requirement of the pattern, it is suggested that the graph derived from the resultant architecture should minimize the amount of cycles, especially between $opLevel$ groupings. In other words, the resultant graph should aim to be a directed acyclic graph. With fewer cycles, there are clearly defined paths for data to flow amongst the nodes. This eases in realization of the pattern in real-world terms. We use this approach in Chapter 4.

As part of our definition of the kill switch pattern, we stipulate that functionality can only be exposed via communication paths. This also means we

have stipulated that communication paths are synonymous with dependency. That is, functionality of base services can only be exposed via a communication path, which creates dependency. As such, higher level services should have a dependency on lower level services, but not vice versa. This creates challenges when developing microservice architectures as services may not be structured in a tiered way.

In many cases, MSA architectures are 'flat'; services all operate amongst themselves, and with no clear chain of communication. For example, take an architecture with all 'level 0' functionality. In this scenario, if we were to try to have all services communicate, the resulting dependency graph would be unpredictable. This creates a scenario where the kill switch pattern would not be able to be implemented. Of interest, this is also difficult to manage from an MSA perspective. If the services all become so coupled that they cannot operate without other 'equal level' services, starting the services in the case of a system failure may not be possible.

One particular scenario of concern is a cyclical dependency. If all services create a cycle with independent nodes, those services essentially have to be treated as one 'supernode'. This could be fine if there are other services in the system, but if these are the only services in a given system, there should be a clear 'point of contact', and only allow the coupled services to communicate amongst themselves.

There are ways to mitigate this. For example, higher level services are able to access the lower level service via sidecar proxies placed at a level higher than the lower level service. This also helps to decouple the two services. Furthermore, multiple sidecars exposing different sets of functionality of the same service can be developed, exposing the underlying service in a tiered way. Should we expose some of those services to one another via sidecars at higher levels, this creates a situation akin to facades in object oriented programming.

Esparrachi, Reilly, and Rentz [42] outline concerns when mapping out microservice dependencies. They outline that these dependencies become a directed graph, and that graphs with no cycles are ideal. They also describe methods of dependency management via passively tracking or actively controlling them. The authors also suggest building services such that they are 'stacked', meaning that services never depend on services below them. This is what we

suggest when developing MSA applications, especially for single-device implementations.

There are tools available that ease tracking dependencies in MSA. In [43], Ma et al. describe such a tool called GSMART: graph-based and scenario-driven microservice analysis, retrieval, and testing. This type of tool can be used for existing and design of MSA applications.

By developing the services with their dependency in mind, and by mapping out their communication paths, we are then able to understand the application as a whole. This also allows an understanding of its critical functionality. As this is a pattern, we do not define or mandate any particular method of doing this.

3.6.4 Related Patterns

As with many other patterns, this pattern is compatible and more useful when combined with other patterns. This section outlines key patterns that kill switch is comparable to, or can interact with.

Cloud Design Patterns

In many respects, the kill switch pattern is a variation of the layered or tiered pattern. However, we are applying layers to disparate services, as opposed to a monolithic applications or deployments. In addition, the levels are arbitrary groupings based on internal and external functionality, where layers or tiers are defined by logical groupings. The layer or tiered approach does not have any opinion of functionality implementation, whereas kill switch is opinionated.

The bulkhead pattern, as described in Section 2.3.1, is similar to the kill switch pattern. However, it is helpful to think of bulkhead as a vertical segmentation of services, whereas kill switch segments it out horizontally. Additionally, the bulkhead pattern is aimed at leveraging cloud functionality such as auto-scaling or redundancy. Kill switch is also opinionated about the grouping of services. One particular configuration of interest is 'kill-switch-in-a-bulkhead'. A series of services, segmented 'horizontally' into bulkheads, can then also implement the kill switch leveling 'vertically'. This allows for individual bulkheads to have mode changes applied to them. Also, it allows for the entire set of services to be controlled in a similar way to orchestration. However, in contrast to

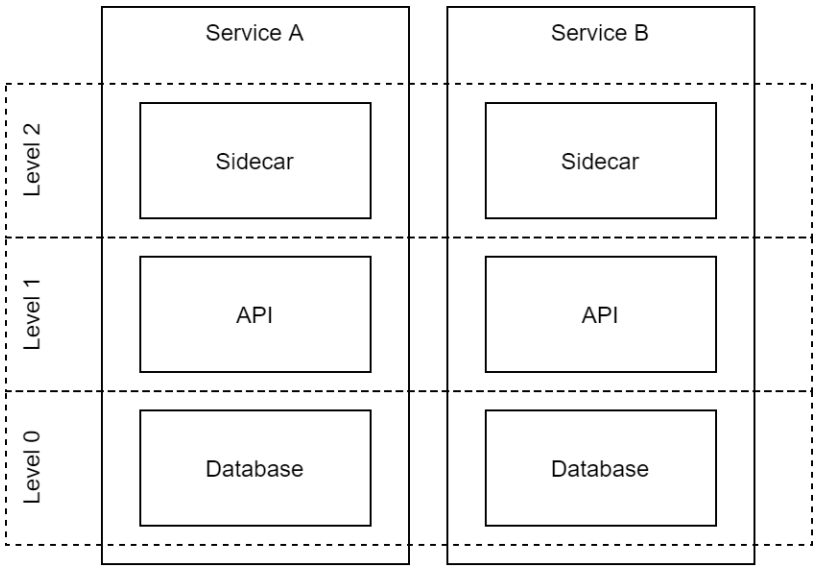


Figure 3.5: A generic kill switch architecture implementing cloud patterns

arbitrary orchestration, defined 'levels' and 'bulkheads' could be started at will.

As an example, consider Figure 3.5. While there are various ways to implement the bulkhead pattern, in this case we assume the services are independently collaborating in 'swim lanes'. That is, each microservice is vertically segmented, with dependencies on higher and lower components. Requests are made asynchronously to each service, and the services are not dependent on one another. In this case, the components of each microservice are able to have identical levels. Alternatively, the services can be individually leveled; level 0, 1, and 2 in Service A can be started differently that service B.

Using sidecar and ambassador patterns with the kill switch lead to some interesting scenarios. As mentioned, when using these patterns, an application can expose functionality at different levels without having to expose the base service. To use an object-oriented term, sidecars can be used as a combination of facades and feature toggles. In other words, the sidecar can be built such that it only exposes part of the functionality of the underlying service to higher-level service. Again, referring to Figure 3.5, assume that the sidecars are only used to expose functionality, and do not provide further augmentation to the service functionality. In this case, the application can expose and shut off access at level 2, if required, by leveraging sidecars.

The circuit-breaker pattern should be used in conjunction with the kill switch

pattern. As parts of an application architecture are designed to be killed, there should be some way for lower $opLevel$ services to prevent further requests to upstream services. This also helps to prevent resource starvation situations, as gracefully backing off connection requests also inherently creates a method of dealing with unavailable resources. Considering Figure 3.5, should the sidecars in level 2 be killed, subsequent requests from internal or external communication endpoints should detect the failed request and back off further attempts.

Safety-Critical Patterns

Our pattern is not labeled as 'safety-critical', and should not be used for such until further research is conducted into the safety implications as outlined by this domain. For example, the pattern does not account for concerns such as anomalies, or replication. Our approach developed this pattern from a 'cloud' and Internet-of-Things perspective. However, the approach led to some similarities between our pattern and those in the safety-critical domain. Of interest are failure types. Our pattern allows a controlled method of dealing with failures, and for implementation of different failure types. This is similar to implementing a 'fail-safe' mode. However, our pattern does not stipulate what type of functionality is exposed at any given time, so strict claims about the guarantees or types of failure mitigation cannot be made.

By having a 'known good' state of critical functionality to fall back to, the architecture itself is able to revert to that state. While the 'safe state' is essentially described as the safest state possible with fewest negative side effects, the safe state pattern takes many forms. For example, if the design safe state of a car is to lock up, and it locks up while driving, this could cause further harm. This is an example of an objectively bad safe state, as too much functionality is removed. In this case, the car should allow manual steering and manual brakes to be used, independent of the ECU. Similarly, the kill switch allows for implementation of these types of states – states that an application can fall back to.

However, kill switch concepts are compatible with basic premise of 'safe states'. Our approach can meet the stipulations of safe state defined by [40] discussed in 2.5 based on the rigour of realization of the pattern, but this is up to the system developer. Our approach takes a 'lighter' vision of this, by allowing a state similar to a 'safe state' to be defined in the case of a perfectly

functioning system. In other words, the 'safe states' available by realization of the pattern are arbitrary, they do not consider underlying hardware, and do not impose any stipulations.

3.6.5 Security and Maintainability

By utilizing this pattern, we assert that the resultant application is more secure and maintainable. The adoption of containers and microservices themselves offer significant security and maintainability advantages if implemented correctly. For example, each service and set of services are able to be deployed individually, without having to update the entire application. This allows for devices to receive 'piece wise' updates. This allows for individual components to be updated, independent of other parts of the system.

However, implementing the pattern also allows for a higher level of security and maintainability. Microservices can lead to spiraling dependency issues. By utilizing the pattern, the resultant architecture enforces a clear understanding of dependencies between components. This is a huge advantage when developing new code for applications of any type; by ensuring that the dependency chain is built directly into the architecture of the system, an understanding of the system is clearer, and changes can be made more easily.

In addition, by incorporating a 'known-good' state into the design of the system, there is always a fallback should higher level services become affected or unusable. The definition of level 0 as a set of critical functionality establishes a baseline in the event of either unknown circumstances, or a security event.

3.6.6 Limitations and Challenges

Every pattern has limitations and challenges. While we have discussed some of them in the above sections, we will expand on them here from this context. We will also discuss some that are out-of-scope for the above sections.

As mentioned in Section 3.6.1, the kill switch pattern is *best suited for applications with discrete critical and non-critical functionality*. Specifically, these levels should be critical functionality and non-critical functionality. While the possibility of introducing multiple levels allows for fine-grained control over reducing system functionality, the design of such a system is quite challenging. In other

words, *use of the pattern can lead to spiraling complexity*. Trying to account for each level is challenging, and mapping the dependencies of microservices is difficult as it is. As such, it forces the application to adopt a tiered approach. While this in itself is not an issue, some applications are not designed to be built in such a way. This is why we stipulate that *this pattern is best-suited for limited scope MSA applications, such as those on IoT devices*.

As mentioned in Section 3.6.3, the kill switch pattern is *best suited for architectures with few to no graph cycles*. If cycles are in the system, it creates a scenario where the services in the cycle should be placed in the same level due to dependencies.

A major limitation is that *the pattern is most effective when paired with a mode change mechanism*. This strays into the area of frameworks. A framework is a cohesive set of components that accomplish a goal, whereas a pattern is a methodology or general structure to solve an architecture problem. Kill switch is decidedly a pattern, and defines methods of building microservice applications with enforced consideration of dependency.

A concern specific to the mode change functionality is *the length of time it takes to start a number of levels*. If a system is in a state where it is raising its level, the system must wait for the services in a group to start. This may take a significant amount of time. This creates situations where emergency mode changes are received. How the mode change deals with this scenario is outside the scope of this work, but safety-critical systems have several methods of recovering from such a scenario [40].

Another limitation is the case of *multiple dependent services with equal importance*. For example, take a critical service that has two dependent services. The services perform very different functionality, but are of equal importance. The kill switch pattern does not have an elegant way of dealing with this scenario; either one service is placed in a higher level than another, or they are placed in the same level. If a scenario causes a mode change that should only affect one of the dependent services, there is no way to decide which to kill. In this case, using the bulkhead pattern to segment out the kill switch 'planes' would be a reasonable solution. However, it does introduce more complexity to the design.

4. Realization Examples

We chose to demonstrate efficacy via example realizations of the pattern. In Section 4.2, we demonstrate a naive realization of the pattern, and implementation of the mode change by modifying the Docker engine API. In Section 4.3, we built an example realization demonstrating the kill switch pattern and demonstrate the usefulness of the resultant architecture using a 'manual' kill switch tool. In Section 4.4, we demonstrate how this kill switch can be used in conjunction with an intrusion detection system to utilize the resultant architecture.

Select source code is available in Appendix B, C, D for posterity, and on Github[1] for easier access. All source code is available upon request.

4.1 Set-Up

In order to demonstrate the kill switch pattern, we have opted to develop the realization examples using commodity hardware. The system used as the target of deployment in each of these realizations is not an IoT system. It is a standard bare-metal system using Ubuntu Desktop as the base operating system, and is running Docker. Further specifications of the system are shown in Appendix A.2. This was done for the following reasons:

- the concepts are easily transferable to regular microservice deployments,
- the nature of virtualization allows this example to be deployed to open IoT systems that support containers,
- to ease the development process.

[1] `https://github.com/davidlennick/killswitch-examples`

4.2 Docker Engine API Modification Example

This was an early effort that lead to the kill switch pattern in its current form, which was published in [41]. Our intent was to explore if implementation of this type of system was feasible. We took a naive approach to levelling, as well as the functionality of services. We deployed a set of 'essential' and 'non-essential' containers, and specified them with a label. On some event, the mode change collected and checked all container labels, and killed any unnecessary containers. We chose this implementation because it is a simple, relatively unopinionated way to demonstrate the concept of the pattern. It was simple to implement, and as a result, simple to understand. It was aimed at demonstrating the efficacy of implementing the pattern without assuming any application type. This also allowed for a demonstration of the trade-offs between implementation of the mode change mechanism using various strategies.

We modified the engine by adding the "kill switch" API endpoint, and added the logic to actually perform the mode change (ie., kill all non-essential containers). The API listened on a Unix socket, and accepted a POST HTTP request with no payload. We implemented the associated logic for the API leveraging existing engine backend interfaces. The logic flow is: a timer is initialized, the container IDs matching the specified label are found, the kill method for each container is invoked, the timer stops, and the execution time is returned. All containers are assumed to be critical unless otherwise specified in order to make development more straightforward. Figure 4.1 outlines the generalized sequence of events we implemented, and how the mode change operates. This process is similar to the mode change algorithm outlined in Section 3.5.

To provide an overview, Figure 4.2b outlines a simplified diagram of the Docker engine and associated logical layers. There are four 'top' logical layers:

- a management layer, which denotes methods of interacting with the Docker engine externally;
- an application layer, which holds application containers;
- the Docker engine layer;

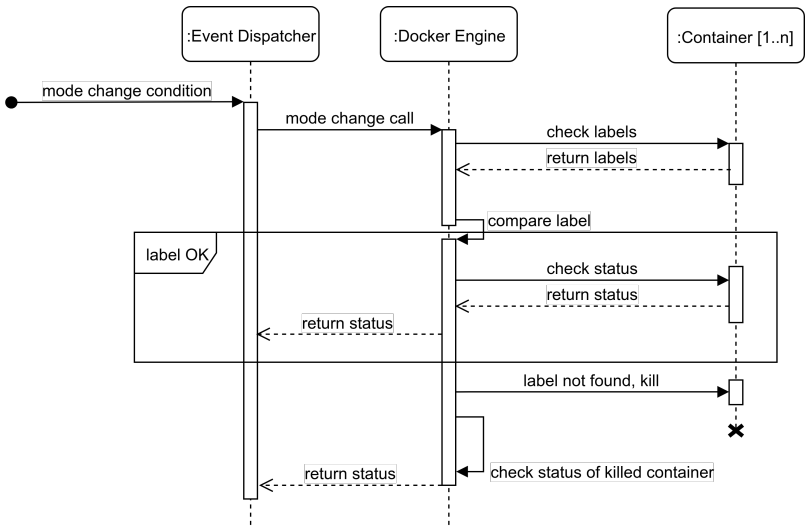
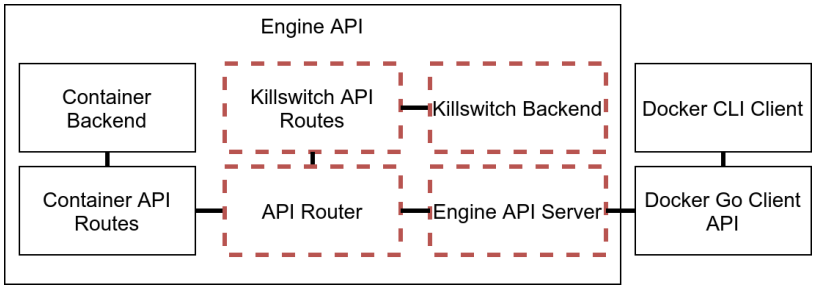


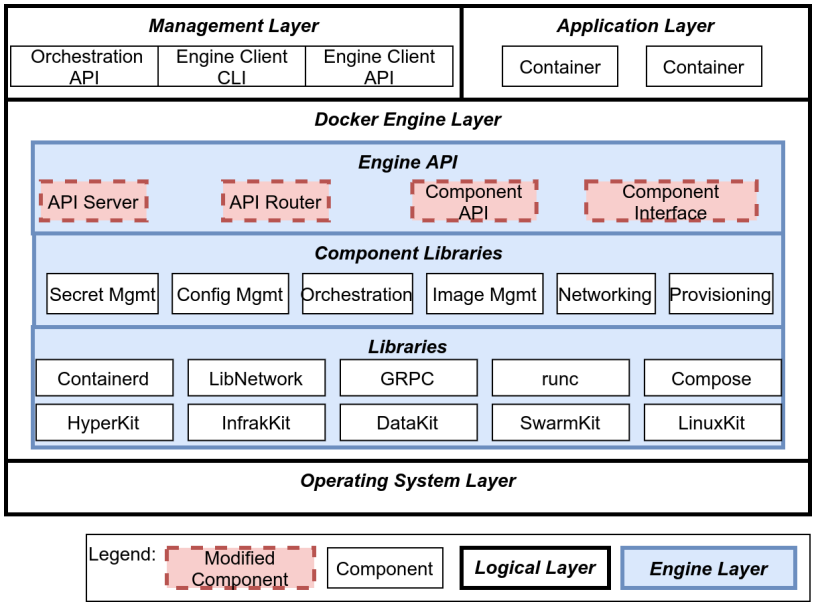
Figure 4.1: Sequence diagram for mode change implementation in Docker

- and the operating system layer, which is whatever operating system Docker can run on (meaning Linux).

The Docker engine itself is separated into three logical layers [44]: the engine API layer; the component libraries; and base libraries. Component libraries combine the lower level libraries to perform tasks such as life cycle management of containers, image creation, modification of networking tables on the host operating system to network containers, or other typical functionality of the engine. The dashed boxes show the logical components that we modified. The diagram shows that only the engine API layer was modified for this implementation. The modified components are further outlined in Figure 4.2a. Modifications to lower levels of the engine, such as the task queue, or the backend modules the APIs leverage, were not explored. The important executed code is outlined in Listing 4.1.



(a) Simplified overview of engine API modifications



(b) Simplified Docker engine block diagram

Figure 4.2: Summary of Docker engine modifications

To verify efficacy, and to further demonstrate implementation of the mode change, we wrote an external Go script leveraging the Docker client Go library to performing the same functionality as the API. This external script is meant to demonstrate an external system, such as an orchestration layer. The code is virtually identical to mode change API code. However, one key difference is that should this script be invoked from a remote machine, the time to react to the event and perform the mode change would be dramatically increased due to network latency.

We used a Bash script as the 'event dispatcher', which calls the internal mode change API using cURL and starts the Go script for the external kill switch. We used the following process to perform testing:

```
// api/router/killswitch/killswitch_routes.go
[...]
var start = time.Now()
var d time.Duration

// actual start
var sig uint64 = 0x9 // kill signal
var filter = filters.NewArgs()
filter.Add("label", "safety=false")

// return filtered containers
containers, err := s.backend.Containers(
    &types.ContainerListOptions{
      Filters: filter,
})
if err != nil {
  return err
}

// kill all the containers that match
for _, c := range containers {
  s.backend.ContainerKill(c.ID, sig)
}

d = time.Since(start)

return httputils.WriteJSON(
    w,
    http.StatusOK,
    d
)
[...]
```

Listing 4.1: Docker engine modifications

```
c_num -> maximum number of containers to test in system
iterations -> number of times to repeat the test

for c in range(c_num):
    kill the daemon
    start the daemon
    clean the environment
    add a "safety" container

    for i in range(iterations):
        PRINT the time
        PRINT c
        add c "non-safety" containers
        call the kill switch API
        PRINT kill switch execution time
        add c "non-safety" containers
        call the local Go script
        PRINT script execution time
```

Listing 4.2: Testing algorithm

- Given a number of containers (c), clean the environment of all existing containers, and add a safety-critical container
- For both the kill switch API and external script:
  - add a number of non-safety-critical containers
  - get the current time
  - call the kill switch and wait for process to complete
  - calculate and print the execution time

Listing 4.2 further describes the testing process.

Using 20 for the number of containers, including the zero-case where no containers are instantiated, and 5 for the number of iterations, we repeated the above process 4 times. This led to twenty iterations for each number of containers. We performed each iteration at various points over the course of a few days to mitigate the variability in test operating environment. Although the container images were not of significance in this implementation, it might

be of interest they all used the Ubuntu image. The test operating environment was a laptop with an Intel i7-8550U, 8GB of DDR4 RAM at 2133MHz, running Ubuntu Desktop 18.04.

Using the test results, we calculated the average execution and standard deviation of execution time for both the kill switch API and Go script. Using these averaged results, we then calculated the time deltas between the kill switch and Go script execution times $(kill\ switch\ execution\ time - Go\ script\ execution\ time)$. Data sets are available on request. Overall, in 13/21 runs (61%), the kill switch was faster. However, all results were within the margin of error, so this result is essentially meaningless – variability can be attributed to a variety of factors such as operating environment. On average, it took approximately 0.32s to shut down each container.

4.2.1 Discussion

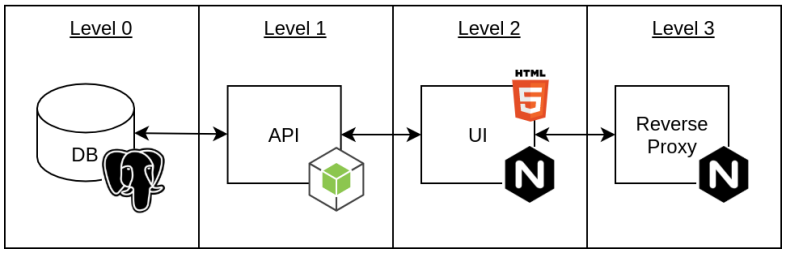
The main takeaways were that the API implementing the mode change was straightforward to write, it had little to no negative impact on performance because it leveraged existing backend systems, that non-essential containers were indeed killed, and that the containers were killed in a comparable amount of time to an external script using the same APIs.

A criticism of this approach is that this architecture pattern consideration should be addressed by an external container orchestration layer, as opposed to the engine layer. To respond, orchestration implementations vary significantly, and the pattern of implementation may not be consistent. Furthermore, orchestration layers may be unreachable or non-functional in the events surrounding the requirement to switch to a safety-critical mode. Including these considerations at the engine level allow for fewer external dependencies – the engines are integral, whereas orchestration is not.

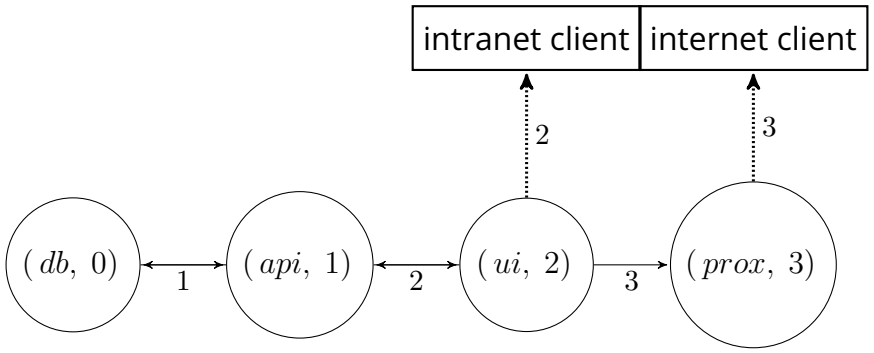
This approach demonstrates a different method of implementing the mode change pattern – in the virtualization engine itself.

4.3 Basic Example

In this scenario, we built a simple model-view-controller (MVC) style web application using the microservice architecture pattern, and the kill switch pattern. A database hosts a data repository for a model, the controller allows access of the model's properties and storage, and the view presents the information received about the model from the controller. This application consists of four parts, as outlined in Figure 4.3a.



(a) A high-level overview of each container, with a corresponding level.



(b) A graph representation of the resultant application architecture from the first example.

Figure 4.3: Diagrams of the basic example

The application itself is is essentially a mock-up, loosely based on a "to do" app. There are a set of tasks, that are either complete or incomplete. All interactions happen programatically with no actual user input available via the user interface. The database uses PostgreSQL and hosts a single table and database. The node.js server connects to the database, makes queries against it, and serves results via an Express representational state transfer (REST) API. On instantiation, the user interface server generates an HTML page. The page

is generated using a simple template, that is populated with results of GET requests to the API server. This is intended to simulate, or imitate, server-side rendering as seen in UI frameworks and libraries like React or Angular. The Nginx server then serves the 'server-side rendered' static HTML file via port 8090. The reverse proxy, as the name suggests, proxies connections at port 80 on behalf of the user interface container's port 8090. In this scenario, port 8090 is be used as an 'intranet' access port, and port 80 is used as an Internet access port.

As mentioned, each component of the application follows the microservice architecture pattern [6]. Each element of the application is separated into its own 'microservice'. As an alternative, if the application were to be built using 'traditional' monolithic architecture, the API and user interface would be written and hosted on the same server at the very least, and at the very worst, tightly coupled. An example may be a PHP server that has in-line database queries on rendered web pages.

To help explain how this is an example of microservices, the UI service could easily be redeveloped internally alongside the existing UI, without affecting service. The UI service could then be deployed internally, without adversely affecting the existing UI service. Similarly, a replacement API could be developed along side the existing one, etc.

Of importance is that the application is structured to follow the kill switch pattern. Each microservice of application has been labelled based on their dependency and functionality. For example, without the database, the application would not function. Similarly, without the API, the application would not function. However, the exposed functionality of the database allows some level of functionality of the app. As such, the database is deemed 'level 0', or the essential service, and the API as 'level 1'. The user interface is classed as 'level 3' as it provides a view of the application functionality. The API provides the functionality of interacting with the model, and the UI renders that information in order to view it in a browser. The levels, services, and communication paths are shown in Figure 4.3b, which serves as a formalization of the graph in Figure 4.3a.

The `docker-compose.yaml` file, defines the levels as part of the `build` definition. An excerpt of this is shown in Listing 4.3, which also further describes how

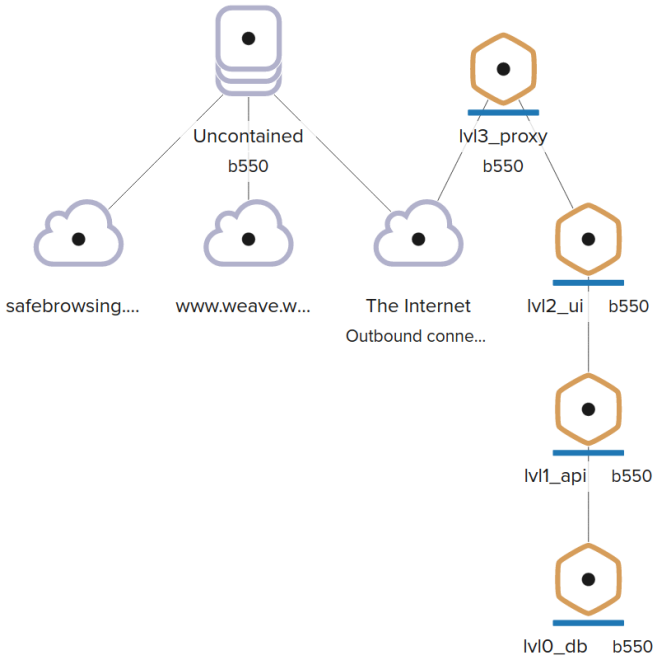
each microservice is deployed. Of interest are the exposed ports, the environment variables used by each microservice, and the names of each.

Weaveworks Scope[2] is software used to visualize container deployments in a variety of infrastructure configurations, including Kubernetes deployments. In this case, we used Scope on a single host to view the deployed containers as shown in Figure 4.4a. The hexagons represent a container running on the device, and are labelled with the container's name. The horizontal lines underneath the hexagons represent a network that each container belongs to. The lines connecting the hexagons represent network communication. The clouds represent separate communication endpoints, and the stack of squares represent a device. We deployed the containers using 'docker-compose', which starts all necessary containers.

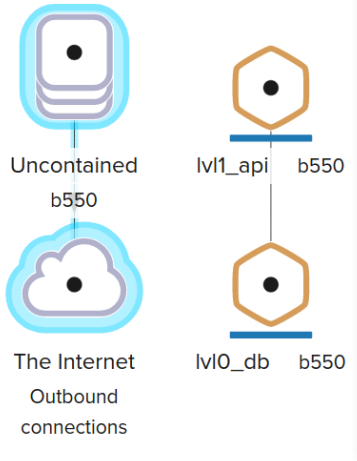
[2] `https://github.com/weaveworks/scope`

```
services:
  lvl0_db:
    build:
      context: ./lvl0/db
      labels:
        level: 0
    container_name: lvl0_db
[...]
  lvl1_api:
    build:
      context: ./lvl1/api
      labels:
        level: 1
    container_name: lvl1_api
    environment:
      API_PORT: 8080
      DB_ADDR: lvl0_db
      DB_PORT: 5432
[...]
  lvl2_ui:
    build:
      context: ./lvl2/ui
      labels:
        level: 2
    container_name: lvl2_ui
    ports:
      - '8090:80'
    environment:
      API_ADDR: lvl1_api
      API_PORT: 8080
[...]
  lvl3_proxy:
    build:
      context: ./lvl3/internet-proxy
      labels:
        level: 3
    container_name: lvl3_proxy
    ports:
      - '80:80'
[...]
```

Listing 4.3: docker-compose.yaml excerpt



(a) In this figure, the basic architecture of the service is shown as actually deployed by using Weaveworks Scope.



(b) Validation of the mode change functionality

Figure 4.4: Architecture visualization of before and after the mode change

We verified the application was returning the expected results from the 'external' port 80, and the 'internal' port 8090 in Listing 4.4. Each `cURL` request returned a 200, meaning that both the ports were actively allowing connections and serving content.

```
$ curl -I http://127.0.0.1:8090 && curl -I http://127.0.0.1:80
HTTP/1.1 200 OK
Server: nginx/1.19.4
[...]

HTTP/1.1 200 OK
Server: nginx/1.19.4
[...]

$ ./killswitch.py 2
{'level': 2, 'verbose': False}
INFO:root:Min/max/active level: 0/3/3
INFO:root:Level 3 containers state change:
INFO:root:container: /lvl3_proxy state: killed
INFO:root:Level 2 containers state change:
INFO:root:container: /lvl2_ui state: none
INFO:root:Level 1 containers state change:
INFO:root:container: /lvl1_api state: none
INFO:root:Level 0 containers state change:
INFO:root:container: /lvl0_db state: none

$ curl -I http://127.0.0.1:8090 && curl -I http://127.0.0.1:80
HTTP/1.1 200 OK
[...]

curl: (7) Failed to connect to 127.0.0.1 port 80: Connection refused
```

Listing 4.4: Killswitch script result for basic example

The script performs the functionality described in Section 3.5, with relevant code outlined in Listing 4.5. To summarize, the script groups the containers, finds the minimum, maximum, and current level the application is running at, and starts killing all containers until it reaches the target level. The script ran on the same host as the Docker engine, which allowed the Docker Python SDK to connect directly to the Unix socket HTTP API. This caused the proxy container

We called the kill switch script to set the application to a lower level; from level 3 to level 2. The mode change found a container at level 3, and killed it. We verified that the container that was killed was the proxy using cURL, and verified that the UI service was still available on port 8090. We then repeated the call using level 1 as the target state, as shown in Listing 4.4. The image shows the

container in layers 2 was killed. We verified that the UI was not available. We again repeated the call with -1, and all containers in the system were killed.

```
def do_killswitch(c: Container, target_level: int):
    if 'level' in c.attrs['Labels']:
        if int(c.attrs['Labels']['level']) <= target_level:
            if c.attrs['State'] != 'running':
                c.start()
        else:
            if c.attrs['State'] == 'running':
                c.kill()
    return c

def wait_until_all_running(client: docker.DockerClient, c_list):
[...]

def killswitch(client: docker.DockerClient, target_level: int):

    min_level, max_level, active_level = find_current_level_stats(
        client.containers.list(all=True, sparse=True))

    grouped_c_lists = group_by_level(
        client.containers.list(all=True, sparse=True))

    if target_level >= active_level:
        for level in sorted(grouped_c_lists.keys()):
            res = [do_killswitch(c, target_level)
                   for c in grouped_c_lists[level]]

            if level <= target_level:
                wait_until_all_running(client, grouped_c_lists[level])

    else:
        for level in sorted(grouped_c_lists.keys(), reverse=True):
            res = [do_killswitch(c, target_level)
                   for c in grouped_c_lists[level]]
```

Listing 4.5: killswitch.py script excerpt

4.3.1 Discussion

Of note, this architecture is similar to the implementation described by Morabito et al. [33], such that it contains a hierarchy of functionality.

By dynamically destroying containers, we were able to remove the user interface from the application. This still allowed the API server and database to run. However, because the API server and database server were not exposed to the host network, they were not accessible. In this case, the system continued to run, but lost all 'external' functionality.

We were able start levels via the mode change as well. Because the health check functionality was defined in the docker-compose file, we were able to wait until all containers in a specific level were healthy before starting the next level.

While this is a simple example, it demonstrates a few things. First, if the application were to be under attack, the administrator would just have to call the script to prevent further calls from reaching the UI service. Because the application was built using MSA, we were also able to decide which services would be placed at specific levels. If the application had been built with the database, API server, and user interface all on one container, this level of control would not be possible.

The implementation of the script also allows for remote connectivity as well. If the Docker engine exposed its management interface over a TCP socket, as opposed to a Unix socket, the script would be able to remotely control the containers on the system. In some respects, this is a method of container orchestration. In other words, we use the pattern to define how the containers are to be orchestrated.

4.4 Intrusion Detection System Example

In this realization, we continue with the basic example 'MVC-via-MSA' application. However, we introduce a more realistic method of triggering the kill switch: an intrusion detection system (IDS).

An intrusion detection system listens to packets crossing a network. Usually, the IDS targets a network interface to monitor. The IDS then categorizes the

traffic it observes based on signatures. If the signature matches a policy in a set of defined rules, the IDS can then perform different actions such as creating alerts or blocking the traffic. In our case, we are using the IDS to monitor the traffic of our host interface. However, this could be configured to only monitor a specific container network, or even a specific container.

We also implemented monitoring and logging infrastructure for the IDS. Namely, we configured a Suricata[3], Elasticsearch[4], Logstash[5], and Kibana[6] (SELK) stack. Logstash is used to ingest data from multiple sources, format it, and then send it to a storage solution. Elasticsearch is an analytics engine used to store, search, analyze, and transform structured and unstructured data. Kibana is a visualization engine for Elasticsearch, which allows a user to create dashboards and visualizations of data. In addition, we configured and deployed Evebox, a purpose-built user interface for viewing events and alerts from Suricata.

Suricata writes alerts and events to a file called `eve.json`. Logstash ingests the logs from `eve.json`, formats them, and sends them to Elasticsearch. Kibana then creates dashboards of IDS statistics, alerts, and other useful visualizations based on the logging information in Elasticsearch. This was implemented in order to help during the development of the kill switch service, and in order to create a more 'thorough' example implementation. This will be explored in Section 4.4.1.

We also introduce a 'kill switch' service, hosted in a container. We used the same script from Section 4.3 to perform the mode change operations. We then developed a monitoring script that used the existing Python module to perform the actions based on events generated by Suricata. The monitoring script reads the logs that Suricata writes to. If an alert matches a rule defined in the kill switch service, the kill switch service performs a mode change.

We configured Suricata to listen on the host's network interface, and wrote a custom rule to monitor for TCP packets coming from an external network with the content 'KILLSWITCH'. This was intended to represent an ambiguous, undefined attack. If the rule is triggered, Suricata raises an alert and writes it to `eve.json`.

[3] `https://suricata-ids.org/`
[4] `https://www.elastic.co/what-is/elasticsearch`
[5] `https://www.elastic.co/logstash`
[6] `https://www.elastic.co/kibana`

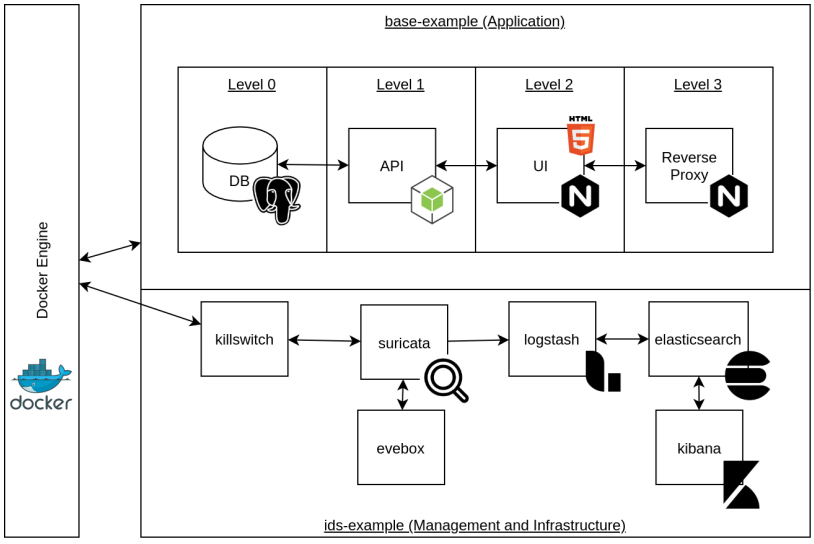


Figure 4.5: A high-level overview of each container in the IDS and example system.

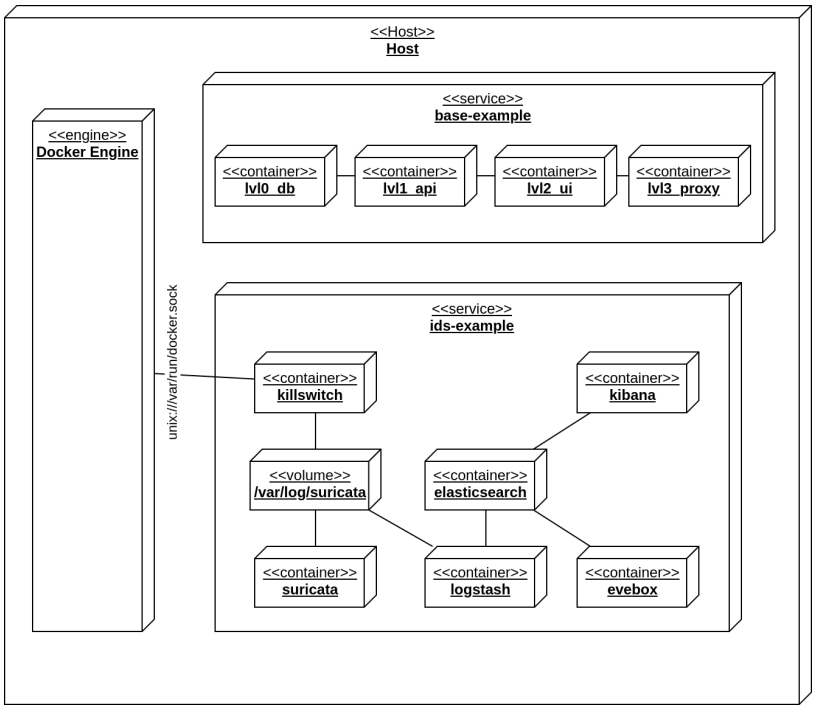


Figure 4.6: A deployment diagram of monitoring infrastructure and the example application. Each set of microservices combine to make a 'service'. In this case, we have the 'base-example' service and the 'ids-example' service.

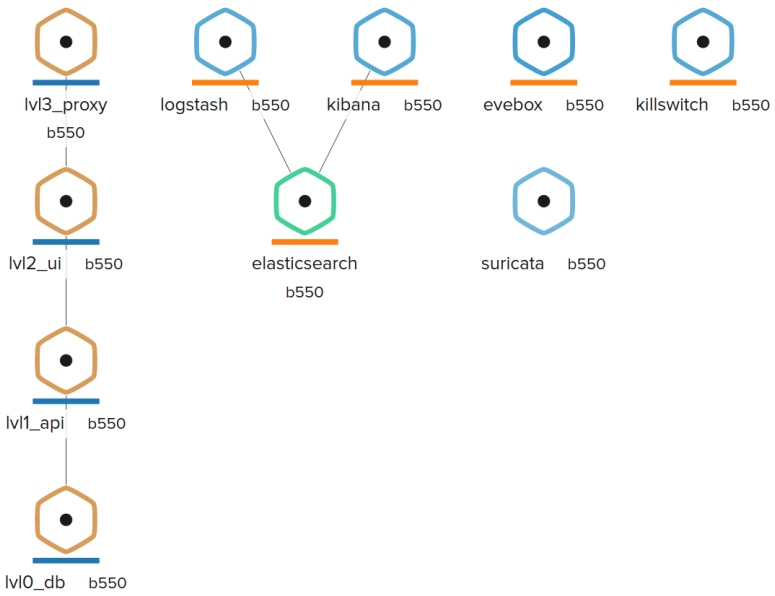


Figure 4.7: A Weaveworks Scope visualization of the IDS example.

A high-level diagram and deployment diagram are shown in Figures 4.5 and 4.6. While our infrastructure and application are all running on the same machine, this scenario was designed to simulate a more realistic approach to IoT MSA applications. In this case, the MVC application would be running on an IoT device. The IDS services would be running on an edge device, and would inspect network traffic of the downstream IoT device for events causing a mode change. In this scenario, the monitoring infrastructure is outside the control of the kill switch pattern realization; that is, the infrastructure is not placed in levels. It will not be affected by invocation of the mode change. This topic is explored in Section 4.4.1.

We began by starting the containers with docker-compose, which is shown in the Scope visualization in Figure 4.7. This started IDS monitoring and logging services, the kill switch service, as well as the application microservices.

We verified Suricata was monitoring for traffic, and verified that all containers started successfully. We then sent a cURL command from an external system to the host port 80 with the content 'KILLSWITCH_2'. This caused Suricata to detect the request as seen in Listing 4.6, which triggered an alert. The alert was written to `eve.json`. The alert was read by the kill switch service, which

performed the mode change to level 2. This disabled the proxy server for the application, which in turn closed port 80. We retried sending the cURL to the host from the external machine, and it was unsuccessful.

The only way to re-enable the services after the mode change was via manual intervention. The system had effectively locked out any further attacks.

```
INFO:root:=========================
INFO:root:Starting
INFO:root:=========================
INFO:root:waiting for eve.json file...
INFO:root:eve.json file found, continuing...
INFO:root:Starting listener on eve_path...
INFO:root:Heard alert!
Level: 2 match:
KILLSWITCH_2 line:
{"timestamp":"2020-11-05T19:49:36.954633+0000","flow_id
   ":74937906789818,"in_iface":"enp42s0","event_type":"alert","src_ip
   ":"192.168.9.16","src_port":42164,"dest_ip":"192.168.9.28","
   dest_port":80,"proto":"6","tx_id":0,"alert":{"action":"allowed","
   gid":1,"signature_id":2002824,"rev":10,"signature":"ET POLICY CURL
    User Agent","category":"Attempted Information Leak","severity
   ":2,"metadata":{"created_at":["2010_07_30"],"updated_at":["2010
   _07_30"]}},"http":{"hostname":"192.168.9.28","url":"/","
   http_user_agent":"curl/7.68.0","http_content_type":"text/html","
   http_method":"POST","protocol":"HTTP/1.1","status":405,"length
   ":157,"http_request_body_printable":"KILLSWITCH_2",
[...]
"app_proto":"http","flow":{"pkts_toserver":4,"pkts_toclient":3,"
   bytes_toserver":430,"bytes_toclient":520,"start":"2020-11-05T19
   :49:36.949690+0000"},
[...]
"payload_printable":"POST / HTTP/1.1\r\nHost: 192.168.9.28\r\nUser-
   Agent: curl/7.68.0\r\nAccept: */*\r\nContent-Length: 12\r\nContent
   -Type: application/x-www-form-urlencoded\r\n\r\nKILLSWITCH_2","
   stream":1}

INFO:root:Min/max/active level: 0/3/3
INFO:root:Level 3 containers state change:
INFO:root:container: /lvl3_proxy state: killed
INFO:root:Level 2 containers state change:
INFO:root:container: /lvl2_ui state: none
INFO:root:Level 1 containers state change:
INFO:root:container: /lvl1_api state: none
INFO:root:Level 0 containers state change:
INFO:root:container: /lvl0_db state: none
```

Listing 4.6: The kill switch service has read the alert from the log and initiates the mode change

4.4.1 Discussion

This architecture is similar to those outlined in Section 2.4.1, such that there is a clear separation between the 'IoT' device domain, and a 'cloud' or 'edge' system system domain.

Strictly speaking, only the kill switch service, and Suricata services are required from the ids-example. However, this allows us to explore the possibilities of this configuration. In the actual event of a scenario causing a mode change, there are a few different possibilities of how to handle the containers that are actually killed.

The most obvious configuration is the one we have outlined - only the application services on the IoT device would be killed. This is what we discussed in the above section.

A different scenario is that all services have implemented the kill switch pattern. In this case, the base example would remain unchanged. The design decision comes with the IDS system.

In this case, there is a clear hierarchy of dependency. The killswitch service is essential. It can function with or without Suricata via manual invocation. In fact, it may be unwise to place it under a level. If the kill switch service is destroyed, mode change functionality is lost.

The next step of dependency, in terms of intended functionality, is the Suricata service. It has its own local storage, so it has no other dependencies. Next, the ELK stack has its own set of dependencies. Elasticsearch is a datastore, and Logstash writes data to it. The difficulties arise because Elasticsearch is the datastore, but it cannot receive information without Logstash. In addition, Elasticsearch has two upstream dependencies: Evebox and Kibana.

The design trade-offs between placing Elasticsearch and Logstash at different levels in this case are minimal. Because they are tightly coupled, it makes sense to place them at the same level. So, that leaves us with the following order of dependency:

- the kill switch service, which we will not define a level for
- Suricata, defined as level 0
- Elasticsearch and Logstash, defined as level 1

- and Kibana and Evebox, defined as level 2

However, we have two distinct 'areas of deployment': the IoT device and the edge device. The design decision becomes whether to apply the mode change to all services across both areas of deployment, or if they should remain separate. This is up to preference; by treating them all as one system, a higher level of 'security' is achieved. By treating them as different, more granular control is achieved. Both are acceptable, and both can co-exist. This would be an example of using the bulkhead and kill switch patterns in conjunction.

However, one more point of interest is a little less obvious: the dependencies across the two systems, or between the two systems in general. Objectively speaking, the application is dependent on the security provided by the IDS. Another design decision is if the kill switch pattern should be applied to the two resulting services.

4.5 Discussion

Via these examples, we have gained several insights.

The pattern is *feasible, and addresses a design problem*. The pattern is *an effective tool for structuring applications, and guiding MSA application design*. It addresses the problem of dependency in MSA architectures by enforcing a clear hierarchy and levels. However, this does not mean that an application is required to follow a top-down approach to development and dependency.

There are *a variety of 'locations' to implement the mode change mechanism*: a script, a service, or a modification to the virtualization method.

An external script is simple, and effective. It can be used either on device, or remotely. It is flexible, allows for easy customization, and is an effective method for an operator or system administrator to invoke a mode change. However, it is difficult to integrate with service deployments, or with automation tools. The script requires direct access to the underlying operating system, which may not exist in the case of a locked-down system.

A service is slightly more difficult to implement than a script, but is more portable. All dependencies are packaged into the service, which allows for the operation mechanism to operate the same way, every time. It allows for automated deployment with the rest of an MSA application. However, the service

is required to be deployed either with the service or as a system service. Furthermore, it requires privileged access to the virtualization method in order to directly control the state of microservices. This would be the primary method of integrating the mode change with a service orchestrator.

Modification of the virtualization method is outside the scope of a typical microservice application, and is not practical. An example is the modification of the Docker engine. However, it allows for controlled, repeatable, and efficient implementation of the mode change. It also provides a centralized point to perform the mode change, without having to expose the engine to low-level access. It allows for easier integration with service orchestration. However, creating a 'modified' virtualization engine strays from the upstream development branch, and requires maintenance to remain up-to-date. It is also subject to any changes that may occur at this level.

Realization of the pattern can be very simple, or very complex. As discussed in Section 4.4.1, there are a variety of considerations that have to be made when implementing the kill switch pattern. The 'basic' example was objectively straight-forward. The IDS example introduced significant complexity, especially as it simulated a multi-device scenario. This alludes to the spiraling complexity discussed as a challenge in Section 3.6.6.

5. Conclusions

This thesis began by asking how to address Internet of Things device security, maintainability, and planned-obsolescence by using microservice architecture. In doing so, we found new and growing areas of research in MSA-based IoT application architectures and MSA dependency. We investigated the state-of-the-art in terms of virtualization, containers and their patterns, cloud design patterns, IoT architectures and MSA architectures, and IoT container performance. We found that while general cloud patterns exist for MSA, there are none that directly address MSA-based IoT devices and systems. We found evidence of architectures that had implemented elements of what we refer to as the *kill switch pattern*. We identified, conceptualized, defined, and proposed the kill switch pattern, a pattern to organize and reduce of functionality of MSA applications in a controlled way. The pattern addresses design problems revolving around microservice dependency, and a 'soft' multi-level version of safe state design. We proposed a basic mode change algorithm to further utilize the kill switch pattern. We analyzed the pattern in the context of level definition, Internet-of-Things devices, communication and dependency, related patterns, and it's limitations and challenges. We created several scenarios, ranging from a simple MSA application architecture realizing the pattern, to an IDS implementation that invoked a mode change, to modification of the Docker engine to evaluate feasibility of the approach.

However, as described in Section 3.6.6, there are trade-offs and limitations when using this pattern. The kill switch pattern is best suited for applications with discrete critical and non-critical functionality. Use of the pattern can lead to spiraling complexity. The pattern is best suited for limited scope MSA application, such as those on IoT devices. The pattern is best suited for architectures with hierarchical dependencies, with few or no cyclical dependencies.

The pattern is best utilized with a mode change mechanism. The mode change is subject to the length of time to start each level.

In addition, there are limitations of our approach of demonstrating efficacy via example realizations. To begin, our implementation by example does not include metric collection of any sort. Analysis of the pattern implementation is based via examples, and is hard to quantify. This requires further investigation. Another limitation is that the example realizations are aimed only at single-device implementations. It would have been interesting to demonstrate the spiraling complexity of implementing the kill switch on a multi-device system, or an all-encompassing IoT system architecture. Furthermore, we only briefly discuss implementation on a 'full' IoT system in Section 4.4.1. This suggests that as more examples are developed, further nuances and studies can be conducted on efficacy and feasiblity of the kill switch pattern and associated mode change mechanism. An additional point of concern are the actual implementations. Developers implement things in a variety of ways, but can lead to the same result. Architectures are no different. It is prudent to assume that the kill switch pattern would not be effective for all implementation styles, and again, requires further investigation.

As is hopefully evident from the discussion in Section 4.4.1, the proposed pattern is versatile. While the concept itself is fairly straight-forward, it has the ability to be quite powerful if realized in a meaningful way. As such, we assert that this pattern is a feasible method of addressing the original issue of IoT security, maintainability, and planned-obsolescence with respect to the analysis in Section 3.6.

5.1 Future Work

There are several areas of future work:

- investigation of the kill switch pattern in the safety-critical domain,
- applicability and adaptation of safety-critical patterns as related to the MSA domain,
- development of frameworks and tooling for easier implementation of the kill switch pattern,

- investigation of implications of the pattern to the cloud domain,
- graph-based dependency tracking and management of microservice architecture

We assert that future work in the area of container-based IoT architecture patterns, accounting for factors such as the ones presented in this thesis, is important for the adoption of IoT in general. By implementing this pattern on IoT devices, in conjunction with full microservice architectures, IoT devices can become more adaptable, secure, and easier to maintain.

Bibliography

[1] N. Bilton, “Nest Thermostat Glitch Leaves Users in the Cold (Published 2016),” Jan. 2016. [Online]. https://www.nytimes.com/2016/01/14/fashion/nest-thermostat-glitch-battery-dies-software-freeze.html [Accessed: 2020-11-12]

[2] S. Ragan, “DDoS knocks down DNS, data centers across the U.S. affected,” Oct. 2016. [Online]. https://www.csoonline.com/article/3133992/ddos-knocks-down-dns-datacenters-across-the-u-s-affected.html [Accessed: 2020-11-12]

[3] A. Greenberg, “Hackers Remotely Kill a Jeep on the Highway—With Me in It,” Jul. 2015. [Online]. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/ [Accessed: 2020-11-12]

[4] “Serious flaws found in multiple smart home hubs: Is your device among them?” Apr. 2020. [Online]. https://www.welivesecurity.com/2020/04/22/serious-flaws-smart-home-hubs-is-your-device-among-them/ [Accessed: 2020-11-12]

[5] RightScale, “RightScale 2018 State of the Cloud Report,” Feb. 2018. [Online]. https://www.flexera.com/blog/cloud/2018/02/cloud-computing-trends-2018-state-of-the-cloud-survey/ [Accessed: 2018-12-04]

[6] J. Lewis and M. Fowler, “Microservices,” Mar. 2014. [Online]. https://martinfowler.com/articles/microservices.html [Accessed: 2019-07-23]

[7] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, “Evaluating the monolithic and the microservice architecture

pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, Sep. 2015, pp. 583–590.

[8] A. Bosche, D. Jackson, M. Schallehn, and C. Schorling, "Unlocking Opportunities in the Internet of Things," Bain and Company, Tech. Rep., Aug. 2018.

[9] B. . Company, "Cybersecurity Is the Key to Unlocking Demand in the Internet of Things," Bain and Company, Tech. Rep., Jun. 2018. [Online]. https://www.bain.com/contentassets/c07267825b6b45cca98034d2abbd850f/bain_brief_cybersecurity_is_the_key_to_unlocking_demand_in_the_iot.pdf [Accessed: 2020-04-30]

[10] "Multi-arch build and images, the simple way," Apr. 2020. [Online]. https://www.docker.com/blog/multi-arch-build-and-images-the-simple-way/ [Accessed: 2020-11-12]

[11] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[12] F. Douglis and O. Krieger, "Virtualization," *IEEE Internet Computing*, vol. 17, no. 2, pp. 6–9, Mar. 2013.

[13] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 275–287.

[14] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)," in *Proceedings of the Sixth USENIX UNIX Security Symposium*, San Jose, USA, Jul. 1996, p. 14.

[15] P.-H. Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root." in *Proceedings of the 2nd International SANE Conference*, May 2000, p. 15.

[16] P. A. Loscocco and S. D. Smalley, “Meeting Critical Security Objectives with Security-Enhanced Linux,” in *Proceedings of the 2001 Ottawa Linux Symposium*, Ottawa, Canada, Jul. 2001, p. 11.

[17] D. Price, A. Tucker, and S. Microsystems, “Solaris Zones: Operating System Support for Consolidating Commercial Workloads,” p. 14, 2004.

[18] “Namespaces(7) - Linux manual page.” [Online]. https://man7.org/linux/man-pages/man7/namespaces.7.html [Accessed: 2020-10-31]

[19] “User_namespaces(7) - Linux manual page.” [Online]. https://man7.org/linux/man-pages/man7/user_namespaces.7.html [Accessed: 2020-10-31]

[20] “Cgroups(7) - Linux manual page.” [Online]. https://man7.org/linux/man-pages/man7/cgroups.7.html [Accessed: 2020-10-31]

[21] B. Burns and D. Oppenheimer, “Design Patterns for Container-based Distributed Systems,” in *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016. [Online]. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns [Accessed: 2020-01-23]

[22] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, “Supporting Microservice Evolution,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 539–543.

[23] A. Akbulut and H. G. Perros, “Performance Analysis of Microservice Design Patterns,” *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, Nov. 2019.

[24] “Sidecar pattern - Cloud Design Patterns.” [Online]. https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar [Accessed: 2020-11-02]

[25] “Circuit Breaker pattern - Cloud Design Patterns.” [Online]. https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker [Accessed: 2020-11-02]

[26] “Ambassador pattern - Cloud Design Patterns.” [Online]. https://docs.microsoft.com/en-us/azure/architecture/patterns/ambassador [Accessed: 2020-11-02]

[27] “Bulkhead pattern - Cloud Design Patterns.” [Online]. https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead [Accessed: 2020-11-02]

[28] P. Hodgson, “Feature Toggles (aka Feature Flags).” [Online]. https://martinfowler.com/articles/feature-toggles.html [Accessed: 2020-11-02]

[29] “Bliki: CanaryRelease,” Jun. 2014. [Online]. https://martinfowler.com/bliki/CanaryRelease.html [Accessed: 2020-11-02]

[30] C. Santana, B. Alencar, and C. Prazeres, “Microservices: A Mapping Study for Internet of Things Solutions,” in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, Nov. 2018, pp. 1–4.

[31] B. Butzin, F. Golatowski, and D. Timmermann, “Microservices approach for the internet of things,” in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2016, pp. 1–6.

[32] E. Al-Masri, “Enhancing the Microservices Architecture for the Internet of Things,” in *2018 IEEE International Conference on Big Data (Big Data)*, Dec. 2018, pp. 5119–5125.

[33] R. Morabito, R. Petrolo, V. Loscrì, N. Mitton, G. Ruggeri, and A. Molinaro, “Lightweight virtualization as enabling technology for future smart cars,” in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, May 2017, pp. 1238–1245.

[34] V. Noronha, E. Lang, M. Riegel, and T. Bauschert, “Performance Evaluation of Container Based Virtualization on Embedded Microprocessors,” in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, Sep. 2018, pp. 79–84.

[35] P. Mendki, “Docker container based analytics at IoT edge Video analytics usecase,” in *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*, Feb. 2018, pp. 1–4.

[36] B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, and O. Hong Hoe, “Evaluation of Docker as Edge computing platform,” in *2015 IEEE Conference on Open Systems (ICOS)*, Aug. 2015, pp. 130–135.

[37] C. A. Garcia, M. V. Garcia, E. Irisarri, F. Pérez, M. Marcos, and E. Estevez, "Flexible Container Platform Architecture for Industrial Robot Control," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, Sep. 2018, pp. 1056–1059.

[38] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, "Orchestration of Microservices for IoT Using Docker and Edge Computing," *IEEE Communications Magazine*, vol. 56, no. 9, pp. 118–123, Sep. 2018.

[39] S. Noor, B. Koehler, A. Steenson, J. Caballero, D. Ellenberger, and L. Heilman, "IoTDoc: A Docker-Container Based Architecture of IoT-Enabled Cloud System," in *Big Data, Cloud Computing, and Data Science Engineering*, ser. Studies in Computational Intelligence, R. Lee, Ed. Cham: Springer International Publishing, 2020, pp. 51–68.

[40] C. Hobbs, *Embedded Software Development for Safety-Critical Systems*. Auerbach Publications, Sep. 2017.

[41] D. Lennick, A. Azim, and R. Liscano, "Container-Based Internet-of-Things Architecture Pattern: Kill Switch," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, May 2020, pp. 74–78.

[42] S. Esparrachiari, T. Reilly, and A. Rentz, "Tracking and Controlling Microservice Dependencies: Dependency management is a crucial part of system and software design." *Queue*, vol. 16, no. 4, pp. Pages 10:44–Pages 10:65, Aug. 2018.

[43] S.-P. Ma, C.-Y. Fan, Y. Chuang, I.-H. Liu, and C.-W. Lan, "Graph-based and scenario-driven microservice analysis, retrieval, and testing," *Future Generation Computer Systems*, vol. 100, pp. 724–735, Nov. 2019.

[44] M. Taneja, "Introducing Docker Engine 18.09," Nov. 2018. [Online]. https://www.docker.com/blog/introducing-docker-engine-18-09/ [Accessed: 2020-12-16]

Appendices

A. Environment Specifications

A.1 System Specifications

```
user@b550:~ $ uname -a
Linux b550 5.4.0-52-generic #57-Ubuntu SMP Thu Oct 15 10:57:00 UTC
    2020 x86_64 x86_64 x86_64 GNU/Linux
user@b550:~ $ neofetch --off
user@b550
---------
OS: Ubuntu 20.04.1 LTS x86_64
Host: MS-7C95 1.0
Kernel: 5.4.0-52-generic
Uptime: 3 days, 6 hours, 8 mins
Packages: 2448 (dpkg), 6 (flatpak), 12 (snap)
Shell: bash 5.0.17
Resolution: 1080x1920, 1920x1080, 2560x1440
DE: GNOME
WM: Mutter
WM Theme: Adwaita
Theme: Yaru-dark [GTK2/3]
Icons: Yaru [GTK2/3]
Terminal: gnome-terminal
CPU: AMD Ryzen 7 3700X (16) @ 3.600GHz
GPU: NVIDIA GeForce GTX 1070 Ti
Memory: 12025MiB / 32102MiB
```

Listing A.1: System information

A.2 Docker Version

```
Client: Docker Engine - Community
 Version:           19.03.13
 API version:       1.40
 Go version:        go1.13.15
 Git commit:        4484c46d9d
 Built:             Wed Sep 16 17:02:52 2020
 OS/Arch:           linux/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.13
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       4484c46d9d
  Built:            Wed Sep 16 17:01:20 2020
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.3.7
  GitCommit:        8fba4e9a7d01810a393d5d25a3621dc101981175
 runc:
  Version:          1.0.0-rc10
  GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
```

Listing A.2: docker version output

B. Kill Switch Source Code

B.1 killswitch/Dockerfile

```
FROM ubuntu:latest

WORKDIR /app
RUN apt update
RUN apt install -y python3 python3-pip

COPY killswitch.py ./
COPY killswitch-service.py ./
COPY rules.json ./
COPY entrypoint.sh ./
RUN python3 -m pip install docker

ENTRYPOINT [ "/app/entrypoint.sh" ]
```

B.2 killswitch/entrypoint.sh

```
#!/bin/bash

rm killswitch.log
touch killswitch.log
/app/killswitch-service.py /var/log/suricata/eve.json /app/rules.json
   &

tail -F /app/killswitch.log
```

B.3 killswitch/killswitch-service.py

```
#!/usr/bin/python3

import json
import os
import time
import threading
import argparse
from pprint import pprint
import logging

import docker
import killswitch as ks

def parse_args():
    parser = argparse.ArgumentParser(
        description='Killswitch service interface')

    parser.add_argument('eve_path',
                        help='set the suricata eve.json file path')
    parser.add_argument('rule_path',
                        help='set the operation level config file path')
    parser.add_argument("-v", "--verbose",
                        help="increase output verbosity",
                        default=False,
                        action="store_true")


    return parser.parse_args()


def tail(stream_file):
    """ Read a file like the Unix command `tail`. Code from https://
       stackoverflow.com/questions/44895527/reading-infinite-stream-
       tail """
    stream_file.seek(0, os.SEEK_END) # Go to the end of file

    while True:
        if stream_file.closed:
            raise StopIteration
```

```
        line = stream_file.readline()

        yield line


def listen_for_alerts(log_path, rules_json):
    """ Read log (JSON format) and insert data in db """

    client = docker.from_env()
    rules = rules_json["rules"]

    with open(log_path, "r") as log_file:
        for line in tail(log_file):
            for level, criteria in rules.items():
              for c in criteria:
                if c in line:
                  logging.info("Heard alert!\nLevel: {} match:\n{} line
                      :\n{}".format(level, c, line))
                  ks.killswitch(client, int(level))


if __name__ == "__main__":

    args = parse_args()

    if args.verbose:
        logging.basicConfig(level=logging.DEBUG, filename='killswitch.
            log')
        logging.debug("Debug logging enabled")
    else:
        logging.basicConfig(level=logging.INFO, filename='killswitch.
            log')
    logging.info("=========================")
    logging.info("Starting")
    logging.info("=========================")

    logging.debug("Checking rule_path...")
    if not os.path.exists(args.rule_path):
      logging.info("rule path does not exist!")
```

```
    logging.info("waiting for eve.json file...")
    while not os.path.exists(args.eve_path):
      time.sleep(1)
    logging.info("eve.json file found, continuing...")

    logging.debug("Importing rules...")
    with open(args.rule_path, "r") as rule_file:
      rules_json = json.load(rule_file)

    logging.info("Starting listener on eve_path...")

    generator = threading.Thread(
        target=listen_for_alerts,
        args=(
          args.eve_path,
          rules_json,
        )
    )
    generator.start()
```

B.4 killswitch/killswitch.py

```
#!/usr/bin/python3
import sys
import os
import time
import argparse
import logging
from pprint import pprint

import docker
from docker.models.containers import Container


def parse_args():
    parser = argparse.ArgumentParser(description='Killswitch CLI
       interface')
    parser.add_argument('level',
                        type=int,
```

```
                    help='set the operation level')
    parser.add_argument("-v", "--verbose",
                    help="increase output verbosity",
                    default=False,
                    action="store_true")
    return parser.parse_args()


def find_current_level_stats(c_list: [Container]):
    curr_max = 0
    curr_min = sys.maxsize
    curr_active = 0

    for c in c_list:
        if 'level' in c.attrs['Labels']:
            curr_val = int(c.attrs['Labels']['level'])

            if curr_val > curr_max:
                curr_max = curr_val

            if curr_val < curr_min:
                curr_min = curr_val

            if curr_val > curr_active and c.attrs['State'] == 'running
                ':
                curr_active = curr_val

    return curr_min, curr_max, curr_active


def group_by_level(c_list: [Container]):
    grouped_c_lists = {}

    for c in c_list:
        if 'level' in c.attrs['Labels']:
            curr_val = int(c.attrs['Labels']['level'])
            if curr_val not in grouped_c_lists:
                grouped_c_lists[curr_val] = [c]
            else:
                grouped_c_lists[curr_val].append(c)

    return grouped_c_lists
```

```
def do_killswitch(c: Container, target_level: int):
    state_change = "none"

    if 'level' in c.attrs['Labels']:
        if int(c.attrs['Labels']['level']) <= target_level:
            if c.attrs['State'] != 'running':
                c.start()
                state_change = "started"
        else:
            if c.attrs['State'] == 'running':
                c.kill()
                state_change = "killed"

    return c, state_change


def wait_until_all_running(client: docker.DockerClient, c_list):

    done = False
    while not done:

        done = True

        for c in c_list:
            curr_c = client.containers.get(c.id)
            # pprint(vars(curr_c))
            if not curr_c.attrs['State']['Running'] or curr_c.attrs['
                State']['Health']['Status'] != 'healthy':
                logging.debug("waiting on {}".format(curr_c.attrs['Name
                    ']))
                logging.debug("state: {}".format(
                    curr_c.attrs['State']['Status']))
                logging.debug("health: {}".format(
                    curr_c.attrs['State']['Health']['Status']))
                done = False

        if not done:
            time.sleep(1)
```

```
def killswitch(client: docker.DockerClient, target_level: int):

    # establish which levels is active, and the min/max levels
       available
    min_level, max_level, active_level = find_current_level_stats(
        client.containers.list(all=True, sparse=True))
    logging.info(
        "Min/max/active level: {}/{}/{}".format(min_level, max_level,
           active_level))

    # group the containers by their level
    grouped_c_lists = group_by_level(
        client.containers.list(all=True, sparse=True))

    # start from lower -> higher levels
    if target_level >= active_level:
        if target_level > max_level:
            logging.info(
                "Target level above maximum available level. Verifying
                   containers are started.")

        if target_level == active_level:
            logging.info(
                "Target level is equal to current level. Verifying
                   containers are started.")

        for level in sorted(grouped_c_lists.keys()):
            res = [do_killswitch(c, target_level)
                   for c in grouped_c_lists[level]]

            if level <= target_level:
                wait_until_all_running(client, grouped_c_lists[level])

            logging.info("Level {} containers state change: ".format(
               level))
            for k, v in res:
                logging.info("container: {} state: {}".format(
                    k.attrs['Names'][0], v))

    # start from higher -> lower level
    else:
        for level in sorted(grouped_c_lists.keys(), reverse=True):
```

```
            res = [do_killswitch(c, target_level)
                   for c in grouped_c_lists[level]]
            logging.info("Level {} containers state change: ".format(
                level))
            for k, v in res:
                logging.info("container: {} state: {}".format(
                    k.attrs['Names'][0], v))


if __name__ == '__main__':

    args = parse_args()
    print(vars(args))

    if args.verbose:
        logging.basicConfig(level=logging.DEBUG)
        logging.debug("Debug logging enabled")
    else:
        logging.basicConfig(level=logging.INFO)
    client = docker.from_env()
    killswitch(client, args.level)
```

B.5 killswitch/rules.json

```
{
  "rules": {
    "0": ["KILLSWITCH_0"],
    "1": ["KILLSWITCH_1"],
    "2": ["KILLSWITCH_2"],
    "3": ["KILLSWITCH_3"]
  }
}
```

C. Basic Example Code and Configuration

C.1 base-example/docker-compose.yaml

```
version: '3'

services:
  lvl0_db:
    build:
      context: ./lvl0/db
      labels:
        level: 0
    container_name: lvl0_db
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U docker && psql -U docker -lqt
          | cut -d \\| -f 1 | grep -qw todo"]
      interval: 1s
      timeout: 5s
      retries: 5
  lvl1_api:
    build:
      context: ./lvl1/api
      labels:
        level: 1
    container_name: lvl1_api
    environment:
      API_PORT: 8080
      DB_ADDR: lvl0_db
      DB_PORT: 5432
    depends_on:
```

```
    - lvl0_db
  healthcheck:
    test: "curl -f http://lvl1_api:8080/ || false"
    interval: 1s
    timeout: 5s
    retries: 5
lvl2_ui:
  build:
    context: ./lvl2/ui
    labels:
      level: 2
  container_name: lvl2_ui
  ports:
    - '8090:80'
  environment:
    API_ADDR: lvl1_api
    API_PORT: 8080
    API_PATH: ""
  healthcheck:
    test: "curl -f http://lvl2_ui:80/ || false"
    interval: 1s
    timeout: 5s
    retries: 5
  depends_on:
    - lvl1_api
lvl3_proxy:
  build:
    context: ./lvl3/internet-proxy
    labels:
      level: 3
  container_name: lvl3_proxy
  ports:
    - '80:80'
  healthcheck:
    test: "curl -f http://lvl3_proxy:80/ || false"
    interval: 1s
    timeout: 5s
    retries: 5
  depends_on:
    - lvl1_api
```

C.2 base-example/lvl0/db/db.sql

```
CREATE DATABASE todo;
\connect todo;

CREATE TABLE tasks(
  id SERIAL PRIMARY KEY,
  task VARCHAR(40) not null,
  is_complete BOOLEAN
);
```

C.3 base-example/lvl0/db/Dockerfile

```
FROM postgres:latest

ENV POSTGRES_USER docker
ENV POSTGRES_PASSWORD docker

ADD db.sql /docker-entrypoint-initdb.d/
```

C.4 base-example/lvl1/api/Dockerfile

```
FROM node:latest

ENV API_PORT=8080
ENV DB_ADDR=127.0.0.1
ENV DB_PORT=8080

WORKDIR /opt/app
COPY package*.json ./
COPY entrypoint.sh ./
RUN npm install
COPY . .
```

```
#ENTRYPOINT [ "/opt/app/entrypoint.sh" ]
#CMD ["sh", "-c", "node server.js"]
CMD ["sh", "-c", "/opt/app/entrypoint.sh"]
```

C.5 base-example/lvl1/api/entrypoint.sh

```
#!/bin/bash

sleep 5 # sleep to let the db start

node server.js
```

C.6 base-example/lvl1/api/package.json

```
{
  "name": "controller",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "MIT",
  "dependencies": {
    "express": "^4.17.1",
    "pg": "^8.4.1"
  }
}
```

C.7 base-example/lvl1/api/queries.js

```
// set up the db
const Pool = require('pg').Pool
const pool = new Pool({
  user: 'docker',
  host: process.env.DB_ADDR,
  database: 'todo',
  password: 'docker',
  port: process.env.DB_PORT,
})

const testConn = (request, response) => {
  pool.query('SELECT NOW()', (err, res) => {
    if (err) {
      throw err
    }
    console.log(`api to db conn (${process.env.DB_ADDR}:${process.env.
       DB_PORT}) ok`)
    //pool.end()
  })
}

const getTasks = (request, response) => {
  pool.query('SELECT * FROM tasks ORDER BY id ASC', (err, res) => {
    if (err) {
      throw err
    }
    response.status(200).json(res.rows)
  })
}

const getTaskById = (request, response) => {
  const id = parseInt(request.params.id)

  pool.query('SELECT * FROM tasks WHERE id = $1', [id], (err, res) =>
     {
    if (err) {
      throw err
    }
    response.status(200).json(res.rows)
```

```
  })
}

const createTask = (request, response) => {
  const { task, is_complete } = request.body

  pool.query('INSERT INTO tasks (task, is_complete) VALUES ($1, $2)',
     [task, is_complete], (err, res) => {
    if (err) {
      throw err
    }
    response.status(201).send(`Task added`)
  })
}

const updateTasks = (request, response) => {
  const id = parseInt(request.params.id)
  const { task, is_complete } = request.body

  pool.query(
    'UPDATE tasks SET task = $1, is_complete = $2 WHERE id = $3',
    [task, is_complete, id],
    (err, res) => {
      if (err) {
        throw err
      }
      response.status(200).send(`Task modified with ID: ${id}`)
    }
  )
}

const deleteTask = (request, response) => {
  const id = parseInt(request.params.id)

  pool.query('DELETE FROM tasks WHERE id = $1', [id], (err, ress) => {
    if (err) {
      throw err
    }
    response.status(200).send('Task deleted with ID: ${id}')
  })
}
```

```
module.exports = {
  testConn,
  getTasks,
  getTaskById,
  createTask,
  updateTasks,
  deleteTask
}
```

C.8 base-example/lvl1/api/server.js

```
#!/usr/bin/env node

const express = require('express')
const bodyParser = require('body-parser')
const app = express()

const db = require('./queries')

// setup
app.use(bodyParser.json())
app.use(
  bodyParser.urlencoded({
    extended: true,
  })
)

app.get('/', (request, response) => {
  response.json({ info: 'Node.js, Express, and Postgres API' })
})

app.get('/tasks', db.getTasks)
app.get('/task/:id', db.getTaskById)
app.post('/task', db.createTask)
app.put('/task/:id', db.updateTasks)
app.delete('/task/:id', db.deleteTask)

setInterval(db.testConn, 1000)
```

```
app.listen(process.env.API_PORT, () => {
  console.log(‘api running on port ${process.env.API_PORT}.‘)
  db.testConn()
})
```

C.9 base-example/lvl2/ui/Dockerfile

```
FROM nginx:latest

ENV API_ADDR=""
ENV API_PORT=""
ENV API_PATH=""

COPY index.html /usr/share/nginx/html/
COPY entrypoint.sh /usr/share/nginx/
COPY pinger.sh /usr/share/nginx/

CMD [ "/usr/share/nginx/entrypoint.sh" ]
```

C.10 base-example/lvl2/ui/entrypoint.sh

```
#!/bin/bash

API_URI="http://$API_ADDR:$API_PORT"

attempt_counter=0
max_attempts=10

until $(curl --output /dev/null --silent --head --fail $API_URI); do
    if [ ${attempt_counter} -eq ${max_attempts} ];then
     echo "Max attempts reached"
     exit 1
   fi
   printf ’.’
   sleep 3
```

```
done

curl -X POST -d task='test task' -d 'is_complete=false' $API_URI/task
SERVER_SIDE_GET_BASE=$(curl $API_URI)
SERVER_SIDE_GET_TASKS=$(curl $API_URI/tasks)

echo $API_URI
sed -i "s@SERVER_SIDE_GET_BASE@$SERVER_SIDE_GET_BASE@" /usr/share/
   nginx/html/index.html
sed -i "s@SERVER_SIDE_GET_TASKS@$SERVER_SIDE_GET_TASKS@" /usr/share/
   nginx/html/index.html
sed -i "s@API_URI@$API_URI@" /usr/share/nginx/html/index.html
/usr/share/nginx/pinger.sh &
exec $(which nginx) -c /etc/nginx/nginx.conf -g "daemon off;"
```

C.11 base-example/lvl2/ui/index.html

```
<h1>Server Render GET /</h1>
<p>SERVER_SIDE_GET_BASE</p>

<h1>Server Render GET /</h1>
<p>SERVER_SIDE_GET_TASKS</p>
```

C.12 base-example/lvl2/ui/pinger.sh

```
#!/bin/bash
while sleep 1; do curl $API_ADDR:$API_PORT/tasks; done
```

C.13 base-example/lvl3/internet-proxy/Dockerfile

```
FROM nginx:latest
```

```
COPY nginx.conf /etc/nginx
# RUN rm /docker-entrypoint.sh
# COPY entrypoint.sh /
# ENTRYPOINT ["/entrypoint.sh"]
```

C.14 base-example/lvl3/internet-proxy/entrypoint.sh

```
#!/bin/bash

nginx -c /etc/nginx/nginx.conf -g "daemon off;"
```

C.15 base-example/lvl3/internet-proxy/nginx.conf

```
#!/bin/bash

nginx -c /etc/nginx/nginx.conf -g "daemon off;"
```

D. IDS Example Code and Configuration

D.1 ids/docker-compose.yml

```
version: '3'

services:
  elasticsearch:
    build: elk/elasticsearch
    container_name: elasticsearch
    volumes:
      - ./elk/elasticsearch/config/elasticsearch.yml:/usr/share/
        elasticsearch/config/elasticsearch.yml
    ports:
      - "127.0.0.1:9200:9200"
      - "127.0.0.1:9300:9300"
    environment:
      ES_JAVA_OPTS: "-Xmx256m -Xms256m"
    # healthcheck:
    #   test: ["CMD-SHELL", "curl --silent --fail localhost:9200/
       _cluster/health || exit 1"]
    #   interval: 30s
    #   timeout: 30s
    #   retries: 3

  logstash:
    build: elk/logstash
    container_name: logstash
    volumes:
      - ./elk/logstash/config:/usr/share/logstash/config
```

```
    - data:/var/log/suricata
  restart: always
  ports:
    - "127.0.0.1:9600:9600"
    - "127.0.0.1:5044:5044"
  #   - "514:5000"
  #   - "514:5000/udp"
  environment:
    LS_JAVA_OPTS: "-Xmx256m -Xms256m"
  depends_on:
    - elasticsearch

kibana:
  build: elk/kibana
  container_name: kibana
  volumes:
    - ./elk/kibana/config:/usr/share/kibana/config
  ports:
    - "127.0.0.1:5601:5601"
  depends_on:
    - elasticsearch

evebox:
  build: evebox
  container_name: evebox
  ports:
    - "127.0.0.1:5636:5636"
  depends_on:
    - elasticsearch
  environment:
    ELASTICSEARCH_URL: http://elasticsearch:9200
    EVEBOX_HTTP_HOST: 0.0.0.0
  command: "/bin/evebox server -e http://elasticsearch:9200"

suricata:
  build: suricata
  container_name: suricata
  network_mode: host
  cap_add:
    - NET_ADMIN
    - SYS_NICE
    - NET_RAW
```

```
    # ports:
    #   - "7200:7200"
    environment:
      IF: enp42s0
    volumes:
      - data:/var/log/suricata

  killswitch:
    build: killswitch
    container_name: killswitch
    environment:
      PYTHONUNBUFFERED: 0
    volumes:
      - data:/var/log/suricata
      - /var/run/docker.sock:/var/run/docker.sock



volumes:
  data:
```

D.2 ids/suricata/Dockerfile

```
FROM ubuntu:latest

WORKDIR /etc/suricata

RUN apt update && \
    apt install -y software-properties-common wget apt-transport-https

# suricata
RUN add-apt-repository -y ppa:oisf/suricata-stable
RUN apt update && apt install -y suricata
ADD suricata.yaml /etc/suricata/suricata.yaml
ADD capture-filter.bpf /etc/suricata/
ADD enable.conf /etc/suricata/
ADD entrypoint.sh /etc/suricata/

# add killswitch rules
```

```
ADD killswitch.rules /var/lib/suricata/rules/

# cleanup
RUN rm -rf /var/lib/apt/lists/* && apt-get clean && apt autoremove -y

# Start the Suricata/Filebeat service when the container is started
ENTRYPOINT /etc/suricata/entrypoint.sh
```

D.3 ids/suricata/entrypoint.sh

```
#!/bin/bash

RED='\033[0;31m'
NC='\033[0m'

red_print () {
  echo -e "\n\n${RED}$1\n
     ############################################################${NC}\n\n
     "
}

red_print "Updating suricata sources..."

suricata-update update-sources

suricata-update enable-source et/open

suricata-update --enable-conf /etc/suricata/enable.conf --local=/var/
   lib/suricata/rules/killswitch.rules

red_print "Starting suricata..."
suricata -D -v -i $IF

red_print "Ouput logs to console..."
tail -F /var/log/suricata/suricata.log
```

D.4 ids/suricata/killswitch.rules

```
alert tcp $EXTERNAL_NET -> $HOME_NET ( msg:"Kill switch trigger";
   content:"GET"; http_method; content:"KILLSWITCH"; priority:1; )
```