**Tab 1**



The Chiron Framework *Revolutionising Knowledge Delivery with LLMs*

2024 Interns: Adib Akbari, Bill Nguyen, Biswas Simkhada, Elizabeth Pynadath, Joseph Hayek, Leo Li, Lucas Young, Sandin Jayaseker

[**Background 4**](#_ru9o31531v82)

[Problem Statement 4](#_22yat7sppm2i)

[Target Audience 4](#_tz714qudc5yw)

[Market Research 5](#_lpa0q6hkvudi)

[**Project Overview 6**](#_y5plaeajc68p)

[Project Planning 6](#_o968vkl90a5x)

[Functional and Non Functional Requirement 7](#_wn8lxnytmdge)

[**Technical Architecture 9**](#_nasoda6uqywr)

[Workflow Diagram 9](#_qb30a2kh9g5v)

[Frontend 10](#_jt8v6ina2zq4)

[React 10](#_tnfajts6oncu)

[Nutanix Birds 11](#_k0om04i21y3z)

[Websockets 12](#_tpehqcyt4poa)

[Client Queue Request Management 13](#_lflim4uc5x4m)

[Backend 13](#_wepb9zy12h1t)

[Python 13](#_u9rmf7ubrny5)

[Flask 14](#_7usyhbwjbwvz)

[Playwright 14](#_hkg2yq3eini)

[Nginx 16](#_po2owfzdc1kq)

[Design Considerations 16](#_wicj3atlumlk)

[Ollama 17](#_6o1qd634mgv2)

[Large Language Model Selection (LLM) 17](#_2b0th62uciqq)

[Model Comparison 17](#_vfsgu8nm8cb9)

[LLM Prompt Design 18](#_oud94jql8oh7)

[Illustrating Our Thought Process 18](#_tz1bfuos1ti0)

[Evolution of the Prompt Design 18](#_p7zv8ewjm4lg)

[Internal KB processing Options 19](#_gsnrf9jkkibk)

[Audio and Video Software Options 19](#_xr88joaoq80s)

[Infrastructure 22](#_r7qh9kecl41k)

[Prism Central 22](#_e9w7ehxr3k9)

[RX 22](#_8qmox186puaj)

[Eevee 22](#_o2ewk7jdbfy)

[OVA Image 22](#_v7qwgxiupvul)

[**Usage 23**](#_6ueogf2w9hev)

[Setting Up Chiron VM 23](#_3xt3kyc7oegq)

[Prerequisites / Guidelines / Recommendations 28](#_eptzvxp4zx5)

[ChironVM Management 29](#_idkfqrstmhyr)

[Frontend UI/UX 30](#_yfo34vx840zy)

[Maintenance : Update Script or Video 33](#_5c205wr6wrqp)

[**Feedback and Constraints 34**](#_48klmu17nonh)

[Feedback 34](#_2bng2f9dv5l9)

[Feedback from SREs 34](#_wzfkewmk51ps)

[Constraints and Possible Solutions 34](#_jbsw6d36725u)

[Hardware Limitations 34](#_8ook89nz2lyv)

[Concurrent Usage 34](#_maaay8nd1f6q)

[Automation of Video Creation 35](#_b8bnmeidus20)

[Sensitive Information 35](#_zd71ztkubgi1)

[Efficiently Handling Article Updates 35](#_8vmw3s7q6gqc)

[Document Length Limits 35](#_89hk8j10k0ap)

[AI Script Output 35](#_2sxehw7k3msc)

[Synthesia 35](#_bssdls4bhbib)

[Prompt 36](#_hv19ltaqrthb)

[Lack of Mobile Optimisation 36](#_qpcfouwrhbua)

[**Future Enhancements 37**](#_yaekz4a123ul)

[Testing 37](#_jczeggdf3ynm)

[LLM Finetuning 37](#_mm7z1tt3ynot)

[Video Hosting 38](#_da05bwm94rbp)

[Copilot Integration 38](#_tic1vfrm6550)

[Prompt Editing 38](#_lcgqceu1zyk2)

[**Conclusion 38**](#_chzgnza18qv)

# 

# Background

## Problem Statement

Nutanix Knowledge Bases (KBs) are designed to help users resolve specific issues with software and hardware components, empowering them to troubleshoot independently. They also serve as an essential resource for Systems Reliability Engineers (SREs) when assisting customers.

However, due to the often complex and detailed nature of KB articles, customers may feel uncertain about following the instructions themselves, sometimes leading to the creation of unnecessary support cases. Videos offer a more accessible and engaging alternative, effectively communicating the same information in a user-friendly format.

Research shows that 85% of participants reported feeling ambivalent or negative about engaging with written content [[1]](#footnote-0). Ultimately, this highlights an opportunity for case deflection rates to be improved. Video-based learning has been proven to enhance the overall learning experience, making it more engaging and effective.

The Chiron Framework has been designed as a tool to address this issue by automating the process of converting KB articles into videos. This offers several advantages to users over directly reading a knowledge base article. Not only does the video summarise lengthy content, making it more digestible, but it also provides tailored demonstrations that replicate real-world environments, helping to address issues more accurately, and enhancing users' confidence in performing troubleshooting or diagnostic steps independently. Furthermore, this format is easy to share with users in remote or restricted environments, ensuring greater accessibility.

The Chiron Framework ultimately aims to improve case deflection by supplementing existing support resources using a multimedia format.

## Target Audience

The Chiron Framework is intended to be used by SREs to enhance incident resolution efficiency by clearly and effectively conveying solutions to customers' break/fix issues. It caters to customers who may prefer video-based support over traditional KB articles.

Beyond its core functionality in customer support, the framework’s adaptability makes it an invaluable tool for internal training and knowledge sharing. For example, it can be utilised to produce engaging multimedia content, such as instructional videos for Confluence and portal pages, that enhances the training of SREs and interns, extending its impact across the organization.

## Market Research

Prof Jim Inc.[[2]](#footnote-1) is an AI-powered platform that transforms educational content into interactive presentations, enhancing engagement and accessibility. Founded by Deepak Sekar, Pranav Mehta, and Maria Walley, the company enables educators and content creators to quickly generate AI-driven lessons, assessments, and avatars that deliver course material. Their goal is to streamline digital education by automating content creation and offering interactive learning experiences.

While Prof Jim focuses on AI-generated educational content for general learning, the Chiron Framework is specifically designed to convert KB articles into scripts, allowing users to record real-world, hands-on demonstrations in a Nutanix lab environment. Unlike Prof Jim, which relies on AI-generated avatars to present lessons, the Chiron Framework empowers users to create live, technical walkthroughs, demonstrating exactly how to execute commands and procedures. This hands-on approach ensures accuracy and practical application, making the Chiron Framework a far more precise and effective tool for customers and SREs who need step-by-step guidance in Nutanix environments.

Other providers, such as Pitory and LTX Studios, also transform scripts into videos. However, these platforms generate content by pairing scripts with loosely related visuals, which is significantly different from the Chiron Framework’s approach.

In conclusion, the Chiron Framework enables users to create high-quality, hands-on demonstration videos, making it one of the top solutions in the market for producing accurate and professional technical content.

# 

# Project Overview

## Project Planning

| **Week 1** | * Group discussions on how to implement the problem proposed by the management team * Assign roles:   + Documentation and Presentation   + Frontend   + Backend   + Nutanix Tools   + Video Creation |
| --- | --- |
| **Week 2** | * Backend LLM (Large Language Model) VM prototype development * Ideate on the themes for script generation website * Investigate KB articles to use as test cases * Start presentation and whitepaper |
| **Week 3** | * Frontend UI interface prototype developed * Narrow-down a selection of KB articles to use as test cases * Keep fine tuning LLM VM prototype * Update presentation and whitepaper * Incorporate editing in the script generation website |
| **Week 4** | * Synthesia AI integration completed * Add more frontend UI features * Update presentation and whitepaper * Get feedback from management and mentors * Highlight the commands in the script output * Get approval for tools used in the framework |
| **Week 5** | * Presentation |

## 

## Functional and Non Functional Requirement

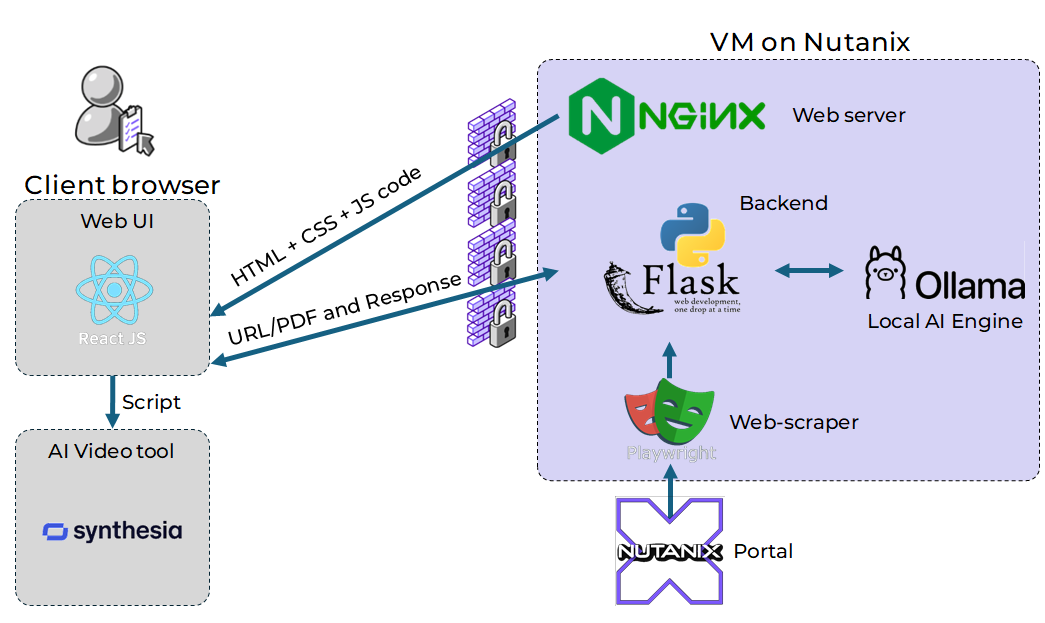
The framework currently has the following functional and non functional requirements.

| **Functional** | **Non Functional** |
| --- | --- |
| * When a user inputs a KB article URL or PDF into the input box, a video script is auto-generated. * Previous scripts are available for review via a sidebar. * Generated scripts can be edited directly on the UI using an edit icon. * Scripts can be copied with a copy icon on the frontend. * For URL-based scripts, a button opens the original article in a new tab. * Videos are easily exportable by download. * Video creation includes integrated screen recording and editing tools. * Script updates are supported through regeneration, re-recording, or annotations. * Users can return to editing slides even after a video is generated. * Tools are provided to hide or blur sensitive information (e.g. IP addresses). * Videos clearly demonstrate command usage. * The system is designed for ease of use with a 10‑minute learning curve and on-demand help via a help icon. * The script generation site links directly to the video creation software and highlights commands for easier proofreading. * An undo function allows users to revert to previous script versions. | * Generated videos are of high quality. * The AI video narrator exhibits natural movement and clear voice quality with a wide range of voice options. * The AI narrator’s size and on-screen position are adjustable. * The complete process (script generation plus video creation) takes approximately 1 hour for an average KB. * Peer evaluation is integrated to enhance accuracy. * Videos of any length can be created, subject to licensing requirements. * The framework is operating system agnostic. * Deployment is straightforward, taking around 30 minutes. * The system operates entirely on in-house tools and AI models without reliance on third-party hosted services. * The UI must provide real-time progress feedback during script generation and video creation, for example, via websockets. * Robust error handling is essential, ensuring immediate and clear notifications for any issues during processing or queue management. * The framework should be designed to scale with additional resources (e.g., GPU integration) to handle higher workloads. |

# Technical Architecture

This section outlines the architectural components of Chiron, how the various components of our system interact with each other, including the research process conducted to identify the optimal tool for each stage of the workflow.

## Workflow Diagram



*Diagram of the software tools utilised within the Chiron framework.*

The initial step of the framework involves the user visiting the user interface, designed using **React**, MateriaUI and developed in JavaScript, which is served from a **Nginx** web server running on a Nutanix VM. On the UI, the user enters a KB article URL (or uploads a PDF). This data is sent to our **Flask** backend, coded in Python, which uses **Playwright** to scrape the KB article page.

The contents from the web scraping (or the uploaded PDF), is then sent to **Ollama**, which runs the Large Language Model and generates the video script, then sent back to the user interface.

The user then takes this generated script, and moves to an external tool **Synthesia** for the video creation process.

## 

## Frontend

### React

The frontend JS framework utilised for this project is React 19 with Material-UI (MUI). This facilitates interactivity, and additionally serves to greatly accelerate the development process due to its simple to use API and widespread adoption. The resultant availability of a diverse range of existing tools and modules, reduces the amount of work required to get our software up and running with the desired functionalities.

| **Feature** | **Implementation** | **Benefit** |
| --- | --- | --- |
| Component Architecture | Modular design | Easy maintenance and updates |
| UI Framework | Material-UI | Production-ready components |
| State Management | React Hooks + Context | Simplified data flow |
| Error Handling | Modal system | Enhanced user feedback |
| Routing | React Router v7 | Seamless navigation |

**Interactive Elements**

* Drag-and-drop file upload
* Real-time script editing
* Progress indicators
* Modal-based error feedback

**User Experience**

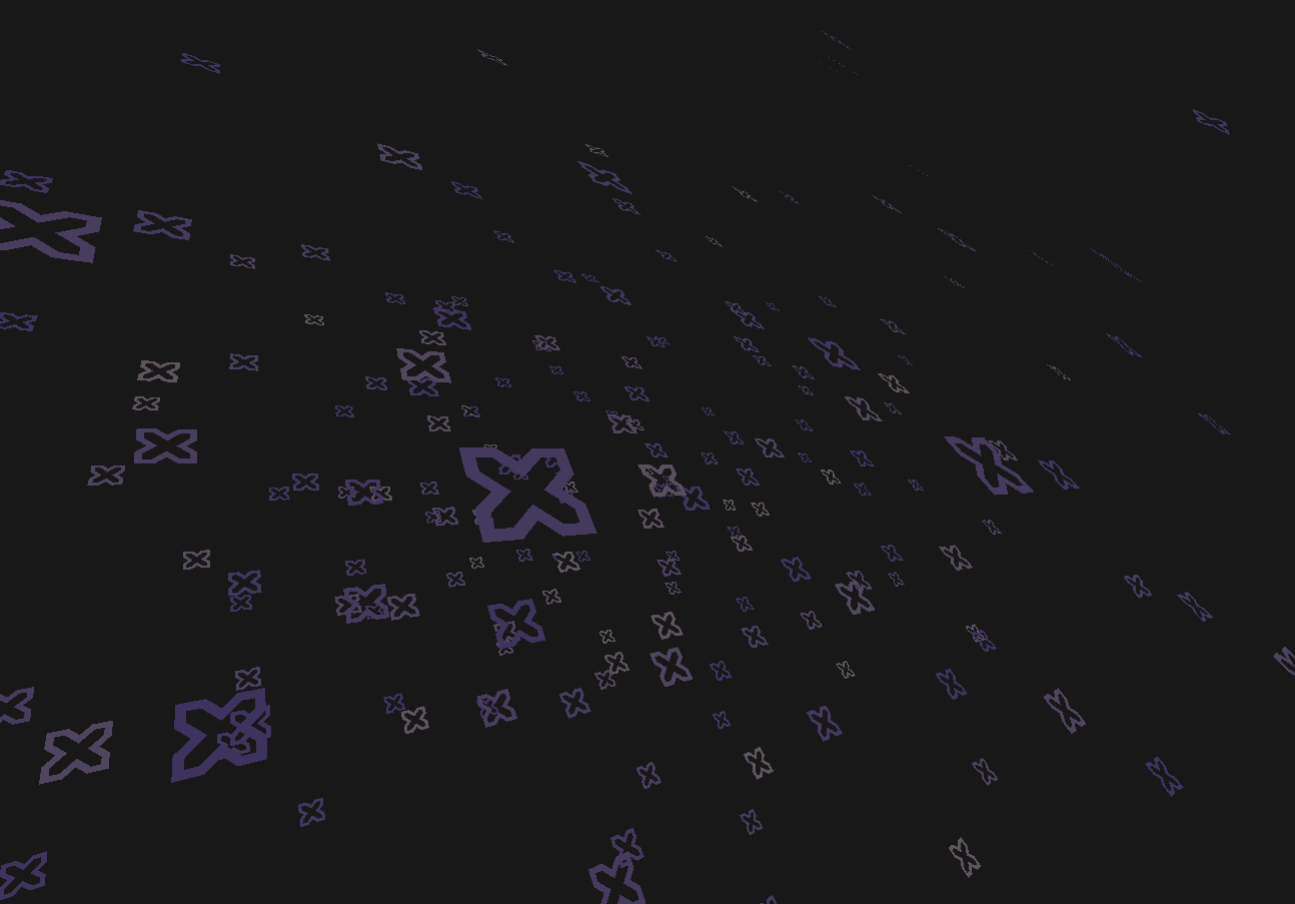
* Responsive design for all desktop screen sizes
* Intuitive navigation
* Clear error messaging
* Live-streaming of script as it’s being generated in the backend
* Background animations for visual appeal

**Enterprise Integration**

* Compatible with all modern browsers.

This implementation creates a robust, maintainable frontend that effectively serves our enterprise users while providing a smooth experience for document processing and video script generation.

### Nutanix Birds



*In the pause, they glide,*

*Nexus fly among the sky,*

*Interactive light.*

This interactive background is named **Nutanix Birds**, after the bird-like movements that the shapes exhibit. Birds was inspired by [VantaJS](https://www.vantajs.com/), however, this tool did not allow us to utilize custom 3D models. Hence, we found the underlying [ThreeJS](https://threejs.org/) code for this animation, then we modified this code to insert a custom 3D model of the Nutanix Nexus.

To achieve this, the Nexus was modelled in [Blender](https://www.blender.org/), then a script was written to manually export the triangles that form this model. This allowed the model to be added to our code.

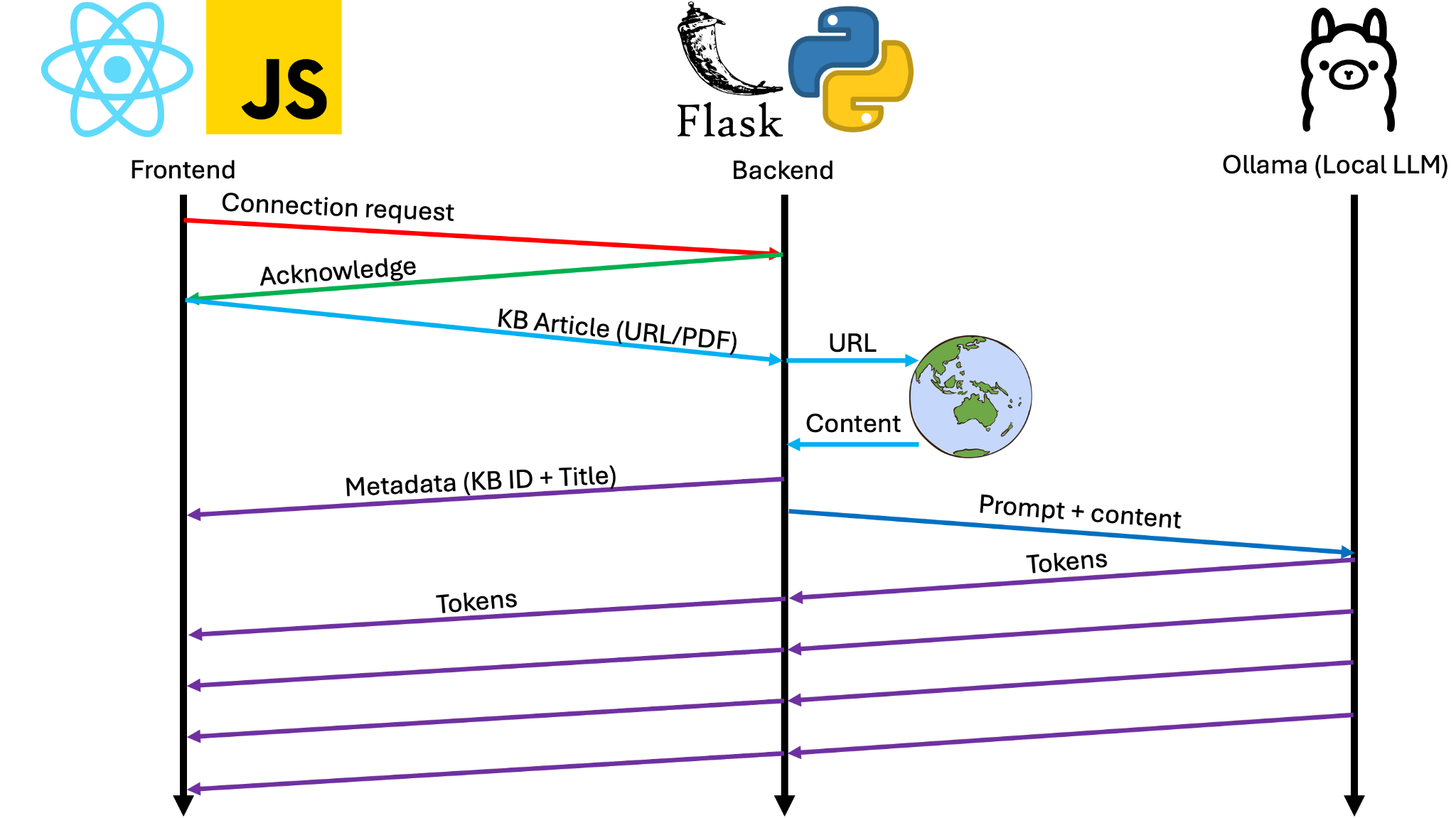
The purpose of Nutanix Birds was to add some interactivity to our frontend, specifically to allow the user to fill the time while waiting for LLM generation to complete. Additionally, we believe it helps in asserting the Nutanix brand identity.

## 

## 

## Websockets

We used the websockets protocol to facilitate omnidirectional communication between the user interface running on the browser (frontend) and the LLM Model running on a Nutanix VM (backend) as the frontend needs to be able send request payloads such as URL strings or PDF files and receives a stream of generated LLM tokens back in real time from the backend.



*Diagram of how the Frontend & Backend interact with each other*

When the frontend boots up, it sends a connection request to the backend running on Nutanix. Once the handshake process is finalised by the server, the user can input an URL or upload a PDF file and it would be sent through the websocket. Note that for the server to finalise a handshake, all AI models must be downloaded, otherwise, clients would have to wait for the server to finish downloading the models. Progress is livestreamed to all clients.

On receipt of the request at the backend, the data of the webpage or PDF file will be scraped to extract the metadata and content. The metadata will be sent back to the frontend for storage and the content will be appended to a system prompt and sent to Ollama, an engine that allows us to run LLMs locally.

As the LLM generates the script, in the form of a series of tokens, these tokens are streamed back to the frontend live via the websocket in real time for display to the user, showing the result as its being generated.

We could’ve accomplished the same basic functionality using simple REST APIs where the frontend would send the request to a specific backend URL then wait for a complete response. We implemented this approach in our initial prototype but quickly realised that it provides poor user experience as there is just a loading screen until the LLM finishes generating the script. However with websockets, it allows us to stream back tokens live and show it to the user, allowing them to see progress and experience less anxious waiting.

### Client Queue Request Management

Managing client requests efficiently is critical to ensuring the stability and performance of our backend AI processing system. Instead of allowing multiple users to concurrently interact with the LLM, which would overwhelm available computational resources, we implemented a queue-based system. This decision was driven by the following key considerations:

**Resource Constraints**

* Running large language models locally (via Ollama) is computationally expensive, particularly when inference is handled on CPU-based infrastructure.
* Multiple concurrent requests would degrade response times and may cause system instability.

**Performance Optimisation**

* A queue ensures that each request is processed sequentially, preventing CPU contention and excessive memory usage, which can lead to crashes or inefficient processing.
* Script generation inference speeds tend to get worse exponentially as more concurrent users are attempting to generate.

**User Experience**

* By controlling the flow of requests, we can provide predictable response times and a smoother user experience.
* Instead of overloading the system with parallel requests, users can be notified of their position in the queue.

**Request Management**  
When a user sends a request, their session ID (sid) is appended to the queue, ensuring an orderly processing system. Upon joining, users receive a WebSocket event notifying them of their position in the queue, allowing them to track their status in real time. The system processes requests sequentially, meaning only the first request in the queue is handled while others wait their turn. If a user disconnects before their request is processed, they are automatically removed from the queue, and all remaining clients receive updated queue positions to reflect the change. To maintain fairness and prevent overload, users can only proceed when it is their turn; any premature requests trigger an error message, ensuring controlled request execution. Additionally, every queue update triggers a broadcast to all connected clients, synchronizing queue positions in real time and maintaining a seamless user experience.

## Backend

### Python

Python was chosen as the primary backend language for the Chiron Framework due to its simplicity, robust ecosystem, and extensive support for automation, data processing, and machine learning. Compared to other backend languages such as Node.js, Java, and Golang, Python offers distinct advantages that align well with the framework's goals.

One of the key reasons for selecting Python is its simplicity and readability. Python's clean and expressive syntax allows developers to write and maintain code more efficiently, which is particularly beneficial for AI-driven projects like Chiron, where complex workflows need to be implemented with minimal overhead.

Compared to Node.js, Python is more suited for CPU-intensive tasks, such as content extraction and AI-driven script generation. While Node.js excels in handling asynchronous I/O and real-time applications, it lacks Python’s depth in computing and automation capabilities. Java, on the other hand, is a strong contender in terms of text processing and scalability, but its verbose syntax and slower development cycle make it less ideal for a project that demands rapid iteration and prototyping. Golang is known for its high performance and concurrency handling, but its ecosystem lacks the specialized AI and web automation libraries that Python offers, such as the Playwright and Websocket features that we used.

Python’s ability to facilitate rapid prototyping, coupled with its strong ecosystem for AI, automation, and web scraping, makes it the most suitable choice for the Chiron Framework. Its ease of use, combined with the diversity of its libraries, ensures efficient development and seamless integration with AI-driven content generation, making it the backbone of Chiron’s backend infrastructure.

### Flask

We used Flask as the core backend framework for handling web requests and real-time web interactions within the Chiron Framework. Flask's lightweight architecture made it an ideal choice for efficiently managing WebSocket communications, enabling seamless integration with our AI-powered content processing pipeline.

For real-time interaction, we used Flask-SocketIO to establish a WebSocket connection between the frontend and backend. This allows bidirectional communication where the frontend can send requests dynamically, and the backend can stream back AI-generated text tokens in real-time. The WebSocket-based approach significantly improves user experience by eliminating long blocking request times and providing progressive feedback as the LLM generates content.

Additionally, Flask was chosen over a similar framework called Django for its lightweight advantage over Django’s rigid structure, which better suits Chiron’s flexible and AI development needs. Flask’s flexibility allowed us to build only necessary components, reducing overhead. Its native WebSocket support with Flask-SocketIO simplifies real-time streaming, whereas Django Channels adds complexity. Flask also offers more efficient API design and lower resource consumption, making it the optimal choice for handling AI-driven content generation.

Flask’s flexibility allowed us to integrate robust security mechanisms such as CORS (Cross-Origin Resource Sharing) handling and graceful shutdown procedures to ensure smooth execution even under high workloads. Flask's ability to handle multiple concurrent user requests efficiently while maintaining low resource consumption made it the ideal backend framework for the Chiron Framework’s AI-powered knowledge extraction and video script generation system.

### Playwright

Playwright is an essential component of the Chiron Framework, automating the process of extracting content in knowledge base articles for input into locally hosted LLMs (Large Language Models). Its reliability, speed, and advanced web driver feature set make it the ideal choice for automating web interactions, reducing maintenance overhead, and enhancing the overall efficiency of the framework.

The decision to use Playwright over other web scraping and automation tools, such as Selenium or BeautifulSoup, was driven by its superior performance, reliability, and feature set. Unlike traditional web scraping tools that struggle with dynamic JavaScript-rendered content, Playwright excels in handling modern web applications, ensuring that even dynamically loaded elements are accurately captured using a natively installed Web Driver on the backend.

Playwright has its own built-in support for multiple browsers without requiring separate drivers, making it significantly easier to set up and maintain. This was one of the key drivers for our decision to use this library over Selenium, as Selenium required external WebDrivers that were incompatible with Rocky Linux based VMs . Furthermore, Playwright's auto-waiting mechanism ensures that elements are interactable before performing actions, reducing the likelihood of failed or incomplete extractions. Playwright also supports parallel execution, making large-scale scraping operations significantly faster compared to Selenium, which often suffers from their WebDriver overhead.

Moreover, Playwright simplifies installation and setup by managing dependencies internally. A single command installs everything needed, whereas Selenium requires manually downloading and configuring browser drivers. Playwright also offers built-in debugging tools such as screenshots, video recordings, and tracing, which make troubleshooting issues more intuitive for developers. These capabilities collectively enhance the Chiron Framework’s ability to automate knowledge extraction and processing at scale.

**Comparison of Playwright, Selenium, and BeautifulSoup**

| **Criteria** | **Playwright** | **Selenium** | **BeautifulSoup** |
| --- | --- | --- | --- |
| **Core features** | Full browser automation, supports multiple browsers, handles dynamic content, network interception, and debugging tools | Browser automation with WebDriver, supports multiple browsers, but slower and requires more setup | Parses static HTML without browser interaction |
| **Ease of Use** | Simple API with built-in smart waiting and automatic handling of web elements | Requires WebDriver setup, more boilerplate code needed | Very simple, works with HTML/XML but limited to static content |
| **Learning Curve** | Moderate, but well-documented with modern API design | Steeper learning curve due to WebDriver configuration and manual waits | Very easy, primarily used for parsing simple HTML data |
| **Integration with other tools** | Easily integrates with modern testing frameworks and APIs, supports CI/CD | Integrates with most testing tools but requires more setup | Works well with requests, pandas, and other data-processing tools but lacks dynamic interaction |
| **Free vs Paid Features** | Fully open-source, no paid features | Fully open-source, but may require third-party tools for advanced functionality | Fully open-source with no additional costs |
| **Performance and Speed** | Fast execution with parallel processing and optimized architecture | Slower due to WebDriver overhead and need for explicit waits | Fastest for static pages, but cannot handle JavaScript-heavy websites |
| **Scalability** | Supports large-scale scraping with multi-threading and headless execution | Can be scaled but requires additional infrastructure and parallel execution setup | Not ideal for large-scale dynamic scraping due to lack of JavaScript execution |
| **Support and Documentation** | Well-documented with active community and Microsoft backing | Extensive documentation and large user base | Simple documentation with wide community support but lacks advanced features |

## Nginx

The project's web architecture leverages Nginx on our Nutanix-hosted VM to serve our React frontend static assets from the local filesystem (frontend /build directory). The Flask backend runs as a systemd service on port 4242, handling the LLM interactions. The frontend makes direct API calls to the backend service via this port. This architecture requires specific SELinux configurations to maintain security while enabling the necessary communication paths. The httpd\_sys\_content\_t context is applied to the frontend build directory, while the httpd\_can\_network\_connect boolean enables required network connectivity. The httpd\_enable\_homedirs boolean allows Nginx to properly serve content from home directories.

A key advantage of this configuration is its simplicity for end users - they access the frontend through the VM's IP address in their browser on port 80, while the application manages backend communication through port 4242 for real-time LLM interactions.

* Nginx eliminates the need to hardcode IP addresses of frontend/backend and allows easy portability of the application.

### Design Considerations

| **Aspect** | **Implementation** | **Benefit** |
| --- | --- | --- |
| Frontend Serving | Static file serving from build directory | Optimized delivery of React assets |
| Backend Communication | Dedicated port (4242) for Flask service | Real-time LLM responses |
| Security | SELinux contexts and network permissions | Enterprise-grade security compliance |
| Deployment | Single VM deployment | Simplified maintenance and updates |
| User Access | Single entry point (port 80) | Streamlined user experience |
| Network Integration | Internal network deployment | Enhanced security and reduced latency |

## Ollama

[**Ollama**](https://github.com/ollama/ollama) is an open-source program that allows us to easily download and run local LLM models. This software was chosen for this purpose, as this software has a large support base and community due to its widespread use.

Ollama runs as a service on the backend VM, whenever a request is received and processed by the backend process into a prompt, the prompt is sent from the backend via Python binding of the Ollama API to the Ollama service for inference.

### Large Language Model Selection (LLM)

We chose to use a self-hosted LLM to ensure security requirements. This allows us to safely use this tool on internal KB articles, without having to send these articles to a third party service provider such as OpenAI or Grok.

With many free and self-hostable LLMs available, a selection process was undertaken to determine the most optimal model for our framework.

The key metric we attempted to optimise for was generation quality to performance. Quality refers to the adherence of the generated output to both the KB article’s contents, and to instructions of the prepended prompt. The performance aspect refers to the inference speed, specifically in tokens/second.

During development, we were limited to running model inference on the CPU, as GPU availability was inadequate (specifically, we did not have access to GPUs that had adequate VRAM). Typically, higher the performance of a model, higher the RAM/VRAM requirements. Hence, to keep our options open in terms of model size, we decided to run inference on the CPU, as we had access to servers with very large amounts of RAM.

### Model Comparison

The following are the models we have tested, alongside some additional findings from our experimentation. They are ranked in terms of suitability for the Chiron Framework.

| Model | Memory Requirements | Findings |
| --- | --- | --- |
| Llama 3.3 70B | 141GB | High-quality output but very slow inference on CPU. Thus: Best model for **production** version, where generation time is less of a concern. |
| Llama 3.1 8B | 16GB | Lower quality than 90B but faster: optimal for **prototyping** due to fast generation speed. |

From the above findings, we recommend using Llama 90B for production use, where output generation speed wouldn’t be a concern. However, for situations where faster generation speed is required (such as prototyping), Llama 8B is recommended.

**Alternatives to a locally hosted LLM**

A software procurement request has been submitted for the use of LLaMa. If the software, or the self-hosting of LLMs are prohibited, alternatives have been considered:

* Using a pre-existing tool in use at Nutanix such as SupportGPT, Glean.
* OpenAI API or equivalent.
* SRE could summarise the article.

## LLM Prompt Design

### Illustrating Our Thought Process

When designing prompts for the Chiron Framework’s AI-powered video script generation, our primary goal was to ensure that the model produced structured, high-quality, and consistent outputs. The prompt evolved across six iterations, each refining aspects of clarity, specificity, and control over the generated text. The key elements we focused on while refining our prompts included:

**Constraint Output Style Across Different LLMs**

* Ensuring the generated script maintained a consistent style and structure, regardless of the model used by eliminating variance in verbosity, tone, or technical detail.

**Clarity in Output Expectations**

* Clearly instructing the LLM **not** to generate additional commentary, preambles, or non-script elements (e.g., "[Intro music plays]").
* Enforcing that the response format was directly usable for Text-to-Speech (TTS) conversion.

**Incorporating TTS Considerations**

* Adjusting pacing for AI voiceovers by **introducing structured pauses (...)** to improve readability and natural flow.
* Directing the model to choose the most common scenario from the KB article when multiple options were presented.
* Avoiding ambiguous phrasing that could lead to mispronunciations in generated speech.

**Enhancing Technical Readability**

* By introducing Markdown formatting (inline code, bash blocks), we made scripts easier to read and proofread.
* Highlighting important commands reduced the likelihood of errors when executing technical instructions.

### Evolution of the Prompt Design

The evolution of our prompt design began with V1 to V3, focusing on foundational refinements. V1 established the basic instruction structure, providing clear guidance on converting KB articles into video scripts.

v1

The following text is a Knowledge Base article for a Nutanix product. This article is to be converted to a video to assist users of the product run the steps outlined in the article themselves. Your task is to generate a script for this video, based on the article contents.

Where multiple options or scenarios are presented in the article, choose the most common path to be presented in the video. Your script will be converted to speech using TTS, and someone will manually generate the visuals based on your script, you should account for this in the pacing of the script.

The KB article is as follows:



V2 introduced constraints to eliminate unnecessary annotations while refining the script’s pacing. V3 further restricted the output to only the script content, ensuring no introductory or concluding statements were included.

v3

The following text is a Knowledge Base article for a Nutanix product. This article is to be converted to a video to assist users of the product run the steps outlined in the article themselves. Your task is to generate a script for this video, based on the article contents.

Where multiple options or scenarios are presented in the article, choose the most common path to be presented in the video.

Your script will be converted to speech using TTS, and someone will manually generate the visuals based on your script, you should account for this in the pacing of the script. For pauses, add “...” on a new line, however, do not include any additional annotation or direction (i.e. do NOT include annotations such as [Intro music plays]), just the script. Do not include any initial or final comments that do not form part of the script (i.e. do NOT include something like “here is your script”).

The KB article is as follows:



Moving into V4 and V5, the structure was formalized by standardizing the INSTRUCTION heading for clarity, reinforcing strict output requirements, such as removing preambles, and explicitly stating that the script would be used as-is for AI-generated speech.

Finally, V6 introduced Markdown-style formatting for commands and technical steps to enhance readability and enforce script structure. Additionally, ellipses ("...") were reinforced for pacing adjustments, ensuring better synchronization with AI-generated speech.

v6

### INSTRUCTION:

The following text is a Knowledge Base or Confluence article for a Nutanix product. This article is to be converted to a video to assist users of the product run the steps outlined in the article themselves. Your task is to generate a script for this video, based on the article contents.

Where multiple options or scenarios are presented in the article, choose the most common path to be presented in the video.

Your script will be converted to speech using TTS, and someone will manually generate the visuals based on your script, you should account for this in the pacing of the script. For pauses, add “...” on a new line, however, do not include any additional annotation or direction (i.e. do NOT include annotations such as [Intro music plays]), just the script. Do not include any preamble, only generate the script that is to be fed directly to an AI TTS (i.e. do NOT include something like “here is your script”).

### REQUIREMENTS:

- Format commands, code snippets, or terminal inputs using Markdown-style formatting (e.g., wrap inline commands in single backtick: `command` and command code blocks in triple backticks: ```bash command```).

- Use ellipses ("...") on a new line to indicate pauses for better pacing.

- Do not include preambles, explanations, or annotations (e.g., [Intro music plays]).

- The script should be written as it will be read by AI TTS, without extra instructions.

### NO PREAMBLE. SCRIPT ONLY

### KB ARTICLE CONTENT:



## Internal KB processing Options

Our application supports PDF and document uploads to process private Knowledge Base (KB) articles and Confluence documents that are not publicly accessible via a direct URL. This feature was implemented to overcome authentication barriers that restricted the ability to scrape article content using Playwright.

When dealing with KB articles and Confluence documents stored within Nutanix’s internal systems, accessing them via URL requires authentication through the Nutanix Portal, which enforces OAuth 2.0 and OpenID Connect authentication via Okta. Acquiring the necessary credentials for automated access would involve a complex approval chain, making it impractical for our use case. Additionally, integrating Playwright with authentication workflows is tedious, as automating credential input using its built-in WebDriver cause problems such as handling session timeouts, multi-factor authentication (MFA), and CAPTCHAs, which disrupts and add excessive inefficiencies to the video automation process.

To bypass these authentication challenges, we used the export-to-PDF functionality available within Nutanix’s KB and Confluence platforms. Users can manually export secured documents as PDFs and upload them directly into our application, which then processes the file locally. This approach eliminates the need for Playwright-based authentication and significantly streamlines the content extraction pipeline.

**Alternatives to Uploading Internal Article PDFs**

If PDF uploads or direct document processing are restricted, alternative solutions have been considered:

* If available, direct API integration could pull article text without requiring manual exports.
* Instead of manual logins, use SSO authentication cookies or API tokens to automate KB retrieval, or retrieve Okta authentication tokens and pass them into a script that extracts data securely.
* If users prefer taking screenshots of KBs, an OCR engine can convert images into text. Adobe Sensei or AWS Textract are some examples of enterprise-level OCR services for structured text extraction.

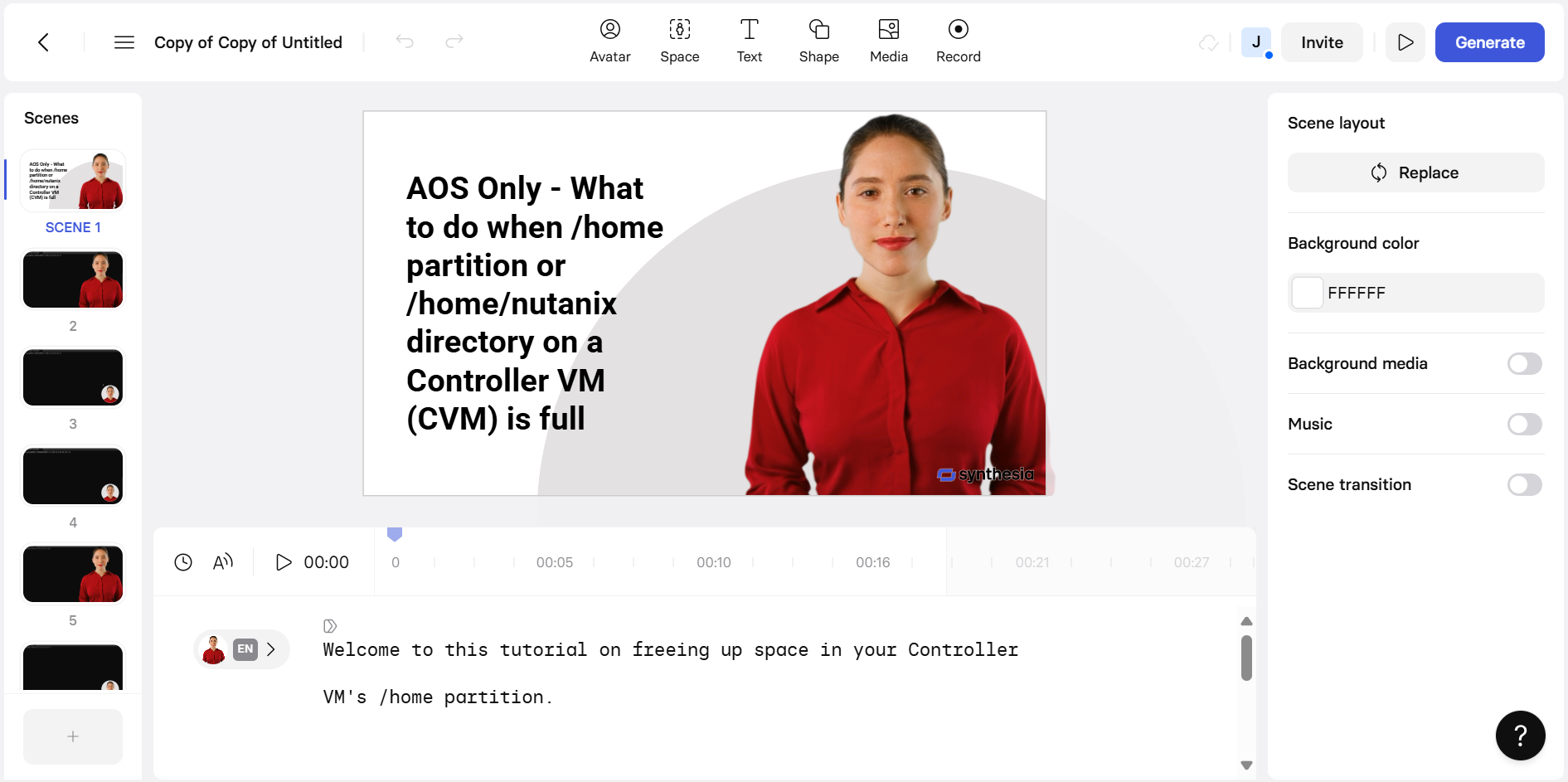
## Audio and Video Software Options

Two software options were investigated for the AI audio and video production; Synthesia and ElevenLabs. Synthesia is currently used in training material and is familiar to the company and trainers. Moreover, its facial animation technology is a standout and engaging feature. Research was conducted on alternatives to see if there was a better fit. ElevenLabs is an AI audio company that offers a competitive product to Synthesia in the AI audio department, however it does not provide a screen recording option. On top of this, it is not currently in use at the company, and will need to go through its own vetting and learning process. Therefore, it was decided that Synthesia would be the best AI audio and video recording option to move forward with.

It would be ideal if the entire process was automated, however auto-generating the video is currently impossible; an SRE needs to record the steps manually from their screen.

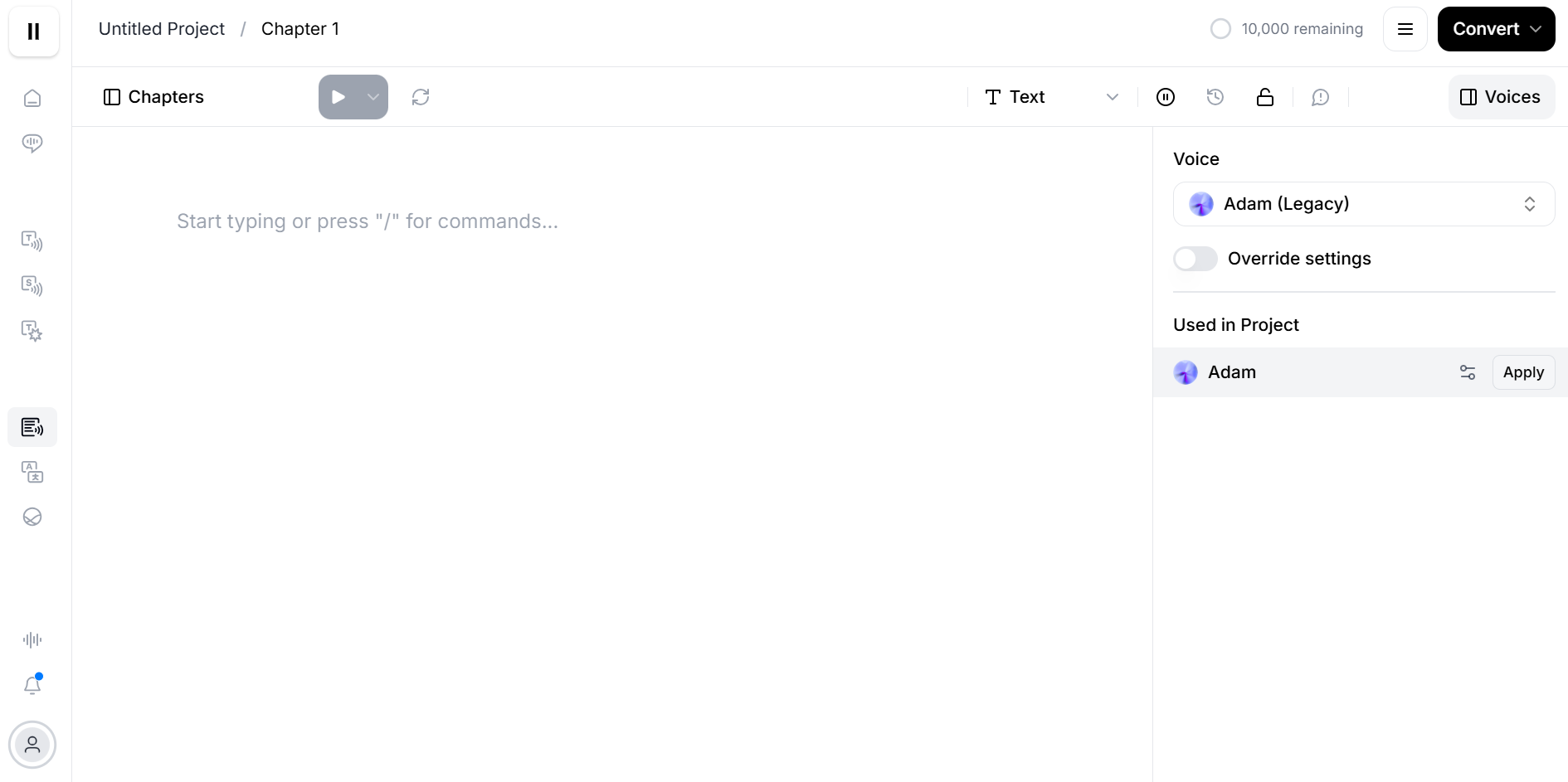
|  | **Synthesia** | **ElevenLabs** |
| --- | --- | --- |
| **Pros** | * Already in use for training materials and familiar to company trainers. * Offers facial animation technology, making content more engaging. | * Provides a wider range of high-quality AI voice options. |
| **Cons** | * AI-generated voices may not be as natural as ElevenLabs. * Requires purchasing licenses for full functionality. | * Lacks a screen recording feature. * Not currently used within the company, requiring additional vetting and onboarding. |

### Synthesia



Synthesia offers multiple voice options for the Text to Speech functionality. We experimented with several of the options available on the free plan, however, all of the options we tested sounded worse (less human sounding). As such, we recommend using the base model (provided that the free plan is used in production).

### ElevenLabs



## Infrastructure

### Prism Central

Prism Central is used to manage multiple clusters and export OVAs. Chiron VM contains Ollama AI service, backend and frontend servers, website and KB scraping code, and other utilities for easy automated deployment. Since OVAs can’t be exported from Prism Element, we deployed a single node Prism Central VM on a reserved cluster to export the Chiron VM as an OVA.

Setting up Prism Central was only required for exporting to an OVA. We had some issues setting it up due to a lack of familiarity, such as setting the correct IP address needed for connecting Prism Element.

### RX

The internal Nutanix Reservation System (RX) is used by Staff to book Nutanix Clusters which can be used to run Chrion VM.

### Eevee

Eevee is the Nutanix Internal IP Address Management system used to reserve pools of IP addresses. It is required to set up subnets on RX clusters which are used by VMs to connect to the internet, a prerequisite for Chiron VM automatic deployment.

### OVA Image

The OVA Image for Chiron VM is stored on the Intern Project Google Drive (Nutanix Google account is required to access it). [Chiron Framework Repo -> Chiron Framework Files.](https://drive.google.com/drive/folders/1jyHSdvQpHuSy24j-QcmagO9OBbEksmap?usp=drive_link)

# Usage

## Setting Up Chiron VM



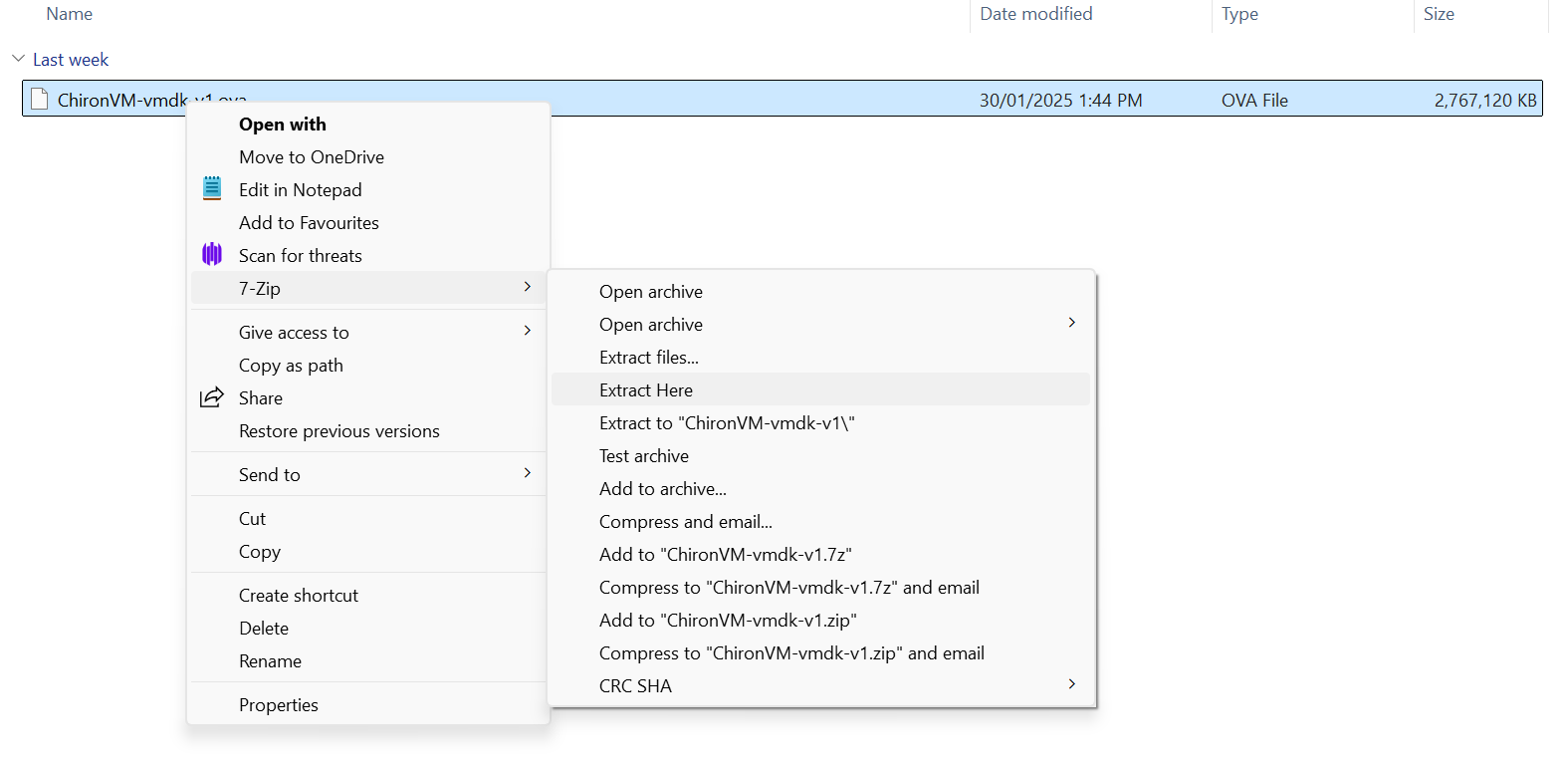
Setting up the Chiron VM is a simple process that involves creating a Virtual Machine on a Nutanix cluster and cloning from an OVA disk image using the image service.

**Step 1** - Download the VM OVA from the [Chiron Frameworks Files folder.](https://drive.google.com/drive/folders/1jyHSdvQpHuSy24j-QcmagO9OBbEksmap?usp=drive_link)

**Step 2** - Extract the OVA from the downloaded google drive zip.

**Step 3** - Extract the files from the OVA using 7-Zip or tar.

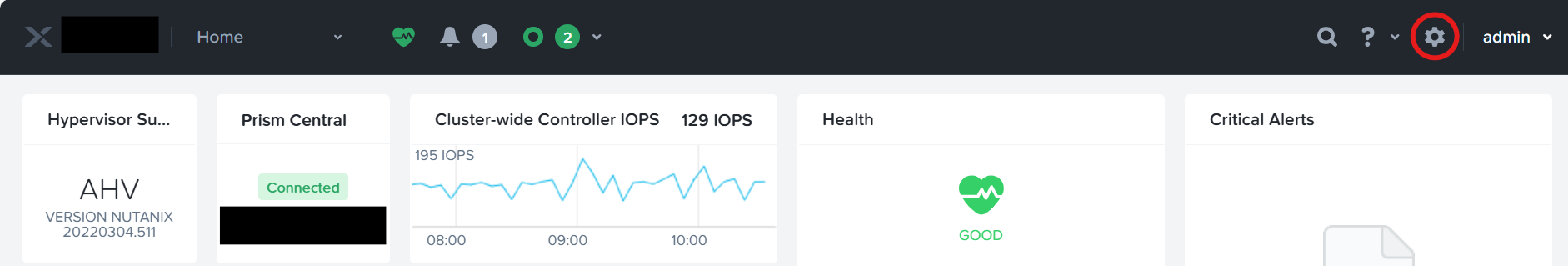
* 7-Zip: Right click on the OVA file > Select Extract Here or extract to a directory.

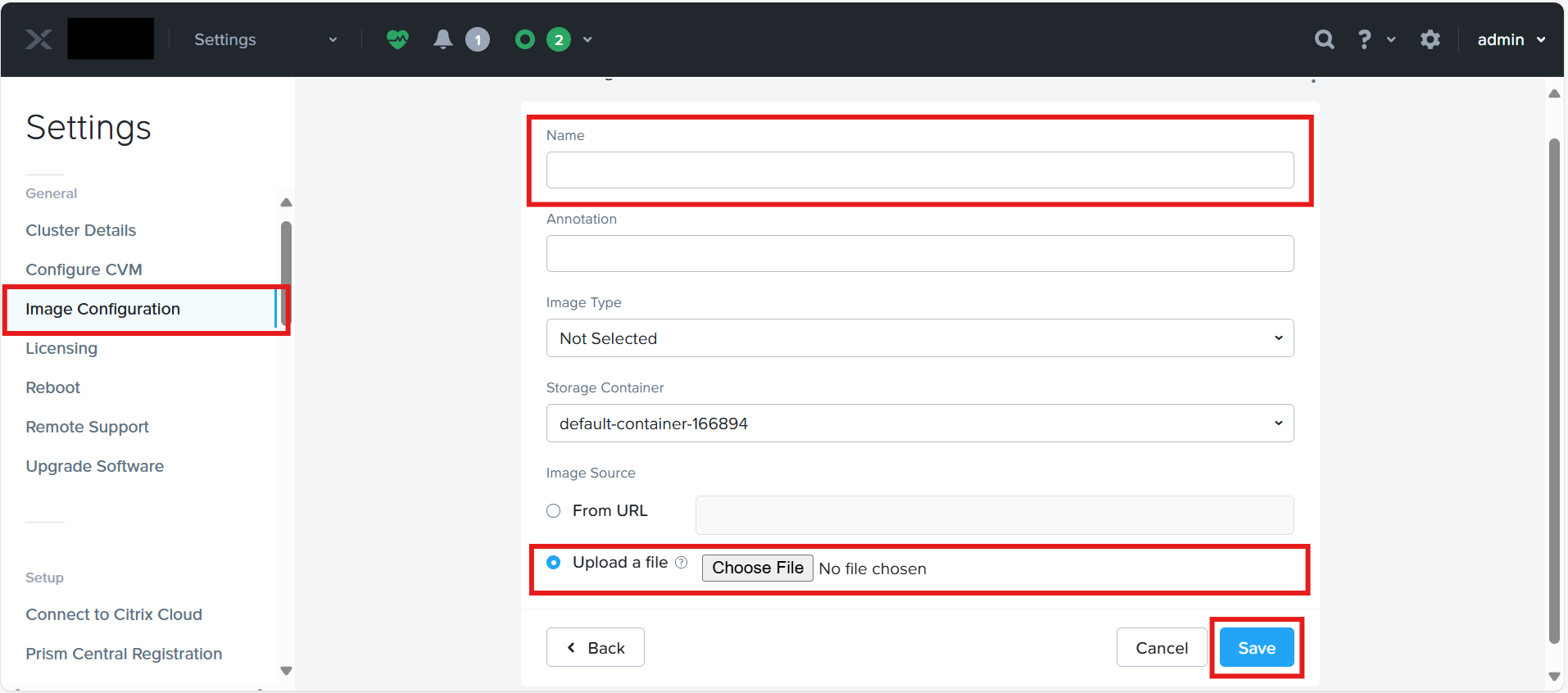


* Tar: On a CMD, Powershell or Terminal, run the command“tar -xf ./path/to/ova/file”.

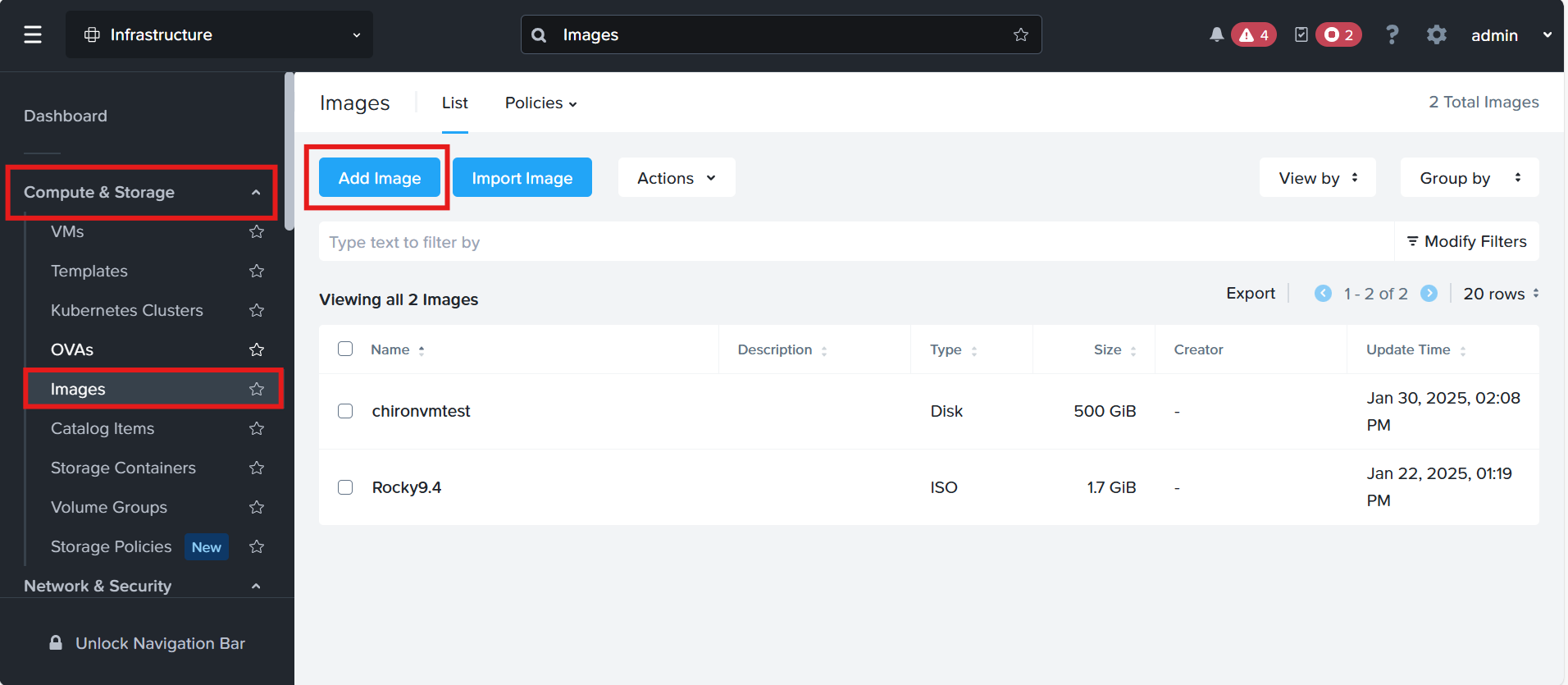
**Step 4** - Upload to your Nutanix Cluster.

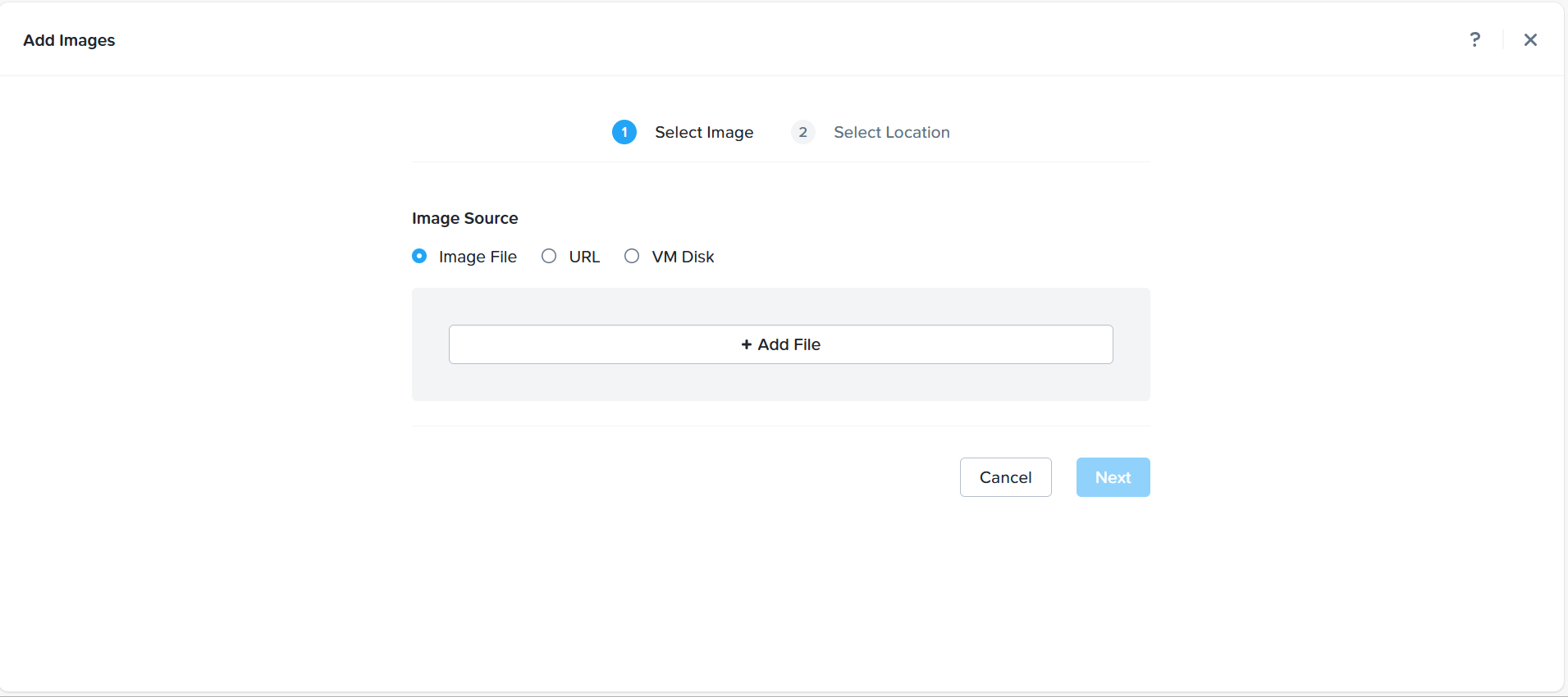
* Prism Element: Login, then go to Settings > Image Configuration > Upload Image > provide a name and Select Upload from File > Click Save.

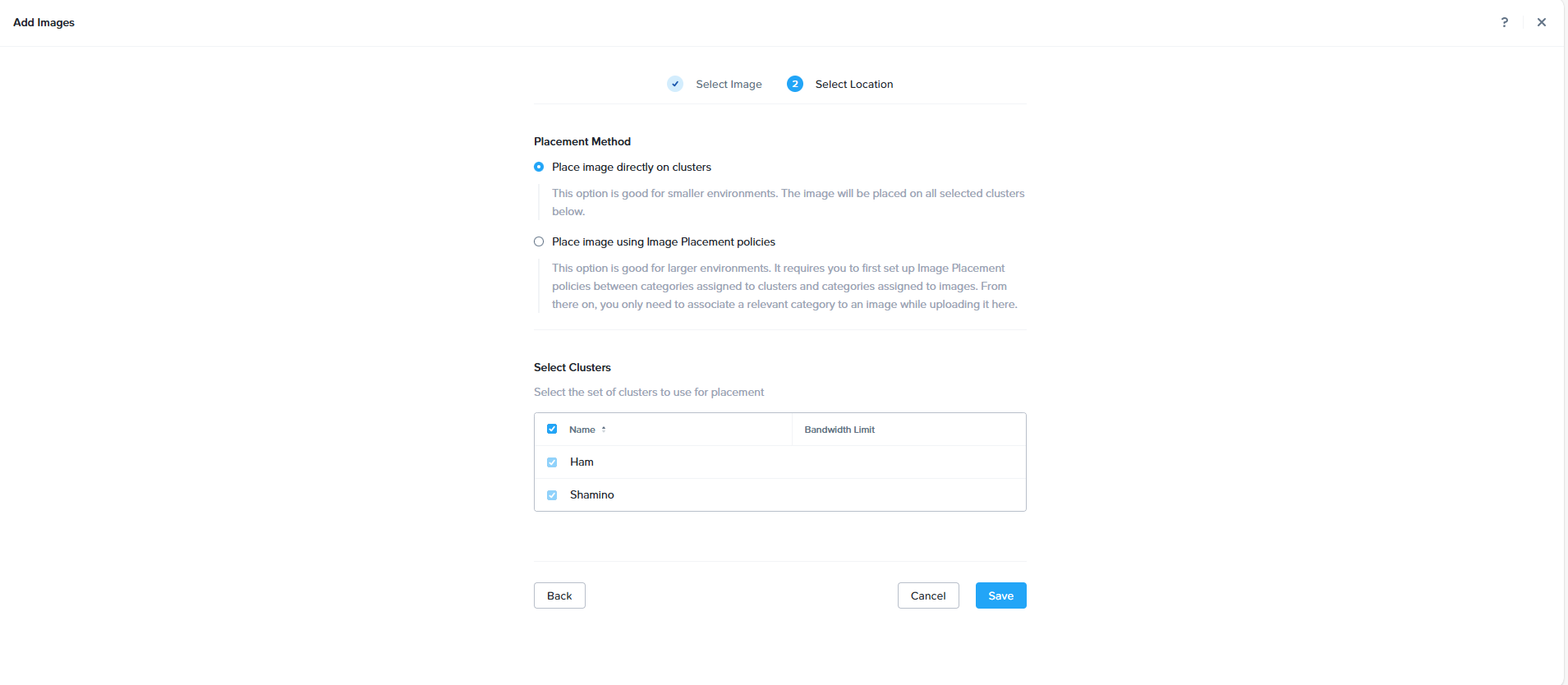




* Prism Central: Login, then go to Images under Compute & Storage > Add Image > Add From File > Select a Cluster to put the Image on or use an Image Placement Policy > Click Save.





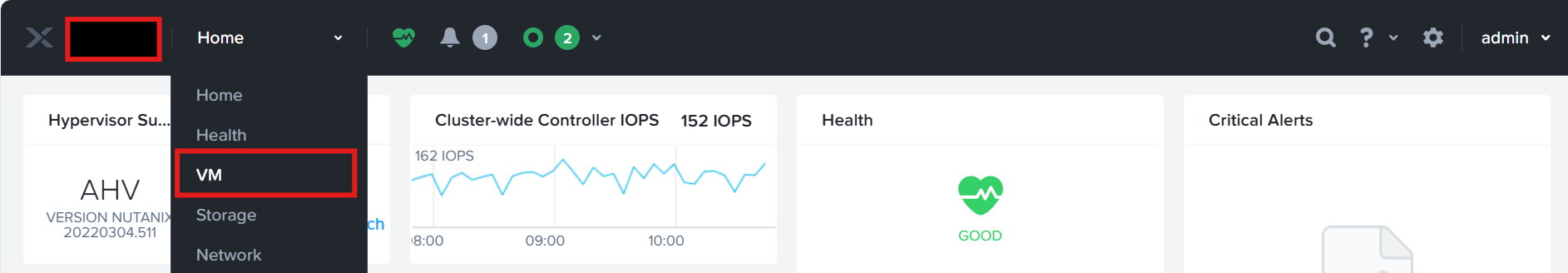


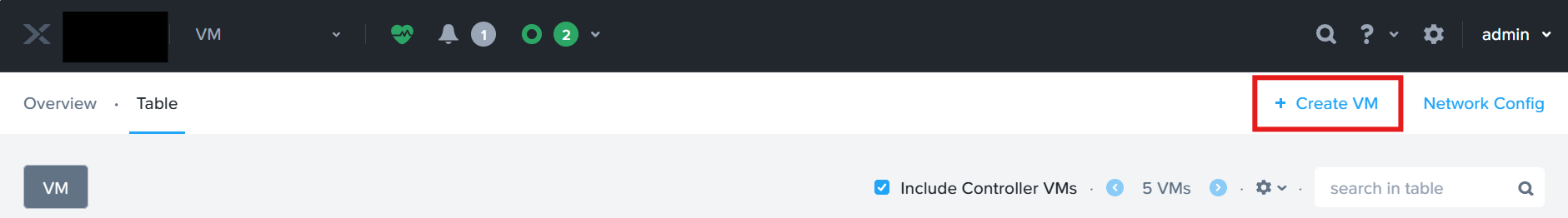
**Step 5** - Wait for the Image to Upload.

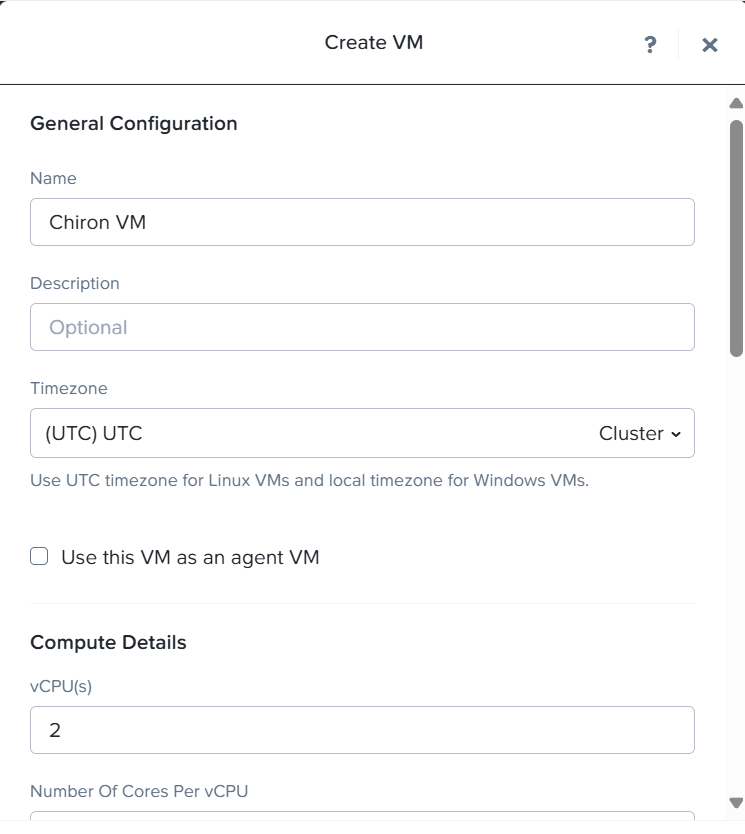
**Step 6** - Create a VM Using the Uploaded Image.

Note: Please ensure that networking is set up on the cluster to allow for automated deployment.

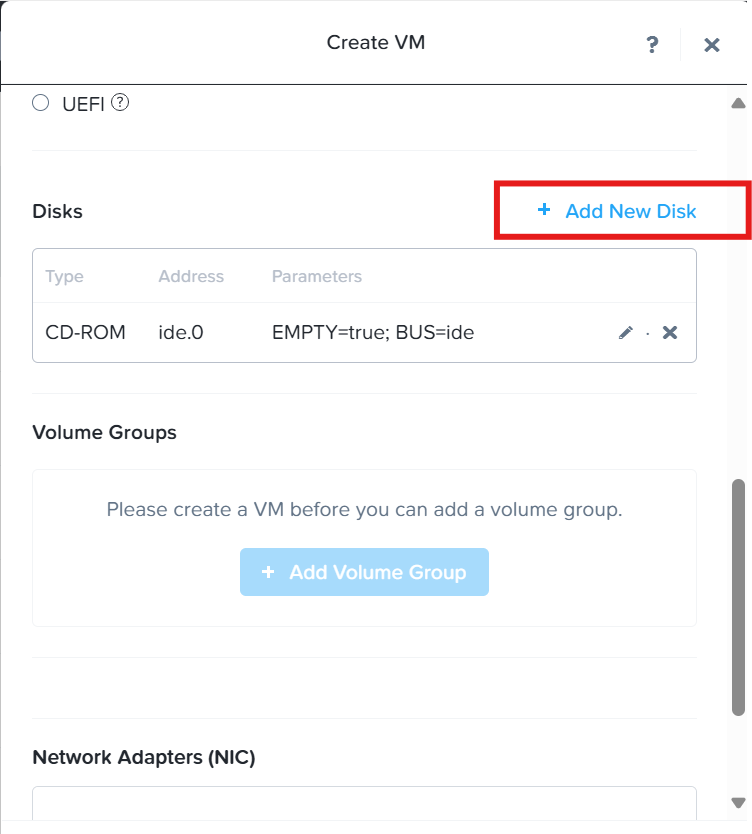
* Prism Element: Drop Down Menu > VM > Create VM > Add Name, vCPU, Cores, Memory, Disk and Network Adapter. Minimum requirements are described in the following section.
* vCPUs: assign as many cores as possible while still leaving enough for the CVM. In our testing, if you only have the Chiron VM running on a cluster, assigning (n - 2) cores where n is the total physical cores give optimal performance.



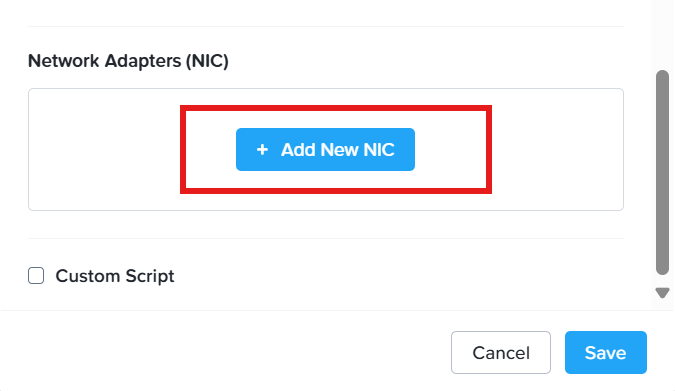


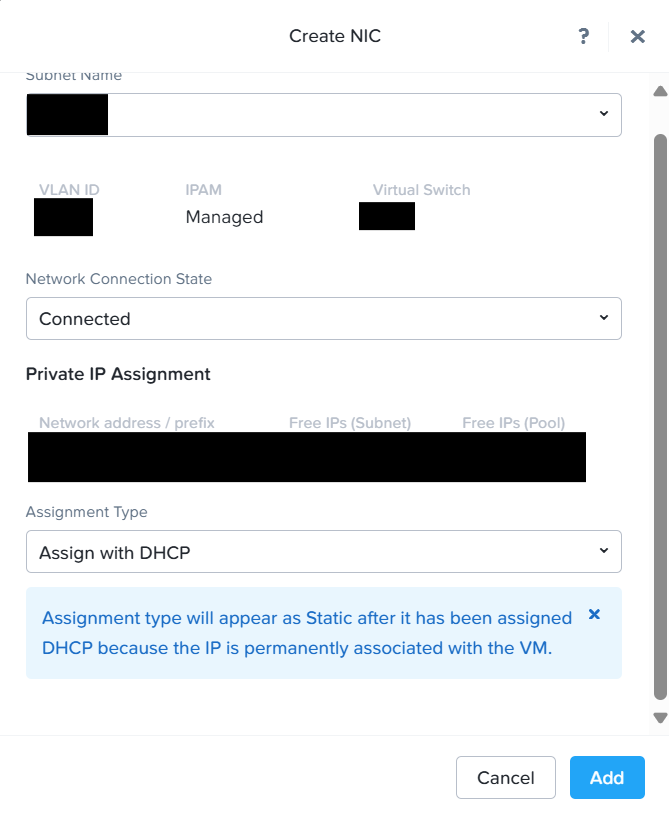


* Add a New Disk > Select Clone From Image Service > Select the Uploaded Image > Click Add



* Add A Network Adapter





* Prism Central: Go to VMs > Create VM > Add Resources > Add Clone Image Disk > Add Network Adapter > Save.

**Step 7** - Power on the VM. On first boot, it will take 3 minutes for the VM to power up and initialise services.

**Step 8** - Go to the IP Address of the VM to trigger download of the AI models, progress is displayed live on your browser. **Do not close this tab or download will be cancelled.**

**Step 9** - Enjoy making videos from KB articles!

Note: For more a more detailed guide to uploading an OVA, please see [KB-3621](https://portal.nutanix.com/page/documents/kbs/details?targetId=kA03200000099TXCAY)

## Prerequisites / Guidelines / Recommendations

Reserve either:

* A Nutanix Cluster via RX (temporary), or
* **A dedicated cluster environment (recommended)**
  + **IMPORTANT**: We **strongly recommend deploying Chiron on a dedicated cluster environment** (e.g., Shazza) for persistent access.

When booking hardware resources, the following **minimum specifications** must be observed:

* Minimum 100 GHz worth of CPU in a single node for usable performance.
* Single node cluster is preferred due to faster Foundation.
* Minimum 256GB of RAM. The Chiron VM itself requires 190GB of RAM to keep the AI models in memory without paging for optimal performance.
* Minimum 1TB of available SSD or NVMe storage. The vDisk is configured as 512GB.
* Internet connectivity. 10gig downlink is recommended for fast AI models download.

An example cluster that satisfies these requirements is Cortana.

VM Recommended Specifications:

* 4 vCpus
* 4 Cores
* 32GB RAM
* 512GB Disk

## ChironVM Management

If you ever want to maintain ChironVM or look under the hood, you can log in locally.

**Login Methods**

* Launch Console in Prism Element or Prism Central.

OR

* SSH to the IP address of ChironVM.

**Credentials**

* Username: chiron
* Password: chiron

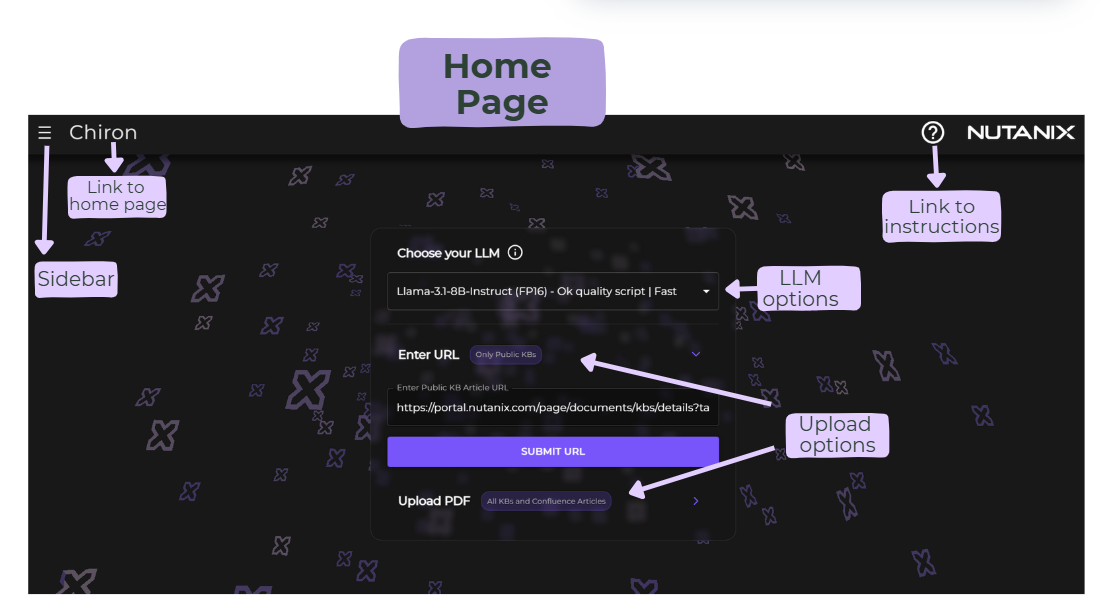
These are directories and commands pertaining to important services running on the Chiron VM, when logged in as the chiron user.:

* ~/chiron/client : Frontend source directory.
* ~/chiron/client/build : Frontend binary directory. This is what gets served by nginx to clients’ browsers. Changes to the frontend source requires a rebuild with `cd ~/chiron/client && npm install && npm run build`.
* ~/chiron/server : Backend source directory, changes to backend source are hot-reloaded by the flask-app service. Note that any type of backend reload disconnects all clients.
* ~/chiron/server/ollama\_parse.py : Contains binding to the Ollama service and list of available AI models. If additional models are added, they will be automatically redownloaded on service restart: “sudo systemctl restart flask-app”.
* /home/ollama\_alt\_home/.ollama/ : storage site of local AI models.
* sudo journalctl -u flask-app --no-pager --output cat -f : backend logs (shows AI models download progress, debug and access log).
* sudo ./dep\_script\_rocky : reinstall/update backend dependencies.
* ~/chiron/server/vm\_deployment.md : Manual VM setup instructions.

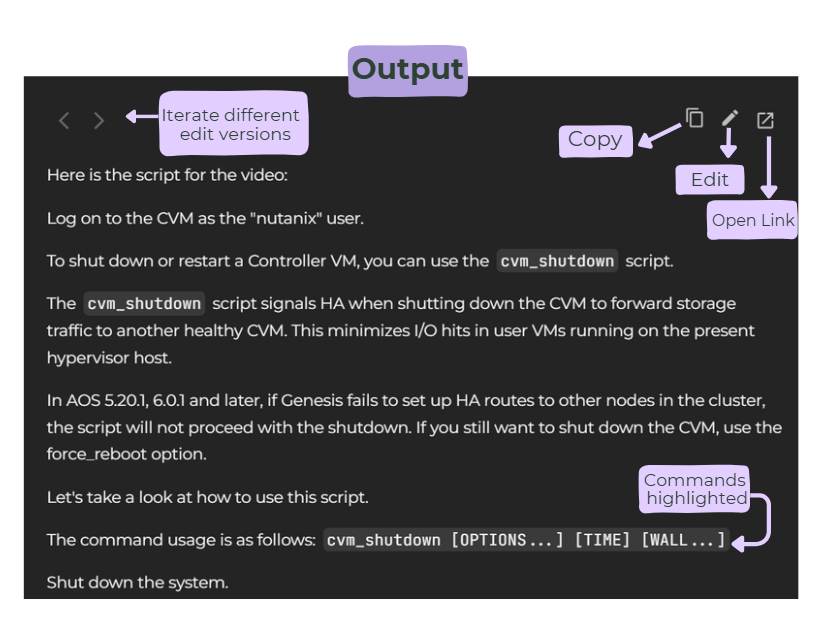
If you would like to manually set up a Chiron VM to your specifications, the instructions can be found [here](https://github.com/Biswas57/chiron/blob/main/vm_deployment.md).

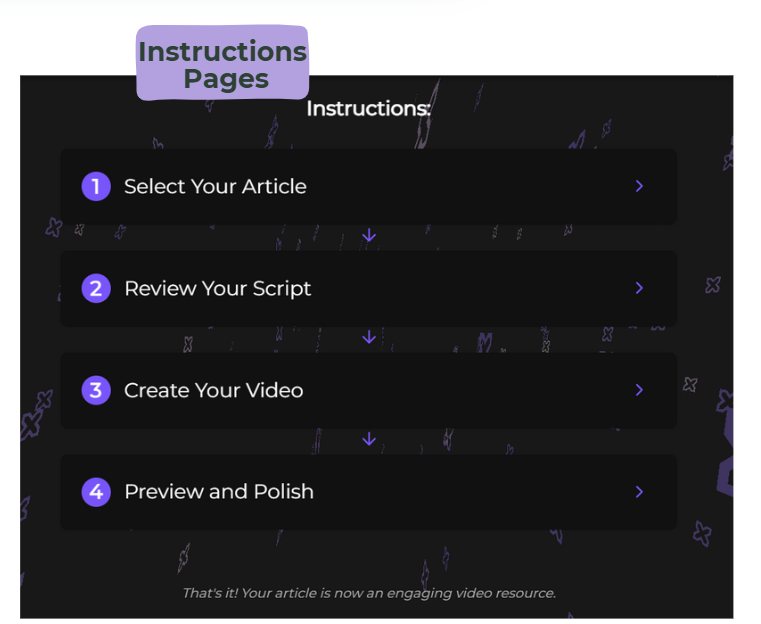
## 

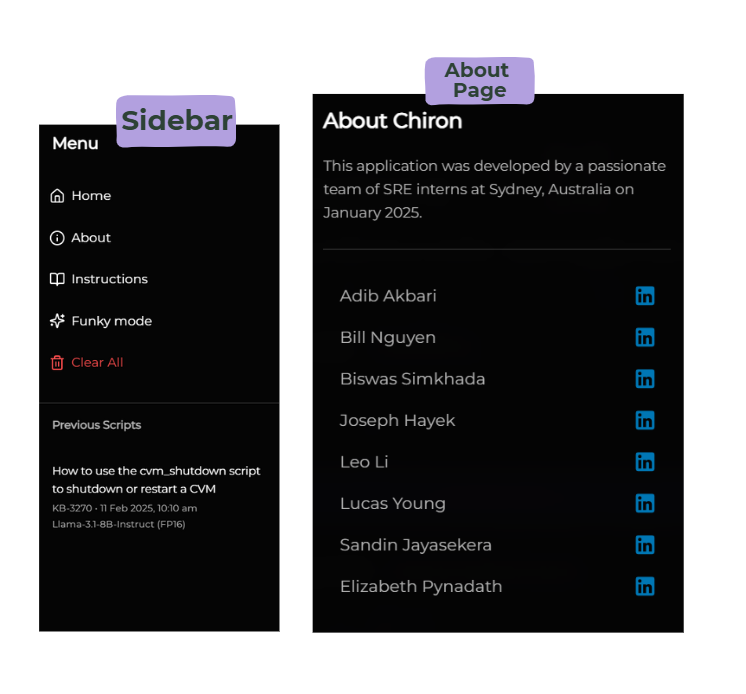
## Frontend UI/UX

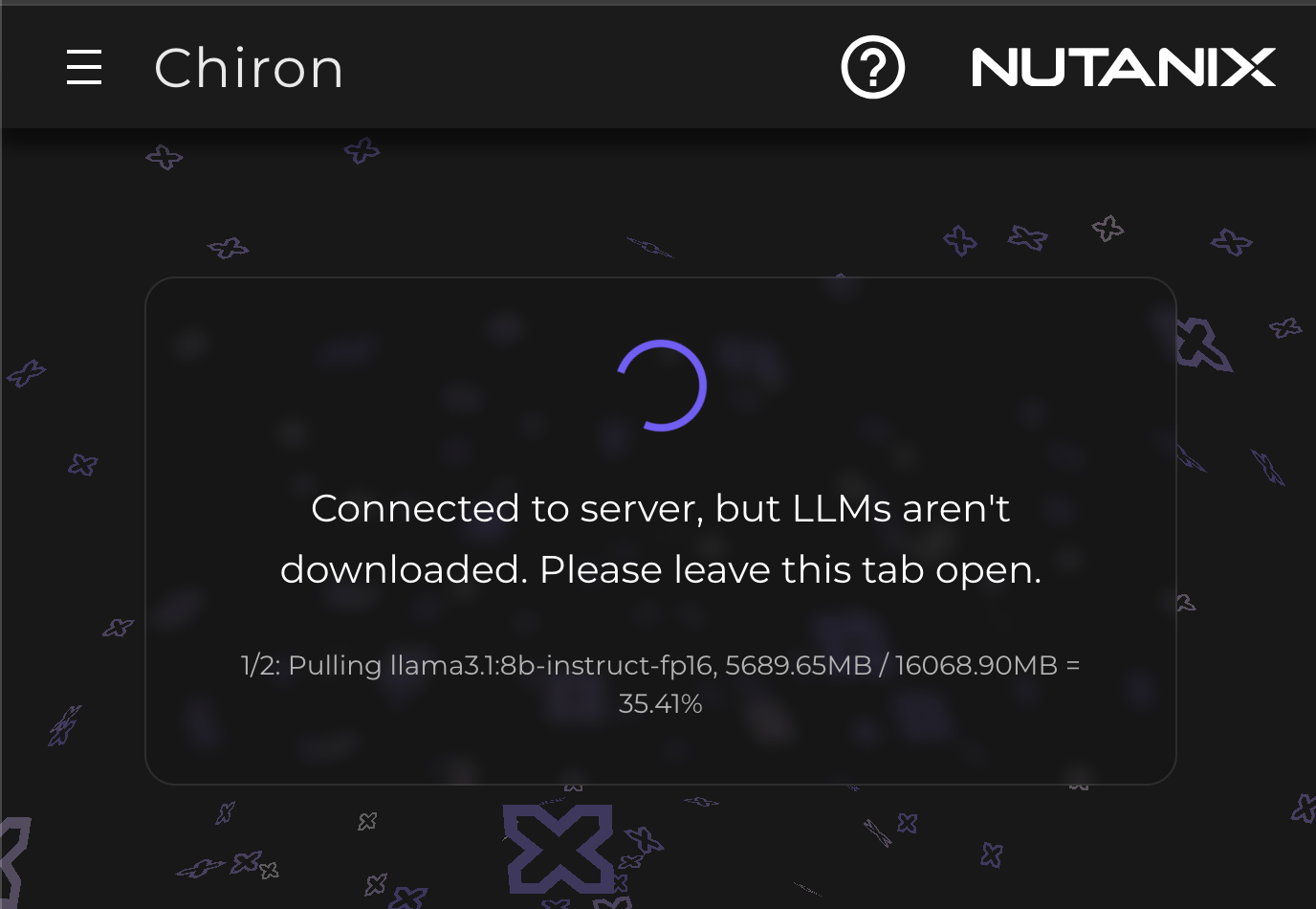
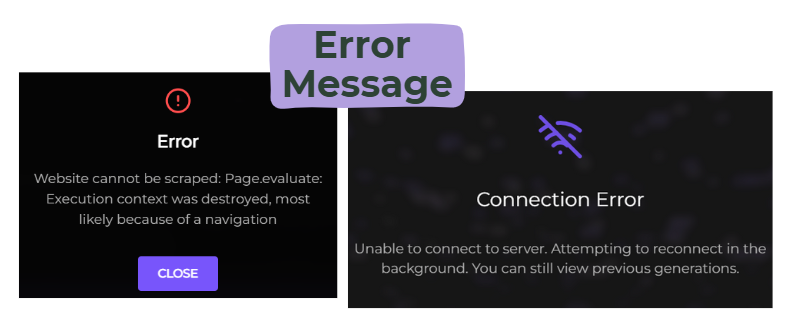




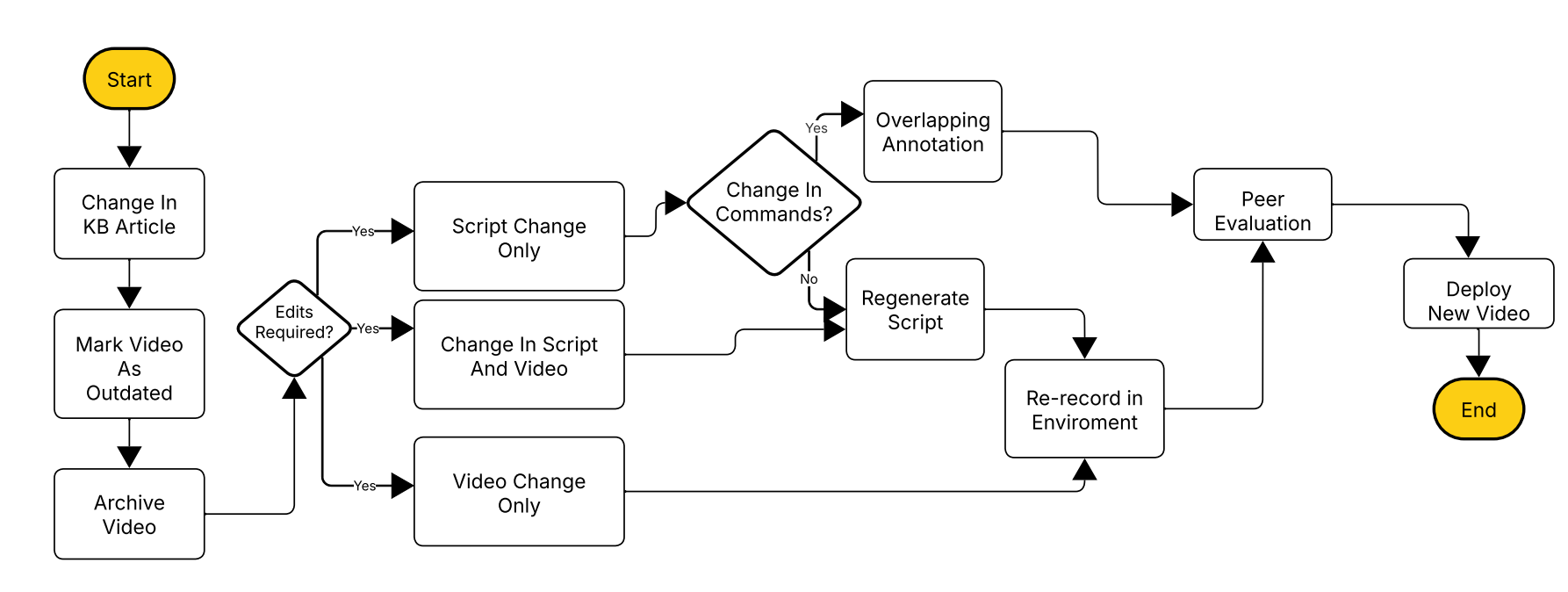








## Maintenance : Update Script or Video



After a video has been generated, there may be changes to the script, commands, or a need for an updated recording environment. The following options are available:

a) If the user needs to update the script commands, they can simply add annotations to the existing screen recording using the Syntheisa tool.

b) If the user needs to update the script’s spoken content, they can enter the new changes or additions into the desired Synthesia slide and regenerate the voice and video.

c) If changes to the environment or commands are required, and the output needs to reflect these updates, the user should record the updated content and replace the relevant recording in the desired Synthesia slide.

d) For significant changes to the knowledge base (KB) articles, the user can regenerate the script via the Chrion website.

Please note this is a rough workflow we came up if the videos were to be integrated with KB articles on the Nutanix Portal, we recommend this to be refined and streamlined with the existing pipeline

# 

# Feedback and Constraints

## Feedback

### Feedback from SREs

| **Suggestions** | **Improvements** |
| --- | --- |
| Extending usage to internal KBs | Implemented a PDF upload option on the frontend website to enable the generation of scripts for internal KBs. |
| Commands should be differentiated in script | When the script is generated, commands are highlighted using markdown style formatting. |
| Ability to edit script after generation | Implemented an edit mode to allow users to delete & modify the generated script text. |
| Ability to convert Confluence articles | Implemented a PDF upload option on the frontend website to enable the generation of scripts for Confluence articles. |
| Ability to view previous scripts generated | Implemented change history and an redo & undo button allowing users to revert to different versions. |

## Constraints and Possible Solutions

### Hardware Limitations

During development, we were limited to hardware available in the SRE labs pool that was available for our use. Running local LLMs gives best performance GPUs with large VRAM to fit the entire model on the GPU. However there were very few GPU clusters, and all GPU clusters have a small VRAM pool. So we decided to focus on running on CPU, which gives vastly inferior performance, but allow us to run larger models on RAM and allow flexibility in deployment as clusters with large RAM are easier to come by.

### Concurrent Usage

Linking back to hardware limitations, running AI models on CPU gives us only 2-3 tokens per second even with a small 8 billion parameters Llama model. If 2 or more users try to generate a script concurrently, our performance would degrade to *seconds per token*, which is unacceptably slow. So only 1 user should use the AI model at any given time. Thus we’ve implemented a queue system where only 1 user can generate a script at any given time. If there are multiple users trying to generate a script, they will get placed in the queue in a First-In-First-Out (FIFO) manner and their document will be serviced by the AI when it is their turn. If a user disconnects (close tab or reload) while they are in the queue, their script generation is cancelled (if applicable) and all proceeding users’ positions in the queue are bumped up. If this system were to be deployed in production, we recommend using a GPU cluster with large VRAM and multiple GPUs. However in our testing, we were unable to get Ollama to distribute the workload to more than 1 GPU, so GPU inference was not further explored as development time was reallocated to more important features.

### Automation of Video Creation

Due to the variability of the KB articles, it is not possible to automate the entire creation process for any KB. Automation ideas were, however, explored. This included ideas such as utilising a UI mock tool that would be able to perform UI actions automatically. However, we would only be able to make this system for a narrow subset of the possible situations presented in a KB article (e.g. entering commands on a terminal window). Hence, this feature was not explored further.

### Sensitive Information

The guidelines of the KB articles state that any sensitive information, including names, IP addresses, and any identifying information should not be present. Assuming the videos produced using this framework meet the same standards, extra work is required to remove this information from the videos.

This specifically applies to the visual elements of the video, since the script should not contain any problematic content (since it is directly based on the KB article, which is already expected to meet these standards). Removing this sensitive information from the visuals are required to be done manually within the video editing software, which in this case is Synthesia.

### Efficiently Handling Article Updates

Any change to the underlying KB article that a given video is based on would likely require a change to the video content. Since the generated video is mostly monolithic, the entire video creation process would generally have to be repeated.

We brainstormed methods of allowing for this eventuality by segmenting the video, which would allow only sections of videos to be modified as articles are changed. However, due to the sequential nature of many of the KB articles, it would likely not be possible to implement this in an efficient manner.

### Document Length Limits

LLMs have a property known as the context window, the size of which dictates the maximum input size for the model. When running the model, this value can be configured (up to a maximum value for that specific model), however, the larger the context window, the larger the memory requirements for running the model.

With the Llama models that we chose, it supports up to 131,072 tokens (a token is typically ¾ of a word). Our backend will count the amount of tokens from the scraped website/PDF and reject any requests that exceed the context window of the selected model.

### AI Script Output

There are some constraints related to the output of the AI.

The LLM output typically includes the relevant content of the KB, including steps on how to perform all required actions. However, some other information may be missing, such as recommendations, requirements, and alternative methods.

Additionally, the phrasing and flow of the generated script has room for improvement, and these changes would have to be performed manually by the user.

As discussed previously, larger LLM models follow prompts with higher reliability, and as such, these issues can be somewhat mitigated at the tradeoff of increased inference time and higher memory requirements.

### Synthesia

Synthesia’s text-to-speech has several minor issues that require workarounds in certain situations.

The pronunciation of certain words sometimes occurs incorrectly. Hence, manual review of the generated video is an essential step of the process. Workarounds to resolve this issue can include putting quotation marks around the problematic word, or experimenting with tweaking the grammar surrounding the word.

A specifically problematic scenario involves the speech generation for CLI commands, and file names. E.g. pronunciation of the ".sh" file extension may be incorrect. In this case, converting the file extension to the words required for pronunciation solves the issue: "dot s h".

### Prompt

Our UI currently offers no option to customise the prompt that is prepended to the KB article before it is passed to the LLM. This feature was evaluated, and was intentionally not implemented to allow the generated scripts to meet a standard baseline (before any manual edits are made by a user).

This feature could be explored in the future if finer control over the generated script is required, e.g. if the script should focus on a specific scenario within a KB.

### Lack of Mobile Optimisation

The Chiron frontend is not optimised for mobile usage, and as such, UI elements may not render correctly or be malformed. This limitation was not addressed during development, as, based on the intended use case, this software would be running on an internal SRE cluster, which would only be accessible via the VPN. Since VPN access is not officially supported on mobile devices, this use case was deemed to be out of scope for our program.

# 

# Future Enhancements

## Testing

To enhance the reliability and maintainability of our application, we propose implementing a comprehensive testing strategy across both frontend and backend components:

**Frontend Testing Strategy**

* Implement Jest and React Testing Library for unit and integration testing
* Focus on critical user flows:
  + Document upload and URL submission
  + Script generation and editing interface
  + Modal interactions and error states
* Add Cypress for end-to-end testing of the complete user journey

**Backend Testing Strategy**

* Implement pytest framework for Flask API endpoints
* Add unit tests for LLM integration and script generation logic
* Implement mock testing for external services and API calls
* Add integration tests for the complete backend pipeline

The implementation of these testing frameworks will provide:

* Increased code reliability through automated test coverage
* Faster detection of regressions during updates
* Improved development workflow with continuous integration
* Better documentation of expected behavior through test cases
* Reduced debugging time and maintenance costs

## LLM Finetuning

To further enhance the accuracy and relevance of generated scripts, fine-tuning the locally hosted AI model will be explored. Instead of relying solely on pre-trained models, techniques such as Supervised Fine-Tuning (SFT) and Reinforcement Learning from Human Feedback (RLHF) could be implemented to adapt the model to domain-specific requirements.

* SFT would involve training the model on historical KB articles and previously generated scripts to refine its understanding of Nutanix-specific terminology and troubleshooting methods.
* RLHF could be employed to incorporate real user feedback, allowing the model to improve over time based on corrections and user preferences.

Hosting a fine-tuned local model would also improve security by ensuring that proprietary KB content never leaves the organization’s infrastructure, while also reducing reliance on external APIs. By implementing these fine-tuning techniques, the model’s output would become increasingly aligned with company-specific knowledge and best practices, enhancing the overall efficiency of the script generation process.

## Video Hosting

Once the video has been generated from Synthesia, it is currently up to the SRE to decide what to do with it. Options include:

* Embedding the video on the Knowledge Base (KB) article page. This way, customers would be able to solve issues without having to raise a ticket, as they can see the process actually being done. However, the video would need to go through the KB approval process.
* The site could check if an existing video for KB exists and notify users if they want to use that or create a new one.

## Copilot Integration

With the recent approval of the use of Microsoft 365 Copilot Chat, future integration with the tool would be optimal. It would remove the setup and hosting process, resulting in a seamless experience with potentially less overhead.

## Prompt Editing

Currently, the prompt is standardised and cannot be edited. However, the SRE may want to summarise a specific section, or focus on a particular method. Having this functionality in the future may be useful. However, the current benefit is that since the prompt is standardised, the output is similar across videos.

# Conclusion

The Chiron framework transforms complex and lengthy internal and public-facing articles into concise, engaging videos that replicate the lab environment. By generating scripts through the Chiron website and recording videos with Synthesia, users can access a more dynamic learning format. This video-based approach has been shown to significantly improve learning outcomes. As Prof. Jim Inc. notes, 85% of participants felt disengaged with traditional reading methods. In essence, Chiron offers a more accessible, engaging, and effective way to learn, surpassing the limitations of conventional materials.

1. *Improving online education: Prof jim case study* *Rapport*. Available at: https://www.rapport.cloud/use-cases/prof-jim (Accessed: 01 February 2025). [↑](#footnote-ref-0)
2. *Prof Jim Inc..* *Prof Jim*. Available at: https://www.profjim.com/ (Accessed: 09 February 2025). [↑](#footnote-ref-1)