AWS + C3.ai

Application Development Time and Cost Savings

A Third-Party Report, Prepared & Written by Premier Cloud Native System Integrator

Note from C3.ai

C3.ai commissioned a third-party system integrator – with extensive experience in developing enterprise applications on the AWS cloud for Fortune 1000 customers – to develop a Predictive Maintenance application for a network of devices, to run on the AWS cloud. The system integrator was given a Product Specification and asked to develop the same application using two approaches:

1. Build the application using only AWS native services;
2. Build the application using the C3 AI Suite in combination with AWS services.

The following report was written by the third-party system integrator to describe their process in developing the application, including a detailed account of developer time, effort, and coding required using each approach.

Readers can download the following documents as separate PDF files:

- Product Specification: Predictive Maintenance Application for a Network of Devices
- Complete Source Code for the AWS Application and for the C3.ai + AWS Application

© 2020 C3.ai | All Rights Reserved | 20_0511

Table of Contents
Third-Party Report

Executive Summary **4**

Background 4

Findings 5

Project Narrative **8**

AWS Build 10

C3.ai + AWS Build 16

Comparative Observations 19

Conclusion **23**

© 2020 C3.ai | All Rights Reserved | 20_0511

Third-Party Report by AWS Premier System Integrator

© 2020 C3.ai | All Rights Reserved | 20_0511

Executive Summary

Our firm, a Premier AWS consulting partner with AWS competencies in Big Data and Machine Learning, was commissioned by C3.ai to conduct the Device Predictive Maintenance development project described in this document, and to prepare the following report of our findings and analysis.

Background

Based on recent research from renowned IT and management consulting firms, the impact of Artificial Intelligence (AI) on the global economy is forecasted to be massive. In a 2016 report, Forrester predicted that AI-driven companies would realize $1.2 trillion in additional annual economic value compared to their laggard peers by 2020. This appears to have been a conservative forecast. McKinsey Global Institute now forecasts $3.5 - 5.8 trillion impact from AI by 2020.

While some of this value will accrue from organizations applying AI to relatively simple datasets and business processes (e.g., sales forecasting, customer churn, facility energy management), the bulk of this value will come from organizations applying AI to complex datasets and business processes (e.g., supply network and inventory optimization, process optimization, maintenance optimization, fraud detection). These complex AI applications require an architecture that ingests data from multiple IoT and transactional systems in near-real time; blends these data together; enhances these data with master data from an array of enterprise, operational, and third party data sources; uses these unified data to train predictive and optimization models to generate actionable insights; and embeds these insights into a business process.

The current trend for building even simple enterprise AI applications is to take a microservices "building blocks" approach, where the end-customer and/or their consulting partners integrate cloud service provider (CSP) microservices together to build the componentry to design, develop, host, and operate an application, followed by building bespoke application logic. C3.ai, on the other hand, offers a cohesive, low-code platform – the C3 AI Suite. This report compares the approach of using CSP building blocks to using the C3 AI Suite in conjunction with CSP microservices.

© 2020 C3.ai | All Rights Reserved | 20_0511

Findings

Three of our senior commercial software engineers built a simple Predictive Maintenance application (the "Application") using native AWS services and compared it to building the same Application on the C3 AI Suite in conjunction with AWS services.

We found that C3.ai reduced the effort and accelerated the development process by a factor of 26 times, while also reducing development risks. For less experienced teams, this could easily increase to 50 - 100 times due to the breadth and complexity of the AWS offering. Further, C3.ai solves the challenges of security, extensibility, and scalability – while streamlining the skills needed in an enterprise development team.

To build the application using native AWS services requires 83,000 lines of source code and documentation. Using the C3 AI Suite with AWS services required 1,450 lines of source code and documentation, a 99% reduction.

Tasks	AWS Application			AWS + C3.ai Application			
	Days	Full-time Equivalent Persons	Effort (Person Days)	Days	Full-time Equivalent Persons	Effort (Person Days)	Savings
Infrastructure Configuration	8.25	3	24.75	0	0	0	100%
Data Model	1	1	1	0.75	1	0.75	25%
Data Integration	3	2	6	0.75	1	0.75	87.5%
Time-Series, Metrics, Machine Learning	14	2.5	35	1.75	1	1.75	95%
Analytics	3	2	6	0.75	1	0.75	87.5%
User Interface and Testing	23	2	46	0.5	1	0.50	98.9%
Total Effort (Person Days)			**118.75**			**4.5**	
Total Lines of Code			**83,000**			**1,450**	

Figure 1: The C3 AI Suite in combination with AWS provided a 26x improvement in developer productivity compared to building the same application using only native AWS services.

© 2020 C3.ai | All Rights Reserved | 20_0511

Time and Cost of Development

As this report illustrates, the C3 AI Suite substantially accelerates development, allowing customers to derive economic value from AI applications faster than by building the necessary platform components themselves.

Risk Mitigation

Even organizations with well-funded IT departments and highly-skilled developers face substantial risks in building, deploying, and maintaining new applications. This risk increases many-fold when the scope grows beyond one simple application. The C3.ai Type System reduces their exposure to the array of risks that face a company developing custom, enterprise-scale software.

Complexity

Stitching together core infrastructure, enterprise software, platform services, data science services, and UI components into a production-scale application is a task of enormous complexity. Companies that attempt to construct custom AI platforms using building blocks risk pouring time and resources into projects that fail to reach production or deliver on their value proposition. C3.ai, by abstracting away the underlying infrastructure and presenting all data and services as accessible, manipulatable Types, minimizes this risk.

Extensibility

Companies that overcome the complexity of building a single AI application face the risk that their work will not be extensible – the data model cannot be extended to serve a second application or code is specific to infrastructure components that the company has decided to abandon or that have become obsolete. Applications built using the C3.ai Type System are tied to metadata rather than the underlying data and storage infrastructure and are thus fully extensible to support new data sources, infrastructure, algorithms, and platform services.

Security

Getting a working application is complex enough – encrypting all data in transit and at rest, limiting access to data at the row and user level, and guaranteeing backup, failover, and redundancy add new layers of complexity and represent major risks for applications that expose the enterprise's most sensitive data. C3.ai, which provides accredited, comprehensive security measures as part of its platform services, mitigates these risks.

Maintenance & Support

AI applications built from scratch risk breaking when data sources, use cases, and support staff change. Companies are forced to divert resources to maintenance, which dilutes the value that AI applications deliver, limits new development, and increases the probability of abandoning applications altogether. C3.ai mitigates this risk in two ways – by offering a flexible platform that requires less maintenance than a bespoke solution, and by providing support resources and training as part of its solution.

© 2020 C3.ai | All Rights Reserved | 20_0511

Scalability

Infrastructure scalability is crucial to enable applications built on small data sets to scale to the enterprise. AWS enables scalability typically through a manual process to request, provision, and integrate additional compute and storage resources. C3.ai employs a modular scale-out architecture that automatically requests, provisions, and releases computing resources based on the need and makes it simple to add more storage and other resources. Application scalability is also crucial to enable organizations to add more data, new sources of data, new transforms of data coming from different parts of the organization, or new application logic to extend previously built use cases. Native AWS development would require structural code changes and rewriting the entire application to incorporate these changes. C3.ai mitigates this risk via the C3.ai Type System's metadata-based abstraction.

Resource Capability

AWS development requires a very broad skill set in an organization's developers. At the most basic level, all developers need an understanding of AWS development principles, while specific team members might require a range of skills – ranging from networking and Linux server configuration to the specific details of the various managed services – unique to each CSP, that come with their own methods of utilization. Additionally, organizations need to make a significant resource investment in DevOps to ensure that what they are building on AWS can mature in a safe and scalable way. For reference, AWS recommends a year of experience working in the AWS platform prior to securing a basic AWS certification.

For C3.ai, a foundation in object-oriented concepts such as inheritance and static typing as well as exposure to the domain-specific syntax are the only critical skills required to understand the development approach. A typical C3.ai developer receives five days of training and is usually proficient in three months depending on skill level and prior experience.

© 2020 C3.ai | All Rights Reserved | 20_0511

Project Narrative

Our team of highly skilled developers compared building a simple Predictive Maintenance Application on a network of connected devices using native AWS services (the "AWS Application") to using the C3 AI Suite (the "C3.ai Application") with AWS services. In both cases, we sought to ingest, unify, and federate the raw data, process it, train a machine learning model that predicts which device is likely to fail in the following 30 days, and build an application user interface.

The provided datasets for the Application included:

- Device type, location, manufacturer, and date of manufacture
- Power grid status
- Device power source location
- Device telemetry
- Device event history
- Device power source data

Building a risk prediction model for each device required that the telemetry/measurement data be analyzed over time. For example, the Application uses the following time-series data:

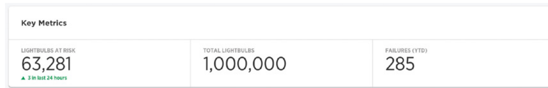
- Average Power per Device – Power usage over time for the device
- Duration On per Device – The total amount of time (in hours) that a device has been powered on up to the interval
- Switch Count per Device – The number of times a device is powered on or off
- Power Grid Status per Building – An external factor indicating whether the local power grid was functional over time at a specific building

The objective of the application is to predict the likelihood of device failure within the next 30 days from a given point in time. It is left up to the development team how to make this prediction, although the industry best practice is to train a machine learning model using the provided data. With this predictive model in place, predictions must be generated as new data is received for each device.

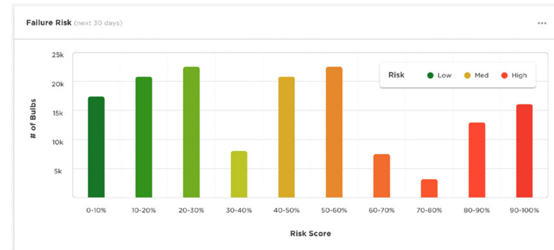
To make the predictions actionable, they must be presented to end users. The user interface for the Application consists of two screens and seven displays reporting on the number, location, risk score, and status of devices as seen below:

© 2020 C3.ai | All Rights Reserved | 20_0511

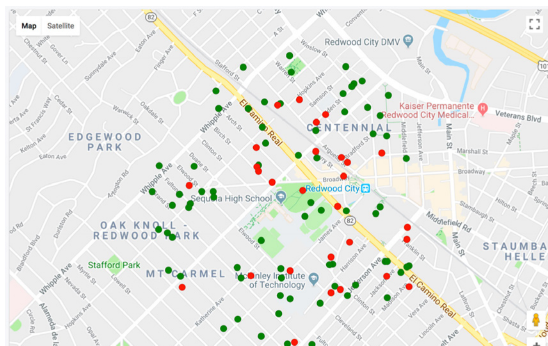
1. A summary of key metrics including the total number of devices at risk, the total number of devices, and the number of failures YTD



2. A histogram showing the distribution of devices, grouped by risk score



3. A map showing the location of all devices, colored green for devices with risk scores <50% and red for risk scores >50%



4. A table of device-level detail, including device ID, risk score, type, manufacturer, and date of install

Lightbulbs

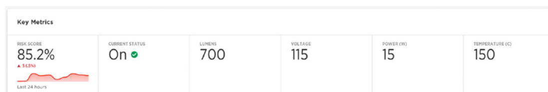
Bulb ID	Risk Score %	Bulb Type	Manufacturer	Date Started
SMBLB1	80	LED	LIFX	10/01/2017 4:00:00 AM
SMBLB2	80	LED	eufy	10/01/2017 4:00:00 AM
SMBLB3	80	LED	Cree	10/01/2017 4:00:00 AM
SMBLB4	75	LED	LIFX	10/01/2017 4:00:00 AM
SMBLB5	75	LED	GE	10/01/2017 4:00:00 AM
SMBLB6	75	LED	Cree	10/01/2017 4:00:00 AM
SMBLB7	70	LED	GE	10/01/2017 4:00:00 AM
SMBLB8	70	LED	GE	10/01/2017 4:00:00 AM
SMBLB9	65	LED	eufy	10/01/2017 4:00:00 AM
SMBLB10	65	LED	Phillips	10/01/2017 4:00:00 AM

Rows per page: 10 1-10 of many 1 2 3 4 5 ...

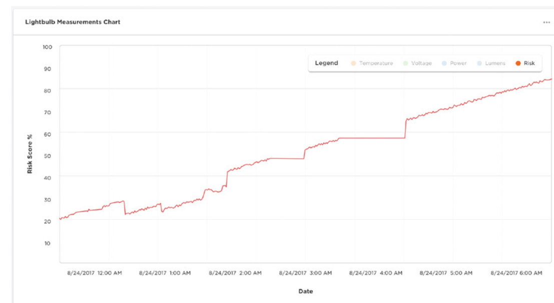
Figure 2: UI Screen 1 – Four displays showing the status and health of the entire population of devices.

All elements on the screen can be filtered by multiple dimensions.

1. A summary of key metrics including the current risk score, status, power and temperature of the selected device



2. A chart illustrating the selected device's risk score over time



© 2020 C3.ai | All Rights Reserved | 20_0511

3. A summary of key metrics including the total number of devices at risk, the total number of devices, and the number of failures YTD

Lightbulb Measurements Table

Timestamp	Risk Score %	Status	Lumens	Voltage (V)	Power (W)	Temperature (C)
2017/10/01 6:00:00 AM	85	On	700	115	15	150
2017/10/01 5:59:44 AM	84	On	700	115	15	150
2017/10/01 5:59:42 AM	82	On	700	115	15	150
2017/10/01 5:58:21 AM	80	On	700	115	15	150
2017/10/01 5:58:15 AM	80	On	700	115	15	150
2017/10/01 5:56:52 AM	79	On	700	115	15	150
2017/10/01 5:56:43 AM	79	On	700	115	15	150
2017/10/01 5:54:15 AM	78	On	700	115	15	150
2017/10/01 5:54:05 AM	78	On	700	115	15	150
2017/10/01 5:53:00 AM	77	On	700	115	15	150

Rows per page: 10 1-10 of many 1 2 3 4 5 ...

Figure 3: UI Screen 2 – Three displays showing the health and history of an individual device (accessed by selecting a device from the table in Screen 1)

AWS Build

The architecture for the AWS Application, as depicted in Figure 4, made heavy use of AWS managed services, including AWS Lambda for serverless processing, Amazon Kinesis for data streaming, Amazon S3 for storing raw data, Amazon API Gateway for RESTful services, and Amazon SageMaker for machine learning training and inference. In addition, we utilized Amazon's Relational Database Service (RDS) and Amazon DynamoDB, a NoSQL distributed key-value store database, for persistence.

This architecture stems from our collective years of experience working with AWS services. Our firm is a Premier AWS consulting partner with AWS competencies in Big Data and Machine Learning, and we have developed and deployed hundreds of applications on AWS for hundreds of Fortune 1000 customers.

At the onset of the project, our team agreed to eliminate the need for the low-level management of server and network resources. This worked well in practice and provided our developers with the flexibility to manage multiple simultaneous workstreams. By breaking the application into numerous independent microservices/components, the team was able to work in parallel integrating each service while avoiding code contention with the other developers.

© 2020 C3.ai | All Rights Reserved | 20_0511

AWS Platform Architecture for Device Predictive Maintenance Application

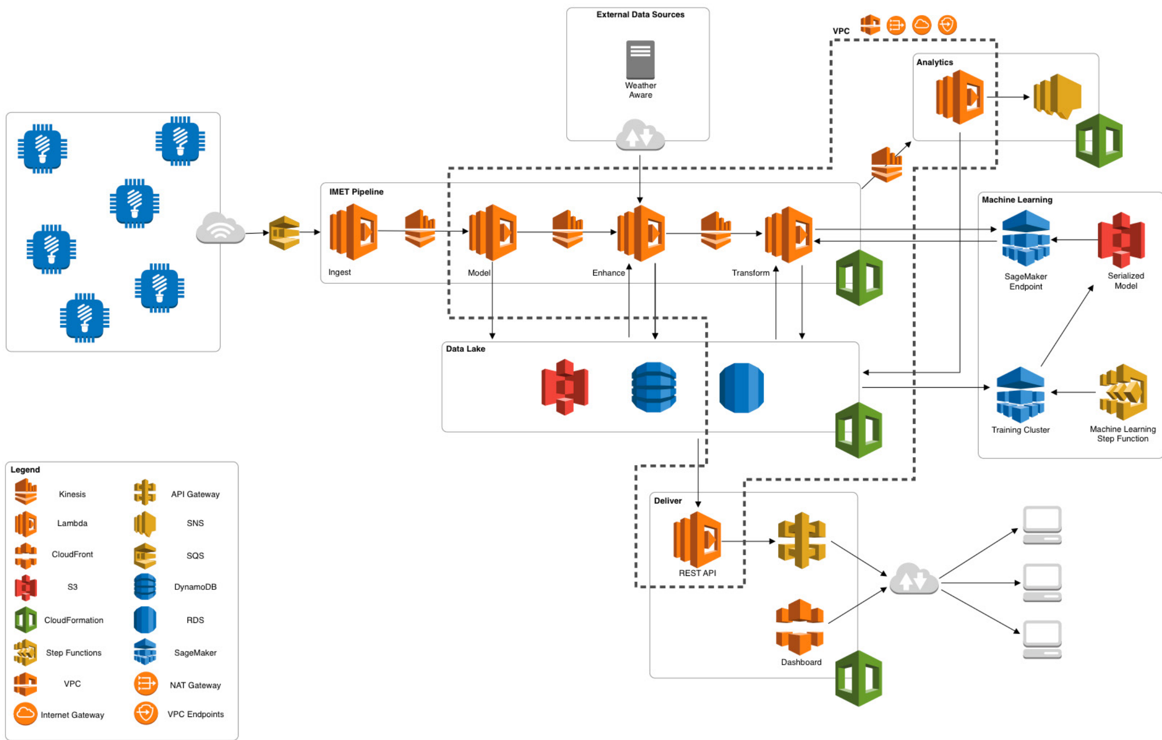


Figure 4: AWS Architecture for Device Predictive Maintenance

© 2020 C3.ai | All Rights Reserved | 20_0511

Infrastructure Configuration

Our work began with creating a new AWS account, configuring basic networking, and establishing access control and security policy permissions. As we began development, the data lake and necessary infrastructure for a data processing pipeline was created. The data lake consisted of Amazon S3, DynamoDB, and RDS. The processing pipeline was a series of AWS Lambda functions that were linked together by Amazon Kinesis Streams, allowing data to flow through each stage of ingestion, modeling, enhancement, and transformation. These resources were managed using Amazon's infrastructure-as-code service, AWS CloudFormation, to easily and quickly create re-useable templates to provision and configure all necessary infrastructure. The easiest component of the architecture, the S3 data lake and its associated data stores RDS and DynamoDB, took 0.5 FTE (full-time-equivalent) days to implement and deploy. This was one of seven CloudFormation templates, each growing in complexity as we built up the architecture layers. Each CloudFormation template ranged from hundreds to thousands of lines of custom code.

Deployment was a manual process that involved uploading the templates into CloudFormation and then running them to create or update the infrastructure. The AWS documentation reasonably described which resource parameters are necessary for a specific template resource and the AWS console provided specific feedback if an attempted template upload had errors or required refactoring. While we were able to understand and respond to these issues rapidly given our experience with AWS, our assessment is that a typical organization would need to build very robust DevOps pipelines and devote significant resourcing to ensure changes are promptly and definitively pushed into the account.

With three FTEs and the scope, we configured the AWS Application's infrastructure in eight days. **An enterprise architecture team with less experience on AWS would reasonably take at least twice as long.** It is notable that infrastructure configuration is a continuous process throughout the development lifecycle and requires ongoing maintenance post-deployment. Each AWS service we utilized has unique networking and permissions configurations that must be tweaked and debugged.

Data Model

To optimize for scalability, extensibility, and usability, we designed the AWS Application to leverage two databases. We used Amazon RDS, a fully managed relational database service, to store static structured data – device type, location, manufacturer, etc. To create our relational data model, custom SQL was hardcoded to define each table in our database. This manual process provided the necessary structure to easily store, organize, and query data. We used Amazon DynamoDB, a fully managed NoSQL distributed key-value store database, to store dynamic data – e.g., device telemetry. Unlike relational databases, NoSQL databases do not require a pre-defined data model.

Without two databases, we would be limited in our ability to filter on data, explore data, and evaluate the timed interval relationships between

© 2020 C3.ai | All Rights Reserved | 20_0511

data objects. It would also couple the scaling of both data sources to a single configuration setup. By separating the two storage services, we can scale DynamoDB independently of RDS for both improved performance and cost savings compared to running a single, larger, and more expensive RDS instance. A two-database architecture was in sync with our microservice-based approach.

While creating the AWS Application's data model was possible for one FTE in eight hours, an enterprise architecture team with less experience on AWS and internal constraints on database structure would reasonably take twice as long. **Further – and this is extremely crucial given our experience – changing or extending the data model would require the data model to be entirely refactored/rebuilt.**

Data Integration

Our initial data integration effort involved manually loading the raw CSV data via MySQL tooling. If the data were in a different format it would have likely involved writing either custom parsers or manually converting the data into a more usable format combined with potentially writing the raw SQL commands to insert this data. Furthermore, this process was entirely manual at the outset of the project to "seed" the initial data with no process in place for the ongoing ingestion of new data. While this would not be a difficult process to engineer, it would require more development time along with a process to be put in place to allow for more power sources, devices, apartments, etc., to be introduced into the system for future use.

While integrating data required two FTEs for three days, an enterprise architecture team with less experience on AWS would reasonably take 50% more time.

Time-Series Metrics and Machine Learning

Once we integrated the raw data, we began preparing it for our machine learning process. To create our time-series metrics/machine learning features, we wrote custom logic in NodeJS and used Amazon Lambda's serverless computing service to execute at run-time. Lambda allowed us to easily deploy our custom logic and orchestrate it with AWS streaming service Kinesis, with less overhead and infrastructure configuration. *However, our team encountered difficulty configuring the networking for the various AWS services we utilized.* While a serverless approach typically removes the need for networking considerations, using Amazon RDS necessitated hands-on networking configuration. Our Lambda functions had to be placed into subnets within our VPC, which then required establishing VPC Endpoints to connect out to the AWS managed services such as AWS Key Management Service, DynamoDB, and SageMaker. In addition, we had to configure a new NAT Gateway to facilitate our enhancement function to make calls over the internet to the weather endpoint provided by C3.ai. These changes required more manual configuration of the networking layer and could prove to be problematic for teams without a strong knowledge of AWS networking concepts. Additionally, this led to extra work that was not estimated and required unplanned developer time.

© 2020 C3.ai | All Rights Reserved | 20_0511

Another key difficulty was transforming our time-series metrics into the proper format for Amazon SageMaker. Rectifying this required a trial-and-error process and relying on colleagues with extensive experience working with SageMaker and its DeepAR algorithm. After revising and refactoring our approach for building the raw model input, we were able to successfully create a process for building machine learning models and for creating a request object that integrated with SageMaker. This effort involved complex custom code and changing the algorithm required significant rework of the preparation process. The lack of usability is one reason an iterative approach needed to be taken until the process fully integrated with SageMaker DeepAR. Once the model and request object were successfully created and integrated, it was easy to get predictions from SageMaker. However, additional code was required to join the predictions into the data model and store in Amazon RDS.

Precision relates to the proportion of device failures that were correctly predicted. Recall is the proportion of actual high-risk devices that were identified correctly. The area under the receiver operating characteristic curve was .795.

While creating 13 time-series metrics and one machine learning classifier was possible with 2.5 FTEs in 14 days, an enterprise architecture team with less experience on AWS, especially with AWS networking concepts, would reasonably take twice as long given their lack of experience.

Analytics

A separate process was created to evaluate and save metrics based on specific rules and use cases. Once a device measurement had been ingested and transformed, an analytics service written in Lambda was used to check if any rules were satisfied. If so, a new record was saved into RDS to mark the analytic as triggered and a message was sent to Amazon Simple Notification Service. This allows for emails, text messages, or other notifications to be triggered so that further action can be taken as a result – e.g., "Inspect this device." Expanding the notifications simply involves writing further use cases and incorporating them into the analytics service.

While this was possible for two FTEs in three days given the team's expertise, a normal enterprise architecture team would reasonably take 50% more time.

User Interface

Building the Application's UI required exposing RESTful APIs that served the results of our time-series metrics. To accomplish this, we utilized Amazon API Gateway with Lambda functions written in NodeJS. The API Gateway configuration was completed in CloudFormation using an OpenAPI specification combined with specific configuration elements for the API Gateway. Configuring Cross-origin resource sharing (CORS), which enables the client application to call the APIs directly from the browser, in the CloudFormation templates was challenging. We reverse engineered the methods and headers that the API Gateway console

© 2020 C3.ai | All Rights Reserved | 20_0511

automatically adds when enabling CORS and figured out the corresponding CloudFormation syntax. We spent many iterations to successfully set up and test CORS. The Device API accessed RDS and DynamoDB, requiring different data access methods to be written for each database and different sets of permissions that needed to be set up and configured in the CloudFormation templates. It would be ideal to establish an architecture that abstracts the data access methods; however, it would require development time to create and maintain the methods and configuration to access those data stores. The API was secured with an API key, which also required moderately complex CloudFormation configuration.

The UI components were built using Angular 6 and hosted with S3 and Amazon CloudFront. TypeScript, SASS, RxJS, Angular Material, and the Angular FlexLayout were the primary front-end technologies utilized for the Angular components. The components can be easily added and removed without affecting the other components on the page. We spent one FTE day to ensure that duplicate API calls would not be made for a component if another component already retrieved the data. The components share a service that provides the API results as a RxJS observable. The observable provides the API data for the components and will refresh the components that are subscribed to it when new data is generated. This allows the components to efficiently retrieve data refreshed without a postback for actions such as filtering. We used client-side filtering, sorting, and paging for the detail tables, but these actions will need to be done on the server side if the quantity of data for the component becomes too large. Server-side filtering, sorting, and paging would add one week to the development effort, plus two to three days for unit testing.

Hosting the front-end application on S3 alone, while possible, does not provide enough granular control over permissions and routing. We used CloudFront as an entry point to S3 and restricted access to only allow CloudFront access to the S3 bucket via an Origin Access Identity. Through CloudFront, we can enable client-side routing functionality and manage the access, custom error messages, and geographic distribution. All the S3, CloudFront, and Origin Access Identity setup was done through a CloudFormation template.

While creating the Application's UI and integrating it with our backend architecture was possible with two FTEs in 23 days, an enterprise architecture team with less experience would reasonably take 20% more time.

© 2020 C3.ai | All Rights Reserved | 20_0511

AWS Implementation Timeline: 26 Person-Weeks

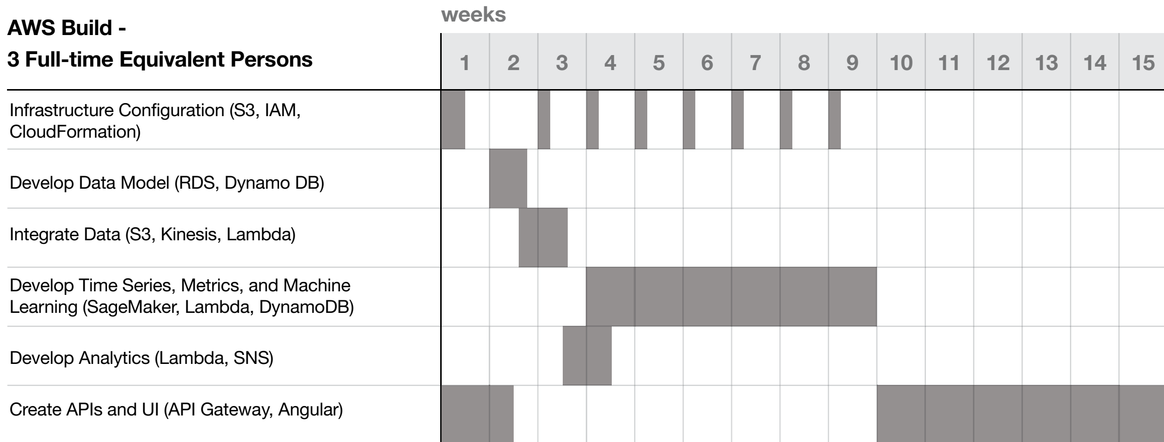


Figure 5

C3.ai + AWS Build

Developing the same Application with the C3 AI Suite in combination with AWS was much simpler.

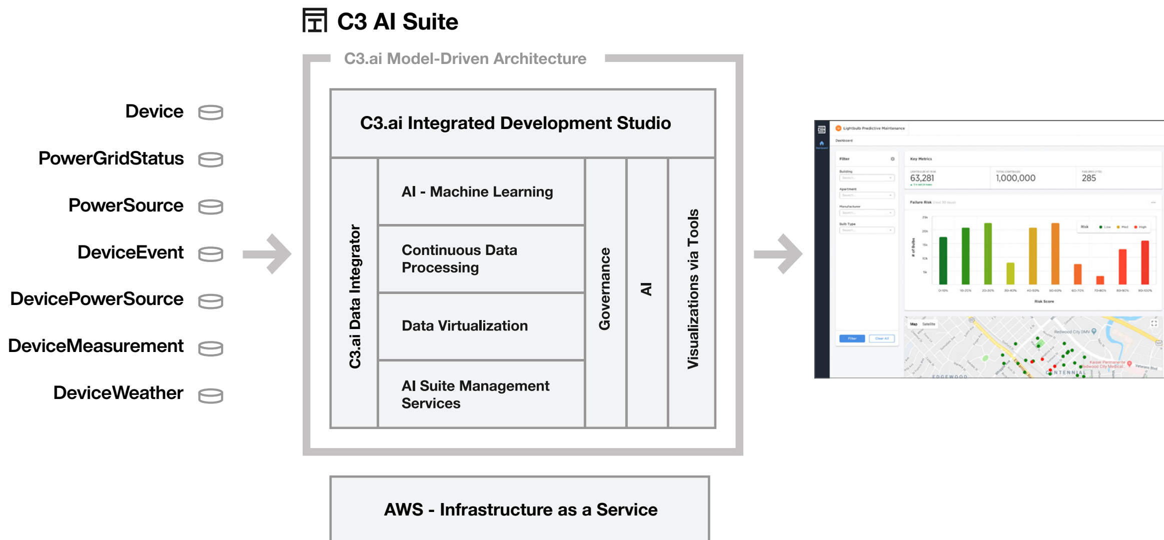


Figure 6: Architecture to build Device PM on the C3 AI Suite.

© 2020 C3.ai | All Rights Reserved | 20_0511

Infrastructure Configuration

The C3 AI Suite does not require any infrastructure to be configured or maintained. Deploying a new instance of the C3 AI Suite on AWS takes four person hours. Deploying a new C3.ai tenant within an existing instance on AWS takes approximately three minutes.

Data Model

We began building the C3.ai Application by creating C3.ai Types for use in our application. Types are representations in code of any business relevant objects – for instance, real-world entities that make up a business – in this case, devices, buildings, facilities, manufacturers, etc. Each Type contains the metadata that define its relevant datastores (distributed file system, relational, NoSQL) and its relationships to other Types in the data model (e.g., one facility has ten devices from a single manufacturer). The C3.ai Type System allows individuals with different functions and specializations – e.g., developers, data scientists, and business analysts – to work on a shared abstraction layer without having to configure or maintain the underlying data federation and storage models, dependencies, or infrastructure. **Building the Application's data model with the C3.ai Type System required six hours and one FTE.**

Data Integration

We then used the C3 AI Suite's native data integration capabilities to integrate, index, and normalize the device data. Prior to integrating data, we created six canonical Types for each of the data sources. The C3 AI Suite includes native functionality to import data from any source – while we worked with CSV files, the C3 AI Suite includes pre-built connectors to commonly-used relational databases, NoSQL databases, and distributed file systems – and map all fields to C3.ai Types for access by data scientists and developers. **Integrating data required six hours and one FTE.**

Time-Series Metrics

We then used our C3.ai Types to generate 13 metrics, which fetch Type data to produce a normalized time-series. Metrics serve as features in machine learning algorithms and can be incorporated into application logic. We also wrote some methods for the Device Type, which allow for more complex calculations on the data using JavaScript or Python. **Creating the 13 metrics required eight hours for one FTE.**

Analytics

Next, we used the C3 AI Suite's native asynchronous processing engine to create data flow events (DFEs). Using DFEs, we created three analytics that automatically generate operator alerts when certain operating thresholds were met/exceeded. These alerts could be routed via email or SMS messages. **Creating these three analytics and configuring the DFEs required one FTE for six hours.**

Machine Learning

We created risk-of-failure scores for the Application's devices using Jupyter Notebook and Python, both supported natively by the C3 AI Suite. Having the full functionality of the C3 AI

© 2020 C3.ai | All Rights Reserved | 20_0511

Suite and C3.ai Type System natively integrated with Jupyter Notebook provides easy access for data scientists to leverage tools that are familiar and effective. We trained a classification model that regressed the metrics SwitchCountWeek and DurationOnInHours against the dependent variable WillFailNextMonth to calculate the probability of failure in the next 30 days. We stored this rolling risk score as its own time-series metric RiskScore. Machine learning algorithms in the C3 AI Suite operate on all existing data, create new data that can be automatically attached to a C3.ai Type for future processing, and automatically update training and make predictions on the latest available data.

For the C3.ai Application, the area under the receiver operating characteristic curve was .990. **Training the machine learning model and the machine learning pipeline required one FTE for six hours.**

User Interface

We incorporated several of our Types and metrics in a web interface built using custom C3.ai HTML and UI templates. We used these to create the dashboard of the Application. The dashboard UI template was one JSON-styled file that contained the code for the components of the dashboard such as a status map, a filter, a histogram and a table. Our UI also included continuously and automatically updated predictive risk scores about the likelihood of device failures (incorporated using the RiskScore metric). Finally, we created a few simple potential roles that would be used by future users of the Application. These roles consisted of restricting users to permissions for specific use cases pertinent to the user. **Building the UI and configuring access controls required one FTE for four hours.**

AWS + C3.ai Implementation Timeline: 1 Person-Week

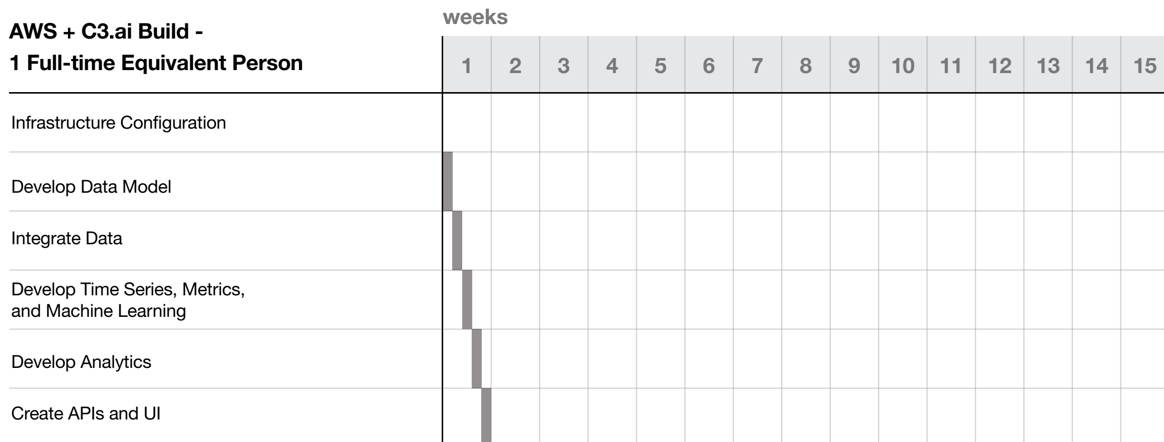


Figure 7

© 2020 C3.ai | All Rights Reserved | 20_0511

Comparative Observations

After building the Application first using AWS native services alone and then using the C3 AI Suite in conjunction with AWS, there were several differences in both output as well as the development experience.

Lines of Code

The native AWS application required 16,000 lines of custom code, including documentation, plus 67,000 lines of code for base functionality. Conversely, using the C3 AI Suite required only 1,450 lines of code including documentation due to the functionality provided by C3.ai Types. Similarly, the AWS Application required three highly experienced FTEs for 10 weeks, whereas the C3.ai Application was completed by one FTE in five days.

Jupyter Integration

Developing the machine learning portions of the AWS Application required significant time. While the full functionality and benefits of the C3 AI Suite and C3.ai Type System are natively integrated with Jupyter Notebooks, the need to develop this functionality from scratch on AWS was a major impediment to data scientist productivity.

API Creation and Publishing

The AWS Application required the development of custom APIs and front-end, comprising approximately 30% of the development effort. In contrast, the C3.ai Type System is fully REST API-enabled, and the APIs are automatically created. The actual lines of code required for the C3.ai Application was orders of magnitude less than the custom Angular application and REST API built for AWS.

Infrastructure Management

Another major difference between the two Applications was the time spent managing the underlying infrastructure. Using AWS, approximately 25% of our development time was focused on creating, configuring, and re-configuring infrastructure. As the application matures, these infrastructure components require ongoing maintenance, taking developers away from more productive tasks (like building new AI applications). Infrastructure setup and management were of no concern in developing the C3.ai Application.

Time-Series Management

An important feature that reduced development time on C3.ai is C3.ai's treatment of time-series data as a first-class operation. With AWS there was significant complexity arising from the need to manually handle datetime data using date library functionality and custom code. C3.ai native functionality is intuitive, which made configuring and adjusting a wide range of possible time-series computations very easy with no need for custom datetime handling.

© 2020 C3.ai | All Rights Reserved | 20_0511

Significant benefits of working on the C3 AI Suite compared to AWS native services alone included:

Key C3 AI Suite Efficiencies

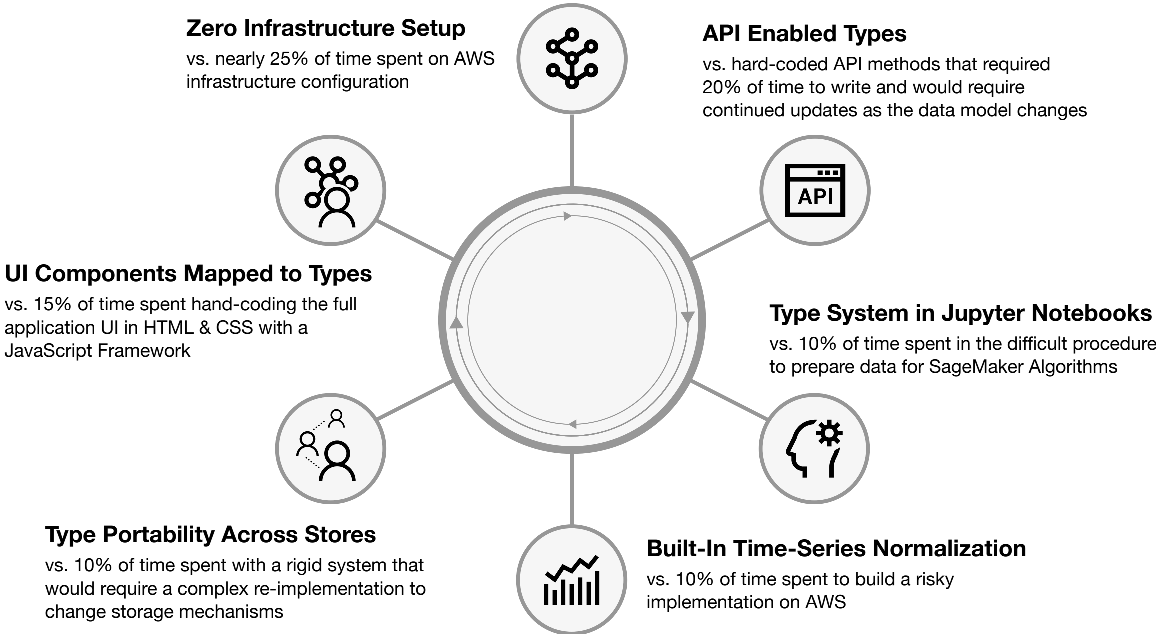
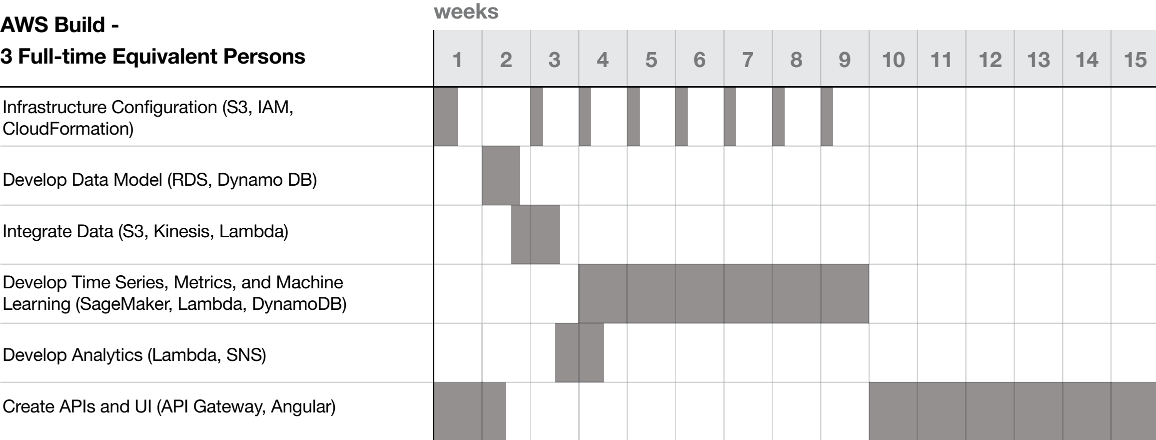


Figure 8

© 2020 C3.ai | All Rights Reserved | 20_0511

AWS + C3.ai and AWS-only Implementation Comparison

As detailed in Figures 9 and 10 below, developing the Application on C3.ai in conjunction with AWS was 26 times faster than on AWS alone.



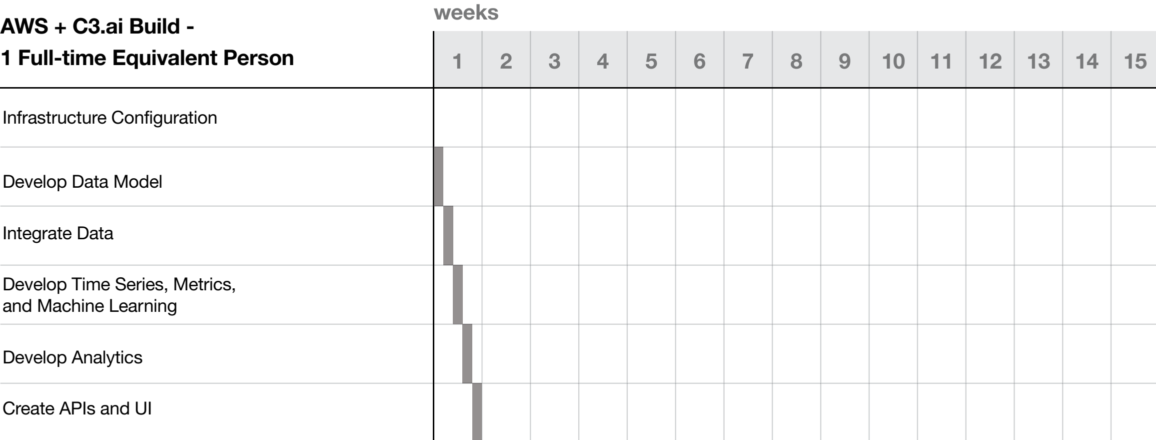


Figure 9

© 2020 C3.ai | All Rights Reserved | 20_0511

Tasks	AWS Application			AWS + C3.ai Application			
	Days	Full-time Equivalent Persons	Effort (Person Days)	Days	Full-time Equivalent Persons	Effort (Person Days)	Savings
Infrastructure Configuration	8.25	3	24.75	0	0	0	100%
Data Model	1	1	1	0.75	1	0.75	25%
Data Integration	3	2	6	0.75	1	0.75	87.5%
Time-Series, Metrics, Machine Learning	14	2.5	35	1.75	1	1.75	95%
Analytics	3	2	6	0.75	1	0.75	87.5%
User Interface and Testing	23	2	46	0.5	1	0.50	98.9%
Total Effort (Person Days)			**118.75**			**4.5**	
Total Lines of Code			**83,000**			**1,450**	

Figure 10

© 2020 C3.ai | All Rights Reserved | 20_0511

Conclusion

In this report, we have described our experience and key findings in using C3.ai in combination with AWS native services, in comparison to using only AWS services, to build a Predictive Maintenance Application for a network of devices. We have documented in the report how using the C3 AI Suite reduced the overall cost and effort required to build the application by a factor of 26 times, while also reducing development risks. The source code required was reduced from 83,000 lines of code to 1,450 lines of code by accelerating AWS development with the C3 AI Suite.

Proven Results in 8-12 Weeks | Visit c3.ai/get-started

© 2020 C3.ai | All Rights Reserved | 20_0511

1300 Seaport Boulevard, Suite 500, Redwood City, CA 94063