Contents

Validated Statistical Computing: 3

A Kubernetes Framework for Regulatory-Compliant R, Python and SAS Deployments . 3

Summary 3

Introduction 3

SAS Workload Processing Architecture 4

SLC Hub Architecture 6

Kubernetes Benefits for Statistical Computing 8

Comparative Analysis: SLC Hub vs Kubernetes Deployment Models 11

SLC Hub Architecture: Integrated Commercial Solution 11

Disadvantages 13

Kubernetes Deployment Model: Flexible Cloud-Native Platform 13

Advantages 14

Disadvantages 15

Decision Framework 16

Hybrid Considerations: 17

Complete Artifact Inventory for SLC Deployment for Kubernetes 18

Persistent Volume (PV) Definitions 18

Persistent Volume Claims (PVC) 18

Storage Classes 18

Secret Management 18

Container Image Artifacts 18

Base Container Image: SLC Worker 18

Build Process: 19

Container Registry 19

Kubernetes Deployment Artifacts 19

Deployment Configuration: slc-deployment.yaml 19

Service Configuration: slc-service.yaml 19

Service Account & RBAC 20

Configuration Management Artifacts 20
Worker Node Configuration 20
Validation Framework 22
R Installation Validation 24
Python Installation Validation 25
Image Build Validation 26
User Interface Components 28
Integration Points 28
Compliance and Audit 29
Performance Optimization 31
Storage Performance Optimization 32
Secrets Management Best Practices: 34
Network Security 34
Conclusion and Recommendations 36
Implementation Roadmap 36
Appendix A: Artifact Checklist 38
Appendix B: Glossary 40
Appendix C: Sample Docker File for Image build of SLC, R, and Python 41
Appendix D: Requirements.txt file for Container Image 46

Paper TT-13

Validated Statistical Computing: A Kubernetes Framework for Regulatory-Compliant R, Python and SAS Deployments

Randy Betancourt, Siemens, Portland OR, USA
Tim Fantuzzo, Siemens, Troy MI, USA

Summary

This paper presents a Kubernetes-based deployment framework for managing validated statistical computing environments using Altair SLC (SAS Language Compiler), R, and Python. The framework addresses FDA regulatory requirements (21 CFR Part 11) while providing scalability, high availability, and proper change control for clinical data analysis environments.

Introduction

The pharmaceutical industry faces increasing pressure to adopt open-source statistical tools (R, Python) alongside traditional SAS workflows while maintaining FDA compliance for computerized systems in clinical trials. Key regulatory requirements include:

- **Data integrity and traceability** (21 CFR Part 11)
- **Audit trails and electronic signatures**
- **System validation documentation**
- **Change control processes**
- **Reproducibility of statistical analyses**

Historically this has meant the management of different software and hardware stacks whose updates and lifecycle management presented significant administrative burdens.

Software stacks are often deployed across multiple hardware systems requiring constant maintenance of the integration points between these disparate systems. To help ease this administrative burden, this paper proposes a series of steps resulting in the management of machine images rather than individual deployments of these components:

- **SLC (SAS Language Compiler)** - Siemens SAS language compiler alternative
- **R deployments** with validated package libraries
- **Python deployments** with controlled environments
- **Storage architecture** able to scale into petabyte ranges, provide a seamless user experience and attempts to minimize administrative overhead burdens

Once images are built for each validated software component, it is configured, tested, and validated, either the SLC Hub architecture or Kubernetes managed cluster enables the cluster management and deployment lifecycle. The bulk of this paper attempts to describe the advantage and dis-advantages of managing large-scale SAS language processing workloads capable of supporting thousands of users and able to manage petabytes of data sets.

Some CIOs we speak to are surprised to find how the SAS user community is able to create and preserve SAS data sets (.sas7bdat) in the tens of petabyte of storage. Some of this storage is justified for meeting regulator requirements, while some of this storage is needless duplication owing to lax storage allocation policies.

SAS Workload Processing Architecture

The evaluate the trade-offs between Kubernetes managed clusters and SLC Hub's cluster management regimen, we must first understand the SAS workload execution stack of both software and hardware.

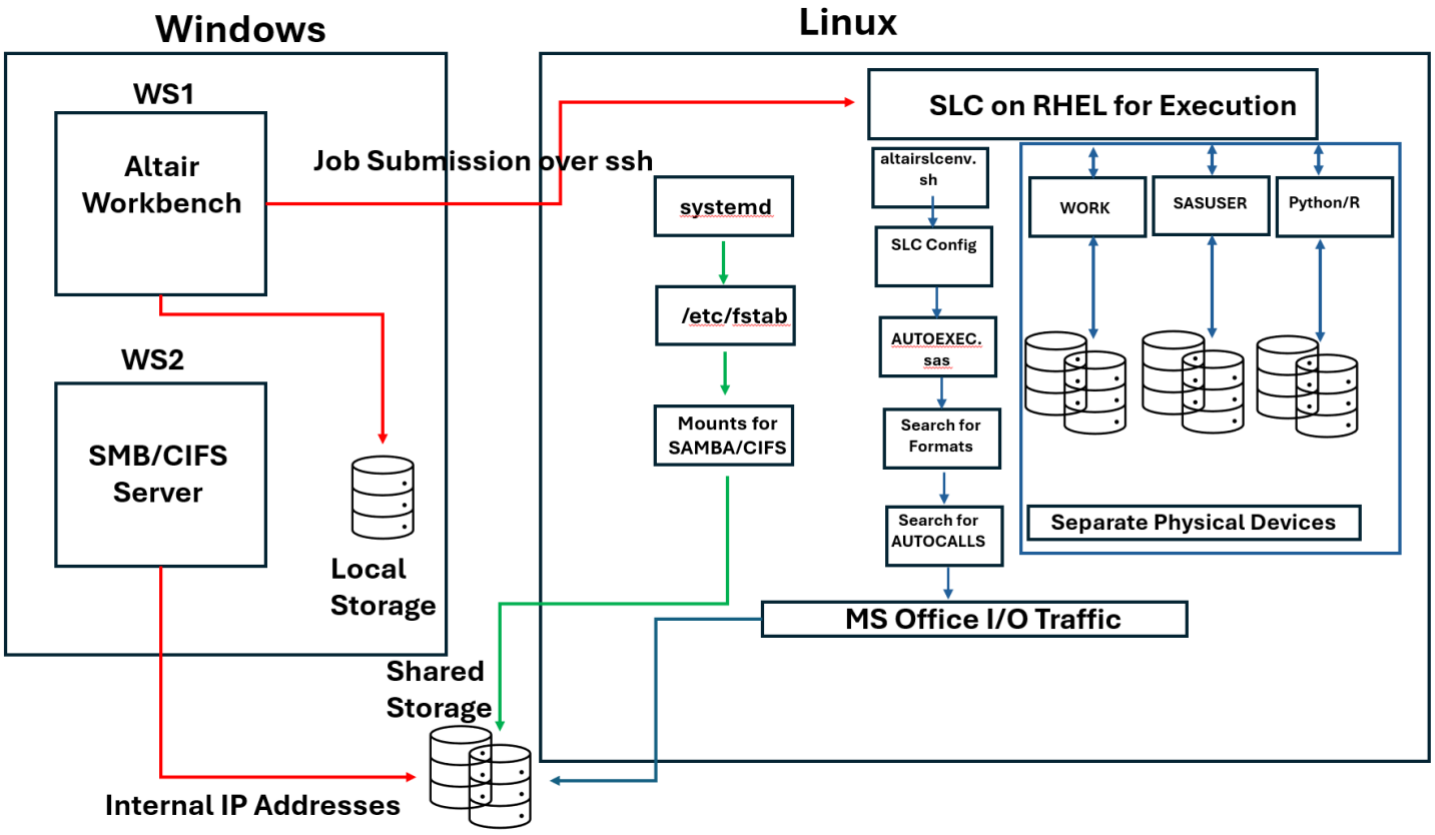

Figure 1. Typical SLC Single Server Deployment

Siemens customers frequently start with deployments of a single, SMP-based server to handle moderate size SAS language processing workloads. See Figure 1, Typical SLC Single Server Deployment.

These deployments are characterized as traditional client-server processing. The user's visual interface is deployed on a Windows desktop or accessed through a VDI or Citrix server for scalability purposes. This single SMP-server deployment model is suitable for LOB deployments supporting less than 20 concurrent users.

Using VDI or Citrix for U/I deployments conforms to the security principles of not allowing corporate data assets to be copied to local storage. Firms can opt to deploy the U/I on Linux, provided they permit the use of an X-server running on both the Linux and the Windows sides.

From the U/I, users perform a one-time configuration set-up for enabling Desktop to remote server connections over ssh. From there, users use the U/I in the traditional manner of composing code, performing execution tasks, and reviewing generated log and output artifacts.

Below, we discuss the use of alternate, light-weight U/Is, like VS-Code.

If an organization prohibits use of ssh, they must configure an alternate authentication mechanism like Kerburous tickets, AD authentication, RACF profiles, etc. Details for authentication integration is beyond the scope of this paper. See the document entitled SSH Key Configuration for SLC Workbench for RHEL.

The Python and R libraries are deployed in this same server environment and SLC finds these libraries by reading a named set of environment variables that are injected into the SLC execution context.

The advantages of this deployment is that it is a well-understood provisioning method with little on-going overheads after the initial configuration is established and validated. And because the number of users is small, only basic consideration is needed for the storage architecture for processing transient and permanent SAS data set files.

The disadvantage is this is a single point of failure (SPOF) model.

Workload scalability is limited, since there is only vertical scaling and no horizontal scaling (adding additional worker nodes). There is no load balancing. Performance bottlenecks can be encountered since each concurrent users invokes a separate instance of SLC. As a result, this deployment is not suitable for supporting high-availability SAS compute and I/O workloads falling under redundancy protocols for DR and data center fail-over scenarios.

This deployment model does not support enterprise SAS language features such as authdomains and a built-in scheduler for executing long-running batch jobs. Authdomains are an abstraction for credentials management inside user's SAS language programs. Configured authdomains prevent users from storing credential strings in clear text as part of the program's execution logic.

This abstraction also hides physical changes in the back-end infrastructure which makes user programs more resilient and less error-prone in cases where infrastructure changes are made. For more details, see the paper entitled, Use Case: SLC Managing Credentials in User Programs.

This deployment model supports batch processing with schedulers like cron or Control/M. These schedulers are external and require additional user inputs and are less convenient that a built-in scheduler. For more details, see the paper entitled, SLC Use Case: Execute Batch SAS Language Programs with cron's scheduler

SLC Hub Architecture

The SLC Hub architecture is designed to address the short comings in the previously described client-server deployment model. The software architecture provides a secure, scalable platform that offers a range of services to support enterprise-class SAS workload processing. These include:

- **Orchestration layer** for load-balanced workload distribution
- **Global metadata objects**: Authdomains for authentication, database connections, resource definitions
- **Highly scalable** with horizontal scaling capability
- **Built-in scheduler** eliminating the need for users to become familiar with external schedulers

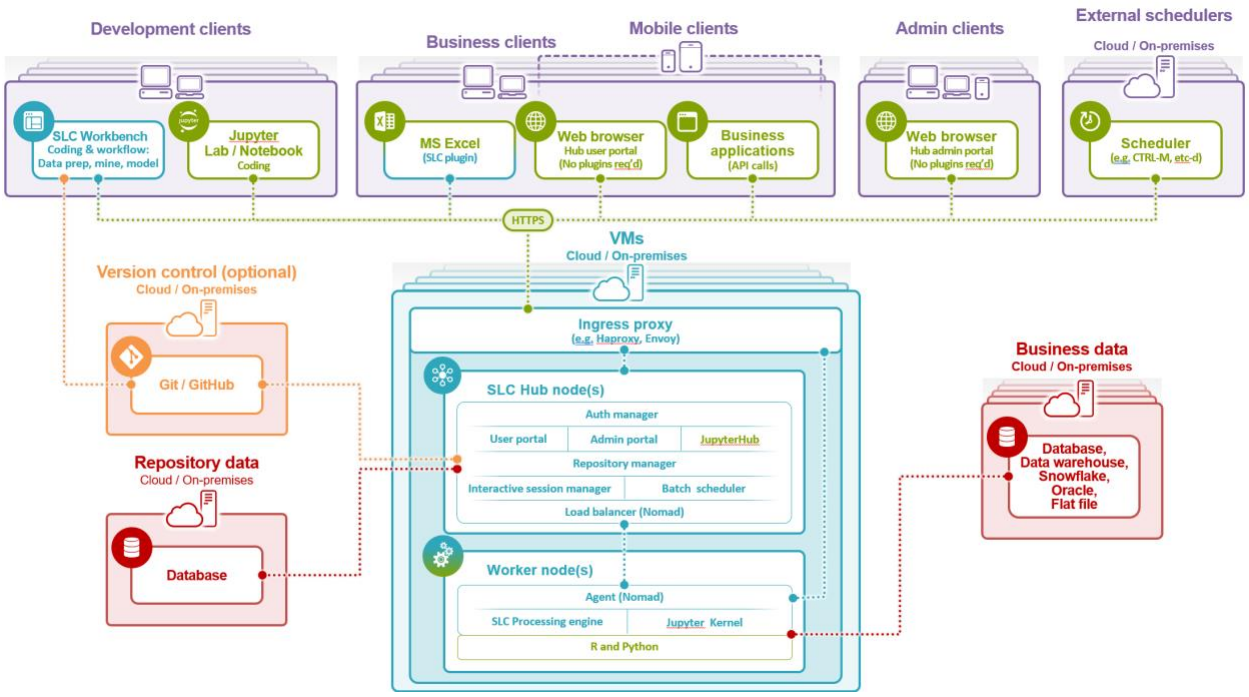

Figure 2. SLC Hub Architecture

The SLC Hub architecture is built to be an elastic and scalable platform. It is composed of a Hub node as the control plane, and a collection of worker or compute nodes managed by the Hub(s) to which workloads are distributed by the Hub node for execution according to workload profile rules. Results are returned to the Hub and returned to end-users and machine-to-machine clients (browsers, SLC Excel plug-in, business applications making REST API requests).

The software has five major sub-components described below.

1. Eclipsed-based Analytics Workbench is a rich GUI/IDE used to develop analytics applications both through visual drag-and-drop Workflows and coding. Programmers compose SAS language programs, SQL, Python, and R. Non-programmers use a palette abstracting coding to visually design workflows.

   During development, programs and workflows can be executed interactively both locally and remotely via the Hub. The Workbench has a visual interface to wrap programs in a parameterized package ready for testing and deployment into a Hub. The parameterized package can be executed and tested locally in the Workbench without needing to be published to a Hub. Once a package is ready for publishing, it can be transmitted to a Hub as a Hub package (described next) using a simple point-and-click wizard in the Workbench.

A Hub administrator defines profiles that map resources to specific worker nodes. For example, requests to invoke a program that reads data from a particular Oracle database may require a profile defined for worker node instances 1 and 2 running Linux. Access to Snowflake requirements are fulfilled by worker node 3 connecting to this database etc.

2. Hub Console. This management portal is accessed by a web-based GUI providing the ability to manage services such as auth domains, library definitions, execution profiles (fine-grain control for multiple SAS language program execution configurations), control of repositories, building workload pipelines (chained programs), job scheduling, REST API deployment and publishing.

   All configuration management and change control activities are performed through the Hub Console by the platform administrator.

3. User Portal. This portal is accessed in an equivalent way to the Hub Console but is intended for end users to run deployed programs and access results.

4. Workers: Deployed programs are orchestrated for execution by the Hub node onto one of multiple worker nodes in a load-balanced fashion. Multiple, different teams of users can have workloads specified to run on specific worker nodes according to security, performance and platform requirement profiles.

   Workloads can include both production scheduled processes as well as interactive user processes. These services are similar to a Kubernetes containerized environment but use conventional VMs as worker nodes rather than container pods.

   To meet high availability requirements for this software, Siemens will provide built-in HA capabilities into SLC Hub in early 2027. This requirement can be met today by deploying multiple, redundant SLC Hubs using 3rd-party software like Veritas Cluster services (VCS).

Kubernetes Benefits for Statistical Computing

The final workload processing model discussed is the use of Kubernetes for cluster management to take advantage of a declarative configuration management style of deployment.

The Kubernetes architecture provides several distinct advantages for managing validated statistical computing environments in pharmaceutical and clinical research settings. These benefits align well with FDA regulatory requirements while offering modern cloud-native deployment patterns.

Unlike the SLC Hub software architecture, the administrator must be familiar with a wider-range of technologies and procedures making this approach more technically challenging than relying on a commercially available software stack.

Specifically, the administrator must be able to take the provided SLC software along with the Python and R libraries and encapsulate them into containerized images. Organizations must already have life-cycle management processes in place for handling these container images.

The initial image build process is a bit of a trial-and-error process, since the image definitions begin with a bare-bones Linux OS, in the author's case. The administrator must plan ahead to ensure all of the requisite libraries, OS utilities, and tooling needed by the end-user software components are available at image creation time.

These defined images are then deployed inside of pods and must contain all software dependencies. The author's experience was that on occasion, SAS language programs would execute a code path which in turn had a dependency on an OS-level service that had not been provisioned at container build time. This required a complete re-build of the image to include this dependency and a new deployment cycle for the pods.

This was also true when calling into Python or R, which in-turn attempts to load a specific library or module. If the requisite library or module was not found inside the pod, then a rebuild of the image was required.

Infrastructure as Code

Kubernetes enables infrastructure management through declarative configuration files, typically written in YAML format. These configuration files define desired states for the computing environment, including pod specifications, service definitions, storage volumes, and network policies. The declarative approach means administrators specify what the environment should look like rather than writing procedural scripts for how to build it.

Version-controlled infrastructure definitions allow teams to track all changes to the computing environment using standard source control systems like Git. Each modification to the infrastructure is documented with commit messages, creating a complete audit trail of who changed what and when. This versioning capability is essential for meeting FDA requirements around change control and system validation documentation.

Reproducible environments emerge naturally from the infrastructure-as-code approach. The same configuration files can be deployed across development, validation, and production environments, ensuring consistency. When regulatory audits require demonstration of environment configurations from specific time periods, teams can check

out historical versions of configuration files and understand exactly how systems were configured at any point in time.

High Availability

Automated failover and self-healing capabilities are built into the Kubernetes control plane. When a pod fails health checks or becomes unresponsive, Kubernetes automatically terminates the problematic pod and launches a replacement. This automatic recovery happens without manual intervention, reducing the mean time to recovery (MTTR) for system failures and ensuring statistical computing workloads continue with minimal disruption.

Pod replication across nodes distributes workloads across multiple physical or virtual machines in the cluster. Kubernetes maintains the specified number of pod replicas and ensures they are spread across different nodes to avoid single points of failure. If an entire node fails due to hardware problems or maintenance activities, pods running on that node are automatically rescheduled to healthy nodes elsewhere in the cluster.

Load balancing built-in through Kubernetes services provides transparent distribution of requests across multiple pod instances. Users and applications connect to a stable service endpoint rather than individual pods. Kubernetes automatically routes traffic to healthy pods and removes failed pods from the rotation. This built-in load balancing eliminates the need for external load balancer appliances and simplifies the overall architecture.

In order to support this feature, the administrator must have strong knowledge of TCP/IP networking principles, understand and trouble-shoot network failures, and have familiarity with Kubernetes networking architecture.

Scalability

Horizontal pod autoscaling allows the computing environment to automatically adjust capacity based on observed metrics like CPU utilization, memory consumption, or custom metrics specific to statistical workloads. When user demand increases during peak analysis periods, Kubernetes automatically launches additional pods to handle the load. Conversely, during quiet periods, excess pods are terminated to conserve resources and reduce costs.

Dynamic resource allocation enables fine-grained control over CPU and memory resources assigned to each workload. Teams can specify resource requests (guaranteed allocations) and limits (maximum consumption) for each pod. Kubernetes uses this information to make intelligent scheduling decisions, packing workloads efficiently across available nodes while preventing resource contention that could impact analysis performance.

Change Control

Immutable container images provide a foundational element for validated computing environments. Once a container image is built containing specific versions of SLC, R packages, or Python libraries, that image cannot be modified. Any changes require building a new image with a new version tag. This immutability ensures that validated environments remain unchanged and prevents configuration drift that could compromise reproducibility of statistical analyses.

Versioned deployments maintain a complete history of all application rollouts. Each deployment in Kubernetes is tagged with a revision number, allowing teams to track exactly which version of each component was deployed at any time. This versioning integrates naturally with validation protocols, where specific software versions must be tested and approved before production use. Teams can demonstrate to auditors that only validated versions were deployed to production environments.

Rollback capabilities provide a safety net when deployments encounter problems. If a new version of statistical software or updated package libraries causes unexpected issues, administrators can instantly revert to the previous working version with a single command. Kubernetes maintains the configuration and container images from previous deployments, making rollbacks rapid and dependable. This capability reduces the risk associated with changes and enables more frequent updates while maintaining system stability required for regulated environments.

Comparative Analysis: SLC Hub vs Kubernetes Deployment Models

Organizations evaluating deployment strategies for validated statistical computing environments must weigh the trade-offs between the integrated SLC Hub architecture and Kubernetes-based cluster management. Both approaches address the limitations of single-server deployments, but they differ significantly in implementation complexity, operational requirements, and organizational fit.

SLC Hub Architecture: Integrated Commercial Solution

The SLC Hub architecture represents a purpose-built solution specifically designed for enterprise SAS language processing workloads. This integrated approach offers several compelling advantages for organizations seeking to minimize operational complexity while meeting regulatory requirements.

Advantages

Simplified Deployment and Configuration

SLC Hub provides a commercially-supported, cohesive software stack where all components are designed to work together from the outset. Organizations avoid the trial-and-error process of building and testing container images, as the vendor has already resolved integration challenges between the Eclipse-based Analytics Workbench, Hub Console, User Portal, and worker nodes.

Installation follows documented procedures with predictable outcomes, reducing the time from procurement to production deployment.

The Hub Console provides a centralized, web-based interface for all administrative tasks including auth domain configuration, library definitions, execution profiles, and job scheduling. Administrators can manage the entire platform through point-and-click operations without needing to edit YAML configuration files or understand Kubernetes networking primitives. This graphical approach significantly lowers the barrier to entry for teams more familiar with traditional enterprise software than cloud-native technologies.

Vendor Support and Maintenance

Commercial vendor support provides significant risk mitigation for regulated environments. When issues arise during validation testing or production operations, organizations have direct access to Siemens' support engineers who understand the complete software stack. This support relationship is particularly valuable during FDA audits when technical questions about system architecture or validation evidence arise.

The vendor assumes responsibility for testing component interactions, ensuring that updates to the Analytics Workbench remain compatible with Hub services and that worker node software maintains proper integration with the control plane. Organizations receive tested upgrade paths rather than independently managing version compatibility across multiple open-source projects.

Lower Specialized Skill Requirements

SLC Hub reduces the need for specialized cloud-native expertise. While administrators must understand SAS language processing concepts and basic system administration, they do not require deep knowledge of container orchestration, service mesh architectures, or Kubernetes networking models. This skill profile aligns better with typical pharmaceutical IT departments where staff expertise centers on database administration, application support, and validation processes rather than DevOps engineering.

Organizations can often leverage existing system administration staff to manage SLC Hub deployments after standard training. The learning curve is gentler compared to Kubernetes, where administrators must master concepts like pod lifecycle management, ingress controllers, persistent volume claims, and container networking interfaces before achieving operational proficiency.

Enterprise Features for Statistical Computing

The built-in scheduler eliminates dependencies on external job scheduling systems. Users can schedule batch jobs directly through the Hub Console without learning separate scheduler syntax or requiring access to cron or Control-M systems. This integration

simplifies the user experience and reduces the number of moving parts that must be validated and maintained.

Auth domains provide critical functionality for credentials management in regulated environments. Users can reference database connections and authentication credentials through abstract names in their SAS programs rather than embedding connection strings or passwords in code. When infrastructure changes occur, such as database server replacements or credential rotations, administrators update the auth domain definitions once rather than modifying countless individual programs.

Disadvantages

Limited Cloud-Native Integration

The SLC Hub architecture uses conventional virtual machines as worker nodes rather than containers, limiting integration with modern cloud-native ecosystems. Organizations cannot leverage Kubernetes-native tools for monitoring, logging, and observability.

Service mesh technologies like Istio that provide advanced traffic management and security policies are not applicable to VM-based worker nodes.

Cloud cost optimization strategies that rely on spot instances, preemptible VMs, or fine-grained resource allocation are more difficult to implement with the VM-based worker architecture. The granularity of scaling is at the VM level rather than the container level, potentially leading to less efficient resource utilization in dynamic workload scenarios.

Delayed High Availability

Built-in high availability capabilities are not scheduled for native SLC Hub support until early 2027. Organizations requiring HA functionality before this timeline must deploy multiple redundant Hub instances managed by third-party clustering software like Veritas Cluster Services. This approach introduces additional complexity, licensing costs, and another technology stack that must be validated and maintained.

The dependency on external clustering software partially negates the simplicity advantage of the integrated Hub architecture, as administrators must now understand both the SLC Hub platform and the clustering solution. Failover testing becomes more complex, requiring coordination between the Hub software and the clustering layer.

Kubernetes Deployment Model: Flexible Cloud-Native Platform

Kubernetes provides a powerful, flexible foundation for deploying statistical computing workloads with native support for modern cloud infrastructure patterns. However, this flexibility comes with substantial implementation and operational complexity that organizations must carefully consider.

Advantages

Infrastructure as Code and Declarative Configuration

Kubernetes configuration files provide precise, version-controlled definitions of the entire computing environment. Teams can track every infrastructure change through Git commits, creating an audit trail that aligns well with FDA validation requirements. The ability to diff configuration changes and roll back to known-good states provides strong change control capabilities that are difficult to achieve with GUI-based management tools.

The declarative approach enables consistent deployments across development, validation, and production environments. The same YAML manifests can be applied to multiple clusters, ensuring that validated configurations remain identical as they progress through the software lifecycle. This consistency reduces validation burden and minimizes environment-specific issues.

Cloud-Native Ecosystem Integration

Kubernetes deployments integrate seamlessly with the broader cloud-native ecosystem. Organizations can leverage powerful observability tools like Prometheus for metrics collection, Grafana for visualization, and Elasticsearch-Fluentd-Kibana (EFK) stack for centralized logging. These integrations provide insights into system behavior that go far beyond what proprietary monitoring solutions typically offer.

Service mesh technologies like Istio or Linkerd can be layered onto Kubernetes clusters to provide sophisticated traffic management, security policies, and distributed tracing capabilities. These capabilities become increasingly valuable as statistical computing workflows integrate with broader enterprise data pipelines and microservices architectures.

Portability and Vendor Neutrality

Kubernetes provides abstraction from underlying infrastructure providers. Workloads defined in Kubernetes manifests can run on AWS, Azure, Google Cloud, on-premises data centers, or hybrid combinations with minimal modification. This portability provides strategic flexibility and competitive leverage when negotiating with cloud providers.

Organizations are not locked into a single vendor's roadmap or pricing structure. If Kubernetes-native alternatives to SLC emerge, or if organizational strategy shifts toward different statistical computing tools, the orchestration platform remains constant while only the containerized workloads change.

Fine-Grained Resource Control

Kubernetes enables precise resource allocation at the pod level, allowing multiple workloads to safely share infrastructure with strong isolation guarantees. Organizations can pack diverse workload types—interactive analysis sessions, scheduled batch jobs, and

long-running computations—onto the same cluster while preventing resource contention through resource requests and limits.
Horizontal pod autoscaling and cluster autoscaling enable dynamic capacity adjustment that closely matches actual demand. During overnight batch processing windows, the cluster can dynamically scale up to handle the load, then scale down during business hours when interactive workloads dominate. This elasticity can significantly reduce infrastructure costs compared to static VM-based deployments.

Disadvantages

Steep Learning Curve and Specialized Skills

Kubernetes operational complexity is substantial and should not be underestimated. Administrators must develop deep expertise across multiple domains including container technologies, networking models, storage orchestration, and security policies.

Understanding pod lifecycle states, init containers, sidecar patterns, and resource quality-of-service classes requires significant investment in training and hands-on experience.

The abstraction layers that make Kubernetes powerful also make troubleshooting challenging. When problems occur, administrators must understand whether issues originate in the application layer, container runtime, Kubernetes control plane, underlying networking infrastructure, or storage subsystem. This diagnostic complexity exceeds what most pharmaceutical IT organizations encounter with traditional enterprise applications.

Complex Initial Setup and Image Management

Building container images for statistical computing workloads involves substantial trial and error. Administrators must start with base Linux distributions and incrementally add all dependencies required by SLC, R packages, Python libraries, and OS-level utilities. The author's experience demonstrates the frustration of discovering missing dependencies only when specific code paths execute in production, requiring image rebuilds and redeployment cycles.

Unlike the SLC Hub where the vendor has pre-integrated components, Kubernetes deployments require organizations to become experts in their own software stack. Every OS library, environment variable configuration, and inter-component dependency becomes the organization's responsibility. This burden is particularly heavy during initial deployment and when upgrading to new versions of SLC or statistical packages.

Networking Complexity

Kubernetes networking introduces concepts foreign to traditional IT operations. Administrators must understand pod networking models, cluster DNS, service discovery mechanisms, ingress controllers, and network policies. Troubleshooting connectivity

issues requires familiarity with tools like tcpdump, Wireshark, and Kubernetes-specific networking debugging techniques.

Organizations must make architectural decisions about ingress controllers (nginx, Traefik, HAProxy), service mesh deployment, and network policy enforcement that have significant implications for performance, security, and operational complexity. These decisions require expertise that many pharmaceutical IT departments may not have developed, as their experience centers on traditional load balancers and firewall rules.

No Vendor Support for Complete Stack

While Kubernetes itself has strong community support and commercial distributions offer enterprise support, no single vendor takes responsibility for the complete statistical computing stack.

Organizations must separately manage relationships and support contracts for the container runtime, Kubernetes distribution, storage solution, networking components, and the SLC/R/Python software layers.

When integration issues arise between these components, organizations cannot escalate to a single vendor to resolve the problem. Instead, they must diagnose which layer is responsible and engage the appropriate support channel. This fragmentation increases risk during critical validation activities or production incidents.

Ongoing Maintenance Burden

Kubernetes clusters require continuous maintenance that extends beyond application updates. Organizations must manage Kubernetes version upgrades, certificate rotations, etcd backups, node operating system patching, and container runtime updates. Each of these maintenance activities carries risk of service disruption and requires careful planning and testing.

The rapid pace of Kubernetes releases means that supported versions age quickly. Organizations must establish processes for regularly upgrading clusters to remain on supported versions, a task that is operationally complex and must be validated in regulated environments. This upgrade cadence often exceeds what pharmaceutical IT departments are accustomed to with traditional enterprise software that may remain on the same major version for years.

Decision Framework

Organizations should consider several key factors when choosing between SLC Hub and Kubernetes deployment models:

Choose SLC Hub when:

- The organization lacks cloud-native expertise and cannot readily acquire it

- Time to production is critical and validation timelines are tight
- The IT department follows traditional enterprise software management practices
- Single-vendor accountability is important for compliance and audit purposes
- The primary workload is SAS language processing with modest Python and R integration
- Infrastructure is primarily on-premises with no near-term cloud migration plans

Choose Kubernetes when:

- The organization has existing Kubernetes expertise or strategic commitment to cloud-native technologies
- Infrastructure portability across cloud providers is a strategic priority
- Integration with modern observability and DevOps tooling is important
- The organization wants to avoid vendor lock-in and maintain architectural flexibility
- Workloads are highly dynamic with variable resource demands
- The statistical computing platform must integrate tightly with broader microservices architectures

Hybrid Considerations:

Some organizations may benefit from a phased approach, beginning with SLC Hub to establish statistical computing capabilities quickly while simultaneously building Kubernetes expertise through other initiatives. Once cloud-native skills mature across the organization, teams can evaluate migration to Kubernetes-based deployment if strategic factors make the transition valuable.

Alternatively, organizations might deploy SLC Hub for production workloads requiring maximum stability and vendor support while using Kubernetes clusters for development and experimental workloads where operational complexity is more acceptable.

The choice between these architectures represents a fundamental decision about organizational capabilities, risk tolerance, and strategic direction rather than a purely technical comparison of features. Both models can successfully support validated statistical computing environments for clinical trials, but they demand very different organizational competencies and operational practices.

Complete Artifact Inventory for SLC Deployment for Kubernetes

Persistent Volume (PV) Definitions

```
# slc-data-pv-files
Purpose: User data storage
Capacity: 50Gi
Access Mode: ReadWriteMany
Storage Class: azurefile-existing
CSI Driver: file.csi.azure.com
Volume Handle: slc-data-csi
Share Name: slcdata

# slc-pgm-pv-files
Purpose: User program storage
Capacity: 50Gi
Access Mode: ReadWriteMany
Storage Class: azurefile-existing
CSI Driver: file.csi.azure.com
Volume Handle: slc-pgm-csi
Share Name: slcpgm
```

Persistent Volume Claims (PVC)

- **slc-data-pvc**: Binds to slc-data-pv-files
- **slc-pgm-pvc**: Binds to slc-pgm-pv-files

Storage Classes

- **azurefile-existing**: For Azure Files integration
- **Mount Options**: dir_mode=0777, file_mode=0777, uid=1000, gid=1000

Secret Management

```
# azure-files-secret
Type: Opaque
Data:
  - azurestorageaccountname: <base64-encoded>
  - azurestorageaccountkey: <base64-encoded>
```

Container Image Artifacts

Base Container Image: SLC Worker

Summarized Dockerfile definition:

```
FROM rockylinux:8
```

Components for requirements.txt file

- Altair SLC 2026
- R with validated packages (foreign, haven, tidyverse, dplyr, tidyr, stringr, lubridate, ggplot2)

- Python 3 with core libraries
- OpenSSH Server for remote access
- ODBC drivers (Simba BigQuery, Microsoft SQL Server)

Key Paths:
SLC: `/opt/altair/slc/2026`
R: `/usr/lib64/R`
Python: `/usr/lib/python3`

Build Process:

- Multi-stage build for size optimization
- SLC License key injection during build
- SSH key configuration for user access
- Environment variable setup via altairslcenv.sh

Container Registry

- **Azure Container Registry (ACR)**: slcimages
- **Image Tag Strategy**: slc-image:latest, slc-image
- **Image Scanning**: Enabled for vulnerability detection

Kubernetes Deployment Artifacts

Deployment Configuration: slc-deployment.yaml

```
Metadata:
  Name: slc-deployment
  Labels: app=slc

Spec:
  Replicas: 2 (scalable)
  Container:
    - Image: slcimages.azurecr.io/slc-image:latest
    - Resources:
        Requests: 4Gi memory, 2 CPU
        Limits: 8Gi memory, 4 CPU
    - Volume Mounts:
        - /slc_data (from slc-data-pvc)
        - /slc_pgm (from slc-pgm-pvc)
    - Environment:
        - PYTHONHOME: /usr/lib/python3
        - RHOME: /usr/lib64/R
```

Service Configuration: slc-service.yaml

```
Type: LoadBalancer
Ports:
  - 8888 (Jupyter/Hub access)
  - 22 (SSH access)
Selector: app=slc
```

```
External IP: Auto-assigned by cloud provider
```

Service Account & RBAC

```
# slc-user ServiceAccount
ClusterRole: edit
ClusterRoleBinding: slc-user-binding
Authentication Token: slc-user-token (long-lived)
```

Configuration Management Artifacts

Location: `/opt/altair/slc/2026/`

1. **altairslc.cfg** (Vendor-supplied - DO NOT MODIFY)
   - Global SLC options
   - License validation settings
   - SASAUTOS base path: !wpshome/sasmacro

2. **altairslc_local.cfg** (Customer modifications)

   -AUTOEXEC /opt/slc_autocall/autoexec.sas
   -sasautos /opt/slc_autocall
   -workterm <termination options>

3. altairslcenv.sh (Environment variable injection)

```
#!/bin/bash
# Executed at SLC initialization

# Library paths
export
LD_LIBRARY_PATH="/opt/altair/slc/2026/bin:/opt/simba/googlebigque
ryodbc/lib"

# ODBC configuration
export ODBCINI=/opt/altair/slc/2026/etc/odbc.ini
export ODBCSYSINI=/opt/altair/slc/2026/etc
export ODBCINSTINI=/opt/altair/slc/2026/etc/odbcinst.ini

# Cloud credentials
export GOOGLE_APPLICATION_CREDENTIALS=/home/user/.gcp/service-
account.json

# SSL certificates
  export SSL_CERT_FILE=/etc/ssl/certs/ca-certificates.crt
```

Worker Node Configuration

Worker Node Specs:
- Minimum: 3 nodes for HA
- Recommended: 16+ nodes for production

- CPU: Sized to match SAS Grid utilization
- Memory: Sized to match SAS Grid utilization
- Storage: Local temp space + network mounts

Network and Security Artifacts

- **Hub Server Access**: HTTPS (443) for web portals
- **Worker Node Access**: HTTPS for Hub communication
- **SSH Access:** Port 22 via LoadBalancer
- **Database Access**: PostgreSQL (5432), ODBC connections

SSL/TLS Certificates

Certificate Locations:

- **System CA Bundle**: `/etc/ssl/certs/ca-certificates.crt`
- **Custom certificates**: `/etc/ssl/certs/` (mounted as ConfigMap)

SSH Keys

- **Location**: `/home/user/.ssh/`
- **Public Key**: id_rsa.pub (deployed in container)
- **Private Key**: Stored securely on client machines
- **Permissions**: 600 for private key, 644 for public key

Kubernetes Secrets

- azure-storage-secret (storage account credentials)
- azure-files-secret (file share access)
- ssh-public-key (user SSH public keys)
- slc-user-token (K8s API access token)
- acr-auth (container registry credentials)

Monitoring and Logging Artifacts

Container Logs:

- SLC logs: `/var/log/slc/`
- Application logs: `/var/log/application/`
- System logs: Captured by Kubernetes (kubectl logs)

Centralized Logging:

- syslog/rsyslog configuration
- Log aggregation (ELK, Splunk, CloudWatch)—not implemented

Monitoring Endpoints

- Resource Usage: CPU, Memory, Disk I/O per pod
- Session Metrics: Active users, running jobs
- Storage Metrics: PV utilization, IOPS
- Network Metrics: Ingress/Egress traffic

Validation Framework

At present, Siemens does not provide pre-validated SLC images, customers must perform their own validation. This section infers validation requirements based on publicly available methodologies.

Installation Qualification (IQ)
Objective: Verify correct installation of SLC software

Documentation Requirements:

- Installation logs and timestamps
- Software version verification
- License validation
- File integrity checks (checksums)

- Directory structure validation

Operational Qualification (OQ)
The author relied on his previously created SAS language programs and created additional programs to conduct this test.

Objective: Verify SLC functions correctly according to specifications
Based on OQ Report (slcoq.pdf): The provided SAS Operational Qualification report shows:

- **Components Tested**: BASE, GRAPH, STAT
- **Tests Executed**: 130
- **Tests Passed**: 129
- **Tests Failed**: 1 (tstcat01)

SLC OQ Testing Strategy: Since SLC is SAS-compatible, perform similar testing:

1. **Base Component Tests**:

- Data step processing (test base suite)
- Procedure functionality (PROC MEANS, FREQ, SQL, etc.)
- Format processing
- Macro processing
- ODS output

2. **Statistical Procedures** (tststat suite):
   - ANOVA, regression, GLM procedures
   - Categorical data analysis
   - Mixed models, survival analysis
   - Descriptive statistics

3. **Graphics Component**:
   - PROC GPLOT, GCHART functionality
   - ODS graphics output

Run OQ test suite

```
/opt/altair/slc/2026/bin/slc -sysin /path/to/oq_tests.sas -log
oq_results.log
```

Custom OQ tests

```
/opt/altair/slc/2026/bin/slc -sysin custom_validation_tests.sas
```

Acceptance Criteria:
- 95% test pass rate
- Known failures documented and justified
- All critical statistical procedures functioning correctly

Performance Qualification (PQ)
Objective: Verify SLC performs adequately in production environment

Test Scenarios:
1. **Baseline Performance**:
   - Standard statistical analysis runtime
   - Large dataset processing (>1GB)
   - Concurrent user simulation
2. **Resource Utilization**:
   - Memory consumption per session
   - CPU utilization patterns
   - Disk I/O throughput
3. **Scalability Testing**:
   - Pod autoscaling verification
   - Load balancer behavior under stress

- Maximum concurrent users

Metrics Collection:

```
# Monitor pod resources
kubectl top pods -l app=slc
```

Stress testing
Use load testing tools to simulate 50+ concurrent users

R Installation Validation

Objective: Ensure correct R version and package installation
Test Plan:

1. **R Version Verification**:

```
R.version.string
```

Expected: R version 4.x.x

2. **Package Installation Verification**:

```
installed.packages()[, c("Package", "Version", "Built")]

# Verify specific validated packages
library(tidyverse)
library(haven)
library(foreign)
```

3. **Package Integrity Check**:
    - Verify package checksums against CRAN
    - Document package versions and sources
    - Lock package versions (renv/packrat)

R Functional Testing
Test Categories:

1. **Data Import/Export**:
    - SAS dataset reading (haven package)
    - CSV, Excel processing
    - Database connectivity (DBI, odbc)

2. **Statistical Functions**:
    - Summary statistics (mean, median, sd)
    - Linear models (lm, glm)
    - Hypothesis testing (t.test, chi-square)

3. **Reproducibility Tests**:

- Set random seeds and verify identical results
- Cross-validate against SAS/SLC results

Example Test Script:

```
# R Validation Test Suite
test_data_import <- function() {
  # Test haven SAS file reading
  df <- haven::read_sas("test_data.sas7bdat")
  stopifnot(nrow(df) == 1000)
}

test_statistical_functions <- function() {
  # Test basic statistics
  x <- rnorm(100, mean=50, sd=10)
  stopifnot(abs(mean(x) - 50) < 5)
}
```

Run tests

```
test_data_import()
test_statistical_functions()
```

Python Installation Validation

Test Plan:

1. **Python Version**:

```
 python3 --version
```

Expected: Python 3.9+

2. **Package Management**:

```
pip list --format=freeze > python_packages.txt
```

3. **Virtual Environment**:

```
 # Verify isolated environments
  python3 -m venv /opt/validated_envs/clinical_v1
```

Python Functional Testing

Test Categories:

1. **Data Science Libraries**:
    - pandas (data manipulation)
    - numpy (numerical computing)
    - scipy (scientific computing)
    - scikit-learn (machine learning)

2. **SAS Software Integration**:

- pandas_sas7bdat (SAS file reading)

3. **Reproducibility**:

```
import numpy as np
np.random.seed(42)
```

Image Build Validation

Verification Steps:

Build Reproducibility:

```
# Tag images with unique identifiers
docker build -t slc-image:v1.0.0-$(git rev-parse --short HEAD) .
```

Verify image checksums

```
docker images --digests slc-image
```

Component Verification:

Inspect container contents

```
docker run slc-image:v1.0.0 /opt/altair/slc/2026/bin/slc -version
docker run slc-image:v1.0.0 R --version
docker run slc-image:v1.0.0 python3 --version
```

Security Scanning:

Scan for vulnerabilities

```
docker scan slc-image:v1.0.0
```

Or use Azure Container Registry scanning

```
az acr check-health -n slcimages
```

Container Runtime Validation

1. **Environment Variables**:

```
kubectl exec -it slc-pod -- env | grep -E
"(LD_LIBRARY_PATH|ODBCINI|PYTHONHOME)"
```

2. **Volume Mounts**:

```
kubectl exec -it slc-pod -- ls -la /slc_data /slc_pgm
```

3. **Service Connectivity**:

```
# Test database connections
kubectl exec -it slc-pod -- /opt/altair/slc/2026/bin/slc -sysin
test_db_connection.sas
```

Traceability Matrix

Requirement	Test ID	IQ	OQ	PQ	Status
SLC Base Functions	TC-001	✓	✓	✓	PASS
R Statistical Functions	TC-002	✓	✓	✓	PASS
Python Data Processing	TC-003	✓	✓	✓	PASS
ODBC Connectivity	TC-004	✓	✓	✓	PASS
Container Scalability	TC-005	✓	N/A	✓	PASS

Change Control Process

Version Control Strategy

- **Git Repository**: Store all configuration files (YAML, Dockerfiles, scripts)
- **Branching Model**: GitFlow (dev, staging, production branches)
- **Commit Messages**: Follow conventional commits standard
- **Pull Requests**: Required for all changes, with peer review

Image Versioning

Versioning Scheme: MAJOR.MINOR.PATCH-BUILD
Example: slc-image:2.0.1-20250815

MAJOR: SLC version change
MINOR: New R/Python packages
PATCH: Configuration updates
BUILD: Build timestamp/commit hash

Deployment Process

1. Build new image with version tag
2. Deploy to DEV environment
3. Run automated validation tests
4. Deploy to STAGING for UAT
5. Obtain approval from stakeholders
6. Deploy to PRODUCTION with rollback plan
7. Monitor for 24 hours
8. Document deployment in change log

User Interface Components

Development Clients

- **Altair Analytics Workbench**: Primary IDE for SAS/SLC development
- **VSCode with SLC Extension**: SSH-based remote development
- **MS Excel with SLC Plugin**: Spreadsheet integration (Hub only)
- **Citrix/Desktop**: Remote access to development environment

SLC Hub Web Interfaces

- **User Portal**: Job submission, monitoring, results retrieval
- **Admin Portal**: User management, resource allocation, monitoring
- **Mobile Access**: Responsive web interface for job monitoring

VSCode Configuration for Remote SLC

```
// .vscode/settings.json
{
  "remote.SSH.remotePlatform": {
    "slc-hub.company.com": "linux"
  },
  "slc.executablePath": "/opt/altair/slc/2026/bin/slc",
  "slc.configPath": "/opt/altair/slc/2026/altairslc.cfg"
}
```

Integration Points

External Schedulers

- **Supported**: Control-M, AutoSys, etc-d, Jenkins
- **Integration Method**: REST API calls to Hub scheduler
- **Authentication**: Service account with API token
- **Job Submission**: HTTP POST with SAS program payload

Data Sources

- **Databases**: Oracle, SQL Server, PostgreSQL, Teradata, Netezza, BigQuery, RedShift, SAP/IQ, DB2, et.al.
- **Cloud Data Warehouses**: BigQuery, Redshift, Snowflake
- **Hadoop/Spark**: HDFS, Hive, Impala
- **File Formats**: SAS7BDAT, CSV, Excel, Parquet, JSON, Iceberg

ODBC Configuration Pattern:

```
[DataSourceName]
Driver=DriverName
Server=hostname
Port=port
```

```
Database=dbname
# Additional driver-specific options
```

Authentication Integration

LDAP/Active Directory:

- **Protocol**: LDAPS (LDAP over SSL)
- **Hub Configuration**: Auth Manager with AD integration
- **User Provisioning**: Automatic via LDAP groups
- **Role Mapping**: LDAP groups → Hub roles

Compliance and Audit

Audit Trail Requirements (21 CFR Part 11)

- Capture and Retain:
- User authentication events (login/logout)
- Job submissions with timestamps
- Code deployment activities
- Configuration changes
- Data access (read/write to validated data)
- System administration activities

Audit Log Format

Timestamp | User ID | Action | Resource | Result | Details
2025-10-15 14:23:01 | jdoe | JOB_SUBMIT | analysis_v2.sas | SUCCESS | Job ID 12345

Electronic Signatures
When Required:

- Final analysis report approval
- Validation document approval
- Critical configuration changes

Electronic Signatures
Implementation (SLC Hub):

- Two-factor authentication
- Digital signature with timestamp
- Non-repudiation through cryptographic hash
- Audit trail linkage to signed document

Signature Components:
Signed By: Jane Doe (jdoe@company.com)

Meaning: Approved for Production Use
Date/Time: 2025-10-15 14:23:01 UTC
Hash: SHA-256: a3f5c8e9d2b1...
Reason: Statistical Analysis Plan Approval

Regulatory Inspection Readiness
Documentation Package for Inspection:

1. **System Architecture Diagram** (this document)
2. **Validation Package** (IQ/OQ/PQ documentation)
3. **Change Control Log** (all system changes with impact assessments)
4. **User Access Matrix** (who has access to what)
5. **Audit Trail Reports** (sample of system activities)
6. **SOPs** (Standard Operating Procedures for system use)
7. **Disaster Recovery Plan** (with test results)
8. **Security Assessment** (vulnerability scans, penetration tests)

Key Inspection Questions - Prepared Responses:

Q: "How do you ensure data integrity?" A: Multi-layered approach:
- Immutable container images with versioning
- POSIX-compliant file system with checksums
- Database transaction logs
- Audit trails for all data access
- Regular integrity checks and validation

Q: "How do you control access to the system?" A: Role-based access control (RBAC):
- LDAP/AD integration for centralized authentication
- Kubernetes RBAC for infrastructure access
- SLC Hub roles for application-level permissions
- Principle of least privilege enforced
- Regular access reviews (quarterly)

Q: "What is your change control process?" A: Formal change control procedure:
- All changes tracked in Git with approval workflow
- Impact assessment for every change
- Validation impact analysis (triggers revalidation if needed)
- Documented testing before production deployment
- Rollback capability for all changes

Q: "How do you ensure system availability?" A: High availability architecture:
- Redundant components (3+ nodes for all critical services)
- Automated failover with Kubernetes
- Regular disaster recovery testing (quarterly)

- 24/7 monitoring with alerting
- RTO/RPO defined and tested

Performance Optimization

Container Resource Tuning

```
# Development/Test Environment
resources:
  requests:
    memory: "2Gi"
    cpu: "1"
  limits:
    memory: "4Gi"
    cpu: "2"

# Production Environment
resources:
  requests:
    memory: "4Gi"
    cpu: "2"
  limits:
    memory: "8Gi"
    cpu: "4"

# Heavy Statistical Workloads
resources:
  requests:
    memory: "8Gi"
    cpu: "4"
  limits:
    memory: "16Gi"
    cpu: "8"
```

Optimization Techniques:
Horizontal Pod Autoscaling (HPA):

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: slc-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: slc-deployment
  minReplicas: 2
  maxReplicas: 20
  metrics:
```

```
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

Pod Disruption Budgets:

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: slc-pdb
spec:
  minAvailable: 1
  selector:
    matchLabels:
      app: slc
```

Node Affinity for Performance:

```
affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: workload-type
            operator: In
            values:
            - statistical-computing
```

Storage Performance Optimization

GPFS/File System Tuning:

- **Block Size**: Optimize for typical file sizes (4MB for large datasets)
- **Prefetch**: Enable read-ahead for sequential access patterns
- **Write Cache**: Enable for improved write performance
- **Parallel I/O**: Configure for multi-threaded SAS/SLC jobs

Azure Files Performance Tiers (if using cloud storage):

- **Premium**: For high-IOPS workloads (up to 100,000 IOPS)
- **Transaction Optimized**: For metadata-heavy operations
- **Hot**: For frequently accessed data
- **Cool**: For archival/backup data

Storage Monitoring:

```
# Monitor PVC usage
kubectl exec -it slc-pod -- df -h /slc_data /slc_pgm
```

Security Hardening & Container Security Best Practices

Run as Non-Root User:

```
# In Dockerfile
RUN useradd -m -d /home/slcuser -s /bin/bash slcuser
USER slcuser
```

Read-Only Root Filesystem:

```
securityContext:
  readOnlyRootFilesystem: true
  runAsNonRoot: true
  runAsUser: 1000
```

Drop Unnecessary Capabilities:

```
securityContext:
  capabilities:
    drop:
    - ALL
    add:
    - NET_BIND_SERVICE  # Only if needed
```

Network Policies:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: slc-network-policy
spec:
  podSelector:
    matchLabels:
      app: slc
  policyTypes:
  - Ingress
  - Egress
  ingress:
```

```
  - from:
    - podSelector:
        matchLabels:
          role: slc-hub
    ports:
    - protocol: TCP
      port: 22
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: postgres
    ports:
    - protocol: TCP
      port: 5432
```

Secrets Management Best Practices:

1. **Never Hardcode Secrets** in code or Dockerfiles
2. **Use Kubernetes Secrets** with encryption at rest
3. **Rotate Secrets Regularly** (90-day cycle recommended)
4. **Use External Secrets Managers** for sensitive credentials (Azure Key Vault, AWS Secrets Manager, HashiCorp Vault)

Example: Azure Key Vault Integration:

```
apiVersion: v1
kind: SecretProviderClass
metadata:
  name: azure-keyvault-slc
spec:
  provider: azure
  parameters:
    keyvaultName: "slc-keyvault"
    objects: |
      array:
        - |
          objectName: db-password
          objectType: secret
        - |
          objectName: service-account-key
          objectType: secret
```

Network Security

TLS/SSL Everywhere:

- **Hub Web Interfaces**: HTTPS only with valid certificates
- **Database Connections**: TLS-enabled connections
- **LDAP**: Use LDAPS (LDAP over SSL)
- **Internal Communication**: mTLS (mutual TLS) between services

Certificate Management:

```
# Using cert-manager for automated certificate management
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: slc-hub-tls
spec:
  secretName: slc-hub-tls-secret
  issuerRef:
    name: letsencrypt-prod
    kind: ClusterIssuer
  dnsNames:
  - slc-hub.company.com
```

Firewall Rules

Inbound Rules:

- **HTTPS** (443) from corporate network
- **SSH** (22) from bastion host only
- **PostgreSQL** (5432) from SLC pods only

Outbound Rules:

- **HTTPS** (443) to internet (for package updates)
- **LDAPS** (636) to Active Directory
- Database ports to data sources

Storage Cost Optimization

Tiered Storage Strategy:

1. **Hot Tier** (Premium/High-Performance):
    - Active analysis datasets
    - Current user workspaces
    - Retention: 30 days
2. **Warm Tier** (Standard):
    - Completed analysis outputs
    - Reference datasets
    - Retention: 90 days
3. **Cold Tier** (Archive):
    - Historical datasets
    - Regulatory retention (7 years)

- Cost-optimized storage (S3 Glacier, Azure Archive)

Compute Cost Optimization Strategies:

1. **Spot/Preemptible Instances**: For non-critical worker nodes (dev/test)
2. **Reserved Instances**: For baseline production capacity (20-30% cost savings)
3. **Autoscaling**: Scale down during off-hours

```
# Example: Scale down at night
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: slc-hpa-night
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: slc-deployment
  minReplicas: 1  # Reduced from 2 during off-hours
  maxReplicas: 10
```

Conclusion and Recommendations

This paper presented a comprehensive Kubernetes framework for deploying and managing validated statistical computing environments (SLC, R, Python) that meet FDA regulatory requirements. The approach provides:

✅ **Regulatory Compliance**: Validated environments with proper change control
✅ **High Availability**: Redundant architecture with automated failover
✅ **Scalability**: Horizontal scaling to match workload demands
✅ **Governance**: Centralized control with audit trails
✅ **Cost Efficiency**: Optimized resource utilization
✅ **Modern DevOps**: Infrastructure as Code with GitOps workflows

Implementation Roadmap

Phase 1: Planning and Design (Months 1-2)

- Finalize architecture design
- Obtain infrastructure approvals
- Provision Kubernetes cluster
- Set up PostgreSQL and object storage

Phase 2: Build and Validate (Months 3-4)

- Build and test container images

- Configure SLC Hub
- Execute IQ/OQ/PQ validation
- Document validation package

Phase 3: Pilot (Months 5-6)
- Deploy pilot environment
- Onboard pilot users
- Collect feedback and metrics
- Refine configuration

Phase 4: Production Deployment (Months 7-8)
- Scale to production capacity
- User training program
- Migrate first production workloads
- Establish support processes

Phase 5: Full Migration (Months 9-12)
- Phased migration of all users
- Decommission SAS Grid
- Optimize and tune
- Continuous improvement

Success Metrics

Technical Metrics:
- System uptime >99.5%
- Job completion success rate >98%
- Average job execution time ≤ SAS Grid baseline
- Resource utilization 60-80% (efficient, not over/under-provisioned)

Business Metrics:
- User satisfaction score >80%
- 100% of validation documentation complete
- Zero regulatory findings related to system
- Total Cost of Ownership (TCO) reduction vs SAS Grid

Compliance Metrics:
- 100% audit trail coverage
- Zero data integrity incidents
- All changes properly documented and approved
- Successful regulatory inspections

Appendix A: Artifact Checklist

Use this checklist to ensure all artifacts are properly managed:

Storage Artifacts

- Persistent Volumes defined and documented
- Persistent Volume Claims created
- Storage Classes configured
- Azure/Cloud storage secrets created
- GPFS/File system mounted and tested
- Backup procedures documented and tested

Container Artifacts

- Dockerfile version-controlled in Git
- Container image built and tagged
- Image pushed to container registry
- Vulnerability scan completed
- Image documented in inventory

Kubernetes Artifacts

- Deployment YAML created and applied
- Service YAML created and applied
- ConfigMaps for configuration files
- Secrets for sensitive data
- RBAC policies defined
- Network policies configured
- Resource quotas set
- Autoscaling policies configured

Configuration Artifacts

- altairslc.cfg baseline documented
- altairslc_local.cfg customizations tracked
- altairslcenv.sh version-controlled
- autoexec.sas deployed and tested
- SASAUTOS macros catalogue
- ODBC configuration files documented

Validation Artifacts

- Validation Plan approved
- IQ test protocol and results
- OQ test protocol and results
- PQ test protocol and results
- Validation Summary Report signed
- Traceability Matrix complete

- Deviations documented and justified

Operational Artifacts

- Monitoring dashboards configured
- Alerting rules defined
- Runbooks for common issues
- Disaster recovery plan documented
- DR test results on file
- Backup and restore procedures tested
- User access matrix maintained
- Audit log retention configured

Compliance Artifacts

- SOPs for system use
- Change control log maintained
- Audit trail reports available
- Electronic signature configuration
- Regulatory inspection package prepared
- Security assessment completed

Appendix B: Glossary

Term	Definition
ACR	Azure Container Registry - managed Docker registry service
ADLS2	Azure Data Lake Storage Gen2 - scalable cloud storage
AUTOEXEC	Automatically executed SAS program at session initialization
CFR Part 11	FDA regulation for electronic records and signatures
CSI	Container Storage Interface - Kubernetes storage plugin standard
GPFS	General Parallel File System (IBM Spectrum Scale)
HPA	Horizontal Pod Autoscaler - automatic pod scaling in Kubernetes
IQ	Installation Qualification - validates correct software installation
ODBC	Open Database Connectivity - standard database access API
OQ	Operational Qualification - validates system functions correctly
PQ	Performance Qualification - validates system performs in production
PV	Persistent Volume - Kubernetes storage abstraction
PVC	Persistent Volume Claim - user request for storage
RBAC	Role-Based Access Control - authorization model
RPO	Recovery Point Objective - maximum acceptable data loss
RTO	Recovery Time Objective - maximum acceptable downtime
SASAUTOS	SAS Autocall Macro Library search path
SLC	SAS Language Compiler - Altair's SAS-compatible software
SPOF	Single Point of Failure - component whose failure stops entire system

Appendix C: Sample Docker File for Image build of SLC, R, and Python

```
FROM rockylinux:8 AS builder
# Install build dependencies
RUN dnf -y update && \
    dnf -y install wget \
      curl \
      bzip2 \
      ca-certificates \
      gcc \
      gcc-c++ \
      make \
      zlib-devel \
      openssl-devel \
      readline-devel \
      sqlite-devel \
      libxml2-devel \
      libcurl-devel \
      xz-devel \
      pcre-devel \
      epel-release && \
    dnf clean all && \
    rm -rf /var/cache/dnf

# Create temp directory
RUN mkdir -p /temp

# Install font dependencies and R with required packages
RUN dnf -y install dnf-plugins-core && \
    dnf config-manager --set-enabled powertools && \
    dnf -y install openblas openblas-devel \
      fontconfig-devel \
      freetype-devel \
      harfbuzz-devel \
      fribidi-devel \
      libpng-devel \
      libtiff-devel \
      libjpeg-devel \
      bzip2-devel \
      pkg-config && \
    dnf -y install --nobest R && \
    dnf clean all && \
    rm -rf /var/cache/dnf && \
    Rscript -e "install.packages(c('foreign', 'haven', 'tidyverse',
'dplyr', 'tidyr', 'stringr', 'lubridate', 'ggplot2'),
repos='http://cran.rstudio.com/', Ncpus=4)"

# Install Python and pip for Python dependencies
```

```
RUN dnf -y install python3-devel python3-pip && \
    python3 -m pip install --upgrade pip setuptools wheel && \
    dnf clean all && \
    rm -rf /var/cache/dnf

# Copy the SLC RPM package and license
COPY altairslc-5.24.3.0.1528_ga_release-1.x86_64.rpm
/temp/altairslc.rpm
COPY license.key /temp/license.key

# Copy requirements.txt for Python dependencies
COPY requirements.txt /temp/requirements.txt

# Install Python dependencies from requirements.txt
RUN python3 -m pip install -r /temp/requirements.txt

# Install and initialize SLC
RUN rpm -ivh /temp/altairslc.rpm

# Apply the license - without expecting output
RUN echo "Setting up SLC license..." && \
    /opt/altair/slc/2024/bin/wps -setinit /temp/license.key && \
    echo "License setup completed" && \
    # Copy license for runtime stage
    cp /temp/license.key /opt/altair/slc/2024/license.key && \
    # Clean up
    rm -f /temp/altairslc.rpm

# Runtime Stage - also using Rocky Linux 8 for runtime
FROM rockylinux:8

# Install runtime packages with essential system utilities
RUN dnf -y update && \
    dnf -y install epel-release && \
    dnf -y install dnf-plugins-core && \
    dnf config-manager --set-enabled powertools && \
    dnf -y install openblas \
      fontconfig \
      freetype \
      harfbuzz \
      fribidi \
      libpng \
      libtiff \
      libjpeg && \
    dnf -y install python3 python3-pip python3-libs glibc-langpack-en
&& \
    # Install OpenSSH server and important utilities for
troubleshooting
    dnf -y install openssh-server openssh-clients \
      procps-ng \
      net-tools \
      iproute \
```

```
      which \
      less \
      findutils && \
    # Generate host keys
    ssh-keygen -A && \
    # Configure SSHD
    mkdir -p /var/run/sshd && \
    # Create user for SSH access
    useradd -m -d /home/trb -s /bin/bash trb && \
    mkdir -p /home/trb/.ssh && \
    chmod 700 /home/trb/.ssh && \
    touch /home/trb/.ssh/authorized_keys && \
    chmod 600 /home/trb/.ssh/authorized_keys && \
    chown -R trb:trb /home/trb/.ssh && \
      # Install SQL Server client libraries
      dnf -y install freetds && \
    # Clean up
    dnf clean all && \
    rm -rf /var/cache/dnf

# Set locale and environment variables
ENV LANG=en_US.UTF-8 \
    LC_ALL=en_US.UTF-8 \
    PYTHONHOME=/usr \
    RHOME=/usr/lib64/R \

PATH=/opt/altair/slc/2024/bin:/opt/altair/slc/2023/bin:/usr/local/bin:
/usr/bin:/usr/local/sbin:/usr/sbin

# Create directories
RUN mkdir -p /data /temp /slc_data /slc_pgm

# Copy requirements.txt for Python dependencies in runtime stage
COPY requirements.txt /temp/requirements.txt

# Install Python dependencies in runtime stage
RUN python3 -m pip install -r /temp/requirements.txt

# Copy R packages and SLC from builder stage
COPY --from=builder /usr/lib64/R /usr/lib64/R
COPY --from=builder /opt/altair/slc /opt/altair/slc
COPY --from=builder /opt/altair/slc/2024/license.key /temp/license.key

# Create simplified SSH config focused on key authentication
RUN echo 'HostKey /etc/ssh/ssh_host_rsa_key' > /etc/ssh/sshd_config &&
\
    echo 'HostKey /etc/ssh/ssh_host_ecdsa_key' >> /etc/ssh/sshd_config
&& \
    echo 'HostKey /etc/ssh/ssh_host_ed25519_key' >>
/etc/ssh/sshd_config && \
    echo 'SyslogFacility AUTHPRIV' >> /etc/ssh/sshd_config && \
    echo 'LogLevel DEBUG3' >> /etc/ssh/sshd_config && \
```

```
    echo 'PermitRootLogin yes' >> /etc/ssh/sshd_config && \
    echo 'StrictModes no' >> /etc/ssh/sshd_config && \
    echo 'MaxAuthTries 6' >> /etc/ssh/sshd_config && \
    echo 'MaxSessions 10' >> /etc/ssh/sshd_config && \
    echo 'PasswordAuthentication no' >> /etc/ssh/sshd_config && \
    echo 'PermitEmptyPasswords no' >> /etc/ssh/sshd_config && \
    echo 'PubkeyAuthentication yes' >> /etc/ssh/sshd_config && \
    echo 'AuthenticationMethods publickey' >> /etc/ssh/sshd_config &&
\
    echo 'AuthorizedKeysFile .ssh/authorized_keys' >>
/etc/ssh/sshd_config && \
    echo 'UsePAM yes' >> /etc/ssh/sshd_config && \
    echo 'X11Forwarding yes' >> /etc/ssh/sshd_config && \
    echo 'PrintMotd no' >> /etc/ssh/sshd_config && \
    echo 'Subsystem sftp /usr/libexec/openssh/sftp-server' >>
/etc/ssh/sshd_config

# Fix PAM nologin issues that prevent users from logging in
RUN sed -i 's/^account\s\+required\s\+pam_nologin.so/#account
required    pam_nologin.so/' /etc/pam.d/sshd

# Make sure there are no nologin files
RUN rm -f /run/nologin /var/run/nologin || true

# Create startup script
RUN echo '#!/bin/bash' > /start.sh && \
    echo 'echo "Starting container setup..."' >> /start.sh && \
    echo '# Fix PAM nologin issues' >> /start.sh && \
    echo 'echo "Checking and removing any nologin files..."' >>
/start.sh && \
    echo 'rm -f /run/nologin /var/run/nologin || true' >> /start.sh &&
\
    echo 'sed -i "s/^account\s\+required\s\+pam_nologin.so/#account
required    pam_nologin.so/" /etc/pam.d/sshd' >> /start.sh && \
    echo '# Verify license' >> /start.sh && \
    echo 'echo "Verifying SLC license..."' >> /start.sh && \
    echo 'if [ -f "/opt/altair/slc/2024/license.key" ]; then' >>
/start.sh && \
    echo '  echo "SLC license found"' >> /start.sh && \
    echo 'else' >> /start.sh && \
    echo '  echo "WARNING: SLC license file not found. Attempting to
reapply..."' >> /start.sh && \
    echo '  if [ -f "/temp/license.key" ]; then' >> /start.sh && \
    echo '    /opt/altair/slc/2024/bin/wps -setinit /temp/license.key'
>> /start.sh && \
    echo '    echo "License setup attempted"' >> /start.sh && \
    echo '  else' >> /start.sh && \
    echo '    echo "License key file not found, SLC may not function
properly"' >> /start.sh && \
    echo '  fi' >> /start.sh && \
    echo 'fi' >> /start.sh && \
    echo 'echo "Checking mount points:"' >> /start.sh && \
```

```
    echo 'df -h /slc_data /slc_pgm || echo "Warning: Mount points may
not be available"' >> /start.sh && \
    echo 'echo "Creating test files:"' >> /start.sh && \
    echo 'touch /slc_data/startup_test.txt 2>/dev/null || echo
"Warning: Cannot write to /slc_data"' >> /start.sh && \
    echo 'ls -la /slc_data/startup_test.txt 2>/dev/null || echo "File
not created"' >> /start.sh && \
    echo 'echo "SLC Container Started" > /tmp/motd' >> /start.sh && \
    echo '# Handle Kubernetes Secret mounted keys' >> /start.sh && \
    echo 'if [ -d "/etc/ssh/keys" ] && [ -f
"/etc/ssh/keys/vscode_pvt_key.pub" ]; then' >> /start.sh && \
    echo '  echo "Found mounted public key, copying to
authorized_keys..."' >> /start.sh && \
    echo '  mkdir -p /root/.ssh' >> /start.sh && \
    echo '  cp /etc/ssh/keys/vscode_pvt_key.pub
/root/.ssh/authorized_keys' >> /start.sh && \
    echo '  chmod 700 /root/.ssh' >> /start.sh && \
    echo '  chmod 600 /root/.ssh/authorized_keys' >> /start.sh && \
    echo '  # For trb user' >> /start.sh && \
    echo '  mkdir -p /home/trb/.ssh' >> /start.sh && \
    echo '  cp /etc/ssh/keys/vscode_pvt_key.pub
/home/trb/.ssh/authorized_keys' >> /start.sh && \
    echo '  chmod 700 /home/trb/.ssh' >> /start.sh && \
    echo '  chmod 600 /home/trb/.ssh/authorized_keys' >> /start.sh &&
\
    echo '  chown -R trb:trb /home/trb/.ssh' >> /start.sh && \
    echo 'fi' >> /start.sh && \
    echo 'echo "Starting SSH daemon with debugging enabled..."' >>
/start.sh && \
    echo '/usr/sbin/sshd -E /var/log/sshd_debug.log -D &' >> /start.sh
&& \
    echo 'echo "SSH daemon started. Container ready."' >> /start.sh &&
\
    echo 'echo "To view SSH logs: cat /var/log/sshd_debug.log"' >>
/start.sh && \
    echo 'tail -f /dev/null' >> /start.sh && \
    chmod +x /start.sh

# Set working directory
WORKDIR /data

# Expose SSH port
EXPOSE 22

# Start services
CMD ["/start.sh"]
```

Appendix D: Requirements.txt file for Container Image

```
numpy==1.19.5
pandas==1.1.5
matplotlib==3.3.4
scipy==1.5.4
scikit-learn==0.24.2
Automat==20.2.0
Babel==2.11.0
Bottleneck==1.3.7
Brotli==1.0.9
Deprecated==1.2.13
Flask==3.0.3
GitPython==3.1.43
HeapDict==1.0.1
Jinja2==3.1.4
Markdown==3.4.1
MarkupSafe==2.1.3
Protego==0.1.16
PyDispatcher==2.0.5
PyGithub==2.4.0
PyJWT==2.10.1
PyNaCl==1.5.0
PyQt5-sip==12.13.0
PyQt5==5.15.10
PyQtWebEngine==5.15.6
PySocks==1.7.1
PyWavelets==1.7.0
PyYAML==6.0.1
Pygments==2.15.1
QDarkStyle==3.2.3
QtAwesome==1.3.1
QtPy==2.4.1
Rtree==1.0.1
SQLAlchemy==2.0.34
Scrapy==2.11.1
Send2Trash==1.8.2
Sphinx==7.3.7
Twisted==23.10.0
Unidecode==1.3.8
Werkzeug==3.0.3
aiobotocore==2.12.3
aiohappyeyeballs==2.4.0
aiohttp==3.10.5
aioitertools==0.7.1
aiosignal==1.2.0
alabaster==0.7.16
```

```
altair==5.0.1
anyio==4.2.0
appdirs==1.4.4
argon2-cffi-bindings==21.2.0
argon2-cffi==21.3.0
arrow==1.2.3
astroid==2.14.2
astropy-iers-data==0.2024.9.2.0.33.23
astropy==6.1.3
asttokens==2.0.5
async-lru==2.0.4
asyncssh==2.17.0
atomicwrites==1.4.0
attrs==23.1.0
autopep8==2.0.4
azure-core==1.32.0
azure-datalake-store==0.0.53
azure-storage-blob==12.24.1
azure-storage-file-datalake==12.18.0
bcrypt==3.2.0
beautifulsoup4==4.12.3
binaryornot==0.4.4
black==24.8.0
bleach==4.1.0
blinker==1.6.2
bokeh==3.6.0
botocore==1.34.69
cachetools==5.3.3
certifi==2024.8.30
cffi==1.17.1
chardet==4.0.0
charset-normalizer==3.3.2
click==8.1.7
cloudpickle==3.0.0
colorama==0.4.6
colorcet==3.1.0
comm==0.2.1
constantly==23.10.4
contourpy==1.2.0
cookiecutter==2.6.0
cryptography==43.0.0
cssselect==1.2.0
cycler==0.11.0
cytoolz==0.12.2
dask-expr==1.1.13
dask==2024.8.2
databricks-sdk==0.44.1
```

```
databricks-sql-connector==4.0.0
datashader==0.16.3
debugpy==1.6.7
decorator==5.1.1
defusedxml==0.7.1
diff-match-patch==20200713
dill==0.3.8
distributed==2024.8.2
docstring-to-markdown==0.11
docutils==0.18.1
et-xmlfile==1.1.0
executing==0.8.3
fastjsonschema==2.16.2
filelock==3.13.1
flake8==7.0.0
fonttools==4.51.0
frozenlist==1.4.0
fsspec==2024.6.1
gensim==4.3.3
gitdb==4.0.7
gmpy2==2.2.1
google-auth==2.38.0
greenlet==3.0.1
h11==0.14.0
h5py==3.11.0
holoviews==1.19.1
httpcore==1.0.2
httpx==0.27.0
hvplot==0.11.0
hyperlink==21.0.0
idna==3.7
imagecodecs==2023.1.23
imageio==2.33.1
imagesize==1.4.1
imbalanced-learn==0.12.3
importlib-metadata==7.0.1
incremental==22.10.0
inflection==0.5.1
iniconfig==1.1.1
intake==2.0.7
intervaltree==3.1.0
ipykernel==6.28.0
ipython-genutils==0.2.0
ipython==8.27.0
ipywidgets==7.8.1
isodate==0.6.1
isort==5.13.2
```

```
itemadapter==0.3.0
itemloaders==1.1.0
itsdangerous==2.2.0
jaraco.classes==3.2.1
jedi==0.19.1
jellyfish==1.0.1
jmespath==1.0.1
joblib==1.4.2
json5==0.9.6
jsonschema-specifications==2023.7.1
jsonschema==4.23.0
jupyter-console==6.6.3
jupyter-events==0.10.0
jupyter-lsp==2.2.0
jupyter==1.0.0
jupyter_client==8.6.0
jupyter_core==5.7.2
jupyter_server==2.14.1
jupyter_server_terminals==0.4.4
jupyterlab-pygments==0.1.2
jupyterlab-widgets==1.0.0
jupyterlab==4.2.5
jupyterlab_server==2.27.3
keyring==24.3.1
kiwisolver==1.4.4
lazy-object-proxy==1.10.0
lazy_loader==0.4
lckr_jupyterlab_variableinspector==3.1.0
linkify-it-py==2.0.0
llvmlite==0.43.0
lmdb==1.4.1
locket==1.0.0
lxml==5.2.1
lz4==4.3.2
markdown-it-py==2.2.0
matplotlib-inline==0.1.6
matplotlib==3.9.2
mccabe==0.7.0
mdit-py-plugins==0.3.0
mdurl==0.1.0
mistune==2.0.4
mkl-service==2.4.0
mkl_fft==1.3.10
mkl_random==1.2.7
more-itertools==10.3.0
mpmath==1.3.0
msal==1.25.0
```

```
msgpack==1.0.3
multidict==6.0.4
multipledispatch==0.6.0
mypy-extensions==1.0.0
mypy==1.11.2
nbclient==0.8.0
nbconvert==7.16.4
nbformat==5.10.4
nest-asyncio==1.6.0
networkx==3.3
nltk==3.9.1
notebook==7.2.2
notebook_shim==0.2.3
numba==0.60.0
numexpr==2.8.7
numpy==1.26.4
numpydoc==1.7.0
oauthlib==3.2.2
openpyxl==3.1.5
overrides==7.4.0
packaging==24.1
pandas==2.2.2
pandocfilters==1.5.0
panel==1.5.2
param==2.1.1
paramiko==2.8.1
parsel==1.8.1
parso==0.8.3
partd==1.4.1
pathspec==0.10.3
patsy==0.5.6
pexpect==4.8.0
pickleshare==0.7.5
pillow==10.4.0
pip==24.2
platformdirs==3.10.0
plotly==5.24.1
pluggy==1.0.0
ply==3.11
prometheus-client==0.14.1
prompt-toolkit==3.0.43
propcache==0.2.0
protobuf==4.25.3
psutil==5.9.0
ptyprocess==0.7.0
pure-eval==0.2.2
py-cpuinfo==9.0.0
```

```
pyOpenSSL==24.2.1
pyarrow==16.1.0
pyasn1-modules==0.2.8
pyasn1==0.4.8
pycodestyle==2.11.1
pycparser==2.21
pyct==0.5.0
pycurl==7.45.3
pydeck==0.8.0
pydocstyle==6.3.0
pyerfa==2.0.1.4
pyflakes==3.2.0
pylint-venv==3.0.3
pylint==2.16.2
pyls-spyder==0.4.0
pyodbc==5.1.0
pyparsing==3.1.2
pytest==7.4.4
python-dateutil==2.9.0.post0
python-json-logger==2.0.7
python-lsp-black==2.0.0
python-lsp-jsonrpc==1.1.2
python-lsp-server==1.10.0
python-slugify==5.0.2
pytoolconfig==1.2.6
pytz==2024.1
pyuca==1.2
pyviz_comms==3.0.2
pywin32-ctypes==0.2.2
pywin32==305.1
pywinpty==2.0.10
pyzmq==25.1.2
qstylizer==0.2.2
qtconsole==5.5.1
queuelib==1.6.2
referencing==0.30.2
regex==2024.9.11
requests-file==1.5.1
requests==2.32.3
rfc3339-validator==0.1.4
rfc3986-validator==0.1.1
rich==13.7.1
rope==1.12.0
rpds-py==0.10.6
rsa==4.7.2
s3fs==2024.6.1
scikit-image==0.24.0
```

```
scikit-learn==1.5.1
scipy==1.13.1
seaborn==0.13.2
service-identity==18.1.0
setuptools==75.1.0
sip==6.7.12
six==1.16.0
sklearn-compat==0.1.3
smart-open==5.2.1
smmap==4.0.0
sniffio==1.3.0
snowballstemmer==2.2.0
sortedcontainers==2.4.0
soupsieve==2.5
sphinxcontrib-applehelp==1.0.2
sphinxcontrib-devhelp==1.0.2
sphinxcontrib-htmlhelp==2.0.0
sphinxcontrib-jsmath==1.0.1
sphinxcontrib-qthelp==1.0.3
sphinxcontrib-serializinghtml==1.1.10
spyder-kernels==2.5.0
spyder==5.5.1
stack-data==0.2.0
statsmodels==0.14.2
streamlit==1.37.1
superqt==0.6.7
sympy==1.13.2
tables==3.10.1
tabulate==0.9.0
tblib==1.7.0
tenacity==8.2.3
terminado==0.17.1
text-unidecode==1.3
textdistance==4.2.1
threadpoolctl==3.5.0
three-merge==0.1.1
thrift==0.17.0
tifffile==2023.4.12
tinycss2==1.2.1
tldextract==5.1.2
toml==0.10.2
tomli==2.0.1
tomlkit==0.11.1
toolz==0.12.0
tornado==6.4.1
tqdm==4.66.5
traitlets==5.14.3
```

```
twisted-iocpsupport==1.0.2
typing_extensions==4.11.0
tzdata==2023.3
uc-micro-py==1.0.1
ujson==5.10.0
unicodedata2==15.1.0
urllib3==2.2.3
w3lib==2.1.2
watchdog==4.0.1
wcwidth==0.2.5
webencodings==0.5.1
websocket-client==1.8.0
whatthepatch==1.0.2
wheel==0.44.0
widgetsnbextension==3.6.6
win-inet-pton==1.1.0
wrapt==1.14.1
xarray==2023.6.0
xlwings==0.32.1
xmltodict==0.14.2
xyzservices==2022.9.0
yapf==0.40.2
yarl==1.11.0
zict==3.0.0
zipp==3.17.0
zope.interface==5.4.0
```