Masters in Informatics Engineering

Building a Scalable Near Real-Time IoT System

PEDRO MIGUEL DOMINGOS DA COSTA

Supervisor: Mário Rela, University of Coimbra
Supervisor: Rafael Jegundo, Whitesmith

Examiner: Amilcar Cardoso, University of Coimbra
Examiner: Marco P. Vieira, University of Coimbra

Master's Thesis 2015/2016
Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

Building a Scalable Near Real-Time IoT System

PEDRO MIGUEL DOMINGOS DA COSTA
pmdcosta@student.dei.uc.pt
Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

Abstract

All Food Industry and retail companies in Europe are required to comply with strict regulatory demands[1] that are essential to the safety of consumers. The regulations require constant monitoring of all refrigeration systems, which often is resource intensive and unreliable if done manually. These companies also rely on fridges and refrigerated cabinets to store key business assets, and malfunctions could incur severe financial losses. While big enterprises already have solutions for cold chain monitoring, the small and medium sized businesses do not, due to complex installation processes, bad user experience, lack of information or elevated costs.

Qold is a cold chain monitoring system being developed by Whitesmith to improve the way SME's operate. It combines custom sensors with clean and simple web and mobile applications. **Qold** uses a network of devices to measure temperature values and report them to a cloud server. This data can be immediately consulted by clients and is analyzed to detect undesirable situations which are promptly reported, preventing possible damages.

At the start of the internship, a minimum viable product (MVP) had been developed and deployed to early adopters. The product had been validated and a new system needed be developed, taking into account scalability concerns. The objective of this internship is the architectural design and implementation of a new system that builds on the MVP and meets the requirements of a system ready to tackle the European market.

Keywords: Cloud Computing, Internet of Things, Stream Processing, Distributed Systems.

Contents

List of Figures

List of Tables

Acronyms

ACLs Access Control Lists. 42
AMQP Advanced Message Queuing Protocol. 17
API Application Programming Interface. 46
ASR Architecturally Significant Requirement. vi, 29

CAP Consistency Availability Partition tolerance. 44
CD Continuous Delivery. 6
CEP Complex Event Processing. 2
CRC Cyclic Redundancy Check. 55

FQDN Fully Qualified Domain Name. 33

HDFS Hadoop Distributed File System. 43
HTTP Hypertext Transfer Protocol. 17

IoT Internet of Things. 1

JSON JavaScript Object Notation. 46
JWT Json Web Token. 46

LSD Lean Software Development. 6

MoSCoW Must have, Should have, Could have, Won't have. 12
MQTT Message Queuing Telemetry Transport. 17
MVP Minimum Viable Project. 1

ORM Object-Relational Mapping. 55
OTA Over The Air. 64

QoS Quality of Service. 3

Rest Representational State Transfer. 46

SMB Small and Medium Businesses. 1

TLS Transport Layer Security. 19
TSD Time Series Daemon. 43

VM Virtual Machines. 21

Glossary

authentication The process of identifying an individual or system.. 15
authorization The process of giving individuals access to system objects based on their identity.. 15

client side Physical space where devices are deployed. 3
cloud Internet-based computing, shared resources, data and information are provided to computers and other devices on-demand. 3
cloud gateway Component on the server side of the system that receives readings from the local gateways.. 17
component A unit of software that is independently replaceable and upgradeable. 3

dependability The fault tolerance of a system.. xi, 16
devOps The DevOps culture emphasizes the collaboration between the roles of development and operations, aiming for a shared responsibility of deployment, maintenance and monitoring of software.. 4

gateway Component that receive readings from several devices and publish them to the cloud through the Internet. Also called local gateways.. 3

ingestion The act of receiving messages and making them available for consumption.. 14

legacy The system that existed at the start of the internship. 1

maintainability Ability of the system to undergo changes with a degree of ease.. xi, 15
manageability An indication of how easy it is for system administrators to manage the application.. xi, 15
measurement A single temperature reading from a single device. 3
MVP A product with just enough features to gather validated learning about the product. 1

near real-time Effectively Real-Time but without guarantees of hitting specific deadlines. Also known as soft real time. 2

orchestration The automated arrangement, coordination, and management of complex computer systems, middleware and services.. 20

performance An indication of the responsiveness of a system to execute any action within a given time interval.. 14
provisioning The act of configuration, deployment and management of a single of multiple components or resources.. III

reliability The ability of a system to bounce back from failures and continue operating in the expected way over time.. xi, 15

scalability Ability of a system to either handle increases in load without impact on the performance or the ability to be readily enlarged.. xi, 14

scale out Scale horizontally, by means of adding additional nodes.. 14

stream Continuous unbounded flow of data. 1

test driven development A technique for building software that guides software development by using automated tests. By forcing the developer to write self testing code it offers a very fast feedback on the software. TDD involves cycling through three steps, writing the tests for a functionality, implementing it until it passes the tests, and then refactoring the code.. 6

validation The evaluation of whether a component/system meets the needs of the stakeholders. 4

verification The evaluation of whether a component/system behaves as expected. 4

1
Introduction

The present document provides support to the project developed during the course 'Dissertation/Internship in Software Engineering'[2], part of the Masters Degree in Informatics Engineering at the Faculty of Sciences and Technology of the University of Coimbra[3].

This report supports for the work developed during the two semesters of the internship, and when applicable will discriminate what was accomplished in each semester. Overall, the first semester was important to gather and document the project requirements, understand the legacy system, design a possible architecture and study the state of the art. During the second semester, the system was built, tested and validated.

The internship was hosted at Whitesmith[4], a multinational technology and software company headquartered in Coimbra, Portugal. Although external consulting software projects represent the core business of the company, Whitesmith is also focused on developing its own products. Qold[5] is one of them, and the focus of the internship.

Qold is a cold chain monitoring system being developed by Whitesmith to improve the way SMB's operate. It combines custom sensors with a clean and simple web application. Qold uses a network of devices to measure temperature values and report them to a cloud server. This data can be immediately consulted by clients and is analyzed to detect undesirable situations which are promptly reported, preventing possible damages. Qold distinguishes itself by being trivial to install and require minimal to no upfront investment.

At the start of the internship, a Minimum Viable Project (MVP) had been developed and deployed to early adopters. The product had been validated and a new system needed be developed, taking into account scalability concerns. The objective of this internship is therefore the architectural design and implementation of a new system that builds on the MVP and meets the requirements of a system ready to tackle the European market.

1.1 Context

1.1.1 Internet of Things

The expression Internet of Things (IoT), was coined in 1999 by Kevin Ashton[7], a British technology pioneer who co-founded the Auto-ID Center at the Massachusetts Institute of Technology. It describes a network structure that connects physical resources and people through software. It enables an ecosystem of smart applications and services that improve and simplify the everyday life and contribute to sustainable growth.

In 2013 Kevin Ashton claimed that IoT is already here, and both Gartner[8] and Cisco[9] claim that IoT is an emerging 'mega-market', and one of the top ten strategic technology trends. In fact, Cisco predicts that by 2020 there will be 50 billion devices connected and a potential market in excess of 14 trillion dollars.

In part due to the large amounts of data generated by these devices, other related topics are also on the rise, such as big data, energy efficiency, commodity hardware, stream

processing, machine learning and analytics. Because the Internet of Things is developing at such a fast pace, new ideas, concepts and technologies are appearing and evolving daily, which was both a challenging and interesting characteristic of the internship.

1.1.2 Big Data

Big data is another growing trend, it is a term that describes the large amounts of data gathered and analyzed for insights by business on a daily basis. But Big data is not just about volume, but also what organizations do with the data. Big data is generally defined using three Vs: Volume, Velocity and Variety.

Volume describes to the terabytes and petabytes of data collected from a variety of sources, including business transactions, social media and sensors. Velocity refers to the streams of data gathered at high speeds that must be dealt with in a timely manner; sensor data for example requires near real-time[6] processing. Finally, data comes in all types of formats, from structured numeric data to unstructured text documents gathered from social networks, blog posts and sensors.

In a Big data architecture, masses of structured and semi-structured historical data are stored in a map-reduce application, such as Hadoop (Volume + Variety), and stream processing is used for fast data requirements (Velocity + Variety). While stream processing was a major aspect of the internship, there was no need for a tool such as Hadoop.

1.1.3 Stream and Complex Event Processing

Complex Event Processing (CEP) and Stream Processing are two very similar paradigms, they are both used to extract information and operate over a boundless stream of data. However, Stream processing engines and CEP engines are different and come from different backgrounds. Streaming processing engines are designed to process data streams with high event throughput and a smaller numbers of queries, whereas CEP engines usually have a large numbers of rules and are optimized to process discreet business events.

CEP engines focus on complex rules/patterns, usually using an higher level language such as a SQL like query language, and tend to be more centralized. They come from Stock market related use cases, and provide primitives such as time windows, temporal event sequences and aggregations.

Stream processing engines allow users to create highly parallel workflows to operate over streams. Most Stream processing workflows are created using programming languages, as opposed to query languages used in CEP engines.

Stream processing is used instead of the traditional database model where data is first stored and indexed and then subsequently processed by queries. Instead, it takes the inbound data while it is in flight, processes it, and connects to external data sources; enabling applications to incorporate data into the application flow, or to update an external database with processed information.

In practice, while CEP is used to find complex patterns in a continuous data stream and respond to those patterns, stream processing is used to parallelize processing as much as possible in order to handle a high event throughput, and perform simple operations.

1.2 Internship

1.2.1 The Problem

As stated before, at the start of the internship, a legacy system had been developed, and while it is further detailed latter on, a brief introduction is required to provide context to the scope of the internship.

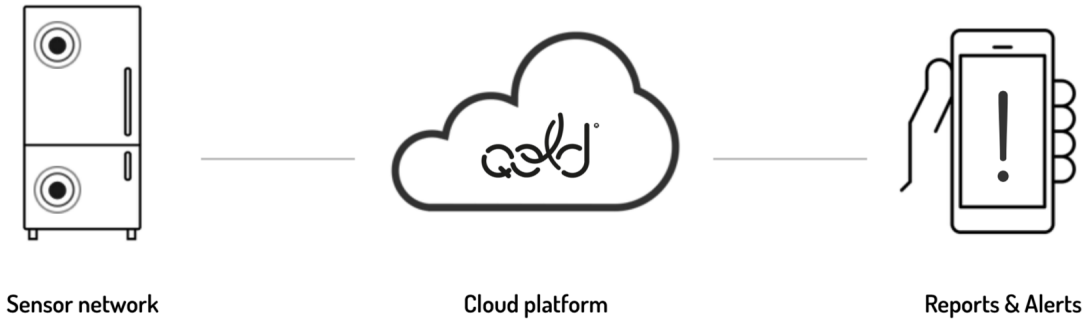

Figure 1.1: Qold System

The system has two client side components, the devices and the gateways. A device is a small, low power sensor which takes temperature readings from the environment. In order to keep power consumption to a minimum, this component does not possess Internet connectivity or any functionality beyond reading and broadcasting measurements through radio signal. The gateways are bigger, more powerful devices that receive readings from several devices and publish them to the Cloud through the Internet.

Although a simplification, the cloud backend can be separated into four components, the Hardware API, Aqora, Qold-app and finally the CEP engine.

The Hardware API is the entry-point of the system; messages received from the gateways are authenticated against a local database and sent to Aqora and CEP. If the temperatures received are too high or too low, the Hardware API will also notify the clients using a SMS service. Aqora is a timeseries database developed at Whitesmith that uses MongoDB[10] as the storage backend. Qold-app is the web application used by clients to configure alarms and consult the readings. And finally the CEP engine is a Fi-Ware Proton[11] platform, being used for QoS, such as alerting the Qold staff when problems with devices are detected.

1.2.2 Scope and Goals

The main focus of the internship was the design and implementation a new architecture for the Qold product. The new architecture should be an evolution of the legacy system, and changes to the Qold product should be done in an incremental way, in order not to disrupt deployments, and constantly validate changes. It is also important to note that, initially the legacy system was not static and evolved in parallel with the new one, in order to achieve short term objectives, prototype features and solve immediate needs. The new system was to be production ready by the end of the internship, and during development both systems were deployed to early adopters.

The architecture developed was focused on the Qold needs, but is also general enough

to serve the needs of other similar IoT projects, and future iterations of the product. Besides the backend system, the intern was also responsible for updating the software and ecosystem of the gateways. This required a rewriting of the firmware and revising the update, deployment and communication mechanisms.

Since the system will have to be maintained and kept updated by Whitesmith staff, it was important to use well known technologies, take the skillset of the Whitesmith staff into account when making decisions, and make sure it was easy to replace system components without having to change others. Detailed documentation was also created for all components.

Finally, the intern was also expected to follow the DevOps[12] movement used at Whitesmith, meaning that he was responsible for all the operational concerns of the components developed.

Although architecture design and implementation of the system were performed by the intern, some components were excluded from the scope of the internship. These exceptions are components that already met quality attributes, dealt with Qold specific business logic or fell outside the interns skillset, such as the Qold-app, and Devices. Following the feedback received from the intermediate evaluation, the system dashboard was removed from the scope of internship, and handled by other Whitesmith staff, and the reverse messaging of gateways was removed from the system entirely, meaning that gateways can only send data, and not receive remote commands.

1.3 Document Outline

- This first chapter introduces the internship and its context, focusing on goals and scope.
- Chapter 2 introduces development methodologies, and focuses on the development process used during the internship.
- Chapter 3 specifies the system requirements and its gathering process.
- Chapter 4 describes the State of the Art research conducted the internship.
- Chapter 5 focuses on architecture and architectural decisions.
- Chapter 6 explains all the operational decisions and tools.
- Chapter 7 details the implementation of system components.
- Chapter 8 describes the verification and validation of the system.
- Finally, Chapter 9 concludes the document, offering a description of future work and a look back at what was achieved.

2
Project Development

This chapter describes the software development process used during the internship. The first section introduces software development concepts and practices which served as base to the process used. The second section introduces the life-cycle of the project. The third section presents the planning process and explains all its stages and concepts. Finally, the identified risks and their mitigation plans are introduced.

2.1 Methodology

During the project, the intern adopted the Agile development methodologies used at Whitesmith, not only to promote integration with the Qold team, but also because it fit the needs of the project. While the new system was developed by the intern, he was integrated on the Qold team. This team was mainly composed of four other members: Gonçalo Louzada as project manager, Luís Antunes as responsible for hardware, Diogo de Bastos was a colleague intern working on further developing the Qold product, but focused on hardware and business, and finally João Nogueira was responsible for marketing and business.

An important aspect of the project, which also motivated the Agile development process chosen is the fast evolution of the IoT, Big data and stream processing contexts; during both the first and second semesters new technologies, updates and patterns were being released almost daily, not always in the form of new features but also new integrations and operational concerns. Several technologies used were also new both to the company and the intern. For these reasons, it was hard to properly plan ahead and make accurate estimations.

Since the project is an evolution of an already in place system, and following the goals of the internship, the new system should be released incrementally. Because of this, the approach taken by the intern, was to properly define the high level architecture from the start, and divide the project into multiple components that could be worked on and deployed individually. Each component followed multiple cycles of design, implementation and testing. Although the high level architecture and the component design didn't change since the final architecture was designed, at the start of the second semester, the implementation and technologies used evolved throughout the project. This division of work was one of the main factors for the Microservices based architecture of the system, explained in Chapter 5 of this report.

2.1.1 Agile Methodology

Agile[13] has emerged out of various people who dealt with the heavy and bureaucratic software development processes in the 1990s and looked for a new approach to software process. This section of the report provides a very brief introduction to Agile software development and attempts to justify its use.

After a few years of ad-hoc development, the notion of methodology was introduced, which imposed a disciplined process upon software development. While trying to make software development more predictable and efficient by designing a process with strong emphasis on planning, it slowed down the pace of development due to heavy bureaucracy. The Agile movement emerged as a reaction to these methodologies, attempting a compromise between no process and too much process, as explained by Martin Fowler:

'Agile methods are adaptive rather than predictive. Engineering methods tend to try to plan out a large part of the software process in great detail for a long span of time, this works well until things change. So their nature is to resist change. The agile methods, however, welcome change. They try to be processes that adapt and thrive on change, even to the point of changing themselves.'

Agile methodologies were not only are a clear fit to the unpredictable nature of the project, but was also the process being used at Whitesmith. Within Agile development, the process used by the Qold team and the intern was based on practices from Lean Software Development[14] and Kanban[17].

2.1.2 Lean Software development

Lean Software Development (LSD) is an approach based on the management process used at Toyota following World War 2, and consists on the pursuit of perfection by eliminating waste. In practice, it promotes the reduction of non-value-added activities and the smoothing of development flow. Lean Software Development does not prescribe any practices, but some activities have become common. David J. Anderson describes some of these activities:

'As motivated by Lean Software Development, work is undertaken in small batches, referred to as iterations or increments. (...) Small batches require frequent interaction with business owners to replenish the backlog or queue or work. They also require a capability to release frequently. To enable frequent interaction with business people and frequent delivery, it is necessary to shrink the transaction and coordination costs of both activities. A common way to achieve this is the use of automation.'

As stated, automation is an important aspect of Lean Software Development, and was put in use during the development of the project. It helped smoothed development, and because most aspects of the project was automated, it promoted fast feedback loops, and reduced the overhead of testing, or performing changes to the system. Test driven development was also used whenever applicable. Although initially considered, the use of a Continuous Delivery[15] tool such as Jenkins[16], was dismissed since the intern was the only developer on the project, and architecture design consisted of a large part of the internship. The overhead of managing the CD tool was not worth its benefits.

'Teams of software developers typically meet in front of a visual control system such as a whiteboard displaying a visualization of their work-in-progress. They discuss the dynamics of flow and factors affecting the flow of work. Particular focus is made to externally blocked work and work delayed due to bugs. Problems with the process often become evident over a series of standup meetings. The result is that a smaller group may remain after the meeting to discuss the problem and propose a solution or process change.'

'Project teams may schedule regular meetings to reflect on recent performance. These are often done after specific project deliverables are complete or after time-boxed increments of development known as iterations or sprints in Agile software development. Retrospectives typically use an anecdotal approach to reflection by asking questions like What went well?, what would we do differently?, and what should we stop doing?'

The Qold team had both daily and weekly meetings, the latter lasting much longer. The daily meetings were used to discuss more urgent situations, as well as remove any blockers. The weekly meetings were important for the planning of the development. Every meeting started with an analysis of what was accomplished the previous week. These look-backs are important to synchronize the team receive outside feedback. The work for the following is also planned and estimated. While monthly objectives were also defined, the fast changing context of the project meant that weekly estimations and planning were far more accurate and helpful. This cycle of weekly planning followed by feedback helped smooth development.

2.1.3 Kanban

Along with Lean Software development, kanban practices were also used during the development of the project. While LSD was important to macro-manage the work on a weekly basis, kanban helped micro-manage development. Kanban is another framework adopted from Toyota. It uses a system of cards, usually represented on a whiteboard, to limit the quantity of work-in-progress at any given stage in the workflow. The whiteboard has several columns representing the steps in the workflow, like backlog, in-progress and completed. A kanban team is only focused on the work that is actively in progress, and once the team completes a work item, they pull the next card from the top of the backlog. The team is free to re-prioritize work in the backlog without disrupting the workflow, because any changes outside the current work items do not impact development. As long as the most important cards are kept on top of the backlog, the development team is delivering maximum value back to the business.

Every month during the first weekly meeting, monthly goals are defined and translated to kanban cards written in user stories, which are augmented with an estimated time required to complete the task and priority. The estimations were almost always decided by the intern, based on experience, as he was the only developer in the team. The priority of the cards was also discussed on the weekly meetings based on the immediate needs of the system, but as with the estimations, they were mostly handled by the intern as he was more familiar with the project. These cards were added to the backlog, and were updated every week or whenever needed.

During development, the kanban whiteboard used was Trello[18]. It allows each card to be added a priority, due date, comments, estimated effort and other characteristics, essentially it allows a task to be completely mapped to a card, along with its context.

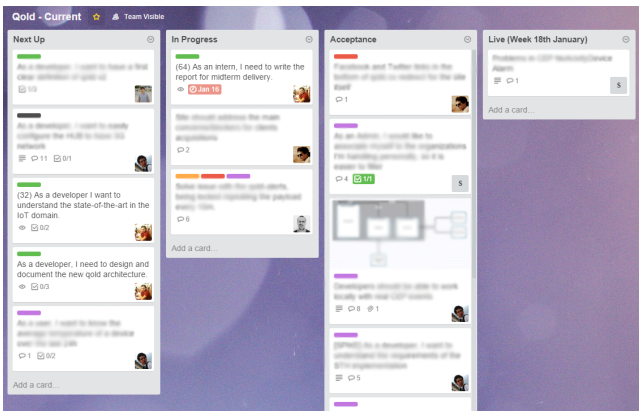

Figure 2.1: Current Trello Board

2.2 Planning

The section briefly introduces the planning followed during the internship, more detailed charts can be found in Appendix A. A total of 16 weekly hours of effort expected for the first semester, and 40 for the second. Since Whitesmith promotes a flexible schedule and a remote-friendly environment, the distribution of time did not follow the usual 8 hours a day for 5 days during the second semester; instead was closer to 5 to 6 hours a day for every day of the week, hence the weekends are present in the tables and charts.

2.2.1 First Semester

The focus of the first semester was the study of the Qold product and preparation of the seconds semester. The preparation mainly consisted of development of new components for the legacy system in order to better understand the project, the study of its architecture and the gathering and analysis of requirements. At the same time, the study of the state of the art in IoT and orchestration was essential to design the architecture of the new system.

Table 2.1 shows a simplified view of the tasks planned for the first semester:

Task	Start	End	Duration (days)
Introduction to Whitesmith	13/09/2015	19/09/2015	7
Internship scope and objectives	20/09/2015	03/10/2015	14
Gather and analysis of requirements	04/10/2015	24/10/2015	21
Study State of the Art	25/10/2015	07/11/2015	14
Study legacy system	08/11/2015	28/11/2015	21
Design new architecture	29/11/2015	19/12/2016	21
Study Tools and technologies	20/12/2016	09/01/2016	21
Write report	10/01/2016	23/01/2016	14

Table 2.1: Tasks planned for the first semester

2.2.2 Second Semester

The kanban methodology followed, promotes the use of continuous release and a priority system as opposed to heavy planning, but due to the size of the internship there was a need to establish deadlines for the second semester. A Gantt chart was initially developed, and suffered several iterations during the development of the project in order to cope with changes to the components and scope. The initial and final versions of the Gantt chart can be found in the Appendix A of this report.

Table 2.2 shows a simplified view of the tasks planned for the second semester.

Task	Start	End	Duration (days)
Planning	08/02/2016	11/02/2016	4
Gateway communication	12/02/2016	20/02/2016	9
Legacy gateway firmware	21/02/2016	27/02/2016	7
Gateway design	28/02/2016	13/03/2016	15
New gateway firmware	14/03/2016	31/03/2016	18
Security	01/04/2016	12/04/2016	12
Raw database	13/04/2016	18/04/2016	6
Device database	19/04/2016	23/04/2016	5
Time-Series Database	24/04/2016	13/05/2016	20
Rest API	14/05/2016	28/05/2016	15
Complex Event Processing	29/05/2016	07/06/2016	10
System validation	08/06/2016	11/06/2016	4
Write report	12/06/2016	30/06/2016	19

Table 2.2: Tasks planned for the second semester

2.3 Risk Analysis

The proposed project was ambitious, a fact that was further mentioned during the midterm evaluation; not only due to its size, but also because it dealt with real world clients, new technologies and because it was in production. Because of this, there was a need to identify possible threats to the success of the project, as well as devise ways of dealing with them. These risks were discussed every weekly meeting by the Qold team. A portion of every meeting was used to evaluate the evolution, priority and importance of every risk. The following risks were identified and analyzed during the internship, and are further documented in Appendix B.

State of the Art technologies and tools
Lack of documentation for technologies and tools
Deployment setup is not trivial
Integration issues between tools
Inexperience with technologies
Integration issues with legacy
Lack of legacy documentation
Parallel Development Divergence
Parallel Development Convergence

Table 2.3: Risks identified

3
Requirements

This chapter describes the gathering and analysis of both the functional requirements and the quality attributes.

An agile product backlog was used to describe the requirements of the system; it is a prioritized features list, containing short descriptions of all functionality desired in the product.

3.1 Requirements Gathering

The requirements gathering started with the study of the objectives and architecture of the legacy system. During the first semester, the intern joined the legacy development team in order to better understand the system; Complex Event Processing rules were implemented and a new automated gateway provisioning process was created. These weeks working with the development team helped realize the main obstacles and limitations of the legacy system. After discussing with the team, the objectives and needs of the system were crystallized in the form of functional requirements and quality attributes.

3.2 Stakeholders

The following actors interact with the system in different ways and will be referenced throughout the report.

User	
Description	Client of the Qold product.
Interaction	Uses the web and mobile applications to consult readings.
	Uses the web and mobile applications to configure alerts.
	Receives temperature and lack of communication alerts

Table 3.1: User stakeholder

Administrator	
Description	Member of the Qold team.
Interaction	Creates and configures gateways and devices.
	Uses the web and mobile applications to consult readings.
	Receives temperature, battery, system and lack of communication alerts.
	Uses the dashboards and logs to monitor the system.

Table 3.2: Administrator stakeholder

Developer	
Description	Developer and member of the Qold team.
Interaction	Creates and configures gateways and devices.
	Uses the web and mobile applications to consult readings.
	Receives temperature, battery, system and lack of communication alerts.
	Uses the dashboards and logs to monitor the system.
	Develops, deploys and configures system components.

Table 3.3: Developer stakeholder

3.3 Functional Requirements

Functional requirements were specified in a product backlog through user stories. A user story is a short and simple description of an interaction between an actor and the system. User stories are aimed at describing what the system or component should do instead of how they do it; they are fast to create, easy to comprehend and can be mapped into acceptance tests for validation. A user story follows the format:

As an <actor>	(Who?)
I want <action>	(What?)
In order to <benefit>	(Why?)

Table 3.4: User story

The MoSCoW[19] (Must have, Should have, Could have, Won't have) approach was used to sort features into a priority order. 'Must' and 'Should Haves' are features essential to the success of the internship. 'Could Haves' are nice to have features that are included if they don't incur too much effort or cost. 'Won't Haves' are features that were excluded from scope, but may be included in a future phase of development.

Since the goal of the internship was the replacement of an already existent system which in practice functioned as a crystallization of the required features, the main focus of the requirements gathering and analysis were the quality attributes.

3.3.1 Product Backlog

The following product backlog was created with the system's main features and user interactions.

Must have
As a User I want to consult the temperature of my infrastructure In order to monitor their activity and condition.
As a User I want to configure temperature thresholds In order to get notified when my infrastructure is too hot or too cold.
As a User I want to configure my contact information In order to keep notification active.
As a User I want to get notified when an anomalous situation is detected In order to prevent damage to my goods.
As an Administrator or Developer I want to build and configure new gateways and devices In order to deploy them to clients.
As an Administrator or Developer I want to automatically register new gateways and devices In order to enable communication with the server.
As an Administrator or Developer I want to get notified when an anomalous situation with the system occurs In order solve and prevent errors or downtime.
As an Administrator or Developer I want to get notified when an anomalous situation with a device occurs In order to prevent data loss.
As an Administrator or Developer I want to update the gateway firmware In order to solve issues and add new features.
As an Administrator or Developer I want to monitor system logs In order solve and prevent errors or downtime.
As a Developer I want easily deploy system component In order to solve issues and add new features.
As a Developer I want manage Administrator and Developer accounts In order add or remove staff from the team.

Table 3.5: 'Must have' requirements

Should have
As an Administrator or Developer I want to monitor system metrics In order to better understand my system and client needs.
As an Administrator or Developer I want to be alerted of gateway firmware errors In order solve and prevent errors or downtime.

Table 3.6: 'Should have' requirements

Could have
As an Administrator or Developer I want to perform gateway maintenance remotely In order solve and prevent errors or downtime.

Table 3.7: 'Could have' requirements

Won't have
As a User I want to configure complex patterns for measurement analysis In order to detect more complex issues.
As a Developer I want to perform batch processing on the reports received In order to better understand my system and environment behavior.
As a Developer I want to apply artificial intelligence on the reports received In order to better understand my system and environment behavior.

Table 3.8: 'Won't have' requirements

3.4 Quality Attributes

Quality attributes are the overall factors that describe the requirements of the system from a non-functionality view. They represent areas of concern that have the potential for application wide impact.

Load and business estimations were conducted during the gathering of requirements in order to support and determine the values required by the system and are documented in Appendix C. According to the estimations, the size of the database is expected to reach 6TB of information by 2020, which is a relatively small storage requirement; and a traffic of about 600 messages per second reaching the server.

Performance Requirements
The system shall be able to ingest at least 600 messages per second.
The system shall be able to store at least 6TB of data.
The system shall present latencies inferior to 6 seconds between ingestion of a messages and it being reflected on the data model.
The system shall present latencies inferior to 2 seconds when querying the data model.

Table 3.9: Performance requirements

Resource Requirements
The monthly communication bandwidth of each gateway shall not exceed 50 Megabytes.

Table 3.10: Resource requirements

Scalability Requirements
The system shall be able to scale out in under an hour until it is able to ingest at least 70000 messages per second while keeping latencies inferior to 2 seconds between ingestion of a messages and it being reflected on the data model.

Table 3.11: Scalability requirements

Security Requirements
Communication between the gateways and the server shall be encrypted and make use of authentication.
Communication between the time-series database and web application shall be encrypted and make use of authentication.
Communication between Administrators and device management should be encrypted and make use of both authentication and authorization.
Remote access to the system should be encrypted and authenticated.
The system should always use encrypted channels when crossing public networks and systems.

Table 3.12: Security requirements

Maintainability Requirements
An Administrator or Developer shall be able to remotely update a gateway in under 2 days.
An Administrator or Developer shall be able to remotely deploy new features to a gateway in under 2 days.
A Developer shall be able to implement new or change existent gateway components without having to alter other gateway components.
A Developer shall be able to implement new or change existent system components without having to alter other components.
A Developer shall be able to deploy new alert patterns to the system without having to alter other components.

Table 3.13: Maintainability requirements

Manageability Requirements
The system shall have centralized logging.
An Administrator or Developer shall be able to monitor the system remotely.
An Administrator or Developer shall be notified when the system is overburdened.
An Administrator or Developer shall be able to deploy a new gateway in under 30 minutes.

Table 3.14: Manageability requirements

Reliability Requirements
The system shall be able bounce-back from 99% of transient failures, within 15 minutes of downtime.

Table 3.15: Reliability requirements

Dependability Requirements
Each client should not have a daily data loss higher than 95%.

Table 3.16: Dependability requirements

4

State of the Art

This chapter summarizes the research conducted during the internship focusing on an IoT reference architecture based on state of the art patterns used in cloud service providers. There was also a focus on understanding the characteristics of a modern production infrastructure stack.

4.1 IoT Reference Architecture

The reference architecture for IoT systems served as a base for comparing state of the art cloud solutions and guide the design of the architecture for the project.

There are six key components in a standard IoT system: the devices themselves; the local gateway that sits between the device and the wider Internet; the cloud gateways that supports the devices; the business engine; the storage solutions and finally the device management component.

4.1.1 Device and Gateway

Devices interact with the world and collect environment data; they are usually small, cheap and have very few resources in terms of compute, storage, battery and connectivity. Devices usually need a local gateway who acts as a bridge to the Internet; gateways have a protocol translation role and could be able to execute local storing, filtering and processing of data before sending it to the cloud.

Gateways usually communicate with the cloud using HTTP polling, WebSockets, AMQP or MQTT. Both MQTT and AMQP are publish-subscribe messaging protocols based on a broker model. MQTT was specifically designed for IoT communication, has a very small overhead and supports intermittent connections.

4.1.2 Cloud Gateway

As the component responsible for the ingestion of data from local gateways, the cloud gateway serves as a shock absorber for incoming data streams and should be a durable message queuing service. The cloud gateway can also provide authentication and authorization mechanisms, as well as handling gateway health-checks.

4.1.3 Business engine

The business engine, or backend of the system contains all the components responsible for the processing and storage of data. The business engine usually follows a hot/cold architecture pattern, in which a 'hot path' processes data as a stream in near real-time, and a 'cold path' performs batch processing and artificial intelligence analysis.

The hot path usually uses complex event or stream processing to support near real-time actions based on data, while the cold path makes use of map-reduce tools such as Apache Hadoop to perform long running analytics. These analytics can also be used as input for machine learning predictive analysis systems to better understand the data and enhance the streaming patterns for example.

4.1.4 Storage

Both the data received from the devices, as well as the information extracted from the business engine needs to be stored for later consumption or analysis; device readings are usually stored in a time-series database, which enables on the fly aggregations and roll-up mechanics on data. Front facing components for user interaction and consumption are usually integrated with the system through the storage component.

4.1.5 Device management

The device management component consists of all services used to control and monitor the state of the devices and gateways, such as near real-time dashboards fed by the 'hot path', and gateway state management services.

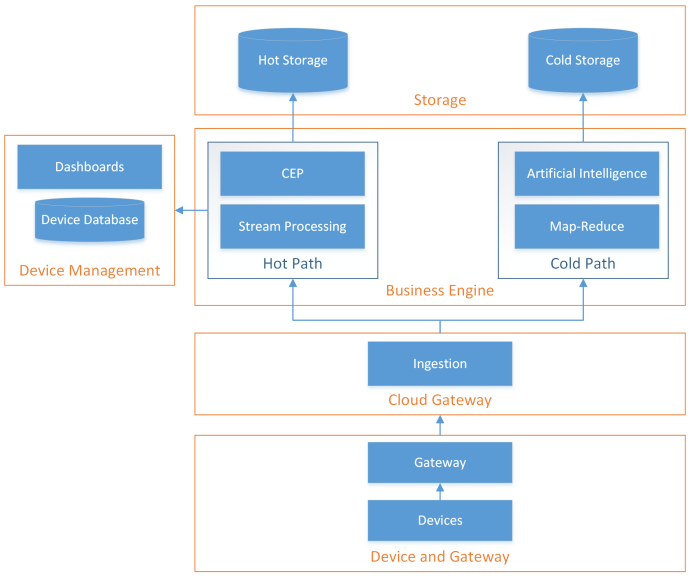

Figure 4.1: Reference IoT architecture

4.2 IoT Cloud Platforms

The end of 2015 saw the launch of new IoT Cloud platforms[20] from some of the biggest cloud providers. These platforms implement the cloud gateway and device management of the reference architecture and are designed to easily integrate with their other cloud services. This section compares the Azure IoT Hub[21] from Microsoft and the AWS IoT platform from Amazon.

4.2.1 Azure IoT Hub

Azure IoT Hub is a cloud gateway service that enables bi-directional communication between devices or gateways and the business engine in the cloud. The communication

channel is reliable, secure and the authentication is per-device using credentials and access control. Azure IoT Hub uses open standard protocols, such as HTTP and AMQP, and Microsoft provides SDKs for implementing the devices. Using IoT Hub devices can send and receive commands and, publish readings.

IoT Hub uses an identity registry where it stores all identity and authentication information about deployed devices, and provides monitoring information like connection status and last activity time. Using the API it is possible to create, retrieve, update, delete and enable devices. The connection between devices and the IoT Hub is encrypted using TLS, authenticated through X.509 certificates and uses authorization, based on identity per device or service.

The business engine component of the system can be implemented using other azure components that easily integrate with IoT hub. The cloud platform provides other services out of box like stream analytics and machine learning services to get data and execute predictive analysis with a lot of available predefined models.

4.2.2 AWS IoT

Unlike Azure, in the AWS platform the devices publish messages using the MQTT protocol. The device state is managed using a 'things shadow' model that stores state information, to which applications can request a change to the device state. The deployed devices will always attempt to mimic the state defined in their 'things shadow'. In practice the 'things shadow' is a JSON document, which can be customized with custom attributes that are part of the devices meta-data. AWS also provides SDKs for implementing the devices. Connection with devices is also encrypted and authenticated using TLS and X.509 client certificates respectively. Amazon also offers easy integration between the IoT and other AWS services using the Rules Engine module.

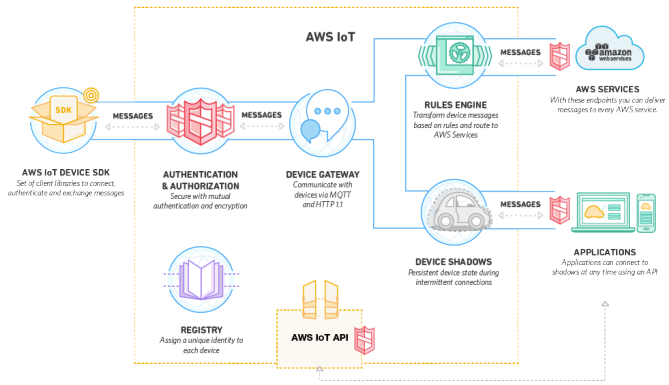

Figure 4.2: AWS IoT architecture

4.2.3 Conclusion

While different at implementation level, both services operate similarly in practice, the main difference being the management of device state; while in Azure commands are sent directly to devices, in AWS devices mimic the state defined on the cloud. The main difference between these services and Qold is the two-way communication and being able to manage each device individually, while in both Qold systems devices and gateways can only be managed in groups through the update system. These features were undesirable

for the Qold system, as it would increase the operational complexity, while offering few benefits for the use case.

Using these cloud services for the Qold system would mean locking it to these platforms which is not desirable for the project, and went against Whitesmith's business goals.

4.3 Operations

In order to reduce the operational complexity and be able to answer the reliability, maintainability and manageability requirements of the system an orchestration solution was necessary; but before comparing tools, it was important to understand the operational aspects of running a production infrastructure stack. This chapter introduces and explains these concepts, which serve as reference for Chapter 6.

4.3.1 Addressing

Figuring out where services are, and communicating that, is a problem being tackled by service discovery tools like ETCD[25], Consul[26] and others. Coordinating services and machines is not hard, but past a few services, the complexity of actually pointing humans to those resources becomes apparent. Realistically humans cannot handle remembering IP and port addresses, or having to update components every time a service restarts. DNS routing and addressing is still the go to solution, but it should not be a manual process.

4.3.2 Health Checking

In a modern infrastructure stack there are a lot of health check solutions integrated in the applications, HAProxy[27], Consul and Zookeeper[67] for example perform their own health checking. It is important however to define the boundaries for these to keeping a clean separation of responsibility, and keep logs and services from being overloaded with health checks.

4.3.3 Monitoring

In order to make sure the system is working as intended, system level statistics and application monitoring is important to prevent system overload and failures. Specialized services, like New Relic[30] and Datadog[29] simplify this process using intuitive dashboards with data gathered through client agents for standard interfaces.

4.3.4 Logging

While monitoring helps prevent issues, logs are for debugging and postmortem analysis. Ideally a centralized logging should be used to reduce maintenance costs and overhead. Popular solutions include Elasticsearch[31] interfaced via Kibana[32], and logentries[33].

4.3.5 Orchestration and Scheduling

Tools like Apache Mesos[34], Fleet[35], Docker Swarm[36] and Kubernetes[37], to varying degrees, supply an abstraction over heterogeneous server resources to give a consistent API for compute, memory and disk to applications, and ensure that those resources are distributed amongst client services. At application level some tools, like Storm[58], enable more complex service patterns, like having the ability to dynamically spin up workers across multiple hosts.

4.3.6 Provisioning

Provisioning refers to the configuration and deployment of machines to a usable state. It is the act of, for example creating and configuring a base linux machine, making it available to the cluster as resources to be consumed. State of the art tools for provisioning include Terraform[39], Ansible[53] and Chef[41].

4.3.7 Isolation

Isolation between applications or services is usually handled with either Virtual Machines (VM) or containers. While VMs are slower to provision, they provide a higher level of isolation. Containers usually provision significantly faster, however they can only make use of process level security. It makes sense to use instances when optimizing for resource isolation, and containers when optimizing for resource pooling.

4.3.8 Persistence

How to manage stateful services is probably the biggest hurdle in state of the art operations. Stateless containers and VMs can easily be cloned and all data shares the lifecyle of the application, but unless data is replicated and shared at application layer, which is not always possible, it needs a filesystem capable of archiving the same guarantees, without sacrificing performance. Distributed filesystems, such as HDFS[66] and AWS S3[43] are currently the best solutions.

4.4 Docker

Docker is a part of the operational stack, but was essential to the development of the system, and as such needs to be introduced.

4.4.1 Introduction

Rather than running a full OS on virtual hardware, container-based abstraction modifies an existing OS to provide extra isolation. Generally this involves adding a container ID to every process and adding new access control checks to every system call. Thus containers can be viewed as another level of access control in addition to the user and group permission system. Since a container does not waste RAM on redundant management processes it generally consumes less RAM than a VM.[71]

Containers are frequently described as lightweight runtime environments with many of the core components of a VM and isolated services of an OS designed to package and execute these micro-services. While containers have long existed as extensions to Linux distributions, each has come with its own flavor. The rise of open source Docker containers over the past year has created a standard for how applications can extend from one platform to another running as micro-services.

Docker containers[70] have recently become available with major Linux distributions and are supported in key cloud services, even Microsoft just rebuilt the back-end of Docker to allow containers to run on Windows.

In many ways, Docker and similar lightweight containers promise to transform the role of the OS and the VM much like the VM has done to the physical bare-metal server environment. While each virtual machine includes the application, the necessary binaries and libraries and an entire guest operating system, all of which may be tens of GBs in

size, containers include the application and all of its dependencies, but share the kernel with other containers. They run as an isolated process in userspace on the host operating system.

Currently there are more than 4 Million developers using Docker, which is impressive for a 3 years old project. Giants of the tech world are already actively using it, such as EA Games, whose development and infrastructure completely shifted to Docker. And while Google who supports Docker, with their Kubernetes platform for example, does not internally use Docker, they have been using containers for many years.

4.4.2 Advantages

Docker aims to allow developers to build services using whatever software they want. Due to the open source movement, there are a lot of software and platforms available to choose from, but those choices have different operational implications. Docker diminishes those implications, allowing developers to use whatever software they want, and make sure they work in production. Because it encapsulates everything needed for an application, from OS to the application binary in a single package, the environment the application is running in development, is exactly the same it will run in production. This is extremely helpful in Continuous Delivery.

4.4.3 Docker Ecosystem

Docker follows a git inspired workflow; changes to containers are committed and can be pushed and pulled to the container registry. A Docker container starts as a Dockerfile, a declarative file where the container base state, operative system, and software installed is specified. This file is compiled to a Docker image, which is actually composed of several image layers. The image represents the read-only component for the container, and its starting point, an image can be instanced multiple times, creating the actual container. The container can be committed to a new image that can be used to create new containers. Images can also be referenced in the Dockerfile, as a base for new images, for example it is possible to write a Dockerfile using an Ubuntu image as a base, write changes to the Dockerfile and compile it to an image. Image layers are also shared among images, for example two images, both built on top of an Ubuntu image will share some image layers, reducing the amount of data that needs to be downloaded and stored.

Docker provides a set of simple tools to manage and orchestrate applications that were used during the internship.

4.4.3.1 Docker-Engine

Docker itself is just a CLI front-end that uses a Rest API to communicate with a Docker-engine. Docker-engine is a daemon that actually runs and manages containers.

4.4.3.2 Docker-Compose

Typically, as the complexity of the system increase, the number of containers per service/system increases rapidly, and orchestration starts to become a problem. To reduce the complexity of managing and deploying containers, Docker-compose was introduced, it uses YAML to describe the containers, their connections, ports, volumes, and other configurations. Using Docker-compose, a single command can launch a complex system described in the YAML file. It also provides useful functionality, like scaling the number of deployed containers from a selected image. While Docker-compose is not directly used

in the project, it is useful to locally simulate the entire cluster, increasing development and testing speed.

4.4.3.3 Docker-Hub

Docker-hub is the public image registry. It has more than 150.000 images, some of them official, but most are community made images. When the docker-engine fails to find an image locally, it will attempt to retrieve it from the hub. The Docker Hub is extremely important for sharing docker images with the cluster; essentially whenever a developer builds or updates an image, it is pushed to the hub and each cluster node will retrieve it from there.

5

Architecture

The architecture specification is an essential step of any software project; it defines the structure and behavior of the different parts of the system, while being a guideline for its implementation. This section of the report discusses the architecture of both the legacy and the new system, architectural decisions and a description of the tools used and considered.

The documentation of the architecture followed the guidelines proposed by Simon Brown[44], which describes four levels of abstraction: 'A software system is made up of one or more containers (web applications, mobile apps, standalone applications, databases, file systems, etc), each of which contains one or more components, which in turn are implemented by one or more classes.' The Component abstraction level is the lowest necessary to understand the system.

As a reference, for the rest of this chapter, a measurement refers to a single temperature reading from a single device, a batch refers to a collection of one or more measurements from one device and a report is a group of batches with one or more measurements from one or more devices. The Developer and Administrator roles have been merged for the duration of this chapter, as their architectural significance and interaction is the same.

5.1 Legacy System

5.1.1 System Context

The software system diagram shown in Figure 5.1 represents what the system is, who is it being built for and how it fits in the existing environment:

- The system is used by the three groups of users defined in the stakeholders section of the report: Users, Administrators and Developers (Merged in Administrator).
- The following external software services are being used:
  - Logentries is log management web service, being used to store and query system logs.
  - New Relic performs routine performance analysis of system nodes and reports noteworthy situations.
  - Github is a Git repository hosting service, which is being used to keep source code and gateway updates.
  - Keen is a dashboard analytics web service, being used to monitor the state of gateways, devices and QoS metrics.
  - Twilio is a web service with a Rest API that sends SMS notifications to users.

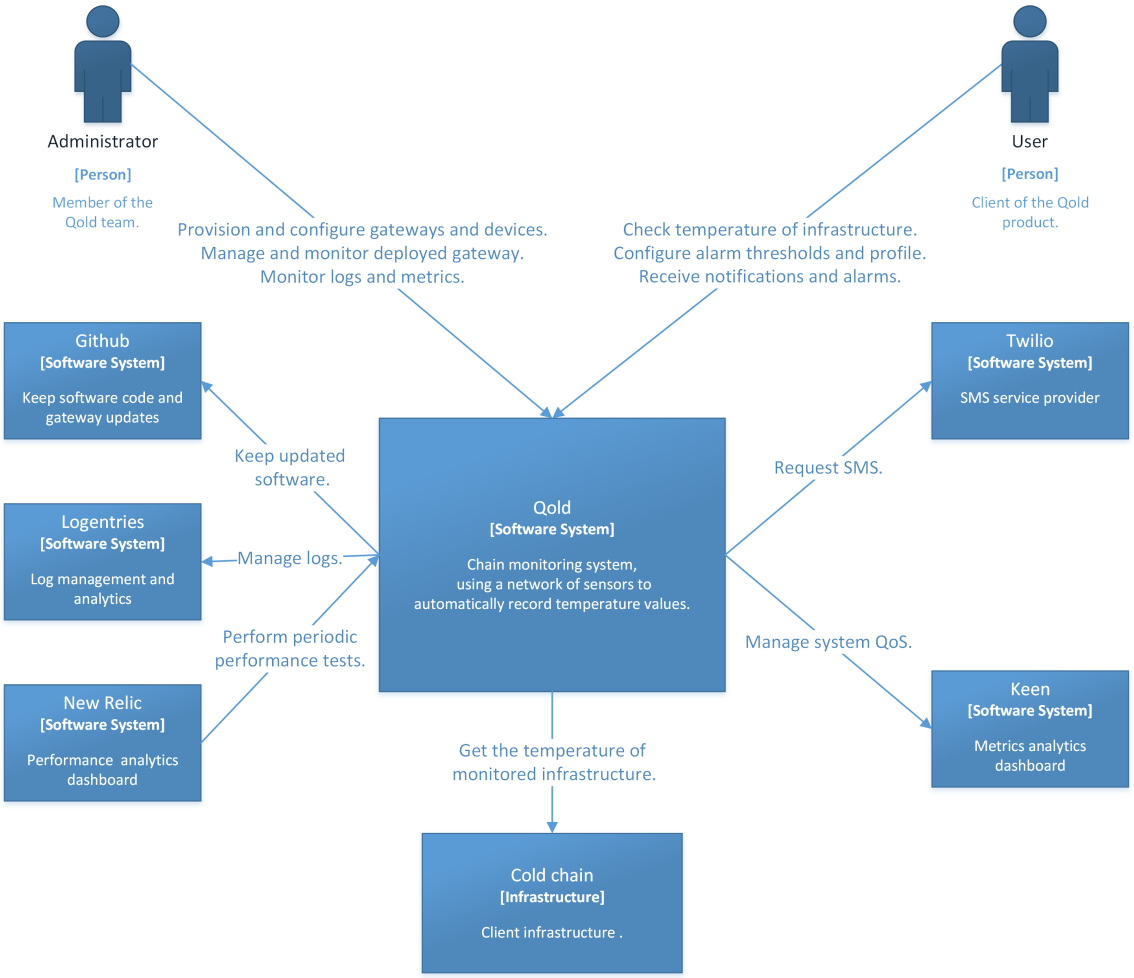

Figure 5.1: Legacy System: Context

5.1.2 Container Context

The container diagrams shows the high-level shape of the software architecture and how responsibilities are distributed across it. It identifies the major technologies used and how the containers communicate with one another. For clarity the Administrator and User interactions are described in separate diagrams.

5.1.2.1 Device and Gateway

The system has two client side components, the devices and the gateways. A device is a small, low power device which takes temperature readings from the environment. In order to keep power consumption to a minimum, this component does not possess Internet connectivity or any functionality beyond reading and broadcasting measurements through radio signal. The gateways are bigger, more powerful devices that receive readings from several devices and publish them to the cloud through the Internet.

The gateways collect, store and deliver device measurements. Their software is written in Python and is kept updated through a bash script that periodically pulls the source code from Github. Any crashes, errors or reboots trigger a software update. Every time a gateway receives a measurement, it attempts to send all undelivered messages to the cloud. Undelivered measurements are temporarily stored on a SQLite database.

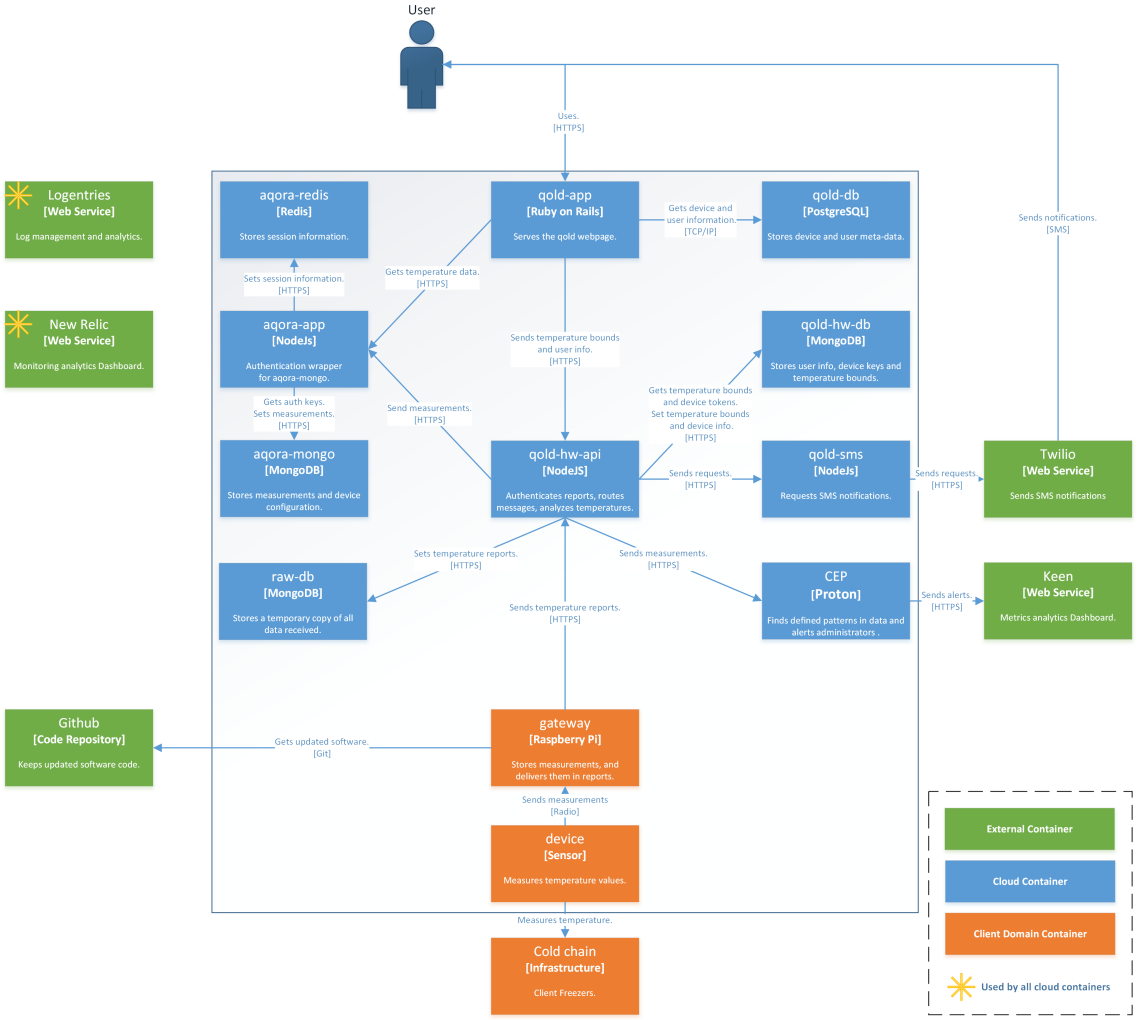

Figure 5.2: Legacy System: User

5.1.2.2 Qold Hardware API and DB

The Hardware API (qold-hw-api) is a NodeJs application and the entry-point of the system. Messages received from the gateways are stored in a Raw database, authenticated against a local database and sent to Aqora and CEP. If the temperatures received are too high or too low, the Hardware API will also notify the clients using a SMS service. The local database is called Qold Hardware Database (qold-hw-db) and is a MongoDB database that stores device information, such as defined temperature bounds and authentication data.

5.1.2.3 Raw Database

The Raw Database (raw-db) is a MongoDB database that stores reports as they are received from the gateways for replay and debug purposes.

5.1.2.4 Aqora

Aqora is a time-series database created at Whitesmith, it is composed by three applications: aqora-app, aqora-mongo and aqora-redis. Aqora is used to store and query the

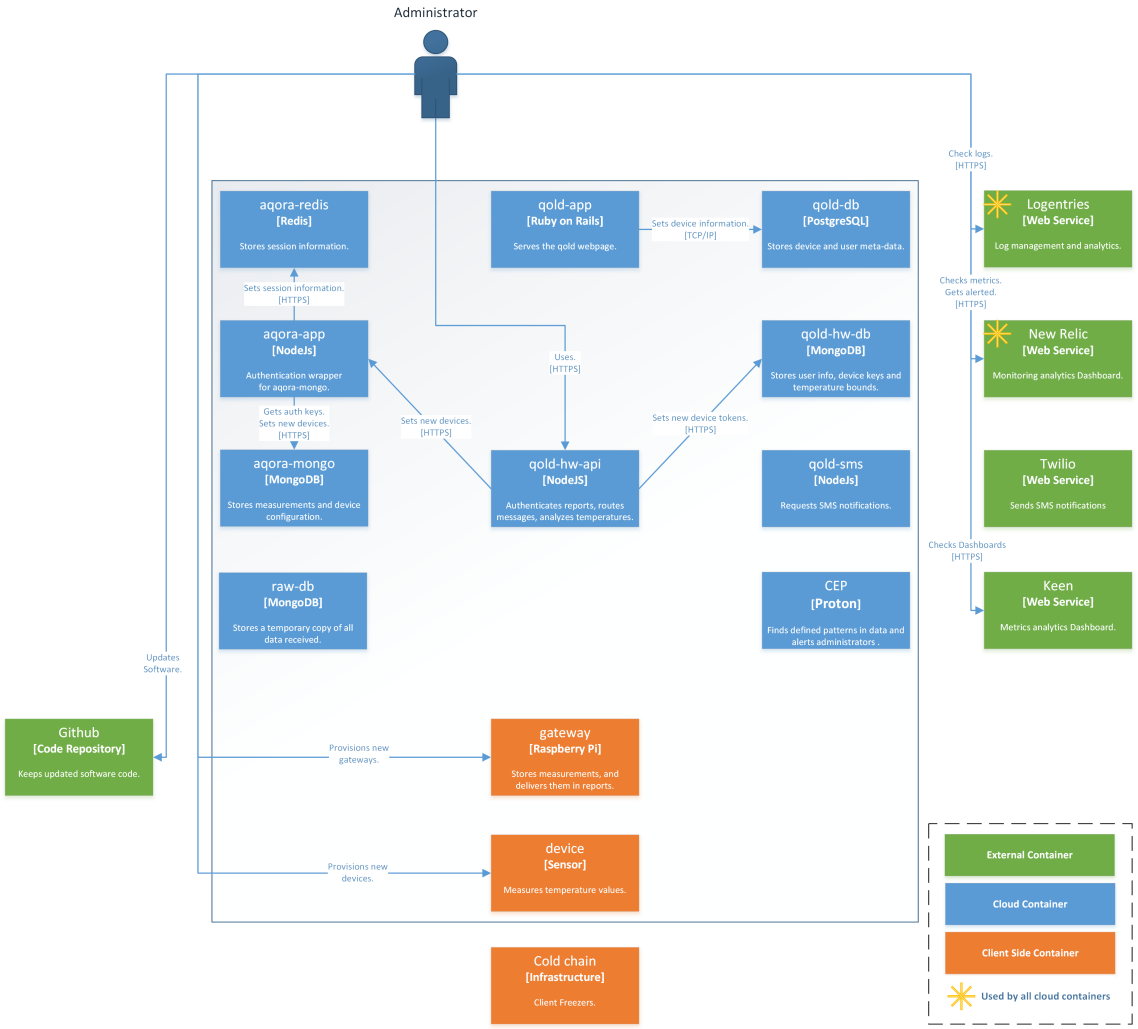

Figure 5.3: Legacy System: Admin

measurements received from devices. The aqora-app is a NodeJs application that serves as an authentication wrapper to aqora-mongo which is a MongoDB database. The session data is stored on aqora-redis, a Redis database.

5.1.2.5 Qold SMS

Qold-sms is a simple NodeJs application that receives HTTP requests with a message and phone number and forwards them to Twilio[52]. It serves as an encapsulation to client messaging.

5.1.2.6 Aqordinator

Aqordinator is the application used by users to interact with the system; it is composed by two components: Qold Application (qold-app), and Qold Database (qold-db). The qold-db is a PostgreSQL database which contains device meta-data, such as which user they are deployed to, their name, where they are, and user info. The qold-app is a Ruby on Rails web application that serves user requests; it hosts the website where clients connect to check their temperature readings, change alarm configurations and other management

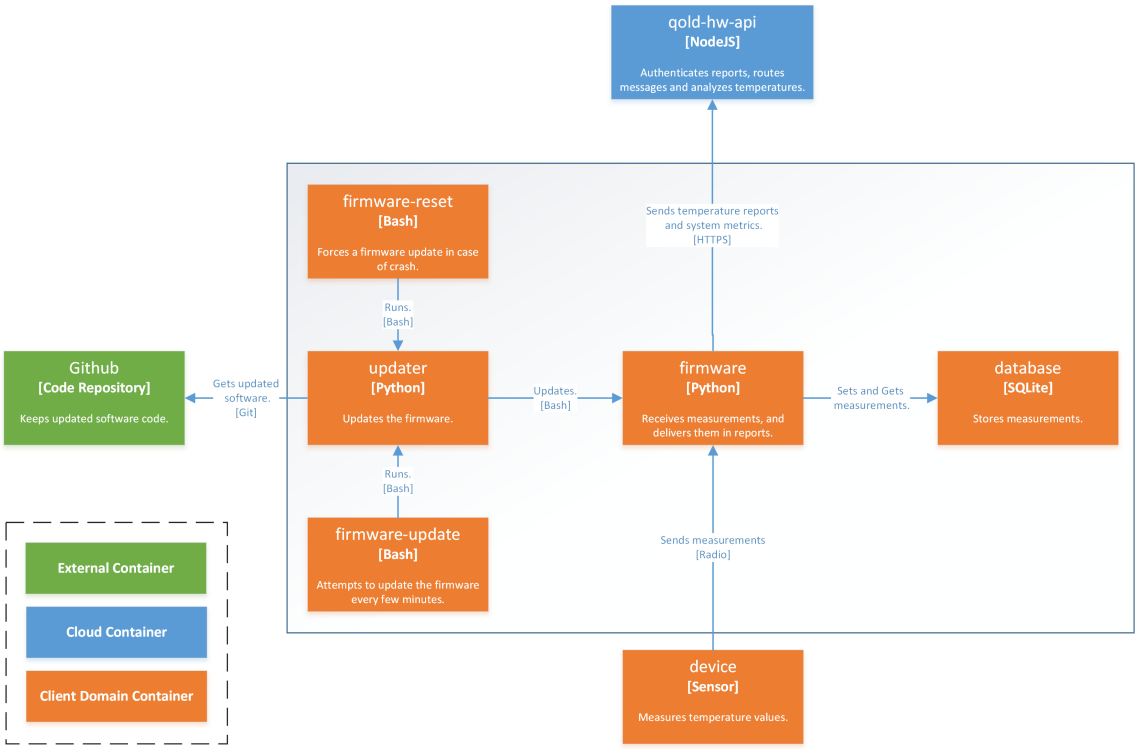

Figure 5.4: Legacy Gateway architecture

operations. Temperature measurements are read from aqora, device information is read from qold-db, and device temperature bounds are written and read from qold-hw-api.

5.1.2.7 CEP

CEP is a Proton[11] complex event processing engine, it detects defined patterns in the data received from the qold-hw-api. The CEP application is only being used for QoS, alerting the Qold staff when problems with devices are detected.

5.2 Architecturally Significant Requirements

The following are requirements that had a profound effect on the design of the architecture. They were extracted from the quality attributes of the system and discussions with the team members. An Utility Tree was used to capture the ASRs, prioritizing them in terms of impact on architecture and business or mission value, ranging from Low to High in the format (*Impact*, *Value*).

Quality Attribute	Refinement	ASR
Performance	Throughput	At peak load, the system is able to ingest 600 measurements per second. (H,H)
	Latency	At peak load, the system should process a measurement and save it to the data model in less than 6 seconds since ingestion. (H,L) A user accesses the application in order to check the temperature of his infrastructure, and the information is queried in under 2 seconds. (M,M) A fridge is left open and the temperature rises beyond acceptable levels; the user is notified within 30 minutes of detection. (M,H)
Scalability	Scale Out	Up to 70.000 devices now make readings each second and the system can be scaled out in order to, at peak load, maintain the same max latency as before. (H,M)
Maintainability	Upgrades to gateways	The gateway-device communication protocol and programming language changes, and a developer can update all deployed gateways with the new software in under 2 days. (M,H) A new service needs to also consume the authenticated messages received by the system, and a new hire with two or more years of experience in the business is capable of integrating the new service in under 1 person-days of effort. (H,M)
Dependability	At least once	The system guarantees at-least-once semantics for message processing. (H,H)

Table 5.1: ASR Utility Tree

5.3 Architectural Style

In order to better respond to the ASRs, the Microservices architectural style was chosen for the system. The microservices[22] architectural style is an approach to software development where a single application is constructed as a suite of small services, each running in its own process, and communicating using lightweight mechanisms. These services are split based on business capabilities and are independently deployable by fully automated processes; there is usually a bare minimum of centralized management of these services. Current supporters of the microservice architecture include Amazon, Netflix, The Guardian and Twitter.

Contrasting with microservices are monolith applications, where all functionality is built into a single large application. Changes to these monolith systems involve building and deploying a new version of the applications, and they can only be horizontally scaled by running many instances behind a load-balancer. Microservices on the other hand offer greater control and granularity since each service can be independently deployable and scalable.

5.3.1 Benefits

5.3.1.1 Technology Heterogeneity

A system composed of multiple services, can have different technologies inside each one. The right tool for each job can be chosen, rather than having to select a more standardized, one-size-fits-all approach that often ends up being the lowest common denominator. If one part of the system needs to improve it's performance, the technology stack can be changed completely to achieve the performance levels required. Data can also be stored differently in different parts of the system.

5.3.1.2 Dependability

When developing a system it is important to make sure that component failure does not cascade, problems need to be isolated so that the rest of the system can carry on working. In microservices, service boundaries become obvious barriers, but in a monolithic system if a component fails, everything stops working. Failure can be handled much easier with Microservices, although there are new sources of failure that distributed systems have to deal with, such as network.

5.3.1.3 Scaling

As mentioned, in a monolithic system every component needs to be scaled together, while in a Microservices system, if one service is constrained in performance, it can individually be scaled. Also, because services can be physically separated, instead of a very powerful machine, several cheaper weaker machines can be used.

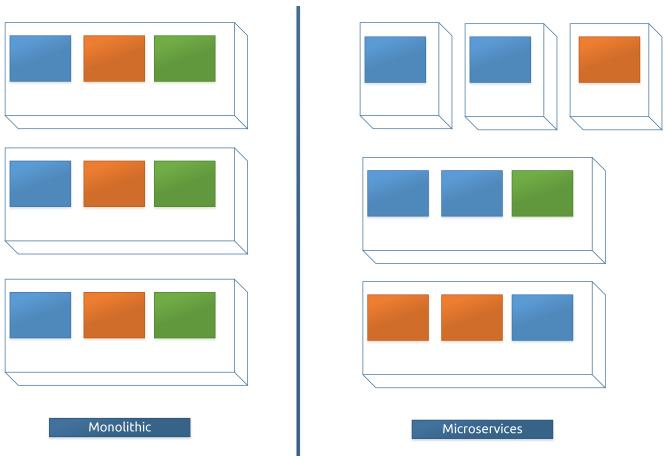

Figure 5.5: Microservices scalability

5.3.1.4 Ease of Deployment

A single change in a monolithic application can require the whole application to be deployed in order to release the change, making it a large-impact, high-risk deployment. Because of that, these deployments end up happening infrequently due to understandable fear, but that translates to more time between releases, until the new version has masses of changes. This not only delays new features, but has an increased chance of failing. With Microservices, each deploy happens independently of the rest of the system. This allows faster releases and if a problem does occur, it can be isolated quickly.

With Microservices, the development team usually takes full responsibility for the software in production, bringing developers into contact with how their software behaves in production. Microservices link perfectly with the DevOps movement.

5.3.1.5 Isolation

Another consequence of using services as components is a more explicit component interface. Often it's only documentation and discipline that prevents clients from breaking a component's encapsulation, leading to overly-tight coupling between components. Services make it easier to avoid this since they have strongly defined bounds. Applications built from Microservices aim to be as decoupled and as cohesive as possible, they own their own domain logic and act more as filters, receiving a request, applying logic as appropriate and producing a response. They are usually organized using REST rather than complex protocols. The two most commonly used protocols are HTTP request-response and messaging over a lightweight message broker.

There is also a decentralization of data management and storage. Each service has access only to the information it requires to fulfill its function, and usually manage their own database, either different instances of the same database technology, or entirely different database systems.

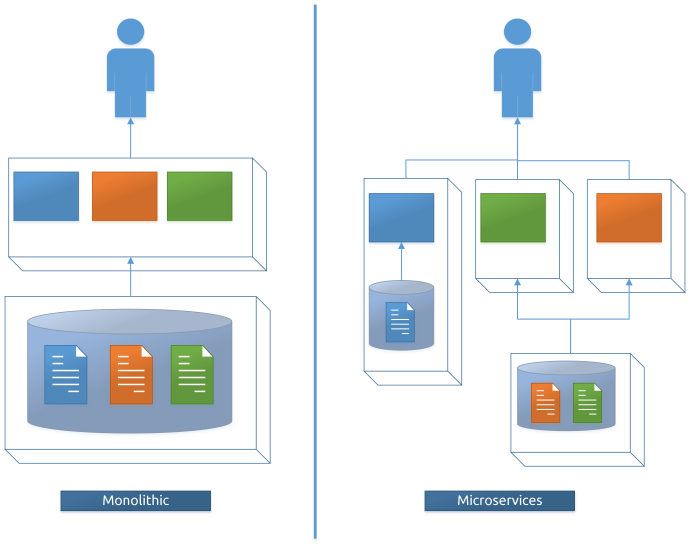

Figure 5.6: Microservices Storage

5.3.2 Disadvantages

As the Microservices pattern is a subset of distributed computing, they are bound to have the same dangers and disadvantages, not commonly found in monolithic systems. These issues are well documented in the *'Fallacies of Distributed Computing'*[24], a set of common misconceptions when dealing with distributed systems. They were introduced in 1994 by L Peter Deutsch.

5.3.2.1 The network is reliable

Network failures are usually not obvious to detect, furthermore when using middleware technologies, where the writing of code be very close to the experience of calling a local

function. Not only that, but when testing a service, local mock data is usually used to represent other services. These situations can create the illusion that the network is reliable, and cause unpredicted behavior when they are put in production or when a network error occurs.

In order to mitigate this issue, the Qold system test suits simulate the loss of services when communicating, and components were kept decoupled whenever possible.

5.3.2.2 Latency is zero

One failure scenario that is commonly tested is for when dependent services simply are not running, that is a useful test but it is probably less likely to replicate operational issues, compared to the scenario where dependent services just run a lot slower. These cases might have unpredictable consequences in the system, such as duplicating data or creating unexpected bottlenecks in the system.

System and component benchmarks were used to identify bottlenecks and choose optimal levels of parallelism and timeout configurations.

5.3.2.3 Bandwidth is infinite

Load balancers usually only cause request latency degradation after they have reached complete bandwidth saturation; knowing that they are close to reaching that point is better than having to react to it. Client-side load balancers are less likely to become bottlenecks, so they are better for environments that have many services.

In order to address this issue DNS round-robin load balancers are being used for gateway connections, and the tools used also reduce this issue at application level when advertising the nodes for incoming connections. Services connect directly to the desired node, there is no need for load balancing.

5.3.2.4 The network is secure

Connections crossing the Internet need to be secure, but adding layers of security will affect the performance of the system. There needs to be a trade-off between security and performance.

5.3.2.5 Topology does not change

Discovery and routing tools are important when building a resilient distributed system; topology shifts always occur in production and will difficult management and reconfiguration. Also, because many topology changes occur due to unexpected failures occurring in production, these reconfigurations will have an increased impact on the system.

The project makes use of both discovery and routing solutions which are part of the orchestration tools. It allows for services to be discovered and connected to using FQDN.

5.3.2.6 There is one administrator

In a Microservices system, because the development team is also responsible for managing and deploying the services, each developer will also play the role of administrator. Operational controls are important, to prevent the production environment from becoming chaotic. Microservice teams should use sophisticated monitoring and logging setups for each individual service such as dashboards showing up/down status and a variety of operational and business relevant metrics.

This fallacy is also addressed in the manageability requirements and the system makes use of centralized logging and monitoring solutions.

5.3.2.7 Transport cost is zero

Going from the application layer to the transport layer is not free, information needs to be serialized to get data onto the wire, not only that but the use of network protocols, instead of local calls has an impact in performance. The cost of managing the network is also not not zero, handling addresses and bandwidth has a monetary and operational cost.

This fallacy cannot be properly mitigated as it is a consequence of the nature of a distributed system. In the current project, the scalability requirements were more important, and the loss of performance was acceptable.

5.3.2.8 The network is homogeneous

Standard technologies and protocols, like HTTP, and data formats, such as JSON, were used to increase interoperability and portability of the system.

5.3.3 Conclusion

This introduction to the Microservices architectural style helps understand the context and considerations taken during the development of the architecture of the system. Being a cloud based application the use of Microservices allows for each service to scale-out, be developed using optimized technologies and be easily replaced and managed.

5.4 System Architecture

5.4.1 System containers

The final architecture for the system can be found in Figures 5.7 and 5.8, followed by the description and explanation of each container. The technological choices are also explained. Figure 5.11 shows the same system, but with a microservices oriented view.

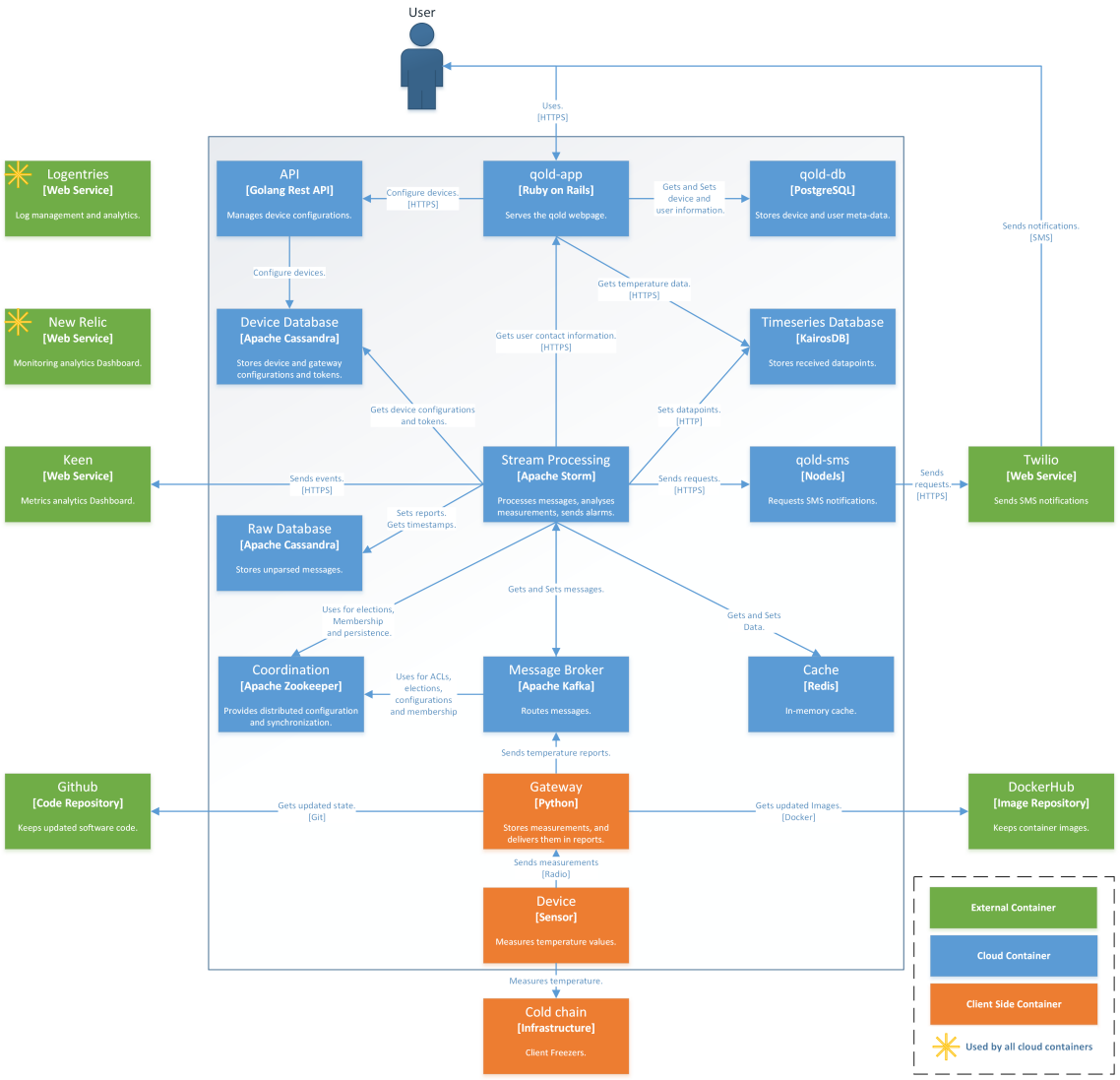

Figure 5.7: System Architecture: User

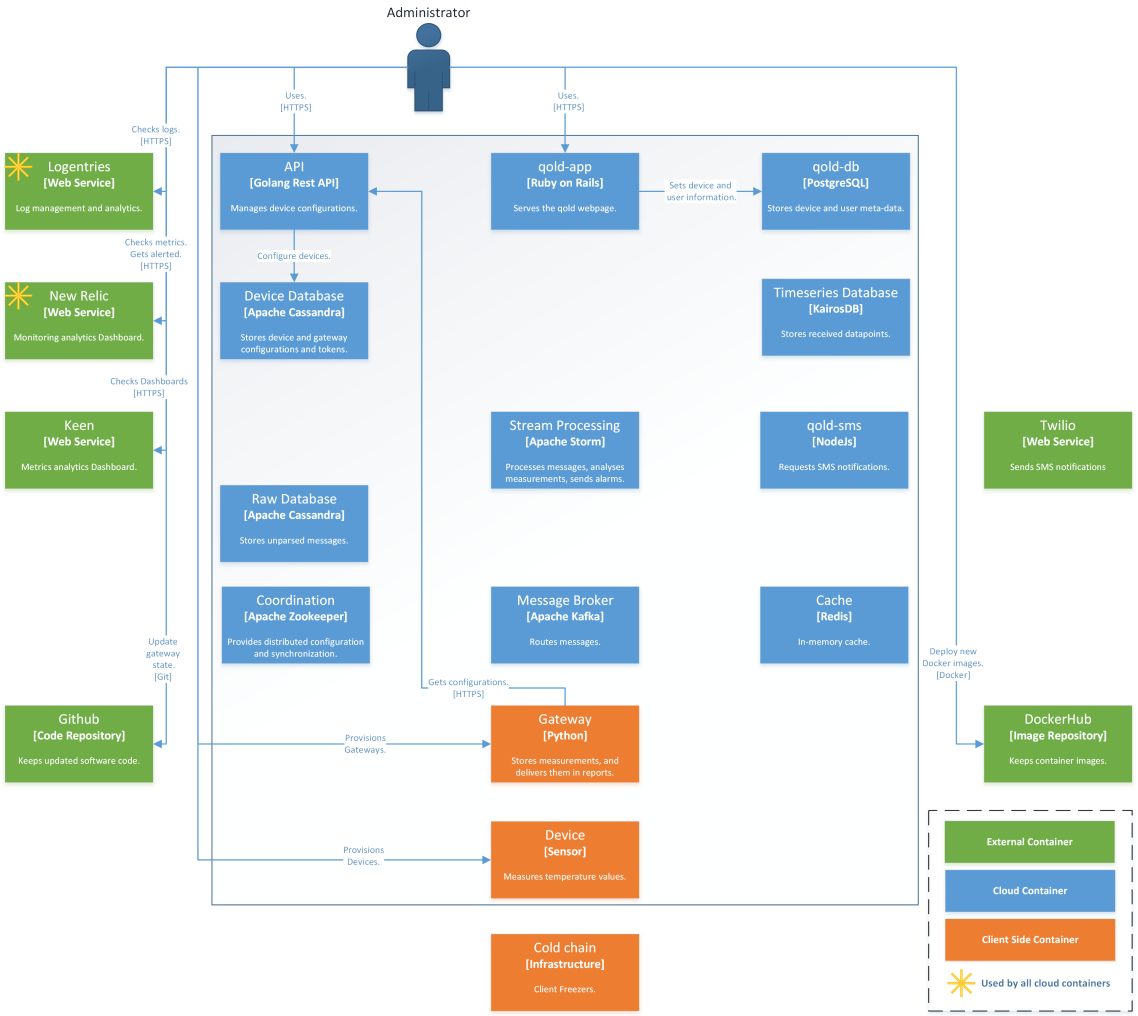

Figure 5.8: System Architecture: Admin

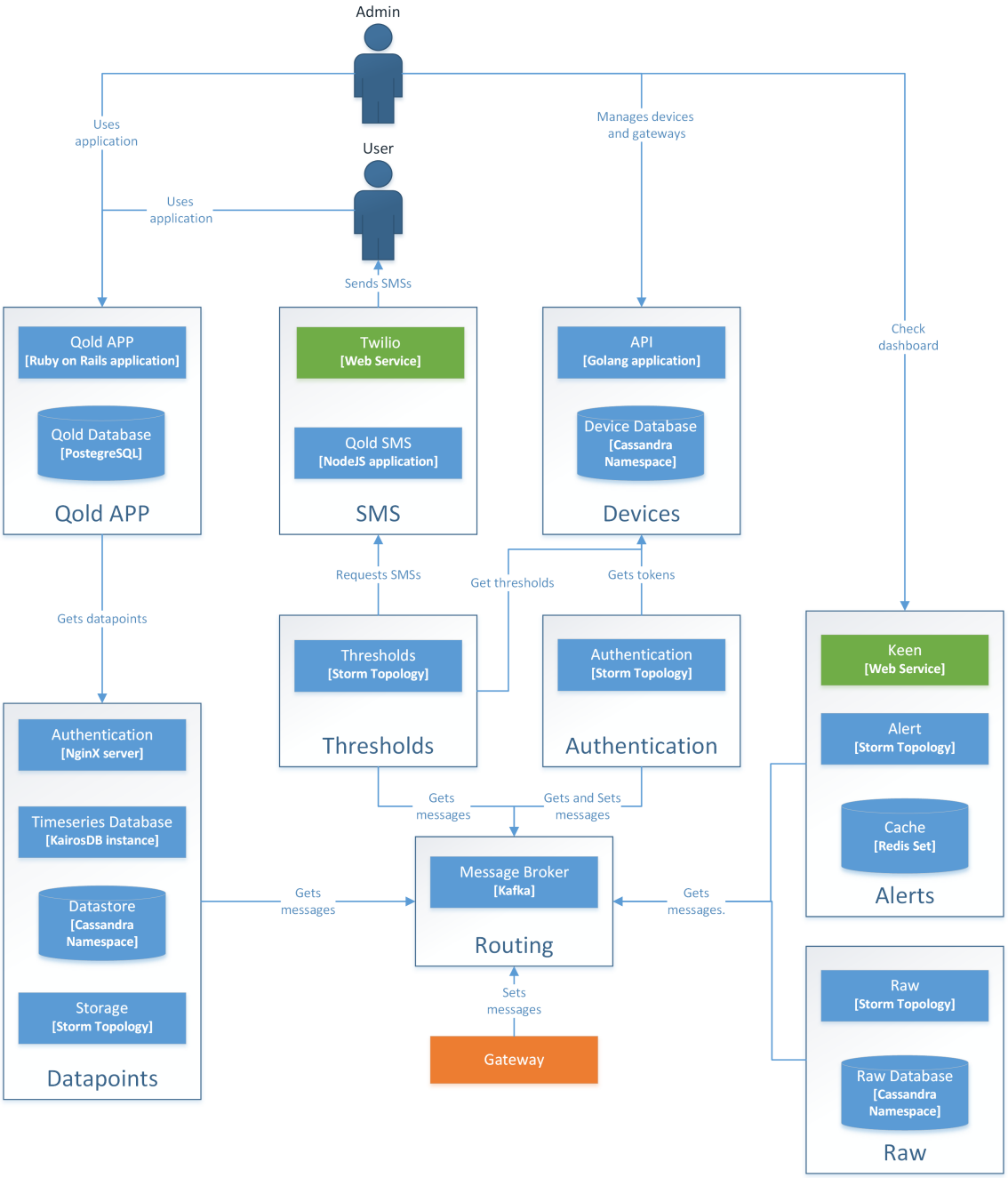

Figure 5.9: System Architecture: Services

5.4.1.1 Legacy containers

Before describing the new containers, it is worth noting that the 'qold-app', 'qold-db' and 'Device' were not changed from the legacy system, as they fall outside the scope of the internship; the 'qold-sms' container did not require changes and was also left intact.

5.4.1.2 Gateway

The architecture of the gateways were changed in order to increase the simplicity and dependability. This container still serves the same purpose as it did in the legacy system,

handling communications between the devices and the cloud, but while all gateways were exactly the same, they now have a identity. In the new system, messages sent must provide a 'gateway id' and 'gateway token' for authentication; this new feature allows for the monitoring of gateways, enabling them to send other data, such as metrics or exception. While it was planned for every gateway to have its own X.509 certificate for authentication, most 'Message Brokers' were not ready to handle a large number of identities; in the end, all gateways share the same client certificate for authorization, but need to use the 'gateway token' for authentication.

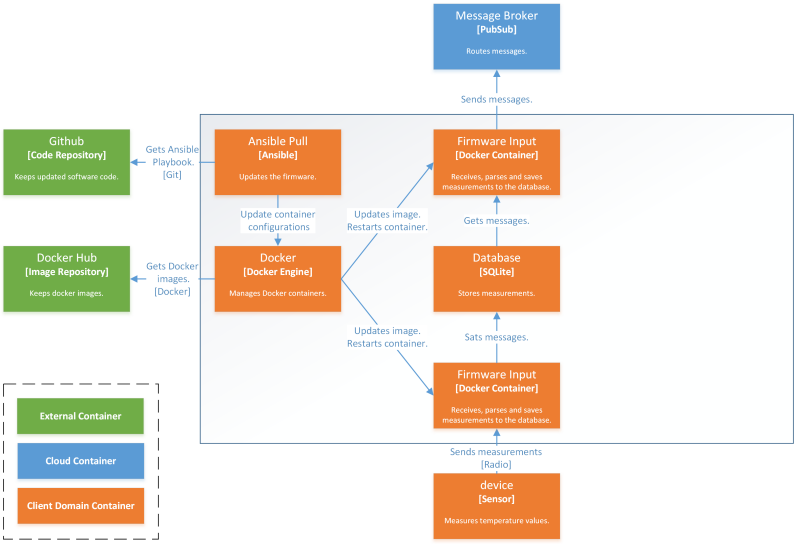

Figure 5.10: System Architecture: Gateway

The update mechanism was also changed, while it used to rely on bash scripts for updating the software, it now uses an Ansible[53] playbook. Once a day, and every time the gateway reboots, it performs and Ansible-pull from github and runs the playbook locally. The best advantage of the new method is that Ansible playbooks are idempotent, and as such will only make changes to the gateway if it does not match the desired state.

The software was also divided into the 'input' and the 'output' components, creating a clear separation between the gathering of readings and the sending of messages to the cloud. These components were developed to be independent of each other, and communicate only through the local SQLite database, that also existed in the legacy system. This separation makes each component simpler to implement, provides isolation in case of failure and allows for the development of components without having to worry with other communications.

Finally, the gateway components were changed to be docker containers. This change provides further isolation, allows for a more diverse range of updates, and ensures that software restarts always bring the components to their initial state. The docker engine is also responsible for restarting components whenever they crash, ensuring they are always running.

5.4.1.3 Message Broker

The 'Message Broker' is a Publish-Subscribe messaging middleware; it replaces the qold-hw-api, is responsible for the ingestion of all gateway messages and the integration of the services in the system. As all asynchronous communication in the system is handled

by this broker, performance, scalability and interoperability were the major requirements when choosing a tool. Associated with the choice of message broker is the communication protocol for both gateway and cluster communication. It is also worth noting that the gateway communication needs to be encrypted and authenticated.

Several tools were evaluated for this role, but it came to either RabbitMQ[54] or Apache Kafka[55] for message broker, and MQTT[56] or AMQP[57] for communication protocol.

At the time of the initial comparison, Kafka 0.9 had just came out, and its analyses provided the following points:

- The Apache Kafka server is simple to setup and scale, the clients on the other hand are not. Only the Java client is officially supported, and while there are open source implementations in other languages, most only still support the 0.8 version of Kafka.
- Kafka persists all messages and they are only removed after a specified period of time; whether they have been consumed or not is not verified, which is interesting for replayability.
- Benchmarks showed that performance is the selling point. Kafka is unmatched in this aspect, being able to ingest millions of messages per second.
- Tests showed that publishing a dummy qold reading, with SSL enabled has a cost of around 6 Kbytes. Apache Kafka also allows for message compression if needed.
- Apache Kafka broker documentation, especially architecture and design are very complete; clients on the other hand depend on the implementation.
- Before Kafka 0.9 there was no tracking of message consumption, the clients needed to know which messages they had consumed themselves. In Kafka 0.9, the broker can save the offset of the last message consumed per client.
- Before Kafka 0.9 there was no support for encryption, authentication or authorization.

In summary, Apache Kafka was a really good choice due to its simplicity, replayability and performance but version 0.8 lacked essential features, and at the time only Java was supported, so it was initially dismissed.

The analysis of RabbitMQ provided the following results:

- RabbitMQ is harder to setup, configurations are not as simple as Kafka and some operations require a broker restart.
- Since RabbitMQ uses mature open standards, most client implementations are very complete and documented.
- Encryption using TLS, and client authentication and authorization are available.
- The broker tracks consumed messages, and they are only deleted once all subscribed consumers have acknowledged them.
- RabbitMQ allows the use of several protocols, such as MQTT and AMQP.
- RabbitMQ can also be horizontally scaled, and while performance is not comparable with Kafka, it is enough for the Qold requirements.
- Tests with SSL encryption and using MQTT showed a bandwidth cost of 2 Kbytes when delivering dummy qold readings.

In summary, while not as performant and simple as Kafka, the RabbitMQ message broker was enough to handle the predicted load; it also offers a more lightweight communication protocol, all the required features and multi-language support.

At the time, the only viable solution was the RabbitMQ message broker. Apache Kafka 0.8 was too simple and lacked required features, such as TLS, authentication and offset tracking, and 0.9 was still not supported by most clients. On the other hand RabbitMQ was a very stable and mature broker, it was supported by many open protocols,

implemented all the necessary features, and met the performance and scalability requirements. RabbitMQ was deployed, and stayed in production for over a month; gateways used the MQTT protocol to communicate with it, due to its lightweight nature, and the AMQP protocol was used internally.

By the time 'Stream Processing' engines were compared, later in the internship, Kafka 0.9 was already supported by most languages, in fact Kafka 0.10 had already been released. During this study, it became apparent that the market was starting to abandon RabbitMQ in favor of newer tools such as Kafka; the RabbitMQ integration with most engines had not been updated in several years, for example. At this time, it was decided that a migration of the system to Apache Kafka was the best solution: integration with 'Stream Processing' engines was significantly easier, the performance and scalability were much higher and all the necessary features were now implemented for the major languages. At this time, the 'gateway output' docker container had already been developed and deployed to clients, so this remote update to Kafka communication also helped validated the architectural choices for the gateway.

5.4.1.4 Stream Processing

The 'Stream Processing' container is responsible for all the message processing in the system. This container is a distributed near real-time computation system, and needed to have high parallelism, performance and scalability. The 'Stream Processing' engine needed to be able to execute the following workflows:

- Authenticate gateway messages.
- Persist authenticated messages to the Raw Database.
- Parse and persist authenticated messages to the time-series database.
- Alert when a temperature or battery value is outside of configured thresholds.
- Alert when a device goes too long without reporting, and when it comes back.

Several tools were evaluated for this role, most notably Apache Storm[58], Apache Flink[59] and Apache Spark(Streaming)[62].

Apache Spark is a fast and general purpose batch processing engine where workflows are defined in a style similar of MapReduce. The Streaming API framework for Apache Spark allows for continuous processing via short interval batches. Apache Spark was dismissed because it is batch oriented and the system requires individual message processing; its data parallel paradigm also requires a shared filesystem such as HDFS[66] which has high hardware requirements.

Apache Flink was an interesting option because it offers more primitives than just Stream processing, such as Complex Event Processing, Machine Learning and graph processing. According to benchmarks it also has good performance and scalability[45]. In the end, Apache Storm was chosen over Apache Flink due to its simpler development paradigm. Apache Flink workflows can also only be implemented in Java, which was not favored at Whitesmith.

Apache Storm is a distributed near real-time computation engine, designed to compute parallel tasks across a large number of nodes. There are two kinds of nodes on a Storm cluster: the master node and the worker nodes. The master node runs a daemon called 'Nimbus'. Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures. Each worker node runs a daemon called the 'Supervisor'. The supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it. Each worker process executes a subset of a topology; a running topology consists of many worker processes spread across many machines. Coordination between Nimbus and the Supervisors is

done through Zookeeper. Both the Nimbus and the Supervisor daemons are fail-fast and stateless, since all state is kept in Zookeeper. Workloads on Storm are called 'topologies', which are Directed Acyclic Graphs, and can be individually deployed, making them akin to services. Each element in a topology contains processing logic, and links between nodes indicate how data should be passed around.

The core abstraction in Storm is the stream, an unbounded sequence of tuples. A tuple is a named list of values, and a field in a tuple can be an object of any type as long as it is serializable. Storm topologies are made up of elements called 'spouts' and 'bolts', which implement the application-specific logic. A spout is a source of data, it can be for example a Kafka consumer. A bolt consumes any number of input streams, does some processing, and emits new streams. Complex stream transformations require multiple steps and thus multiple bolts. Bolts can be implemented in multiple programming languages and can do anything from run functions, streaming aggregations, or connect to databases. When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream.

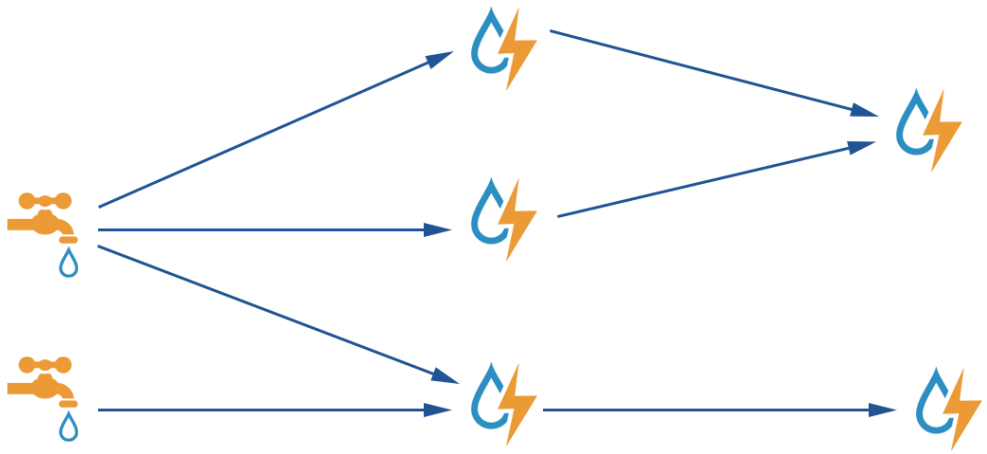

Figure 5.11: Storm Topology

Each node in a Storm topology executes in parallel, and the level of parallelism can be specified, and then Storm will spawn that number of threads across the cluster to do the execution, always attempting to spread those tasks across machines. Storm will automatically reassign any failed tasks and Storm guarantees that there will be no data loss, even if machines fail or messages are dropped. The at-least-once semantics guaranteed by storm work by retro-propagation of acknowledges in the topology. A bolt will only acknowledge a tuple, when all child bolts have.

Figure 5.12 shows component view of the Stream Processing container.

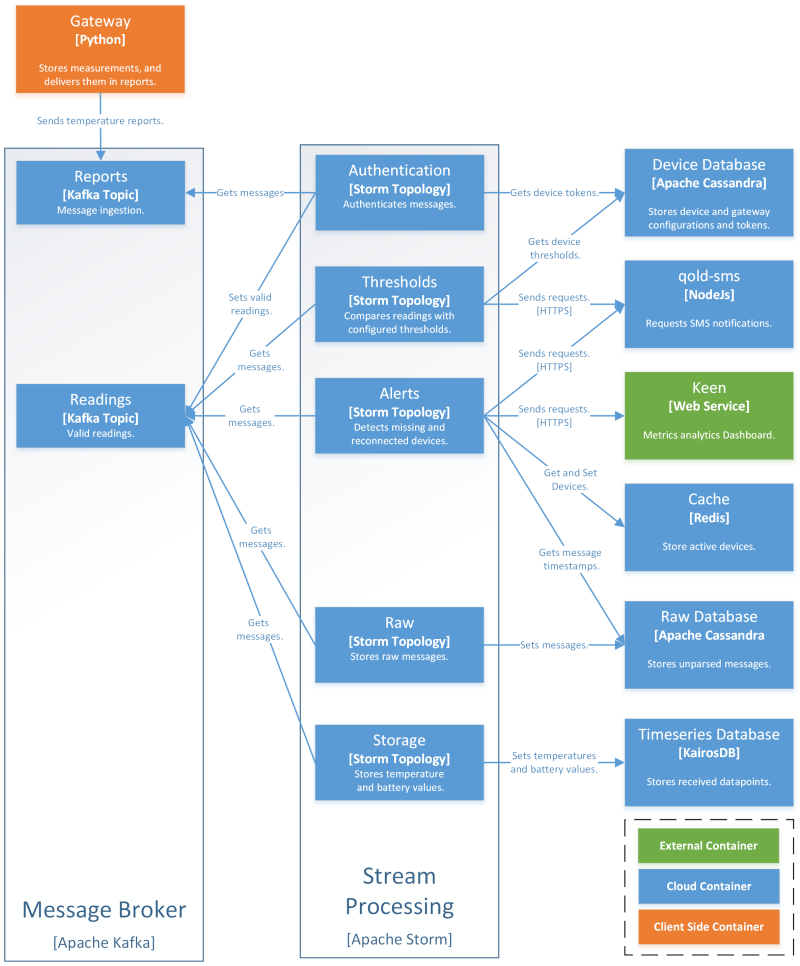

Figure 5.12: System Architecture: Stream Processing

5.4.1.5 Coordination

Both Apache Storm and Apache Kafka require an Apache Zookeeper[67] cluster. Apache Kafka uses Zookeeper for storing Access Control Lists (ACLs), hold partition leader election, storing and synchronizing configurations, and managing cluster memberships. Apache Storm uses Zookeeper for worker membership and election, and for persisting state.

5.4.1.6 Cache

Apache Storm uses at-least-once semantics when processing messages, and if any failure occurs it will restart consuming any unacknowledged messages. The variable state withing the topologies on the other hand is not kept between restarts; an external cache solution is necessary to make sure state is not lost. The cache container is also used to keep state inside topologies, since they are Directed Acyclic Graphs and therefore cannot communicate in the reverse direction. The common choice for this container is Redis[68], an in-memory distributed key-value store, as it is easy to setup, has no management overhead and is extremely performant.

5.4.1.7 Time-series Database

The datapoints received from the devices are stored in the 'Time-series Database', essentially replacing Aqora from the legacy system.

The key difference between time-series data and regular data is the need to always be tracked, monitored, down-sampled and aggregated over time. Data life-cycle management, summarization and large range scans of many records are what separate time-series from other database use cases. With time-series it's common to request a summary of a larger period of time, which requires going over a large range of data points to perform some computation or aggregation. This kind of workload is very difficult to optimize for a distributed key value store.

With the growing IoT and DevOps trends, many time-series databases are being developed, either to deal with the growing amounts of data or simplifying the monitoring of system metrics. From all the research conducted during the internship this was the topic with more options; unfortunately none of them stand out. While there are numerous time-series databases, they either require an enormous operational overhead, are designed to handle a very specific use case, or do not scale. The most relevant options were InfluxDB[64], OpenTSDB[63] and KairosDB[**?**].

InfluxDB is an open source database written in Go designed to handle time series data with high availability and performance requirements. InfluxDB is probably the most trendy time-series database at the time of writing, due to its easy installation and management. Ultimately InfluxDB was not chosen due to its scalability: at the beginning of the internship it could not scale out past three nodes, and eventually scaling became a paid feature.

OpenTSDB consists of a Time Series Daemon (TSD), meaning that it is essentially a layer, built on top of a datastore. OpenTSDB nodes are stateless and independent of each other, so it can scale linearly, and all data is persisted to the HBase[69] datastore. Each TSD uses the open source database HBase to store and retrieve time-series data, and users never need to access HBase directly. The HBase schema is highly optimized for fast aggregations of similar time series to minimize storage space. The main issue with OpenTSBD is HBase, which is not trivial to install or manage since it needs an HDFS file system.

Due to the operational overhead of HBase, as well as design differences, some OpenTSDB developers decided to create a new time-series database using Cassandra has the datastore, and created KairosDB. KairosDB is also a TSD, with the same advantages and scaling characteristics as OpenTSDB but Cassandra is much simpler to manage than HBase, and does not require HDFS. Essentially, each KairosDB node is stateless and independent, and use a Cassandra namespace for persistence. Since Cassandra is linearly scalable at least until a few million writes per second[48], KairosDB is also linearly scalable due to its independent architecture. For these reasons, kairosDB was chosen as the Time-series database for the system.

Each time-series, which are called metrics in KairosDB consists of a value, with an associated timestamps, and a list of custom tags. The Qold system is currently using two metrics, one for the battery of the devices, and one for the temperature values:

Metric	Value	Timestamp	Tags
Temperature	temperature (double)	timestamp (long)	device_id (int)
Battery	battery (int)	timestamp (long)	device_id (int)

Table 5.2: Time-series metrics

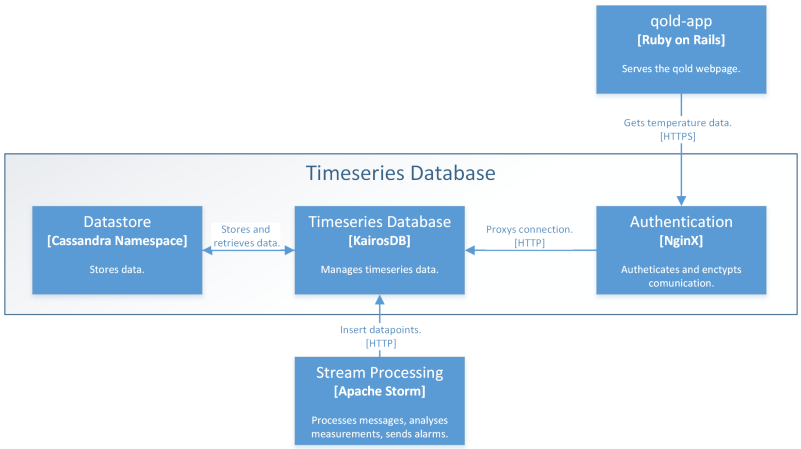

Figure 5.13: System Architecture: Time-Series

While the Time-series database does not need encryption or authentication for intra-cluster communication, as it is handled at network level, it does need it for communications from outside the cluster, such as from the qold-app. KairosDB lacked these required features, and as such an encryption and authentication server was added to the container, in the form of an nginx proxy. This service handles HTTPS communications from outside the cluster, performs X.509 certificate authentication and proxys valid traffic to KairosDB using HTTP. The qold-app currently holds the only valid certificate.

5.4.1.8 Raw Database

The purpose of the 'Raw Database' is to store all the unparsed authenticated messages received from the gateways as they arrive. The immediate purpose for these messages is re-playability and debugging in cause of failures, but they can also be used latter on the life-cycle of the project if the need for a 'Cold path' ever arises; a batch processor can use these raw messages to perform machine learning, for example. As such, the main requirements for the 'Raw Database' were Availability and Partition tolerance from the CAP theorem, and the ease of management and deployment. Considering these requirements, the obvious choice was to use Apache Cassandra; not only does it fit the requirements, it also has easy integration with batch processors and it was already being used for the time-series data. Using a different name-space from the time-series data ensures that while living in the same machines, both databases would be individually deployable and configurable.

The 'Raw Database' has only one table, with the json message, the device id and the insertion time as a unique timestamp. The device id and the insertion time are a composed primary key, and the insertion time is also a clustering key. This means that an hashing function will make sure to distribute the devices across the cluster, but at the same time all the messages from each device are stored on the same Cassandra node; this is important, because Cassandra is able to keep a sorted list of messages by the clustering

key, in this case the insertion time. In practice, this means that while it is impossible to get a sorted list of messages, it is possible to get a sorted list of messages per device.

Raw		
device_id	INTEGER	PK
insertion_time	TIMEUUID	PK,CK
message	TEXT	

Table 5.3: Raw table

5.4.1.9 Admin Database

The 'Admin Database' is used to store the configurations of devices, gateways and administrators, it does not however keep any information or relation to users. This database does not need high performance or scalability, as it wont hold much data, or be subjected to many queries. As such the most important requirements were ease of management; because of this the choice fell once again on Cassandra, as it was already setup and would introduce very little complexity to the system.

The 'Admin Database' has four tables, two for storing device and gateway configurations, one for creating counters and one for storing admin credentials. The configurations are stored as a 'String' to 'String' hashmap, and can be queried directly; the authentication tokens are hashed using bcrypt; the device state is used to toggle active or inactive devices that should be ignored.

Admins		
username	TEXT	PK
password	TEXT	
permission	TEXT	

Table 5.4: Admin table

Devices		
id	INTEGER	PK
auth	TEXT	
state	BOOLEAN	
config	map<TEXT, TEXT>	

Table 5.5: Device table

Gateways		
id	INTEGER	PK
auth	TEXT	
config	map<TEXT, TEXT>	

Table 5.6: Device table

Resource_counters		
resource	TEXT	PK
next_id	INTEGER	

Table 5.7: Counters table

5.4.1.10 Qold API

In order for Administrators to interact with the 'Admin Database', a JSON Rest API was developed. Administrators login in the Qold API by sending credentials that are matched with the data on the Admin table and receive a JWT[49] which is valid for an hour; other API requests require a valid JWT.

This API is also used during the provisioning of gateways. Administrators login to the API and supply the gateway provisioning playbook a JWT, which will be used by each gateway to register themselves and get an id and token.

6

Operations

In order to reduce the operational complexity of managing the system and respond to the requirements of the project, the best practices of building a state of the art container production stack were researched and followed. This chapter describes the technologies used when building the cluster and how they fit together in the overall stack.

6.1 Production Stack

The system stack was designed to match the following qualities:

- **Self healing and self managing** If a machine or application fails, the system should attempt to bounce-back without intervention.
- **Support microservices** Each service should be able to be easily scaled and managed by different teams.
- **Efficient** The stack should not require much intervention, and should not consume too much resources.
- **Debuggable** A complex system can be hard to debug, it is important to have good monitoring and logging strategies.

The final stack is based on a scheduling and orchestration tool called Kubernetes:

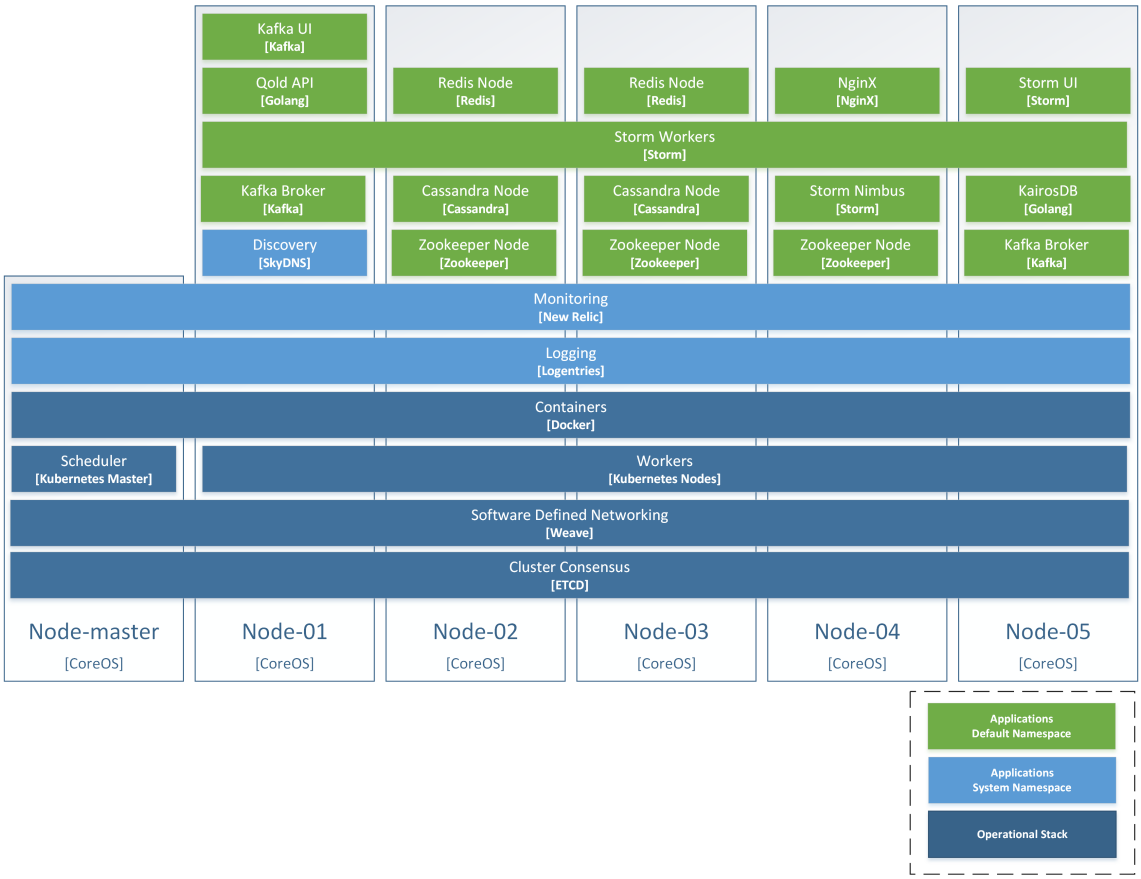

Figure 6.1: Qold Production Stack

6.1.0.1 Operative System

The OS layer provides the lowest level of execution environment. Traditionally this is where applications execute when individual machines are provisioned with different software stacks. Over the lifetime of servers and software, however, the evolution of individual machine configurations start to become unwieldy. Containers on the other hand treat the entire OS as one application package that can be managed as an independent unit, and because of that, they can be higher on the stack. In the container stack, the OS of the machines is smaller and less important, it needs just enough to get a container engine up and running. The chosen OS for the system was CoreOS.

CoreOS is a Linux distribution built to make large, scalable deployments on varied infrastructure simple to manage; it maintains a lightweight host system and uses Docker containers for all applications. The main host system is relatively simple and dismisses many of the common components of traditional distributions, such as a package manager. Instead, all additional applications are expected to run as Docker containers, allowing for isolation, portability, and external management of the services. At boot, CoreOS reads a user-supplied configuration file to do some initial configuration, and connect with other members of a cluster, start up essential services, and reconfigure important parameters. One interesting aspect of CoreOS is that it keeps itself updated; while distributions like Ubuntu have yearly releases, that need manual update, CoreOS has almost weekly releases. CoreOS has three release channels that can be chosen: 'Alpha', 'Beta' and 'Stable'. Whenever an update is released for the selected channel, all the CoreOS machines in the cluster download and install the update, and reboot. In order to avoid loss of availability,

the reboot is coordinated to make sure only one machine reboots at a time. Updates will also not break the applications, since they are running inside containers.

6.1.0.2 Bootstrapping System

After the machines are created and CoreOS is installed, they need to be configured, coordinated and provisioned with the rest of the stack. This task is handled using Ansible. Which is an open source automation solution that distinguishes itself for being very simple to use. Ansible is used by creating playbooks which are essentially a list of tasks in yml format. This tasks run in sequence and create a chain of events in all the hosts defined on an inventory file. Unlike other similar tools, Ansible does not use an agent on the remote host, or a centralized server. Instead Ansible uses SSH to connect to the hosts. The tasks created in the playbook use Ansible modules which are written in Python to execute on the remote hosts, these tasks also have the advantage of being idempotent. Ansible playbooks look like the following snippet:

```
- name: SSL | Create directories for ssl certificates
  file: path=/home/core/ssl/{{ item }} state=directory mode=0755
  with_items:
    - ca
    - minion

- name: SSL | Copy CA certificates
  copy: src={{ item }} dest=/home/core/ssl/ca
  with_items:
    - ca.pem
    - ca-key.pem
```

Ansible is being used to install and configure the CoreOS machines with the rest of the operational stack.

6.1.0.3 Cluster Consensus

In order to mitigate the complexity of managing individual servers, they must be abstracted into a cluster, in other words a collection of resources. However, with the addition of more machines, there are also more points of failure, and the need for primitives found in multiprocessor programming, such as equivalent of locks, message passing, shared memory and atomicity across this group of machines. The ETCD daemon is a highly available key-value store that solves this need by storing and distributing data to each of the hosts of the cluster. Applications can retrieve information from the store by connecting to the local client interface on their local machine. All ETCD data is be available on each node, regardless of where it is actually stored. Leader elections are also handled automatically. In the Qold system it keeps consistent configurations for all the other elements of the operational stack, like the addresses of the members of the cluster or the overlay networks.

6.1.0.4 Network Virtualization

Traditionally, Docker containers can only connect to other containers on the same host. Weave solves this issue by creating a virtual network that connects the containers across multiple hosts and enables their automatic discovery. Applications use the network just as if the containers were all plugged into the same network switch, without having to

configure port mappings or links. In practice, weave encapsulates the packets sent to containers in different hosts while they travel between machines. In the Qold system, Weave also provides encryption of data while it travels between hosts using the NaCl crypto libraries.

6.1.0.5 Kubernetes

Kubernetes is an open source platform created by Google, for managing containerized applications in a clustered environment. It attempts to create a layer of abstraction over the infrastructure that allows for various levels of control over applications in containers.

The controlling unit in a Kubernetes cluster is called the master server, and serves as the main management contact point for administrators. It also provides cluster-wide management of the worker nodes. The master server runs services that are used to manage the cluster's workload and direct communications. The API service is one of the main management point of the cluster, it allows users to deploy and configure workloads and organizational units. The controller manager service is used to handle the replication processes defined by replication tasks; the details of these operations are written to etcd, where the controller manager watches for changes and implements the replication procedures that fulfill the desired state. The scheduler service assigns workloads to specific nodes in the cluster; it reads the service's operating requirements, analyze the current infrastructure environment, and places the work on an acceptable node or nodes. The scheduler is also responsible for tracking resource utilization on each host.

The worker nodes contact with the cluster group through a service called kubelet. This service is responsible for relaying information to and from the master server, as well as interacting with the etcd store to read configuration details or write new values. The kubelet service receives commands and workloads from the master and then assumes responsibility for maintaining the state of the work on the minion server.

To manage and connect to Kubernetes, a tool called 'kubectl' is used. This tool provides encryption and authentication, and behaves as if the cluster was running locally.

While applications are always deployed as Docker containers, the workloads that define each type of work are specific to Kubernetes. A pod is the basic unit that Kubernetes deals with, and it represents one or more containers that should be controlled as a single application; this association leads all of the involved containers to be scheduled on the same host and share volumes and IP space. A more complex workload is a replication controller, which is a framework for defining pods that are meant to be horizontally scaled. A replication controller provides a template which is a complete pod definition and the number of desired replicas. While pods are deleted if they are killed or crash, the cluster will always maintain the desired number of replication controller pods, even if the applications in the container crash or the machine fails.

Besides automatically handling the containers on the cluster, Kubernetes also simplifies managing the applications running in them, providing many useful functions, such as: load-balancing connections, rolling-updates and auto-scaling applications, and handling secrets.

Kubernetes workloads are created as yml or json files which are fed to 'kubectl', in the format:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
```

```
replicas: 2
template:
  metadata:
    labels:
      name: nginx
  spec:
    containers:
    - name: nginx
      image: nginx:latest
```

6.1.0.6 Logging

The logentries service provides centralized logging to the cluster. Essentially, a logentries Docker container is deployed to every host of the cluster, which connects to the Docker API and retrieves the stdout logs from every container. These logs are sent to the logentries web service, where they can be queried by Administrators and Developers.

6.1.0.7 Monitoring

The New Relic service provides centralized monitoring to the cluster. A New Relic agent Docker container is deployed to every host of the cluster, and is constantly retrieving system metrics and sending them to the New Relic web service which can be used to inspect the load of the system and configured to send alerts.

6.1.0.8 Discovery

The SkyDNS is an addon that integrates with the Kubernetes API and provides automatic container discovery to applications.

7

Implementation

This chapter introduces the development decisions and implementation details of some system components.

7.1 Qold API

The Qold API was developed in order to provide a secure and easy way of managing the 'Admin Database'. The API was implemented in Golang an open source programming language developed at Google and released in 2009.

The API was developed in a modular fashion, with the networking separated into three component types: routers, controllers and services. The routers map the endpoints and HTTP methods to each controller. The controllers are responsible for getting and setting the body and headers of each request and response, and for calling services. The services are responsible for executing the business logic.

In order to use the API, an Administrator needs to login to the system using HTTP basic authentication and a JWT, with the expiration time of one hour, will be supplied in return; this token must be used for all subsequent requests. A JSON Web Token (JWT) is a compact way of representing claims to be transferred between two parties. It consists of three main components: an header object, a claims object, and a signature. These three properties are encoded using base64, then concatenated with periods as separators. The header contains a JSON structure identifying the type, which is 'JWT' and the hashing algorithm to use. The claims object contains the issuer of the claim, the issued-at time, the expiration time and the subject of the token. Finally, the signature is made up of the hashing of the header, the payload and a secret that is held by the server. JWTs provide a way for clients to authenticate every request without having to maintain a session or repeatedly pass login credentials to the server. The API also provides encryption, in the form of HTTPS.

The API has the following endpoints, which are further detailed in the Appendix D of the report:

Endpoint	Method	Description
/api/v1/admin/token	POST	Create Token
/api/v1/admin/token	GET	Check Token Validity
/api/v1/admin/users	POST	Create User
/api/v1/admin/users/{user}	GET	Check User
/api/v1/admin/users/{user}	PUT	Update User
/api/v1/admin/users/{user}	DELETE	Delete User
/api/v1/admin/devices	POST	Create Device
/api/v1/admin/devices	GET	List Devices
/api/v1/admin/devices/{device}	GET	Get Device
/api/v1/admin/devices/{device}	DELETE	Delete Device
/api/v1/admin/devices/{device}/config	PUT	Config Device
/api/v1/admin/devices/{device}/login	POST	Login Device
/api/v1/admin/devices	PUT	Create Device using legacy API
/api/v1/admin/gateways	POST	Create Gateway
/api/v1/admin/gateways	GET	List Gateways
/api/v1/admin/gateways/{gateway}	GET	Get Gateway
/api/v1/admin/gateways/{gateway}	DELETE	Delete Gateway
/api/v1/admin/gateways/{gateway}/config	PUT	Config Gateway
/api/v1/admin/gateways/{gateway}/login	POST	Login Gateway

Table 7.1: API Endpoint

7.2 Gateway

The gateway implementation can be separated into three sections: the updates, the provisioning and the software.

7.2.1 Update System

The update mechanism used for the Qold gateways is inspired on the Resin[51] service. To update the software of a gateway, a Developer needs to build the Docker image for that software container and upload it to the DockerHub, with a bump in version number. For example, currently the image for the gateway 'input' container is 'whitesmith/gateway:input-1.5'. After the image has been pushed to the DockerHub, the 'gateway-update' git repository needs to be updated to match the new desired state. This repository holds a single yml file, called 'local.yml', which is an Ansible playbook with the following format:

```
---
- hosts: localhost
  user: user
  tasks:
  - name: qold_input container
    docker:
      name: input
      image: whitesmith/gateway:input-1.5
      state: reloaded
      restart_policy: always
      command: /root/firmware/qold qold_input
      volumes:
```

```
    - /user/data/:/root/data/
  - name: qold_output container
    docker:
      name: output
      image: whitesmith/gateway:output-1.5
      state: reloaded
      restart_policy: always
      command: /root/firmware/qold qold_output
      volumes:
      - /user/data/:/root/data/
```

A cronjob makes sure that every day, and at every reboot, the gateway runs an Ansible-pull on the git repository. The Ansible-pull will fetch any changes to the playbook and execute it. If the image is not found locally, the gateways will pull them from the DockerHub. The 'state: reloaded' variable will make sure that the running containers are only restarted if they need to be updated. The 'restart_policy: always' will tell the docker-engine that the container should always be running, as such, even if the container crashes, Docker will restart it.

Because docker compresses the images in the DockerHub, and only pulls the image layers it doesn't already have, the bandwidth cost of this update method is very small.

7.2.2 Provisioning

Initially an Ansible playbook was developed with the basic configuration, files and packages common to all gateways. This playbook was run against a gateway, and its SD card was cloned as an image file. A second Ansible playbook was then developed witch provides unique configuration data for each gateway; this playbook relies on a administrator provided JWT for accesing the Qold API. When provisioning new gateways, the Ansible playbook will make them request new credentials from the Qold API. The provisioning of new gateways consists then, in cloning the image file to the SD card, and running the Ansible playbook to provide unique identification.

7.2.3 Gateway Software

As described in the Architecture Chapter, the gateway software was separated into two components, one responsible for receiving data from the devices and another for publishing messages to the cloud. Both components are written in Python, are single threaded and synchronous. As this software is running in deployed gateways which are not easily accessible, its simplicity was essential, in order to reduce bugs.

The code for receiving and parsing device messages was left mainly intact, as it had been working for a long time, and changes could easily introduce bugs. But during the development of the gateway software, the team decided to add at-least-once semantics and Cyclic Redundancy Check (CRC) to the connection, in order to avoid data loss due to corruption or loss of signal. Other team members implemented the changes to the devices, and the intern updated the gateway software. The CRC was implemented and the gateways would now acknowledge the device communications. The messages received from the devices are parsed and a JSON message is built with the format: `{"device":{"id":123,"auth":789},"reading":{"temp":15,"time":123,"batt":75}}`. This message is stored in the sqlite database, using an Object-Relational Mapping (ORM) model. The database only has one table, with the following model:

Messages		
id	INTEGER	PK
timestamp	INTEGER	
message	STRING	
topic	STRING	

Table 7.2: Messages table

The 'output' container will poll this table every few seconds and send any messages it finds. This container initially used MQTT to communicate with RabbitMQ, but was later changed to use the Kafka driver. This container sends messages to the topic provided in the database row, essentially allowing for different containers to create messages that should be delivered to different topics. It also uses at-least-once semantics, only deleting the message from the database once it has been acknowledge by Kafka.

The communication with the Message Broker is encrypted using TLS and authenticated as the user 'gateway'; essentially, every gateway has the same certificate. The Kafka deployment has two opened ports, one using Plaintext which is only available inside the cluster and another with TLS enabled which is public. Kafka expects all connections on the TSL port to be authenticated through X.509 certificates or they are denied access, but connections to the Plaintext port are assigned the 'ANONYMOUS' user by default. Kafka is configured to give full access to the 'ANONYMOUS' user, but only 'Write' permissions to the 'gateway' user. This way, the gateways can only push messages to the Broker, while any access within the cluster can read, write or create any topic.

7.3 Stream Processing

This section describes all the topologies built for the Qold system.

7.3.1 Raw Topology

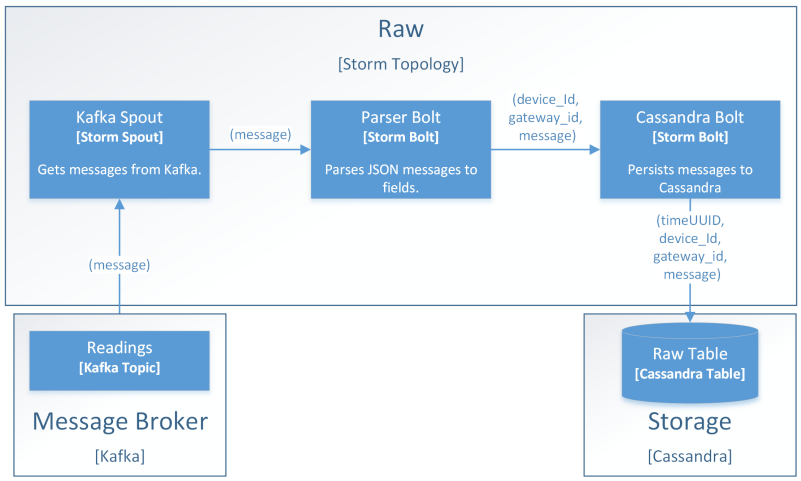

Figure 7.1: Storm Topologies: Raw

The Raw is the simpler storm topology, it basically receives a stream of reading from Kafka, reads those JSON messages to storm fields and emits them to the official Storm-

Cassandra bolt which will insert them to the database.

7.3.2 Authentication Topology

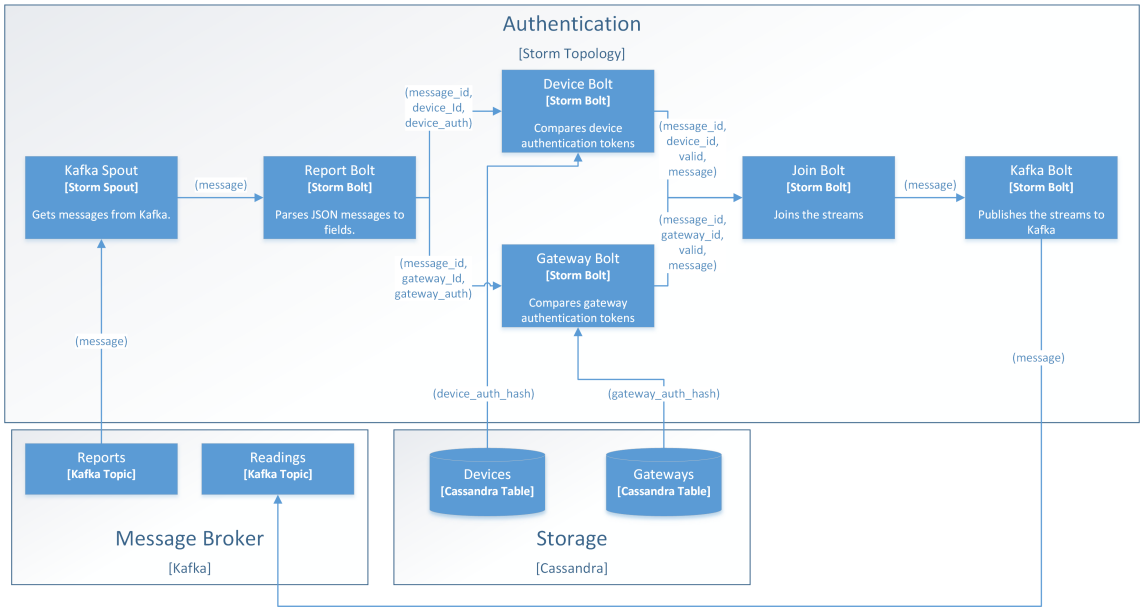

Figure 7.2: Storm Topologies: Authentication

The Authentication topology receives a stream of reports sent by gateways to Kafka and starts by reading those JSON messages to storm fields. Those fields are emited as two streams, one with the device credentials, and one with the gateway credentials; each stream also has the message id.

The ‘Device Bolt’ and the ‘Gateway Bolt’ use fields grouping by ‘device id’, meaning that all messages from the same device will be handed to the same ‘Device Bolt’ worker. This enables the topology to only querying the database for the device/gateway credentials once, whenever an unknown device/gateway report is received, and keep a cache of device credentials in an HashMap; messages from the same device/gateway will simply be compared against the local cache, reducing external calls to the database.

The ‘Device Bolt’ and ‘Gateway Bolt’ will authenticate the credentials in parallel, and a ‘Join Bolt’ will aggregate both streams and check if both authentication processes were successful. Valid messages are emited to the official Storm-Kafka producer bolt and published to the ‘Readings’ topic to be consumed by other topologies.

7.3.3 Datapoints Topology

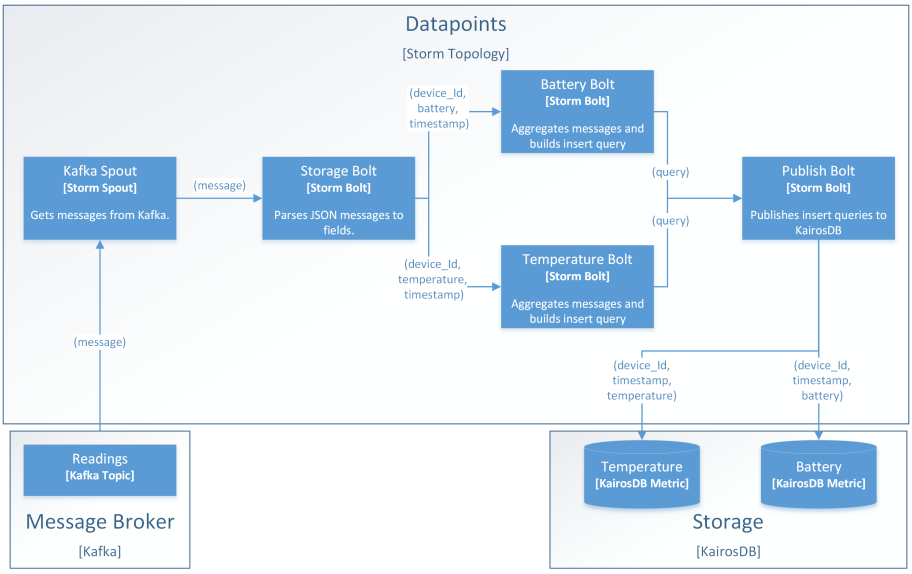

Figure 7.3: Storm Topologies: Datapoints

The Datapoints topology receives a stream of reading from Kafka, reads those JSON messages to storm fields and emits them as two streams, one for temperature and another for battery values. This topology makes use of a new feature in Storm 1.0, windowing primitives: the 'Temperature' and 'Battery' bolts receive readings in batches of 3 seconds. The insert query for KairosDB is built on these bolts using all the readings in the batch, and are persisted to KairosDB through the Publish Bolt. Because readings are persisted in batches, it significantly reduces the impact on KairosDB. Also worth noting that readings are acknowledged to Kafka in batches after being persisted to KairosDB.

7.3.4 Thresholds Topology

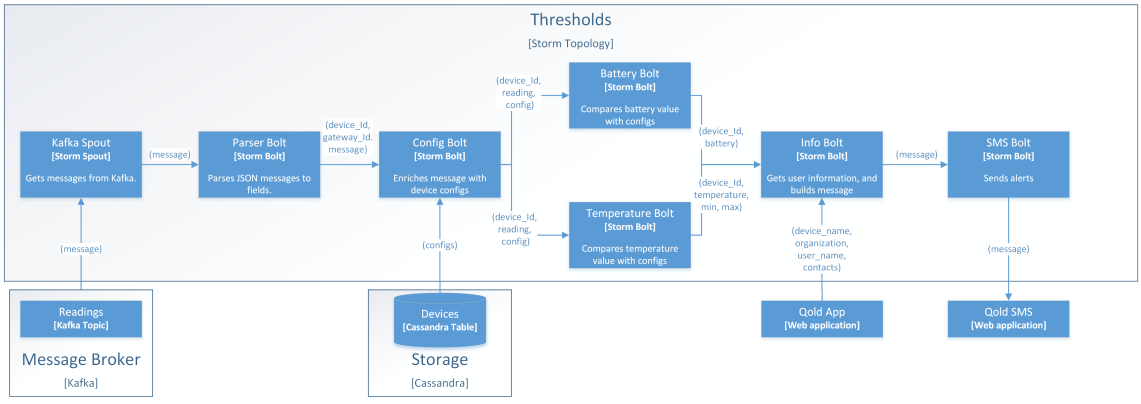

Figure 7.4: Storm Topologies: Thresholds

The Thresholds topology receives a stream of readings from Kafka, reads those JSON messages to storm fields and emits them to 'Config Bolt' using fields grouping by 'device id'. Every time it receives a message from an unknown device, the 'Config Bolt' gets its

configurations from Cassandra and stores them in an HashMap; messages from known devices will simply use the local cache, instead of a database query. The bolt will also update the cached configurations every 10 minutes, for every known device. This bolt will emit two streams, one for temperature and one for battery values, using fields grouping by 'device id'.

The 'Temperature Bolt' and the 'Battery Bolt' compare the temperature and battery values respectively with the supplied configurations, and decide if the user should be alerted. These bolts also keep an HashMap with the time of the last alert sent for each device, and will only trigger new alerts, once every hour per device. If an alert is indeed necessary, the bolts will emit a message which will be received by the 'Info Bolt'. This bolt will enrich the alert with device information gathered from the Qold App, such as user phone number, device name and owner, and emit the built message. The message will be received by the 'SMS Bolt' which will make a request to the Qold SMS service.

7.3.5 Alert Topology

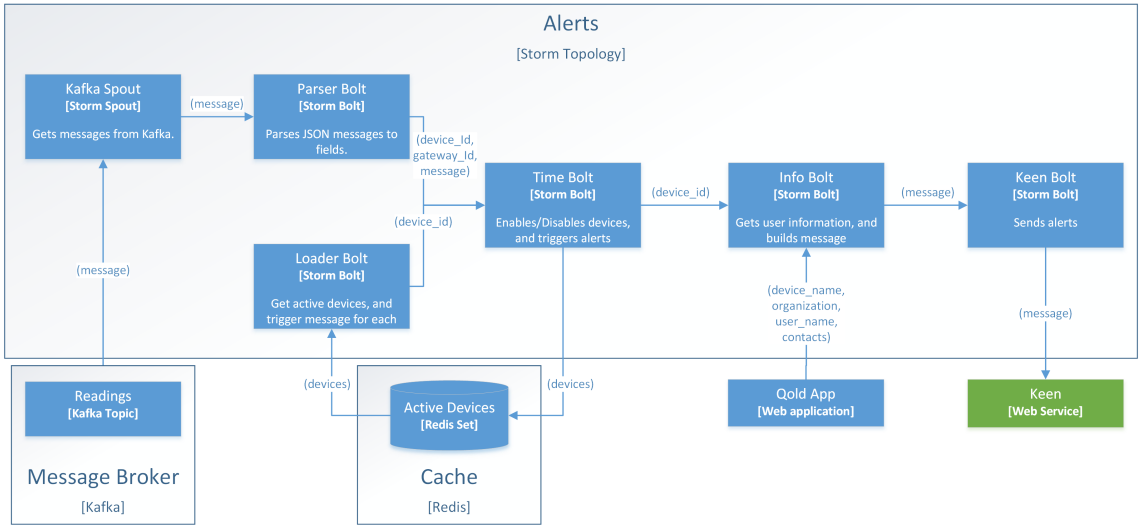

Figure 7.5: Storm Topologies: Alert

The Alert topology has essentially two flows.

The first flow occurs when a reading is received from Kafka. This JSON reading is transformed to storm fields and received by the 'Time Bolt' using fields grouping by 'device id'. The 'Time Bolt' maintains an HashMap with the time of the last reading received for each device. If a reading is received for a device that last communicated more than an hour ago, the bolt emits a message and adds the device id to the Redis cache.

The second flow is triggered automatically every 15 seconds on the 'Loader Bolt'. Once the bolt is triggered, it will read a list of devices from the Redis cache and emit a message for each. The message is received by the 'Time Bolt', which is able to distinguish these messages from the ones received from the 'Parser Bolt'. The 'Time Bolt' will check the HashMap for the time of the last reading of each device, and emit a message if it finds one whose last reading was more than an hour before.

All messages are received by the 'Info Bolt'. This bolt will enrich the alert with device information gathered from the Qold App and emit the built message. The message will be received by the 'Keen Bolt' which will make a request to the Keen API.

7.3.6 Legacy Topology

Finally one last topology was developed, the Legacy topology. This topology only has one Kafka spout and two bolts, and was developed in Python to serve as a reference when building topologies in other programming languages besides Java. The topology has one 'Parser Bolt' that receives authenticated readings from the Kafka spout, transforms the received readings into messages in the legacy format and emits them. A final 'Legacy Bolt' publishes these messages to the legacy qold-hw-api. This topology serves as the integration point between the two systems.

8

Verification and Validation

The verification and validation phase, is essential to evaluate the success of the project. The verification consists on making sure the project was properly developed, mainly through testing. Validation on the other hand is used to make sure the right project was built, and that it achieves the goals it aimed for.

8.1 Verification

As mentioned, the process adopted for the internship envisioned the use of test driven development. This technique was used with different degrees of success, it worked really well during the API development and in some components of the gateway. As the stream processing topology development used a very specific framework and the programmer does not have full control over the environment they are running, this proved more challenging. Despite that, all system components were validated through testing.

Before delving into the verification of each component, it is interesting to notice that the system essentially has three execution environments. The production cluster, is running on a cloud provider and handling client data, as such it was not used for testing. The testing environment is a mirror of the production environment and is only deployed when needed; essentially thanks to Ansible and Kubernetes, going from creating virtual machines on a cloud provider to having the whole system deployed and setups takes only a few minutes, and this setup was automated in order to speed up testing. Since all system components are running on docker container, it is also possible to simulate the entire system on the developers computer, using Docker Compose to build the local 'cluster'.

8.1.1 Gateway

In order to facilitate testing and development, the gateway component was simulated using Docker containers. This meant that any number of fully functional gateways could be simulated immediately, either locally of to the cloud, but the devices could not however. In order to simulate the messages received from devices, the communication part of the source code was replaced by dummy messages for testing. These tests were built using the binary packets received from the devices, and are used to simulate the entire operation of a gateway except for the actual receiving of device messages. These dummy messages provided two test modes: a sequence deterministic messages that could be used to verify the system; and a never ending stream of randomly generated messages that could be used to evaluate throughput and simulate a real-world workload.

The deterministic messages were used not only to test the gateway software, but also the rest of the system. Essentially, gateways were simulated, and the logs of each system component would be automatically checked after a few seconds and compared to the oracle. This mode was therefore essential to provide end-to-end testing to the entire

system. To benchmark other system components, the second workload was also used to generate data.

The integration with the 'Message Broker' was also tested, essentially a set of scripts were developed that simulated gateways with missing or incorrect certificates, stopped the 'Message Broker' or the gateway execution to simulate loss of connectivity and ensure at-least-once semantics.

8.1.2 Qold API

Before starting development on the Qold API, the endpoints, their functionality and messages were decided and tests were built. The tests were initially developed using 'curl', but were latter changed to the 'Go test' framework from the Go language. These tests made API requests with both valid and invalid data, and compared the responses to the oracle. Besides functionality, the tests also covered authentication, authorization, and loss of connection to the database in order to evaluate the security and dependability of the API.

8.1.3 Kafka

Apache Kafka was tested in order to make sure the authentication and authorization were working properly, and also to evaluate the effect of failures in the system. As mentioned, gateways with invalid of missing certificates were simulated to make sure they could not read or access other topics. The brokers were also independently and simultaneously killed in order to assess data loss and effect on communication with producers and consumers.

8.1.4 Casandra and KairosDB

These databases were tested by supplying various messages and comparing the stored data with the oracle.

8.1.5 Storm

The Kafka integration was tested using the the simulated gateways and their deterministic data, to test the topologies themselves it was faster and easier to replace the Kafka spout with a dummy that supplied deterministic messages. The effect of these messages on the topology was automatically compared to the oracle. Unlike the rest of the system, Storm has an additional development environment; while it can be tested locally by simulating a Storm cluster using Docker Compose, it can also run straight from the IDE. This local mode of execution speeds up development and testing, since it does not require packaging or deploying. In order to test the topologies, bolts with test logic were injected in the topologies, and would make sure the received message was the expected. Integration with external systems on the other hand, had to be tested manually, has there was no way of simulating the request. In order to test semantics and data loss, some test bolts would purposely fail to ensure the messages were replayed. The coupling and reliability of the topologies were also tested by shutting down other system components, if a message fails it will be replayed, and if a topology fails, it will be automatically restarted by the Nimbus daemon.

8.1.6 System

As explained, the system was also tested using the deterministic messages from the gateways, and an automated script would gather the logs of each component and query the databases, to make sure all was working as intended.

8.2 Validation

8.2.1 Functional Requirements

The functional requirements of the system were validated through meetings with the Qold team and the supervisor at Whitesmith, and empirically through the deployment of the system. From these meetings and the normal functioning of the system, it was concluded that all the 'Must Have' and 'Should have' requirements were achieved, and as expected the 'Won't have' were not; the 'Could have' requirements were also not achieved as there is no way to remotely access the gateways, that goal was removed during the change of scope in the beginning of the second semester.

8.2.2 Quality Attributes and ACLs

The validation of quality attributes is mostly handled through empirical observation, and internal and external benchmarks. Due to the overlap between the Quality Attributes and ACLs they were validated simultaneously.

8.2.2.1 Performance and Scalability

Performance and scalability benchmarks were made and are further detailed in the Appendix E of this report.

Apache Kafka benchmarks showed that the current deployment is capable of ingesting 1000s of messages a second, well beyond the performance requirements, and can scale to ingest much more. These results were not surprising, since according to official[73] and external[74] benchmarks it is capable of ingesting up to a few million messages a second with only commodity hardware. It is also interesting to note that Kafka has been used at LinkedIn since 2011, where it was initially developed, and their deployment handles of a trillion messages a day.

The performance evaluation on Apache Storm also showed interesting results. The current deployment, which although distributed has minimum parallelism, is capable of handling a few 1000s of messages a second in some topologies, and latencies inferior to 1 second. All topologies are able to handle the load predicted for the following years. It is important to note that the current deployment has the minimum resources for a Storm cluster; increasing memory, cpu, bolt parallelism, workers or nodes would severely increase the throughput of the topologies. Although the minimum requirements are more than enough to handle the predicted load, it is also a lot more demanding than the legacy system in terms of resources. The overhead of scalability and parallelism of the system are noticeable in smaller deployment. Storm scalability can also be validated through external benchmarks[76].

Finally, KairosDB was also tested, and presented acceptable results for the Qold use case, querying a 1000 datapoints in less than a second. As explained in the architectural chapter, both Cassandra and KairosDB are linearly scalable[77] due to their architecture.

8.2.2.2 Resource Requirements

In order to achieve the resource requirements for the system, the gateways should not exceed 50 Megabytes of communication bandwidth per month. The cost of a Qold reading using Kafka averages in 6 Kbytes. Each device sends a reading every 15 minutes, bringing the total to an average of 16.75 Mbytes per month. By reducing the producer heartbeat to once every 5 minutes, which weights 1 Kbyte, the monthly total averages on 25 Mbytes. This value is half of the requirements and can be brought even lower if needed by batching messages, at the cost of latency.

8.2.2.3 Security Requirements

The intra-system communication is encrypted through the Weave network virtualization. The gateway communication is uses encryption through TLS and authentication through a X.509 certificate and a unique secret token. The communication between the web application and the time-series database also uses TLS and X.509 client certificates by proxying the connection through Nginx. Access to the cluster requires the use of either SSH to the machines, or the 'kubectl' application which also uses TLS and client certificates for encryption and authentication. The access to the Qold API requires administrator credentials, is encrypted using TLS and reduces credentials exposition by using JWTs for session management.

8.2.2.4 Maintainability Requirements

The update and deployment mechanisms for the gateways were rewritten; after an update was pushed, all gateways will get it within a day, whether it is an update to existing components or new containers. The containerization of the gateway also provides isolation and decoupling of components, allowing them to be swapped without interfering with the rest of the software.

As topologies are independent of each other and communicate asynchronously through Kafka, any number of alerts can be created and deployed without interfering with each other.

During the final weeks of the internship, the intern working on the hardware and business of the Qold product developed a new device and communication protocol. The new protocol lacked Python drivers, and so the 'input' container for the gateways was developed by him in C++, and deployed to gateways. The update did not require access to the gateways, nor any changes to the other containers. The container was built and deployed by him within one day, empirically validating the Maintainability ACL. The change of backend system form RabbitMQ to Kafka, also required minimal changes to the 'output' container and was updated OTA to all the deployed gateways.

8.2.2.5 Manageability Requirements

The Logentries, New Relic and Keen services handle most of manageability requirements of the system. Using these services, any Administrator can access the system logs and metrics through the Internet. Keen helps Administrators understand the state of the system, Logentries allows them to check the logs of every application, and New Relic displays operational metrics and sends alarms when the system resources are lacking. Through the new gateway playbooks and the Qold API, a gateway can be deployed withing 15 minutes.

8.2.2.6 Reliability Requirements

All the system applications either are or were configured to be 'fail-fast', meaning that any error or exception will make them exit or crash. Every time a container stops, regardless of the reason, Kubernetes will bring it back up, even if it needs to change the node the container is running in. Kubernetes makes sure that all stopped containers are restarted withing 5 minutes, bringing the container to its original state.

8.2.2.7 Dependability Requirements

95% is the accepted data loss per day per client, because it ensure that, if the data is sparsely lost it will not affect the use case of the product, and if those messages are lost at in succession, an alarm will trigger due to loss of communication and the team can respond to it. In order to reduce data loss, at-least-once semantics were used. The 95% value was validated empirically through the use of the system in production.

9
Conclusion

9.1 The Internship

This report documents the path taken by the intern during the design and implementation of a scalable near real-time system for the IoT. The project proved to be extremely interesting, challenging and educational for several reasons: most technologies used were very recent and new to the intern, which required a lot of research, and trial and error in order to make them work, mainly when documentation was lacking. The opportunity to design and build a system meant for production was also very encouraging. Solving and researching operational issues ended up being a big part of the internship. The overhead introduced from dealing with real clients, and proved to be higher than expected, but also provided a safe environment to learn about the issues of dealing with a production environment. Overall the internship provided an opportunity to further develop the engineering, operational and programming skills, and in that regard it was very complete project.

The development process was essential to the success of the project; the Kanban board helped plan development by setting monthly objectives and weekly tasks, while the team meetings provided fast feedback to the past, current and next tasks.

The people at Whitesmith were incredibly sportive of the whole internship, and provided a really good environment to work and learn. From the Qold team, to Rafael and everyone at Whitesmith, everyone helped make the internship a success.

9.2 The Future

While the core of the system was developed, the every day needs of the team need to be addressed. During the development of the project, new ideas and features for the system were considered, but since they were outside the scope of the internship, they have been postponed. None of the features require a change in the system, but instead small additions, such as new alerts and topologies for the stream processor. These system was designed with the goal of making these features easy to add and develop.

From the project envisioned initially, only one aspect wasn't achieved: the gateways were initially designed to have individual certificates for client authentication. Instead, due to time constrains and lack of support from some tools, the authentication is being handled through token. This feature was not required by Whitesmith, but was a personal objective for the intern, so it does not compromise the project in any way.

9.3 The Project

A characteristic of IoT systems is the large volume of data and throughput it creates, due to the regular readings made by devices; a small IoT system can have many times

the performance requirements of other bigger web services. Because of this, the main requirement for these systems is usually the scalability, because even at the start of the product a high throughput is expected and the system must be able to grow along with its business. Qold is now ready to take on the European market, with a reliable and scalable system, capable of growing along with it.

References

[1] Hazard Analysis Critical Control Point: REGULATION (EC) No 852/2004 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 29 April 2004

[2] Dissertation/Internship Course: https://apps.uc.pt/courses/PT/unit/79642/15102/2015-2016

[3] University of Coimbra: http://www.uc.pt/

[4] Whitesmith: http://www.whitesmith.co/

[5] Qold: https://www.qold.co/

[6] Eric Tschetter, *'Real Real-Time. For Realz'*: http://druid.io/blog/2013/05/10/real-time-for-real.html

[7] Kevin Ashton: http://newsroom.cisco.com/feature-content?articleId=1558161

[8] Gartner: http://www.gartner.com/newsroom/id/3143521

[9] Cisco: http://share.cisco.com/internet-of-things.html

[10] MongoDB: https://www.mongodb.org/

[11] Proton: http://catalogue.fiware.org/enablers/complex-event-processing-cep-proactive-technology-online

[12] Rouan Wilsenach, *'DevOps Culture'*: http://martinfowler.com/bliki/DevOpsCulture.html

[13] Martin Fowler, *'The New Methodology'*: http://www.martinfowler.com/articles/newMethodology.html

[14] David J. Anderson, *'Lean Software Development'*: https://msdn.microsoft.com/en-us/library/hh533841(v=vs.120).aspx

[15] Martin Fowler, *'Continuous Delivery'*: http://martinfowler.com/bliki/ContinuousDelivery.html

[16] Jenkins: https://jenkins.io/

[17] Dan Radigan, *'A brief introduction to kanban'*: https://www.atlassian.com/agile/kanban/

[18] Trello: https://trello.com/

[19] MoSCoW: International Institute of Business Analysis, *'A Guide to the Business Analysis Body of Knowledge'*

[20] Paolo Patierno, *'An IoT Platforms Match: Microsoft Azure IoT vs Amazon AWS IoT'*: https://paolopatierno.wordpress.com/2015/10/13/an-iot-platforms-match-microsoft-azure-iot-vs-amazon-aws-iot/

[21] Azure IoT Hub: https://azure.microsoft.com/en-us/services/iot-hub/

[22] Sam Newman, *'Building Microservices'*:

[23] Martin Fowler, *'Microservices'*: http://martinfowler.com/articles/microservices.html

[24] L Peter Deutsch, *'The Eight Fallacies of Distributed Computing'*: https://blogs.oracle.com/jag/resource/Fallacies.html

[25] ETCD: https://github.com/coreos/etcd
[26] Consul: https://www.consul.io/
[27] HAProxy: http://www.haproxy.org/
[28] Zookeeper: https://zookeeper.apache.org/
[29] Datadog: https://www.datadoghq.com/
[30] New Relic: https://newrelic.com/
[31] Elasticsearch: https://www.elastic.co/
[32] Kibana: https://www.elastic.co/products/kibana
[33] Logentries: https://logentries.com
[34] Apache Mesos: http://mesos.apache.org/
[35] Fleet: https://coreos.com/using-coreos/clustering/
[36] Docker Swarm: https://docs.docker.com/swarm/overview/
[37] Kubernetes: http://kubernetes.io/
[38] Apache Storm: http://storm.apache.org/
[39] Terraform: https://www.terraform.io/
[40] Ansible: https://www.ansible.com/
[41] Chef: https://www.chef.io/chef/
[42] HDFS: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[43] AWS S3: http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html
[44] Simon Brown, *'The Art of Visualizing Software Architecture'*:
[45] Apache Storm vs Apache Spark, Apache Flink Benchmarks:
https://www.infoq.com/news/2015/12/yahoo-flink-spark-storm
[46] Paul Dix, Why Time-Series Matters For Metrics Real-Time and Sensor Data:
https://influxdata.com/wp-content/uploads/2016/05/
Time-Series-Tech-Paper-6-6.pdf
[47] Baron Schwartz, Time-Series Databases and InfluxDB:
http://www.xaprb.com/blog/2014/03/02/time-series-databases-influxdb/
[48] Netflix, Cassandra Benchmark: http://techblog.netflix.com/2011/11/
benchmarking-cassandra-scalability-on.html
[49] JWT: https://jwt.io/
[50] NaCI: http://nacl.cr.yp.to/
[51] Resin: https://resin.io/
[52] Twilio: https://www.twilio.com/
[53] Ansible: https://www.ansible.com/
[54] RabbitMQ: https://www.rabbitmq.com/
[55] Apache Kafka: http://kafka.apache.org/
[56] MQTT: http://mqtt.org/
[57] AMQP: https://www.amqp.org/
[58] Apache Storm: http://storm.apache.org/
[59] Apache Flink: https://flink.apache.org/
[60] Apache Cassandra: http://cassandra.apache.org/
[61] KairosDB: https://kairosdb.github.io/
[62] Apache Spark: http://spark.apache.org/streaming/
[63] OpenTSDB: http://opentsdb.net/
[64] InfluxDB: https://influxdata.com/
[65] Golang: https://golang.org/
[66] HDFS: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[67] Zookeeper: https://zookeeper.apache.org/
[68] Redis: http://redis.io/

[69] HBase: https://hbase.apache.org/
[70] Docker: https://www.docker.com/
[71] Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio, *"An Updated Performance Comparison of Virtual Machines and Linux Containers"*:
[72] CoreOS: https://coreos.com/
[73] Official Apache Kafka Benchmark: http://kafka.apache.org/07/performance.html
[74] External Apache Kafak Benchmark: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines
[75] Apache Kafka at LinkedIn: https://engineering.linkedin.com/apache-kafka/how-we_re-improving-and-advancing-kafka-linkedin
[76] Apache Storm, Flink and Spark Benchmarks: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at
[77] Apache Cassandra Benchmark: http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html
[78] Digital Ocean: https://www.digitalocean.com/
[79] Eurostat: *Annual detailed enterprise statistics for services (NACE Rev. 2 H-N and S95).*

Appendix

A

Planning

Gantt charts were created at the start of each semester, in order to plan and provide an map of the project development.

Due to unforeseen circumstances, as well as the Agile nature of the development process and the fast nature of the context, some adaptations were necessary and Gantt charts evolved. This section contains both the initial and final Gantt charts and task tables.

A.1 First Semester

The first semester went mostly according to plan, with some weeks lost at the end due to extra work required for other courses.

Task	Start	End	Duration (days)
Introduction to Whitesmith	13/09/2015	19/09/2015	7
Learn internal tools			
Study development methodologies			
Internship scope and objectives	20/09/2015	03/10/2015	21
Study the Qold system and market			
Study reference IoT systems			
Requirements	04/10/2015	24/10/2015	14
Discuss Qold objectives with the team			
Analysis of functional requirements			
Analysis of quality attributes			
State of the Art	25/10/2015	07/11/2015	21
Research State of the Art IoT systems			
Research IoT technologies			
Study Legacy system	08/11/2015	28/11/2015	21
New architecture	29/11/2015	19/12/2016	21
Design of high level architecture			
Validate Architecture			
Tools and technologies	20/12/2016	09/01/2016	21
Research and compare tools and technologies			
Write report	10/01/2016	23/01/2016	14

Table A.1: Tasks planned for the first semester

ID	Task Name	Start	Finish	Duration
1	Introduction to Whitesmith	13/09/2015	19/09/2015	7d
2	Scope and Objectives	20/09/2015	03/10/2015	14d
3	Requirements	04/10/2015	24/10/2015	21d
4	State of the Art	25/10/2015	07/11/2015	14d
5	Legacy system	08/11/2015	28/11/2015	21d
6	Design Architecture	29/11/2015	19/12/2015	21d
7	Research Technologies	20/12/2015	09/01/2016	21d
8	Write Report	10/01/2016	23/01/2016	14d

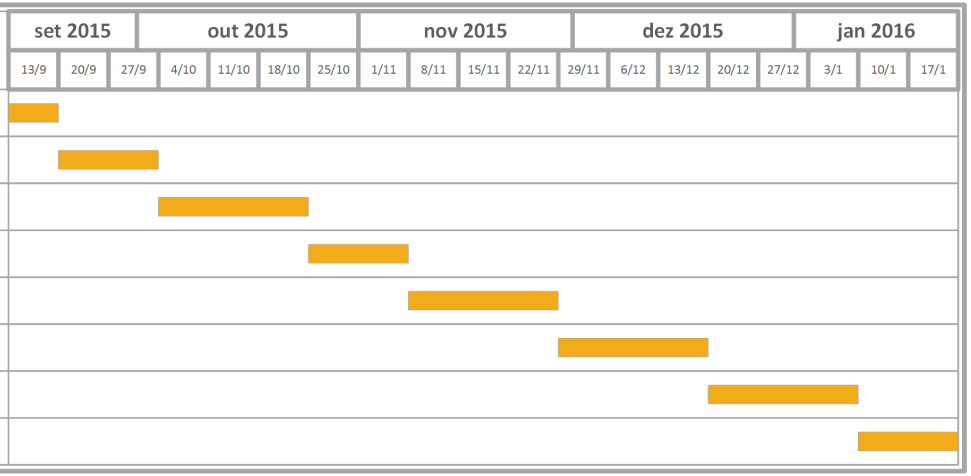

Figure A.1: 1 Semester Gantt: Initial

Task	Start	End	Duration (days)
Introduction to Whitesmith			
Learn internal tools	13/09/2015	14/09/2015	2
Study development methodologies	15/09/2015	19/09/2015	5
Internship scope and objectives			
Study the Qold system and market	20/09/2015	25/09/2015	6
Study reference IoT systems	26/09/2015	03/10/2015	8
Requirements			
Discuss Qold objectives with the team	04/10/2015	06/10/2015	3
Analysis of functional system requirements	07/10/2015	17/10/2015	11
Analysis of quality attributes	18/10/2015	24/10/2015	7
State of the Art			
Research State of the Art IoT systems	25/10/2015	28/10/2015	4
Research IoT technologies	29/10/2015	03/11/2015	7
Research Big Data	04/11/2015	07/11/2015	3
Legacy system			
Study the legacy system architecture	08/11/2015	13/11/2015	6
Automate gateway provisioning	14/11/2015	20/11/2015	7
Legacy CEP engine rules development	21/11/2015	28/11/2015	8
New architecture			
Design of high level architecture	29/11/2015	05/12/2015	7
Tools and technologies			
Research tools and technologies	03/01/2016	06/01/2016	4
Compare solutions	07/01/2016	08/01/2016	2
Write report	10/01/2016	23/01/2016	14

Table A.2: Tasks completed during the first semester

A.2 Second Semester

The planning of the second semester suffered some changes, mainly due to the choice of focusing more time on orchestration. The gateway development and data ingestion also took longer than expected.

ID	Task Name	Start	Finish	Duration
1	Introduction to Whitesmith	13/09/2015	19/09/2015	7d
2	Scope and Objectives	20/09/2015	03/10/2015	14d
3	Requirements	04/10/2015	24/10/2015	21d
4	State of the Art	25/10/2015	07/11/2015	14d
5	Legacy system	08/11/2015	28/11/2015	21d
6	Design Architecture	29/11/2015	05/12/2015	7d
7	Research Technologies	03/01/2016	08/01/2016	6d
8	Write Report	10/01/2016	23/01/2016	14d

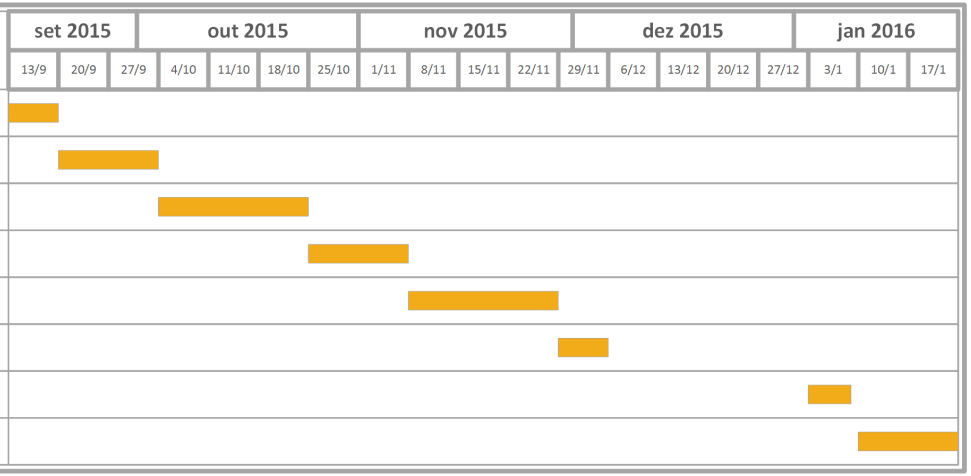

Figure A.2: 1 Semester Gantt: Final

Task	Start	End	Duration (days)
Planning	08/02/2016	11/02/2016	4
Change scope of the Internship			
Change architecture			
Gateway Communication	12/02/2016	20/02/2016	9
Research data ingestion tools			
Research communication protocols			
Deploy data ingestion tool			
Study Legacy gateway firmware	21/02/2016	27/02/2016	7
Gateway Design	28/02/2016	13/03/2016	15
Design the gateway architecture			
Study update and provisioning solutions			
Implement update and provisioning playbooks			
New gateway firmware	14/03/2016	31/03/2016	18
Implement device-gateway component			
Implement gateway-cloud component			
Add new gateway features			
Security	01/04/2016	12/04/2016	12
Research encryption mechanisms			
Research authentication mechanisms			
Deploy network encryption layer			
Raw Database	13/04/2016	18/04/2016	6
Research NoSQL technologies			
Deploy raw database			
Implement module to persist data			
Device Database	19/04/2016	23/04/2016	5
Design data model			
Deploy device database			
Time-Series Database	24/04/2016	13/05/2016	20
Research time-series database technologies			
Deploy time-series database			
Implement module to persist datapoints			
Rest API	14/05/2016	28/05/2016	15
Research Rest best practices			
Design API endpoints			
Implement the API			
Complex Event Processing	29/05/2016	07/06/2016	10
Research Complex Event Processing tools			
Deploy event processing tool			
Implement rules			
System validation	08/06/2016	11/06/2016	4
Write Report	12/06/2016	30/06/2016	19

Table A.3: Tasks planned for the the second semester

ID	Task Name	Start	Finish	*Duration*
1	Planning	08/02/2016	11/02/2016	4d
2	Gateway Communication	12/02/2016	20/02/2016	9d
3	Legacy gateway firmware	21/02/2016	27/02/2016	7d
4	Gateway Design	28/02/2016	13/03/2016	15d
5	New gateway firmware	14/03/2016	31/03/2016	18d
6	Security	01/04/2016	12/04/2016	12d
7	Raw Database	13/04/2016	18/04/2016	6d
8	Device Database	19/04/2016	23/04/2016	5d
9	Time-Series Database	24/04/2016	13/05/2016	20d
10	Rest API	14/05/2016	28/05/2016	15d
11	Complex Event Processing	29/05/2016	07/06/2016	10d
12	System validation	08/06/2016	11/06/2016	4d
13	Write Report	12/06/2016	30/06/2016	19d

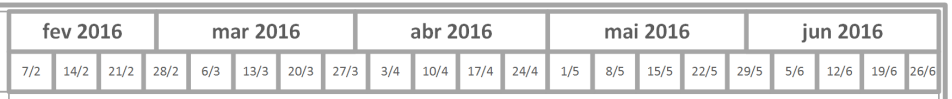

Figure A.3: 2 Semester Gantt: Initial

Task	Start	End	Duration (days)
Planning			
Change scope of the Internship	08/02/2016	08/02/2016	1
Change architecture	09/02/2016	11/02/2016	3
Gateway Communication			
Research data ingestion tools	12/02/2016	13/02/2016	2
Research communication protocols	14/02/2016	14/02/2016	1
Study and prototype Kafka	15/02/2016	17/02/2016	3
Study and prototype RabbitMQ	18/02/2016	20/02/2016	3
Study Legacy gateway firmware	21/02/2016	27/02/2016	7
Gateway Design			
Design the gateway architecture	28/02/2016	03/03/2016	5
Study update and provisioning solutions	04/03/2016	04/03/2016	1
Implement update and provisioning playbooks	05/03/2016	13/03/2016	8
New Gateway Firmware			
Implement device-gateway component	14/03/2016	23/03/2016	10
Implement gateway-cloud component	24/03/2016	27/03/2016	4
Orchestration			
Research orchestration challenges	28/03/2016	29/03/2016	2
Research orchestration solutions	30/03/2016	01/04/2016	3
Study Kubernetes	02/04/2016	06/04/2016	5
Deploy Kubernetes	07/04/2016	17/04/2016	11
Deployment			
Deploy RabbitMQ	17/04/2016	17/04/2016	1
Improve gateway communication semantics	18/04/2016	29/04/2016	12
Integrate RabbitMQ and the legacy system	30/04/2016	30/04/2016	1
Deploy new gateways to clients	31/04/2016	31/04/2016	1
Databases			
Research time-series database technologies	01/05/2016	6/05/2016	6
Research NoSQL technologies	07/05/2016	09/05/2016	3
Research encryption and authentication	10/05/2016	10/05/2016	1
Configure Cassandra and KairosDB databases	11/05/2016	13/05/2016	3
Design data model	14/05/2016	15/05/2016	2
Deploy Cassandra	16/05/2016	16/05/2016	1
Deploy KairosDB	16/05/2016	16/05/2016	1
Rest API			
Study the Go language and frameworks	17/05/2016	17/05/2016	1
Study Rest best practices	18/05/2016	18/05/2016	1
Design API endpoints	18/05/2016	18/05/2016	1
Implement tests for the API	19/05/2016	19/05/2016	1
Implement the API	20/05/2016	22/05/2016	3
Deploy Rest API	23/05/2016	23/05/2016	1
Stream Processing			
Research Stream Processing tools	24/05/2016	24/05/2016	1
Deploy kafka and remove RabbitMQ	25/05/2016	28/05/2016	4
Update gateway firmware to use kafka	29/05/2016	30/05/2016	3
Deploy Storm	01/06/2016	03/06/2016	3
Implement topologies	04/06/2016	13/06/2016	10
System validation	14/06/2016	16/06/2016	3
Write Report	17/06/2016	30/06/2016	14

Table A.4: Tasks completed during the second semester

ID	Task Name	Start	Finish	*Duration*
1	Planning	08/02/2016	11/02/2016	4d
2	Gateway Communication	12/02/2016	20/02/2016	9d
3	Legacy gateway firmware	21/02/2016	27/02/2016	7d
4	Gateway Design	28/02/2016	13/03/2016	15d
5	New gateway firmware	14/03/2016	27/03/2016	14d
6	Orchestration	28/03/2016	17/04/2016	21d
7	Deployment	17/04/2016	30/04/2016	14d
8	Databases	01/05/2016	16/05/2016	16d
9	Rest API	17/05/2016	23/05/2016	7d
10	Stream Processing	24/05/2016	13/06/2016	21d
11	System validation	14/06/2016	16/06/2016	3d
12	Write Report	17/06/2016	05/07/2016	19d

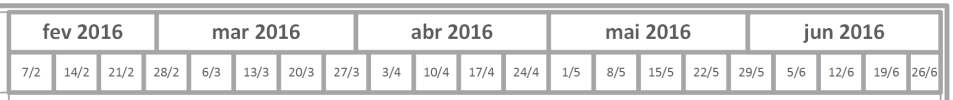

Figure A.4: 2 Semester Gantt: Final

B

Risk Management

The following risks were identified and evolved throughout the internship, they were discussed and prioritized every weekly meeting. A Trello board was used to keep track of the risks and organized them by priority. Although the impact, probability and timeframe changed during the development, the version presented is representative of the project as a whole.

Inpact\Probability	High	Medium	Low
High	R2	R1, R5	R4
Medium		R3	R9
Low			R6, R7, R8

Figure B.1: Risk exposure matrix

State of the Art technologies and tools	
Condition	The use of State of the Art technologies and tools, leads to problems, such as unexpected bugs and undocumented issues.
Consequence	Can cause delays in development, or force changes in architecture, tools or deployment strategy.
Impact	High
Probability	Medium
Timeframe	Long
Mitigation	Limit the number of technologies. Prepare alternatives for each tool used. Study articles of working deployments.
Observations	The impact and probability of this risk were High during the whole project. The risk did not impact development.

Table B.1: Risk 01

Lack of documentation for technologies and tools	
Condition	The technologies and tools used lack proper documentation.
Consequence	Can cause delays in development, or force changes in architecture, tools or deployment strategy.
Impact	High
Probability	High
Timeframe	Long
Mitigation	Prepare alternatives for each tool used. Study articles of working deployments. Study the documentation when comparing tools.
Observations	The impact and probability of this risk were High during the whole project. The risk affected development, Storm in particular severely lacks documentation.

Table B.2: Risk 02

Deployment setup is not trivial	
Condition	The deployment setup is not trivial, using uncommon physical configurations.
Consequence	Can cause delays in development, or force changes in architecture, tools or deployment strategy.
Impact	Medium
Probability	Medium
Timeframe	Medium
Mitigation	Study articles of working deployments. Study the deployment setup when comparing tools.
Observations	The impact and probability of this risk were High during the initial weeks, changed to low after Kubernetes was deployed. The risk affected development during Kubernetes and Kafka deployment.

Table B.3: Risk 03

Integration issues between tools	
Condition	The integration between tools is not trivial or is outdated.
Consequence	Can cause delays in development, or force changes in architecture, tools or deployment strategy.
Impact	High
Probability	Low
Timeframe	Long
Mitigation	Study articles of working deployments. Study the deployment setup when comparing tools. Study the integration of the different tools when comparing them.
Observations	During most of development the impact and probability of this risk were High and Low respectively. The risk affected development during Storm deployment, as integration module with RabbitMQ was outdated.

Table B.4: Risk 04

Inexperience with technologies	
Condition	The use of new technologies and tools, has higher learning curve than expected.
Consequence	Can cause delays in development, or force changes in architecture, tools or deployment strategy.
Impact	High
Probability	Medium
Timeframe	Long
Mitigation	Limit the number of technologies. Prepare alternatives for each tool used. Study articles of working deployments.
Observations	During most of development the impact and probability of this risk were High and Medium respectively. The support received at Whitesmith during the first weeks, kept this risk from coming true.

Table B.5: Risk 05

Integration issues with legacy	
Condition	The new system cannot be easily or gradually integrated with the legacy.
Consequence	Can cause delays in development, or require changes in the components.
Impact	Low
Probability	Low
Timeframe	Medium
Mitigation	Design a decoupled architecture. Prepare integration strategies early.
Observations	The impact and probability of this risk were Low during the project. Both systems were decoupled enough to ensure a smooth integration.

Table B.6: Risk 06

Lack of legacy documentation	
Condition	The legacy system lacks important documentation.
Consequence	Can hinder development or integration between systems.
Impact	Low
Probability	Low
Timeframe	Short
Mitigation	Communicate with the development team directly.
Observations	The impact and probability of this risk were Low during the project. The existing documentation was enough.

Table B.7: Risk 07

Parallel Development Divergence	
Condition	The legacy system evolving in parallel diverges to much from the initial specification.
Consequence	Can cause integration issues, or require changes the architecture.
Impact	Low
Probability	Low
Timeframe	Medium
Mitigation	Communicate with the development team.
Observations	The impact and probability of this risk were Low during the project. The legacy project did not evolve during the second semester.

Table B.8: Risk 08

Parallel Development Convergence	
Condition	The legacy system evolving in parallel implements common components first.
Consequence	Can cause duplicated work, integration issues or require changes in the architecture to accommodate small divergences.
Impact	Medium
Probability	Low
Timeframe	Long
Mitigation	Communicate with the development team.
Observations	The impact and probability of this risk were Low during the project. The legacy project did not evolve during the second semester.

Table B.9: Risk 09

C

Load estimation

Whitesmith is currently focusing on the Portuguese market, where the Food and Beverage sector was reported to be around 76 thousand enterprises in 2012. After establishing sales in Portugal, the goal is to expand to the UK, and eventually the European market. The UK market is approximately the same size as Portugal, and at the European level the sector has more than 1,547 million enterprises[79].

C.1 Business Estimation

Using Whitesmith's business estimations, and the fact that each client has on average 15 equipments, the following estimations can be made:

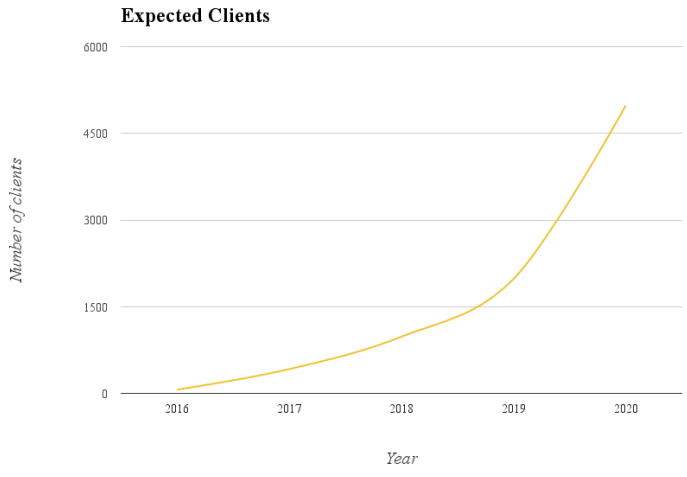

Figure C.1: Whitesmith's business estimation

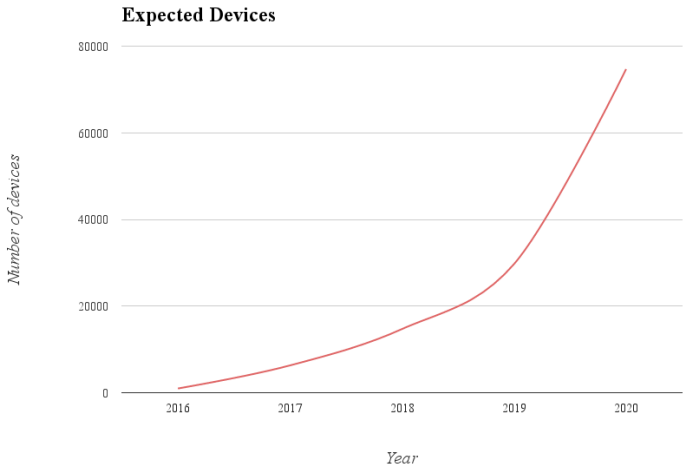

Figure C.2: Device estimated

C.2 Load Estimation

Using the legacy system as a reference, each reading sent by a gateway is an average of 200 bytes in size. Assuming a worst case increase in volume by a multiplier of 5 and that each device reports only once every 2 minutes, the following projections have been made for the next 5 years of operation. Even if the values end up not representing real life condition, they should at least be in the same order of magnitude and serve as a good baseline.

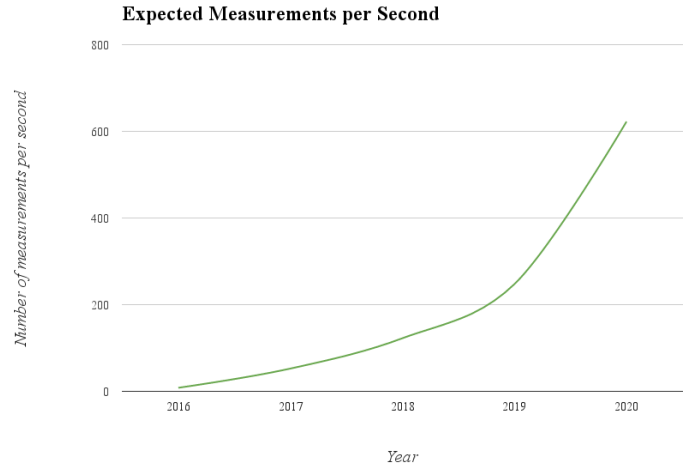

Figure C.3: Messages estimated

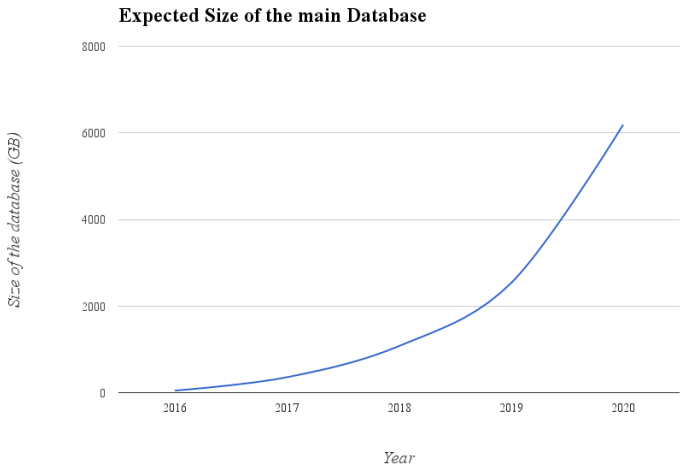

Figure C.4: Data volume estimated

D

API Endpoints

The Qold API has the following endpoints for authentication management:

Endpoint	/api/v1/admin/token
Method	POST
Description	Login
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Basic <base64(<username:password>)>]"`
Request Body	
Response Body	`{"token":"<token>"}`
Status Code	200

Table D.1: Endpoint: Create Token

Endpoint	/api/v1/admin/token
Method	GET
Description	Check token validity
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	
Response Body	
Status Code	200

Table D.2: Endpoint: Check token validity

Endpoint	/api/v1/admin/users
Method	POST
Description	Create User
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	`{"username":"user","password":"pass",` `"permission":"permission"}`
Response Body	
Status Code	201

Table D.3: Endpoint: Create User

Endpoint	/api/v1/admin/users/user
Method	GET
Description	Show User
Request Header	`"Content-Type":"application/json", "Authorization":"[Bearer <token>]"`
Request Body	
Response Body	`{"username":"(user)","password":"(hash)", "permission":"(permission)"}`
Status Code	200

Table D.4: Endpoint: Show User

Endpoint	/api/v1/admin/users/user
Method	PUT
Description	Update User
Request Header	`"Content-Type":"application/json", "Authorization":"[Bearer <token>]"`
Request Body	`{"username":"(user)","password":"(pass)", "permission":"(permission)"}`
Response Body	`{"username":"(user)","password":"(hash)", "permission":"(permission)"}`
Status Code	200

Table D.5: Endpoint: Update User

Endpoint	/api/v1/admin/users/user
Method	DELETE
Description	Delete User
Request Header	`"Content-Type":"application/json", "Authorization":"[Bearer <token>]"`
Request Body	
Response Body	
Status Code	204

Table D.6: Endpoint: Delete User

The Qold API has the following endpoints for device management:

Endpoint	/api/v1/admin/devices
Method	POST
Description	Create Device
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	
Response Body	`{"id":2,"auth":64089519511846157,"state":false}`
Status Code	201

Table D.7: Endpoint: Create Device

Endpoint	/api/v1/admin/devices
Method	GET
Description	List Devices
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	
Response Body	`[{"id":1,"state":false},{"id":1234,"state":true,` `"config":{"maxTemperature":"10.54","minTemperature":"-10.54"}}]`
Status Code	200

Table D.8: Endpoint: List Devices

Endpoint	/api/v1/admin/devices/device
Method	GET
Description	Get Device
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	
Response Body	`{"id":1234,"state":true,` `"config":{"maxTemperature":"10.54","minTemperature":"-10.54"}}`
Status Code	200

Table D.9: Endpoint: Get Device

Endpoint	/api/v1/admin/devices/device
Method	DELETE
Description	Delete Device
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	
Response Body	
Status Code	204

Table D.10: Endpoint: Delete Device

Endpoint	/api/v1/admin/devices/device/config
Method	PUT
Description	Config Device
Request Header	`"Content-Type":"application/json", "Authorization":"[Bearer <token>]"`
Request Body	`{"maxTemperature":"10.54","minTemperature":"-10.54"}`
Response Body	
Status Code	200

Table D.11: Endpoint: Config Device

Endpoint	/api/v1/admin/devices/device/login
Method	POST
Description	Login Device
Request Header	`"Content-Type":"application/json", "Authorization":"[Bearer <token>]"`
Request Body	`{"auth":"64089519511846157"}`
Response Body	
Status Code	200

Table D.12: Endpoint: Login Device

Endpoint	/api/v1/admin/devices
Method	PUT
Description	Create Device using the legacy API
Request Header	`"Content-Type":"application/json", "Authorization":"[Bearer <token>]"`
Request Body	`{"comfort_interval":{"min":1,"max":5}, "group":"Beach","name":"Kitchen", "user":{"name":"Owner","phone":"+123456789"}}`
Response Body	`{"Id":1234,"Auth":64089519511846157, "State":false,"Config":{"maxTemp":10,"minTemp":-10}}`
Status Code	200

Table D.13: Endpoint: Leagcy Device

The Qold API has the following endpoints for gateway management:

Endpoint	/api/v1/admin/gateways
Method	POST
Description	Create Gateway
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	
Response Body	`{"id":1,"auth":"B12F31EBDDF2D982895195964FEAB0065031430B"}`
Status Code	201

Table D.14: Endpoint: Create Gateway

Endpoint	/api/v1/admin/gateways
Method	GET
Description	List Gateways
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	
Response Body	`[{"id":2,"config":{"connectivity":"4G"}},{"id":1234}]`
Status Code	200

Table D.15: Endpoint: List Gateways

Endpoint	/api/v1/admin/gateways/gateway
Method	GET
Description	Get Gateway
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	
Response Body	`{"id":2,"config":{"connectivity":"4G"}}`
Status Code	200

Table D.16: Endpoint: Get Gateway

Endpoint	/api/v1/admin/gateways/gateway
Method	DELETE
Description	Delete Gateway
Request Header	`"Content-Type":"application/json",` `"Authorization":"[Bearer <token>]"`
Request Body	
Response Body	
Status Code	204

Table D.17: Endpoint: Delete Gateway

Endpoint	/api/v1/admin/gateways/gateway/config
Method	PUT
Description	Config Gateway
Request Header	`"Content-Type":"application/json", "Authorization":"[Bearer <token>]"`
Request Body	`{"State":"Active", "connectivity":"4G"}`
Response Body	
Status Code	200

Table D.18: Endpoint: Config Gateway

Endpoint	/api/v1/admin/gateways/gateway/login
Method	POST
Description	Login Gateway
Request Header	`"Content-Type":"application/json", "Authorization":"[Bearer <token>]"`
Request Body	`{"auth":"B12F31EBDDF2D982895195964FEAB0065031430B"}`
Response Body	
Status Code	200

Table D.19: Endpoint: Login Gateway

E

Benchmarks

All the benchmarks were conducted in a testing cluster that mimics the production deployment. The cluster is composed of 7 DigitalOcean[78] 20€ droplets. All the Qold system applications were running during the tests.

E.1 Kafka

In order to test Kafka's performance and scalability, 7 gateways were simulated across the cluster, one per machine. Each gateway published 100000 messages to the Kafka broker simultaneously. There was only one Kafka broker.

Messages	**Time(ms)**	**Throughput**
100000	28170	3549.875754
100000	26269	3806.768434
100000	31048	3220.819376
100000	28064	3563.283922
100000	30535	3274.930408
100000	27949	3577.945544
100000	28194	3546.853941

Table E.1: Kafka Throughput

In total Kafka was able to ingest an average of 24471.97958 messages per second, with a standard deviation of 324.6539515 across all gateways. The messages were than retrieved by a consumer that took 72 seconds, bringing the consumer throughput to around 9722 messages a second.

E.2 Storm

Before running the Storm benchmarks, the Kafka topic was filled with millions of messages, to eliminate the producer variable. Storm has an official UI which provides throughput and latency data for running topologies. Each topology was benchmarked while consuming 100.000 messages.

E.2.1 Authentication Topology

The Authentication Topology consumed 100.000 messages in 145 seconds, averaging a total of 687 messages processed per second. The topology latency was 1.46 milliseconds. The latency of each bolt:

Bolt	Time(ms)
Report Bolt	1.26
Device Bolt	0.069
Gateway Bolt	0.069
Kafka Bolt	0.014
Join Bolt	0.048

Table E.2: Storm Latency: Authentication

E.2.2 Raw Topology

The Raw Topology consumed 100.000 messages in 14 seconds, averaging a total of 7099 messages processed per second. The topology latency was 0.013 milliseconds. The latency of each bolt:

Bolt	Time(ms)
Parser Bolt	0.047
Casandra Bolt	0.086

Table E.3: Storm Latency: Raw

E.2.3 Datapoints Topology

The Datapoints Topology consumed 100.000 messages in 195 seconds, averaging a total of 514 messages processed per second. The topology latency was 90 milliseconds. The effective latency though is 3 seconds, because inserts are batched using Storm windows. The latency of each bolt:

Bolt	Time(ms)
Storage Bolt	0.03
Battery Bolt	0.001
Temperature Bolt	0.001
Publish Bolt	90

Table E.4: Storm Latency: Datapoints

E.2.4 Thresholds Topology

The Thresholds Topology consumed 100.000 messages in 21 seconds, averaging a total of 4676 messages processed per second. The topology latency was 0.213 milliseconds. The latency of each bolt:

Bolt	Time(ms)
Battery Bolt	0.1
Temperature Bolt	0.06
Info Bolt	0.001
Config Bolt	0.023
Parser Bolt	0.03

Table E.5: Storm Latency: Thresholds

E.2.5 Alerts Topology

The Alerts Topology consumed 100.000 messages in 36 seconds, averaging a total of 2799 messages processed per second. The topology latency was 0.38 milliseconds. The latency of each bolt:

Bolt	Time(ms)
Info Bolt	0.001
Loader Bolt	0.001
Time Bolt	0.358
Parser Bolt	0.021

Table E.6: Storm Latency: Alerts

E.3 KairosDB

Datapoints were inserted to KairosDB during the Storm benchmark and 30 queries were performed, requesting 1000 points. This query was chosen because it is similar to the queries performed in the Qold system. The queries averaged in 210 milliseconds with a standard deviation of 16.26663605 milliseconds.