SOCCRATES

SOC & CSIRT Response to Attacks & Threats based on attack defense graphs Evaluation Systems

D7.3
SOCCRATES Platform Best Practices Guide

Deliverable type:	Report
Contributing work packages:	WP7 – Pilot applications and evaluations
Due date of deliverable:	31st October 2022
Submission date:	31st October 2022
Dissemination level:	PU

Responsible organisation:	Vattenfall IT services Poland Sp. Z.o.o.
Editor:	Malgorzata Greń
Revision:	1.0

Abstract	This deliverable delivers lessons learned of the implementation of the SOCCRATES platform in an actual SOC environment, described in a best practice guide.
Keywords:	Lessons learned, procedures, issues, quality, co-operation, project, installation, configuration, solutions, guidance, security automation.

Management summary

This deliverable contains the lessons learned of the implementation of the platform in an actual SOC environment (based on the pilot experience) described in a best practice guide. This report includes practical guidance on:

- preparation
- installation (pointer to installation manual on github)
- configuration
- operation

The deliverable lists the lessons learned in Software architecture, installation, testing and scalability. The main target audience for this deliverable is everyone who wants to install and try the SOCCRATES platform by themselves (e.g. SOC analysts, security researchers, vendors) or anyone interested in experiences with large and complex software projects.

Contributor(s)	Federico Falconieri (TNO) Maciej Kosz (Vattenfall) Martin Eian (mnemonic) Philip Scheel (mnemonic) Fredrik Borg (mnemonic) Sebastiaan Tesink (TNO) Piotr Kijewski (Shadowserver) Irina Chiscop (TNO) Paul Smith (AIT) Bajraktari Agron (AIT) Christophe Kiennert (IMT) Susana Gonzalez Zarzosa (ATOS) Jesus Villalobos Nieto (ATOS) Ville Alkkiomäki (WithSecure)

Reviewer(s)	Paul Smith (security and content)

Security Assessment	**See deliverables table for security assessment requirements**
Approval Date	27/10/2022
Remarks	None

TABLE OF CONTENTS

1 Introduction – Rationale of this document

This section introduces the SOCCRATES project and defines the goals of this deliverable.

1.1 The SOCCRATES project

SOCCRATES (SOC & CSIRT Response to Attacks & Threats based on attack defence graphs Evaluation Systems) is an EU funded project under the Horizon2020 programme that has the following main challenge:

How can SOC and CSIRT operations effectively improve their capability in detecting and managing response to complex cyber-attacks and emerging threats, in complex and continuously evolving ICT infrastructures while there is a shortage of qualified cybersecurity talent?

The main objective of SOCCRATES is to develop and implement a security automation and decision support platform (‘**the SOCCRATES platform’)** that will significantly improve an organisation’s capability (usually implemented by a SOC and/or CSIRT) to quickly and effectively detect and respond to new cyber threats and ongoing attacks.

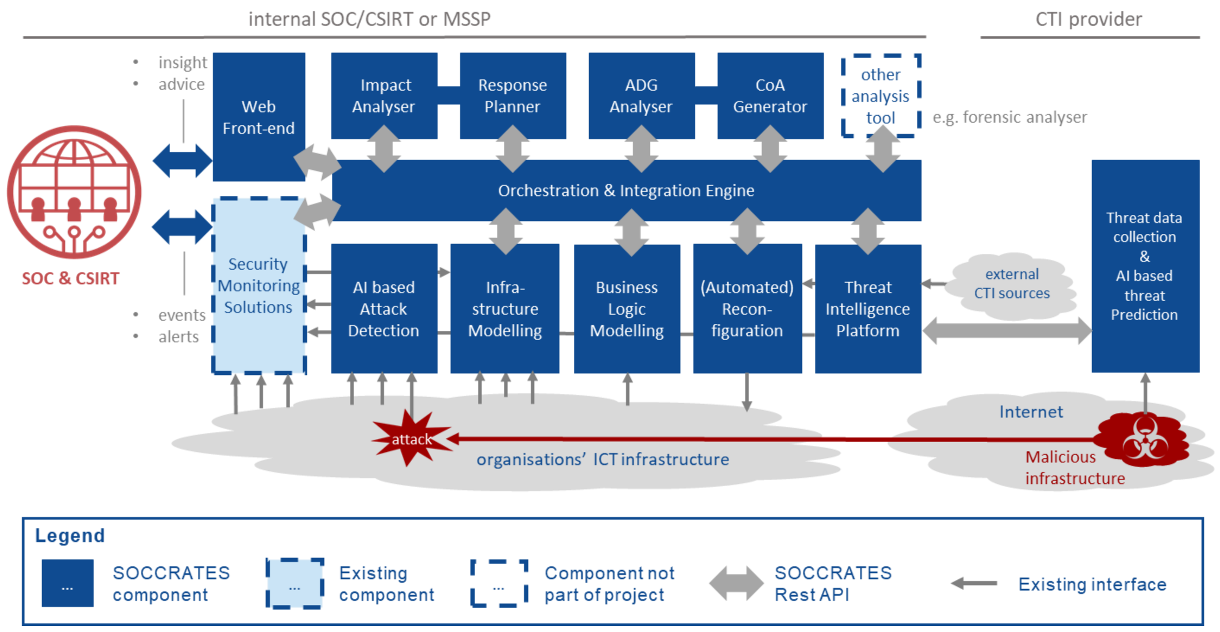

Figure 1.1 – The SOCCRATES platform

The SOCCRATES platform (see Figure 1.1) consists of an orchestrating function and a set of innovative components for automated infrastructure modelling, attack detection, cyber threat intelligence utilization, threat trend prediction, and automated analysis using attack defence graphs and business impact modelling to aid human analysis and decision making on response actions, and enable the execution of defensive actions at machine-speed.

SOCCRATES has the following concrete project objectives:

1. Deliver the SOCCRATES platform consisting of an orchestration function and a unique integration of innovative background solutions that seamlessly work together.
2. Show that the SOCCRATES platform can improve SOC operations by evaluating the SOCCRATES platform in two diverse real-life pilot environments.
3. Examine and illustrate the benefits of automation for selected SOC activities to help manage the cyber security skills gap in organizations.
4. Prepare for successful exploitation by the SOCCRATES partners of the individual innovated components and the integrated SOCCRATES platform in commercial products that are offered to the market and are available for the European (business) community.

Please visit www.SOCCRATES.eu for more information on the SOCCRATES project.

1.2 This deliverable

This deliverable describes lessons learned of the pilot preparation in the best practice guide.

1.3 Structure of this deliverable

This deliverable contains information and lessons learned of the implementation of the SOCCRATES platform described in a best practice guide. Chapter 2 describes the lessons learned in terms of software architecture, testing, software operations and scalability. The main target audience is everyone who would like to try and implement the SOCCRATES platform by themselves.

2 Lessons Learned

2.1 Software Architecture & Maturity

2.1.1 Data collection

One of the most important data element for the SOCCRATES platform is network telemetry. SOCCRATES components that operate directly on network flow records are:

- Business Impact Analyzer (BIA) – to derive dependencies between assets
- ABC Tool and L-ADS – to detect anomalies in network traffic

To avoid data duplication and standardize data format, network flow records sent to the SOCCRATES platform are stored in an Elasticsearch instance from where they are made available for other components.

There are two main elements involved in gathering network flows:

- Flow exporter – watches network traffic and creates network flow records
- Flow collector – receives and stores flow records from Flow exporter

Flow exporters are usually routers or other layer-3 devices but network flow records can be also exported by virtualization hypervisors such as VMware ESXi or dedicated software which listens on a network interface and observes traffic delivered by SPAN port or network tap. Flow collector is a software providing storage and pre-processing capabilities. There could be more than one exporter sending data to a single flow collector.

Selection of certain network flow export option usually determines the level of visibility over network traffic and consequently the detection capabilities. For instance, having network flow records only from perimeter devices won’t allow for detection of anomalies in network traffic that doesn’t pass through perimeter devices, thus limiting opportunity for lateral movement detection inside internal network segments.

Some examples of open-source software capable of network flow processing useful for the SOCCRATES platform are:

Software	Website	Selected features
Suricata	https://suricata.io/	Creates text/JSON files with flow records out of raw traffic observed (e.g. from network tap or SPAN port)
nfdump	https://github.com/phaag/nfdump	Receives standard NetFlow/IPFIX records and allows export in text/JSON format
YAF	https://tools.netsa.cert.org/yaf/	Captures network traffic and creates IP-FIX flow records

Importantly, the SOCCRATES platform expects bidirectional flows (i.e. aggregating packet/data in and out accounting in a single flow record) and the tool of choice must support this.
Flow record exports by Suricata & nfdump can be ingested to Elasticsearch using Filebeat or/and Logstash. Both options are viable and were used during SOCCRATES pilot preparation however nfdump required custom modification because default JSON output doesn't support bidirectional flows. For that reason on one of the pilot sites nfdump was recompiled and optimized for JSON output.

2.1.2 Review of the chosen Software Architecture

The SOCCRATES platform is an interconnected mesh of services, deployed as containers (with the unique exclusion of SecuriCAD). This is also known as the "distributed monolith" pattern, or rather anti-pattern as it comes with several downsides. The main one is that it forces components developers to agree and synchronize constantly on interfaces. Any change in an upstream component must be followed by a change in downstream components. If updates are not orchestrated properly this leads to integration bugs. Orchestrating development of a distributed monolith is not impossible, but it is very difficult because it requires to have someone that familiar with all the details of the system, exactly as you would in a classic software monolith. Furthermore it is not easy to maintain good observability into the system, because there is no one place to monitor the calls between the services. Finally, interconnectivity makes integration testing harder because it becomes necessary to either deploy multiple components at test time or to mock them, and both options have downsides.
With hindsight, the SOCCRATES platform could have followed an event driven microservice architecture, with components producing and consuming events to/from an event bus rather than talking directly to each other. The advantage of this architecture is that component developers only need to agree on the schema of the events, which are far less likely to change as frequently as the application programmable interfaces (APIs) between the components. Events are also much simpler to mock than (potentially chain of) components, which makes integration testing as simple as adding event related (consumption/production) unit tests. Being able to easily mock events also makes it simple to stress-test an application, for instance to evaluate performance. Event driven architecture provides a natural built-in observability solution: the event bus. Finally, event driven architecture, when done right, are also naturally scalable because more consumers can consume more events in parallel. This is not the case in the "distributed monolith" architecture. However this would have allowed to avoid only the

scalability problems encountered in the pilot sites within the newly developed components: scalability issues with existing components would have been the same.

2.1.3 Technology Readiness Level (TRL) [1]

The mnemonic and Vattenfall pilot sites are TRL 7-9[2] environments, however, while the ambition for the SOCCRATES platform was to achieve TRL 6[3]. For the project, getting the TRL 6 demonstrator environment running in full meant that the target of the original project proposal was achieved. To maximize the exposure of actual SOC analysts to the platform and its automation capabilities, however, the project sought ways to also evaluate it in higher TRL environments. To this end, a hybrid setup was established that went well beyond the original ambitions:

- At the mnemonic pilot site, the project was able to get the workflow for use case 1 (specifically the "initial access" scenario) running on actual operational infrastructure. For a limited period of time, this allowed actual analysts to conduct evaluations with the SOCCRATES platform on actual alerts that had been triaged in the mnemonic SOC. Effectively this means that selected platform capabilities were operated and trialled at TRL 7.
- The demonstrator environment was used to let analyst teams at both Vattenfall and mnemonic become acquainted with the platform in its entirety. For all of the project's use cases (1-5), therefore, both pilot teams evaluated the SOCCRATES platform in its full TLR 6 manifestation.

The experience from the pilot sites was that components with low TRL did not work as intended in the operational environments, both due to the amount of data to be processed and due to code maturity issues such as hard-coded variables that did not match the pilot infrastructure configuration. A large amount of time and effort was spent on improving the code of these components, which caused significant delays. Conversely, we did not experience code maturity issues for components that had a target TRL of 7 of higher, although some AAD components with TRL 7 or higher were not able to process the amount of data present at the mnemonic pilot site.
Our key lesson learned is that the target TRL did not match the pilot environments. In order to execute a pilot in an operational environment, *all* components to be validated must have a target TRL of at least 7.

2.2 Testing

The importance of software testing has been perhaps the most valuable lesson learned in SOCCRATES software development and operations. We have learned the hard way how difficult it is to test large, heterogeneous software projects
Test design should happen during system design, not as an afterthought. Challenges in test design can help identify architectural issues: for instance if testing a component in isolation is too difficult this can be an indicator that perhaps components are too interdependent.
In this section we review how we have performed these four categories of testing:

- unit tests: to test individual features of software components
- integration tests: to test the interaction between components

[1] https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf
[2] Technology Readiness Level (TRL) 7: System prototype demonstration in operation environment
[3] TRL 6: Technology demonstrated in relevant environment

- deployment tests: to test if software components can be deployed
- performance tests: to test software performs appropriately

2.2.1 Unit Testing

Code coverage is a measure of how many lines of code of a program is covered by a certain test. Since no code coverage requirement was given, each component developer had freedom in choosing how much effort to put in writing and maintaining unit tests. Unsurprisingly, most components did not include any unit test. This led to many bugs being discovered in integration testing, deployment testing or, more often than not, at pilot site deployment. Writing good unit tests is not trivial, and it can be very time consuming to implement them for graphical user interfaces (3 in total were developed as part of the SOCCRATES platform: SOCCRATES main frontend, response planner frontend, business impact analyser frontend). Test code needs to be updated and maintained as the program under test evolves.

Software engineering best practices indicate that bug discovery should happen as close as possible to the developer (as opposed to happening later, in software operations). If the issue happens in operations the developer may not have full access to the deployment environment (for security reasons, for instance). This slows down the bug fixing significantly for two reasons. First of all, if developers cannot reproduce a bug in an environment they fully control (e.g. their laptop, or their deployment infrastructure) there are little chances they can fix them quickly. Furthermore, it may be necessary for developers and operators to schedule a synchronous session to understand where the problem is, leading to more delay.

It is hard to estimate what kind of impact the lack of unit tests had on the success of the project. However, since a large amount of bugs were discovered in the pilot phase, the worst possible moment, impact was likely significant. The lesson learned here is that unit testing should be mandated as a requirement to component owners from the beginning of the project. Code coverage should be an official factor in establishing deployment readiness, and functional requirements should be formally tested through unit tests. Finally, all unit tests should be automated through Continuous Integration in a central repository so that identifiable bugs don't make it to the intermediate software release candidate.

2.2.2 Integration Testing

Whereas the goal of unit testing is to identify bugs within a software component, the goal of integration testing is to identify bugs between software components. The interaction between components can be genuine (as is the case in end-to-end testing) or mocked (common when components consume/produce data). Integration testing was performed manually at various stages as a task of work package 6, using a waterfall methodology. This approach had at least two issues: lack of agility and incompleteness due to the complexity of the software architecture. Lesson learned about the software architecture are described in section 2.1.2.

Software in SOCCRATES was designed and tested with waterfall methodology but execution was done with agile methodology. Just in the last three months, when one would expect the software to be more or less complete, there have been almost 2000 changes by 16 developers to fix bugs or add features. Manual testing "gates" just could not keep up. As a result, many integration bugs were discovered at the pilot sites, with the delays and overheads already mentioned in section 2.2.1. In the last months of the project we have successfully automated the testing of cortex analysers that were previously performed manually.

The Advanced Anomaly Detection component of work package 4, perhaps the most complex component of all, showcases a more successful integration testing story. There the entire AAD was configured and continuously deployed in a staging environment from the first year of the project. Whereas the components of SOCCRATES were designed as a mesh of services calling each other, detectors deployed in the AAD analysed data from a shared database and sent messages to a shared message queue independently. To perform integration testing it was then sufficient to mock the data in the shared database and to verify if expected messages showed up in the message queue. Unfortunately, this was not sufficient to discover scalability issues early, but only because AAD performance requirements had not been established. If they had been, this setup would have allowed us to synthetically produce more data and measure formally if the AAD was able to keep up.
The lesson learned here is that as a bare minimum there should be accordance between software engineering methodology and execution. Then, in practice, we see that developers tend to prefer agile development. Integration testing should be performed automatically and frequently, as part of software continuous integration. The experience performing integration testing in the AAD suggests that a looser architecture simplifies integration testing.

2.2.3 Deployment Testing

Complex software projects seldom are trivial to configure and deploy. The SOCCRATES platform is no exception. To guarantee that the deployment would work at the pilots, the project DevOps engineer configured Continuous Integration/Continuous Development (CI/CD) pipelines to automatically deploy each component in a staging environment after each fix/update. This worked well: deployment bugs were discovered quickly and because the responsible developers were notified automatically. As a further positive side effect, developers were also able to quickly test interactively (manually) if the software behaved well after they introduced their fix/feature.
The lesson learned here is that continuous deployment works well as a formal way to test the deployment code. It should be configured as early as possible in the development lifecycle, as it provides benefits of its own and further more.

2.2.4 Performance Testing

The fact that software may work correctly in a test environment can be misleading if such environment is not representative of the target operational environment. One aspect of this "representativity" is the scale of the data that the software will need to handle. Scalability requirements for the SOCCRATES components were not identified and, thus, they were not tested. As a result, scalability issues were identified only at the pilot sites.
The lesson learned here is that performance requirements must be identified and tested appropriately. If the software architecture makes it too complicated to stress test, perhaps it should be revisited. It's easier to stress-test an event based architecture than a mesh of interdependent microservices architecture: one can just synthetically create an enormous amount of events and see what happens.

2.3 Software Operations

2.3.1 Software distribution

All components of SOCCRATES, with the only exception of SecuriCAD, are containerised. Each component source code is packaged in a Docker container image. Container images are basically a snapshot of an operative system, application source code and the libraries and programs necessary to run an application. The main benefit of using this approach is that containerised applications are easy to distribute and can reliably run anywhere, similarly to how real life containers provide a standard way to move goods across the global rail, road and sea infrastructures. The one downside of this approach is that container images must be rebuilt (jargon for updated) every time the source code changes. This can easily be done manually, however in practice this is not suitable for a modern development lifecycle where tens or even hundreds of changes are performed on a software component in a single day. The solution is to perform automated builds through continuous integration. Every time the source code is altered this triggers a pipeline of jobs. Among these jobs there are tests. If the tests succeed, then the container image is rebuilt and stored in the software repository container registry, from which they can be easily downloaded at deployment time (provided there is an internet connection, more on this in later sections…). Adopting this approach was instrumental to performing continuous deployment for testing purposes and it provided a reliable way to ship software to the pilot sites. Furthermore, it allowed to increase the productivity of the team because software developers could focus on the actual changes they wanted to make rather than on how to package and distribute them.

The following pre-existing, partner owned, components are part of the SOCCRATES platform: ABC tool, DNS ninja, L-ADS, AMiner, ACT and SecuriCAD. All of these, except SecuriCAD, had to be adapted be deployed as containerized applications. The aforementioned benefits of containerisation (above all portability) came at a cost: whereas it has been simple to develop containerized applications from scratch, considerable work had to be spent on containerization of pre-existing software. This task is as hard as the application is complex. If containerization of a stateless single process application with no bootstrapping is very simple, it is much harder to do it for a stateful multi process application. Containerisation best practices demand that each container should run only one process, thus containerisation of multi process applications means having to setup multiple container images. State should be handled separately, for instance by another container running a database. In some cases it may be easier to work around legacy software rather than altering it directly, this can be done through another container in the so called sidecar pattern. For instance, DNS ninja did not output alerts as kafka events, however its source code could not be altered. A sidecar was developed to receive alerts in another format from DNS ninja, and then convert them to kafka events. If an application needs bootstrapping, this may have to be performed by yet another container (a so called init-container). DNS ninja ended up being made of four containers, the abc tool of two containers (one for performing training, the other to perform detection), L-ADS of four containers, Aminer of three, ACT of five. Attempts were made to containerise SecuriCAD, but were quickly abandoned due to the excessive amount of work required.

2.3.2 Container orchestration

Container orchestration is the automation of the tasks required to run and monitor containerized applications. The importance of container orchestration was one of the lessons learned in the first pilot at Mnemonic. In that pilot we only deployed the containers of the advanced anomaly detection components. We did so in an "imperative" fashion, and we run into the limits this approach. Deployments were performed manually and there was no proper way to monitor the deployments and roll back in case of issues. The decision was made to migrate to a declarative container deployment solution to address these limitations. The choice was between Kubernetes and docker swarm, the two leading open source container orchestration technologies. We were aware that Kubernetes was the most popular and probably best container orchestration solution available, however we opted for docker swarm because it required only minimal changes to our existing docker-compose files. Furthermore, nobody

in the project team had experience with Kubernetes, which has a pretty steep learning curve. Docker swarm behaved reasonably well but with hindsight it probably would have been much better to pick Kubernetes.
Kubernetes has a large ecosystem of observability solutions such as Prometheus (a tool that allows to monitor deployment performance and set alerts for downtime/errors), docker swarm does not. Open source software that we heavily relied upon (Elasticsearch, mysql, postgresql, minio, traefik, authelia, kafka) has great official support and documentation to deploy with Kubernetes but almost no support for docker swarm. A great deal of time was spent into configuring these tools to work with swarm. It would have been unnecessary if we had chosen to work with Kubernetes. For instance, considerable time was spent into configuring the advanced anomaly detection redundant Elasticsearch docker swarm deployment. But elastic itself maintains a Kubernetes helm chart that allows for a one line installation of complex multi node clusters. Helm charts that allow simple configuration and deployment of all the other open source tools exist. We also encountered a number of network related operating system specific bugs. Docker swarm has smaller community of users, which also means that less people experience, report or have an interest in fixing these issues. Kubernetes provides an easy way to perform deployments remotely, docker swarm does not: deployment needs to be performed from a machine which is part of the swarm. Gitlab, the software collaboration platform used for SOCCRATES, offers no special integration with swarm whereas Kubernetes integrates natively with gitlab, allowing a more productive software development experience. Aforementioned helm charts allow for an easy way to package and distribute Kubernetes applications. There is no such thing for docker swarm, we had to come up with our own solution for this. Finally, cloud providers do not offer managed docker swarm as a product. Whereas one can easily, quickly and cheaply get a Kubernetes cluster on Azure or Google Cloud Platform, there is no such thing for docker swarm. A very large amount of work went into setting up and maintaining an internal docker swarm cluster at TNO.

2.3.3 Software installation

The software used in the project pilots was, for the most part distributed as docker containers and associated docker-compose.yml files for deployment in a standardized way. Additionally, site specific configurations were stored in its own repository. The one exception for this was the installation of SecuriCAD, which was installed on a standalone machine due to incompatibility issues with Docker. This method of installation eventually reduced the issue of installation to a few command lines, primarily divided between assigning the labels to the Docker hosts to limit the components from moving away from the machines where their disk storage were, and a command for each component to be started. From the Pilot sites perspective, this was a very good experience compared to other complex systems.

2.3.4 Software updates

Overall, while the experience of installing the components as explained above was a positive one, updates had some issues. These were mostly related to frequent updates in the projects. Due to the architecture of both pilot sites (see section 2.2) being based on isolation of the network, the transfer of update files was more of a pain than it should have been. It would, however, probably not have improved by any other method of distribution. Had we known the amount of updates we were going to do, we would probably have prioritized making scripts for download of Docker images and uploads/installations to the pilot environment. A suggestion which came later, but which would have proved useful due to the process of uploading the full code repository to include all necessary configurations, was to separate the repository used for building the Docker containers from the repository used by the pilot sites to install and update. Since the code base was quite big, and often included sample data, it proved to be a bottleneck during updates.

2.3.5 Familiarity with software stack

A consideration which should probably have been taken was the amount of familiarity with the chosen software stacks. While we had competent users/administrators for the shared components, there were some issues we had which would have proven a lot easier to deal with should we have had more expert-level administrators. Example one of this was the administration of Elasticsearch. which was a central component to the event analysis portion of the pilot. While it was not a showstopper, we experienced issues doing some of the advanced configuration and management early due to a lack of Elasticsearch expertise. Similarly, we also had some issues with network communication between Docker containers, which was due to a interaction between the Docker network stack, CentOS and VMware ESXi. Luckily, in this case we had the expertise to troubleshoot the network communications, but it could easily have been a bug which would require a lot of work to diagnose. All in all, this highlights the importance of having experts available for development and diagnosis of whatever software stack you decide to use. This was exacerbated by Covid19 issues not allowing us to have the expected amount of in-person debugging at the pilot sites.

2.3.6 Debugging and monitoring opportunities

A health monitoring system like Nagios could have provided alerting when not all components were functioning. At the second pilot site, we had some baseline health monitoring using this set up to monitor for common system level issues, like disk, CPU and RAM usage, as well as monitoring of Out of Memory events in the system logs. This proved invaluable when debugging, especially since Docker is designed to be resilient to issues and will automatically restart containers with issues. Grafana or a similar observability platform could have provided statistics on throughput to identify issues and verify component interaction. This does, however, require that all components in the platform expose this data somewhere so the data can be retrieved. For projects with complex software, with many components that need to work together, we recommend prioritizing such statistics.

2.4 Scalability

A central theme in terms of scalability is the concept of the three Vs of big data. These are Volume, Velocity and Variety, and they all affect how systems deal with the amounts of data we expect to see in advanced systems such as SOCCRATES, especially for the analysis engines seen in the AAD system.

2.4.1 Initial Scaling

2.4.1.1 Generic scaling of projects

Initially, we reported the projected amounts of data in events per second (Velocity) to the other project participants, as that is our most common bottleneck by experience. Internally, we scaled machine hardware for the common storage of events (Elasticsearch) after volume, since we have an abundance of CPU and RAM (compared to system requirements provided for all components) on the machines dedicated to use in the pilots. This was quickly proven to be insufficient, as several of the AAD components had approaches to data analysis which made the volume the bigger issue. This was especially the case for the components which use machine learning iteratively on the data, but do not use/offload storage requirements to Elasticsearch.

The data seen in our network-related traffic is luckily uniform enough that the issue of Variety doesn't come up, though other projects which do take-in other types of data with more variance should be aware that many of the systems made for big data use compression extensively to keep the storage requirements down. The presence of highly variable data should be taken into account when scaling the infrastructure.

2.4.1.2 Assessing data volume

Some of the issues faced later in the project specifically stems from the lack of a comprehensive sample set of data which could be used by the project participants to develop their projects and train their models. Providing large scale samples from either of the pilot sites was impossible since both sites had strong privacy/security requirements. Initially, we thought providing statistics on how much data throughput we see (both Velocity and Volume) over time would be sufficient for accurate estimations of requirements, especially for runtime requirements. The hope that the statistics as opposed to samples would prove sufficient would later show to be wrong, when the AAD components could not deal with the amount of data in the mnemonic pilot site.

Similar projects should strongly consider either choosing at least one pilot site which is capable and willing to provide large scale samples to project partners, or the project should dedicate some work-hours to synthesize required data.

Such sample data would have several appreciable benefits, including helping the component owners realistically provide system requirements, allow for automatic integration testing, allowing automatically testing that throughput does not fall under the level where data can be dealt with in real time, and allow the developers to synthetically inject suspicious traffic in a normal traffic flow and thus allow for testing of detection capability.

2.4.1.3 Providing system requirements

The experience of the pilot site owners was that it was hard to get accurate minimum/recommended requirements for the components which should run in our environments. No doubt aided by the issues described above, especially our inability to provide large scale samples beforehand to the component developers, we experienced severe overruns on most components, ranging from slight overuse of CPU to an estimated 1000 time requirement of RAM for a component. While many of these issues were resolved by means of improvements to the code, the issues faced with the AAD components show just how catastrophic such underestimation can be to the successful execution of Pilot activity.

If possible, requirements should be stated as a function of relevant data, i.e. 1 GB ram per Y MB/s traffic or 1 CPU core per 1000 nodes in the infrastructure model. However, also be aware that algorithms have a tendency of being quadratic or worse, so make sure the requirements scale with the algorithms used in the component.

2.4.2 Scaling Underway

2.4.2.1 Deduplication of effort

The AAD components were developed by the project partners previous to the SOCCRATES project beginning. As such they had very different intended input methods of data input and formats of data. A common method of subscribing to a data flow from Elasticsearch was made using a Docker container with Logstash, which could be used by the project participants to natively receive log files. In an ideal world, all input of log data should have been done as a continuous stream of data, but to be able to reuse code with minimal changes it served as a good way to retrieve log files from the common storage setup. It also allowed the developers to use the Elasticsearch central log repository even if they weren't well acquainted with it. Initially, there was also a tool developed to ingest netflow data and output subnet information. This tool was among those having issues with performance. However, since Elasticsearch had been chosen as the log repository for the project, we could instead use the native support for aggregation of IP networks instead of the project tool. Sometimes, the choice of supporting technologies can let you avoid development time.

2.4.2.2 Docker limit rebalancing

The use of Docker to limit individual components in a complex system like SOCCRATES was invaluable. Especially early in the full scale pilot, we experienced several components not coping with the data amounts in the pilot sites well, usually growing their RAM usage until the Out Of Memory killer in the host operating system killed critical processes. The ability of Docker containers to set container/process limits allowed us to keep the overall SOCCRATES system running, even if individual components were misbehaving and attempting to use more than their share of resources. It also allowed us to test if certain components could be made to work with more/less resources, without doing major changes.

2.4.2.3 VM Resource Rebalancing

Similar to the Docker limits above, we also had to rebalance RAM and CPU between different components. Since all machines were installed as virtual machines, where two of the Docker hosts and the SecuriCAD system shared one physical host, we were allowed the flexibility to move resources around as needed, without having to add new resources, or having to move files. The latter would have been a requirement if we moved a Docker container using a persistent volume to another Docker machine.

2.5 Pilots

The first SOCCRATES pilot was according to plan. The second pilot however, in which the complete platform would be deployed on two pilot locations, did not completely go according the plan. This is not unexpected in such a complex and large project. We have evaluated the reasons for changes and delays, and have learned from the analysis.

The importance of physical meetings when coordinating across many partners from different institutions and countries is not to be overseen. SOCCRATES was hit hard by the COVID-19 restrictions in travel. All planned face-to-face meetings in order to get the platform up and running on the pilot sites were cancelled. Both pilot sites where the full platform was to be validated had strict restrictions on on-line access, which meant there were no on-line possibilities for external partners to help in implementation and testing. Everything had to go through e-mail, chat and/or Gitlab. The plan was to have all component owners on-site for the implementation, and with this possibility cancelled due to COVID-19, we had significant problems with sorting the issues and bugs emerging as the platform was being deployed component by component.

In June 2022 we were able to have *one* face-to-face debugging session on the mnemonic pilot site with an engineer from *one* component owner. In the three days spent together, we estimated that the amount of work successfully completed corresponded to approximately 4 weeks of trying to complete the same over e-mail, chat and/or Gitlab. Considering the amounts of components in need of running in the complete platform, this corresponds well with the delays we faced in the pilot execution.

There was an unexpected amount of issues and bugs emerging when deploying the platform on the pilot sites in mnemonic and Vattenfall. This, combined with all of the learning points described in the sections above, were significant contributions to the delays in the pilot evaluation.

Abbreviations

This glossary serves as inventory of abbreviations used in the document.
This is a standard glossary, used for all SOCCRATES report deliverables; it will be expanded when necessary.

Acronym	**Description**
ACT	semi-Automated Cyber Threat intelligence
ADG	Attack Defence Graph
AEF	Argus Event Format
AI	Artificial Intelligence
AIT	AIT Austrian Institute of technology
API	Application Programming Interface
APT	Advance Persistent Threat
ATOS	ATOS Spain
AV	AntiVirus
BPMN	Business Process Model and Notation
CC	Command and Control
CERT	Computer Emergency Response Team
CMDB	Configuration Management Database
CSIRT	Computer Security Incident Response Team
CoA	Course of Action
CTI	Cyber Threat Intelligence
DC	DataCentre
DGA	Domain Generated Algorithm
DNS	Domain Name System
EDR	Endpoint Detection and Response
ELK	Elasticsearch/Logstash/Kibana
FRS	Foreseeti
FSC	F-secure
ICT	Information and Communication Technology
IDS	Intrusion Detection System
IMC	Infrastructure Modelling Component
IMT	Institut Mines Télécom – Télécom SudParis
INTF	Interface
IoC	Indicators of Compromise
IP	Internet Protocol
IPS	Intrusion Prevention System
IRM	Incident Response and Management
ITIL	Information Technology Infrastructure Library
KTH	Kungliga Tekniska högskolan - Royal Institute of Technology
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
M_n	Infrastructure Model (at time *n*)
MNM	mnemonic
MSSP	Managed Security Service Provider
MTTD	Mean Time To Detection

NOC	Network Operations Centre
OT	Operational Technology
OS	Operating System
RORI	Return on Response Investment
SDN	Software Defined Network
SHS	Shadowserver
SIEM	Security information and event management
SOAR	Security Orchestration, Automation and Response
SOC	Security Operation Centre
SOCCRATES	SOC & CSIRT Response to Attacks & Threats based on attack defence graph Evaluation Systems
SSL	Secure Sockets Layer
TAP	Test Access Point
TIP	Threat Intelligence platform
TLS	Transport Layer Security
TNO	Nederlandse Organisatie voor toegepast natuurwetenschappelijk onderzoek
TTC	Time To Compromise
UC	Use Case
VLAN	Virtual LAN
VM	Virtual Machine
VTF	Vattenfall