1. Technical Evaluation Criteria: The evaluation of the technical proposal will consider the following:

a. Understanding of the Problem - The technical proposal was evaluated to determine the extent to which it demonstrates a clear understanding of all features involved in solving the problems and meeting and/or exceeding the requirements presented in the task and the extent to which uncertainties are identified and resolutions proposed.

b. Feasibility of Approach - The technical proposal was evaluated to determine the extent to which the proposed approach is workable and the end results achievable. The technical proposal was evaluated to determine the level of confidence provided the Government with respect to the Offeror’s methods and approach in successfully meeting and/or exceeding the requirements in a timely manner.

2. Proposal Summary:

The Offeror’s proposal included the following elements:

Brief summary paragraph of what the proposal contains without evaluation, commentary or any opinion.

The Offeror proposed to team with [insert number of subs] subcontractor(s).

**Ablevets**

Proposal Summary:

Ablevets succinctly summarizes its solution - including all dependencies and components to satisfy the all RTEP requirements in a single overarching technical architecture (Exhibit 3, page 4).

This suite of integrated components, and HOW they are created and implemented are as follows:

2.1 Generation of Native Data models from Fileman Data Dictionary

\* Provides the foundational model for VICS data classes and objects

\* Initial, Native Model is over 2500 classes and 65,000 properties

2.2 Refining the Native Data Model for VICS

\* Common Native Model is standardized, normalized subset of Native Model

\* Common model enables national clients and services across all 130 VistA systems

\* Each VICS Data Model class is sufficiently defined to allow for automatic translation between any VICS object and the native objects it refines

\* Automatic translation between Native and Common Native models provides backwards compatibility between distinct VistA systems and national services model

Exhibit 4: Guidelines for refining relevant classes of the Common Native Model

Exhibit 5: VICS Data Model Categorization

=> Enables measurement and distinction between

VA-specific data vs standard data

To allow continuity of VA services while introducing generic COTS services

Patient data vs metadata

To provide new access control policies

2.3 Developing Common Utility Services

Exhibit 6: Common Utility Services

2.4 Enabling a Portable VICS interface

Allows porting of VICS interfaces over a “commercial Cloud-based EHR solution”

New clients: REST; VPR clients: VPR emulator; CPRS clients: RPC emulator

2.5 Emulate RPCs

\* Service layer over VICS

\* Enables “VistA Domain Test Suite” to establish behaviors of VistA

\* Prevents code injection and security problems

\* Categorizes RPCs to ones that READ or CHANGE the system

\* Distinguishes those that affect the patient record (“patient RPC”) from

those dealing with metadata (“meta RPC”) – allowing new access control policies

\* Distinguishes utility RPCs - reducing complexity, allowing common RPC re-use

2.6 Emulate VPR RPC

=> An atypical RPC - one with wide (not narrow) range of data domains query

=> Each RPC will be emulated by one VICS

=> Will emulate the XML form of VPR used by JLV and production VA clients

2.7 Use Node.js compatible Cloud-based NoSQL data storage (“VICS Data Store”)

=> Leverages Filemans NoSQL store Native Model definitions

=> Leverages largest industry-standard Cloud data storage from

Amazon (DynamoDB), Microsoft Azure (CosmosDB), and MongoDB

2.8 Develop RPC router

=> Required to “sniff” and audit all RPC traffic between CPRS and VistA

=> Enables audit, isolation, and emulation of all CPRS RPCs

=> Redirects relevant RPCs to VICS

=> Enables centralized management of “utility” RPCs repeatedly called by CPRS

=> Executes a single RPC in both VistA and VICS

2.9 Implement backward synchronization service

=> enables backwards compatible interfaces

=> enables migration off of VistA/MUMPS to VICS

**Technical Summary**: Each component of this architecture maps clearly to the requirements of the RTEP. All components are developed using Javascript/Node.js. There are no MUMPS dependencies in the final VICS architecture.

**Overall**: The vendor specifies in clear, implementable, technical detail the rationale for all components, WHY each component is required, HOW all components are created, and HOW they work together to provide a complete, composite VICS architecture.

**B3 Group**

Total pages: 15

Technical: 1-6 (6 pages)

VIP generic: 7-13 (6 pages)

Proposal Summary:

The overall approach of B3 is based on using the work performed in a previous government-funded research project (VistA data project) in which they were not involved. The B3 technical approach contains verbatim, unmodified copies of text and figures (figure 2 and figure 5) from the VistA Data Project website and slideware previously submitted to the government.

These figures make no mention of backwards synchronization required to enable the final solution to be legacy VistA/MUMPS independent. Nor does the figure show any form of centralized services (VICS). Since these technical issues were not addressed in the architecture or the text, a complete solution was impossible to assess.

B3’s approach included a “Javascript based approach” using the Intersystems Cache add-on module for Node.js. According to B3, this Intersystems add-on module “can expose VistA’s data as a single, secure, symmetric read-write, server side interface to all underlying data”. B3 provided an example of how this Intersystems Node.js module would provide “MUMPS emulation using a Javascript/Node.js-driven model-driven replacement” (page 4).

However Node.js is simply a Javascript run-time environment (“plumbing”) and has no functionality beyond this. Specifically, it is not a off-the-shelf “model-driven MUMPS emulator” as is assumed in this response. (If that were the case, there would be no need for this RTEP). As a result, the entire problem of how to create the model-driven MUMPS emulation – the foundation of this RTEP - is missing from the response.

**Prosphere**

Proposal Summary:

The overall approach of Prosphere to provide VICS is summarized in their overall architecture (Exhibit 6). Their plan is to provide an “integrated micro services architecture” to compartmentalize common web service functionalities.

The technical approach is to modify both MUMPS components (Exhibit 11) to add an “Entity file” to Fileman, and wrap these Fileman/Entity files with web services.

Note this creates a MUMPS/Fileman dependency, which means the final solution cannot be legacy MUMPS-independent.

As we can easily see in their overall figure (Exhbit 6; Blue=MUMPS and Orange = VICS) that the final solution (orange) depends on the legacy MUMPS (blue).

Assessement: This fails the most important criteria of the final solution being legacy MUMPS indepdendent.

* Node.js is plumbing; it has nothing to do with modelling, and has no such capabilities
* No mention of VICS in the figure or text
* Does not address the VICS, but rather it repeats the VDP figures and text (verbatim)

NO MENTION OF VICS IN THIS FIGURE

NO CORRESPONDING TEXT TO EXPLAIN THE FIGURE

The text does not correspond to the figure.

The figure does not show any centralized service

What are these components?

How do they interact?