Microservices in Audiovisual Archives: An Exploration of Constructing Microservices for Processing Archival Audiovisual Information

Annie Schweikert, New York University, USA
Dave Rice, City University of New York, USA
DOI: https://doi.org/10.35320/ij.v0i50.70

Abstract

Properly managing audiovisual archival material requires identifying, using, and possibly creating the right tools and workflows to facilitate archival objectives. In creating these workflows, two models are possible. One model is the monolithic architecture, which includes complex all-in-one systems (for instance, a comprehensive digital asset management system). Another model is the microservice architecture, which combines independent tools into a loosely coupled system based upon common underlying standards and understandings. In a microservice architecture, an individual tool may be added, replaced, or upgraded independently of the other tools.

This document describes and examines strategies for designing lightweight microservice environments for the processing of digital, file-based, audiovisual data within an archive. It guides the reader through the design of a simple example microservice architecture by establishing foundational archival frameworks for microservice design, describing examples of packages and microservices tailored to audiovisual archives, and finally demonstrating an end-to-end workflow.

This document does not intend to be a standard for the design of audiovisual microservices, but seeks to contribute a use case to the work and dialogue of many audiovisual archives exploring and implementing microservice structures; see in particularly the compiled, collaborative documentation at https://github.com/amiaopensource/open-workflows. This document presumes an overview understanding of the Reference Model for an Open Archival Information System[1] (OAIS). Since the document intends to focus on archival routines for audiovisual content, an introduction to FFmpeg[2] can also be helpful.

Microservices and Monoliths

Properly managing audiovisual archival material requires identifying, using, and possibly creating the right tools and workflows to facilitate archival objectives. Such tools may include various independent utilities that change based on the material; for analog video media, this might include cleaning supplies, a video tape player, a time base corrector, or a video card for digitization. Archives of analog media are generally rich with discrete tools independently selected for their own focused objectives. These items may more or less be selected independently and combined into a loosely coupled system, based upon common underlying standards and understandings. An individual tool may be added, replaced, or upgraded independently of the other tools.

Alternatively, a tool could be in the form of a complex all-in-one system, for instance a comprehensive digital asset management system or an archival vendor that cares for multiple aspects of a project. Archives of digital media, for which tasks can be more easily automated, present the opportunity to integrate tools into a single centralized system.

1 https://public.ccsds.org/pubs/650x0m2.pdf
2 https://www.ffmpeg.org

This system can manage multiple tasks through a single application and provide a comprehensive, foundational environment for archival workflows. For example, a complex digital asset management system may be able to facilitate cataloging, digitization, transcoding, and access as a single, self-contained product, thereby clustering the objectives of several tools into a singular, comprehensive system.

These descriptions illustrate the difference between ***microservice architectures*** and ***monolithic architectures*** as applied to an audiovisual archive. Both styles have distinct opportunities and challenges. A monolithic system, such as a complex digital asset management application, may suggest reliability and efficiency, as control of each task is integrated under a single company's umbrella. However, if the monolith contains some strong functions and some weak functions, it may be difficult to adapt the architecture to an archive's preferred workflow, or remix the monolith with other, preferred tools. The addition of a plugin architecture or API may allow the application to integrate new objectives and roles, but in such an application individual tools do not operate independently of the product.

In a microservice architecture, on the other hand, many tools that accomplish discrete, bounded tasks—each task a "microservice"—are combined into archival workflows like links in a chain. The customizability of such a workflow structure requires independent knowledge of each tool and its functions, the field's standards and best practices for each task, and the ground-up construction of an archive's workflows. Microservices may therefore necessitate a much more comprehensive understanding of all tools and objects involved, and present greater risks to the archive if misunderstandings are integrated into the design. Nonetheless, a microservice-based archival environment supported by the archive's staff and/or related communities may be easier to independently evolve over time, and may be easier to steer or customize to the unique needs of its collections or its organization.

Both systems work to accomplish the objectives of an audiovisual archive: to preserve, manage, and make accessible its archival materials. The customizability of a microservice architecture should not imply that monolithic architectures are inherently less suited to archival administration. Some of the reasons to favor popular monolithic architectures include situations when an archive wants to use a tool that is known to be compatible and comprehensive, or when the workflow defined by a monolithic architecture matches the goals of the workflow defined locally by the archive. However, there are situations where a microservice approach is favorable. Such situations may include:

- when the design of workflows is required to be agile and responsive to serve its target communities, whose evolving needs may demand continuous development, flexible scaling, and a shortening of the time between software development and release;
- when the staff of the archive has or is willing to acquire comprehensive technical knowledge of its objects, processes, and tools, and/or desires to work with a provider that empowers the archive to implement highly customized workflows based on this technical knowledge;
- when different archival communities and institutions wish to pool resources in a collaborative, open source approach to preservation workflows;
- optionally, when the responsibility for the function, maintenance, and design of archival functions is appropriate to rest on the archive's personnel rather than an outsourced company;
- when the opportunity for a monolith is not worth the cost of a long-term investment, such as when the work is managed in a temporary state or the evolution of technology is paced such that the archive must be prepared to examine and replace its tools and components on an ongoing basis.

The administrator of the archive who works on a web of microservices, as opposed to within the boundaries of a monolith, encounters unusual opportunities for creativity as well as constraints. In his book *Code 2.0*, Lawrence Lessig offers a comparison between the coded reality of Second Life and real-world legal systems. He describes how, in the real world, it typically constitutes illegal nuisance and/or trespassing to fly an airplane at a low height over someone's private property, but airplanes are free to fly at a high height over the same property. He describes how in Second Life any character may fly at a height of more than 15 meters over someone else's virtual land, but may only move in the space from 0-15 meters over someone else's land if enabled to do so by the landowner's settings. Lessig describes:

> *But notice the important difference. In real space, the law means you can be penalized for violating the "high/low" rule. In Second Life, you simply can't violate the 15-meter rule. The rule is part of the code. The code controls how you are in Second Life. There isn't a choice about obeying the rule or not, any more than there's a choice about obeying gravity.*
>
> *So code is law here. That code/law enforces its control directly. But obviously, this code (like law) changes. The key is to recognize that this change in the code is (unlike the laws of nature) crafted to reflect choices and values of the coders.*[3]

The monolith may present a similar experience as Second Life, where the options and opportunities may be bounded by what is in code that is uncontrolled by the archive; for instance, the options allowed in a particular decision may be fixed within a drop-down menu of an interface, or a yes/no dialogue box, and limit opportunity.

This document will describe and examine strategies for designing lightweight microservice environments for the processing of digital, file-based, audiovisual data within an archive. It presumes an overview understanding of the Reference Model for an Open Archival Information System (OAIS). The document also makes references to programming archival routines in command languages, but seeks to provide examples in pseudo-code rather than favoring any particular computer language. Since the document intends to focus on archival routines for audiovisual content, a basic introduction to FFmpeg may be helpful.

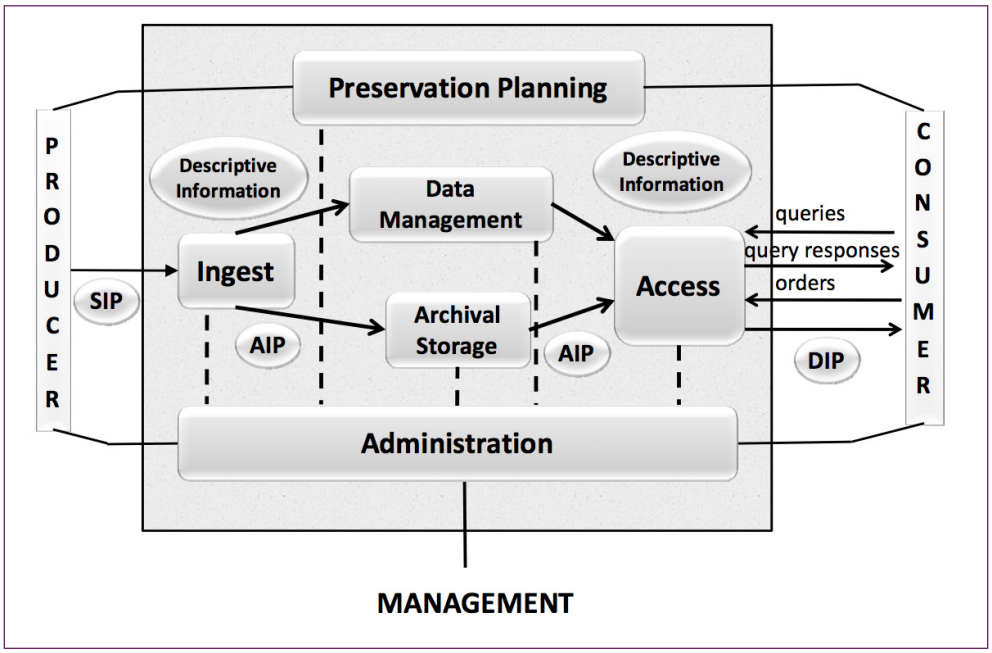
Though this document does not address cross-archives collaboration in detail, it is important to note that microservice-based archival designs are often more successful when employing collaboration amongst archival communities and open source approaches. Many examples of open, archival microservice documentation may be found at https://github.com/amiaopensource/open-workflows. This document does not intend to purport to be a standard for the design of audiovisual microservices, but seeks to contribute to and build upon this successful dialogue and implementation across audiovisual archives.

1. Building Microservices with the Open Archival Information System Reference Model

The Open Archival Information System (OAIS) reference model defines mandates, objectives, and methods for archives, and provides a clear set of guidance to the design of both monolithic or microservices systems. The OAIS documentation also provides a set of vocabulary for describing basic components and functions within an archive, as

3 Lessig, Lawrence. 2011. http://codev2.cc/download+remix/

well as a structure that divides workflow between submission, dissemination, and archival management. The workflows that bridge these states are referred to as Ingest, in which a submission is moved into a state of archival management, and Access, in which content is moved from archival management to dissemination. The content in its form before ingest workflows, within archival management, and after access workflows is called the Submission Information Package (SIP), Archival Information Package (AIP), and Dissemination Information Package (DIP), respectively.



"Figure 4-1: OAIS Functional Entities." From Consultative Committee for Space Data Systems, Reference Model for an Open Archival Information System (OAIS), Recommended Practice, CCSDS 650.0-M-2 (Magenta Book). Issue 2, June 2012.

An archive will likely conceive and define several forms of SIPs and DIPs for different purposes. If an archive is too open or undefined in regards to the expectations of the SIP, the Ingest process and subsequent archival management of the content may be harder to automate as it becomes less clear what can be expected. Some archives may be able to constrict submission into a single form, but often the diversity of incoming content and its creators necessitates a short list of SIP definitions to accommodate different workflows. Likewise, an archive may need to support multiple types of access, such as supporting access to audiovisual content in the form of a web stream for the public or as a higher quality file for production work.

1.1 SIP Definitions

Content Information[4] arriving at or created within an archive for preservation may be considered a Submission Information Package (SIP). In its analog equivalent, a SIP could be something simple and self-contained such as a videotape with a handwritten label, or it may be more complex, such as a collection of media objects accompanied by submission documentation. A digital SIP, by comparison, could arrive as a set of video files with embedded metadata or as a chaotic collection spread across many hard drives. An archive receives SIPs from submitters, who may be creators (such as producers in a broadcast archive) or donors (common in cultural heritage organizations).

Requiring too little information or context to accompany a qualifying SIP could cause risks or confusion in the ongoing management of the Content Information. On the other hand, mandating too much information may bottleneck the Ingest process or burden submitters. It is recommended that the structure, form, and minimal requirements of a SIP are defined in collaboration between the archive and the submitters. There may be worthwhile reasons to make controlled customizations to SIP definitions for different Ingest workflows. Employing the concept of locally defined SIPs during the acquisition of content helps arrange content more precisely, so that the boundaries between one SIP and another is clarified, so that more automation is feasible, and so that there may be a greater awareness of the state or quality of the SIP.

In audiovisual archives the SIP is generally composed of audiovisual media (Content Information) and supporting metadata (Preservation Description Information). For analog formats, the media and metadata may be physically attached—for example, labels on a videocassette—but metadata could exist as separate documents or even emails that provide information about the media. With some digital audiovisual collections, that metadata may be wholly embedded within the file (such as EXIF or IPTC data for images, or ID3 tags for audio), but it is more likely that archives will receive media with supplemental metadata as separate forms or files when obtaining a SIP.

OAIS requires archives to establish a SIP Definition to clarify the requirements and recommendations for SIPs. For instance, an archive may require that certain paperwork or a web-form must be completed for each SIP with pertinent data such as the identifiers, title, description, and access rights. Archives may also mandate that the media contained in a SIP adhere to a predefined list of formats that the archive is prepared to manage. The SIP Definition should also distinguish the media (Content Information) from the supporting metadata (Preservation Description Information) in a manner that is unambiguous and clear. Furthermore, the SIP Definition should establish the boundaries between one SIP and another clearly, for example defining whether a directory containing a single collection of multiple media files constitutes one SIP or a set of multiple SIPs.

4 'Content Information' is defined by the OAIS as "A set of information that is the original target of preservation or that includes part or all of that information."

A selection of recommendations for SIP Definitions that are useful in building Ingest workflows includes:

- Define what forms of media (whether analog or digital) are accepted
- Define what metadata types are required and recommended
- Define what form the metadata should be provided in, such as paperwork, xml, csv, informal text)
- Define the structure for digital media delivery (e.g., whether files should be grouped into directories to represent a form of organization or semantics)[5]

1.2 SIP + ? = AIP

Why can't the SIP simply become the AIP?

The OAIS reference model states that the SIP alone is not well-prepared enough to be considered for long-term storage, and is thus not suitable to automatically be considered as an AIP (Archival Information Package). The SIP is missing pertinent Preservation Description Information, which consists of four categories of metadata (provenance, context, reference, and fixity) and includes information such as checksums, identifiers, or preservation action documentation. Many of these values must be generated during the Ingest process. The Content Information of a SIP may also require some review or quality control work to ensure that the data is as it should be, is not malformed, and is identified correctly. The AIP should add what is needed for readiness for long-term storage and a status of permanence. AIPs generally will require information for reference, provenance, fixity checking, and access rights, even if this information was not part of the SIP.

For audiovisual materials, the AIP may also add specifics such as technical metadata reports (such as those produced by FFmpeg or MediaInfo), frame checksums (such as produced by `` `ffmpeg -i INPUT -f framemd5 -an OUTPUT.framemd5` ``), and possibly derivatives. Whereas non-audiovisual file formats may support quick generation of DIPs (such as derivatives); audiovisual content is often very large and more time-consuming to process, thus there is more of an incentive to generate DIPs in the process of generating the AIP.

1.3 DIP Definitions

Content Information being passed to the user is usually normalized and compressed from the Content Information as submitted (i.e., the original content packaged within the SIP and AIP). It is often packaged and delivered with the same metadata from the AIP.

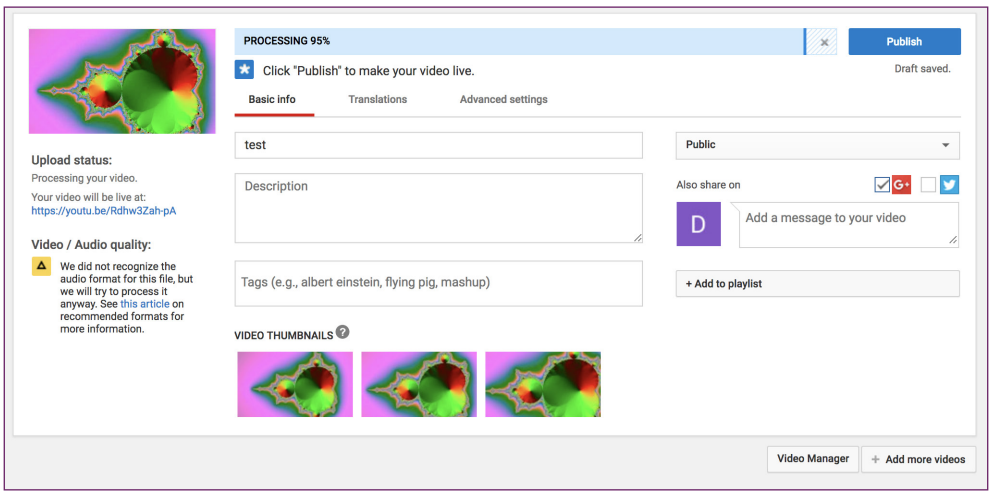
The DIP is relatively simple; it can consist of as little as a derivative and a metadata file. Considered within the frame of microservices, it is also simple to generate, as it requires only a consistent access-level file and a pre-existing metadata file. However, it is worth attention at the time of Ingest actions, as it is simple to generate derivatives from Content Information files to have on hand for quick access if this task is performed within the chain of Ingest microservices.

5 An example of Archivematica's definition of a SIP Structure may be found at https://wiki.archivematica.org/SIP_Structure. Archivematica will be discussed in greater detail below.

1.4 An OAIS-like Example: YouTube

To provide an example of the SIP to AIP transformation, let us consider YouTube. The submission interface of YouTube offers a very controlled method for providing a SIP. Here the SIP includes exactly one audiovisual file plus a controlled set of metadata including:

- Title
- Description
- Tags
- Privacy Status
- Category
- License



While the YouTube SIP may simply be a single media file (Content Information) and a webform entry (similar to Preservation Description Information), YouTube generates and gathers a substantial amount of additional information in order to prepare for long-term storage, management, and access for the content. These additions may include:

- Automated captions
- User reviews
- Content flagging
- Several derivatives of various sizes and media formats
- Embed codes and short urls
- Assessment of audio watermarks
- View count and statistics on use
- Technical metadata

YouTube's methods to convert their SIPs into "AIPs" add a significant amount of information to the package in order to increase the functionality, accessibility, and reuse of the material. However, the original Content Information is often transcoded to YouTube's standards, and the closest to Preservation Description Information that YouTube's package comes is the Access Rights and Context implied by the video's availability on YouTube. Ultimately, this added information does not conform to the OAIS AIP specifications, and YouTube's packaging procedure comes closer to producing a DIP.

1.5 An OAIS-like Example: Internet Archive

The Internet Archive (archive.org) manages a more flexible SIP definition than YouTube. Here the submission process offers many more metadata options, including comprehensive opportunities to provide customized metadata. Additionally, the archive can access multiple files within a single SIP. The Internet Archive provides some transparency to its techniques for processing a SIP into an AIP.

For example, let us consider this Internet Archive item as an AIP: https://archive.org/details/umatic_controlled_damage

- All contents (including both Content Information and Package Description Information) may be found at https://archive.org/download/umatic_controlled_damage
- The archive adds a manifest that documents checksums and file attributes for all files: https://archive.org/download/umatic_controlled_damage/umatic_controlled_damage_files.xml
- This document also lists whether a file was considered part of the original submission (source="original") or if it was created by an Internet Archive microservice (source="derivative"). The package also maintains an XML file that concentrates all descriptive information for the package, found at: https://archive.org/download/umatic_controlled_damage/umatic_controlled_damage_meta.xml
- The processing history of that AIP since the Ingest started can be found at https://archive.org/history/umatic_controlled_damage. From this location, an archivist may review the logs of each microservice applied to the package and perform a comprehensive audit of all processing actions.

1.6 An OAIS Example: Archivematica

Finally, Archivematica is an example of a digital preservation system explicitly based on the OAIS reference model. Archivematica offers a package of free and open-source tools for digital preservation, including ingest, storage, and access to digital archival content.[6] Archivematica's SIP is highly structured, to the point where inclusion of certain files (such as metadata files) trigger specific workflows (in this case, processing by the Archivematica system).[7] Documentation is clear and publicly available on the Archivematica Wiki.

The process of turning this SIP into an AIP is similarly highly structured and includes reorganizing SIP content, generating metadata, and performing audits of the SIP and AIP. The final version of the AIP is packaged in accordance with the Library of Congress Bagit specification; the "data" directory includes the original content, logs from the Archivematica Ingest process, thumbnails for access, and a file with metadata. A full sample AIP structure can be found at https://wiki.archivematica.org/AIP_structure.

6 "What is Archivematica?," *Archivematica documentation*, Artefactual Systems Inc., 2019. https://www.archivematica.org/en/docs/archivematica-1.9/getting-started/overview/intro/#intro

7 "SIP Structure," *Archivematica development wiki*, Archivematica contributors, 30 Mar. 2017. https://wiki.archivematica.org/SIP_Structure

2. Expressing Package Definitions and Considering Storage

Because monolithic digital archiving systems manage most or all aspects of archival packaging and storage, archivists are not always required or able to define the structure and storage location of the AIP. These attributes may be defined behind-the-scenes or be described in system documentation.

Without a monolith, it becomes more pertinent for the archive to define and describe its Information Packages clearly in order to build microservices upon the expectations and standardization that the package definitions can bring. In addition, the ability to standardize Information Packages from the ground up also allows the archivist to craft standardized microservices that hook neatly into the structure of the Information Package. Shaping an Information Package according to the needs of the archive then empowers the archivist to create workflows that similarly address the needs of the archive. By the end of such a process, the archive can be powered by chains of microservices linked together, building on each other and on the Information Packages they define.

2.1 Guide to Pseudo-Code

The OAIS reference model is a framework with a wide range of possible implementations, not a prescriptive template. However, in order to describe relationships between packages and microservices in a practical way, this paper will provide the reader with concrete examples of packages and microservices. Examples throughout the paper are drawn from and based on the set of microservices in use in the archives of City University Television (CUNY TV), the television station of the City University of New York.

For the sake of expressing package definitions through this document, the following pseudo-code system is proposed:

- By default names represent the path of a **file** of an Information Package.
  - Names ending in a '/' represent a **directory**.
- Components of names wrapped in '`{}`' (curly brackets) and preceded by a '`$`' represent variables that would be conditional to each Information Package.
- Components of names followed by an '`*`' (asterisk) indicates that the file is separately stored from the rest of the Content Information.
- Names wrapping in '`[]`' (square brackets) represent data that is not stored as a file, such as a database record. The use of square brackets also indicates that the data is separately stored from the rest of the Content Information.
- Names may be followed by the following flags that are wrapped in '`()`' (parentheses) and are comma-delimited.
  - `1`: indicates that exactly one occurrence of the data is required
  - `?`: indicates that zero or one occurrence of the data is required
  - `+`: indicates that more than one is allowed

For instance, a SIP definition for a single file and an associated web-form record could be expressed as:[8]

Submission Information Package Expression (Example 1)

```
${CONTENT} (1)
[web-form-record] (1)
```

If the received content is then received and arranged to create an Archival Information Package which stored in a directory structure that arranges the data, the result may appear as follows:

Archival Information Package Expression (Example 2)

```
${PACKAGE-UUID}/content/${CONTENT} (1)
${PACKAGE-UUID}/metadata/web-form-record.txt (1)
```

A more complex Archival Information Package for video files may appear as follows:

Archival Information Package Expression (Example 3)

```
${PACKAGE-UUID}/content/${CONTENT} (+)
${PACKAGE-UUID}/metadata/web-form-record.txt (1)
${PACKAGE-UUID}/metadata/logs/ingest.txt (1)
${PACKAGE-UUID}/metadata/technical-metadata.txt (1)
${PACKAGE-UUID}/metadata/checksums.md5 (1)
${PACKAGE-UUID}/metadata/mets.xml (1)
[database record of descriptive metadata] (1)
```

Alternatively, in cases when the archive intends to store technical metadata and records of preservation events in a database as opposed to a package, the same data could be defined as:

Archival Information Package Expression (Example 4)

```
${CONTENT_UUID} (+)
[web-form-record] (1)
[database event record of ingest process] (1)
[database object record of technical-metadata] (1)
[database object record of checksums] (1)
[database record of METS data] (1)
[database record of descriptive metadata] (1)
```

The Dissemination Information Package made at the same time as these AIPs might hold derivatives, their logs, and descriptive metadata to guide the user.

Dissemination Information Package Expression (Example 5)

```
${PACKAGE-UUID}/derivatives/web/${CONTENT}.mp4 (+)
${PACKAGE-UUID}/derivatives/edit/${CONTENT}.mov (+)
${PACKAGE-UUID}/metadata/logs/make-web-derivative.txt (+)
${PACKAGE-UUID}/metadata/logs/make-edit-derivative.txt (+)
[database record of descriptive metadata] (1)
```

8 The names used here are provided as examples and not necessarily recommendations.

2.2 Separating Content Information and Preservation Description Information

The AIP is a theoretical concept with many different implementations, none of which are required to be wrapped up in a literal package; the examples above present methods of separating Content Information from Preservation Description Information. Examples 3 and 4 describe identical content, just packaged differently. (A migration from example 3 to example 4 would comprise a lossless repackaging.) In Example 3, the package is relatively self-contained, consisting of a directory structure with original content files and supporting technical and administrative metadata (part of the Preservation Description Information, which includes records of preservation actions taken). The descriptive metadata alone is stored in a separate database, constituting Descriptive Information. Example 4 expands on the separation of the AIP by moving the original content into preservation storage, pre-created derivatives into a different storage system, and all metadata (descriptive, administrative, and technical) into database records.

Separating content from metadata in storage can be a necessity for audiovisual collections, due to relatively larger data storage and cost requirements. For instance, an archive may determine that larger original content should be stored on lower cost LTO tapes, while the derivatives remain in online storage because of its greater priority for access. This scenario optimizes storage, but increases the need and potential risks involved in ensuring that all links between the components of the AIP are well-maintained and understood. For instance, the discovery of a descriptive metadata record for a package in a database should inform as to the location and identity of the package that it describes. Similarly, the package in storage should indicate and reference the related set of descriptive information.

In both examples 3 and 4, the descriptive metadata is stored in a separate database, which allows the description to evolve over time as the archive learns more about the content, changes description standards, or corrects information (in contrast to the content of the Archival Information Package, which is considered a permanent set of data that is not to be changed). With this separation, the AIP could be moved into offline storage, such as LTO tape, as there is minimal or no reason for the archive to change the package.

2.3 Defining Expected Files

In the process of defining package definitions, an archive should also set standards for the files they expect and create for preservation and access. If the archive specifies that all video files for preservation should be wrapped in Matroska and encoded using FFV1 and FLAC, the archive can verify that the files stored for long-term preservation or access conform to these standards by writing a file format policy. For example, a file intended for broadcast can be compared to a standard broadcast file definition, or policy,[9] with tools such as MediaConch.[10] A file successfully validated against a policy is then shown to be compliant with local standards.

9 See a sample broadcast policy at https://github.com/mediamicroservices/mm/blob/master/makebroadcast_policies.xml

10 See a description of how MediaConch policy implementation works at https://mediaarea.net/MediaConch/Documentation/HowToUse

3. Microservices for Ingest

3.1 Roadmap for Ingest

Having clear documentation on how Information Packages can be stored or arranged is essential for microservice development and the development of storage strategy. To deploy each microservice, one will need to understand:

- If the input is indeed a Package,
- If the Package is malformed or valid,
- If the Package qualifies for the microservice called,
- If the microservices has already been applied to the Package or not,
- Where to find any information needed to initialize the microservice (context or user-selected options), and
- Where to store resulting data and where to log resulting information.

Audiovisual archives can perform Ingest actions on their content either manually, or using tools and libraries developed to address audiovisual media. Some of these tools include `mediainfo` and `exiftool`, which report values for embedded metadata; `mediaconch`, which can report whether audiovisual files conform to predefined technical specifications, among other functions; and `ffmpeg`, a suite of libraries and programs that can perform a wide range of tasks on audiovisual files (including `ffprobe`, the metadata reporting component of FFmpeg). Tools that monitor fixity do not have to be specific to audiovisual media; for example, `hashdeep` is a command line utility that can recursively generate checksum reports from directories in order to produce a comprehensive manifest. The utility `bagit` could also be considered to fulfill this role.

Following are a series of example microservices for Ingest.

3.2 Validating a Submission Information Package

Submission Information Package structures should be defined by the archive, and validated according to that definition. For an example, see `Submission Information Package Expression (Example 1)` in this paper.

ValidateSubmissionPackage

As part of archival management, a validation process for packages must be created in order to support any ongoing testing, auditing, or review of the status and health of the packages created. Validating Submission Information Packages is the first step in ensuring a successful preservation workflow, giving the archivist the ability to identify malformed or incomplete submissions early.

Microservice Steps:

1. Receive Content Information (for example, a video file) and Preservation Description Information (for example, a web form entry) through a submission form, command line interaction, or other mechanism.
2. Verify that checksums are created and/or documented for all files that need them.
   a. If checksums are not created, generate checksums using `MakeChecksums` (detailed in next section).
3. If the SIP structure is validated according to local definitions, conditionally pass the SIP straight into technical metadata generation (in next section).

3.3 Generating Preservation Description Information

These microservices generate or report metadata that are expected in the Preservation Description Information section of the AIP.

MakeChecksums
This microservice generates checksums. It is most obviously applicable to Content Information, which must be monitored for fixity throughout its preservation life, but it may be applied to other directories or files. Because of this flexibility, it may be called again within other microservices and not just as a single step in the workflow.

Microservice Steps:
1. Accept an Information Package as an input (such as a SIP validated by `ValidateSubmissionPackage`).
2. Determine if the output of this microservice is already created or not in need of update, and if so stop proceeding.
3. Iterate through all Content Information files within the `./content` subdirectory (and optionally through the `./derivatives` subdirectory) and generate checksums to document all such files.

Example: Creating a sha256 checksum manifest with hashdeep using relative links
```
hashdeep -c sha256 -rl ${CONTENTS _ DIRECTORY}
```

MakeDFXML
In addition to support the writing of simple checksum manifests that list the file name and associated checksum, hashdeep can also produce DFXML files (Digital Forensics XML). The advantage of a DFXML report over a checksum manifest is that the DFXML includes file attributes, file sizes, file dates, and information about the environment in addition to the filenames and checksums. Such contextual information can be very helpful to assessing or resolving checksum mismatches at a later point.

Microservice Steps:
1. Accept an Information Package as an input (such as a SIP validated by `ValidateSubmissionPackage`).
2. Determine if the output of this microservice is already created or not in need of update, and if so stop proceeding.
3. Iterate through all Content Information files within the `./content` subdirectory (and optionally through the `./derivatives` subdirectory) and generate DFXMLs to document all such files.

Example: Creating a DFXML with md5 with hashdeep using relative links
```
hashdeep -c md5 -drl ${CONTENTS _ DIRECTORY}
```

MakeTechnicalMetadata
This microservice extracts values for technical metadata embedded within the Content Information files. It can exist as a single microservice that generates a standardized series of metadata reports—for example, multiple logs (generated by multiple tools) for all Content Information files. It can also exist as a set of smaller microservices that generate logs from only tool, which can then be selected or combined as needed based on the Content Information's file type or purpose.

Microservice Steps:

1. Accept an Information Package as an input (such as a SIP validated by `ValidateSubmissionPackage`).
2. Determine if the output of this microservice is already created or not in need of update; if so, stop proceeding.
3. Iterate through all Content Information files within the `./content` subdirectory (and optionally through the `./derivatives` subdirectory). Generate a technical metadata report to document all such files, using one or more reporting tools such as `ffmpeg`, `ffprobe`, `mediaconch`, `mediainfo`, and `exiftool`.

The technical metadata reported in this microservice may be used to make decisions within other microservices. For instance, the FFmpeg command used in `MakeWebDerivative` may be different depending on metadata such as the audio channel layout. Because FFmpeg can act as both a reporting and a processing tool, it may be more consistent to instead use `ffprobe` in the generation of technical metadata. (FFprobe and FFmpeg are derived from the same source library.) If `mediainfo` and `ffmpeg` differ on an assessment of a file, using FFprobe would avoid incorporating any processing logic into an FFmpeg command.

Example: Creating an XML with FFprobe

```
ffprobe ${INPUT_FILE} -show_format -show_streams -show_data
-show_error -show_versions -show_chapters -noprivate -of xml=q=1:x=1
> ${TECHMD_FFPROBE}
```

Example: Creating an XML with MediaInfo and MediaTrace data with MediaConch

```
mediaconch -mi -mt -fx ${INPUT_FILE} | xml fo > ${TECHMD_MEDIACONCH}
```

Example: Creating an XML to document the files and directories within a section of a package with `tree`

```
tree -DaNXs --du --timefmt "%Y-%m-%dT%H:%M:%SZ" ${PACKAGE_SECTION} >
${TREE_XML}
```

3.4 Creating and Validating an Archival Information Package

In Section II, we looked at example Information Package structures, with a SIP represented in Example 1 and an AIP represented in Example 3. In these examples, as in all successful implementations of SIPs and AIPs, all data from the SIP (Example 1) is stored within the AIP (Example 3) in a way that provides context to the role of each file. However, the AIP contains many new pieces of information that were not part of the SIP, but are added to support long-term storage and maintenance of the SIP's content. The added data infers that several workflows or programs (microservices) were applied to the content. There are many routes that may be developed or established to get from the defined SIP of Example 1 to the defined AIP of Example 3; the below microservices illustrate one route to creating and validating the AIP.

PackageContent

This microservice accepts Content Information and Preservation Description Information and organizes them into a subdirectory structure, in order to clarify the role of each piece of data within the package.

Microservice Steps:

1. Gather or request any additional data required by the archive for the initial generation of an AIP; for example, logs created in the process of running microservices.
2. Verify that the file as submitted adheres to archival preservation standards, for example checking that the audio codec, video codec, and important technical metadata values match the preservation file profile defined by the archive.
3. Create a unique Package Identifier based on (for example) the filename or media identifier from the Preservation Description Information submission. Create a directory based on that identifier.
4. Create sub-directories `./content` and `./metadata` and file the Content Information and Package Descriptive Information within those directories.

ValidateArchivalPackage

Archival Information Packages will require more extensive validation than those of Submission Information Packages, considering the amount of Preservation Description Information generated. During the growth of a digital archiving program, the tests and checks within a validation procedure may be extended and expanded to cover more and more specific checks. The list below presents some tests that such a microservice may consider.

Microservice Steps:

1. Verify that checksums are documented for all files that need them, as defined by the archive's local policy for Archival Information Packages.
2. Verify that checksums are accurate to all files by regenerating the same checksums with the same algorithm and comparing it to the ones stored; this may be accomplished by calling the `MakeChecksums` microservice and comparing its output with the previous `MakeChecksums` output.
3. Check to make sure that all anticipated microservices have properly run. This check can be accomplished by using a Bash file test operator[11] to check that all anticipated output file and directories are present, or by using exit codes to evaluate the success or failure of operations.

ValidateDisseminationPackage

It is efficient to generate derivative files for Dissemination Information Packages at the same time as Archival Information Packages; these derivatives should also be checked for conformance to local specifications.

1. Verify that the files generated by the microservices adhere to locally-set derivative specifications. For instance, check if the output of `MakeWebDerivative` uses the H.264 video codec and AAC audio codec as defined by local plicy.
2. Verify that checksums are accurate to all files by regenerating the same checksums with the same algorithm and comparing it to the ones stored; this may be accomplished by calling the `MakeChecksums` microservice and comparing its output with the previous `MakeChecksums` output.

11 See, in particular, the -f, -s, and -d tests as described here: https://www.tldp.org/LDP/abs/html/fto.html

3.5 Creating Derivatives

Other microservices generate derivatives from Content Information files to have on hand for quick access. In this case, derivatives are included within a DIP rather than an AIP, as local conventions on access files can change over time and complicate the validation of the AIP itself.

MakeWebDerivative

This microservice uses Content Information from the package to create a derivative using FFmpeg for packaging in the DIP. A derivative created during the creation of the AIP can then be stored for quicker access to the content, rather than requiring that a derivative be created at the time of a request for access. In this microservice example, a derivative will be created to deliver content on the Web.

Microservice Steps:

1. Accept an Information Package as an input (such as the output of `PackageContent`).
2. Identify what data to use as a source in the creation of a derivative. In this case, the source can be identified as the file within the `./content` subdirectory.
3. Identify if the anticipated output already exists. If so, stop proceeding.
4. Create a unique Package Identifier based on (for example) the filename or media identifier from the Preservation Description Information submission. Create a directory based on that identifier, with something to signify the package's status as a DIP.
5. Create a new subdirectory (if none already exists) in the package to store the anticipated derivative. In this example, the service directory is called `DIP/derivative/web/`.
6. Create a new subdirectory (if none already exists) in the package to store the the log associated with making the anticipated derivative, in this case `DIP/metadata/logs/`.
7. Use FFmpeg with the source file from the AIP's `./content` subdirectory to create a web-optimized derivative in the `DIP/derivative/web/` directory.
8. Generate checksums for derivatives using `MakeChecksums`.
9. Log the FFmpeg process and other aspects of the microservice's event into a log file within `DIP/derivative/web`.

Strategies for generating derivatives should negotiate the processing opportunities of the archive, access systems to be deployed, and the potential needs of the communities potentially served with this content. Currently, web-ready derivative profiles include H.264 video with AAC audio in an MP4 container (as implemented in the example), or VP9 video with Opus audio in a WebM container.

Example: Creating a Web-ready derivative

```
ffmpeg -i ${CONTENT_INPUT} -c:v libx264 -movflags faststart -pix_fmt
yuv420p -crf 18 -c:a aac -ac 2 {WEB_DERIVATIVE}.mp4
```

MakeEditDerivative

`MakeWebDerivative` is very similar to `MakeEditDerivative`, except that the process generates a derivative prepared for a different type of access, such as file transfer or editing. Whereas `MakeWebDerivative` aims to generate a derivative well-prepared for use in web streaming and access, this microservice run through the same steps but could use au-

diovisual encodings and container formats prepared for likely editors or production staff.

Microservice Steps:

1. Accept an Information Package as an input (such as the output of `PackageContent`).
2. Identify what data to use as a source in the creation of a derivative. In this case, the source can be identified as the file within the `./content` subdirectory.
3. Identify if the anticipated output already exists. If so, stop proceeding.
4. Create a unique Package Identifier based on (for example) the filename or media identifier from the Preservation Description Information submission. Create a directory based on that identifier, with something to signify the package's status as a DIP.
5. Create a new subdirectory (if none already exists) in the package to store the anticipated derivative. In this example, the service directory is called `DIP/derivative/edit/`.
6. Create a new subdirectory (if none already exists) in the package to store the the log associated with making the anticipated derivative, in this case `DIP/metadata/logs/`.
7. Use FFmpeg with the source file from the AIP's `./content` subdirectory to create a web-optimized derivative in the `DIP/derivative/edit/` directory.
8. Generate checksums for derivatives using `MakeChecksums`.
9. Log the FFmpeg process and other aspects of the microservice's event into a log file within `DIP/derivative/edit`.

Operations for generating files for editing use should consider the needs and systems of those likely to request such derivatives. Additionally, the archive should consider whether creating a derivative for editing use is necessary, as the Content Information files may be appropriate to edit as-is.

Example: Creating an edit-ready derivative

```
ffmpeg -i ${CONTENT_INPUT} -c:v prores_ks -profile:v 3 -flags +ildct+ilme
-c:a pcm_s16le {EDIT_DERIVATIVE}.mov
```

3.6 Event Logging

The OAIS reference system requires that preservation events, or the actions and outcomes of microservices, be logged and included as a component of the AIP's Preservation Description Information. Such logs provide details regarding preservation events and can be particularly helpful when the processing of AIPs requires auditing. For example, if a microservice under a particular version and/or scenario is later discovered to be flawed, having records of what versions of what microservices were run on what packages with what options can help the archive better react to bugs as they are discovered and corrected.

The Preservation Metadata: Implementation Strategies (PREMIS) metadata standard defines a structure for event logging and documents the following concepts:

PREMIS Event Element	Context in microservice logging
`eventIdentifier`	A unique identifier to refer to the event.
`eventType`	A general classification of the sort of event. See, for example, the vocabulary at http://id.loc.gov/vocabulary/preservation/eventType.html
`eventDateTime`	The point of time or range of time in which the event occurs.
`eventDetailInformation`	Information about the event, such as the name of the microservice.
`eventOutcomeInformation`	Details about the result of the microservice, including whether the event completed successfully and any logged data from the process.
`linkingAgentIdentifier`	Linking agents are agents that had an effect on the resulting AIP, and can include the name of the operator, the name and version of the involves software, and/or the name and version of the microservice.
`linkingObjectIdentifier`	Identification of the objects processed and/or created by the microservice.

Such data could be pushed by each microservice to a component of the AIP, such as a database record or log file within a package. The goal of such logging is to ensure that the contents of the AIP are independently understandable. To this end, it is important to document what parts of the AIP were part of the SIP, what parts were generated from the SIP, and by what processes the information was generated.

The principles of preservation event logging neatly dovetail with microservices that generate technical metadata or validate packages. For example, a fixity check is not just evidence of a microservice having completed successfully, but its execution and success or failure also may be stored as an event and eventOutcome within the PREMIS dictionary.[12] For such events,

Many microservices for audiovisual content use FFmpeg for actions such as derivative-creation, validation, and frame checksums. FFmpeg supports a `-report` option or an `FFREPORT` environment variable[13] which is used to log the console output of FFmpeg into a log file. Other command-line utilities can be logged by piping the standard error or standard output to a file. If events are logged to text files, then the design of the AIP should document practices for naming log files, for example ${microservice-name}_${microservice-version}_${datetime}.txt.

12 See page 258 of the current PREMIS Data Dictionary for more detail, https://www.loc.gov/standards/premis/v3/premis-3-0-final.pdf

13 See -report in http://ffmpeg.org/ffmpeg-all.html

4. Building an Ingest Script

4.1 Microservice Commonality

The microservice examples listed above demonstrate that once a packaging definition is established by an archive, the archive's microservices can share substantial commonalities. For instance, `MakeWebDerivative` and `MakeEditDerivative` are nearly identical in that they both check if the microservice is eligible to proceed, log their procedures, and file their output. Both `MakeTechnicalMetadata` and `MakeChecksums` operate similarly, except for the type of tool used (and the resulting report stored as part of the AIP). And `ValidateArchivalPackage` and `ValidateDisseminationPackage` both take the step of verifying that the files generated by the microservices adhere to locally-set file specifications—only the profiles themselves are different.

Commonalities can be exploited in development by creating common functions or code snippets that can be shared across a set of microservices. These common functions can be turned into a local microservice library. For example, in the microservice library in use at CUNY TV, common functions are defined in a script resource called `mmfunctions.` Functions are called from this central resource by each microservice as needed.[14] Defining commonalities across microservices in a single place not only saves lines of code, but means that updates to a common function only need to happen in one place. For example, if multiple microservices report information from a certain field in a database, the database pull can be defined in a single central function that is called by each microservice. If the archive eventually changes the location of this database or the name of the field, this information can be edited in that central function, allowing each microservice to be updated with a single edit.

Successful commonalities structures also depend upon shared parameters and common libraries. A set of microservices built up with consideration of local standards will naturally refer to similar settings and structures. Attention should be paid to selecting tools that are compatible with each other. For example, using MediaInfo to report metadata in `MakeTechnicalMetadata` means other tools that rely on the MediaInfo library—for example, MediaConch—will be easy to integrate and can potentially provide many more points of connection.

4.2 Linking Microservices Together

In some cases, one or more microservices may be combined into one, larger microservice, in order to create a combination of tasks that always take place together. This choice takes advantage of the wide range of tools that can perform Ingest tasks, including some that can accomplish the tasks of multiple microservices. For example, the `baginplace` function of BagIt combines the goals and results of the PackageContent and MakeChecksums microservices, as defined above, but at the same time. Combining microservices in this matter can be useful if it becomes clear that the two microservices always take place together. In other cases, retaining separation at a granular level between microservices integrates them better into workflows.

Example End-to-End Microservice Architecture

An example chain of microservices using the templates put forth in this paper might look like the following:

14 https://github.com/mediamicroservices/mm/blob/master/mmfunctions

1. Submission (creation and validation of SIP):
    a. Initial submission of Content Information and Preservation Description Information
    b. *ValidateSubmissionPackage*
        i. Call *MakeChecksums* as part of validation procedure.
        ii. Conditional continuation if package passes this microservice.
2. Ingest (creation and validation of AIP):
    a. *MakeDFXML*
    b. *MakeTechnicalMetadata*
    c. *PackageContent*
    d. *ValidateArchivalPackage*
        i. Call *MakeChecksums* as part of validation procedure.
        ii. Conditional continuation if package passes this microservice.
3. Access (creation and validation of DIP):
    a. *MakeWebDerivative*
        i. Call *MakeChecksums* as part of derivative creation procedure.
    b. *MakeEditDerivative*
        i. Call *MakeChecksums* as part of derivative creation procedure.
    c. *ValidateDisseminationPackage*
        i. Call *MakeChecksums* as part of validation procedure.
        ii. Conditional confirmation if package passes this microservice.

Packages Generated by Example End-to-End Microservice Architecture

At the end of this chain of microservices, the following packages will have been created. The italicized content indicates files or information generated by microservices. Content without italics represents information that must be supplied by the creator or archivist.

Submission Information Package

- Content Information
    - `${CONTENT} (1)`
- Preservation Description Information
    - *`[checksum file] (1)`*
    - `[web-form-record] (1)`

Archival Information Package

- Content Information
    - Content Data Object
        - `${CONTENT} (1)`
    - Representation Information
        - `[extracted technical metadata] (+)`
- Preservation Description Information
    - Provenance Information
        - *`[log files from microservice events] (+)`*
    - Context Information
        - `[database record of relationships with other database materials] (+)`
    - Reference Information
        - `[web-form-record] (1)`
        - `[database record of descriptive metadata] (1)`
        - `[database record of access metadata] (1)`

- Fixity Information
  - *[checksum file] (1)*
  - *[DFXML file] (1)*

Dissemination Information Package

- Content Information
  - */derivatives/web/${CONTENT}.mp4 (+)*
  - */derivatives/edit/${CONTENT}.mov (+)*
- Preservation Description Information
  - [database record of descriptive metadata] (1)
  - *[log files from microservice events] (+)*

4.3 Collaboration with Other Archives

These examples of microservice commonality and interconnectedness demonstrate how complex a microservice-based archival design can become. An archive might rightfully find that maintaining and updating code is too large a task to assign alongside archivists' day-to-day work. Microservice-based architecture is often more successful when employing collaboration amongst archival communities and open source approaches, wherein multiple stakeholders can contribute to a common goal and share responsibility for maintenance, as well as the advantages of additions and updates. Many examples of open, archival microservice documentation may be found at https://github.com/amiaopensource/open-workflows.

5. Advanced Considerations for Audiovisual Files and Derivatives

The example FFmpeg commands presented in `MakeWebDerivative` and `MakeEditDerivative` consist of a one-line command to convert a video file into a derivative. However, in some environments a more complex approach may be required. For instance, in broadcast environments, a single presentation may be digitized from two videotapes to two files; an original video file may include color bars and black frames that aren't needed for inclusion within some types of access files; or a video file might include multiple audio tracks that may need to be down-mixed or selectively picked for inclusion with an access file. These complications can either be employed as concatenation or trimming microservices, or—if the archive determines they should be applied consistently in tandem across the archive's materials—may still represent a single cohesive microservice.

When dealing with multiple timelines, it is important to make the relationship explicit between the original file and its edited timeline. Concatenation and trimming are a preservation event[15] and its occurrence should be noted in a log generated by the microservice, or inserted into a structured metadata standard such as METS. If employing these methods on the original object, it is also important to generate a new checksum for later package validation and fixity checks.

5.1 Temporal Selection

Often videotapes are digitized in a manner that includes video content that is supported by colorbars, informational slates, black frames, countdown, static noise, or other visual information that surrounds a program. With videotapes, generally such supporting infor-

15 For example, they may be logged as a "modification" according to the PREMIS vocabulary, with more information stored as a PREMIS Event Detail.

mation is recorded into a tape to contextualize a presentation but is not intended to be a part of the presentation itself. For instance, the digitization of a videotape may result in a 34 minute preservation file that may contain the following segments:

- 00:00 - 01:00: Color bars
- 01:00 - 01:30: Informational slate
- 01:30 - 01:40: Countdown
- 01:40 - 31:40: A presentation
- 31:40 - 32:40: Black frames
- 32:40 - 34:00: Static noise recorded until the digital recording was stopped.

For some forms of access it may be appropriate to create derivatives that represent the full timeline of the digitization; however, in other cases, that is not how the presentation is intended to be shown. FFmpeg has several options that can be used to support temporal selection so that an output derivative represents only a range of time of the input.

To fit temporal selection into the microservice scenario presented above, the `PackageContents` could request or identify the starting and the ending time of the intended presentation within the timeline of the preservation file. With the 34 minute video file depicted above, we may want an access derivative to represent the time range from 01:40 - 31:40 to mimic the intended presentation that the videotape would have been used to create. If `PackageContents` can store intended start and end times within the package, then subsequently `MakeWebDerivative` could be written to check for the presence of starting and ending times and, if so, then apply them while constructing an FFmpeg command.

Example: `MakeWebDerivative` with trimming at head and tail

```
ffmpeg -ss ${STARTING_TIME} -i ${CONTENT_INPUT} -c:v libx264 -movflags faststart -pix_fmt yuv420p -crf 18 -c:a aac -ac 2 -to ${ENDING_TIME} {WEB_DERIVATIVE}.mp4
```

5.2 Multi-file Input

The examples above presume that the Content Information of each package is a single video file. However, more complex video representations may need to be supported. For example, a camera may produce a continuous recording through the creation of multiple concurrent files, or a single presentation may be digitized from two videotapes to two files. It may be desirable to concatenate multiple preservation-level files into a single, master preservation file, or a single derivative.

To support such cases, the `PackageContents` microservice may be extended to support the ingest of multiple files into a single package. FFmpeg documents several methods to support concatenating multiple files into a single output.[16] If the technical characteristics of the package's content files are similar (same frame size, codecs, etc.) then FFmpeg's concat demuxer is recommended.[17] To use this tool, a text file should be generated listing the input files in order, with optional start and end times for each file. If this text file, named 'package_input.txt', lists:

```
file 'INPUT_FILE_1'
```

16 See "Concatenate" documentation in the FFmpeg wiki, at https://trac.ffmpeg.org/wiki/Concatenate
17 See "Concat Demuxer" in the FFmpeg documentation, at http://ffmpeg.org/ffmpeg-formats.html#concat-1

```
file 'INPUT_FILE_2'
file 'INPUT_FILE_3'
```

Then an FFmpeg command can be extended to join together multiple files by referencing that text file as the input, as in the examples below.

Example: Concatenating preservation files

In this example, the preservation files to be concatenated are already in the proper preservation file format, and need only be joined.

```
ffmpeg -f concat -i package_input.txt ${CONTENT_CONCATENATED_ID}
```

Example: Concatenating files in the production of a Web derivative

In this example, the preservation files are already in the proper preservation file format, and need only be joined.

```
ffmpeg -f concat -i package_input.txt -c:v libx264 -movflags faststart
-pix_fmt yuv420p -crf 18 -c:a aac -ac 2 {WEB_DERIVATIVE}.mp4
```

6. Conclusions

By defining a consistent and OAIS-inspired packaging structure within an archive, a microservice environment can be developed and expanded as the archive integrates services and functions. Packaging techniques should make a clear distinction between the object or objects that are the focus of preservation and any derivative files or metadata that support access and knowledge about those objects. Any microservice processing with the AIP as an input must be able to understand the AIP structure sufficiently to determine what specific files within it should be used in the microservice's work.

Many archives have been developing and sharing microservices amongst each other, as can be seen in places such as https://github.com/amiaopensource/open-workflows. Currently implementations of AIP structure do not have a standardized manner of expressing that structure, as in similar to a way in which an XML Schema can document an XML expression and be used to assess its validity. On the other hand, the development of more methods to transform AIPs from one archive's AIP implementation to another, or the development of microservices that are more easily shared amongst archives, are likely outcomes as archives integrate more collaboration and sharing in their own microservice implementations.

While monolithic systems for management of digital archives are still considered to be crucial requirements within many corners of the field, archives generally do not aspire for such systems to maintain the same level of permanence that is expected for the media and metadata of archival collection. Over time, as monoliths become obsolete or are replaced with another monolith, the migration of the media and metadata from one monolith to another often becomes a complex, expensive, and/or risky pain point in the timeline of archival management. The use of microservice architectures within audiovisual archives puts the media and metadata itself rather than the system at the center of archival management. Individual microservice components may be improved or replaced on an individual basis in a manner that facilitate a more natural evolution of an archive's gradual expanse in services, functions, and adherence to standards.

This work is published under a Creative Commons Attribution-ShareAlike 4.0 License[18]

18 https://creativecommons.org/licenses/by-sa/4.0/