Automated Injection of Curated Knowledge Into Real-Time Clinical Systems
CDS Architecture for the 21st Century

by

Preston Victor Lee

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved September 2018 by the
Graduate Supervisory Committee:

Valentin Dinu, Chair
Robert Greenes
Davide Sottara

ARIZONA STATE UNIVERSITY

December 2018

ABSTRACT

Clinical Decision Support (CDS) is primarily associated with alerts, reminders, order entry, rule-based invocation, diagnostic aids, and on-demand information retrieval. While valuable, these foci have been in production use for decades, and do not provide a broader, interoperable means of plugging structured clinical knowledge into live electronic health record (EHR) ecosystems for purposes of orchestrating the user experiences of patients and clinicians. To date, the gap between knowledge representation and user-facing EHR integration has been considered an "implementation concern" requiring unscalable manual human efforts and governance coordination. Drafting a questionnaire engineered to meet the specifications of the HL7 CDS Knowledge Artifact specification, for example, carries no reasonable expectation that it may be imported and deployed into a live system without significant burdens. Dramatic reduction of the time and effort gap in the research and application cycle could be revolutionary. Doing so, however, requires both a floor-to-ceiling precoordination of functional boundaries in the knowledge management lifecycle, as well as formalization of the human processes by which this occurs.

This research introduces ARTAKA: Architecture for Real-Time Application of Knowledge Artifacts, as a concrete floor-to-ceiling technological blueprint for both provider heath IT (HIT) and vendor organizations to incrementally introduce value into existing systems dynamically. This is made possible by service-ization of curated knowledge artifacts, then injected into a highly scalable backend infrastructure by

automated orchestration through public marketplaces. Supplementary examples of client app integration are also provided. Compilation of knowledge into platform-specific form has been left flexible, in so far as implementations comply with ARTAKA's Context Event Service (CES) communication and Health Services Platform (HSP) Marketplace service packaging standards.

Towards the goal of interoperable *human* processes, ARTAKA's treatment of knowledge artifacts as a specialized form of software allows knowledge engineers to operate *as a type of* software engineering practice. Thus, nearly a century of software development processes, tools, policies, and lessons offer immediate benefit: in some cases, with remarkable parity. Analyses of experimentation is provided with guidelines in how choice aspects of software development life cycles (SDLCs) apply to knowledge artifact development in an ARTAKA environment.

Portions of this culminating document have been further initiated with Standards Developing Organizations (SDOs) intended to ultimately produce normative standards, as have active relationships with other bodies.

DEDICATION

To my mother, Carol.

ACKNOWLEDGMENTS

Foremost, this work would not be possible without the unwavering commitment of family and friends, in particular my parents. Thank you immensely for your support. As a small token of gratitude, do not feel compelled to read further. ☺

It is in great optimism I hope that the *Architecture for Real-Time Application of Knowledge Artifacts* (ARTAKA) becomes a baseline system design for the pluggable application of both established *clinical decision support* (CDS) and emergent fields such as *deep learning* into electronic health record systems, the former being primed for revitalization. No comprehensive architecture is ever drafted in a vacuum, however, and a shortlist of specific contributions deserves mention.

Reference implementation work for both *KNARTwork* and *TastyTerm* example client projects was partially aided by contract labor funded by Piper Foundation. Since initial authoring, KNARTwork is now maintained as part of Clinical Quality Framework. TastyTerm was prototyped against the Health Service Platform Consortium (HSPC) FHIR Sandbox system, and additionally OntoServer from Australian e-Health Research Centre.

The Context Event Service is informed by the work of the Object Management Group's API4KP request for proposals (RFP) response by Mayo Clinic.

The Health Services Platform (HSP) Marketplace application programming interface (API) specification initially drafted as part of this work has, at the time of this writing, grown legs on its own as is being pursued by the HL7 Service Oriented Architecture (SOA) Work Group with the goal of being balloted as a collaborative effort.

HL7 interest in the ARTAKA components – that is to say its contributors – are a fire behind the light.

Reference implementation infrastructure has been partially provided by the Health Services Platform Consortium (HSPC), currently merging with the Clinical Information Interoperability Council (CIIC): may the forest never be lost in the trees.

ARTAKA-compatible exemplar event handlers have been partially developed by students participating in the Health IT Innovation Collaboratory (HII-C) at Arizona State University as part of the Vertically Integrated Projects program. Dozens of students have already contributed to HII-C in multidisciplinary forms and are expected to continue for the life of the program. ARTAKA will hopefully serve as a basis for integrating scads of complex CDS service concepts into high-level applications.

Sincere thanks to my advisory committee and department at Arizona State University, for the flexibility in pursuing a rather unconventional -- and elongated -- path to stand directly on the shoulders of giants casting intimidatingly long shadows.

TABLE OF CONTENTS

Page

Page

Page

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

DISCLOSURES AND DISCLAIMERS

I declare no conflicts of interest. To further avoid any potential for vendor bias and reduce risk of intellectual property rights issues, I have consciously avoided detailed investigation of encumbered technical works that have not already been subject to public disclosure. Any such basis has been cited appropriately. These works include proprietary aspects of Allscripts Application Store, Cerner App Gallery, Epic App Orchard, and any/all other restricted works. Any remaining uncited similarity is likely due to alignment of architectural principles, and should not be construed as implicit endorsement, nor statement of compatibility with, any vendor-specific solution. For these reasons, vendor-specific systems will generally not be mentioned elsewhere in this document.

All exemplar projects have been developed "in the open" and are hereby made available – if not already – under the Apache 2.0, MIT, and/or Created Commons license. Please see the documentation for individual projects for licensing terms.

Though I have published multiple works in the BMI domain relevant to the PhD program largely surrounding *data*- and *information*-level collaboration, these have largely been *excluded* from this manuscript as they are not directly relevant to the topic at hand except in small intersection. They have been provided to my program committee separately as additional works as partial contributions to program requirements and documentation of scientific contribution. They will *not* be artificially attached to this manuscript as appendices.

Excerpts on content licensing is being prepped for separate journal submission as a separate document. Formatting and structure of this document has drafted to meet other external requirements and, as such, naturally does not conform to the ASU Graduate College's published formatting requirements.

BACKGROUND

20th-century clinical decision support systems (CDSS) have gravitated around highly localized applications of localized knowledge. Through early decades, successes in computerized physician order entry (CPOE), alerts and reminders, point-to-point integrations, point-of-care information retrieval and myriad functional areas have not suffered from a lack of methodological ideas[1], competing ideals or clinical interest, but to date have collectively failed to induce an influx of technologically advanced marvels as initially envisioned.[2,3] The future has not yet arrived[4], and the multitude of fingers to point is dwarfed by the number of targets to blame.

Attempting to address roadblocks of the clinical decision support (CDS) community from such a pessimistic perspective, chronicling failures ad nauseam to attribute blame, is not particularly productive if we are set on swiftly making broad course corrections to enable the next golden age of CDS innovation. Rather, it is my intention to present a next-generation view of CDS architecture for knowledge-driven systems, such that the field may successfully incorporate a broader array of approaches and attract radically different ideas: not merely incremental improvement to specific isolated applications. The architectural approach described in this manuscript intends to bridge the gap between emergent trends in CDS knowledge authoring and management to the real-world application of those artifacts in production systems. Exploration of these mechanics is extremely important to both clinical informatics and computer science disciplines, as traditional client-server software engineering patterns do not address the nature of knowledge-driven applications, especially in the context of health standards.

Context-based content filtering, multi-user sessions, localized machine learning, workflow branch prediction, intelligent grouping, simulation, and infinite other creative support methods require new approaches to CDS architecture, and more precisely, a generalized mechanism for generalized client-side UI orchestration that incorporates incremental changes to underlying knowledge bases *without* reengineering of client software to explicitly invoke known CDS modules.

Ontologic investigation and modeling of UI concepts is not particularly new[5], but has also not made inroads in practice. Model-driven UI development has failed to upset the overwhelmingly popular use of manually-authored model-view-controller (MVC) design, generally accredited to Smalltalk and the subject of decades of variation and analysis for both user-facing and middleware applications.[6] ARTAKA's approach is extremely complementary to this reality, and is demonstrated and analyzed in practical terms.

In addition to the UI event-related discussion and reference implementation, I offer guidance on measured evaluation of concrete implementations. Particular attention is paid to a single, oft-neglected quantifiable characteristic that is vital to the future of CDS systems – and care in general – that no large organization, healthcare or otherwise, can sustain without in the 21st century: *scalability*.

Irrespective of what it means to an organization, scalability is always desirable, always critical, and simultaneously always *elusive*. We acknowledge the importance of scalability without clear consensus on its meaning, and build mammoth systems with latent belief that integration of individually scalable SOA components results in a

systemically scalable system. Books on the subject address scalability as a first-class consideration [7,8] in all aspects of architecture, but are often treated as lesser concerns in healthcare. A single function can be optimized locally by careful profiling and thoughtful local engineering decisions, but the edges between can only be scaled by requirements and design. No local code optimization can compensate for a lack of forethought on the lack of common vocabulary and isosemantically safe exchange of data structures across N interaction boundaries, solvable by proactive policy and governance. No clever database query can eliminate unsustainable economic costs of implementing a medical logic module (MLM) in production. CDS, as a field, is ripe for head-to-toe shift towards *systemic* scalability in a hot-pluggable architecture.

RESEARCH IMPLICATIONS

The importance of CDS architecture scalability as a system- and global-level clinical quality measure is a cross-cutting concern throughout biomedical informatics domains, and far from solely an engineering matter. This scope is further exemplified by widespread acknowledgement of challenges in the management of curated clinical knowledge intended to affect production systems downstream.[9,10] Collaborative bodies such as the Trust Framework Work Group have emerged to frame the problems and guide policy, but these are early efforts.

With appropriate traditional configuration management in place, deployment processes can be mitigated in a manageable manner[11,12]; but even so, deploying any change into a system affecting patient care caries fundamental danger.[13] Updating the available code system version of a terminology such as SNOMED CT, LOINC, or

RxNorm is not a simple find/replace operation when the semantics of those codes *differ*. In a prospective future where no single development team can even *be aware of* the entirety of live CIS content upon which clinical decisions are made, centrally-coordinated outage events no longer become possible. Yet, we must find a way for clinical subject matter experts (SMEs) and knowledge engineers to deploy *more* actionable knowledge *faster* in an environment of potentially exponentially greater complexity.

Further complicating matters, a patient's care is no longer delivered by a single country internist operating within the comfort of an isolated layered onion. Coordination events across networked onions introduce additional complexities of security, governance, and legal dilemmas largely outside the scope of current interoperability standards.[14]

Worse, standards developing organizations (SDO)s tend to assume that health data are inherently portable in a computationally timely fashion: another latent belief for which there is little evidence. Precision medicine studies can easily yield terabytes of raw data per patient. [15] While live CDS modules are unlikely to draw much value in raw query of en masse data[16] -- as opposed to highly processed and annotated secondary data derived through a vetted pipeline -- it is perfectly reasonable to expect current trends in data production and transfer to continue.[17] Certain contexts will warrant completeness beyond what Virtual Medical Record (vMR)-like models can provide. Accounting for these data sources as well as recent advances in differential privacy[18], we must architecturally acknowledge that the reality that CDS will sometimes necessitate moving computational lambdas off site, into black box environments where only

preapproved result profiles may emerge. This is not in full alignment with specifications such as Decision Support Service (DSS) or CDS Hooks. Other barriers may require the same solution, such as where collaborating organizations are not willing or able to exchange protected health information (PHI), data are partitioned vertically, or infrastructural “cloud bursting” manipulates data affinity as a function of load over time.[19-21]

The foremost contribution of this work is in definition of the ARTAKA CDSS, adaptable for organizations of all sizes, in which interoperable content may be authored, exchanged, validated, and executed in a secure and automatable manner, and such that growth of the burden is linear relative to the size of the patient population and volume of executable content. And as a secondary aim, provide guidance on the evaluation of concrete implementations rooted in existing best practices and analysis of the reference.

CDS is an incredibly broad topic. Next-generation clinical decision support will most likely not be characterized by incremental advances in the status quo of hard-coded decision logic written by knowledge and software engineers, but by rapidly adaptable, fluid systems driven by clinically-deployed knowledge modules: authored, managed, introduced, revised, and removed from active daily use on clinically-driven timelines without delay or downtime.

This shift is not only paramount to freeing providers from vendor implementation lock-in and associated consulting timeline and cost constraints, but requisite to the ability of HIT organizations to meet rapidly increasing demands for software maintenance. In the commercial realm, vendor organizations will not be able to satisfy resource demands

if a software engineering effort is required to implement every instance of executable knowledge in a context orders of magnitude more complex. It is an implausible path forward.

In a recent case within Veterans Health Administration (VHA), a questionnaire was developed to assess patient suicide risk. The form was defined using the Health Level 7 (HL7) CDS Knowledge Artifact specification for content and control flow, bound to underlying terminology identifiers, and demonstrated to automatically render in a manner close to the original intent. While not perfect, this work has not been incorporated into the VistA Electronic Medical Record (EMR) system in its true form for a number of reasons, most notably that doing so would have required a substantial software engineering effort commitment.

In the future, questionnaires and other forms of documentation templates are but one example of actionable clinical knowledge that *must* be fully compilable, validatable, and deployable across the entirety of a networked organization in response times limited only by safe human processes. More recently, VHA has expanded its effort by development of approximately ~104 distinct HL7 CDS Knowledge Artifacts, none of which has attained plug-and-plug operability in any known system. It would be clinically unacceptable to accept an enterprise architecture stretching the implementation response turnaround through years of backlogged artifact queues and artifact-specific services and tooling. The state of the art in knowledge-based systems, however, does not appear to have dramatically improved over recent decades.[22]

From an informatics perspective, reducing these burdens requires the entire HIT stack to be architected to accommodate generalized execution of hot-pluggable content. Pragmatically, this operation of the executor framework must be managed by a HIT organization, with knowledge-as-software content managed by SMEs and clinical informaticists, and process governance and compliance at the separation of concerns boundary. The SOA defined by ARTAKA is intended to accommodate standards-based knowledge modules from domain-specific SMEs, support the validation process of individual modules to stakeholder parties, and integrate alongside or on top of a live EHR platform on demand in a fully automated manner when clinically approved. The entirety of this process must be based on open, practical standards from top to bottom, and depends on provider and vendor acceptance to garner widespread adoption.

Within this vision, bridging between formally-represented clinical knowledge and cross-institutional execution requires, amongst other things, openly available knowledge compilers that translate technology-agnostic models into type-specific runtime code, platform-agnostic service container packaging, and automatic binding of data and service dependencies at runtime using late binding.

RESEARCH AREA COMPETENCIES

I am a career systems architect with over 15 years in applied software engineering, with 8 in BMI disciplines in a number of roles spanning public and private organizations. I currently serve as a systems architect for Veterans Health Administration in Knowledge-Based Systems, Chief Systems Architect for Biomedical Informatics at Arizona State University (ASU), and am a contributor to the Object Management Group (OMG), Health Level 7 (HL7), Health Services Platform Consortium (HSPC), OSEHRA, and assist in departmental efforts in BMI at ASU. Prior published works include papers on the scalable, secure syndication of collaborative bioinformatics studies over the Internet, perspectives on the nature of biointelligence, separate patent applications on the use of hypergraphs for logical biomedical modeling and visualization of N-of-1 study data, and other topics, in addition to software projects and libraries in the BMI domain. I completed my written and oral comprehensive exams in 2017.

Prior, I have completed AA and AGS degrees, an undergraduate BS degree in Computer Science and Software Engineering, and Masters in Business Administration (MBA). I returned to ASU briefly to teach OOA&D concepts and database-backed enterprise application engineering as adjunct faculty at the polytechnic campus under Drs. Gary and Lindquist.

Outside of BMI, I have focused on architecture and development of large-scale applications and middleware, and have broad experience in private commercial practice, research institutions such as the Translational Genomics Research Institute and Van Andel Institute, and public entities such as VHA.

I have been an active professional in BMI for 8 years and have been collecting incremental thoughts on this subject since enrolling in the program. As a cross-cutting topic and result of my personal interests and background, most literature sources have come from two general areas: medical informatics and computer science. A perusal of the bibliography reveals most publications fall into these areas. With broad experience in many related disciplines, I am qualified to contribute to this specialized intersection of CDS and applied computer science.

MISUNDERSTANDING MAINSTREAM WORKS

All systems have limits: some by constraints imposed by implementation concerns, others by innate qualities of a specification or architecture. Standards-oriented solutions, in particular, are prone to the latter due to the inherent inability of implementors to create extensions universally compatible with all implementations of the standard proper.

The ARTAKA design is informed and motivated by such boundaries of current and upcoming standards. Context Event Service, as the prime example, is designed to complement some of these inherent limitations without introducing a competitive turf war for well-covered use cases. This section expands on notable misconceptions, and the relation to mitigating capabilities presented in ARTAKA and constituent services.

FAST HEALTH INTEROPERABILITY RESOURCES

Health IT's predominant standards body is Health Level 7 (HL7), stewarding decades of mission-critical interoperability specifications such as "v2". At present, personal observation has revealed huge expectations and hype surrounding Fast Health Interoperability Resources (FHIR), as well as anything leveraging FHIR as a dependency. Anecdotally, almost all new project efforts at HL7 appear to somehow reference, if not directly incorporate, FHIR.

FHIR originated as a REST-like approach to health data interoperability[23] suitable for modern service-oriented architectures (SOA) primarily exchanging JSON or XML documents either as middleware or directly with client applications.[24]

PLATFORMS AND CLINICAL MODELS

One of the most common mischaracterizations of FHIR is that it provides all core data exchange structures to effectively create a data interoperability bridge between multiple organizations. This is categorically *false*. FHIR, by its own description, "..is a *platform* specification".[25] FHIR provides base data structures and semantics for a FHIR-based API, but is somewaht silent on the selection of the particular clinical *models* being exchanged, harkening to equivalent widespread issues with HL7 v2.[26,27]

In non-healthcare domains, APIs are commonly developed to provide lock-step alignment between logical representations and underlying physical implementations, typically manifested in some form of database. The semantics and constraints of each field are declared in the API, and deviations usually result in incompatibility. Health IT provides a slight anomaly in that this style of bottom-up design is frequently frowned upon as too limiting. "Local customization", particularly in clinical informatics, is generally considered requisite to a degree. For this reason, FHIR separates the concept of a "resource" into a highly customizable type hierarchy. Each resource definition provides field-level metadata that may or may not permit expansion or contraction of constraints. To achieve true semantic clarity of what is represented by a given resource instance, a clear resource "profile" must be followed. Argonaut [28] and US-Core[29] are prime examples of interoperability efforts focused on developing clear semantic profiles of FHIR resources.

Due to the complexity of the ecosystem, the current likelihood of defining a profilable context event model for orchestration of *all* client applications from compatible

backend knowledge services is nil. Doing so would fragment client applications to, most likely, only support a vendor-specific profile in ways that preclude simultaneous support for competing Context Event Service implementations. Further, as Context Event Service is primarily an event broker to/from knowledge agents, event profiling will not scale to potentially thousands of underlying agents operating in concert. Any such model needing to operate in this environment must be lightweight and semantically crisp. This approach is in no way incompatible with FHIR, though may irk FHIR evangelists that the FHIR Subscription resource is not directly used by CES. Also, CES' real-time push mechanism relies on standard HTML 5 Server Sent Events that are not supported, nor mentioned by, FHIR Subscription as of the STU 3 release. (See the section on HTML 5 Server Sent Events (SSE) for details.) CES provides for complete flexibility of data models and does not currently define a "profiling" mechanism of the core (and only) event structure. The section on Event Profiling & Developer Collaboration provides additional discussion.

Module Maturity

FHIR is balloted in consolidated revisions; however, the robustness of individual resources varies greatly according to a governed "maturity level" classification system. HL7's granular maturity model used in FHIR provides a sensible compartmentalization mechanism to allow natural ebb and flow of loosely related areas of the overall specification. The caveat is that consuming parties can easily view the high-level state of the specification – notably the "Standard for Trial Use" (STU) moniker – and infer that this applies equally to all portions of the work. This is not so.

FHIR “Level 5” resources under Clinical Reasoning Module are largely categorized at maturity levels 0-2 at present[30] on a 0-6 scale. The area is subject to significant revision, has limited real-world use, and is not in complete lock-step alignment with the XML document-based HL7 CDS Knowledge Artifacts specification.

This is notable in addressing a reason that the ARTAKA design is not bound to any specific model or method of knowledge representation: no generally accepted means exists in executable form in practice. FHIR Clinical Reasoning Module is likely to gain much traction in the future, but at time of initial experimentation did not provide sufficiently mature definition to warrant detailed experiments.

Of additional particular interest are FHIR’s PlanDefinition and CarePlan resources. These types loosely correlate with core data structures in process models suites such as BPMN, CMMN, and DMN – collectively *MN -- and underlying execution engines. The *MN suite, governed and stewarded by Object Management Group (OMGG), is of great interest to the clinical informatics community, as they are general-purposes solutions with numerous existing vendor solutions already in production use globally. The intersection of the *MN and FHIR worlds is currently under investigation by the OMG Healthcare Domain Task Force (HDTF), conveining at work group meeting in 2017 and 2018 thus far.[31]

The HDTF has released a *MN “Field Guide” [32] providing best practices for knowledge engineers tasked with production or consumption of *MN in healthcare. In current work, the community is exploring the exact relationships between OMG *MN

and HL7 FHIR down to the field level, with the aim of providing both whitepapers and proof-of-concept prototypes illustrating the work.

As it relates to ARTAKA, the resultant runtimes produced by the HDTF or similar efforts are intended to manifest as knowledge *agents* discussed in Agents, though the platform-specific means by which a specific process model is agent-ized is beyond the scope of this manuscript. Regardless of the means, process- or workflow-focused agents are easily able to detect and emit relevant changes using the same CES mechanisms used for client orchestration. An external change to specific CarePlan, for example, would trigger a lightweight event carrying the URI of the instance, URI representing the nature of the change, and originating actor.

ARTAKA, via CES' event backplane, unifies event-driven thinking with frontline user applications, and can be used to bind applications, workflow runtimes, and CDS services without introducing strong coupling and point-to-point integrations.

HATEOAS

REpresentational State Transfer (REST) is an architectural style. There is no governing board validating adherence to REST principles, nor any form of certification or adjudication on usage of the term. As such, colloquial usage of "REST" generally does not mean strict adherence to the principles outlined by Fielding.[33] "REST" in practice usually implies a service accepting and returning JSON and/or XML-structured documents, manipulated with standardized HTTP verbs, at URLs paths declared as nouns. Beyond that, very little can be reasonably assumed.

Of the most ignored concepts of REST is the notion of Hypertext As The Engine Of Application State (HATEOAS). In essence, HATEOAS encourages changes to front-end application state based on explicit transitional guidance provided by services as part of their native resource specifications. Despite the positive long-term outlook of REST, almost no REST services, including FHIR, provide a focal means for service-driven contextual state management, instead leaving application usage purely in the hands of the user.

This is fine. However, ignoring the *ability* to do so in individual API definitions is extremely problematic when use cases call for direct, real-time, server-side client orchestration. CES' API directly supports this type of push-enabled HATEOAS, regardless of whether FHIR forms the basis of the domain model.

SMART-ON-FHIR (SOF)

The SMART-on-FHIR specification originated as a common way of "launching" a front end-only user interface against FHIR resources managed by an EHR or other system of record. It is fairly simple to understand and establishes an appropriate separation of concerns and degree of abstraction between data and application layers in a SOA. It is already supported to significant degree by the leading EHR vendors, with entire revenue models designed to profit off the ecosystem.

SoF's core launching method is not new technology. For technologists outside health IT, it is easily explained as existing, industry-standard OAuth 2.0 and OpenID Connect[34] with precoordinated launch parameters and custom scope definitions for domain-specific resource authorization. Other than the format of these values, there is

nothing HIT related about SoF. It is a simple precoordination of OAuth 2.0 usage by design and works well for its purpose. The specification has since moved to HL7 for standardization.

Access Controls

The danger of SoF is overmarketing of its capabilities. As effectively a "profile" of OAuth 2.0, SoF does not provide any form of context management or state propagation. The scope authorization mechanism is debatably too simple, as most examples do not show how complex, fine-grained control may be implemented as is necessary when field-level censoring is required.

Backend Services

SoF only addresses a small number of common, albeit important, use cases regarding launch of front-end clients against a FHIR authority. *SoF does not have an equivalent covering the inverse scenario* where a backend service must be distributed for use by existing clients. This is for good reason that no non-proprietary de facto standard has emerged. Elements exist, but standardized means for automated publication, validation, and distribution of CDS Hooks services, as a choice example, is nonexistent.

ARTAKA addresses these issues through the Health Services Platform Marketplace: a formalized vendor-neutral specification for "gallery"-like uses present in the SoF community [35,36], and notably compatible with SoF gallery use cases, that has specific considerations for backend services and the local infrastructure that will need to

run it. Additionally, the HSP Marketplace supports distribution of SoF application when they need to be run *on premise*: a requirement almost never addressed in practice.

The Marketplace model also recognizes the need for curation of service images, and accounts for these needs via a simple non-prescriptive state model. Role-based access and permissions are all dynamically configurable, and authentication is designed to use OAuth 2.0 OpenID Connect in a manner fully compatible with, and already present in, SoF architectures.

Through detailed review of the relevant works, coupled with additional hands-on implementation, it is apparent that these technological contributions are needed regardless of a commitment to CES. These concerns are discussed in depth in Health Services Platform Marketplace. Reference implementations have been developed and are available under Open Source license.

CDS HOOKS

At present, much additional hype exists in the clinical interoperability and CDS communities around CDS Hooks: a simple REST-like mechanism for invocation of remote CDS services.[37] For client-side engineers, it is appealing to support due to the low learning curve, ease of implementation, and lightweight nature, relative to more heavyweight standards. CDS Hooks service implementations may be called from any actor capable of making HTTP requests using JSON: effectively all web-based, mobile, desktop, and middleware types of applications.

EXPLICIT NOUN-VERB

CDS Hooks is based on the design principle of explicit invocation. Each "hook" name follows a noun-verb pattern indicative of the type of event that has occurred. [38] At time of this writing, three hooks are explicitly defined on the CDS Hooks website in addition to the template. It is reasonable to expect the official set of noun-verb hooks will grow immensely due to ease of defining new hook types.

The disadvantage to this principle is that of scalability. If the specification becomes a long-term mainstream function of EHRs, the list of potential noun-verbs may grow exponentially. While it is not technically problematic to do so, there are no "noun-only", "verb-only", or multi-noun/verb semantics. Allowing for extremely granular hooks such as "patient-selection-changed-from-to" would quickly result in an explosion in use case-specific hooks. Such an approach is unscalable, as is amplified by the profile-specific nature of FHIR resources and combinatorics of possible client and server integration.

CES' event structure allows for this, and through existing fields may actually "wrap" CDS Hooks requests, but CES defines no specific set of supported or unsupported noun/verb combinations. No central registry exists by design. Events typically carry equivalent fields, but unlike CDS Hooks, CES-integrated applications are encouraged to emit and respond to events of interest over relying on a central registry of supported CDS events. Thus, any and every possible combination of CDS Hooks noun-verb is possible, regardless of the noun or verb. CES is based on the principle of use case-agnostic events.

SYNCHRONOUS RESPONSE

CDS Hooks invocations are synchronous, allowing services to return an array of "cards" in response. Each card may include a set of related "links" to resources, and "suggestions" for user "actions".

In this regard, ARTAKA provides similar and compatible capabilities to CDS Hooks. CES' event "parameters" field is the equivalent of CDS Hooks' action "resource" field. It is an arbitrary object, optional, and may contain payload data specific to the intent of the service that must be specified by the underlying service in advance to permit client integration.

Where CES is architecturally divergent lies in the nature of determinism. CDS Hooks is fundamentally synchronous, while CES is fundamentally asynchronous. For CDS Hooks clients, the context of the response is always to be interpreted with respect to the state of the client at the time it was invoked. There is little ambiguity or flexibility: one request results in one response.

CES does not permit explicit CDS invocations, and as a natural consequence there is no inherent notion of "response". Events may crudely be characterized as observations of things that happen either in a real or simulated timeline, in past, present or future. Request/Response-style usage is but *one* type of CDS supported by this paradigm. CES' peculiar approach to event semantics may be used to implement request/response, but clients are likely better served by adopting the expectation that CDS is a fluid, integrated experience, regardless of whether or not it is explicitly asked for. CDS Hooks' simplicity is a double-edged sword: it is only capable of supporting applications when intentionally invoked, and even then, only for the particular noun-verb function referenced.

Unconventionalities of CES' event model are discussed in Orchestration, Simulation, and Time Travel, and wields its own form of multi-dimensional sword.

SIMULATION

Closely related to the pros and cons of invocation synchronicity, CDS Hooks operations are generally assumed to operate on the "real" state of data and the known world. These is no defined mechanism for addressing "What if..?" use cases needed for computing the effects of hypothetical actions. This is necessary in cases such as:

- Precomputing an output that needs to be available immediately when needed but is computationally implausible to perform on demand. "Prefetch" is a prime example of practical out-of-order execution used to deceive the user into thinking a system is faster than it really is.
- Forward-chaining of consequences from a critical datum that is censored, unavailable, driven by chance, or where a state must be assumed but is not observable.
- Triggering of scheduled services that cannot wait on globally defined wall-clock intervals. For example, a patient data sync agent normally executing at 2am daily may need to be triggered at a highly accelerated redefinition of 2am.
- Finding systemic optimizations via continual revalidation of the SOA using evolutionarily updated algorithms and models, such as prediction of staffing needs.

CDS Hooks services implemented with strictly functional semantics, where invocations are isolated and carry no side effects, may be safe to use for simulation purposes, but this is far out of scope from the intent of the specification. Early attempts to use simulation methodology in clinical practice such as at Georgetown University Hospital in the 1980's[39] were not long lasting.

Cellier discusses in a 1977 ACM SIGSIM paper, "..there exist problems which cannot be modelled in a proper way by either purely discrete or purely continuous simulation elements"[40], and given the highly complex nature of clinical environment it is likely that any ARTAKA or ARTAKA-like platform will exist in a middle-ground where discrete observations are the currency of event publication and subscription, but change to sources of record occurs continuously and cannot be assumed to align with any discrete event.

ARTAKA aims to allow for the application of knowledge not only for "real" data and contextual states, but *universes of belief* for simulation of potential outcomes. The mechanism is further used as the basis for orchestrating client state changes as a simulation of past user events that did not occur. Knowledge agents behind the curtain of the CES API always receive and emit events scoped to a given *timeline*. In functional cases, whether or not these events occur on real or simulated timelines is largely immaterial.

The definitions of *timelines* and the notion of a clock as an malleable event generator is firmly rooted in decades of existing research and best practices in the simulation and modeling domain.[41] These are not mainstream concepts in HIT,

however, and ARTAKA attempts to bring numerous beneficial aspects of discrete-event and continual simulation, not uncommon in other domains[42,43], into the realm of CDS. Law et al[44] provided a comprehensive breakdown of such approaches in the early 1990's.

Simulation architectures generally do not use the real-world clock to drive processing cycles. Doing so would bind the time to complete a simulation to the equivalent duration of wall-clock time, largely defeating the purpose of a given simulation if obtaining a result requires waiting until after the real-world consequences would have occurred. Use of proportional coefficients to change the duration of a second still needlessly introduces delays for quickly running and no-op cycles. This is architecturally solvable by redefining time itself to be *event*-driven as opposed to *clock*-driven. ARTAKA's smallest, defined, discrete increment of such clock tick events is one second. The nature of CDS as a specialized form of simulation is discussed in greater detail in Orchestration, Simulation, and Time Travel.

Data Access

CDS Hooks' allows for two ways for services to fetch data:

1. The caller fetches FHIR data directly, based on a template set of queries declared by the service.
2. The caller passes credentials to the service, and the service is allowed to masquerade on behalf of the caller.

CES allows for both styles, though as a general security principle the second method should not be used in any system. Sharing credentials is a highly flawed approach to data access authorization, not the intension of the OAuth protocol flows, and is extremely troubling to see in the official specification. [45]

ARTAKA is generally assumed to the operating in a trusted environment, such that disclosure of user credentials is never necessary. Caveats to this approach are discussed in Experimentation, Evaluation and much deeper future investigation is needed.

CES also differs from CDS Hooks in that FHIR is supported, but not required. FHIR resources have been used in ARTAKA experiments, but no particular data representation or model is presumed since most CES events typically only pass data by URI reference. The decision to avoid passage of fully hydrated data structures in ARTAKA is rooted in systemic scalability in highly orchestrated environments.

In a future state where CDS is not a "periodic" invocation, but a constant, ongoing stream of orchestrations across client and server boundaries, the event volume will be innately high. To compensate for the reality of network and computing limits, payloads must compensate by shedding as much weight as possible. CES does allow for heavyweight payloads to support CDS Hooks "card"-like cases, but usage is sanctioned only to cases where it is truly necessary, such as when performing information retrieval operations from internal databases that cannot be accessed directly from the client, or in transmitting predictive modeling outcomes[46] not permanently stored.

APPROACH AND SCOPE

As a cross-cutting research interest in CDS architecture, the architecture development approach has been heavily based on prior works from clinical informatics, approaches to scalable systems design from generalized computer science, and standards across both.

Existing specifications and tools have been used to the greatest extent possible, with a strong preference towards solutions with very permissible licenses. In the case of software, this entails the MIT, Apache 2.0, and BSD 2- and 3-clause licenses, and excludes any viral or overly restrictive license such as the GPL and AGPL. This decision is in 100% alignment with the Health Services Platform Consortium (HSPC) licensing policy on the same subject.

ARCHITECTURAL COMPONENTS

Previous works in published CDS architecture[13,47,48], as well as expert consensus [49], tend to share a number of thematic elements regarding system boundaries. Typically, there exists an authority of patient records (usually an EHR), source of external knowledge, externalized services for applying the knowledge to patient records, connecting interfaces, and an overarching "platform" and/or enterprise service bus (ESB)-like framework governing integration practices. Individual cases warrant variations, but this is a common baseline. Greenes' "conceptual model of CDS design components" [4] aligns with this observation as the most generalized form.

This separation of concerns is not due, however, to HIT-specific causes. Rather, it is a natural expression of externalizing service functions from a centralized source of

truth in a service-oriented architecture. In addition to present-day hybrid cloud architectures designed specifically for CDS[19,50], infrastructure-as-a-service (IaaS) vendors expose generalized external lambda execution though services such as Google Cloud Functions[51], Amazon Lambda[52], and Azure Functions[53] that potentially may be leveraged in distributed CDS architectures, though none were developed specifically targeting CDS or healthcare.

The ARTAKA design augments Greenes' generalized CDS conceptual model by explicitly addressing a number of next-generation concerns, and in the case of CDS invocation, provides for reversal of the paradigm entirely. This fundamental departure in the nature of invocation flows brings with it a new set of pros, cons, and qualities that do not always directly map to traditional service integration patterns.[54-56]

EVENT ORIENTATION

Synchronous request/response-style calls, such as those used in the hypertext transfer protocol (HTTP) protocol of the world wide web (WWW), are a straightforward way of integrating a CDS (or other) service. Design, implementation, and testing are limited to linear, blocking calls that do not inherently introduce non-deterministic aspects to client software. Unfortunately, this can also impose strict design limitation on CDS designs for live user-facing applications, in that certain actions must be traceable back to some user-invoked action. This is not a "bad" paradigm, per se, but tends to introduce busy polling operations when needing to implement asynchronous client-side event handlers.[57] This was the de facto approach used to implement push notifications in web applications for many years.[58]

ARTAKA *reverses* this paradigm. All operations are *asynchronous* unless otherwise allowed, though it is not always possible to do so. Further, clients do not explicitly "invoke" any particular CDS module or function as conventional wisdom would dictate with respect to HL7 Decision Support Service (DSS) or CDS Hooks. Instead, any MVC-based client is provided with a real-time connection to a backend event delivery system, the *ARTAKA Context Event Service* (CES), responsible for brokering events related to the user's activities and implied cognitive processes to the relevant CDS services of interest, and asynchronously triggering those functions. For those accustomed to client-initiated CDS such as CDS Hooks and DSS this is likely to create some initial discomfort, though as we will demonstrate, it is not exclusionary to continued use of explicit MLM invocation. In fact, explicit MLM invocation is extremely easy to support in ARTAKA, though a response, if any, will always be asynchronous to the invocation.

The HL7 Infobutton Manager standard, a straightforward and practical information retrieval mechanism, is also easy to emulate in either a manner indistinguishable from true Infobuttons, *as well as* a fully automated mode in which information is delivered as a response to UI context, thus potentially designing out user clicks streamlining the delivery of context-sensitive information.

Outside of backwards compatibility, ARTAKA's paradigm change is due to several primary reasons:

1. **Scalability**. It will be impossible for application developers to continuously integrate with new CDS Hook cards types, DSS modules, custom service

application programming interfaces (APIs) etc at a rate mirroring the growth of clinical knowledge.

2. **Separation** of concerns. ARTAKA CES clients have no awareness of the underlying knowledge "modules" being used within the black box, unlike DSS where clients must have *a priori* knowledge of, and encode explicit reference to, known support modules.
3. **Generalization**. Much like the generalized conceptual CDS model, it is not inherently clinical, and in fact, makes no reference to any clinical concept in specification form, other than relying on knowledge from the clinical domain. The design may thus be reused to apply knowledge in other domains.
4. **Modernization**. Fielding's dissertation covering *representational state transfer* (REST)[33,59] changed the way architects and developers think about systems integration, though Hypertext As The Engine Of Application State (HATEOAS) is not nearly as well understood as state representation itself.[60] Anecdotally, it is rarely applied as intended. ARTAKA provides for a conceptually similar form of HATEOAS in that server-side logic is able to provide actionable state transition guidance, based on knowledge, that may or may not be applied by a client.

Full-Duplex Message Initiation

Moving to a purely event-oriented CDS paradigm requires clients to participate in both client-initiated and server-initiated communication. On the WWW, this has been historically challenging for clients to implement due to the stateless, client-initiated

nature of the HTTP v1 protocol. The stateless design of HTTP v1 did not provide a method for server-side software to "push" an unsolicited event to a client, further complicated by firewalls, network address translation (NAT), and other network-level restrictions. For SMART-on-FHIR and other web-based clients, this is now possible using HTML Server Sent Events (SSE), and/or Internet Engineer Task Force's (IETF) standardized WebSockets protocol. Neither SSE nor WebSockets, however, address application-level messages structures that ARTAKA defines.

Temporal Session Context

Event-Condition-Action (ECA) rules are an example CDS method typically used to apply stateless logic; given a triggering event and retrieved set of data conditions, execute some action. In the case of stateful user sessions, though, there may not be a single discrete event to which a CDS action applies, but rather a *fluid contextual situation* based on recent activity implemented using complex event processing (CEP). For example, given the user:

- is an endocrinologist,
- searching a local knowledge management system for diabetes management guidelines, and
- is now looking at their calendar in a completely separate application

..an ARTAKA-based system may determine the user may be interested in an overlay of all patient appointments over the next 5 days due for blood tests referenced by the guideline: based neither on any *one* event, nor selection event of any particular patient

record, but because they are engaging in a *stream of cognitive activities* and are likely interested in this knowledge. This is analogous to the highly targeted content narrowcasting approaches used in other domains, most notably media and marketing companies such as Internet-based television service Hulu, or cross-system click tracking by Facebook, Amazon or Google.

ARTAKA provides CDS services with a longitudinal model of user events through a Temporal Event Store (TES) referencing generalized ontologic event types, application-specific MVC actions, data targets, timestamps, and event timelines for simulation and UI orchestration purposes.

GUIDELINES FOR IMPLEMENTATION EVALUATION

Care providers must be able to gauge the suitability of CDS solutions such that realistic expectations may be set and programmatically validated across the board, as opposed to purely qualitative checklists centered around IT policy.

For purposes of evaluating systems built on ARTAKA principles, I suggest a shortlist of constituent qualities requisite for any CDS artifact or CIS to be scalable to the organization, across three interrelated areas:

- Human: burdens placed on people, acts that they perform, and the rules that govern them.
- Technological: limits of mathematical feasibility of computing systems, short of significant leaps forward in computer science. This is strictly constrained to the algorithmic and physical limitations of currently-available devices within reach of modest organizations.
- Economic: dollars required of local institutions and impacts on macroeconomic ability to produce and consume as a function of public health.

In this broad-stroke definition, SME, software development and informatics development:

- Times are human.
- Prices are economic.
- Devices are technological.

Heuristically, implementations of the architecture may be quantitatively evaluated using a number of measures spanning all three pillars of interest. These factors include:

NEAR-ZERO ECONOMIC VARIABLE COST

CDS MLM projects, as with software, are notoriously costly to implement and maintain sustainably. Personnel costs are high, especially in highly-regulated expert domains such as healthcare.[61,62] In ideal form, the economic burden of incrementally-improved CDS systems is largely limited to the fixed costs of deployment and budgetable costs of maintenance. In terms of variable costs, the growth of CDS modules in a system should **not** statistically correlate to support costs. These criteria are intended to encourage a number of downstream behaviors, including:

- Trending towards test-driven development (TDD) principles such as unit- and integration-test automation as first-pass regression validation that generate concrete coverage data using clinically-meaningful, representative test cases.
- Dissuading adoption of solutions that require human-intensive custom local data and service bindings, such as the well-known curly braces issue of Arden Syntax[63] standardized and endorsed by HL7.
- Preference toward developing medical content across organizational boundaries in the open, using unencumbered licenses devoid of usage fees.
- Leveraging non-healthcare-specific content development processes and tools using well understood, generalized methods.

POSITIVE DIMINISHING RETURNS

Borrowing again from economic theory and strongly related to reducing variable costs, economic scalability of a solution should not suffer from negative incremental returns until beyond the projected upper limit of the system as a whole. Doing so reasonably protects the organization from no-win situations in which *any* change to *anything* is guaranteed to incur negative net value.

In a future of CDSS with modules routinely being imported, authored, revised, tested and decommissioned fluidly from live production use, local architects and other engineers *must* negotiate realistic order-of-magnitude upper-bound N values to assure that, during the expected useful life of the system, the critical value is not surpassed such that module increments result in negative utility. No system nor individual device can be deemed scalable if it cannot define the claims by which is should be evaluated.

Diminishing marginal returns apply not only to horizontal scalability, but vertical scalability as well. In a given assembly line with a number of stations between 1..N, there exists a point at which introducing an additional station to divide-and-conquer workflow activities no longer improves the utility of the overall site. Over-partitioning activities in a CDSS in a noble effort to separate concerns can have the opposite effect due to the added complexity of over-compartmentalization. Separation of concerns is often in competition with minimization ofcomplexity, as the total number of system interfaces, N, relates to the potential number of system-system integrations exponentially according to the function N(N-1)/2, not linearly.

SME-DRIVEN MODULE LIFECYCLES

The translation of knowledge acquired by clinical SMEs to machine-readable and executable form is an imperfect manual art requiring a mix of clinical familiarity, technical aptitude, and tribal knowledge. The specifications reflect this. Expecting SMEs with in-depth specialized domain experience to deal with raw and highly technical XML, JSON, proprietary syntaxes in addition to authoring actual content is unrealistic, and the system must support highly assistive and restrictive authoring applications such that SMEs are comfortable representing knowledge with user-friendly tools and with only minimal support by clinical informicists.

CLEAR ARTIFACT INDEX REQUIREMENTS

An unintended side effect of "separation of concerns" design practices is an inability to optimize across interface boundaries. In fully integrated systems, such as silicon-level system-on-chip (SoC) designs[64], individual components may be optimized with full knowledge of the inner workings of every other component. This tacit knowledge provides for both design-time and validation-time evaluation "white box" testing.[65,66]

In microservice-oriented architectures such as ARTAKA, individual knowledge executor agents are assumed to be black boxes unless otherwise specified. In cases such as an ARTAKA Patient Corpora Cluster (PCC), a specialized array of executor agents, this means that queries cannot be assumed to be optimized in accordance with existing internal indexes. A degree of cooperation and goodwill must exist between developers on both sides of service boundaries to assure the whole is not overly burdened by the overheating of its parts.

FAULT TOLERANCE

All things eventually fail. With all components of a system exhibiting less than 100% availability rates, networked systems are thus cursed with their incrementally added components resulting in an exponential decrease in the availability of *all* parts. This effect on the mean time between failure (MTBF) is a recognized issue of service oriented architecture[67,68], and must be addressed through diligent design practices to ensure that services at the terminal nodes of the dependency graph do not cascade into complete systemic collapse.

AVAILABILITY OVER CONSISTENCY

Medical systems, in some ways, have clearer design priorities than in other domains. EHRs systems cannot go down every time a slight change to a business process must be made. In similar vein, updating a software process in such a way that requires a temporary component outage should not affect the availability of the EHR or other systems. As a rule of thumb, availability of the system should trump hard guarantees of eventual consistency. The ARTKAK Patient Corpora Cluster is notably assumed to exhibit this characteristic.

ELIMINATION OF VALIDATION OVERHEAD

ARTAKA views declarative knowledge as a specialized form of software. This is not only a philosophic stance, but practical, as it permits decades of study and applied work in automated regression testing to be applied to the validation of clinical knowledge.[69-71]

All ARTAKA agents should provide comprehensive full-suite regression test cases bundled with agent images distributed through ARTAKA Marketplaces. This allows for on-site integration testing of 3rd-party agents and is intended to drastically reduce the "implementation time" of licensed agents.

Security

Putting security into a manuscript concerning dynamic CDS architecture may seem slightly tangential, however, a portion of analysis of the experimental reference implementation is dedicated exclusively to vulnerabilities in future-facing coordinated CDS. Issues tend to originate in standards such as HL7 v2 being designed under a simpler presumption of trust between systems. Today, in-flight specifications generally do not include information security considerations far beyond their pre-Internet lineages, and shallowly recognize domain-agnostic work in the area.[72] The bigger the target, the bigger the attack surface, and it should be expected that 21st-century CDS will unfortunately be accompanied by targeted attacks via PHI, CDS, and even terminology-level vectors. Failure to account for the CISO's concerns in our assessment of systemic CDS scalability could have grave consequences at the global level.

Medical information security is an exceptionally important topic in present-day informatics, and warrants significantly greater attention in the research community. Interoperability outside the firewall introduces a league of attack vectors to the patient, organization, and broader population that must be addressed at a foundational level. In this characterization of security scalability, the classical notion of point-to-point

authentication, authorization and encryption is woefully inadequate. Implementations must also consider:

- **Participant identity**. The concepts of user identities, accounts, and authorities are closely related, confusing, and sometimes implemented in semantically contradictory manners. For CDS purposes, clinicians must be able to coordinate across care boundaries by participating in inter-network architectures, reciprocally recognizing local identities without imposing unreasonable burden on the issuing institution.
- **Audit trails**. When breaches occur, forensics requires logs. In a large care network, comprehensive log availability to digital first responders is nearly impossible without pre-coordination with developers, full cooperation of local authorities, and dedicated staff efforts, possibly across jurisdictions and aided by court orders.
- **Tamper resistance**. Structural and semantic interoperability can allow for safe clinical exchange of information, but does not inherently verify that the exchange *itself* was safe. Complicating the matter, what amounts to "cleaning up" or "data mapping" by an exchange or other intermediary party may qualify as falsification or fabrication to another.
- **Accidental disclosure and discrimination**. Scrubbing of exported data to contain only "relevant" information is extremely difficult in a medical context, as the definition of "relevant" is a localized concept. Prevention of "leaks" is further complicated by the question of whether internal computational agents *should* be

allowed to access data that the user is not. ARTAKA agents are assumed to operate in a trusted environment, but no explicit architectural decision is mandated on whether a given agent may operate in a fully authorized manner or be scope constrained to the abilities of an active user, if any.

- **Timing**. A drawback of event-driven system design is sensitivity to event timelines and order. Foundational infrastructure such as the Domain Name System (DNS) have been subject to such vulnerabilities[73], as have web applications[74]. Attacks are real, and implementations need be cognizant that rouge actors may intend to induce undesired outcomes through generation of carefully orchestrated event timelines.

IMPLICIT LICENSING OF TRANSITIVE DEPENDENCIES

This content has been extracted into separate paper and will not be reproduced here. In short, for production environments it is necessary to limit the effects of proprietary content and software licenses on the entire system. Organizations should define a base set of policies and permissible licenses such that all actors need not be burdened by complicated low-level programmatic checks on license compatibility.

No assumptions should be made that content distributed or published through the National Institute of Health (NIH) Unified Medical Language System (UMLS) license implies “open source” or public domain terms. Consideration of the license(s) relevant to each individual work is necessary.

LIMITATIONS AND RISKS

While the intended impact of this research is to guide long-term organizational and national interests in design of interoperable CIS by providing a vendor-neutral view of CDS subsystem architecture based on current and emerging standards, only so much can be specified in generalized form. Flexibility to "localize" is both a blessing and a curse.

Scalability can be measured according the outlined considerations, but any evidence attained in a lab environment is purely synthetic. Research in this area generally acknowledges both the syntheticness of data and lab-constained nature of experimentation as a limiting factor. The reference implementations evaluated and analyzed have not been used in production context, and beyond peer review at both the architectural concept level and code level can provide no guarantees on suitability. The base performance qualities of the reference architecture, in particular, are only meaningful in the context of the deployment environment. Different implementations are expected to make different optimizations, to exhibit completely different performance profiles. Further, these "wind tunnel" experiments cannot account for all the complexities and compromises made in large, legacy-riddled HIT environments. For this reason, the reference implementations are not expected to be useful in definitely demonstrating scalability. They are an initial baseline for design of future systems.

In terms of risks, the biggest is scope. Even in minimal form, the number of related works, moving parts and application specifications necessary to implement an ARTAKA-based SOA is daunting, as will be the resolution of any organization-specific gaps. Within each functional area, temptation will always be present to add more, go

deeper, and fix things out of scope. This can be mitigated by keeping forefront the observation that scalability, in all things, is best achieved by *removal* of the need to scale in the first place. Similarly, flexibility to future implementers is granted through concise, clear interfaces not overburdened in a rigid framework.

ARCHITECTURAL OVERVIEW

Implementing the ARTAKA paradigm requires a distinct set of interoperable components described in this chapter. Once present, knowledge-driven services may be "plugged in", provided they meet the interface specifications of the overall system. Each component will be detailed, with discussion focused on:

1. The *original* aspects of the architecture, and
2. Items necessary for plugged-in service implementations to be exchanged across implementations.

While we will discuss components outside these categories, it is for reference purposes only, as deviations do not invalidate the utility of the architecture nor cripple the ability of knowledge services to be portable across implementations. Each component is ideally interoperable across ARTAKA implementations to facilitate cross-vendor solutions, but due to the highly complex nature of doing so, some vendor flexibility is warranted to provide for patchworks and variant architectures.

COTS components implementing sufficient existing specifications will be identified with supplemental external reading called out, as existing works based on open standards do not need to be repeated here in depth.

At a high level, ARTAKA is functionally divided into two hemispheres working as a whole and connected by several mechanisms, designed to be in line with high availability (HA) requirements of production HIT environments and familiar to

practitioners of mainstream complex event processing. The overall architecture is illustrated in Figure 1.

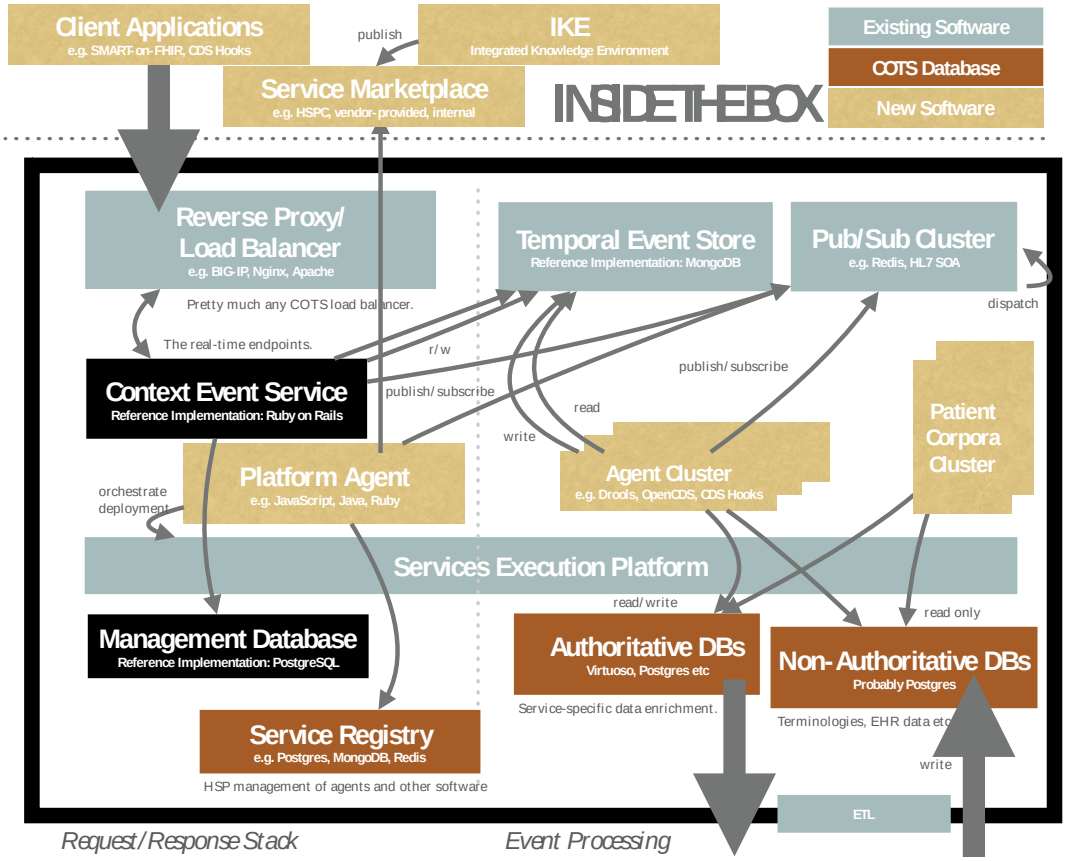

FIGURE 1 ARTAKA BOX DIAGRAM

Peripheral to the system, ARTAKA references an:

- **Integrated Knowledge Environment (IKE)** for authoring, compiling, managing, testing, and exchanging of knowledge resources in both source and executable form.
- **Health Services Platform (HSP) Marketplace** implementation and local agent for publishing, curating, validating, and deploying executable services in an automated manner.

- **Service Execution Platform** for interoperable executable artifacts.

ARTAKA's left-side systems provide:

- **Reverse Proxy/Load Balancer**. This is a generic component of HIT environments that should come as no surprise to HIT implementors, with the exception of several requirements we will identify.
- **Context Event Service (CES) API** for application integration. This is the programming interface used by applications such as SMART-on-FHIR clients to implement knowledge-driven activities. As this interface is the only public interface for the system, it will be explained in detail in dedicated chapter: Context Event Service.
- **Temporal Event Store**. ARTAKA knowledge-driven services are *stateless*, but often need to operate in the context of a stateful user session environment. All access in managed
- **Health Services Platform Agent**. This allows new service implementations to be automatically hot-plugged into a live production ARTAKA instance.

ARTAKA's right-side systems provide:

- **Triggered Agents** that receive and respond to client activities via Complex Event Processing (CEP). These instances are the heart of the system, as individual instances constitute the meat of a local deployment, are consist of compiled local knowledge, manually authored software, facades for complex subsystems, and adapters for proprietary services and specifications. They are entirely

customizable. More sophisticated implementations may integrate like-minded frameworks such as Apache Kafka and Apache Samza in a implementation-specific manner.

- **Scheduled Agents** for periodic jobs based on clock tick events.
- **Autonomous Agents** that run continuous on their own terms, regardless of an external triggering mechanism.
- **Patient Corpora Cluster** that keeps patient data in scope-isolated random access memory (RAM).
- **External Data** caches for databases and services authoritatively maintained outside the ARTAKA system.

INTEGRATED KNOWLEDGE ENVIRONMENT (IKE)

Recalling that ARTAKA views knowledge engineering as a specialized from of software engineering, the IKE is a specialized form of an integrated development environment (IDE) that is the staple of many software developers daily lives, particularly in statically typed languages such as Java and C#. The scope of an IKE includes:

- Authoring: create/read/update/delete of standardized types of knowledge documents via individual artifact editors.
- Compilation: 100% programmatic transform of declarative artifacts into executable ARTAKA binaries using platform-specific technologies. These may be fully Open Source, such as Knowledge Is Everything (KIE) including Drools, proprietary to a vendor, or combination thereof.

- Testing: At minimum, regression support for change control and validation in the context of the backend implementation. This allows authors to run and debug knowledge resources *directly* in the IKE without requiring in-depth technical platform knowledge that a clinical informaticist is unlikely to have.
- Integration: Leverage the run-time SSE API capabilities of the backend, including event topics via direct, programmatic integration of server-side events.
- Simulation: run and debug against synthetic or real patient records, when applicable to the scope of the knowledge.
- Governance: appropriately apply business processes, policies, and frameworks.
- Community: sustainable in a pluggable manner via community extensions and maintenance, ideally all in the Open Source domain as an informal collaborative effort across organizations and stewarded by several primary benefactors. Variants of this paradigm are common to most successful IDEs.

No stringent specification is necessary and maximal creativity in design of IKE environments is encouraged, however, baseline environments are necessary to spur innovation. With current momentum towards FHIR-based knowledge objects, IKEs able to launch against backends supporting the FHIR Clinical Reasoning Module are most likely to attract initial attention.

KNARTWORK

As an entry point into IKEs, I have developed the web-based KNARTwork application as a standalone tool for authoring HL7 CDS Knowledge Artifacts (KNARTs).

I have presented this work as a standalone contribution to clinical informatics tooling at the 2017 HL7 FHIR Applications Roundtable[75], and moved future stewardship of project source code to the Clinical Quality Framework (CQF) in 2017 where it may be maintained as a community interest amongst similar projects. At time of this writing, composite artifacts have not yet been standardized, though schemas have been posted informally by the HL7 KNART work group.

Using KNARTwork, knowledge engineers may create new KNARTs, load existing ones, download documents in native XML format, and preview them directly on the web using UI widgets. It has already been used by end users comfortable managing source documents as an out-of-band process using git/subversion, Dropbox, email etc. Support for FHIR Clinical Reasoning Module with automated conversion tools is a future feature of great interest.

Repository Browsing

One of more recent developments to the KNARTwork IKE is addition of a simple knowledge repository browser feature. Prior to the changes, the application provided a purely standalone experience that did not use any APIs or databases such as RDBMS or NoSQL systems, thus requiring manual acquisition of XML KNARTs.

With the changes, KNARTwork may now be launched via a special `/browser` page using a single query parameter pointing to a separate repository web server. During startup, the browser makes a web service call to the repository for a `manifest.json` file. This manifest, maintained and provided by the remote repository, references any number of distinct documents, batched into logical groups for presentation purposes. Each

manifest "item" has a name, MIME type, and set of arbitrary text tags/keywords the end user may use to search for it. Using this information, KNARTwork presents a logical web-based view of the manifest and a text search control for filtering the results. Several alternate presentation modes are also provided to dynamically regroup content by type or tag.

For all files of a MIME type supported by the KNARTwork editor, a special link is provided to open the document directly on the web. This eliminates the need for the user to download the document to their local computer, launch KNARTwork as normal, and manually open the file. All other file types may be opened directly in the browser when supported by the local operating system environment, or simply downloaded for loading into a separate application.

To help troubleshoot the availability of files referenced in the manifest, the browser provides an "Audit" button. When pressed, an HTTP HEAD call is made against *every* referenced file. All files that appear to be available are highlighted in green; files that appear to either be missing or are unavailable for security or network reasons are highlighted in red.

As of this writing, the KNARTwork manifest format is now being pursued as a baseline format for exchange of metadata between Veterans Health Administration (VHA) and Agency for Health Research and Quality (AHRQ) for artifact publication and distribution.

Customization

KNARTwork is useful as-is, but is intended to be extended by vendors to add tight integration with vendor-specific ARTAKA backend systems. With this in mind, KNARTwork intentionally does *not* come with a backend data store – nor *any* type of backend -- out of the box. In other words, it is a web-based application but does *not* have any server-side component, by design. it is assumed that the customizing party has their own mechanism for doing so. Integration with the AHRQ's CDS Connect project is currently being considered.

Content vendors with existing libraries of HL7 KNARTs and supplemental artifacts may also use the KNARTwork browser by creating a manifest.json file, either manually or via a content management system, and launching the KNARTwork browser using its hosted repository URL. The repository server only needs to allow cross-origin resource sharing (CORS) requests and be accessible by the end user, such that KNARTwork is able to load the manifest and content directly from the user's browser.

HEALTH SERVICES PLATFORM MARKETPLACE

The concept of an application "marketplace" or "app store" is nothing new. The HSP Marketplace, in particular, addresses the problem of exchanging backend service implementations in a vendor-neutral manner, which is necessary to exchange executable artifacts across SOAs with plug-n-play interoperability. A Marketplace is a location where *executable* artifacts are published for exchange, such as backend CDS Hooks implementations, runnable ECA rules derived from KNARTs, raw FHIR resource services, persistent data stores, and effectively any other service that may be wrapped in an Open Container Initiative-compatible image, such as via Docker.

In broad strokes, the Marketplace provides a server-side equivalent of SMART-on-FHIR (SoF), as SoF only concerns client-side app integration and also does not address on-premise deployments of client web applications: not uncommon in enterprise IT. The design principles are inspired by the business processes that made other computing ecosystems such as the (all proprietary) Apple, Google, and Amazon app stores successful in general-purpose computing.

The Marketplace specification also provides an "app store"-like experience for HIT professionals to explore published services, and install it locally or to a sandbox environment with a point-and-click experience similar to that of consumer desktop software. This allows:

- HIT orgs to search for new services across all participating vendors, and deploy them in a 100% automated fashion into on-prem, cloud, and/or hybrid infrastructure, using 1 or more Marketplace instances in any public/private combination.
- Developers to directly submit new (and update existing) service builds.
- Marketplace operators to curate, review, and publish vendor submissions.
- Compliance validators to automate certification activities.
- All parties to optionally authenticate with existing SSO credentials needed for SoF apps/architectures.

This concept has evolved from an HSPC side project into a proposed HL7 project with the intent on moving towards informative ballot, at minimum. A primary selling

point of the specification itself is complete agnosticism to programming language/frameworks, database, I/O technologies, and most notably, vendor. It accounts for SMART-on-FHIR client applications that require on-site deployment, but does not require use of SMART, nor FHIR. Without being duplicative of the HL7 track, the deliverables include:

- REST JSON API (Representation State Transfer JavaScript Object Notation Application Programming Interface) and platform-independent model (PIM) of specified objects, fully compatible with the OAuth 2-based nature of SMART-on-FHIR.
- Exemplar HSPC reference implementation of the Marketplace Service and Web UI/Client.
- Reference database schema for relational database management systems.

Due to the success of this concept and community-driven nature of HL7, the content produced here may not necessarily reflect what, if anything, eventually emerges from the SDOs.

The Marketplace is not technically a store, in that it does not directly facilitate financial transactions. Publication of commercial software images is highly encouraged, though the acquisition of proprietary licenses is currently an out-of-band activity between the ISV and consuming party. Commercial software vendors must allow limited use of their service images for integration validation and evaluation purposes prior to achieving a "published" state in a Marketplace, and must remain so to remain publicly discoverable.

The HSP specification aims to ease service deployment woes in an infrastructure-neutral manner, taking into consideration that most organizations run services in a combination of:

- locally-provisioned and managed virtual machines using on-premise IaaS tools
- cloud services such as AWS and Google Cloud
- bare metal machines and appliances for legacy and one-off use cases

To make automated health service provisioning possible, all services meet a highly opinionated set of interoperability criteria prior to being published in a Marketplace, the majority of which must be validated automatically. A compatible Platform conceptually unifies three areas of an IT architecture:

- Packaging - How individual service releases are produced by independent software vendors (ISVs) and consumed by IT groups.
- Registration - Definition and announcement of a service release's capabilities and dependencies.
- Orchestration - Automated service dependency resolution and deployment into the local IT architecture.

SERVICE PACKAGING

Services are declared to the Marketplace via a client such as the official web UI or other compatible application. Each service is further programmatically bootstrapped into an execution environment and may be subject to additional validation, depending on specific declared capabilities. All service implementations must be:

- Containerized into a single, OCI-compatible image, with Docker Community Edition as the gold standard and runtime verification tool.
- Ephemeral. All persistent data must be saved to an external database and declared as a service dependency, if needed.
- Programmatically verifiable. Services providing standardized capabilities must support a mode for exercising declared APIs via a "smoke test" suite triggered via the marketplace. Service submissions failing to pass smoke tests on declared capabilities should be automatically rejected. Marketplace operators may provide such tests at their discretion. Tests specific to a specific service or version may be run as an out-of-band activity prior to submission to a Marketplace. Additionally, ISVs should run applicable test suite(s) on service versions prior to review and publication.
- Horizontally scaled. The number of concurrently running containers will usually be more than 1.
- Dynamically scalable. Instances are scaled up/down at at any time. Note: For HTTP services, the use of sticky sessions is prohibited, in favor of JWTs or similar lightweight tracking for session state data.
- Single process per container task. If a service requires, for example, a single image to be run once as a web service and again as a separate worker node, these alternate entry points should be declared at publication time.

- Domain name (DNS) agnostic. No domain name, SSL/TLS context, and locale-specific settings may be hard-coded into the service. Configuration is always injected at runtime.
- Unencrypted. HTTP services are assumed to be encrypted at a separate layer providing SSL/TLS as a separate service providing reverse proxy load balancing.
- Compute constrained. Images must define the maximum per-task RAM requirement at image publication time, and manage use of memory to prevent exceeding this boundary.
- x86-64. 32-bit binaries are fine as well, but other CPU architectures are not currently supported.
- Self-bootstrapped. Every image must be able to bootstrap itself into a functional, default state with zero human intervention. This is declared at publication time and is used for service validation, local consumer evaluation testing, and for seeding production deployments.
- Stoppable at container shutdown time within 10 seconds.
- Good critizens executing in good faith that they do exactly what they say.
- Traceable. Health via process monitoring internal to the container should support prevailing standards of practice for the applicable software language/framework in use.
- Logged. Services should log to standard out/error, and must not be written to the file system. PHI/PII must not be logged unless explicitly enabled by the administrator via injected configuration flags and in a compliant environment.

Services should not be:

- Data payloads. While nothing prevents using a Marketplace for data distribution, service images are intended for software use. Large data files should not be bundled into software images. Rather, container initialization steps should be implemented that download requisite data or pulls them from a configured database.
- Hardware dependent. This is software that requires specific physical daughter cards, dongles, CPU serial numbers etc. ARTAKA has no means of binding to hardware dependencies. Future extension to image metadata will likely need to make special considerations, however, for frameworks such as OpenCL [76-78] that aggregate underlying GPU hardware into abstract interfaces.

BEST PRACTICES

Services are encouraged to be:

- SSO Aware using OpenID Connect and/or SAML. If a service requires user logins, it should be declared as needing an SSO IDP such that configuration can be provided at run-time.
- Provisionable via IETF SCIM 2 API implementations for individual and batch user and group management that is commonly used by ActiveDirectory and other identity management systems.
- Profilable via [79], which supports tracking of user session context across services for comprehensive system benchmarking.

- Incorruptible in the event they are killed without notice.
- Offline-freindly to environments where no Internet access is permitted, or is subject is quality of service disruptions.

At present, this specification is being proposed to HL7 under stewardship of the SOA Work Group.

HEALTH SERVICE PLATFORM

A Health Services Platform is the infrastructural fabric capable of running service containers packaged according to Marketplace requirements. Due to the close relationship between Marketplace functions and platform runtime, they are two sides of the same coin. While an ARTAKA platform may fully operate without any integrated service Marketplace, doing so limits the potential of automating deployment of knowledge-based (and traditional) services acquired from external parties.

The HSP specification is not a strict specification, per se, but a profile of how to use existing technologies in an interoperable way. Enterprise IT environments generally already have strategic directions on how core virtualization infrastructure is managed, and a prescriptive enterprise architecture strictly prohibiting deviations would be unlikely to make traction in existing environments.

Simply put, an HSP is fundamentally three things: a

1. Cluster of servers capable of running arbitrary services packaging according to Marketplace specifications.
2. A orchestration framework for management service distribution across the farm.

3. An agent synchronizing state changes made by 0..N authorized Marketplaces with the target state of the orchestration controller.

In the provided reference implementation, these capabilities are provided as follows:

1. A horizontal set of virtual machines running Ubuntu Linux with Docker Community Edition.
2. Out-of-the-box Docker Swarm with Rancher and Portainer.
3. A minimalistic proof-of-concept agent listening for changes to the Marketplace-managed target HSP state, capable of initializing new service instances when they are requested.

Note that the reference implementation intentionally refrains from using all proprietary functions of Docker, effectively substituting the proprietary Docker "Store" for the proposed open, interoperable, vendor-neutral specification pursued through HL7. Real-world implementations should choose a mainstream management system such as Kubernetes.

HEALTH SERVICES PLATFORM AGENT

The HSP Agent is a minimalistic service running on the local HSP listening for state changes in target platform state, as perceived by all configured Marketplace(s) as well as the local orchestration system. The HSP Agent is minimalistic in nature, intending only to bridge the Marketplace API with the native platform orchestration system.

The HSP agent proof of concept does very little, only showing how push messages sent from a reference Marketplace may be translated into action by the container platform. More sophisticated deployment profiles are the responsibility of the orchestration system. In any event, implementors are encouraged to use a stateful system for HSP management to match the stateful nature of the Marketplace's model of local capabilities.

REVERSE PROXY/LOAD BALANCER

This is the simplest component of the architecture, requiring no custom development. It may be split into separate load balancer and reverse proxy components, a single instance of a software application such as Nginx or Apache HTTPD, a cloud service, or a hardware application delivery controller such as produced by BIG-IP, Barracuda or many other vendors. The primary requirements are:

- Provides high-availability proxying and balancing for continuous delivery environments leveraging rolling deployment strategies
- Scales SSL/TLS encryption to a suitable number of concurrent connections
- Supports HTML Server Sent Events (SSE)
- Allows for manipulation of HTTP headers for cases where Cross-Origin Resource Sharing and caching controls require fine tuning
- Provides for automated certificate management through a commercial vendor or free provider such as non-profit Let's Encrypt.

Beyond these base requirements, no particularly esoteric features are required. Most COTS options suffice.

AGENTS

A knowledge "agent" is essentially an isolated event handler programmed to connect to the pub/sub cluster and, in most cases, integrate with CES' event publication endpoint. The control flow entry point for most agents is via subscriptions to individual *channels,* such as specific topics originating from client applications or other agents. The Redis instance used in the reference implementation may be used for NoSQL object storage, but is used exclusively for implementing the pub/sub pattern within the reference implementation. There is a conceptually infinite number of potential channels, but in reality, agents will want to focus on handling events on a small number of known channels, or in the case of person- or session-specific channels, channel *patterns* using the asterisk "*" as a glob character: e.g. "artaka://people/*" and "artaka://sessions/*", respectively.

THE CLOCK

Events need not originate from the client side. The reference implementation of CES is bundled with alternative entry points that emit clock tick events on dedicated channels on a *periodic* based. Ticks at a specific minute, hour, day etc is neither supported nor encouraged, though implementations may do so, if desired, to implement cron-like scheduling at the architecture level. In cases where an agent must execute 15 minutes prior to a shift change, for example, the relevant agents should subscribe at the

minute granularity and then filter for the correct ticks within the agent itself according to injected configuration parameters.

The clock is implemented as an agent that publishes on the following channels, with each event emitted after a delay from the prior event.

- artaka://ticks/second
- artaka://ticks/minute
- artaka://ticks/hour
- artaka://ticks/day

As the period of each channel is defined as the delay from the *previous* event on that channel, an hourly tick will not broadcast, for example, in an absolute sense exactly at the top of the hour. The only guarantee is that one will be emitted a relative hour *from the last tick*.

Clients may also wish to subscribe to these for testing purposes or as a means for triggering *client-side* jobs that may be awkward to account for in web applications, such as session expiry or duplicate session detection from agents implementing CEP-based functions. No useful information is included in tick events other than:

- topic_uri and model_uri fields matching the above channel names.
- Populated timestamps

Unlike most other events, however, clock ticks:

- Are **not** recorded in the TES, or otherwise tracked. Agents wanting a clock history must track it themselves.

- Do not include person_id, session_id, parameters, or other details
- Have no memory of "parent" time events, nor pointers to future events.

Ticks should not be relied upon, for example, to be broadcast *exactly* every 60 minutes. Agents sensitive to high levels of precision should subscribe to a more precise channel and check a "last run" variable on every tick.

Scheduled Agents

A scheduled agent is no different than a regular agent, expect that the triggering event is the passage of time, such as via clock tick channels. This is necessary since the containerized and elastically scalable nature of agents – which is required by the HSP to facilitate cross-implementation compatibility of agents -- makes cron-like scheduling awkward, if not impossible.

Rather than relying on traditional system level scheduling such as cron within an agent, the suggested implementation pattern is to implement *service*-level scheduling by subscribing to the most applicable "tick" event and scaling any additional intra-container workers or inter-container agent instances via service logic. This approach:

- Simplifies services from needing to be aware of the underlying job scheduling mechanism, likely specific to the platform or agent. Rather, developers author the agent to subscribe to the most appropriate "tick" event. Services may simply trust that the tick is accurate, or perform additional validation as a means of throttling.
- Allows administrators to use reuse the existing ARTAKA pub/sub cluster infrastructure, including all management tools. Without this, job scheduling in

containerized environments will vary greatly depending on the whims of the agent developer.

EXAMPLES

The CES reference includes a small library of example agents useful as templates or inspiration. The "Message of the Day" agent subscribes to application initialization and document load events from "ui2-ontology" concepts, and emits a random inspirational message back to the session that originated the event.

"Agent Smith" is subtly different, in that it listens for the cross-section of *all* user activities based on a channel *pattern* using the "*" glob character, and emits a quirky message.

"See Also", implemented by collaborator Austin Michne, is a compelling example that subscribes to text selection events of formal knowledge documents, searches for relevant literature references, and emits external links and metadata for those articles. This sample agent showcases how ARTAKA can be used for clinical information retrieval in event-driven architecture, accomplishing similar goals to HL7 InfoButton Manager but based on contextual cues over explicit CDS invocation. Further refinements of the example can use the session history provided by the TES to gain additional contextual inputs, including other recent activities in *other* applications, patient details etc.

PATIENT CORPORA CLUSTER

Agents requiring access to patient data have several significant considerations requiring planning, especially if those agents require write-back capabilities to the EHR or other patient data source. While agents may perform the heavy computational lifting, they are ephemeral, multi-tenant services, and do not come bundled with any source of patient data. As agents are a specialization of HSP-packaged services and are therefore bound by a common set of configuration requirements, data configurations such as database connection URLs and locations of service endpoints including FHIR servers are injected at runtime.

Patient data is a particularly volatile resource. In the ARTAKA paradigm it must be accessed concurrently from any number of agents, thus introducing many potential issues with cache coherency. To assist in mitigating this issue, implementations are strongly encouraged to implement a non-authoritative patient corpora cluster (PCC) for agent use.

A PCC itself is a simple concept. It is a dynamically scaled set of patient corpus agents dedicated exclusively to providing a high-performance query interface for patients of active interest. Each patient corpus agent is scope limited to the patient-specific data, potentially aggregated from multiple sources. Spawning of a new agent should, in general, take place by a PCC management agent subscribed to event channels used for selection or loading of records, such as a corresponding "*select" topic_uri with a model_uri. Upon dynamic instantiation of a corpus, the agent self-initializes based on authoritative data, denormalizing and precomputing as meaningful to local needs, and based on expected load. This is a realistic method for creating patient-specific semantic

indexes dynamically, without needing a requirement to load *everything* about *everybody* into a single memory space and index.

The same approach is recommended for population-level agent queries by aggregating only relevant data fields horizontally across records into purpose-built indexes. An influenza outbreak detection and response agent would, for example:

1. update the index at frequent intervals to receive changes made to EHRs or registries, out of band with ARTAKA components.
2. respond to events published by EHR clients already integrated with ARTAKA CES.
3. Frequently perform CEP queries agent the index.
4. emit detection events out to external clients in the form of orchestrations.

Unique Agent Characteristics

Patient corpus agents are unlike most other agents in several ways. Most immediately noticeable from a design perspective is that they are intended to be dynamically created and destroyed. Creating a patient corpus based on a selection event is reactive, but may also be done proactively based on upcoming scheduled encounters or orders. Similarly, they should be terminated from the HSP by expiry, or even self-destruct after a pre-set maximum idle period. This is necessary to prevent the PCC from growing uncontrollably with stale subjects, and keep the operational footprint more-or-less correlative to the amount of care being delivered on a given day.

Secondly, corpus agents are encouraged and expected to provide a traditional request/response endpoint with support for semantic query and reasoning. These

endpoints, when available, are intended to be used directly by other agents without needing to go through event pub/sub cycles. Internally, they are intended to be miniature standalone services, and generally contain an internal (ephemeral) database system bound to the lifecycle of the HSP corpus agent container in which in resides. This capability may be raw SPARQL, OMG API4KP [80], or other mechanism.

Thirdly, for corpus agents to be useful, their lifecycle events need to be made available to other agents. If an agent decides, for example, a potentially time consuming query should be run *now*, or deferred until after a corpus agent is available for that patient, this event is of potential interest to other system actors. While ARTAKA does not explicitly provide a registry for discovery of active agents, it is not against the paradigm to so. If a centralized service registry capability is deployed, however, it should be done generically, and ideally as part of the HSP-level implementation. A simpler form of decentralize discovery in trusted environments may be accomplished via libraries such as my journeta library[81]: based on UDP broadcast that does not use any form of centralized registry and supports client-side callbacks for both online and offline events.

Lastly, patient corpus agents are intended to be bound to a timeline. In simply cases, the null timeline may be assumed. If this is done, however, any form of simulation requiring writeback should be avoided, as doing so would untentionally corrupt the real-world patient record with that of a simulated record.

WRITEBACK

A PCC implemented at full potential with the ability to write changes both to the underlying EHR and/or PCC, then, must embue corpus agents with the ability to handle

copy-on-write semantics, wherein a write event may spur a diff from the real-world event timeline. This diff may be represented as a full-fledged copy of the given corpus agent, a diff file written to the global cache and passed along with any subsequent query to the timeline-less corpus agent, internal branching/forking mechanism specific to the semantic query engine, or combination thereof.

The PCC is a performance-enhancing architectural feature that will be needed for any significant implementation in practice, but is nevertheless optional. Simple implementations without a PCC may decide to query an EHR via FHIR, v2/v3, or similar interface directly, as needed. Regardless of the approach used for a PCC, the implementor must be keenly aware of the consequences to cache coherency, lest subtle data corrupts be discovered via dangerous runtime outcomes.

CONTEXT EVENT SERVICE

ARTAKA's Context Event Service (CES) is the primary means by which *all* client applications – SMART-on-FHIR, non-FHIR, rich clients, systems, and others -- interface with knowledge agents by means of brokering "context events": concise records each representing a discrete, atomic topical occurrence by either a human or system actor. CES allows for concurrent communication of events amongst different user sessions, agents, and even completely unrelated applications manipulating shared data models, all on different development cycles and release timelines. Events are a polymorphic type emitted from and consumed by clients to allow pluggable backend services to indirectly drive user experience without any tight coupling between client software and backend agents. It is the API bridge between clients and internal complex event processing infrastructure, and the reference implementation may be modified to substitute the pub/sub system, Temporal Event Store (TES), or other components with similar counterparts.

SCOPE

CES' functional purpose is to facilitate complex, dynamic user experiences akin to the ways:

- GPS systems aid the driver of a car utilizing maps and routes provided by backend services, wherein the system facilitates delivery of suggested "paths" through client application state while also being able to continuous adapt to unexpected user deviations.

- iOS and Android software platforms continuously monitor for contextual changes – location, time, calendar appointments etc – to develop personalized behavioral models that invoke guidance at appropriate times, such as “Leave now to get to work on time.”.
- Wearable technologies learn from baseline biorhythms to detect potentially hazardous health events such as cardiac arrhythmias.
- Internet of things (IoT) devices mesh to form event streams for home automation, security, and remote wellness monitoring.

The client-side Context Event Client reference library has been developed as both a proof of concept and functional demonstration of how interactions with ARTAKA CES are expected to occur in a number of special cases. Further, the GPS analogy is meaningful in that ARTAKA’s take on complex event processing has similarities to the way Uber’s Kafka-based infrastructure [82] supports scaling to the real-time needs of a safety-critical domain, demonstrating that the commonalities of the paradigm are not unrealistic to produce in a healthcare context.

As a proof-of-concept, the production KNARTwork application has been modified to integrate with a Context Event Client (CEC) library and configured against a live ARTAKA CES reference implementation.

From an application architecture standpoint, CES itself is an unconventional design. It operates in a stateful, asynchronous manner that in some senses is the polar opposite of traditional stateless synchronous web services and specifications discussed in Misunderstanding Mainstream Works. This design illustrated in Figure 2 was presented

longer form in a poster session at the Mobilizing Computable Biomedical Knowledge conference in 2018.

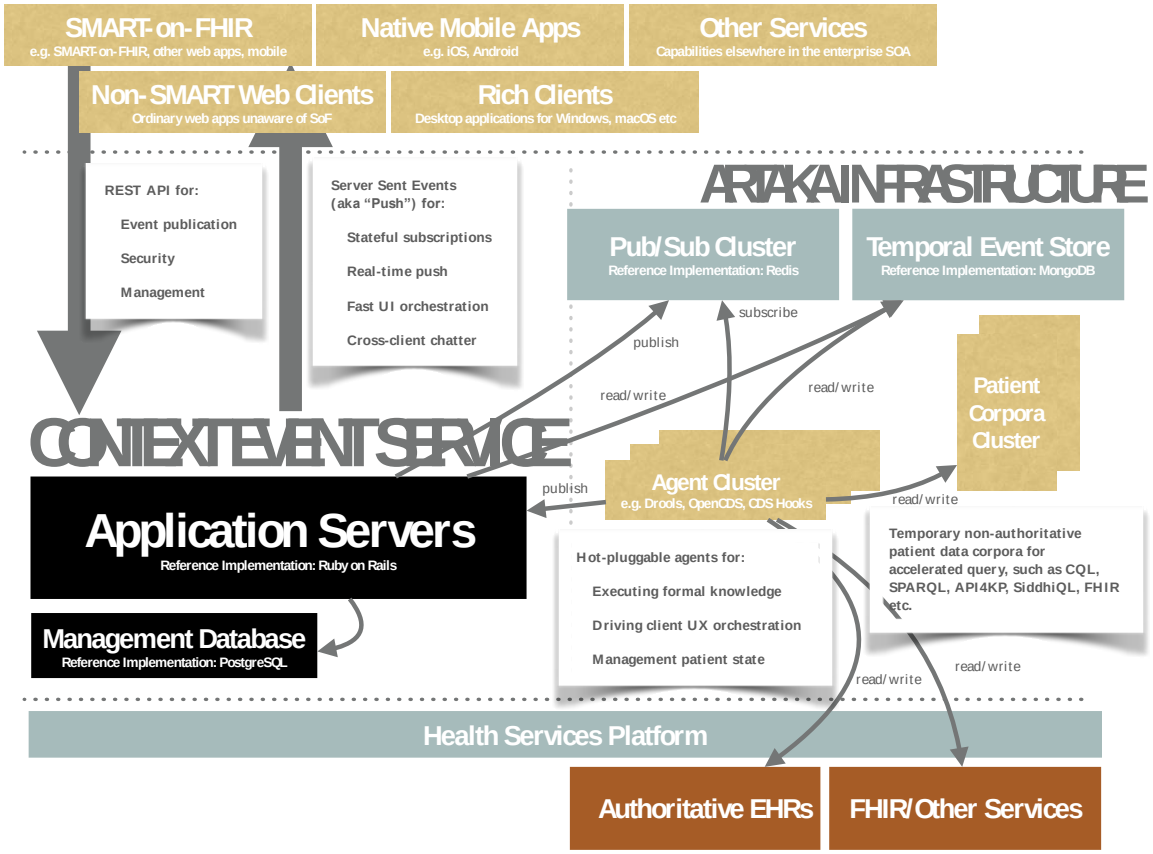

FIGURE 2 ARTAKA CONTEXT EVENT SERVICE

CES is largely devoid of a conventional APIs such as would typically be exposed by REST or SOAP endpoints, with APIs only provided for security purposes and management of slow-moving system objects. Outside of those authentication-related and management operations, CES does not actually provide a means for clients to explicitly invoke any type of CDS service. This deviates from decades of conventional wisdom

such as is applied to the design of DSS where CDS is typically invoked by clients imbued with *a priori* knowledge of existing support modules. In ARTAKA CES clients, the application is only instrumented with a CES client library. The quantity and quality of backend agents is assumed to be completely different across deployment environments. The sequence diagram shown in Figure 3 illustrates the flow of event publication to CES.

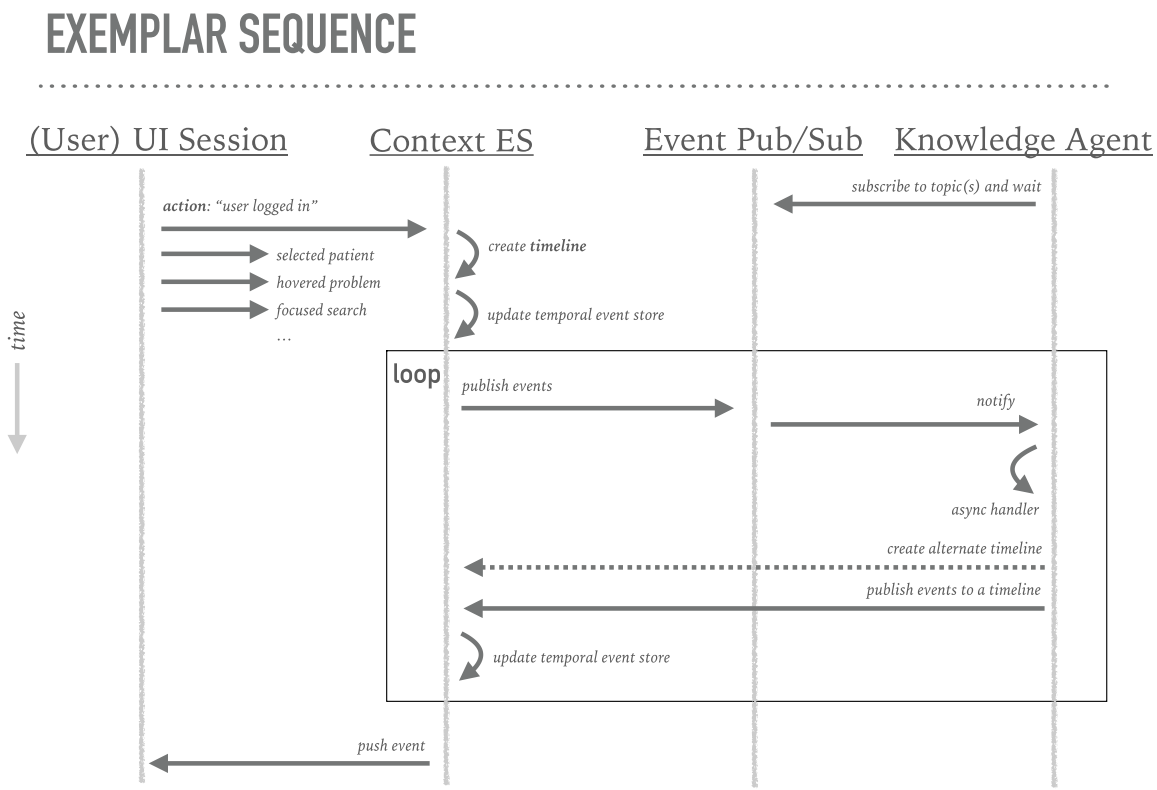

FIGURE 3 EVENT PUBLICATION FLOW

The primary purpose of this architectural decoupling is to free app developers from needing to hard-code support for specific support services, and vice versa, thereby allowing for easier scaling of development and IT practices through independent ebb and

flow of software clients and knowledge-based agents. Requiring client app developers to programmatically account for the potential invocation of tens of thousands of artifacts non-deterministically, for example, is completely implausible, especially in a world where declarative knowledge is expected to be deployed into production continuously and without system disruption. If application code must be constantly updated based on available backend agents, development cycles will be impossible to manage, leaving applications extremely fragile and constantly breaking due to fluctuating agent availability. CES thus provides the glue for event-driven μSOAs, wherein knowledge-based agents are dynamically injected and constantly changing, whether by programmed knowledge executables (KXs), KXs complied from structured declarative knowledge, and anywhere in between.

A second reason for philosophical departure[83] is acknowledgement that not every type of application support function can be forced into the request/response pattern of REST and SOAP, where the notion of a push event is completely out of scope. Historically, applications have checked for changes in remote systems by *polling* at periodic intervals, or via a similar approach known as long polling. This was done for technical reasons largely due to the stateless nature of the HTTP protocol as initially conceived.

HTML 5 Server Sent Events (SSE)

For the bulk of its existence, the WWW has relied on many HTTP requests made in rapid succession from a client (usually a browser) to a server to retrieve resources to render a page. Each request required the establishment of an underlying TCP connection

at layer 4[84], and was disconnected after each call. For decades, this resulted in immense amounts of repetitive TCP connections being made to the same server, only to be followed by further round trips of SSL and HTTP. Worse, since connections were dropped once a resource was retrieved, servers had no way to send a message to a client out of band with a client-initiated request/response cycle due to clients usually residing behind firewalls, network address translation (NAT), or other network devices preventing inbound network connections. Fast forward to present day, and most backend web application frameworks remain rooted in these architectural assumptions that no stateful client connection is present, and no reliable means of pushing content to the client exists. REST-like design and JSON data representations currently reign supreme for greenfield projects.

In recent years, leading software organizations such as Google have made major investments in providing efficient, stateful, bi-directional methods of general-purpose communication. Efforts such as WebSockets and SPDY shared great enthusiasm, culminating in the final ratification of the HTTP/2 (HTTP 2.0) specification in 2015 as well as the introduction of HTML 5 Server Sent Events (SSE). Web browsers and middleware are continuing to implement support for HTTP/2.

Modern applications often need to be notified of an event without explicitly making a request. Of the numerous mechanisms potentially available, SSE provides a modern mechanism for *subscribing* to these types of messages. SSE is unlike WebSockets, however, in that it only allows for server-to-client propagation. All event *publication* to CES from an external client thus occurs via traditional REST.

EVENT MODEL

Events are the lifeblood of ARTAKA, with nomenclature largely congruent with that of the complex event processing (CEP) community.[85] CES' internal model uses a single polymorphic "event" class for all types of events, transmitted as a JSON associative array. Events emitted from the user-facing client are typically UI-related *topics* from an HCI ontology such as ui2-ontology -- “focus”, “click”, “submit” etc -- but may also be system-related or more abstract in nature. Headless clients, for example, may control operations that do not correlate to any interactive user action. While the provided examples use the “ui2-ontology” for UI-specific event topics, CES has no specific dependence, or even awareness of, ontologic concepts. CES accepts any URI-encoded values for event fields of URI type and does not directly perform any semantic validation.

Every event carries several required fields and many optional ones, as follows:

Field	Type	Require	Example	Comments
topic_uri	**URI**	**Yes**	**artaka://ui2-ontology/hover**	**User or system action that has occured**
model_uri	**URI**	**Yes**	**fhir://Patient/42**	**URI-encoded resource of interest**

controller_uri	**URI**	**Optional for client**	**"knartwork://relatedResources"**	**Client-specific identifier for the controlling code**
agent_uri	**URI**	**Required for agents**	**"artaka://agents/hello_world"**	**Set by an agent to identify itself**
action_uri	**URI**	**No**	**"artaka://actions/orchestrations/link"**	**Relevant client-side function, if known**
parameters	**JSON**	**Yes**	**{“urls” : […], “foo” : true}**	**Any valid JSON object**
session_id	**UUIDv4**	**No**	**06dae8ff-4682-40ac-b13e-6d71c0508b41**	**Added to POSTed client events**

				automatically prior to publication. Agents must set it manually
person_id	**UUIDv4**	**No**	**06dae8ff-4682-40ac-b13e-6d71c0508b41**	**The user relevant to the event. *Not* a subject, such as a patient**
timeline_id	**UUIDv4**	**Usually**	**06dae8ff-4682-40ac-b13e-6d71c0508b41**	**Temporal universe; explained in later section**
parent_id	**UUIDv4**	**No**	**06dae8ff-4682-40ac-b13e-6d71c0508b41**	**The logical basis, if any**
next_id	**UUIDv4**	**No**	**06dae8ff-4682-40ac-b13e-6d71c0508b41**	**For cases where**

				logical order differs from temporal order

Regarding the "action_uri" field, note that it is primarily intended to be used to indicate the client-side processing function specifically responsible for the state change, if known. It may be set optionally by both clients and agents, and if so, should likely be set to a URI-encoded name of a local function indicating the precise origin or recipient of the event such as the method name within the MVC controller corresponding to the "controller_uri" field that is required for client-emitted events.

EVENT PUBLICATION, SUBSCRIPTION, AND BROKERING

CES brokers events between client applications and internal agents. It does not, however, directly push events to internal agents. Instead, CES publishes events to a connected a pub/sub system – Redis in the provided reference implementation – that is deployed and scaled separately from CES.

When CES receives an event from any source, it is injected with a session_id and person_id, when possible, saved to a local database, and indexed, all before the event is published. This is architecturally important to prevent race conditions where subscribed agents receive the event prior to it being committed to the temporal store. After the event is injected with missing fields and saved, it is partitioned to the field level, then broadcast

to every referenced pub/sub *channel*. “Relevent” channels are identified as the set of all field values keyed with a “*_uri” prefix, as well as several channels implicit to event published event. In the above table, for example, the:

- topic_uri will cause broadcast to the "http://www.ke.tu-darmstadt.de/ontologies/ui_detail_level.owl#mouse-single-click" channel
- model_uri will cause broadcast to the “fhir://Patient/42 channel”
- controller_uri will cause broadcast to the "knartwork://relatedResources" channel

In addition to *_uri topics, the person_id and session_id fields are also implicitly considered to be relevant channel names. They are always encoded into URI format using an “artaka://” prefix with an English pluralization of the object name, e.g. “artaka://people/06dae8ff-4682-40ac-b13e-6d71c0508b41” and “artaka://sessions/06dae8ff-4682-40ac-b13e-6d71c0508b41”, respectively. This matrixing of events into separate broadcast events is architecturally significant, as it facilitates agents that respond to:

- user activity *across* multiple applications *concurrently*.
- agent-enforced business rules at the session level, such as pushing of single log out/off (SLO) events from an SSO system.
- model state changes (such as EHR updates) across *all* applications.
- comprehensive activity flow tracking across the entire enterprise.

Concepts identified by channel URIs are encouraged to be selected from a recognized and appropriate ontology, though this is not required. Since CES is unaware

of any such associations, an event to "..#mouse-single-click", for example, will never be implicitly broadcast to any parent-level channels. Publishing (or republishing) events to parental concepts is an activity that must be implemented by an agent with awareness of these relationships. As a direct consequence, interested agents should generally subscribe to a *set* of channels at appropriate levels in the taxonomic hierarchy and perform any filtering internally.

While the reference implementation of CES has been designed as lightweight as possible, CES API compatibility may also be achieved using adapter layers around Kafka, Knowledge Grid[86,87] or other complementary frameworks.

It is also worth reemphasizing that CES itself is not a pub/sub system and has no need to replicate one. Pub/Sub is a common capability to many systems architectures, and for the scope of work done for the ARTAKA reference materials there is little reason to pursue such a path of making enhancements to this specific component of the architecture. This is debatably in contrast to HL7's Event Publish & Subscribe Service Interface[88] (EPS) published in 2015 as a contribution to the Health Services Specification Project[89]: a jointly recognized effort between HL7 and OMG and co-sponsored by HL7 Clinical Decision Support and Service Oriented Architecture work groups.

ARTAKA establishes a strict separation of CES, internal pub/sub, and Temporal Event Store. EPS does not draw specific lines defining where sub-service boundaries lay, if any, and exposes a much broader scope of interfaces with prescriptive approaches to role-based access control. Additionally, many capabilities such as event replay and

federation are explicitly supported. EPS is philosophically intended to be inclusive of most every type of pub/sub scenario an architect is likely to encounter, many of which CES either explicitly descopes or does not explicitly address. As an additional bonus, EPS at large appears safe to expose directly to other services in the SOA, whereas ARTAKA's internal pub/sub system and agent cluster are assumed to operate behind the façade of CES. EPS also defines a CES-like broker is defined at the functional level.

Downsides to EPS as-is are that:

- It is only a Service Functional Model (SFM). It does not actually provide implementation guidance to the level that applications may build applications against.
- The functional model is extraordinarily complicated for a pub/sub system design. The specification itself is 136 pages long, with around ~40 discernable internal domain model types (p.25). Event digests, full-fledged user role, security model, multiple exception handling mechanisms (p.46), topic management, and other features all create complexity not only for implementors, but for application integrators. The specification is clear that it may be decomposed to be implemented in any number of ways, but the whole of the solution nevertheless requires a large commitment and high learning curve. No implementations from anyone other than the submitting vendor appear to have emerged to date.
- Even with strong positions or role-based access, it is effectively impossible to define specific permissions without knowledge of the model resources and operations being permitted. For this reason, ARTAKA descopes access control

enforcement from the pub/sub system completely, trusting CES and individual knowledge agents to make their own determinations on what sessions are authorized to receive granular data elements.

- Due to the high-level nature of the specification, it is unclear how implementations work in the use cases of web-based applications such as SMART-on-FHIR that typically have no known backend service provider(s) until launch time.

It may be argued well that either CES is currently unharmonized with but philosophically aligned with EPS, CES is too divergent from EPS to warrant harmonization, or some middle ground depending on desired functional capabilities. At the very least, there is strong intersection of interests and use cases between CES' lightweight yet opinionated approach and EPS' more heavy-handed yet accommodating approach. What is ultimately best for knowledge-driven healthcare applications is to be determined, though explicitly-defined mechanisms for user experience orchestration beyond high-level functional models are sorely needed. Some of the drawbacks and limitations of CES' more open-ended current approach are discussed in Event Profiling & Developer Collaboration and Topic vs Action URIs.

SCALE

Due to the granular level of events and horizontal slicing of internal broadcast topics, CES generates immense amounts of internal event traffic. The CES deployment and pub/sub cluster must be scalable appropriately in a horizontal manner. Other than the

Temporal Event Store, all channels should be configured to be 100% ephemeral and with no history. (This is different than EPS.) In other words, a subscribed agent should have no way of asking the pub/sub cluster for *past* events: only the Temporal Event Store may be used for that purpose. These design decisions allow the pub/sub cluster to effectively *ignore* all published events that have no active subscriptions. No caching or buffering is necessary. Thus, if CES emits person_id traffic in an implementation where no agent is listening, it will simply be disregarded. "*If a tree falls in a forest and no one is around to hear it, does it make a sound?"*

Under ARTAKA, "No."

DATABASE / TEMPORAL EVENT STORE (TES)

The TES is a temporally-ordered object store authoritative for maintaining session-specific history. For illustrative simplicity within the reference implementation, this capability is included within the CES database's PostgreSQL schema. In large-scale production use, however, the "events" table specifically is extremely hot and should be broken out into a separately-scaled entity, such as a sharded MongoDB cluster with built-in request routing for distributed Map/Reduce needed for complex event processing queries across a temporal window. [90] Indexing of the various *_uri and *_id fields is highly recommended, though should be accompanied with aggressive ETL jobs into a separate data warehouse structure to keep the TES lean. Figure 4 illustrates how the CES reference implementation system objects relate to the "events" table.

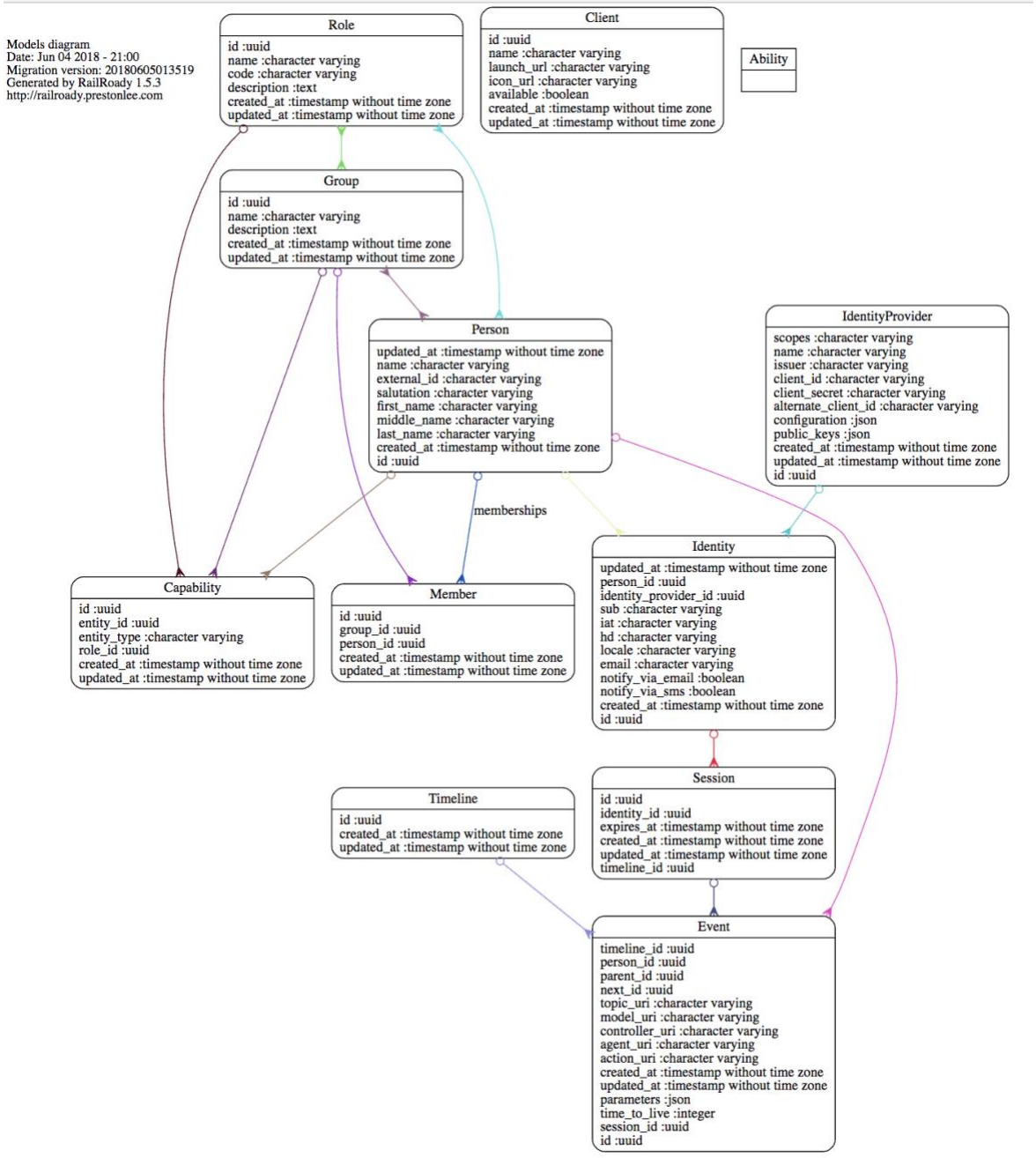

FIGURE 4 CES REFERENCE IMPLEMENTATION SCHEMA

Alternate implementations may leverage other solutions designed specifically for this purpose, such as WSO2 CEP with SiddhiQL[91-94] for optimized trigger-based

window queries, or semantic query by replacing the simplistic TES model with a semantic database pre-optimized with caches for demanding agent queries.

Client Subscriptions

Client applications – web-based or otherwise – are all notified of events in the same way: HTTP/2 with HTML SSE. This is a simple operation underpinned with broad support for the standard across the most used web browsers, with the current exception of Microsoft browsers. CES supports subscriptions at the "/stream" endpoint after acquiring a session identifier, generally as part of an authentication process. (Guidance is provided in the following section.) A client subscription will always implicitly include events published to session-specific channel of the form "artaka://sessions/<uuid>", and all other subscriptions must be included in a URL-encoded, comma-separated list of a "channels" parameters, e.g. "h2://ces.local/stream?channels=list,of,channels".

Authentication and Authorization

With the fabric of HTTP/2 and HTML SSE in place, most security considerations that apply to traditional web services also apply to CES. In the following discussion, note that the reference software provides a schema and session mechanism compatible with all popular SSO systems, but not a full implementation as the authentication and authorization portion of the architecture is intended to be married to local infrastructure.

All with FHIR, client authentication is necessarily loosely defined as to allow for maximum compatibility with existing health IT environments. There is no singular and universally accepted IAM protocol. The reference implementation requires a *bearer*

token for most calls, which may be acquired via an HTTP GET to the "/sessions" endpoint. This GET call, in real-world deployment, must be replaced with either a standards-based authentication flow such as chained SMART-on-FHIR launches against a common IDP, or similar mechanism. The CES and Marketplace schema have been intentionally designed with the common characteristic of supporting *multiple* IDPs concurrently. This is necessary for large enterprises and collaborative care scenarios that authenticate users across trusted boundaries.

The bearer token mechanism is completely in line with OAuth-based APIs, and is required for both CES' "/stream" SSE endpoint and traditional REST endpoints. A bearer token obtained from CES is a JSON Web Token (JWT) that represents the active client session, and must be included on every request as an HTTP header in the form of:

Authorization: Bearer <the_jwt>

A JWT may be decoded by the client, but also includes a cryptographic signature salted with a secret only known to the CES deployment. Any tampering of the JWT-based session is thus detected by CES and rejected on any call that included it. Alternative implementations may choose to preventively invalidate the referenced session identifier, assuming that doing so does not introduce a denial of service attack vector. Additional security precautions may also be implemented to prevent session replay attacks, though the reference implementation prescribes no specific approach.

The CES reference implementation includes a "role" and "group" model, both in line with the marketplace, that supports dynamic enforcement of granular privileges.

From the perspective of a legitimately-authenticated user, the attack surface likely to exhibit implementor bugs is the "/event" publication endpoint and corresponding SSE "/stream".

Most client events should have no expectation of a response, such as those emitted based on field selection or model changes. All events broadcast to their session (referred by their JWT) will automatically be routed to them. Some clients, however, may desire subscriptions to a large numbers of non-session streams, or streams intended to be used internally between agents. This is allowable, and CES facilitates this by allowing the list of client channel subscriptions to include a "*" wildcard/glob character. This is problematic for security reasons, however, and clients must not be able to arbitrarily subscribe to any channel they chose. Further discussion on this issue is provided in the Experimentation, Evaluation chapter.

The CES reference implementation provides very little channel filtering as doing so is highly dependent on the integrated authorization mechanism in relation to system roles. Real-world deployments should use a pattern-based whitelist of approved channels tied to the roles the user – or more accurately the identity they are using at the moment – is allowed to access.

Similarly, clients must be prevented for broadcasting nonsensical or malicious events into the system. As ARTAKA is asynchronous, non-deterministic, and has no innate notion of session "stickiness" to specific agents, controls must be enforced to assure users are not able to affect the state of other sessions, such as by attempting to fabricate or falsify events by mimicking a trusted agent. Allowing this would permit a

new notion of client-side man-in-the-middle and timing attacks. The reference restricts these interactions with a design consideration that clients are not allowed to set an agent_id nor session_id with an additional permission normally reserved for trusted agents. Further, normal clients are never allowed to subscribe to any other non-session channels than those whitelisted in the configuration.

API Overview

An initial intent in early CES design was to declare the entirety of the interfaces using an open specification language such as OpenAPI (OAS) [95], formally known as Swagger, or RAML, "..an application of the YAML 1.2 specification" [96]. Numerous proprietary and open source tools were used to attempt this, as both OAS and RAML are generally-accepted languages for modern web-based services.

In all cases attempted, no mainstream language nor tool provided ample support to encompass the CES API due to a heavy focus on REST and lack of support for Server Sent Events (SSE). Neither tooling nor API specification languages have caught up to the type of SSE interface exposed by CES. In lieu of a comprehensive API document that would typically be expected, I have provided an overview shown in Table 1 as well as additional documents and diagrams in the reference implementation linked from Table 3. A computable representation of this information should be of high priority once support is available in a mainstream language such as OpenAPI.

Verb	URI Pattern	Controller#Action
OPTIONS	/*all(.:format)	application#cors_preflight_check
GET	/timelines(.:format)	timelines#index

POST	/timelines(.:format)	timelines#create
GET	/timelines/:id(.:format)	timelines#show
PATCH	/timelines/:id(.:format)	timelines#update
PUT	/timelines/:id(.:format)	timelines#update
DELETE	/timelines/:id(.:format)	timelines#destroy
GET	/events(.:format)	events#index
POST	/events(.:format)	events#create
GET	/events/:id(.:format)	events#show
PATCH	/events/:id(.:format)	events#update
PUT	/events/:id(.:format)	events#update
DELETE	/events/:id(.:format)	events#destroy
GET	/people/:person_id/identities(.:format)	identities#index
POST	/people/:person_id/identities(.:format)	identities#create
GET	/people/:person_id/identities/:id(.:format)	identities#show
PATCH	/people/:person_id/identities/:id(.:format)	identities#update
PUT	/people/:person_id/identities/:id(.:format)	identities#update
DELETE	/people/:person_id/identities/:id(.:format)	identities#destroy
GET	/people(.:format)	people#index
POST	/people(.:format)	people#create
GET	/people/:id(.:format)	people#show
PATCH	/people/:id(.:format)	people#update
PUT	/people/:id(.:format)	people#update
DELETE	/people/:id(.:format)	people#destroy
GET	/groups/:group_id/members(.:format)	members#index
POST	/groups/:group_id/members(.:format)	members#create

GET	/groups/:group_id/members/:id(.:format)	members#show
PATCH	/groups/:group_id/members/:id(.:format)	members#update
PUT	/groups/:group_id/members/:id(.:format)	members#update
DELETE	/groups/:group_id/members/:id(.:format)	members#destroy
GET	/groups(.:format)	groups#index
POST	/groups(.:format)	groups#create
GET	/groups/:id(.:format)	groups#show
PATCH	/groups/:id(.:format)	groups#update
PUT	/groups/:id(.:format)	groups#update
DELETE	/groups/:id(.:format)	groups#destroy
GET	/roles/:role_id/capabilities(.:format)	capabilities#index
POST	/roles/:role_id/capabilities(.:format)	capabilities#create
GET	/roles/:role_id/capabilities/:id(.:format)	capabilities#show
PATCH	/roles/:role_id/capabilities/:id(.:format)	capabilities#update
PUT	/roles/:role_id/capabilities/:id(.:format)	capabilities#update
DELETE	/roles/:role_id/capabilities/:id(.:format)	capabilities#destroy
GET	/roles(.:format)	roles#index
POST	/roles(.:format)	roles#create
GET	/roles/:id(.:format)	roles#show
PATCH	/roles/:id(.:format)	roles#update
PUT	/roles/:id(.:format)	roles#update
DELETE	/roles/:id(.:format)	roles#destroy
GET	/clients/:id/launch(.:format)	clients#launch
GET	/clients(.:format)	clients#index
POST	/clients(.:format)	clients#create
GET	/clients/:id(.:format)	clients#show

PATCH	/clients/:id(.:format)	clients#update
PUT	/clients/:id(.:format)	clients#update
DELETE	/clients/:id(.:format)	clients#destroy
GET	/identity_providers/:id/launch(.:format)	identity_providers#launch
GET	/identity_providers(.:format)	identity_providers#index
POST	/identity_providers(.:format)	identity_providers#create
GET	/identity_providers/:id(.:format)	identity_providers#show
PATCH	/identity_providers/:id(.:format)	identity_providers#update
PUT	/identity_providers/:id(.:format)	identity_providers#update
DELETE	/identity_providers/:id(.:format)	identity_providers#destroy
GET	/sessions(.:format)	sessions#callback
POST	/sessions(.:format)	sessions#create
DELETE	/sessions(.:format)	sessions#destroy
GET	/status(.:format)	welcome#status
GET	/stream(.:format)	events#stream
GET	/	welcome#landing

TABLE 1 ENDPOINT OVERVIEW WITH REFERENCE IMPLEMENTATION LOCATION

ORCHESTRATION, SIMULATION, AND TIME TRAVEL

The most peculiar and abstract aspect of ARTAKA is in use of an optional *timeline_id* field common to all types of events brokered by the CES. While the base nature of CES and agents have already been discussed, it is necessary to have an additional understanding of event timeline semantics to design compatible systems. Without awareness of how these events manifest, they may appear to have materialized out of error. ARTAKA considers passage of time itself to be an event and provides an optional *timeline_id* field for all events as a means of partitioning intentionally distinct logical event universes.

ONTOLOGICAL DISAMBIGUATION

One of the primary design goals of ARTAKA is to provide a sensible means of knowledge-driven client UI orchestration based on pluggable knowledge: authorable, testable, deployable, and maintainable by *clinical* domain experts without deep technical knowledge of the underlying system. A key challenge of doing so is being able to disambiguate *observations* of events that:

- should have occurred, but did not: “Session S should have prefetched model M.”
- have occurred or are occuring presently: “Session S fetched model M.”
- will occur: “Session S will fetch model M.”
- may occur: “Session S will probably select patient P.

Disambiguating these different uses of the concept of “fetching” would be unwise using only singular atomic terminologic codes, as doing so would require immense

amounts of duplication in model ontologies. CDS Hooks, for example, only defines a "patient-view" hook as "The patient whose record was opened..", a *past* event. Stating a record is *in the process of* opening or *will open* in the future would require a separate, and arguably duplicative, hook type definition.

The above examples can also be categorized as statements of *past*, *present*, and *future* state, respectively. Most event-driven systems, in practice, only make statements of the present, generally via a singular timestamp of high precision for simplify. Backend agents performing predictive or simulation functions, however, may view ARTAKA as a form of complex event processing or event streaming system wherein simulation of future state may be accomplished by applying all normal agent functions at a logical speed faster than real-world wall clock time in a divergent timeline.

TIMELINE CONNECTIVITY

An ARTAKA "timeline" is a chronology of events that occur within a logical universal of discourse between agents, clients, and data. It is a temporal space that partitions events that would otherwise need to uniformally fall into a single, one-dimensional, linear order.

To be able to simplify orchestration and allow for complex autonomous event simulation simulateously, ARTAKA agents must be able to emit and receive observations of the past and future regardless of whether they actually have or will occur. The disambiguation occurs via a *timeline* associated with every event. This notion of event timelines alleviates the need for ontological activity models for past and future, as the values of timestamp fields in the structure implies their meaning. The created_at and

updated_at fields are always accurate to the "real-world" clock, even if the event is modeled to have occurred in the past/future. A non-null effective_at field, when present, is used to semantically convey a timestamp that is different from created_at. Clients and agents should not set effective_at if the event occurred has occurred at the same time as created_at. Further, all ARTAKA events should always be normalized to a tenseless form to prevent aforementioned terminologic explosion.

ARTAKA agents do not directly perform any form of traditional command-based orchestration of client UI. All orchestration activities are represented as a form of simulation: alternate past and possible future events that may or may not coincide or converge with real-world events experienced by the user. As events are always stated in tenseless form and on varying timelines, they may occur in universes of discourse divergent from or convergent to the user's perception of reality. If/When clients observe these orchestration events, they may or may not choose to "merge" the event from an alternate timeline onto its own default/real-world timeline, if agreeable from the perspective of the user, by first performing the corresponding action and subsequently publishing an updated *copy* of the event to the default timeline, referencing the causal trigger via a genealogical "parent_id" identifier. The same notion applies to *future* or scheduled events.

Orchestration hints, as the primary client example, are represented as events that have occurred in an alternate past of the same session, whose timeline may or may not have converged -- or will converge in the future -- to the "default" timeline associated to the user session. This convergence is represented via an event's *next_id* pointing to an

event in the default timeline. The "next" event, when pointing back to the default timeline, is likely to have already occurred in the real world, and may have been the trigger that created the alternate past.

Similarly, predictive events are also represented in present tense at a *future time* on an *alternate timeline* that must have diverged, at some point, from any "default" timeline via a *parent_id* pointing to a real-world event that triggered the branching. The soft requirement for all timelines to diverge on converge from a common source ensures all timelines are kept connected, but does not prevent actors from creating incoherent situations where, for example, an event is its own ancestor. Contiguousness of the TES graph may also be broken if/when events fall outside the event horizon and are subject to pruning.

NULL TIMELINE

Timelines themselves do not have any awareness of or pointers to other timelines: only events do so indirectly through pointers to events on other timelines. When a timeline is created, it carries no data other than the "real" timestamp of creation. It will be automatically purged by the system after a preset period if and only if they are found to contain an empty set of events.

The null timeline is the only timeline with special semantics, as its existence is not actually retained in any tangible way. It does not have any concrete start and/or end. As such, it cannot be purged or created, since there is nothing to purge or create. It is reserved for use only in cases where scoping of an event to a single timeline does not make sense or is undesirable. System-wide global clock ticks are a primary example, as

the notion of "current wall clock time" is useful on all timelines, even if operating at a logical time different from the real world. Events that occur on the null timeline may only be said to be inconsistent with the notion of a single timeline, and no other conclusions may be inferred from the lack of a timeline on an event.

CLIENT SEMANTICS

When a CES session is created, CES automatically creates a new timeline or use by the client, referenced by a 128-bit UUID field in the bearer token. From a client-side perspective, this is known as the "default" timeline for the session. When a client receives an event from an agent on the same timeline, it is known to have occurred "in the real world", at the exact date(s) indicated by the created_at timestamp. This occurs when, for example, the user has multiple application windows open that use the same session but have separate connections to the server. Clients may only publish events on their current default timeline. When a client attempts to publish an event to the null timeline, it will be implicitly set to the default. Clients attempts to publish to a non-default timeline should be rejected.

Since event topics carry no tense, Table 2 demonstrates colloquial types of orchestrations, and the ARTAKA representation. Td is the default session timeline, and T<x> is an alternate timeline identifier.

Orchestration	Representation
Prefetch patient model X for session Y now.	Patient model X fetched at <effective_at> on timeline T42 preceding <triggering_event_id>.
Reauthenticate the user in 5 minutes.	User authenticated at <effective_at> on T42.
Mark the document as complete and then save it.	Two events on a single alternate timeline: Document state set to "complete" at <past effective_at> on T42 followed by save at <past effective_at+1> on T42 followed by <triggering_event_id> on Td.

TABLE 2 REPRESENTATIONS OF ORCHESTRATION HINTS

In the last example, the event trigger caused a chain of actions to be created in an alternate past timeline. The agent(s) continued contributing events to that timeline until it converged with the triggering event on the default timeline. If both "mark complete" and "save" events were published by the same agent, the "mark complete" event will likely have a next_id indicating there *will* be a subsequent event coming that should be processed in the same client-side sequence of operations, if they are processed at all. The "save" will subsequently have a pointer to the original triggering event, likely created by the client itself.

If/When CEC (the client side) observes these orchestration events, it may or may not choose to "merge" them onto the default timeline by actually performing the function and then publishing a copy of event to the default timeline referencing the causal trigger via a "parent_id" in the new event.

AGENT SEMANTICS

When an agent receives an event, it has no implicit knowledge of the meaning of the referenced timeline. And while events are usually scoped to a session, there is not presumed concept of a "default" timeline from an agent's perspective. All timelines are treated equal.

While an ARTAKA "agent" is a very generalized concept, at their core they are event-triggered services that may perform reasoning, complex event processing, do database or repository I/O operations, be scheduled, invoke other agents etc. They are usually stateless, and always multi-tenant/user/context. This is familiar and comfortable.

Where agents become more abstract is that they are inherently multi-*timeline.* Agents are "grounded" to wall-clock time via global clock tick events, but are generally unaware of whether they are operating on "real world" (aka client "default") timeline events or those from alternate universes unless an effective_at timestamp is provided, though the TES may be queried if necessary. Agents that accept and/or emit events from alternate pasts and futures may be triggered from any number of timelines simultaneously, and even cross timelines when events converge (for past events) or diverge (in future events). Knowing which timeline to use, and when, follows a few rules of thumb.

- Agents that emit responses to specific types of events within the context of a user session should only use the triggering timeline for statements of events that actually happened. "Notification sent" and "User logged out" are event cases

where the act occurs on the same timeline as the trigger. The timeline_id of the triggering event should be used in events published by the agent.

- Orchestrations should occur on alternative timelines, as the acts did not occur from the perspective of the client. They would be fabricated statements on their default timeline, and *if* the client decides to merge them into their default timeline, the client will re-publish updated copies that may subsequently be received by other agents. Agents may create and manage timelines via CES REST API calls.
- Cross-agent chatter and system events exist outside the scope of a single timeline, and should thus use the null timeline.

For event simulation requiring modification of model state, it is highly recommended to consider global data resources writable only from the null timeline, and represent timeline-specific differences via model overlays using copy-on-write semantics. Each patient corpus, then, is always specific to a known timeline. This prevents the corruption of data resources in the null timeline by constraining changes to a separate universe, ideally in the PCC such that different agents may all query the same semantic view of the alternate timeline.

To prevent infinite event looping scenarios across agents, generally considered to be stateless by design, agents involved in the creation of event timelines should always proactively check for livelocks of their own creation by querying the TES prior to publishing any new events. If such conditions are a known risk, the agent should self-impose an internal tracking and/or throttling mechanism to prevent accidentally flooding the system. Due to the high-volume, low-latency nature of agents, the risk of accidental

denial of service via livelock increases exponentially with the number of deployed agents. Agents that prune system objects, such as the "pruner" agent provided in the CES reference implementation, are further recommended to do so aggressively and at frequent intervals triggered from clock tick events.

EXAMPLES REVISITED

In Figure 3 of the Context Event Service chapter, several situations and a generalized sequence are discussed in which UI orchestrations may be desired. With a deeper understanding of the way CES facilitates this process, it may be helpful to expand this sequence in the context of ARTAKA's event orchestration semantics.

In Figure 3, a client application has presumably already been instrumented with support for CES. This instrumentation into the application is further assumed to capture UI actions at a global or similarly high level in the controller hierarchy such that notable interactions with visual controls/widgets results in discrete events being published to CES via the REST JSON API. In this example, the user has "logged in" to the application and is, or is about to be, sitting in a "home" screen and state within the application. At this stage, the user likely has just "launched" the application (in the case of SMART-on-FHIR) or started the application from their desktop computer or mobile device. Let us expand this scenario but orchestrating several useful state changes to the client using both frontend and backend event capabilities:

- Restoration of user preferences.
- Synchronizing with concurrent sessions.
- Preloading of patient data for detailed query.

Most applications have some form of persistence mechanism allowing for user preferences to be saved and applied at a later point in time. Color schemes, visual layout, display language etc are all common user settings. When implemented, this is usually engineered in an application-specific manner. This can also be implemented with an ARTAKA agent operating with central authority for managing these data across *all* applications. In this case, the fictitious user preference management agent would detect the login event, retrieve any existing preference information for the specific user and application, and emit a preference model_uri back to the client session with any application-specific parameters in the payload.

Similarly, specifications such as HL7 CCOW permit certain session context fields to be synchronized across *multiple* different applications. This can be enabled in ARTAKA-integrated applications by use of an agent detecting the same login event. Instead of integrating with an underlying database, however, an external CCOW service is used. If a current patient selection is found via CCOW, for example, this selection may be relayed to the client with a "…#select" topic_uri, FHIR resource reference as a model_uri, and directed to a "PatientController" via application-specific controller_uri and/or action_uri, if known. This orchestration event would contain a timeline_id, set by the CCOW integration agent, different from the "default" timeline used by the user session, with an effective_at timestamp in the past, and pointing to the triggering login event via next_id. Setting of these fields in combination allows the client's named (or anonymous) controller responsible for managing patient selection to automatically select

the patient within the client, in turn publishing a "…#select" event back to CES on the "real" session timeline.

Regardless of who or what selected the patient of interest, CES' broadcast of a patient selection is indicative that *some* session is likely to have more detailed inquiry into the given patient in the near future. This specific case is intended to be handled by one or more Patient Corpora Cluster management agents. Once detected, the PCC manager is now able to materialize a patient-specific semantic database for the patient, and internally broadcast its availability, location, and capabilities.

With sufficiently low-latency and high-performance engineering, these stories may ideally be executed so quickly that the user has no awareness of such choreography. The application simply launched quickly, into a reasonably expected state based on their activities in other applications, and is extremely responsive to continued use.

EXPERIMENTATION, EVALUATION, & FUTURE DIRECTION

Enterprise architecture deliverables are often accused of being overly academic, with little concern for the plausibility of designs. To demonstrate real-world value, I have provided an environment supplementary to this work both as evidence of feasibility, and guidance to future implementors. This environment is hosted under the artaka.org domain at time of this writing, and includes the reference implementation of CES, a version of KNARTwork instrumented with the CEC client library, and handful of exemplar agents, containerized HSP environment, TES, reverse proxy balancer, pub/sub system, object cache, and all associated databases.

Fortuitously as this manuscript was in early draft state, several developers became available through grant money initially provided by the Piper Foundation to promulgate works in biomedical informatics. These hours were leveraged wherever possible to gain hands-on feedback from the perspective of external developers new to event-driven architecture, tasked with implementing CEC and reference CES/agents into the existing KNARTwork application.

The successes and failures of doing so would have been impossible to predict, and many of the challenges discovered along the way have already resulted in positive changes at the component level. Perhaps most importantly, these lessons learned set clearer direction for future work outlined in this chapter, organized by the area of concern being discussed. Moving forward, two broad categories have emerged in which most shortcomings fall: clarity around information security practices in the integrated

enterprise, and the idiosyncrasies concerning the nature of events and timelines. Both these headings are deep rabbit holes of investigation and will benefit greatly from further review of existing literature and current directions of CDS as a field.

Various reference implementation projects and direct links are provided in Table 3.

Project	**Link**	**Remarks**
Context Event Service/Temporal Event Store	https://github.com/preston/context-event-service	Also includes clock agents, event pruner, and example agents.
Context Event Client	https://github.com/preston/context-event-client	TypeScript/JavaScript
KNARTwork	https://github.com/cqframework/knartwork/	Contributed to CQFramework in 2017.
HSP Marketplace Server	https://github.com/preston/hsp-marketplace-server	Pursuing HL7 adoption.
HSP Marketplace UI	https://github.com/preston/hsp-marketplace-ui	Web UI. AngularJS.

HSP Marketplace Agent	https://github.com/preston/hsp	Supplemental proof of concept client for integration into local orchestration environment.

TABLE 3 REFERENCE IMPLEMENTATION SOURCE CODE RESOURCES

All reference systems have been packaged according to HSP-compatible service criteria, and a docker-compose file has also been created for ARTAKA- and CES-oriented developers to quickly start a local development sandbox with minimal overhead. This docker-compose file is packaged with CES, and utilizes a .env file that must be configured for the local databases available to the local system.

HOT SPOTS

CES provides the publication endpoint for both clients and agents. Rather unexpectedly, the components subject to the most experimentation turned out to be CES and corresponding adjustments to the polymorphic event model. In retrospect, this should have been expected as CES and the event representation are the glue that binds all ARTAKA actors together.

While agents may technically write directly to the pub/sub system, it is discouraged since direct-broadcast events are not recorded in TES nor propagate to clients connected to CES. Excluding runtime characteristics of any specific agent, the "hottest" aspects of ARTAKA to date are the pub/sub cluster and TES, and the manner in which CES' must interact with it.

CES and pub/sub Redis cluster were jointly stress tested with numerous simultaneous mock clients and real agents running to search for bugs as well as issues fundamental to the architecture itself. The most significant discoveries relate to determinism.

POSTing to CES' /events REST endpoint is trivial to implement but extremely sensitive to changes. In initial implementation, an event POST would perform two operations simultaneously:

- Save the event to TES and return the result immediately to the client, to maximize performance by removing any and all I/O delays.
- Broadcast the event to agents via pub/sub.

With this implementation, agents directly querying TES (such as the event pruner) would exhibit sporadic failures while trying to retrieve the windowed historical context of an event of interest. The reason for these odd and difficult to reproduce issues was that the asynchronous nature of CES did not guarantee that an event was fully committed to TES prior to broadcasting it via the pub/sub system. In rare occasions an agent would receive an event, attempt to retrieve it from TES, and fail to do so. After changing CES to fully commit the event to the TES *prior* to pub/sub broadcast, the issue appeared to be resolved.

Live Locks

During course of prototype agent development, several cases emerged in which a stateless agent unwittingly flooded the system with an infinite loop of event interactions with either:

- Itself
- another agent
- a client that "merged" an agent event into its own timeline and essentially rebroadcast the merged event in such a way that triggered the same agent

While event-driven architecture is extremely powerful, the observation of these types of occurrences highlights the amount of vigilance needed to prevent, let alone detect, infinite loops. Agent-side filtering of one's own events – that is, events with a agent_uri that matches the URI of the receiving agent – provides a quick win, but does nothing to remove possibilities of circular loops between agents. In one case, the live locking was so severe that a TES table lock created by the event pruner agent caused CES to suffer buffer overflows from inability to keep up with event publication demands.

Production systems will need special monitoring agents specifically designed to detect, and potentially stop, live lock events determined to threaten the availability of the system. ARTAKA does not prevent these safety features from being added, but also does not provide a built-in solution. A consideration for subsequent revisions of the architecture is a requirement for agents to be able to process a *halt* event topic. Receipt of a halt would result in a temporary suspension of activities when notified of events matching a provided pattern. This may or may not work in practice, as any sort of

system-wide deadman switch is typically as easy to abuse as to use, and the difference difficult to detect. Anyone can pull a fire alarm.

NETWORK RELIABILITY

From the onset, the ARTAKA concept has been subject to the runtime conditions that client event pub/sub is never guaranteed, nor is event order. This was generally not a problem due to the non-critical nature of the exemplar application, but requires developer awareness to handle properly.

The primary negative effect of non-reliability is that client code must be authored to provide "best effort" processing of events received from CES. In the case of sequences – e.g. a "load model M" followed by a "select field F" – out-of-order or partial receipt of the events will result in a client state not fully aligned with the expectations of the emitting agent. For agents, this also implies that 100% accurate state mirroring is impossible, as agents have no built-in means of verifying events were received, let alone processed.

The majority of reliability-related issues during CES experimentation came from infrastructure-level causes. In the initial sandbox environment, I discovered a network device between the Internet and CES application server was preemptively dropping inactive TCP connections between clients and the host server. This showed up in the CES logs as user disconnects, even though the client test applications believed they were still connected. Brower-side automatic reconnection attempts were problematic and difficult to reproduce.

For IT environments where the network is not fully controllable, the use of stateful SSE may reveal inadequacies of the underlying infrastructure. In the local case, the issue was not resolvable until the application server was moved at a cloud-based environment on Amazon Web Services where a software defined network (SDN) could be established with the correct configuration.

Agent Access Controls

ARTAKA agents are assumed to operate in a trusted environment. The access level provided agents to access underlying data sources is left undefined, but it is clear that an architectural stance must be made early in the development of an implementation.

The position simplest to implement is to assume that agents, as trusted operators operating in a secured environment, are allowed largely unfettered access to data layer resources, most notably the EHR. The problem, however, is that without a security context of the session, agents may inadvertently run queries or perform operations that the end user is not allowed to read or execute, respectively. Outcomes can result in accidental disclosure or unintentional privilege escalation by using agents as a middleman to run operation at privilege level the user does not possess.

The most secure approach is to limit agent capabilities to the scope of the user session, when applicable. Using the existing 'scope' values of a SMART-on-FHIR-authenticated session, for example, allows abiding agents to implement self-imposed access checks prevent privilege escalation. Of course, asking every agent developer to consistently and correctly supply support in agent implementations would likely be a

mixed bag. If this approach is used, security context field would/will need to be injected by CES prior to publication of the event to the relevant channels.

Publication and Subscription Filtering

Clients should be able to subscribe to a wide range of channels, but never to any channel that could potentially leak sensitive data. Implementing this in CES is tricky, and the CES reference implementation does not provide a fully flushed out security model for limiting the scope of subscriptions that can be created by clients.

For production use, the straightforward means of doing so is to add a "permissions" object field to the Role type: an approach I have used successfully in other RBAC designs. This field would store a whitelist of allowable channel *patterns* for users assigned this role. As with all secure web applications, these privileges would be evaluated upon every API call.

An additional benefit of using such a permissions model is that it would also serve as a mechanism for agent publication authorization management. As agents publish to CES in the same way as external CES clients, and a unified publication permissions model would provide a layer of internal security as well for all CES API operations.

Event Profiling & Developer Collaboration

While engaging in both client and agent development, a common recurring question was, "What URI should I use for X when Y happens?" Since event payloads contain numerous URI fields and there are very few restrictions on their use, developers

naturally need some explicit guidance on reasonable values to use. This question applies equally to both client and agent developers.

Even when only developing within the context of a single organization, documentation is necessary to catalog local ontologies and the specific allowable concepts referenced during event publication. Without such a catalog, which can be as simple as a wiki page, developers are forced to guess or make point-to-point decisions based on *a priori* knowledge of specific agent behavior. From experience working as a team, these guesses are usually wrong, even when ontologies and models used for the URI spaces are identified. Subtle semantic interpretations of events varies greatly by application and subjective interpretation of the developer.

CES events, as a polymorphic model with optional fields, unfortunately will require a level of "profiling" not currently accounted for. This mechanism should avoid prescriptive constraints, but support extremely clear usage semantics that may not be evidence from strict consideration of only the referenced ontologies and/or models. This should be similar in spirit to the mechanisms of the same name used in specification of the FHIR API, though in a more constrained manner to only allow for specializations that do not break the "is-a" semantics of the core CES event type. Allowing for expansion of the structure or semantics of a CES "event" would require coordination amongst all agent and application developers that is unlikely to succeed at scale. Thus, changes to CES' event resource should occur as governed revisions of CES itself. This future addition of event profiles implies additional changes to other aspects of ARTAKA, such as the Health Services Platform Marketplace specification.

The HSP Marketplace specification includes only a limited and loosely defined mechanism for declaring service operability models. It is a known limitation that, at time of this writing, awaits detailed feedback from the HL7 community to resolve sufficiently for supporting FHIR services in such a way that semantic resource interoperability can be determined with 100% automated declarative compatibility checking. This will require a change to the Marketplace's current platform-independent model (PIM), platform-specific model (PSM) of the reference implementation, and API. Regardless of the exact nature of the changes, it will require consideration to support FHIR, but not be constrained to it. Doing so will allow ARTAKA knowledge agents to declare the event and model types supported and/or required in a way comfortable to the SDO community.

The other major area of necessary developer collaboration is the event payload field. As this field may be any arbitrary lightweight JSON object, clients and agents have no strict guidance on how to use it for specific use cases. In retrospect, this field effectively functions as a developer-defined microschema. Both agent and client developers must clearly document any/all uses of this field such that the structure and semantics are clear to all parties. Examples are the most practical way of documenting usage, though formalization of the microschemas using JSON Schema[97] or similar is preferred. A formal schema governing payload contents better allows for automated validating and mocking.

Topic vs Action URIs

During the course of instrumenting the KNARTwork application with the CEC library, it became apparent that the difference between the event topic_uri and action_uri

fields is ambiguous. The topic_uri, being required for all events, was used in all cases, but the value of the action_uri field is unclear. In highly orchestration user experiences it is expected that action_uri will have value, but in simple cases such as information retrieval appears to be slightly duplicative. A profiling or profile-like approach such as discussed in Event Profiling & Developer Collaboration is needed to disambiguate exact uses.

Timeline Explosion

The usage of event timelines for orchestrations, shown as the optional "create alternate timeline" call in Figure 3, appears to work well. In highly orchestration applications, note that this call to the CES API is made very frequently, as often as once per orchestration event. There are three potential issues caused by this design decision that may warrant adjustments.

First, the number of timelines managed by CES can theoretically grow at the same rate and scale as the number of events. Timeline resources are managed by the TES, however, the growth of these objects was not intended to grow as fast as it does when using agents driving high levels of UI orchestration.

Second, an API call to CES involves a network I/O call to a different system. Doing so introduces precious latency to agent processing times while waiting for CES to create and return the new timeline. Additionally, a creation of a timeline itself generates a event. This can result in live locks, race conditions, and unnecessary flooding of the system with timeline management events.

A potential change worth consideration is having CES create a *single* alternate timeline in addition to the default timeline when the client session is created, and injecting the alternate timeline UUID into events shunted into the pub/sub system. This would allow the minimalistic timeline model to remain unchanged while allowing the "create alternate timeline" call to be eliminated from the sequence diagram to simply attain a valid timeline UUID for events occurring in an alternate past.

Third, ARTAKA has a very lax notion of a simulation event horizon[98,99]: the points before and after a discrete place in time beyond which all events are censored or deleted. This is problematic for agents querying TES since it, as currently defined, is neither a closed world nor open world. Semantic queries extending beyond the event horizon have no guarantees that what is known to TES constitutes all that has been known to TES, presenting an immediate epistemic dilemma. In cases where censoring or purging has occurred, it is difficult to discern the nature or magnitude of what was removed.

In future revisions, TES needs a clear definition of an event horizon that must be observed by agents. Most likely, a system-wide cutoff would need to be decoupled from the physical act of pruning. This approach allows for agents to establish clear boundaries around queries regardless of the state of garbage collection.

TIMELINE AGNOSTICISM

The decision that agents, by default, would remain indifferent to real or simulated timelines is intended to be a simplifying assumption to ease the learning curve for agent developers. Real-world usage suggests, however, that any agent performing write

operations to the PCC, EHR or other external system *must* have some minimal awareness of the timeline context to prevent simulated timelines from corrupting real-world data stores. That is, to apply the copy-on-write semantics discussed in the section on the Patient Corpora Cluster, agents must be able to know when and how it is ok to create, update, and delete data.

One potential approach for patient-centric agents is for CES to inject a reference to the appropriate PCC authority(ies) for reading and writing. This is messy, however, and also breaks a separation of concerns barrier by forcing to CES to be aware of the PCC. CES is not intended to be aware of *any* internal systems except for the pub/sub cluster, TES, and it's own management database.

A more general solution may be to build upon the security permissions injection approach discussed in Agent Access Controls. This would allow CES to guide the consumption constraints of subscribed agents without necessarily introducing awareness of a PCC.

CONCLUSIONS

ARTAKA is a hybrid architectural paradigm combining flagship aspects of traditional MVC application architecture, UI orchestration, knowledge representation and reasoning, complex event and stream processing, and simulation, in a coherent package harmonized with current health IT standards. The architectural blueprints provided, with particular attention paid to elements of lessor prior academic focus, defines a logical path for clinically-authored knowledge artifacts to drive the user experience of production applications using current or future knowledge representation standards.

Hands-on experimentation and evaluation of ARATKA's primary architectural components has provided encouraging results suggesting that the architecture is both achievable in real-world use and highly valuable in advanced cases. At a high level, the potential effects of adoption could significantly impact numerous areas of knowledge management and application chains.

FOR DECLARATIVE KNOWLEDGE

Providers, payers, academics, and industry stakeholders have long seen the potential of declarative knowledge. In healthcare, Arden syntax wet the tongues of clinical informaticists, but due to the "curly braces problem" that limits executional portability, and non-technical issues, has failed to make knowledge executable.

More recently, the HL7 CDS Knowledge Artifact specification improved on a number of shortcomings by standardizing on a polymorphic XML-based document schema for representing 3 specific types of knowledge: order sets, documentation templates, and ECA rules. Through field review of approximately ~104 individual

artifacts and source material, many remaining shortcomings are evident. Feedback to the HL7 work group from myself and many others is guiding recommend changes to the specification, such as by introducing a "composite" type that effectively serves as a more powerful form of ECA rule to glue knowledge-level artifacts together using event-based triggering. Due to the current gravity around FHIR, the fate of the Knowledge Artifact Specification (KAS) is unclear. Mainstream runtime environments may or may not emerge.

Regardless of the fate of the Knowledge Artifact specification, the primary purposes of using any such standard is to enable L4[100] *computability* of the content by compiling it to executable form in a platform-specific manner. ARTAKA's event-driven and decoupled semantics provide interoperable pipes and glue for these executable artifacts to drive clinical UX through use of precoordinated topics. In the CES reference implementation, the event pub/sub mechanisms show how a rendered KAS documentation template can, for example, emit selection events back to the server side where *any number* of other artifacts may subscribe to the outcome, and in turn trigger small UI orchestrations such as hiding/showing of subsequent questions, or more complex operations such as starting a business process in a completely different department through complex event processing. The implications of any architecture loosely similar to ARTAKA pertaining to the incremental introduction of declarative knowledge cannot be understated.

FOR DEEP LEARNING

Often at odds with proponents of formal clinical knowledge modeling is the impressive results from application of modern machine learning. ARTAKA's comprehensive TES provides ML practitioners with a complete structured set of human and agent activities necessary to model complex behaviors of which SMEs may not even be aware. By integrating the CEC into SMART-on-FHIR and other applications, the TES can be shunted into a data warehouse for analysis of comprehensive behaviors, both in and out of the EHR. The traceability of ARTAKA events across unrelated applications is a capability that cannot be trivial replicated by most application development frameworks used in singularity, and while most enterprise applications have *some* form of traceability, enterprise-wide audits of user activity are rare. The reference implementation does not provide built-in tools for ML-based agents as no real-world set of training data is yet available. Additionally, it should generally be assumed that ML-based agent development is driven by custodians of the relevant data.

It is difficult to offer predictions on the role of deep learning in event-driven CDS. The future is bright, however, and ARTAKA is set to support them.

FOR HEALTH IT

For software engineers working in a clinical context, the regulated and highly controlled nature of the domain can be overwhelming for innovators attempting to introduce disruptive change. The sometimes decades-long delays between discovery and live application of knowledge can be maddening and feel overcautious when evidence suggests delays to changes will result in a net negative to outcome quality. A core element to such frustrations is the necessary alignment of priorities and timelines across

clinical and technical stakeholder groups to bring a unit of formalized knowledge into practice.

Traditionally, manual programmatic translation has been performed to mirror SME-driven knowledge in executable form, and usually localized to the extent that reuse of the executable artifacts are non-interoperable. Such has been the case with HL7 Arden syntax. With the introduction of compilable languages such as CQL, runtime architectures such as ARTAKA, and configuration management principles from the software engineering domain, the stage is set to break the mandatory alignments of clinical and technical managers. If clinical SMEs are able to effectively cut out manual IT efforts from the knowledge management cycle, clinicians will be able to significantly shorten the delays to fully develop, exchange, customize, test, deploy, and maintain executable knowledge artifacts.

REFERENCES

1 Kawamoto K, Houlihan CA, Balas EA, *et al.* Improving clinical practice using clinical decision support systems: a systematic review of trials to identify features critical to success. *BMJ* 2005;**330**:765–0. doi:10.1136/bmj.38398.500764.8F

2 Greenes RA, Bates DW, Kawamoto K, *et al.* Clinical Decision Support Models and Frameworks: Seeking to Address Research Issues Underlying Implementation Successes and Failures. *Journal of Biomedical Informatics* 2017;:1–27. doi:10.1016/j.jbi.2017.12.005

3 Kawamoto K, Jacobs J, Welch BM, *et al.* Clinical information system services and capabilities desired for scalable, standards-based, service-oriented decision support: consensus assessment of the Health Level 7 clinical decision support Work Group. *AMIA Annu Symp Proc* 2012;**2012**:446–55.

4 Greenes RA. *Clinical Decision Support*. Academic Press 2014.

5 Imam FT. An ontology-oriented approach to represent and compare the functional behaviour of event-based systems. 2017.

6 GuangChun L, Lu W, Hanhong X. A novel web application frame developed by MVC. *ACM SIGSOFT Software Engineering Notes* 2003;**28**:7. doi:10.1145/638750.638779

7 McGovern J, Sims O, Jain A, *et al. Enterprise Service Oriented Architectures*. Berlin/Heidelberg: : Springer-Verlag 2006. doi:10.1007/1-4020-3705-8

8 Erl T, Carlyle B, Pautasso C, *et al.* SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST. *The Prentice Hall service technology series* 2013.

9 Kawamoto K, Hongsermeier T, the AWJO, *et al.* Key principles for a national clinical decision support knowledge sharing framework: synthesis of insights from leading subject matter experts. *academicoupcom* doi:10.1136/amiajnl-2012-000887)

10 Serbanati LD, Ricci FL, Mercurio G, *et al.* Steps towards a digital health ecosystem. *Journal of Biomedical Informatics* 2011;**44**:621–36. doi:10.1016/j.jbi.2011.02.011

11 Berczuk SP, Appleton B. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley Longman Publishing Co., Inc. 2002.

12 Hage J. Software configuration management. 2010.

13 Paterno MD, Goldberg H, Simonaitis L, *et al.* Using a Service Oriented Architecture Approach to Clinical Decision Support - Performance Results from Two CDS Consortium Demonstrations. *AMIA* 2012.

14 Heckle RR. Security Dilemma - Healthcare Clinicians at Work. *IEEE Security & Privacy* 2011;**9**:14–9. doi:10.1109/MSP.2011.74

15 Becich M, Santana-Santos L, Gullapalli R, *et al.* Next generation sequencing in clinical medicine: Challenges and lessons for pathology and biomedical informatics. *J Pathol Inform* 2012;**3**:40. doi:10.4103/2153-3539.103013

16 Welch BM, Loya SR, Eilbeck K, *et al.* A proposed clinical decision support architecture capable of supporting whole genome sequence information. *J Pers Med* 2014;**4**:176–99. doi:10.3390/jpm4020176

17 Curran J, Fenton N, Freedman D. *Misunderstanding the internet*. 2016.

18 Dwork C. Differential Privacy: A Survey of Results. In: Agrawal M, Du D, Duan Z, *et al.*, eds. Berlin, Heidelberg: : Springer Berlin Heidelberg 2008. 1–19. doi:10.1007/978-3-540-79228-4_1

19 Oh S, Cha J, Ji M, *et al.* Architecture Design of Healthcare Software-as-a-Service Platform for Cloud-Based Clinical Decision Support Service. *Healthcare Informatics Research* 2015;**21**:102–10. doi:10.4258/hir.2015.21.2.102

20 Bahga A, Madisetti VK. A Cloud-based Approach for Interoperable Electronic Health Records (EHRs). *IEEE J Biomed Health Inform* 2013;**17**:894–906. doi:10.1109/JBHI.2013.2257818

21 Marco-Ruiz L, Maldonado JA, Traver V, *et al.* Meta-architecture for the interoperability and knowledge management of archetype-based clinical decision support systems. *BHI* 2014;:517–21. doi:10.1109/BHI.2014.6864416

22 Hayes-Roth F, Jacobstein N. The state of knowledge-based systems. *Communications of the ACM* 1994;**37**:26–39. doi:10.1145/175247.175249

23 Bender D, Sartipi K. HL7 FHIR: An Agile and RESTful approach to healthcare information exchange. IEEE 326–31. doi:10.1109/CBMS.2013.6627810

24 Mandel JC, Kreda DA, Mandl KD, *et al.* SMART on FHIR: a standards-based, interoperable apps platform for electronic health records. *Journal of the American Medical Informatics Association* 2016;**23**:899–908. doi:10.1093/jamia/ocv189

25 Getting Started with FHIR. https://www.hl7.org/fhir/modules.html (accessed 7 Aug2018).

26 Mead CN. Data interchange standards in healthcare IT--computable semantic interoperability: now possible but still difficult, do we really need a better mousetrap? *J Healthc Inf Manag* 2006;**20**:71–8.

27 Rajeev D, Staes CJ, Evans RS, *et al.* Development of an electronic public health case report using HL7 v2.5 to meet public health needs. *Journal of the American Medical Informatics Association* 2010;**17**:34–41. doi:10.1197/jamia.M3299

28 Argonaut Project. http://argonautwiki.hl7.org/index.php?title=Main_Page (accessed 7 Aug2018).

29 Marquard B, Bashyam N, Haas E, *et al.* FHIR US Core Implementation Guide. http://www.hl7.org/fhir/us/core/ (accessed 7 Aug2018).

30 FHIR Clinical Reasoning Module. http://hl7.org/fhir/clinicalreasoning-module.html (accessed 15 Aug2018).

31 OMG Healthcare DTF. https://www.omg.org/healthcare/ (accessed 6 Aug2018).

32 *Field Guide to Shareable Clinical Pathways using BPMN CMMN DMN in Healthcare Version: 1.0*. https://bookstack.hspconsortium.org/books/field-guide-to-shareable-clinical-pathways-using-bpmn-cmmn-dmn-in-healthcare-version-10

33 Fielding R. *Fielding - 2000 - Architectural Styles and the Design of Network-based Software Architectures*. 2000;:1–180.

34 Sakimura N, Bradley J, OpenID MJT, *et al.* OpenID Connect Core 1.0 incorporating errata set 1. *imgsaufca*

35 HSPC SMART-on-FHIR Application Gallery. https://gallery.hspconsortium.org (accessed 6 Aug2018).

36 SMART Health IT App Gallery. https://apps.smarthealthit.org (accessed 6 Aug2018).

37 Braunstein ML. SMART on FHIR. In: *Health Informatics on FHIR: How HL7's New API is Transforming Healthcare*. Cham: : Springer, Cham 2018. 205–25. doi:10.1007/978-3-319-93414-3_10

38 CDS Hooks 1.0. https://cds-hooks.org/hooks/template/ (accessed 6 Aug2018).

39 Kumar AP, Kapur R. *Discrete simulation application-scheduling staff for the emergency room*. New York, New York, USA: : ACM 1989. doi:10.1145/76738.76880

40 Cellier FE. Combined continuous/discrete system simulation languages: usefulness, experiences and future development. *ACM SIGSIM Simulation Digest* 1977;**9**:18–21. doi:10.1145/1102505.1102514

41 Preissl R, Wong TM, Datta P, *et al. Compass: a scalable simulator for an architecture for cognitive computing*. IEEE Computer Society Press 2012.

42 Wang Z, Cui Y, Shi J. A Framework of Discrete-Event Simulation Modeling for Prognostics and Health Management (PHM) in Airline Industry. *IEEE Systems Journal* 2017;**11**:2227–38. doi:10.1109/JSYST.2015.2466456

43 Iannoni AP, Morabito R. A discrete simulation analysis of a logistics supply system. *Transportation Research Part E: Logistics and Transportation Review* 2006;**42**:191–210. doi:10.1016/j.tre.2004.10.002

44 Law AM, Kelton WD. *Simulation modeling and analysis*. 1991.

45 CDS Hooks 1.0. https://cds-hooks.org/specification/1.0/#passing-the-access-token-to-the-cds-service (accessed 6 Aug2018).

46 Khalilia M, Choi M, Henderson A, *et al.* Clinical Predictive Modeling Development and Deployment through FHIR Web Services. *AMIA Annual Symposium Proceedings* 2015;**2015**:717–344. doi:10.1136/amiajnl-2013-002033

47 Zhang Y-F, Gou L, Tian Y, *et al.* Design and Development of a Sharable Clinical Decision Support System Based on a Semantic Web Service Framework. *J Med Syst* 2016;**40**:118–14. doi:10.1007/s10916-016-0472-y

48 Kawamoto K, Medical DLJOTA, 2007. Proposal for Fulfilling Strategic Objectives of the U.S. Roadmap for National Action on Decision Support through a Service-oriented Architecture Leveraging HL7 Services | Journal of the American Medical Informatics Association | Oxford Academic. *academicoupcom*

49 Middleton B, Boxwala AA, Overhage JM, *et al.* Getting Hooked on CDS - Toward an Open Standard Architecture for Clinical Decision Support in Leading Electronic Medical Records. *AMIA* 2017.

50 Dixon BE. A pilot study of distributed knowledge management and clinical decision support in the cloud. *Artificial Intelligence in Medicine* 2013;**59**:45–53. doi:10.1016/j.artmed.2013.03.004

51 Cloud Functions - Serverless Environment to Build and Connect Cloud Services | Google Cloud Platform. https://cloud.google.com/functions/

52 AWS Lambda – Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda/

53 Azure Functions—Serverless Architecture | Microsoft Azure. https://azure.microsoft.com/en-us/services/functions/

54 Fowler M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc. 2002.

55 Freeman E, Robson E, Bates B, *et al. Head First Design Patterns: A Brain-Friendly Guide*. 2004.

56 Hohpe G, Woolf B. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. 2004.

57 Ludovici A, Calveras A. A proxy design to leverage the interconnection of CoAP Wireless Sensor Networks with Web applications. *Sensors (Basel)* 2015;**15**:1217–44. doi:10.3390/s150101217

58 Loreto S, Saint-Andre P, Salsano S, *et al.* Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC Editor 2011. doi:10.17487/rfc6202

59 Severance CR. Roy T. Fielding - Understanding the REST Style. *IEEE Computer* 2015.

60 Liskin O, Singer L, Schneider K. *Teaching old services new tricks: adding HATEOAS support as an afterthought*. New York, New York, USA: : ACM 2011. doi:10.1145/1967428.1967432

61 Lairson DR, Chung TH, Smith LG, *et al.* Estimating development cost of an interactive website based cancer screening promotion program. *Eval Program Plann* 2015;**50**:56–62. doi:10.1016/j.evalprogplan.2015.01.009

62 Insup Lee, Pappas GJ, Cleaveland R, *et al.* High-Confidence Medical Device Software and Systems. *Computer* 2006;**39**:33–8. doi:10.1109/MC.2006.127

63 Samwald M, Fehre K, de Bruin J, *et al.* The Arden Syntax standard for clinical decision support: Experiences and directions. *Journal of Biomedical Informatics* 2012;**45**:711–8. doi:10.1016/j.jbi.2012.02.001

64 Lyonnard D, Yoo S, Baghdadi A, *et al.* Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. New York, New York, USA: : ACM Press 2001. 518–23. doi:10.1145/378239.379015

65 Nidhra S, Applications JDJOESA, 2012. Black box and white box testing techniques-a literature review. *academiaedu*

66 Khan ME, Appl FKIJACS, 2012. A comparative study of white box, black box and grey box testing techniques. *Citeseer*

67 Choi SW, Her JS, Kim SD. Modeling QoS Attributes and Metrics for Evaluating Services in SOA Considering Consumers' Perspective as the First Class Requirement. IEEE 398–405. doi:10.1109/APSCC.2007.72

68 Xie L, Luo J, Qiu J, *et al.* Availability “weak point” analysis over an SOA deployment framework. IEEE 473–80. doi:10.1109/NOMS.2008.4575170

69 Basit MA, Baldwin KL, Kannan V, *et al.* Agile Acceptance Test-Driven Development of Clinical Decision Support Advisories: Feasibility of Using Open Source Software. *JMIR Med Inform* 2018;**6**:e23. doi:10.2196/medinform.9679

70 Orso A, Shi N, Harrold MJ, *et al. Scaling regression testing to large software systems*. ACM 2004. doi:10.1145/1041685.1029928

71 Chun BN. DART: Distributed Automated Regression Testing for Large-Scale Network Applications. In: *Principles of Distributed Systems*. Berlin,

Heidelberg: : Springer, Berlin, Heidelberg 2004. 20–36. doi:10.1007/11516798_2

72 Kwon J, Johnson ME. Protecting Patient Data-The Economic Perspective of Healthcare Security. *IEEE Security & Privacy* 2015;**13**:90–5. doi:10.1109/MSP.2015.113

73 Ariyapperuma S, Mitchell CJ. Security vulnerabilities in DNS and DNSSEC. IEEE 335–42. doi:10.1109/ARES.2007.139

74 Felten EW, Schneider MA. *Timing attacks on Web privacy*. New York, New York, USA: : ACM 2000. doi:10.1145/352600.352606

75 Lee P. Interoperable, Computable Clinical Content and Integrated Knowledge Environments. 2017. 22.http://www.hl7.org/events/fhir/roundtable/2017/12/final.presentations.cfm

76 Hu B, Sharif U, Koner R, *et al.* Random Finite Set Based Bayesian Filtering with OpenCL in a Heterogeneous Platform. *Sensors (Basel)* 2017;**17**:843. doi:10.3390/s17040843

77 Češnovar R, Štrumbelj E. Bayesian Lasso and multinomial logistic regression on GPU. *PLoS ONE* 2017;**12**:e0180343. doi:10.1371/journal.pone.0180343

78 Kalinowski J, Wennmohs F, Neese F. Arbitrary Angular Momentum Electron Repulsion Integrals with Graphical Processing Units: Application to the Resolution of Identity Hartree-Fock Method. *J Chem Theory Comput* 2017;**13**:3160–70. doi:10.1021/acs.jctc.7b00030

79 OpenTracing.org. http://opentracing.io (accessed 13 Jul2018).

80 API4KB. http://www.omgwiki.org/API4KB/doku.php (accessed 13 Jul2018).

81 preston/journeta. https://github.com/preston/journeta

82 Chen S. Scalable Real-time Complex Event Processing at Uber – Kafka Summit. Published Online First: 8 May 2017.https://kafka-summit.org/sessions/scalable-real-time-complex-event-processing-uber/

83 Luckham D. SOA, EDA, BPM and CEP are all Complementary Part II. 2007. http://www.complexevents.com/2007/07/08/soa-eda-bpm-and-cep-are-all-complementary-part-2/

84 Advisor NBPN, 2000. Understanding the OSI 7-layer model. 3AD.

85 Luckham D, Schulte R. Event Processing Technical Society. 2011. http://www.complexevents.com/2011/08/23/event-processing-glossary-version-2/

86 Flynn AJ, Boisvert P, Gittlen N, *et al.* Architecture and Initial Development of a Knowledge-as-a-Service Activator for Computable Knowledge Objects for Health. *Stud Health Technol Inform* 2018;**247**:401–5.

87 Flynn AJ, Bahulekar N, Boisvert P, *et al.* Architecture and Initial Development of a Digital Library Platform for Computable Knowledge Objects for Health. *Stud Health Technol Inform* 2017;**235**:496–500.

88 Fry E, Goodnough J, Nanjo C. **HL7 Version 3 Specification: Event Publish & Subscribe Service Interface – Release 1 – US Realm**. 2015. http://www.hl7.org/implement/standards/product_brief.cfm?product_id=390

89 Kawamoto K, Honey A, Rubin K. The HL7-OMG Healthcare Services Specification Project: motivation, methodology, and deliverables for enabling a semantically interoperable service-oriented architecture for healthcare. *J Am Med Inform Assoc* 2009;**16**:874–81. doi:10.1197/jamia.M3123

90 Cugola G, Margara A. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)* 2012;**44**:15–62. doi:10.1145/2187671.2187677

91 Jayasekara S, Perera S, Dayarathna M, *et al. Continuous analytics on geospatial data streams with WSO2 complex event processor*. New York, New York, USA: : ACM 2015. doi:10.1145/2675743.2772585

92 Jayasinghe M, Jayawardena A, Rupasinghe B, *et al. Continuous analytics on graph data streams using WSO2 complex event processor*. New York, New York, USA: : ACM 2016. doi:10.1145/2933267.2933508

93 Akram N, Siriwardene S, Jayasinghe M, *et al. Anomaly Detection of Manufacturing Equipment via High Performance RDF Data Stream Processing: Grand Challenge*. New York, New York, USA: : ACM 2017. doi:10.1145/3093742.3095100

94 Vegh L, Miclea L, 2016. Secure and efficient communication in cyber-physical systems through cryptography and complex event processing. IEEE 273–6. doi:10.1109/ICComm.2016.7528290

95 What Is OpenAPI? https://swagger.io/docs/specification/about/ (accessed 10 Aug2018).

96 RAML Version 1.0. https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/ (accessed 14 Aug2018).

97 JSON Schema. http://json-schema.org

98 Steinman JS, Steinman JS. Discrete-event simulation and the event horizon. *ACM SIGSIM Simulation Digest* 1994;**24**:39–49. doi:10.1145/182478.182490

99 Steinman JS, Steinman JS. *Discrete-event simulation and the event horizon part 2: event list management*. ACM 1996. doi:10.1145/238793.238841

100 Boxwala AA, Rocha BH, Maviglia S, *et al.* A multi-layered framework for disseminating knowledge for computer-based decision support. *Journal of the American Medical Informatics Association* 2011;**18**:i132–9. doi:10.1136/amiajnl-2011-000334