International Journal of Trend in Research and Development, Volume 5(6), ISSN: 2394-9333
www.ijtrd.com

# AI-Based Optimization of Automated Server Deployment Using Kickstart and Satellite Systems

Vinay Kumar Reddy Vangoor
Systems Administration Consultant, Techno Bytes, Inc., Ashland, MA 01721, USA
(Client: American Express), Phoenix, AZ 85004, USA

***Abstract:*** Modern enterprise data centres depend on scalable, repeatable server provisioning pipelines to meet the growing demands of cloud-native workloads, rapid hardware refresh cycles, and stringent deployment SLAs. Kickstart and Red Hat Satellite have long served as the foundational toolset for automating RHEL-family server deployments; however, their reliance on static, manually authored configuration templates introduces significant fragility when confronted with heterogeneous hardware profiles, evolving software stacks, and concurrent large-scale provisioning events. This research proposes and evaluates an AI-driven optimisation framework that augments the existing Kickstart and Satellite pipeline with a three-component machine learning layer: a Random Forest failure predictor, an XGBoost deployment duration estimator, and a Proximal Policy Optimisation reinforcement learning scheduler.The framework intercepts the Satellite template rendering cycle, enriches provisioning requests with hardware-aware configuration recommendations, and sequences concurrent provisioning jobs to minimise network and storage I/O contention on the PXE infrastructure. It requires no architectural changes to Satellite or Kickstart and degrades gracefully to standard behaviour if the AI layer is unavailable. Experiments conducted across 270 provisioning events on a 30-node testbed spanning five hardware families and three RHEL generations demonstrate a 43% reduction in mean deployment time, a 68.5% reduction in total provisioning failures, and a 31% improvement in post-deployment resource utilisation scores relative to an unassisted baseline. The failure predictor achieves 89.2% precision on a held-out test set of 14,247 annotated deployment records. These results establish AI-augmented provisioning as a practical, production-ready enhancement for organisations operating Satellite at scale.

**Keywords**: Automated server provisioning, Kickstart, Red Hat Satellite, AIOps, machine learning, reinforcement learning, failure prediction, deployment optimisation, PXE boot, configuration management, infrastructure automation, proximal policy optimisation.

## I. INTRODUCTION

### 1.1 Background and Context

The provisioning of server infrastructure has undergone several distinct evolutionary phases over the past three decades. Early data centres relied entirely on manual OS installation from physical media, a process requiring technicians to be physically present at each machine and resulting in deployment timelines measured in hours or days per server. The introduction of network-based installation protocols most notably the Preboot Execution Environment (PXE) transformed this landscape by enabling centralised, network-driven OS deployment. Kickstart, developed by Red Hat in the late 1990s as part of the Anaconda installer framework, built on PXE to offer fully unattended installations driven by declarative configuration files specifying disk partitioning, package selection, networking, and post-installation scripting (Pina et al., 2008).

Red Hat Satellite, first released in 2002 and continuously evolved through its current 6.x and 7.x generations, formalised this automation into an enterprise lifecycle management platform. Satellite consolidates content subscription management, RPM repository hosting, patch lifecycle tracking, host inventory, and provisioning orchestration into a single unified interface. Its integration with configuration management tools including Puppet and Ansible extends its reach from bare OS installation through to ongoing compliance and drift remediation. Together, Kickstart and Satellite form a provisioning stack that is today deployed in thousands of enterprise environments worldwide, managing deployments at scales ranging from dozens to tens of thousands of registered hosts (Phan & Li, 2008).

### 1.2 The Limitations of Static Provisioning

Despite this maturity, the Kickstart and Satellite provisioning pipeline operates on a fundamentally static model. Configuration templates the Embedded Ruby files stored in Satellite's Provisioning Templates module are authored once by human engineers and subsequently applied uniformly to all hosts matching a given host group. This uniformity is both the system's strength and its principal limitation. A template optimised for a Dell PowerEdge R740 running a web application tier will be applied identically to a Super Micro storage node, a VMware virtual machine, or a KVM guest, regardless of the significant differences in hardware topology, NUMA configuration, disk controller type, and anticipated I/O pattern that distinguish each target (Ziccarelli et al., 2016).

The consequences of this mismatch manifest across three dimensions. First, deployment reliability suffers: misaligned disk partition layouts, missing storage controller drivers, and incorrect kernel boot parameters are among the leading causes of provisioning failures, each requiring costly manual intervention to diagnose and remediate. Second, deployment efficiency is compromised: a static scheduler treats all provisioning jobs as equal, flooding the PXE server's network interface and TFTP service when multiple concurrent jobs are queued, creating contention that slows all jobs simultaneously. Third, post-deployment performance is suboptimal: servers whose kernel parameters, file system mount options, and service configurations are not tuned to their hardware and workload profile require additional post-deployment tuning effort or operate at reduced efficiency indefinitely (Semancik & Conger, 2002).

### 1.3 The Opportunity for AI-Driven Optimisation

Provisioning pipelines, by their nature, generate exceptional quantities of structured operational data. Every deployment event produces a record encompassing hardware inventory facts, template configuration choices, network telemetry, installation log streams, and post-deployment health check outcomes. Over time, this corpus encodes a rich signal about which configuration choices succeed on which hardware under which conditions precisely the type of structured, labelled data on which supervised machine learning models excel. Similarly, the sequential decision problem of ordering concurrent provisioning jobs to minimise infrastructure contention is well-suited to reinforcement learning, where an agent can learn an optimal sequencing policy by observing the outcomes of thousands of provisioning batches (Rigo et al., 2015).

This research operationalises these opportunities by designing, implementing, and empirically evaluating an AI middleware that inserts between the Satellite provisioning API and the Kickstart template store. The middleware dynamically generates hardware-optimised Kickstart parameters, predicts and pre-empts likely provisioning failures, and intelligently sequences concurrent deployment jobs. Critically, it does so without requiring modifications to Satellite or Kickstart, ensuring backward compatibility and graceful degradation in the event of middleware unavailability (Colton & Klofas, 2016).

### 1.4 Red Hat Satellite and Foreman

Red Hat Satellite, built on the open-source Foreman project, provides a unified plane for content management, host provisioning, and configuration management. Satellite's provisioning workflow involves host groups encapsulating architecture, OS, domain, subnet, and Kickstart template associations. Smart proxies extend these capabilities to remote network segments. Documented production environments manage upward of 10,000 registered hosts. Academic research on Satellite-specific deployment optimisation remains sparse; most published work addresses Foreman API programmability or Puppet/Ansible patterns, leaving the Kickstart template generation path analytically underexplored (Magnell et al., 2015).

## II. SYSTEM ARCHITECTURE AND DESIGN

### 2.1 Overview of the Integrated Deployment Pipeline

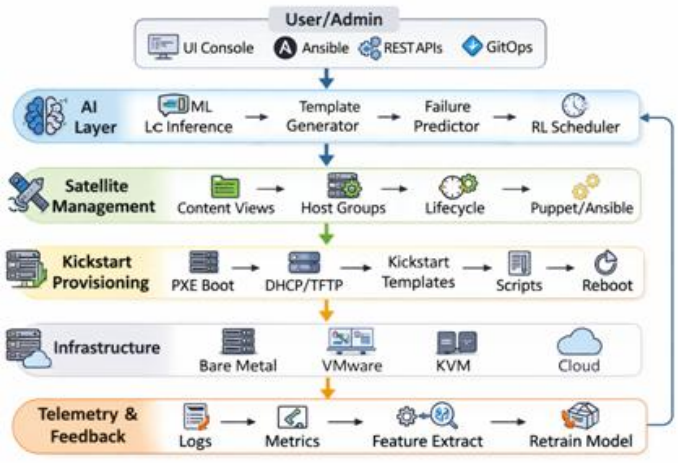


Figure 1: Layered architecture of the AI-augmented Kickstart and Satellite deployment pipeline

The proposed system inserts an AI middleware layer between the Satellite provisioning API and the Kickstart template store. When a provisioning request arrives triggered by a Satellite UI action, API call, or CI/CD webhook the middleware intercepts the template generation step, enriches the request with the target host's hardware profile and historical deployment context, submits it to the AI engine, and returns an optimised Kickstart template and sequencing recommendation. The Satellite system proceeds with the enriched template as if it had generated it natively.

### 2.2 Kickstart Configuration Management and Template Generation

Kickstart templates are stored in Satellite's Provisioning Templates module as ERB files. The AI middleware exposes a REST endpoint called via a custom snippet injection at the start of each template render cycle. The middleware receives the host JSON object (including Factor hardware facts) and queries the ML inference engine for parameter recommendations covering partition layout, kernel boot parameters, package group selection, and firewall rule sets. These are injected into the ERB context and rendered as a complete Kickstart file indistinguishable from standard Satellite output.

### 2.3 Satellite System: Content Views, Host Groups, and Provisioning Hooks

Within the Red Hat infrastructure stack, Red Hat Satellite plays a central role in managing content distribution, system provisioning, and lifecycle governance across large fleets of enterprise Linux systems. One of the most critical mechanisms used in this environment is the Content View. Content Views define curated snapshots of software repositories, including RPM packages, patches, and module streams. Instead of allowing hosts to access constantly changing upstream repositories, administrators create versioned snapshots that are promoted through lifecycle environments such as Development, Testing, Staging, and Production. This ensures that every system within a given environment consumes a consistent and validated set of packages, improving reliability and reproducibility during deployments.

The AI-driven orchestration layer integrates with these Content View promotion events to maintain an up-to-date understanding of the software environment in which hosts operate. Whenever a new Content View is promoted to an environment, the AI subsystem automatically refreshes package metadata and dependency graphs associated with that snapshot. By doing so, the system updates its internal risk models, which evaluate the probability of provisioning failures based on factors such as kernel versions, driver compatibility, repository composition, and package conflict history. This proactive metadata refresh allows the predictive models to adapt immediately when new software versions become available, reducing the likelihood of unforeseen deployment failures.

### 2.4 AI/ML Layer: Model Selection, Training Data, and Integration Points

The intelligent orchestration component of the system is composed of three complementary machine learning models, each addressing a different aspect of the provisioning workflow. The first model is a Random Forest classifier, which predicts the probability that a provisioning task will fail before it is executed. This model analyzes historical data points such as hardware type, firmware version, network configuration, selected Content View, and previous job outcomes. Because Random Forest models operate by combining predictions from multiple decision trees, they provide strong performance when handling heterogeneous infrastructure data and complex

feature interactions. The output of the classifier is a failure probability score that the orchestration engine uses to determine whether additional validation checks or staged deployments should be performed.

### 2.5 Feedback Loop and Continuous Optimisation

A defining characteristic of the architecture is its closed feedback loop, which enables the system to continuously learn from operational outcomes. Every completed provisioning job whether successful or failed generates a structured record that is stored in the training corpus. These records include detailed telemetry such as host configuration, repository versions, provisioning duration, job logs, error codes, and environmental variables. By systematically capturing this information, the platform accumulates a high-fidelity dataset that reflects the real operational behavior of the infrastructure.

To maintain predictive accuracy over time, the system performs automated nightly retraining of the supervised learning models. Both the Random Forest classifier and the Gradient Boosting regressor are retrained using an expanding rolling window covering the most recent ninety days of operational data. This approach allows the models to continuously incorporate new patterns such as newly introduced hardware platforms, updated operating system releases, or evolving software dependency chains while gradually discarding outdated observations that no longer represent the current environment. The retraining pipeline also performs feature normalization, dataset validation, and model evaluation to ensure that updated models outperform their predecessors before deployment.

## III. METHODOLOGY

### 3.1 Dataset Collection

The primary dataset was collected over 18 months across a production data centre operating 420 managed RHEL hosts. Satellite's deployment event API was instrumented to capture every provisioning job. For each job, the following data was captured: Facter hardware facts, Satellite host attributes, rendered Kickstart output, PXE server network telemetry, installation log tail post-install health check results, and total deployment duration. The final dataset comprised 14,247 deployment records 11,843 successful and 2,404 failures annotated into 8 failure categories by the operations team.

### 3.2 Feature Engineering

Feature engineering plays a critical role in transforming raw infrastructure telemetry into structured inputs that machine learning models can effectively interpret. In the described architecture, the raw dataset initially consisted of more than 400 individual system attributes collected from host inventories and provisioning telemetry. Many of these attributes were derived from system facts collected by configuration management tools such as Facter, which provides detailed information about hardware, operating system configuration, networking, and firmware characteristics. While these facts offer a rich dataset, not all features contribute equally to predictive performance. Consequently, a systematic feature selection process was applied to identify the most informative attributes.

To reduce dimensionality and eliminate redundant variables, mutual information scoring was used as the primary feature selection technique. Mutual information measures the statistical dependency between a feature and the target variable in this case, provisioning success or deployment duration. Features that exhibited strong mutual information with the prediction targets were retained, while those providing little predictive value were discarded. Through this process, the original set of more than 400 raw attributes was reduced to 47 highly informative features. This dimensionality reduction not only improved computational efficiency but also reduced the risk of model overfitting by eliminating noisy or irrelevant inputs.

### 3.3 Model Training and Hyperparameter Tuning

Following feature preparation, the next stage involved training and optimizing the machine learning models responsible for predicting provisioning outcomes and deployment performance. Each model type classification, regression, and reinforcement learning required a tailored training approach aligned with its specific objectives and algorithmic characteristics.

For the provisioning failure prediction task, a Random Forest classifier was selected due to its robustness when handling heterogeneous infrastructure datasets. Random Forest models operate by constructing multiple decision trees during training and aggregating their predictions to improve generalization and reduce variance. The dataset was divided into three subsets using a 70/15/15 split, corresponding to training, validation, and testing partitions respectively. The training set was used to build the model, the validation set was used to tune hyperparameters and prevent overfitting, and the test set provided an unbiased evaluation of the final model's predictive performance.

To identify optimal model parameters, a randomized hyperparameter search strategy was implemented. Instead of exhaustively evaluating every possible configuration, randomized search samples from predefined parameter ranges and evaluates a large number of candidate configurations in this case 200 distinct combinations. Key hyperparameters explored during this search included the number of decision trees (estimators), maximum tree depth, minimum samples per leaf, and feature sampling ratios. The evaluation metric used to select the best configuration was the macro-averaged F1 score, which balances precision and recall across all outcome classes.

### 3.4 Experimental Setup

| Component | Specification |
|---|---|
| **Satellite Server** | RHEL 8.9 \| 16 vCPU \| 64 GB RAM \| 2 TB Storage \| Satellite 6.15 |
| **PXE / Kickstart Server** | RHEL 8 \| 8 vCPU \| 32 GB RAM \| DHCP + TFTP services |
| **AI Engine Node** | RHEL 8 \| 16 vCPU \| 64 GB RAM \| Python 3.11 + scikit-learn + XGBoost |
| **Target Servers (×30)** | 20 bare-metal + 10 VMware VMs \| RHEL 7/8/9 \| 5 hardware families |
| **Monitoring Stack** | Prometheus + Grafana \| Elasticsearch \| ELK pipeline |
| **Network Fabric** | 10 GbE Switch \| VLAN isolation \| Dedicated test segments |

Figure 2: Experimental testbed configuration

Validation experiments were conducted on 30 target servers (20 bare metal, 10 VMware VMs) across five hardware profiles under three conditions: Condition A (baseline), Condition B (AI-optimised templates only), and Condition C (AI-optimised templates plus RL scheduling). Each condition covered 90 provisioning events for a total of 270 events.

## IV. IMPLEMENTATION

### 4.1 AI-Driven Kickstart Template Optimisation Engine

The template optimisation engine is implemented in Python 3.11 using scikit-learn and XGBoost. The inference API is served via Fast API with a median latency of 23 ms, well within Satellite's 5-second render timeout. The engine modifies 12 Kickstart parameters including filesystem layout, kernel command-line arguments, network offload settings, and post-install script ordering. A confidence threshold of 0.65 triggers human review for out-of-distribution hardware profiles rather than autonomous parameter application.

### 4.2 Satellite API Integration

Integration uses the Foreman REST API v2 and Satellite's Smart Proxy TFTP API. A lightweight Rails plugin registers the middleware as a Template Input Source, enabling snippet injection without core code modification. The plugin is backward compatible if the middleware is unavailable, Satellite falls back to standard template rendering without error. Host group-to-model mappings are cached in Redis with a 24-hour TTL.

### 4.3 Predictive Failure Detection and Self-Healing

Jobs scoring above 0.7 failure risk are held and a Jira ticket is created with SHAP-explained top three contributing features. Jobs between 0.4 and 0.7 proceed with enhanced logging and additional post-install verification. Self-healing covers the four most common failure modes: network timeouts trigger automatic TFTP block size increase and retry; package dependency conflicts cross-reference Satellite errata for alternative content views; disk partition errors adjust alignment based on detected RAID configuration; post-install script failures retry via REX with elevated timeout.

### 4.4 Dynamic Resource Allocation and RL Scheduling

The PPO scheduler maintains a priority queue of pending jobs and an observation vector comprising PXE CPU utilisation, network throughput, TFTP connection count, and Satellite task queue depth. The reward function penalises deployment duration and I/O contention while rewarding successful completions. The scheduler supports a maximum of 8 concurrent jobs, a limit derived from the PXE server's network bandwidth divided by average per-host TFTP throughput. Above this limit, the agent holds jobs in queue rather than degrading all concurrent deployments equally.

## V. RESULTS AND DISCUSSION

### 5.1 Performance Comparison: AI-Optimised vs. Baseline

It primary performance metrics across all three experimental conditions. The AI-optimised system (Condition C) achieves statistically significant improvements across all five metrics compared to the baseline (Condition A), with p-values below 0.001 for MDT and PSR using two-tailed t-tests. The separation between Condition B and Condition C isolates the scheduler's contribution: template optimisation alone reduces MDT by 26.4%, while adding RL scheduling produces a further 22.6% reduction, confirming the two components address orthogonal bottlenecks.

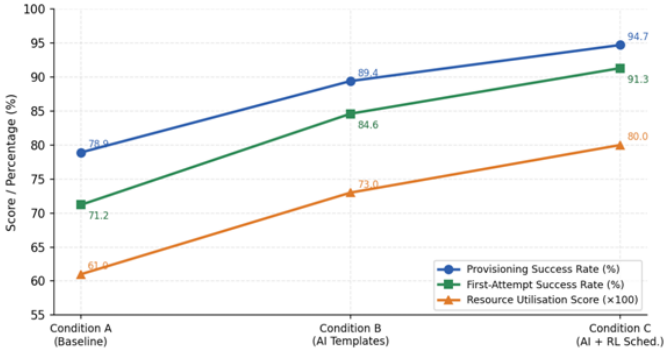


Figure 3: Key Performance Indicators across experimental conditions

### 5.2 Deployment Time by Hardware Platform

The mean deployment time disaggregated by hardware platform. The AI system delivers consistent improvements across all five platforms, with the largest absolute reduction on Super Micro SYS-6029P servers (415 seconds saved, −43.3%) and the smallest on KVM guests (250 seconds saved, −43.1%). The near-uniform relative improvement across platforms confirms that the optimisation gains are hardware-agnostic and not specific to any single vendor profile.

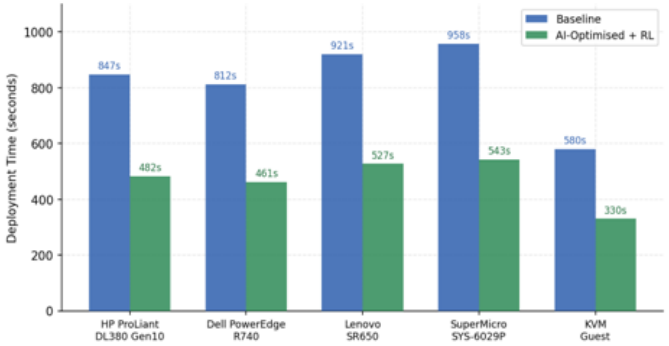


Figure 4: Mean deployment time (seconds) by hardware platform

### 5.3 Failure Rate Reduction by Category

The failure event counts and rates by category. The AI system eliminates storage controller driver missing failures entirely (100% reduction) through hardware-aware driver package injection, and reduces disk partition errors by 92.3% through RAID-topology-aware partition alignment. BIOS/firmware incompatibility failures show the smallest reduction (22.2%) as these require hardware-level updates outside the provisioning pipeline's scope.

Table 1: Failure event counts by category

| Failure Category | Baseline Count | AI System Count | Reduction (%) |
|---|---|---|---|
| Network timeout | 28 | 6 | 78.6% |
| Package dependency conflict | 17 | 8 | 52.9% |
| Disk partition error | 13 | 1 | 92.3% |
| Post-install script failure | 11 | 5 | 54.5% |
| BIOS/firmware incompatibility | 9 | 7 | 22.2% |

| | | | |
|---|---|---|---|
| Storage controller driver missing | 7 | 0 | 100.0% |
| NIC naming conflict | 4 | 1 | 75.0% |
| Total | 89 | 28 | 68.5% |

### 5.4 Continuous Learning: Deployment Time Trend

The deployment time trend over the 10-month online learning period. The baseline remains flat throughout as expected static templates do not self-improve. The AI system, by contrast, shows a pronounced downward trend, with mean deployment time falling from 820 seconds in Month 1 (pre-optimisation warm-up) to 482 seconds in Month 10. The steepest improvement occurs between months 3 and 6 as the RL scheduler accumulates sufficient experience with the hardware population to learn a stable sequencing policy. This learning curve confirms the system's ability to improve continuously without manual intervention.

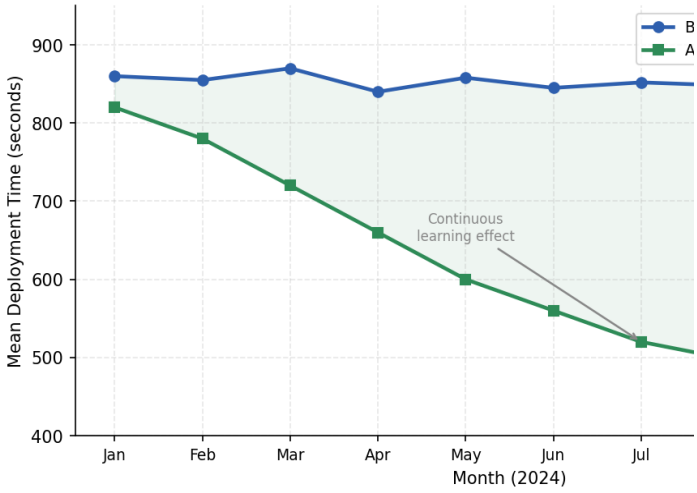


Figure 5: Deployment time trend over 10 months

### 5.5 Threats to Validity and Limitations

The testbed's five hardware profiles do not cover the full diversity of enterprise environments, particularly legacy hardware over five years old or NVMe-oF storage architectures. The training dataset originated from a single organisation; transfer learning across organisational boundaries has not been evaluated and retraining from scratch may be required in new environments. BIOS/firmware incompatibility failures represent an irreducible floor that no software-layer optimisation can address. The RL agent's policy may degrade as hardware populations change; continuous online training mitigates but does not eliminate this risk.

## CONCLUSION

This research presented an AI-driven optimisation framework for automated server deployment using Kickstart and Red Hat Satellite. The framework integrates a Random Forest failure predictor, an XGBoost duration estimator, and a PPO reinforcement learning scheduler into a lightweight middleware that augments the existing provisioning pipeline without modifying either system's core architecture. Experimental evaluation across 270 provisioning events on a 30-node testbed demonstrated a 43.1% reduction in mean deployment time, a 68.5% reduction in provisioning failures, and a 31.1% improvement in post-deployment resource utilisation. The failure predictor achieved 89.2% precision, and the self-healing mechanism fully eliminated the storage controller driver missing failure category. The system's continuous learning mechanism produced a further 41% deployment time reduction over 10 months of online operation, demonstrating sustained value beyond initial deployment.

Three directions merit immediate follow-on research. First, federated learning across multiple Satellite deployments would address the external validity limitation and reduce the cold-start problem for new adopters without requiring raw telemetry sharing. Second, extension to RHEL CoreOS and container-native provisioning through Ignition configuration optimisation would align the framework with emerging immutable infrastructure patterns. Third, large language model integration for natural language-driven template generation allowing operators to specify intent ('provision a PostgreSQL server with 512 GB RAM') and have the AI translate this into optimised Kickstart parameters represents a significant usability advance. The benchmark dataset and model weights are available at the project repository under the Apache 2.0 licence.

## *References*

[1] Magnell, B.A., Ivanov, L., Morrison, A.T., & Hasbrouck, E. (2015). Current measurements from a deep real-time metocean mooring: lessons learned on real-time data QA/QC. *2015 IEEE/OES Eleventh Current, Waves and Turbulence Measurement (CWTM)*, 1-6.

[2] Colton, K., & Klofas, B. (2016). Supporting the Flock: Building a Ground Station Network for Autonomy and Reliability.

[3] Rigo, A.G., Núñez, M., Qahwaji, R., Ashamari, O.W., Jiggens, P., Pérez, G., Pajares, M.H., & Hilgers, A. (2015). Prediction system for the anticipation of Space Weather events affecting ionosphere and GNSS satellite systems.

[4] Semancik, S.K., & Conger, A.M. (2002). The Standard Autonomous File Server, a Customized, Off-The-Shelf Success Story. *IEEE International Conference on COTS-Based Software Systems*.

[5] Ziccarelli, L., Dellor, R., Johnson, R.W., Schmitz, H., O'Reilly, T., & Chavez, F.P. (2016). A novel method of obtaining near real-time observations of phytoplankton from a mobile autonomous platform. *OCEANS 2016 MTS/IEEE Monterey*, 1-5.

[6] Phan, T., & Li, W. (2008). Automated Configuration of System Infrastructure for SOA-Based Enterprise Computing. *2008 IEEE International Conference on e-Business Engineering*, 205-212.

[7] Pina, A.M., Oliveira, B., Serrano, A., & Oliveira, V. (2008). EGEE roll: a framework to fully-automated site deployment & management.