



Azure + 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 Azure cloud for Fortune 1000 customers – to develop a Predictive Analytics application for a network of devices, to run on the Azure 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 Azure native services;
2. Build the application using the C3 AI Suite in combination with Azure 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 document as a separate PDF file:

- [Product Specification: Predictive Maintenance Application for a Network of Devices](#)



Table of Contents

Third-Party Report

Executive Summary	4
Findings	4
Background	10
The Azure Native Solution	13
Comparison Tools in Detail	14
Project Narrative	17
C3.ai + Azure Solution	20
The Azure Native Solution	23
Comparative Observations	37
Project Metric Comparison	37
Developer Experience Inputs	40
'ilities in Detail	42
Maintainability	42
Usability	46
Affordability	48
Functionality	49
Interoperability	53
Security	56
Conclusion	57

Third-Party Report by Azure Premier System Integrator

Executive Summary

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

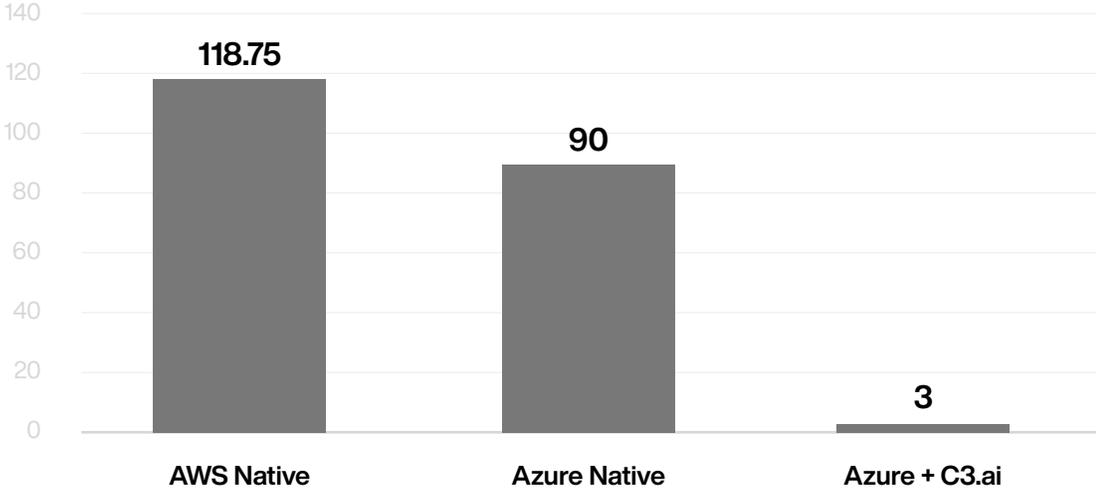
Findings

Three Azure system integrator experienced software engineers (“the team”) built a simple predictive analytics application for AI-enabled devices (“the Application”) on C3.ai’s platform in combination with Azure (“C3.ai + Azure”) and compared it to building a similar application using only Azure native services (“Azure Native”). The team found that building the Application on C3.ai + Azure accelerated development by a factor of 18 times, while reducing effort and risk through its architectural approach. These findings have been determined through a thorough analysis of developer experience metrics and inputs, scoring the two platforms based on third-party system integrator’s ‘ilities framework, and performing a SWOT analysis based on those scores.

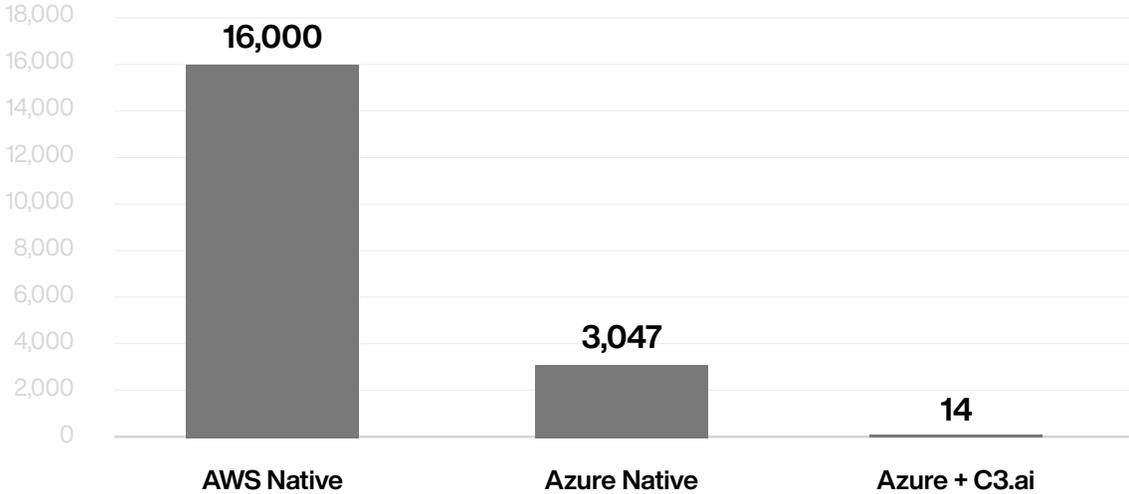
Developer Experience Findings

For the C3.ai + Azure implementation, the team evaluated two approaches: C3.ai Low-Code (i.e., Visual Studio Code in the C3 AI Suite) and C3.ai No-Code (i.e., the C3.ai Integrated Development Studio, or C3.ai IDS). Both the developer experience metrics that were collected, as well as the inputs from the development team, show that C3.ai + Azure required significantly less development time than Azure Native, and is more pleasant to work with overall.

Total Effort (FTE Days)



Lines of Custom Code



Third-Party Report by Azure Premier System Integrator

Below are the time and lines of code benchmarks based on the third party system integrators work for a predictive analytics application.

Metrics	Azure Native Low-Code	C3.ai Low-Code + Azure	Effort Comparison Using C3.ai + Azure
Total Effort (FTE Days)	90 days	5 days	Reduced by 18x
Lines of Custom Code	3,047	822	Reduced by 3.7x

Table 1. Developer Experience Metrics for Azure Native compared to C3.ai Low-Code

Metrics	Azure Native Low-Code	C3.ai No-Code (IDS) + Azure	Effort Comparison Using C3.ai + Azure
Total Effort (FTE Days)	90 days	3 days	Reduced by 30x
Lines of Custom Code	3,047	14	Reduced by 217x

Table 2. Developer Experience Metrics for Azure Native compared to C3.ai No-Code

In a previous engagement, the third-party system integrator was engaged to perform a similar comparative analysis between C3.ai + AWS and AWS Native. Two developer experience metrics were collected:

Total Effort and Lines of Custom Code.

Metrics	AWS Native Low-Code	C3.ai Low-Code + AWS	Effort Comparison Using C3.ai + AWS
Total Effort (FTE Days)	118.75 days	5 days	Reduced by 23.75x
Lines of Custom Code	16,000	822	Reduced by 19.5x

Table 3. Developer Experience Metrics for AWS Native compared to C3.ai Low-Code

Developer Experience Metrics

The team collected two developer experience metrics while building on both the C3.ai + Azure and Azure Native platforms. The metrics were applied both to the C3.ai + Azure Low-Code Solution and to the C3.ai + Azure No-Code Solution, which is also comparable to Azure Native.

C3.ai + Azure Low-Code

Using the C3.ai + Azure Low-Code solution as a basis, the team found that C3.ai + Azure compared favorably in both developer experience metrics. The Low-Code solution was the solution utilized to build the Application.

C3.ai + Azure No-Code (IDS)

The team was also asked to compare C3.ai + Azure No-Code (IDS), to provide input on C3.ai's configuration-based solution. After walkthroughs and consulting with C3.ai's expert solution architects, the team found that C3.ai + Azure No-Code also provided a better experience for predictive analytics application development than Azure Native in both developer experience metrics.

For detailed information regarding the metrics used, see [Comparison Tools in Detail](#).

1. **Time** – Using C3.ai + Azure, building the Application took a single developer 1 week. Implementing the same solution on Azure Native took three developers 6 weeks to complete. Developing on the C3.ai Platform was 18x faster than using Azure Native services.

2. **Lines of Code** – Developing the Application on C3.ai in IDS eliminated the need for code except for a single custom function that consisted of 14 lines of code. Using Azure Native, the majority of data transformation tasks required custom code, and the team ended up writing 3,047 lines of code. Leveraging C3.ai + Azure decreased the lines of code written by a factor of 217x which contributed significantly to the reduction in time.

'ilities Findings

The 'ilities Framework is a comparative analysis tool used by the team to objectively evaluate and compare solutions and platforms. When building the predictive analytics application across both C3.ai + Azure and Azure Native, the team focused their feedback through the lens of the 'ilities Framework to provide a standardized methodology for their evaluations.

The 'ilities include the following measures:

1. Maintainability
2. Flexibility
3. Scalability
4. Affordability
5. Usability
6. Functionality
7. Interoperability
8. Security

When evaluating the C3 AI Suite across the 'ilities Framework, the team utilized both qualitative and numeric metrics for the comparison. The numeric metrics are:

1. Does Not Meet Expectations
2. Somewhat Meets Expectations
3. Meets Expectations
4. Exceeds Expectations
5. Exceptional Performance

C3.ai + Azure compared favorably with Azure Native across almost all of the 'ility dimensions. The team found that C3.ai + Azure scored very high on **Maintainability**, which includes **Flexibility**

and Scalability, and also was clearly superior to Azure in **Affordability** due to reduction in TCO and quicker time to value. Additionally, the simplicity and ease of use of C3.ai + Azure garnered high **Usability** ratings, and the **Functionality** of the C3 AI Suite was viewed as close to Exceptional. Finally, C3.ai + Azure exceeded expectations in comparison with Azure Native when considering integrations and visualizations in the area of **Interoperability**.

C3.ai + Azure and Azure Native had similar ratings with regard to **Security**, because the protocols and underlying infrastructure are not differentiated.

C3.ai + Azure vs. Azure Native

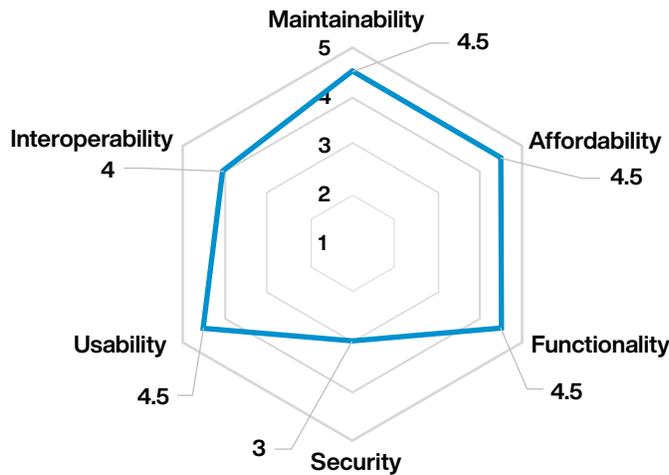


Figure 1. 'ilities Scores for C3.ai + Azure in Comparison to Azure Native

For detailed descriptions of each 'ility, see [The 'ilities Framework](#).

SWOT Analysis

Based on the developer experience metrics and the ‘ilities Framework analysis, the team has used the SWOT Framework to characterize C3.ai + Azure in comparison with Azure Native. The 2x2 grid below shows how each characteristic of the C3.ai + Azure platform fits into the analysis.

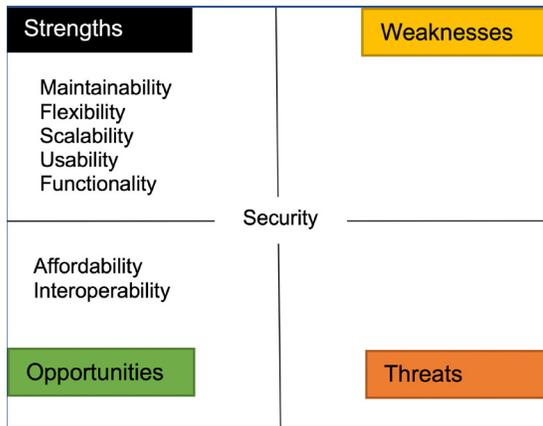


Figure 2. SWOT Analysis

Strengths (Internally Facing)

The C3.ai + Azure platform provides superior **Maintainability, Flexibility, and Scalability**. Due to its elegant architecture, developers can quickly adjust data models and data sources to adapt to changing use cases and requirements without needing to touch underlying infrastructure. Additionally, the software makes it extremely easy to add capacity as needed to scale to any load.

Across **Usability**, the simplicity of the platform provides an interface that creates an excellent developer experience.

In the area of **Functionality**, the team found that C3.ai + Azure scored very high – near to Exceptional – compared to Azure Native. As a pure AI platform, all the required functions are centrally located, which creates an outstanding ease of operations for developers.

Weaknesses (Internally Facing)

When evaluating the C3.ai + Azure platform in comparison to Azure Native, the team did not find any weaknesses.

Opportunities (Externally Facing)

Compared with Azure Native, C3.ai + Azure shines with outstanding Affordability. The platform's ability to build, scale and maintain the platform much more quickly than Azure Native will provide an exponential improvement in Total Cost of Ownership, and is a significant differentiator in the marketplace. The team also found that C3.ai + Azure exceeded expectations with regard to **Interoperability**, in comparison to Azure Native. C3.ai's model-driven architecture creates an ease of use for integrations and visualizations that provides developers and clients with a differentiated experience.

Threats (Externally Facing)

When evaluating the C3.ai + Azure platform in comparison to Azure Native, the team did not find any threats.

Neutral (No Advantage or Disadvantage)

For Security, C3.ai + Azure leverages a Virtual Private Cloud and various other data and network security layers. This is similar to Azure Native's solution, and the team did not find any significant differences in this area.

Background

The C3 AI Suite is a versatile complement of applications that are used to perform artificial intelligence functions such as predictive analytics, supply chain and inventory optimization, fraud detection, and maintenance operations. The C3 AI Suite partners seamlessly with all three major cloud service providers: Azure, AWS, and Google Cloud Platform.

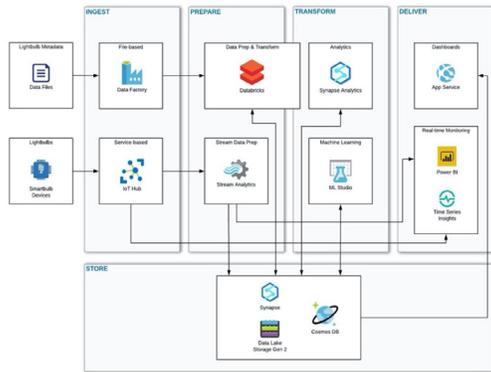
While these same functions can be developed natively on cloud platforms such as Azure, C3.ai is confident that its architectural approach to the C3 AI Suite applications provides a better developer experience and packaged costs for their clients.

Why the Architecture Matters

C3.ai has built the C3 AI Suite to be an accelerator for industries leveraging AI/ML and IoT to solve complex problems at scale. The third-party system integrator (the "team") built a predictive analytics application for AI-enabled devices using the purpose-built C3 AI Suite platform in combination with Azure, and found that its model-based architecture drove significant improvements when compared to the structured programming approach taken on Azure alone.

Visually, the simplicity of the C3 AI Suite on Azure ("C3.ai + Azure") architecture, versus the architecture of the predictive analytics application on Azure alone ("Azure Native"), speaks for itself.

Predictive Analytics on Azure Native



Predictive Analytics on C3.ai + Azure

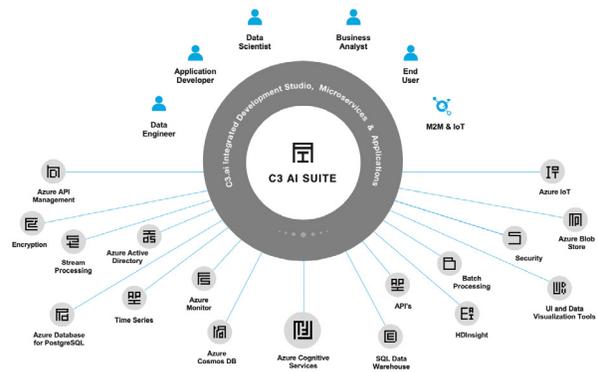


Figure 3. Architectural Comparison between Azure Native and C3.ai + Azure

From an industry perspective, there are benefits and risks to C3.ai’s model-driven architectural approach, both for IT and for the business. C3.ai has consistently found that the benefits outweigh the risks, particularly with regard to total cost of ownership and time to value.

Approach	Total Cost of Ownership (“TCO”)	Time to Value
<p>Model-Driven Architecture C3.ai + Azure</p>	<p>⬇️ Decreased</p> <ul style="list-style-type: none"> Application build can be accomplished by one developer Application maintenance is done only on the application, not on its underlying infrastructure Application scales enterprise-wide, managing millions of models through a single instance 	<p>⬇️ Decreased</p> <ul style="list-style-type: none"> Developers ramp up quickly on each application, making them productive quickly Valuable new features can be produced quickly once the core application is built

Approach	Total Cost of Ownership ("TCO")	Time to Value
Structured Programming Azure	<p>⬆️ Increased</p> <ul style="list-style-type: none"> Application build requires multiple developers Application maintenance must be performed both on the application and its underlying infrastructure, causing expensive dependencies to accrue and multiply Scales across the enterprise, but requires management of hundreds of AI projects to utilize the same data 	<p>⬆️ Increased</p> <ul style="list-style-type: none"> Developers' training and design time for each custom application slow productivity Building new features may require a rebuild of the entire core, slowing time to value

Table 4. Model-Driven vs. Structured Programming Advantages

To test this belief, C3.ai engaged a premier third-party enterprise systems integrator, to build a predictive analytics application for a network of devices ("Application") on both the C3.ai + Azure and Azure Native platforms. This report details and compares the developer experience of the team during these buildouts.

The Azure Native Solution

The team built a predictive analytics application on Azure that is both in accordance with Microsoft’s reference architecture, and indicative of what C3.ai’s customers would build for such a use case.

High-Level Architecture

The architecture utilizes out-of-the-box Azure products to accomplish the following functions:

1. Ingest both initial seed data and streaming data;
2. Determine alerts and predict metrics;
3. Store input data, intermediate data, and results; and
4. Present the data for analysis.

Azure Data Factory and IoT Hub ingest seed and real-time data. Data are persisted at various times throughout the process into different

storage resources depending on the structure and availability requirements of the data. Hot, structured data are stored in Synapse, a data warehouse solution in Azure; hot, unstructured data are stored in Cosmos DB, a NoSQL data store; and cold data are stored in Azure Data Lake Storage Gen2 (hereafter referred to as “Azure Data Lake”), a data lake solution. Databricks is used to enhance and transform the data. After the machine learning features are prepared, Azure ML Studio is used to train the machine learning model and deploy it to an endpoint so that predictions can be generated for streaming data. Finally, all the device data are displayed as interactive visualizations on Power BI, Microsoft’s business analytics dashboarding service.

The below diagram shows this high-level architecture:

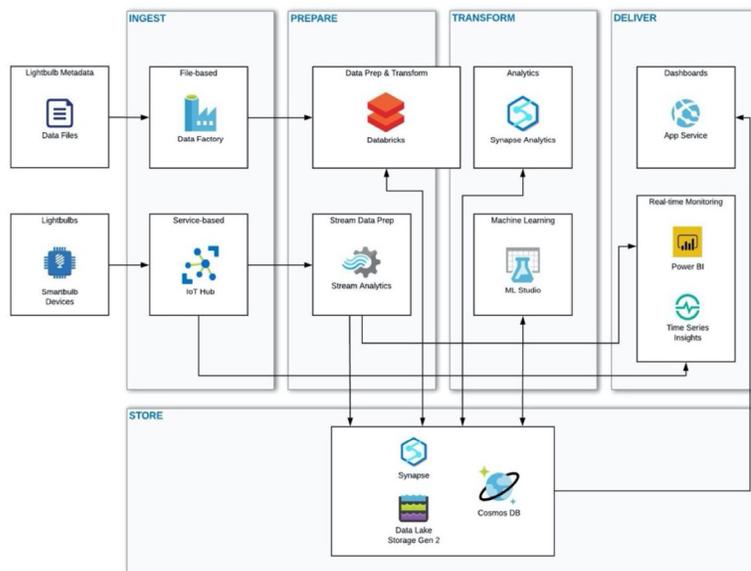


Figure 3. Azure Native Architecture

Comparison Tools in Detail

Developer Experience Metrics and Inputs

The team tracked three metrics to measure the complexity of developer experience: amount of time to develop the entire Application, amount of time to develop key pieces of the Application, and lines of code to develop the Application.

Metric	Task	Measurement	Level tracked
Amount of Time	Developing the entire application with one FTE	Days	Project Level
Amount of Time	Developing key pieces of the Application	Hours	Task Level; added and rolled up to Epic Level
Lines of Code	Any customizations where configuration is no longer usable and code is needed to achieve parity	Integer	Project Level

Table 5. Comparison Tools in Detail

To further understand and document the complexity of developer experience on each platform, the team captured screen shots of configurable features and capabilities at the user story level.

These developer experience metrics and inputs supplied critical data points for the differentiators C3.ai evaluated in this report.

The 'ilities Framework

In addition to the developer experience metrics, the team leveraged their proprietary 'ilities Framework to capture other, more complex aspects of the comparison. The 'ilities Framework describes a solution (the Application on C3.ai + Azure) by determining how well it compares to a similar solution (the Application on Azure Native) across eight factors: Functionality, Usability, Affordability, Maintainability, Flexibility, Scalability, Interoperability, and Security.

	Criteria	Description	Example Sub-Criteria		
Business	1. Functionality	Solution's ability to deliver its required capabilities and meet the business needs	<ul style="list-style-type: none"> • Specific features • Reporting 	<ul style="list-style-type: none"> • Specific requirements 	<ul style="list-style-type: none"> • Error handling
	2. Usability	User's productivity when working with the solution	<ul style="list-style-type: none"> • Assistance • Learnable 	<ul style="list-style-type: none"> • Modular • Productive 	<ul style="list-style-type: none"> • Structured
	3. Affordability	Solution's overall cost including acquisition and on-going maintenance	<ul style="list-style-type: none"> • Hardware costs • Licensing costs 	<ul style="list-style-type: none"> • Implementation costs 	<ul style="list-style-type: none"> • Support costs • Training costs
Technical	4. Maintainability	Level of effort required to keep solution running while in production including problem resolution and ongoing support	<ul style="list-style-type: none"> • Manageable • Operable 	<ul style="list-style-type: none"> • Recoverable • Analyzable 	<ul style="list-style-type: none"> • Testable • Upgradeable
	5. Flexibility	Solution's ability to accommodate additional business processes or changes in functionality	<ul style="list-style-type: none"> • Adaptable • Configurable 	<ul style="list-style-type: none"> • Maneuverable 	<ul style="list-style-type: none"> • Modifiable
	6. Scalability	Solution's ability to support additional users while meeting quality of service goals	<ul style="list-style-type: none"> • Capacity • Throughput 	<ul style="list-style-type: none"> • Resource utilization • Response time 	<ul style="list-style-type: none"> • Reliability
	7. Interoperability	Solution's ability to interact effectively with other systems or components	<ul style="list-style-type: none"> • Integration protocol 	<ul style="list-style-type: none"> • Loosely coupled • Tiered 	<ul style="list-style-type: none"> • Legislative compliance
	8. Security	Solution's ability to prevent unauthorized disclosure, loss, modification or use of its data or functionality	<ul style="list-style-type: none"> • Access Control • Encryption 	<ul style="list-style-type: none"> • Secure design • Auditability 	<ul style="list-style-type: none"> • Authentication

Table 6. Theilities Framework

Updating the SWOT Analysis

Finally, bringing in an overall platform perspective inclusive of an industry view, the team has provided a Strengths, Weaknesses, Opportunities, and Threats (SWOT) Analysis comparing C3.ai + Azure to Azure Native. To remain consistent and avoid duplicating work, the SWOT Analysis from the previous report on AWS was brought in and updated.

The SWOT Analysis includes:

- Platform Strengths (Internally Facing to Users)
- Platform Weaknesses (Internally Facing to Users)
- Platform Opportunities (Externally Facing to Market)
- Platform Threats (Externally Facing to Market)

Project Narrative

The team compared building a simple predictive analytics application (the “Application”) using native Azure services (“Azure Native”) to using the C3 AI Suite built on Azure (“C3.ai + Azure”). In both cases, the team sought to ingest, unify, and federate the raw data, process it, train a machine learning model that predicts the likelihood of failure within the next 30 days for each device, and build an application user interface.

The provided data sets for the Application included:

- Device type, wattage, location, manufacturer, and date of manufacture
- Power grid status
- Device fixture location
- Device telemetry including watts, lumens, voltage, and temperature
- Device event history
- Device fixture 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:

- Average Lumens per Smart Device – Light generation over time for the smart device
- Average Power per Smart Device – Power usage over time for the smart device
- Duration On per Smart Device – The total amount of time (in hours) that a device has been switched on up to the interval
- Switch Count per Smart Device – The number of times a device is switched 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

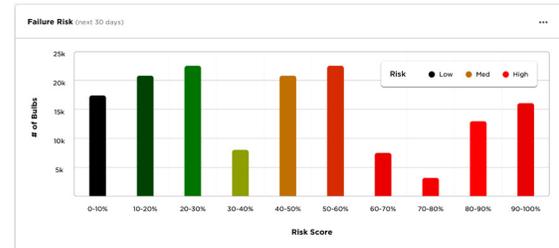
The target variable is to predict the likelihood of device failure within the next 30 days from a given point in time. It is left up to the development team how to make this prediction, although the industry best practice is to train a machine learning model using the provided data. With this predictive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

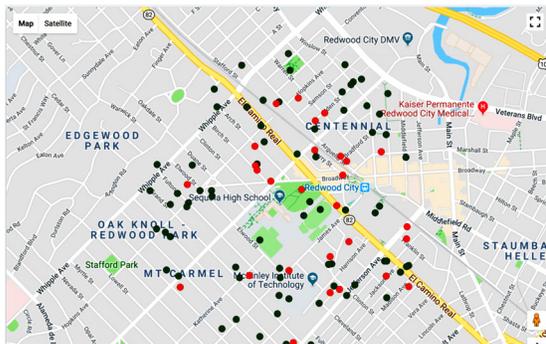
1. A summary of key metrics including the total number of devices at risk, the total number of devices, and the number of failures YTD

Key Metrics		
LIGHTBULBS AT RISK 63,281	TOTAL LIGHTBULBS 1,000,000	FAILURES (YTD) 285

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

Bulb ID	Risk Score %	Bulb Type	Manufacturer	Date Started
SHBLB1	80	LED	LIFX	10/01/2017 4:00:00 AM
SHBLB2	80	LED	eufy	10/01/2017 4:00:00 AM
SHBLB3	80	LED	Cree	10/01/2017 4:00:00 AM
SHBLB4	70	LED	LIFX	10/01/2017 4:00:00 AM
SHBLB5	70	LED	GE	10/01/2017 4:00:00 AM
SHBLB6	70	LED	Cree	10/01/2017 4:00:00 AM
SHBLB7	70	LED	GE	10/01/2017 4:00:00 AM
SHBLB8	70	LED	GE	10/01/2017 4:00:00 AM
SHBLB9	60	LED	eufy	10/01/2017 4:00:00 AM
SHBLB10	60	LED	Philips	10/01/2017 4:00:00 AM

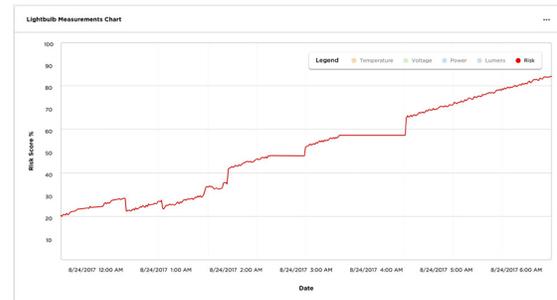
Figure 4: UI Screen 1 – Four displays showing the status and health of the entire population of devices.

All elements on the screen can be filtered by multiple dimensions. Selecting an individual device from the table presents a device detail screen, with details as seen below:

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



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

Timestamp	Risk Score	Status	Lumens	Voltage	Power	Temperature
2017/08/24 0:00:00 AM	20	On	700	115	15	150
2017/08/24 0:59:44 AM	25	On	700	115	15	150
2017/08/24 1:59:42 AM	30	On	700	115	15	150
2017/08/24 2:58:21 AM	45	On	700	115	15	150
2017/08/24 3:58:15 AM	55	On	700	115	15	150
2017/08/24 4:58:52 AM	65	On	700	115	15	150
2017/08/24 5:56:43 AM	75	On	700	115	15	150
2017/08/24 5:54:15 AM	80	On	700	115	15	150
2017/08/24 5:54:05 AM	85	On	700	115	15	150
2017/08/24 5:53:00 AM	85	On	700	115	15	150

Figure 5: 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)

C3.ai + Azure Solution

Developing the Application with the C3 AI Suite on Azure was uncomplicated. Learning the C3.ai Type System and building the Application required five days of training.

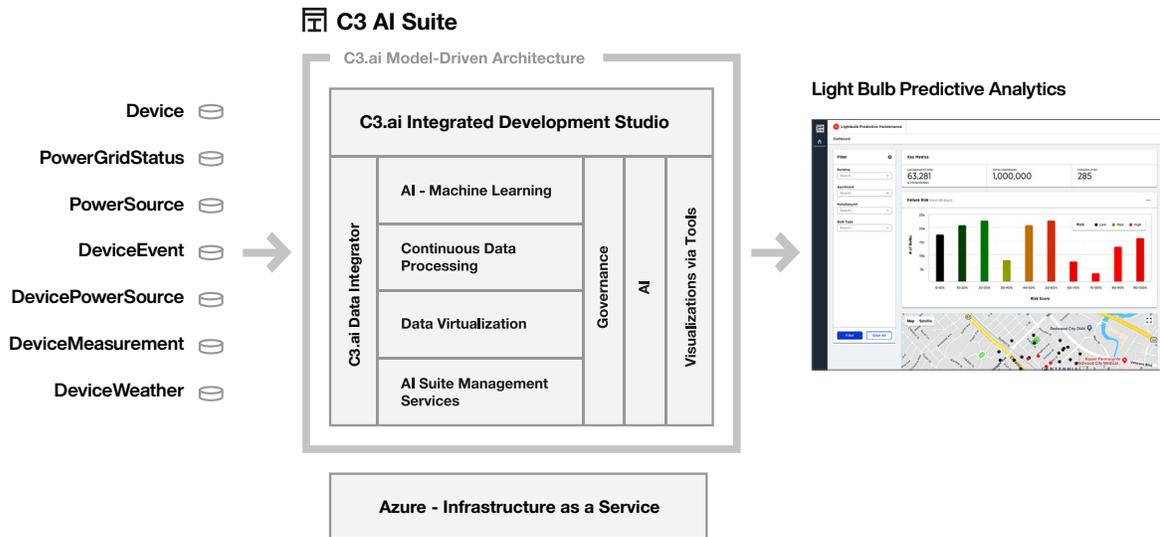


Figure 6: Architecture to build predictive analytics on C3.ai + Azure.

Infrastructure Configuration

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

Data Model

The team began building the Application by creating C3.ai Types to comprise a unified data model. Types are code representations of real-world objects 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 developer.

Data Integration

Next, the team used the C3 AI Suite's native data integration capabilities to integrate, index, and normalize the device data. Prior to integrating data, a Canonical Type was defined for each of the six data sources. The C3 AI Suite includes native functionality to import data from any source – while the team worked with CSV files, C3.ai includes pre-built connectors to commonly-used relational databases, NoSQL databases, and distributed file systems. All fields on Canonical Types are mapped to a data source to define the incoming data model which de-couples the Types used by data scientists and developers from external changes. Integrating data required six hours and one developer.

Time Series Metrics

The team then used C3.ai Types to generate 13 time series metrics, which fetch C3.ai Type data to produce a normalized time series. Initially, the team was introduced to Simple Metrics which are useful for common time-series manipulation requests (sum, average, min, max, etc.). Next, the team began developing Compound Metrics, built as extensions of Simple Metrics, to be incorporated into application logic and serve as features in the machine learning development process. The team also wrote additional methods for the SmartDevice Type which allow for more complex calculations on the data using JavaScript or Python. Creating the 13 metrics took eight hours for one developer to complete.

Analytics

Next, the team used C3.ai's native, asynchronous processing engine to create data flow events (DFEs). Using DFEs, the team 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 took one developer six hours to complete.

Machine Learning

The team created risk-of-failure scores for the Application using Jupyter Notebooks and Python, both supported natively by C3.ai. By having the full functionality of the C3.ai and C3.ai Type Systems natively integrated with Jupyter Notebooks, data scientists are provided easy access to leverage familiar tools and efficiently develop solutions. A classification model was trained that regressed the metrics SwitchCountWeek and DurationOnInHours against the dependent variable WillFailNextMonth to calculate the probability of device failure in the next 30 days. Device failure was determined by instants where the device status and related power grid status were “on” with a lumen reading of “0”. The system stored the periodically generated risk scores as another time series metric, RiskScore. Machine learning algorithms on C3.ai operate on all existing data, create new data that can be automatically attached to a C3.ai Type for future processing, automatically update training, and make predictions on the latest available data. For the Application, the area under the receiver operating characteristic (ROC) curve was .990. Training the machine learning model and the machine learning pipeline took one developer six hours to complete.

User Interface

The team incorporated several C3.ai Types and time series metrics in a web interface built using custom C3.ai HTML and UI templates. These were then used to create the dashboard of the Application. The dashboard UI template was one JSON file that contained the code for the components of the dashboard such as a status map, a filter, a histogram and a table. The UI also included automatically updated predictive risk scores about the likelihood of smart device failures (incorporated using the RiskScore metric).

Finally, a few potential roles were created that would be assigned to future users of the Application. These roles enable administrators to restrict user permissions for specific needs of different user types. Building the UI and configuring access controls took one developer four hours to complete.

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



Figure 7. Predictive Analytics Application on C3.ai + Azure

The Azure Native Solution

The architecture for the Azure Native application made heavy use of out-of-the-box Azure-managed services, including Azure IOT Event Hubs for data ingestion; Azure Synapse Data Warehouse, Cosmos DB, and Azure Data Lake for data persistence; Azure Databricks for data transformation; Azure ML Studio for machine learning training and inference, as well as Microsoft Power BI for data visualization. This architecture stems from our collective years of experience working with Azure services at a deep level – our firm is a Gold-level Microsoft consulting partner with competencies in Cloud Platform, Application Development, Data Platform, and Data Analytics. We have developed and deployed hundreds of applications on Azure for hundreds of Fortune 2000 customers.

At the onset of the project, the team agreed to use as many out-of-the-box native features as possible, leveraging pre-built Azure components and only writing custom code and queries where the built-in features were too limited to provide the requisite functionality.

The Azure architecture used for building out the Application is based on a reference architecture by Microsoft, which includes Databricks on Azure. It has been thoroughly reviewed by C3.ai senior architects to ensure the architecture is representative of how a C3.ai customer would approach setting up their own predictive analytics application.

Developer experience metrics were tracked while developing the Application, including time required to complete the entire project and time required to develop each high-level component. Where code was utilized, the team tracked the number of lines of code as well.

The architecture diagram below displays the overall Azure architecture and how the various components interact to meet the Application specification.

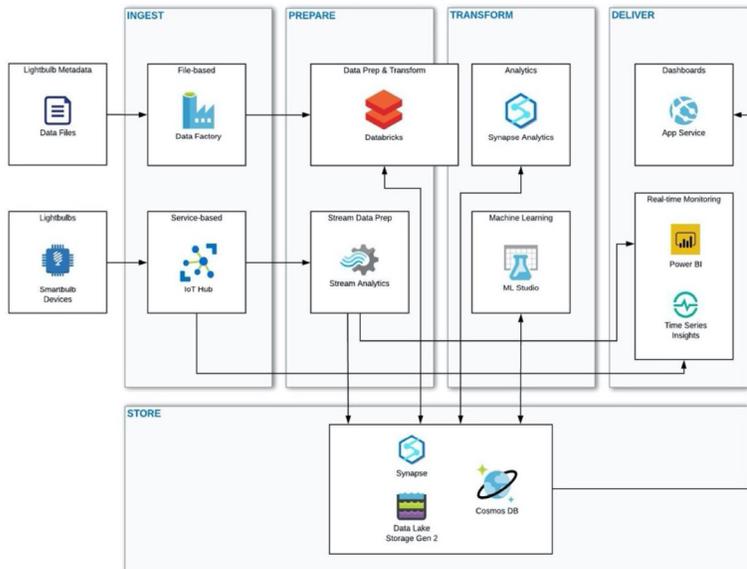


Figure 8. Azure Native Architecture

Below we describe the buildout of the Azure solution in detail, starting with the infrastructure configuration and ending with displaying the results using PowerBI, an Azure native visualization tool.

Infrastructure Configuration

In order to parallelize environment setup and minimize developer conflicts, the team divided the infrastructure tasks into the following categories: basic access, ingestion, storage, transformation, machine learning, and delivery. The work began by creating a new Azure subscription and configuring basic access control and security policy permissions. Once the subscription was configured, the team was able to begin provisioning the services in Azure needed to build the Application.

Initially, the team created resources to support the ingestion process for historical and streaming

data – Azure IoT Hub, Azure Stream Analytics, and Azure Data Factory. After the team had the services necessary to ingest data, Azure data stores were provisioned to persist data with different structure and availability needs – Azure Cosmos DB (unstructured-hot), Azure Synapse (structured-hot), and Azure Data Lake (cold). Next, the team set up the services in Azure needed to transform device data and perform machine learning tasks – Azure Databricks and Azure Machine Learning Workspace. In a newly created subscription, Azure Databricks is limited by the default compute quota. In order to remove that restriction, the team opened a ticket with Microsoft to increase the compute quota and the issue was resolved in less than one day. Finally, the team provisioned a Power BI instance to deploy dashboards for end users to derive actionable business intelligence from streaming data and risk predictions.

Deploying resources in Azure can be done via the Azure Portal or using the Resource Manager API. The team leveraged the no-code option of deploying resources via the Azure Portal. Modifying existing resource configurations is a manual process that can also be done via the Azure Portal with no code. Maintaining the underlying infrastructure is handled by Azure and requires no effort from the development team.

Leveraging the Azure Portal, the team set up the infrastructure in 5 developer days.

Data Model

The team built the Data Model in accordance with the Canonicals in C3.ai. To optimize the different forms of incoming data for read and write efficiency, time series data were loaded into CosmosDB as JSON objects and non-time series data were stored as tables in Synapse. The schemas of the Synapse tables were very close to those of the files provided by C3.ai, apart from the column names. For example, the column representing “Smart Device ID” has varying names in each of the seed files (e.g., “id”, “smartDeviceId”, “SN”). These differences in the incoming data were normalized when loaded into the data warehouse.

Structured Data in Synapse

To create the Synapse tables, the team first uploaded the raw CSVs into a blob container in Azure Data Lake. A connection from Synapse was then created, registering the blob container as a Synapse Datastore. Next, SQL queries were written to create an external table on each of the seed files.

From these external tables, the team created the working tables through SQL queries. Values from the external tables that defaulted to string values were converted into integer, decimal, and datetime types where appropriate. Column names and value formats were corrected to be consistent across tables. Primary keys and indexes were added to optimize table operations. While the process of creating internal tables from external tables could also be used for some data cleaning operations such as omitting null values or only taking values within a certain range, the team did not need to build in that functionality because the data contained no null values or other data anomalies that required cleaning.

The team wrote approximately 300 lines of SQL to connect to Azure Data Lake and create the external and internal tables in Synapse. Modeling the relational data in Synapse took a single developer 7 days.

Unstructured Data in Cosmos DB

Configuring Cosmos DB to handle unstructured device data primarily consisted of determining the appropriate partitioning strategy. Cosmos DB is a highly available, distributed data store that works best when partitions are created to reflect natural divisions in the data being stored. The team decided to partition the device data by SmartDeviceId, optimizing for the common use case of querying measurement data pertaining to a specific device.

Modeling our key-value data in Cosmos DB took a single developer 3 days.

Data Integration

The team defined data integration as the process of combining data from different sources into a single platform. Integration begins with an ingestion process and includes any subsequent steps necessary to model, enhance, and transform the data. Ultimately, a successful data integration pipeline enables the delivery component of a solution to provide actionable business intelligence.

The team began integration by importing seed data into the environment. First, the provided flat files with raw device data were uploaded into Azure Data Lake. Next, structured data was imported into Synapse and unstructured data into Cosmos DB. For Synapse, the structured flat files were referenced as external tables and used to create internal tables that would hold metadata about each entity (e.g., Smart Device, Fixture, Building). For Cosmos DB, the hourly time series data was imported through the Azure Stream Analytics Job that was developed to handle streaming data. Leveraging Stream Analytics allowed the team to test the job and ensure consistency between processing historical and live data. All seed data are retained in Azure Data Lake for reference, debugging, and creation of new environments in the future.

Ingesting the seed data took the team 3 developer days.

Streaming data enter the Azure Native environment via Azure IoT Hub endpoints. Every hour, a file is sent to the IoT instance with simulated device data which is processed by the Stream Analytics job tested on the seed data. Once

processed, the data are persisted in Cosmos DB and Azure Data Lake.

Configuring the IoT Hub to ingest streaming data took a single developer 2 days.

After streaming data are persisted in Azure Data Lake, the Azure Data Factory pipeline that orchestrates the machine learning process is triggered. The pipeline will pass the file path location of new data to a Jupyter Notebook in Azure Databricks, which is responsible for generating features and retrieving predictions. The predictive model was developed in Azure ML Studio and hosted on an Azure Kubernetes Cluster.

Setting up an Azure Data Factory pipeline to process data and connecting the Jupyter Notebook to the predictive endpoint took a combined effort of 20 developer days.

Time Series Metrics and Analytics

Stream Analytics

Streaming data enter Azure via IoT Hub and are immediately passed through Stream Analytics. The team configured simple Stream Analytics Jobs to format the incoming data in ways that suited downstream processes – specifically, preparation for machine learning operations.

Creating a new job requires minimal code (~20 lines), but still took a single developer half a day to complete. Most of the effort consisted of testing transformations against incoming data and ensuring stability of downstream processes.

The team attempted to do more complex transformations of streaming measurements to enhance the data earlier on in the pipeline and reduce the workload of downstream processes. For example, logic was developed to calculate the Metric “SwitchCountPreviousWeek” using a SQL-like windowing function. Unfortunately, Stream Analytics could only handle minor gaps in the data, which was not sufficient for the predictive analytics use case. The decision was made to use Databricks for all complex transformation logic moving forward.

Databricks

Once the seed data was ingested into the data stores, the team began developing the application metrics in Databricks. Azure Databricks is an Apache Spark based platform optimized for big data analytics on services in the Azure platform. Developing the metrics in Databricks consisted of writing custom Python logic in Jupyter Notebooks that would retrieve the appropriate time-series input and generate the expected time-series output. Implementing the required analytics involved writing additional custom Python logic that leveraged existing metric functions to generate alerts. In order to accelerate the development process, the team leveraged pandas, a popular open-source tool for data analysis and manipulation that is included by default in a Databricks Python environment. **Pandas** offers data structures and operations for manipulating time-series data and integrates well with the Python Spark library, **pyspark**, to maximize the benefits of Spark distributed processing capabilities.

Using pandas, a single developer was able to implement 13 time series metrics and 3 analytics in 6 days.

Implementing custom Python logic in Databricks Jupyter Notebooks would be risky and difficult to maintain in a production environment. While Databricks does offer integrations with version control tools, the team found it difficult to manage when Notebooks were connected to other Azure services. Additionally, writing complex logic in Jupyter Notebooks is not considered best practice and introduces a significant challenge for testing and code quality assurance. To mitigate these risks, the team decided to export the metric logic into a custom Python package, **pybulb**, that could be maintained in the repository.

Exporting the code, writing unit tests, and integrating the package into the data flow took the same developer another 10 days.

Machine Learning

Databricks

The team used Databricks for the initial machine learning pipeline. The pipeline includes ingesting data from an outside source, using time series metrics to extract features for ML model development, training and testing different ML models, and then deploying an inference pipeline to generate predictions from streaming data.

In Databricks, the team used a set of Python classes and methods in a Jupyter notebook to integrate metrics, as described above. These were used in Databricks to generate a Pandas dataframe with the solution’s ML features. After

exploring these data, the team found that the number of “true” values for “WillFailNextMonth” was less than 2% of the total number of samples. Therefore, the oversampling method SMOTE was utilized to balance the training data. These data were used to train a logistic regression model and validated with the testing data.

The team generated statistics to evaluate the logistic regression model, such as accuracy, precision, recall, and AUC. ML Flow, a ML Lifecycle management platform, was utilized both to track iterations of the ML model, save metrics and artifacts related to each run, and to save the model.

The team used a separate notebook for the inference pipeline. This notebook is triggered when new streaming data are added into Azure Data Lake to process the streaming data. Risk scores are generated for each streaming device and written to Azure Data Lake. Alerts are generated when metadata of streaming devices meet a specific condition. For example, when a device has a temperature of over 95 degrees, an Overheat alert is generated. Such alerts are then written to a table in Synapse.

In total, the team wrote approximately 500 lines of code in Databricks to develop the machine learning model and inference pipeline. This took two developers a total of 6 days to complete.

Azure ML

Model Training, Testing and Deployment

Many C3.ai customers prefer to use Azure ML over Databricks for their production ML pipelines. To reflect this use case, the team replaced the model training, testing, and deployment in the initial Databricks implementation with an Azure ML pipeline.

Compared to Databricks, Azure ML connected seamlessly to Azure storage resources. Datasets were easily generated from files and tables that existed in the Synapse tables and blob storage. Once the ML features were prepped and loaded into Azure ML, it was very simple to train data, score models, and evaluate results with Azure’s click-and-drag Designer interface. Machine learning and data manipulation processes were represented by modules that can be easily connected to indicate data flow.

Using the same training and testing data, the team evaluated 6 different two-class classification algorithms in much less time than it would have taken to do the same in Databricks. In the Azure solution, 250 lines of code were re-used from the Databricks solution to generate ML Features and 100 lines were re-used to write predictions and alerts to the backend. Approximately 50 additional lines of code were written in Databricks to handle Azure ML predictions.

Setting up data in Azure ML, comparing six classification algorithms, deploying our model as an endpoint, and writing code to access that endpoint in Databricks took 6 developer days.

Feature Engineering

The team re-used the Databricks code for feature engineering because Azure ML's Designer requires near-ready ML feature data sets. The team found the data transformation capabilities of Designer to be a bit lacking when compared to the wide-ranging capabilities of the ML frameworks available on Databricks (e.g., scikit-learn, TensorFlow, PyTorch, H2O). It handled simple transformations, such as deleting duplicates, filling in missing values, and joining tables very easily with its click-and-drag interface. More complex transformations required a SQLite Query module or Python Script module to execute.

To mirror the metrics functionality in C3.ai, the team used a series of classes and functions in Databricks to generate features. When moving to Azure ML Studio, in comparison to using Notebooks in Databricks, it was noted that the coding experience in the Python Script module was not robust. There is no way to execute code in different blocks and view output, and everything must be defined in a single function. In addition, the entire ML pipeline must be run to execute the Python script module, and any errors other than syntax errors did not have line numbers, only the name of the exception. For data sets that require complex transformations for feature engineering, a more robust coding environment would be recommended.

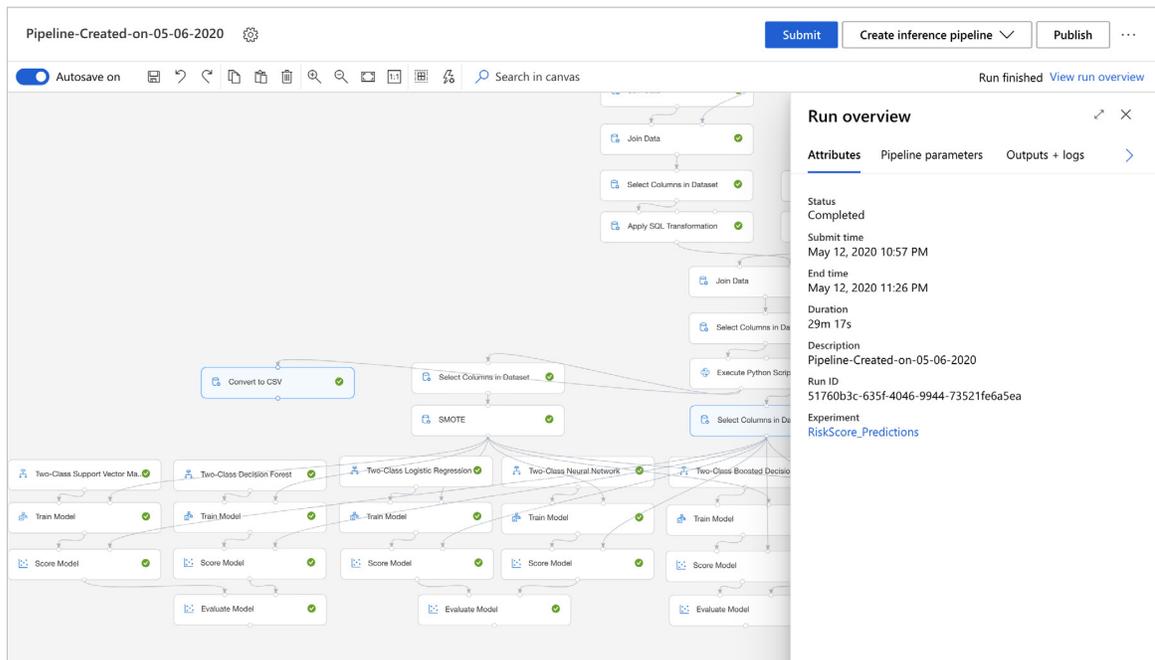


Figure 9. Azure ML Designer Interface: Training, testing and evaluating six classification algorithms on data oversampled for WillFailNextMonth.

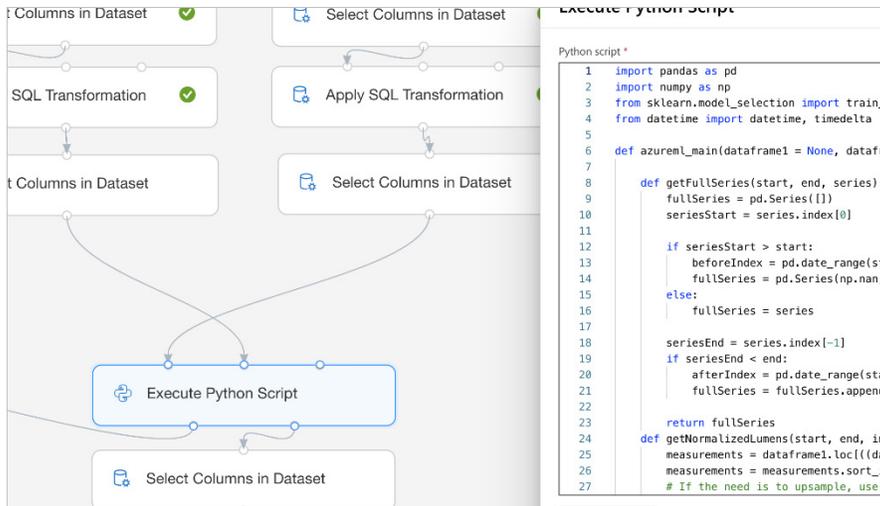


Figure 10. Python Editor in Azure ML Designer: Basic text editor without Intellisense, syntax correction, or debugging functionality.

User Interface

Power BI was used to implement the Application user interface. The UI was built in Power BI Desktop and then published to the Power BI Service where it could be distributed. The original plan was to leverage Power BI only to build the data visuals needed for the UI. Those visuals would then be embedded in an Angular application hosted in Azure. However, Power BI has the capabilities needed to build entire UI in a single two-page report and so the Angular app was removed in favor of this simpler approach. The steps required to create the Application UI as a Power BI report can be condensed into four categories: Connect, Transform, Model, and Visualize. These categories are described in detail below.

Connect

The first step to implement the report was to connect to data sources in Synapse and Databricks. This was a trivial task since each of these data sources have native data connectors within Power BI. Synapse can be accessed as a SQL Server database and Databricks' underlying data store is an Apache Spark database. The required connection strings are available in the Azure Portal.

There are three mechanisms available for bringing data into Power BI: Import, Direct Query, and Streaming. Import, as the name implies, involves importing the full data set from the source database into Power BI memory. Conversely, Direct Query will create a connection to the source database and pull data as needed based on transformations and filters. Finally, Streaming can be used to send live data directly to Power BI for storage and use. Generally, streaming will be used in parallel with Import. The initial data set will be imported, and live data will be streamed in using tools such as Azure Stream Analytics. The Application report currently uses Import for all data sourced in Azure. There is technically a fourth mechanism known as Custom Data, which is defined to hold raw reference data stored within Power BI and not an external data store. This will be discussed in more detail in the Transform section since it is not used to connect to an external database.

It took less than half an hour for each connection depending on the research required to find the appropriate connection strings. This is a no-code solution.

Transform

Once a connection is established, data are transformed and enhanced using Power BI's Power Query Editor. The Power Query Editor is used to transform and enhance data imported from an external database as well as create new tables from scratch. More advanced transformations can be defined using Power BI's M language, which is the scripting language that backs the Power Query Editor.

1. The Fixture table in Synapse contains the relationship between all Fixtures, Apartments, and Buildings. This table was transformed into three tables when brought into Power BI.
 - a. Fixture – A table containing all unique Fixture IDs and the Apartment IDs they belong to.
 - b. Apartment – A table containing all unique Apartment IDs and the Building IDs they belong to.
 - c. Building – A table containing all unique Building IDs
2. The Smart Device table in Synapse was split into three tables when brought into Power BI.
 - a. Smart Device – The raw data in the Smart Device table
 - b. Manufacturer – A table containing all unique Manufacturer IDs
 - c. Device Type – A table containing all unique Device Type IDs
3. The date time column in the Smart Device Measurements table is stored as a Unix Timestamp in Databricks. When that table is brought into Power BI, a custom column is created that transforms the timestamp into a Date Time string.
4. All tables have data type transformations defined. Power BI tries to guess, but in certain places string type needs to be turned into a decimal type and decimal type needs to be turned into a percentage type.
5. All tables and many columns were renamed for readability within Power BI Desktop.

Custom tables required by the Application Report:

1. Failure Risk Group – A custom table was created for use in the histogram visual which contained all the percentage groups that would be shown. This table defined using the Power Query Editor “Enter Data” wizard. All rows were entered by hand.
2. Date Times – This table contains all dates and times (at an hourly granularity) from the first date in the Smart Device Measurements table to the end of the year of the current date. This table is created by invoking a custom function written in M that takes as input a start date and an end date.

It took roughly one day to add the required transformations. This is a low-code solution where the Power Query Editor automatically generates the M code required for the defined transformations. For all transformations, 106 lines of code were generated.

It took less than 30 minutes to create the Failure Risk Group table. This table was created in a wizard and required no code. The Power Query Editor generated 3 lines of code to define pulling the data set in from the JSON file it lives in. It took roughly half a day to create the Date Times table. The M function that generates the table contains 18 lines of code. The Power Query Editor generated 5 lines of code to execute the function and pull in the resulting data set.

In total, it took one developer around a day and a half and 127 lines of code to create the transformations required for the User Interface on the Azure Native solution.

Model

After the transformations are finished, foreign key relationships between the tables must be modeled. Power BI will make its best guess at defining these automatically, but because the data sets in Azure are stored in two separate databases, some relationships need to be fixed. Power BI provides a wizard where these relationships are visually represented. Below is an image showing this screen and all relationships defined for the Application. The relationships can be defined via drag and drop or through a secondary wizard where columns are selected.

This is a no-code solution that takes one developer less than an hour to complete.

Below is an image showing the model wizard and all relationships defined for the Application.

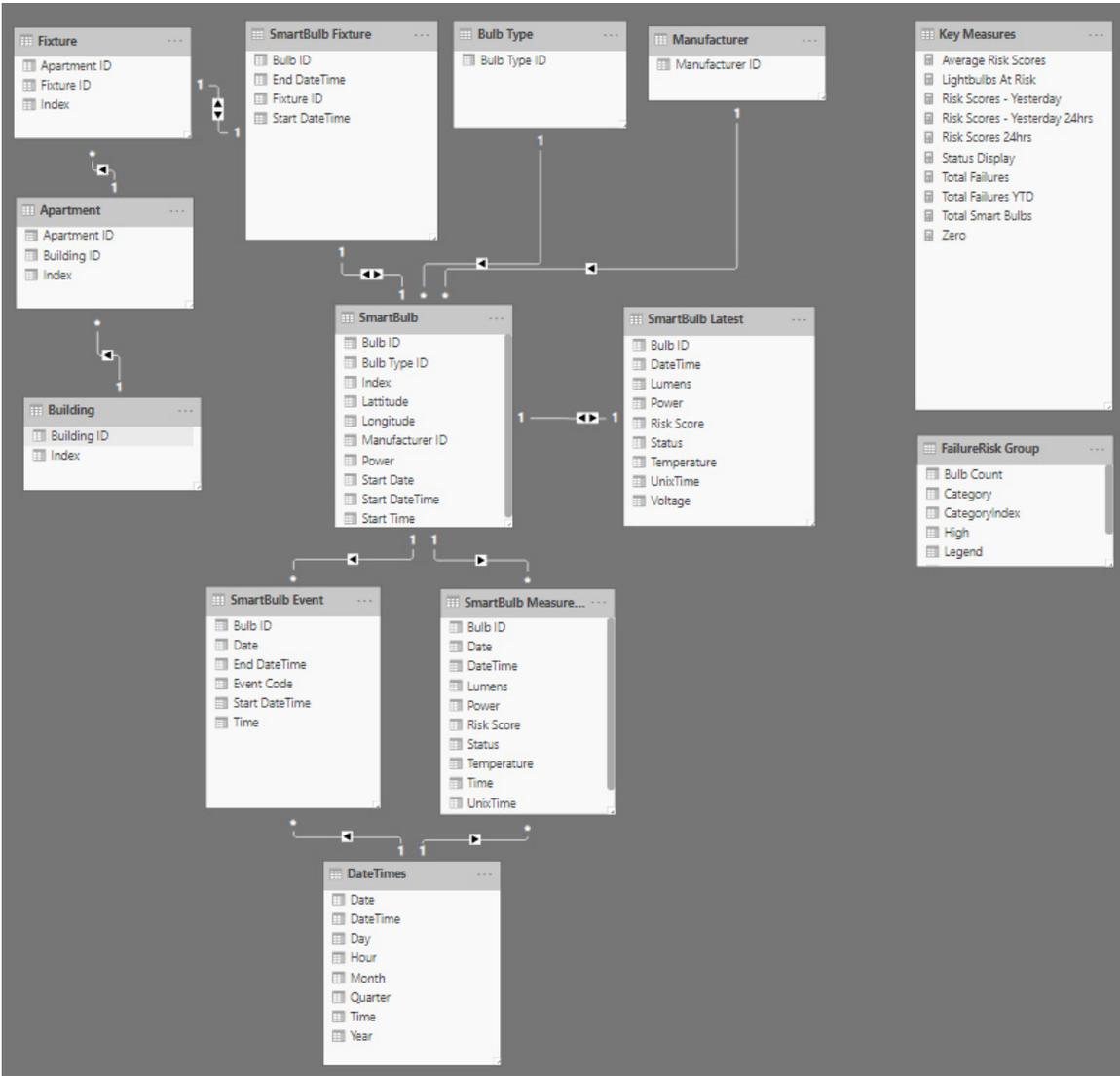


Figure 11. PowerBI Model Wizard and Relationships

Visualize

The final step in building out the report is adding the visuals. Visuals displaying raw data will directly reference imported data and can be built very rapidly. However, visuals that need to perform more complex analysis on imported data will use Measures written using Power BI's Data Analytics Expressions (DAX) queries. DAX can also be used to define calculated columns on existing tables, which is needed for the histogram visual and current value visuals. Each visual needs to be styled to fit the spec.

Dashboard Visuals:

1. Summary
 - a. Three card visuals displaying summary data.
 - i. Each card is backed by a Measure.
 1. Devices at Risk
 2. Total Devices
 3. Failures (YTD)
 - b. One Text Box used for the title.
 - c. One Rectangle object and one line object used to create borders.
 2. Filters
 - a. Four Slicer visuals used for filtering.
 - i. Each Slicer directly references a table column.
 - b. One Button used to clear all filters. This is implemented by utilizing Power BI's bookmarking feature. A bookmark was created that points to the report in a state without filters applied. This bookmark was then attached to the Clear All button.
3. Risk Score Histogram
 - a. One Histogram visual.
 - i. Data comes from the Failure Risk Group table.
 1. Axis categories are created in the table definition.
 - a. 0 – 10%
 - b. 10 – 20%
 - c. Etc.
 2. Device Count Per Category is defined as a calculated column with DAX.
 3. Legend labels are created in the table definition.
 - ii. Styling is applied.
4. Device Map
 - a. One Map visual.
 - i. Latitude and Longitude data comes from the Smart Device table.
 - ii. Tooltips in addition to Latitude and Longitude are backed by Measures.
 1. Devices at Location
 2. Devices at Risk
 - iii. Styling is applied.

5. Devices Table
 - a. One Table visual.
 - i. All columns except for Risk Score come from the Smart Device table.
 - ii. Current Risk Score is defined as a calculated column attached to the Smart Device Latest table.

Device Details Visuals:

1. Summary
 - a. Five card visuals displaying current data.
 - i. Each card is backed by a Calculated Column in the Smart Device Latest table.
 1. Current Status
 2. Current Lumens
 3. Current Voltage
 4. Current Power
 5. Current Temperature
 - b. One KPI visual displaying risk score data
 - i. The KPI is backed by two Measures and a Date Time column.
 1. Risk Scores Previous 24 Hours
 2. Risk Scores Previous 48 Hours
 - c. One Rectangle object and one Line object used to create borders.

2. Device Measurements Chart
 - a. One Line Graph Visual
 - i. X Axis is defined by the Date Time column in the Date table.
 - ii. Y Axis references data directly from the Smart Device Measurement table.
 - iii. Styling is applied.

3. Devices Table
 - a. One Table visual.
 - i. All columns come from the Smart Device Measurement table.
 - ii. Styling is applied.

It took roughly 15 developer days for a single developer to build this report. Adding the visuals is all drag and drop. It took 70 lines of code to define Measures and Calculated Columns using DAX. An additional 7 developer days were required to validate functional capabilities and research acceptance testing options for Power BI. The team decided not to incorporate acceptance testing into the build because Microsoft provides sufficient metrics for Power BI performance.

Azure Implementation Timeline

The following diagram shows the timeline required to implement the Application on Azure. In comparison to the C3.ai + Azure solution, the Azure Native solution required 3 developers for 6 weeks, instead of 1 developer for one week with C3.ai + Azure.

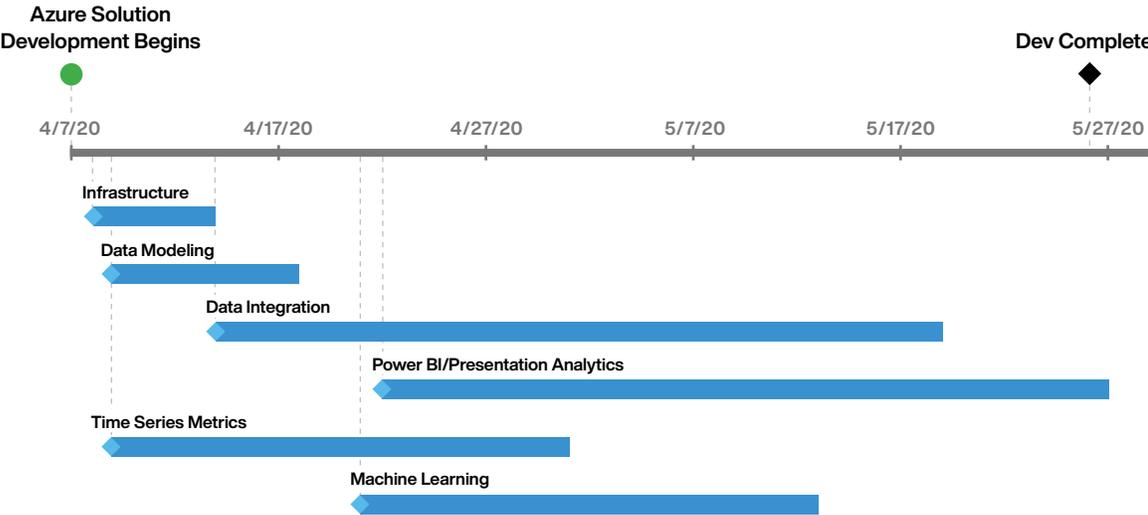


Figure 12. Azure Native Solution Build Timeline

Comparative Observations

After building the solution on both C3.ai + Azure and Azure Native, there were differences in both project metrics and developer experience. Overall, the C3.ai + Azure solution saves time and effort while reducing delivery risk over the Azure Native solution.

Project Metric Comparison

The team calculated three metrics to compare between the C3.ai + Azure and Azure Native predictive analytics implementations: total amount of time (days) to develop the complete Application, amount of time (hours) to develop each key component of the Application, and the lines of code necessary to customize the solution where no-code tools were inadequate.

C3.ai + Azure Low-Code

In all metrics, the team found that C3.ai + Azure (i.e., the C3 AI Suite in combination with Azure) is faster

and simpler for predictive analytics application development than Azure Native. The Azure Native application required three highly experienced developers for 6 weeks, whereas the C3.ai + Azure application was completed by one developer in 5 days. The Azure Native application required 3,047 lines of custom code. Comparatively, the C3.ai + Azure solution was written using only 822 lines of code due to the functionality provided by C3.ai Types. Detailed component-level hour and configuration step results are shown below.

Third-Party Report by Azure Premier System Integrator

Metric	Task	Measured In	Level Tracked	C3.ai + Azure Low-Code Estimated	C3.ai + Azure Low-Code Actuals ¹	Azure Native Estimated	Azure Native Actuals
Amount of Time	Developing the entire application with one FTE	Days	Project Level	3 Days	5 Days	180 Days	90 Days
Amount of Time	Developing key pieces of the Application	Hours	Task Level; added and rolled up to Epic Level	N/A	Design Time 0 Hours Build Data Lake and Ingest Data 0 Hours Model and Enhance Data ² 7 Hours Transform Data ³ 18 Hours Deliver Data 0 Hours Analyze and Visualize Data ⁴ 3 Hours	Design Time 80 Hours Build Data Lake and Ingest Data 40 Hours Model and Enhance Data 80 Hours Transform Data 60 Hours Deliver Data 100 Hours Analyze and Visualize Data 120 Hours	Design Time 52 Hours Build Data Lake and Ingest Data 19 Hours Model and Enhance Data 46 Hours Transform Data 179.5 Hours Deliver Data Not Needed Analyze and Visualize Data 110.5 Hours
Lines of Code	Any customizations where configuration is no longer usable and code is needed to achieve parity	Integer	Project Level	N/A	822	N/A	3,047

Table 6. Developer Experience Metrics (C3.ai + Azure Low-Code)

¹Based on C3.ai Academy's Fundamentals training timeline

²Console & Type Systems

³Data Integration, Methods, Timeseries, Metrics, Data Science Fundamentals, Analytics & DFEs, Jobs, and Queues

⁴UI Framework

C3.ai + Azure No-Code (IDS)

We also compared our Azure Native’s metric results against C3.ai + Azure No-Code (i.e., the C3.ai Integrated Development Studio in combination with Azure) reference build metrics provided by C3.ai’s solution architects, and confirmed that the tracked metrics show favorably towards the C3.ai + Azure No-Code solution, both in less time spent developing, and lower lines of custom code needed.

Metric	Task	Measured In	Level Tracked	C3.ai + Azure No-Code (IDS)	Azure Native Actuals
Amount of Time	Developing the entire application with one FTE	Days	Project Level	3 Days	90 Days
Lines of Code	Any customizations where configuration is no longer usable, and code is needed to achieve parity	Integer	Project Level	14	3,047

Table 7. Developer Experience Metrics (C3.ai + Azure No-Code)

Metrics from C3.ai + AWS Application Comparative Analysis

We also extracted comparable metric results from the previous efforts to build the Application using AWS Native tools.

Metric	Task	Measured In	Level Tracked	C3.ai + AWS Low-Code	AWS Native Actuals ¹
Amount of Time	Developing the entire application with one FTE	Days	Project Level	5 Days	118.75 Days
Lines of Code	Any customizations where configuration is no longer usable, and code is needed to achieve parity	Integer	Project Level	822	16,000

Table 8. Developer Experience Metrics from Previous C3.ai + AWS Application Comparative Analysis

¹Metrics sourced from C3.ai + AWS Application Comparative Analysis document

Developer Experience Inputs

Developers found that having all the capabilities related to data ingestion, pipeline development, machine learning, and visualization in one place was extremely helpful when developing in C3.ai + Azure. In comparison, Azure Native is a general-purpose computing platform and developers must pick the appropriate technologies out of an enormous lineup of services. Additionally, the purpose-built C3.ai Type system and configuration-based system means that once a developer is sufficiently trained and working on an appropriate problem space, the higher level of

abstraction than Azure Native's offerings mean increased productivity and fewer low-level details to understand, build, and maintain.

However, developers utilizing C3 AI Suite's Type System miss the richness of developer tooling that has sprung up for mainstream programming languages and cloud-based configuration languages like ARM templates.

The team captured visuals that showed the work they did on the Application in each platform.

C3.ai + Azure – 5 Full-Time Equivalent (FTE) Days

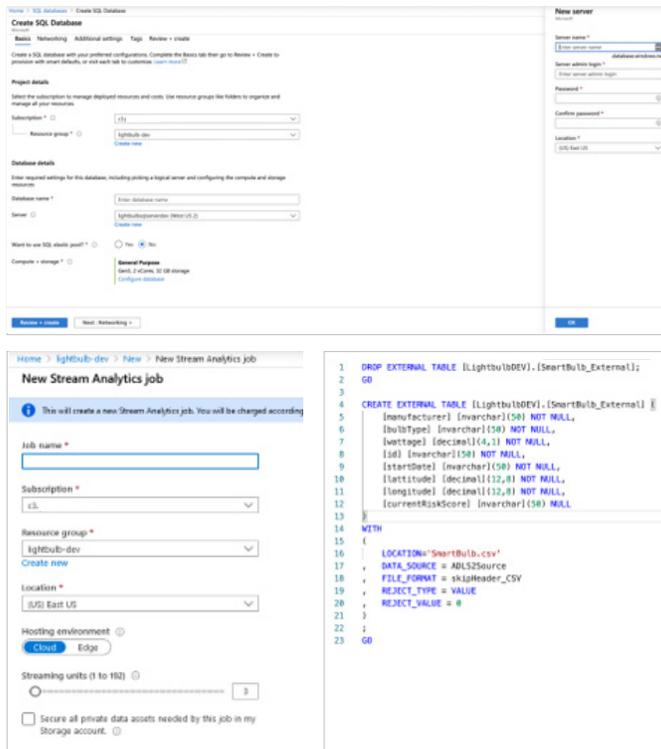


Figure 13. Developer Inputs from C3.ai + Azure

Azure Native – 90 FTE Days

Key Metrics		
Lightbulbs at Risk	Total Lightbulbs	Failures (YTD)
100	100	0

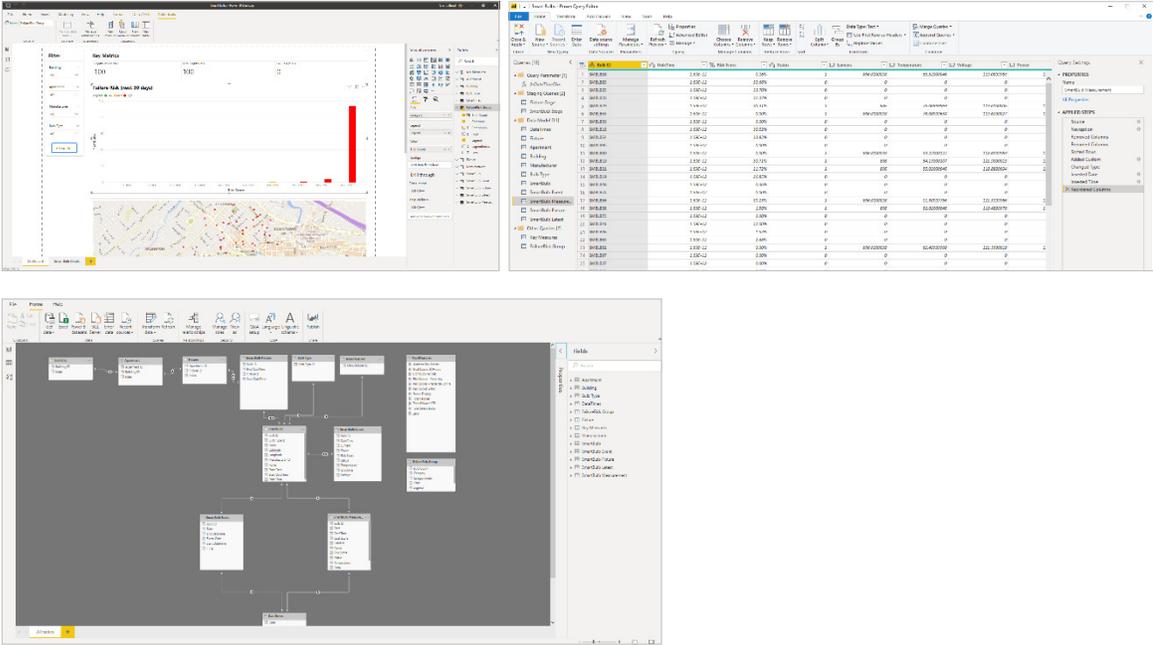


Figure 14. Developer Inputs from the Azure Native Build

'ilities in Detail

The team used a framework with six dimensions, called 'ilities, to compare development of the Application on C3.ai + Azure to development on Azure Native. Each 'ility presented below describes the definition of the 'ility, shows the factors that make up the 'ility, and assigns a numerical score to each factor following the below scale:

1. Does Not Meet Expectations
2. Somewhat Meets Expectations
3. Meets Expectations
4. Exceeds Expectations
5. Exceptional Performance

Additionally, each 'ility includes detailed findings with additional observations. The six 'ilities shown below are functionality, maintainability, usability, affordability, interoperability, and security.

Maintainability

In engineering, **maintainability** is the ease with which a product can be maintained to correct defects and their cause, repair and replace components, and prevent unexpected working conditions. Industrial predictive analytics applications on the scale of those on C3.ai + Azure are very complex. Compute happens across a vast and varied set of infrastructure, tools, platforms, and languages. Additional complexity is added as the solution must evolve to satisfy changing business requirements. Infrastructure services

should be interchangeable and should scale to accommodate changing workloads. The ability to avoid downtime and correct defects is a key differentiator.

Factors and Scores

To evaluate **Maintainability** in comparison with Azure Native, the team reviewed C3.ai + Azure across four factors: **Pipeline Stability**, **Manageability**, **Flexibility**, and **Scalability**.

Pipeline Stability: 4 – Exceeds Expectations

The data types are very well-defined in C3.ai + Azure, which means that changing a data format requires only one change to the Ingest step, rather than needing to change the format throughout the pipeline, as required when building in Azure Native. As a result, this architectural design provides a significant advantage over Azure Native.

Manageability: 4 – Exceeds Expectations

In C3.ai + Azure, all code is version-controlled, making it simple for developers to manage changes, recover earlier versions, and analyze code in smaller chunks. By contrast, in Azure Native, extra effort is required, adding complexity to all three actions.

In addition, C3.ai + Azure provides a layer of protection for production applications beyond Azure Native in situations where recovery operations are necessary. Teams can easily manage or upgrade underlying services, even if that means swapping out for a comparable service, when unexpected events pose a serious threat to the operability of a critical application.

Flexibility: 5 – Exceptional Performance

Based on the program's ability to handle different use cases, data models and inputs as needed, and its abstraction layer which allows developers to quickly adjust lower-level details, C3.ai + Azure is exceptionally flexible in comparison to Azure Native.

Scalability: 5 – Exceptional Performance

Although all Cloud Service Providers ("CSPs") allow developers to add capacity, the C3 AI Suite

adds a layer of abstraction so that developers do not need to interact directly with the hosting platform infrastructure when more capacity is needed. Additionally, the ease with which data can be moved across data stores allows developers to scale through differing data storage solutions depending on solution fit for the use case.

Detailed Findings

Infrastructure Flexibility

As AI solutions become increasingly ubiquitous, development teams will often be deploying applications that comprise a critical part of business operations. The model-driven approach used to develop on C3.ai, including the modeling of infrastructure components, decouples custom code from the underlying architecture, which decreases the risk and impact caused by unexpected events.

On C3.ai + Azure, making infrastructure changes is a simple developer task because of the abstraction layer the C3.ai Type System provides. Developers can modify existing resources and change the underlying infrastructure with minimal effort. Making infrastructure changes on Azure Native is also trivial when modifying existing resources; however, changing the underlying infrastructure (e.g., swapping PostgreSQL for Azure SQL) on Azure Native will require careful management of potential impacts to the environment. Any integration points with the changing infrastructure will have to be updated, and refactoring existing code will likely be necessary.

For example, if a solution architecture included Cosmos DB and Microsoft released a critical update that required code changes, then the development team would have to upgrade their Cosmos DB instance and implement a fix. If the team built their solution on C3.ai + Azure, then a comparable data store could be used to replace Cosmos DB, or the corresponding model could be updated to leverage the latest changes. On Azure Native, the team would have to scramble to upgrade Cosmos DB and make necessary code changes, a risky undertaking that could lead to extended downtime and negatively impact key business processes.

Infrastructure Scalability

Leveraging the C3.ai Type System, developers can quickly provision environments capable of scaling without investing time in selecting and configuring the appropriate services. Azure Native requires customers to choose between a multitude of services and configure chosen resources properly. If configured correctly, Azure Native provides a platform for creating solutions that scale automatically and allows customers to minimize costs by leveraging the “pay-for-what-you-use” benefits of the cloud. Like the Azure Native experience, if specific resources are necessary for a particular use case, C3.ai + Azure also allows customers to be selective. Furthermore, the C3.ai Platform will automatically scale the entire data pipeline based on user settings and live throughput requirements. Users are able to manually intervene if needed to create additional resources.

ML Model and Pipeline Management

ML Pipeline and Deployment

C3.ai + Azure has a durability advantage over Azure Native because the training, deployment, and prediction processes exist in a single, closed-loop system, thus minimizing the risks of a single point of failure causing disruptions to business-critical functions.

For example, in the Azure Native solution, once the machine learning model has been trained in Azure ML, a real-time inference pipeline can be generated from the training pipeline and deployed to a REST API for consumption. A Databricks job is triggered when new streaming data has been ingested to generate predictions using this endpoint and write them to storage. In C3.ai + Azure, trained models can be written to an inference-pipeline, which generates predictions once incoming data are processed by C3.ai Data Flow Events and Analytics. Those predictions are then written directly to a C3.ai Type object. As a result, the C3.ai + Azure pipeline does not need to process data across services by using triggers, authentication, and endpoints.

ML Model

In C3.ai + Azure, developers can create and handle new use cases with very little overhead when compared to Azure Native.

If a model needs to update to include new features or train with a different algorithm, C3.ai + Azure can handle this easily by training the updated model and overwriting the current model in the inference pipeline. To do the same in Azure Native requires updates across multiple services, such

as the data transformation notebook in Azure Databricks and changes to models and pipelines in Azure ML Studio, causing a significant increase in development effort and a decrease in time to value.

Data Model Management

Updating Data Models

When changing data models to handle new use cases, data models in C3.ai + Azure have an advantage over those in Azure Native, for two reasons:

1. **One-Step Updates.** In C3.ai + Azure, data models are represented as C3.ai Canonical Models with fields that reference the data source, so that any change to the Canonical Model updates all the fields that reference that data. In Azure Native relational stores, any new data written to or deleted from one table will not propagate to its child tables unless configured to do so.
2. **Version Control.** Relational stores in Azure Native are largely programmed as procedures such as SQL queries, rather than as objects such as C3.ai Types. Procedural programs are more difficult to version control and edit than C3.ai Types, because they must be written to delete and re-create tables if the data schema needs updating.

Monitoring

Runtime Management

C3.ai + Azure is a fully managed platform, which eliminates the need for developers to create or configure dashboards to monitor infrastructure performance of production applications. Azure Native has strong tooling for monitoring performance; several important metrics can be monitored in dashboards that are available in the Azure Portal by default and additional dashboards can be configured to track other metrics of interest.

Usability

Usability is the ease of use and learnability of software that can be used by specified consumers to achieve quantified objectives with effectiveness, efficiency, and satisfaction in a quantified context of use. To achieve this, teams that implement predictive applications should have an easy-to-understand development environment with the tooling and support they need to work productively. Platforms that allow developers to learn and create with ease and have reliable native and third-party support rate highly in usability.

Factors and Scores

To evaluate **Usability** in comparison with Azure Native, the team reviewed C3.ai + Azure across two factors: **Developer Efficiency** and **Developing the Mental Model**.

Developer Efficiency: 4 – Exceeds Expectations

With the C3 AI Suite, C3.ai + Azure covers all the requirements to develop an AI solution in the cloud. The features needed to process data, manage ML pipelines, and create user interfaces are available in one place, which is helpful for developers who may be overwhelmed by the need to manage several services for the same purpose in one of the other cloud platforms.

Developing the Mental Model: 4 – Exceeds Expectations

C3.ai + Azure performs well in this area by combining the work of several different services in a typical AI developer environment in one place. The training offered is comprehensive enough for a new developer to understand the basics

for creating custom solutions. While there is a steep learning curve to using C3.ai Types, once the concepts are understood the elegance of the architecture means that only a few components need to be learned for the Application. If there are any knowledge gaps, developers can reference the C3.ai community, trainings, and documentation. The simplicity of the C3 AI Suite is an advantage that elevates C3.ai + Azure over more complex platforms like Azure Native.

Detailed Findings

Developer Tools

The C3.ai + Azure solution leverages the C3.ai Type System to efficiently create and manage robust data models and ML pipelines without the need to manage multiple environments, services, and programming languages. C3 AI Suite's Standardized Design Language (SDL) also tightly integrates with C3.ai Types to create consistent, efficient, and scalable user interfaces that include components such as graphs, maps, status summaries, and filters out of the box. Azure Native, in comparison, utilizes different services for each part of the architecture. Developers must jump between these services to create and validate business logic, which hinders developer productivity.

C3.ai Types also provide the additional benefit of type annotation, a tool that allows users to document the business logic of all C3.ai Types, including data models, pipelines, metrics, etc. This tool benefits users who are collaborating on the platform or referencing previous work and has no equivalent on Azure Native.

Community

The Azure Native solution consists primarily of Azure services, Databricks, and Power BI. The Microsoft Tech Community is a web forum managed by Microsoft for discussion around various services such as Office 365, Bing, and Azure. However, this forum does not have a section for Power BI and the Azure content is sparse at around 3,000 entries. This is in comparison to that on popular forums such as Stack Overflow, which has 8,000+ results with the Power BI tag and 80,000+ results with the Azure tag. Databricks also has its own native community with about 6,000 posts, with an additional 2,000 posts in Stack Overflow. C3.ai has a much smaller community, since its tool has a much smaller user base compared to Azure Native. The key disadvantage to this is that results are not likely to populate through the search engine and the available posts likely provide inadequate coverage of potential questions, especially for smaller topics such as Machine Learning and Security. As a result, answers to questions on the C3.ai Community forum are largely dependent on the effort and availability of C3.ai staff.

Training

Microsoft offers 50 certifications and exams across various Azure services. There are more than 100 mini-courses on their website to learn how to set up Azure resources and develop on their platform. In addition, instructor-led training is available through Microsoft Learning Partners, a worldwide partner network that delivers flexible, role-based, customized training and certifications in Microsoft technologies in blended learning, in-person, and online formats. Many free and

paid third-party courses for Azure also exist on the internet. C3.ai has fewer and more focused training programs for the C3 AI Suite (Low-Code), IDS (No-Code), and ML Studio (Machine Learning) offerings. Trainings are administered by C3.ai in-person or remotely through Coursera with office hours held by C3.ai staff.

Documentation

Microsoft has very extensive documentation on most Azure services. Most search results lead to one or more relevant web pages with tutorials, example code, and explanations. However, documentation was not available to explain the functionality of some of Azure's newer or recently updated services. Power BI documentation is not as comprehensive. In some cases, it was challenging to understand specific functions because the documentation did not include example code. For C3.ai, documentation is not available through the search engine; partners and customers must access documentation through the C3.ai developer portal. The information is comprehensive on both development topics and specific C3.ai Types. However, unlike the documentation on Azure, various filters are used for navigation rather than a hierarchy of articles on the left pane. Documentation is not an area of differentiation as both platforms provide a more than adequate level of documentation for their size and complexity.

Support

User Management

Azure Native has standard user management features to control the roles and access privileges of users. This can be used to restrict users from

accessing, changing, and provisioning specific resources. C3.ai + Azure offers similar functionality with permissions and access conditions for roles in both their low-code and no-code environments. This is not an area of differentiation between C3.ai + Azure and Azure Native.

Service Level and Support

Azure Native has 99.95% average uptime across all its services. The Service Level Agreement promises different uptimes ranging from 99%

to 99.999% depending on the service, its tier, and the number of availability zones to which it is deployed. When services are down, support engineers promise an initial response time ranging from 15 minutes to 8 hours, depending on the Azure support plan. C3.ai + Azure relies on not only the uptime of the same Azure resources as Azure Native but also those managed by C3.ai. C3.ai also promises uptimes ranging from 99% to 99.9% and has maintained an average of 99.95% uptime over the last twelve months.

Affordability

Affordability is the solution's overall cost including acquisition and ongoing maintenance.

Factors and Scores

To evaluate **Affordability** in comparison with Azure Native, the team reviewed C3.ai + Azure in comparison to Azure Native across three factors:

Developer Productivity; Ramp-Up Time; and Design Time.

Developer Productivity: 4 – Exceeds Expectations

Due to its being a single platform containing all required components and integrations, it is easy for a single developer to create a predictive analytics application on C3.ai + Azure. It only took each of the team's developers five business days to create the Application. It is a much more labor-intensive task for a single developer to create a similar predictive analytics application on Azure Native; in fact, it takes 18 times as long.

Ramp-Up Time: 4 – Exceeds Expectations

When developing on C3.ai + Azure, each developer must learn the C3.ai Type System, and it takes 3-6 months to become truly proficient. Proficiency with Azure Native takes much longer, as there are a variety of different reference architectures and use cases to master.

Design Time: 5 – Exceptional Performance

One of the most time-consuming tasks in Azure Native is design time – working through the high-level architecture, deciding on which components and services to utilize, and detailing and revising the solution as the build takes shape. Frequent updates to the platform provide more options but make architectural decisions more challenging. Although the team came into the comparative analysis project with a reference architecture in mind, and it was agreed to very quickly, approximately 52 hours were spent in design time. With C3.ai + Azure, the solution was pre-built, which cut design time to near zero.

Detailed Findings

There are additional considerations that impact the Affordability of C3.ai + Azure in comparison to Azure Native. The main consideration is Total Cost of Ownership (“TCO”), which is related to Maintainability.

Total Cost of Ownership (“TCO”)

The C3 AI Suite’s model-based architecture builds in a lower TCO when compared to cloud native solutions. It uses an abstraction layer that facilitates and shortens typical developer activities, such as:

- Learning the platform language and architecture
- Building low-code and no-code solutions
- Fixing bugs
- Creating new ML models
- Changing integrations to data sources and services
- Changing integrations to the underlying CSP infrastructure, such as adding Databricks to Azure

Functionality

In software engineering and systems engineering, a **functionality** refers to a function of a system or its component, where a function is described as a specification of application behavior between outputs and inputs. All functionality that is provided by the platform accelerates the time to value for any applications that are built on it.

When using C3.ai + Azure, fewer developers are required to build solutions, and each developer takes significantly less time to perform each activity; for example, building a predictive analytics application takes one FTE five days, while building the same application on Azure Native takes three FTEs 30 days (see details here). The additional complexities and customizations involved in the Azure Native solution impact the above listed developer activities and will naturally require higher levels of ongoing operational costs to maintain and enhance, thus driving up TCO. This results in a lower overall TCO for C3.ai + Azure in comparison to Azure Native.

Factors and Scores

To evaluate **Functionality** in comparison with Azure Native, the team reviewed C3.ai + Azure’s suite across one factor: **AI Platform**.

AI Platform: 4.5 – Exceeds Expectations

The platform allows developers to create data pipelines, machine learning models, and gather analytical insights very simply. C3.ai + Azure seems to follow industry best practices in all of these aspects by default and developers can create a robust data and analytics pipeline with little overhead. In addition, the platform does a good job of abstracting away service level details and allows users to focus on application development.

Detailed Findings

Provision Infrastructure

C3.ai is a fully managed platform that leverages a model-driven approach to provide a layer of abstraction on top of the underlying infrastructure components. Thus, teams using C3.ai + Azure are not required to have significant knowledge of the Azure services used to comprise a solution's infrastructure.

On Azure Native, the Azure Resource Manager (ARM) service provides a unified experience for provisioning infrastructure across nearly all services on the Azure platform. Customers have the option of deploying services via the Azure Portal (no-code) or using ARM templates (code). While the functionality provided by ARM templates is substantial, enabling an infrastructure-as-code approach, the syntax and authoring toolchain for ARM templates are complex and require experience to take full advantages of the features provided. Either option requires users to have an understanding of the resources available on Azure and some knowledge of networking to deploy anything beyond a basic architecture.

Define Data Model

The advantage of data modeling in C3.ai + Azure is the ease of manipulating C3.ai Type objects compared to that of relational data stores in Azure Native. In Azure Native, database schema are the data models for relational data stores and each model represented as a table. Therefore, any deviations of the data model from the source data schema requires the defining of parent-child table relationships, the setting of primary and foreign keys, and complex joins. In contrast, C3.ai Types, which are represented as objects that reference the source data directly, provide more freedom when creating the data model because entity attributes can be defined through expressions, which abstract the data manipulation logic from the user. This abstraction layer allows the user to interface with the data (specifically using Canonical and Transformation Types) without requiring knowledge of the source data. A developer on Azure Native would typically be required to understand how to model data for multiple data sources.

Build Data Integration

Data integration on C3.ai + Azure is simplified by the C3.ai Type System. Developers on C3.ai + Azure define Canonical Models to represent business objects located in one or multiple data stores. Downstream interactions with objects represented as Canonical Models can be implemented with no dependence on the source. Additionally, effort to create transformations, manipulate time-series data, and generate alerts is significantly reduced using C3.ai Types that leverage the native, asynchronous processing engine. As a result, data pipelines built on C3.ai + Azure are inherently maintainable and scalable.

On Azure Native, significant effort and experience is necessary for a developer to establish data pipelines that support a typical AI solution – massive data volume, varying data sources, and diverse data structures. Multiple services exist on Azure to support this effort with drag-and-drop interfaces, but no single service abstracts the full spectrum of data integration tasks which can be defined using the C3.ai Type System.

Process the Data

Time Series & Metrics

Harnessing the power of IoT for AI solutions requires development teams to handle large amounts of unstructured, time-series data. C3.ai makes it less complex for developers to tackle these problems using metric Types. Using the C3.ai Type System, teams can define time-series transformations of normalized data across space and time with minimal code. For example, a simple metric could be used to find the average voltage of all Smart Devices in a particular building the last month. AI solutions often require more complex transformations than averaging a single data point, so developers on C3.ai must also implement compound metrics. Similar to simple metrics, developers can define compound metrics using C3.ai Types with minimal code – typically one line of expression-like syntax in a file with other basic metadata (e.g., metric name). Additionally, the C3.ai Developer Console provides functionality for customers to quickly iterate on metric implementations and visualize the output.

Achieving a similar feedback loop on Azure Native typically requires the integration of multiple services or experience with common visualization

libraries that are available in Azure machine learning environments (e.g., Matplotlib, Seaborn). The team found that implementing similar metrics on Azure Native, both simple and compound, required leveraging common Python libraries used by data scientists, and took on average 10x longer.

Analytics

AI solutions often provide value to end users via notifications. On C3.ai + Azure, Analytic Types make it easy for developers to trigger alerts based on Data Flow Events and previously defined Metrics. Developers have the option to override processing behavior by implementing custom logic in JavaScript, making it possible to satisfy diverse business requirements. For example, the team configured an alert any time a device was defective with a simple JavaScript implementation based on the **HasEverFailed Metric**. Any Analytics defined on C3.ai + Azure leverage an asynchronous processing engine, the Analytics Container Engine, to notify users when thresholds defined by the business are exceeded as new data are received.

Implementing Analytics on Azure Native required the team to inject custom notification logic in the Transform step of a data pipeline to imitate the functionality provided by the C3.ai Asynchronous Processing Engine. The team found this took twice as long on Azure Native for developers with previous data wrangling experience and expected a greater level of maintenance effort to be required in the future.

Error Handling and Logging

While the solution returns error messages when provisioning, the wording in the messages is not as specific as developers would need to easily identify the issue. Troubleshooting was somewhat challenging given the fact that the user was required to interact with the API (making individual calls) to inspect important data. A suggested improvement is to create a job/worker/queue dashboard to surface the errors as they occur.

However, the C3 AI Suite does provide a robust logging mechanism for deployed applications. It automatically collects usage data from all points in the pipeline and makes that data easily accessible to users. In contrast, a developer would need to aggregate this information from multiple services while using Azure Native.

Data Science & Machine Learning

Feature Engineering

Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work.

Features in ML models consist of transformations and enhancements to the Application's source data and metrics. If the feature engineering part of the pipeline requires heavy data transformation work, then a programming environment such as Jupyter Notebooks is recommended. C3.ai + Azure and Azure Native both have a version of Notebooks integration, so this is not an area of differentiation. However, outside of Notebooks, C3.ai + Azure has the advantage of C3.ai Transforms, which provide a layer of abstraction from data transformation logic. This allows complex data transformations to be defined as

simple expressions, significantly simplifying the feature engineering process. As a result, feature engineering on C3.ai + Azure is more capable and simpler to code than feature engineering on Azure Native and other competing platforms.

ML Model Tuning

Given a set of machine learning features and a machine learning algorithm, there are ways to control the machine learning process to yield different and perhaps better results. The main methods of model tuning include re-sampling the data (over and under-sampling), changing the algorithm's hyperparameters, and specifying a specific solver for the algorithm. C3.ai + Azure and Azure Native both support these functions and the team did not experience any area of differentiation between the two platforms.

ML Model Evaluation

Model evaluation involves the analysis of performance metrics across different models, thresholds, and score bins. Typical metrics include accuracy, precision, recall, F1-score and predictions versus actuals, such as false positive rate and true positive rate. These metrics can also be extrapolated into graphs such as a ROC curve or precision-recall curve. Both C3.ai + Azure and Azure Native offer this functionality out of the box. In Azure Native, performance metrics, graphs, and score bin data are automatically displayed in a visual interface. On C3.ai + Azure, these evaluation parameters must be extracted programmatically by executing individual commands through the C3.ai developer console. There are also no commands to display graphs or change the threshold post-training. Therefore, Azure Native has a slight advantage over C3.ai + Azure in this space.

User Interface and Programmatic Tooling

Business Intelligence Integrations

C3.ai + Azure can integrate with SDL to solve most common business intelligence use cases. SDL provides components such as graphs, maps, status summaries, and filters out of the box. For more complex use cases requiring a third-party business intelligence tool, it is possible to connect directly to the C3.ai + Azure data source.

As it stands, developers would need to leverage the API generated by the C3.ai Type System or connect directly to the underlying data service to integrate with an external business intelligence solution. While it is very similar to what an Azure Native solution would require, this solution causes developers to lose the service abstraction usually gained when using the C3 AI Suite. To that end, C3.ai is currently developing a native Power BI data connector so that developers can easily integrate

with Microsoft's BI solution without needing to know the underlying services. This will give C3.ai + Azure a significant advantage over an Azure Native solution.

API Gateway

An API Gateway provides a single-entry point to a defined group of services. It often provides management features around common API functionality such as security, caching, and load balancing.

C3.ai + Azure automatically generates a basic API from the C3.ai Type System, saving development time that would be spent to develop data access methods, create API specifications, provision an API environment, configure CORS, and write unit tests.

Interoperability

Interoperability is the solution's ability to interact effectively with other systems or components.

Factors and Scores

To evaluate **Interoperability** in comparison with Azure Native, the team reviewed C3.ai + Azure across three factors: **Integration**, **Delivery**, and **Portability**.

Integration: 4 – Exceeds Expectations

Based on the team's experience with C3.ai + Azure, it appears the Application can bring in data

from any source, and the developer can easily transform data into C3.ai Types using an object-oriented model.

Delivery: 4 – Exceeds Expectations

Currently it is not possible to integrate visualizations built on the C3.ai platform with other applications. The ability to leverage a BI solution (Power BI, Tableau, Qlik, etc.) using data connections is an interoperable strength of Azure Native. While it is possible to leverage the API generated by the C3.ai platform to integrate with

external BI solutions, there are currently no native data connectors to facilitate a simple connection. However, C3.ai is actively developing these data connectors including PowerBI, which will allow it to match the visualization interoperability of Azure Native.

Detailed Findings

Data Integration

Canonical Model

C3.ai Canonical Models are a subset of the C3.ai Type System that represent key business objects from existing IoT and enterprise systems. Defining Canonical Models on C3.ai + Azure significantly reduces data integration efforts because these Types can take advantage of the C3.ai Type System native capability to read/write Types across a wide variety of data stores. In contrast, developers working on Azure Native have to define data models specific to chosen data stores, which is a rigid approach that leads to increased maintenance as solutions evolve. A more robust architecture would likely involve an Object Relational Mapping (ORM) library, but no existing ORMs support spanning multiple data stores. The tradeoff of using the C3.ai Type System to define data models is that C3.ai Types are proprietary to the C3.ai platform. These models cannot be exported and used outside of the platform nor can outside models be imported directly into the C3.ai platform before being redefined using the Type System.

API (Application Programming Interface)

Exposing data is an essential part of delivering business value from AI solutions, and C3.ai provides an out-of-the-box API for end users to consume all data on the platform to analyze and

derive insights. On Azure Native, development teams typically spend 2-3 weeks developing custom APIs that require maintenance and performance monitoring. To maximize interoperability, API solutions should implement the Open Data Protocol (OData) in order to simplify consumption and take advantage of inherent integrations with common reporting solutions (e.g. Excel, Power BI, Tableau). An additional 2-3 weeks of effort is needed to properly build APIs that comply with OData, but an early investment of resources will allow teams to focus more on business needs and less on development.

C3.ai is also actively developing a native Power BI data connector which would allow developers to connect to a C3.ai + Azure solution without needing to consider the underlying data store or even use the generated API.

Data Storage

The C3.ai Type System makes it easy to connect to various data sources with simple Type definitions. Any Types defined in C3.ai as persistable will be stored in Postgres by default. Adding more Data Source Types is straightforward – developers can use any of the numerous templates for common open source and cloud data stores or setup custom configurations. While it's possible to integrate with open source or other CSP data stores using Azure Native, developers usually design architectures comprised of Azure services to maximize efficiency and leverage native integrations. Integrating with external data stores, such as AWS DynamoDB, would lead to additional configuration efforts and potentially require custom development. See example storage options below:

Storage Type	C3.ai + Azure	Azure Native
Relational	Postgres (Default) AWS RDS GCP Cloud SQL All Azure Offerings and more...	Azure SQL Azure Postgres Azure MySQL
Key-Value	Cassandra AWS DocumentDB GCP Cloud Firestore All Azure Offerings and more...	Azure Blob Storage Azure Cosmos DB
Multi-Dimensional	AWS Redshift GCP BigQuery All Azure Offerings and more...	Azure Synapse SQL DW HBase in HDInsight

Table 9. Data Storage Types for Predictive Analytics Application on C3.ai + Azure and Azure Native

Security

Security is the solution's ability to prevent unauthorized disclosure, loss, modification, or use of its data or functionality. This is a critical concern for mission critical predictive applications that are on based on extremely sensitive business data, as an attacker could cause critical harm to business operations. Security can be divided across the following sub-criteria: Secure Design, Authentication, Access Control, Encryption, and Auditability.

Factors and Scores

To evaluate **Security** in comparison with Azure Native, the team reviewed C3.ai + Azure across multiple factors typically included in Virtual Private Clouds.

Security: 3 – Meets Expectations

C3.ai + Azure offers security like that of other Cloud Service Providers. Role-Based Access Controls are included as well as an OAuth implementation out-of-the-box using Okta. Roles and Groups are defined as JSON objects in the project solution so they can be added to a code repository for better version control.

Conclusion

A team of three experienced software engineers built a simple predictive analytics application for AI-enabled devices on C3.ai’s no-code platform – the C3 AI Suite – in combination with Azure (“C3.ai + Azure”), and compared it to building a similar application using only Azure native services (“Azure Native”).

The team found that building the application on C3.ai + Azure accelerated development by a factor of 30 times over Azure Native, while reducing effort and risk through C3.ai’s model-driven architecture. The team also concluded that C3.ai + Azure required significantly less development code than Azure Native and is more pleasant to work with overall.

Metrics	Azure Native Low-Code	C3.ai No-Code (IDS) + Azure	Effort Comparison Using C3.ai + Azure
Total Effort (FTE Days)	90 days	3 days	Reduced by 30x
Lines of Custom Code	3,047	14	Reduced by 217x

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



C3.ai

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