Use the Programmable Installer

Contents

Introduction

This document describes the Programmable Installer, a spec-driven automation platform for deploying Cisco software stacks like NSO and CNC.

Programmable Installer

Field	Value
Product	Programmable Installer
Document type	Technical white paper — architecture, implementation, and outcomes
Primary audience	Solution architects, Platform engineers, DevOps / SRE, Delivery leads
Secondary audience	Engineering management, Security reviewers, Program managers

---

Abstract

The Programmable Installer is a spec-driven automation platform for deploying and operating Cisco software stacks—including Network Services Orchestrator (NSO), Crosswork Network Controller (CNC), Crosswork Data Gateway (CDG), and Business Process Automation (BPA)—on enterprise Linux (RHEL family) and associated infrastructure where applicable (VMware vCenter, OpenShift, KVM, air-gapped Kubernetes). The system separates declarative intent (YAML specifications and optional guided intent generation) from execution (Ansible roles and playbooks), with a Python control plane that packages artifacts, verifies bundles before long-running installs, prepares encrypted secrets, and orchestrates validation gates.

This white paper explains the architectural layers, primary data flows, implementation patterns (including a hybrid data-driven artifact verification model), deployment modes (native, containerized, online, air-gapped), and the validation and logging frameworks. For delivery and platform organizations, the platform aims to reduce manual toil, surface misconfiguration and missing binaries early, and standardize automation across products and topologies while preserving environment-specific parameterization.

Keywords: infrastructure automation, declarative deployment, Ansible, YAML specification, air-gap packaging, artifact verification, Crosswork, NSO, CNC, CDG, BPA, validation policy, DevOps.

How to Read this Document

Role	Suggested focus
Decision-makers / leads	Abstract; §1 Value proposition; §8 Benefits and risk posture; §10 Conclusion
Solution architects	§3–§6 (architecture, specification model, implementation, deployment patterns)
DevOps / SRE / delivery engineers	§5–§7; Appendix B; companion internal white paper annexes
Security reviewers	§7 Security and compliance posture; trust boundaries in §3.2

---

1. Value Proposition

1.1 The Delivery Problem

Enterprise installation of multi-tier network automation and orchestration products is traditionally high-touch: long runbooks, many manual steps, version skew between sites, and failures that surface hours into a process (missing NEDs, wrong OVA paths, incomplete air-gap image sets). That pattern increases cost, elongates change windows, and makes audits harder.

1.2 The Programmable Installer Approach

The Programmable Installer treats an installation as a program parameterized by a specification: topology, versions, platform choice (vCenter vs OpenShift vs vanilla VMs), file paths, and entitlements. Automation is idempotent where possible, repeatable across customers, and front-loaded with checks so that “not ready” is a fast, explicit outcome before cluster or product install.

1.3 What Programmable Means in this Context

- Declarative: Operators describe *what* must be deployed; playbooks implement *how*.
- Data-driven verification: Expected artifacts are derived from tables and rules rather than ad-hoc scripts per release, when patterns are stable.
- Policy-gated quality: Pre- and post-deployment validation runs under hierarchical policies, with structured reports and optional ticketing integration.
- Multi-modal operation: Interactive menus for first-time users; CLI and frozen binaries for CI/CD-style repetition; Docker-based flows for standardized runtime images.

---

2. System Context

2.1 Actors and Environments

Actor / system	Role
Delivery engineer	Authors or generates specs, runs packaging, vault preparation, validation, orchestrator, and Ansible
Installer host	Linux control node (native or container) with Python, Ansible configuration, disk for

Actor / system	Role
	artifacts
Target infrastructure	vCenter, OpenShift/KubeVirt, or vanilla VMs per specification
Artifact sources	Internal mirrors, entitlement layouts, software distribution—environment-specific
Downstream systems	Monitoring, change management, optional JIRA workflows

2.2 Trust Boundaries

1. Specifications express intent; they can reference paths and non-secret parameters. Secrets must flow through Ansible Vault workflows not plain-text spec fields where avoidable.
2. Artifact storage must be integrity-protected; verification focuses on presence and naming alignment with the spec—extend with organizational controls (checksums, signed bundles) where policy requires.
3. Installer-to-target SSH is a high-privilege path; compromise of the installer host is high impact. Hardening and access control are operational prerequisites.

---

3. Architectural Principles

1. Declarative first: User intent in YAML; automation interprets it consistently.
2. Separation of planning and execution: Python plans, verifies, and orchestrates gates; Ansible executes infrastructure and product steps.
3. Composable automation: Site-level playbooks import focused playbooks (preinstall, Kubernetes tracks, product installs).
4. Progressive disclosure: Interactive setup for onboarding; flags and scripts for advanced automation.
5. Same codebase, multiple runtimes: Native AlmaLinux/RHEL paths and Docker-based paths share one repository layout.
6. Explicit air-gap support: Package on a connected machine; transfer a bundle; install prerequisites and deploy without runtime dependency on public networks.

---

4. Logical Architecture

4.1 Layers

Layer	Responsibility
Specification	Topology, versions, platforms, paths, entitlements
Control plane	Packaging, bundle verification, vault helpers, validation driver, orchestrator CLI
Automation plane	Host prep, Kubernetes lifecycle, product install and day-one configuration
Artifacts	Binaries, images, charts, OVAs, tarballs

4.2 End-to-end Data Flows

1. Package flow: The prerequisite tooling downloads or stages files into specific location, optionally

producing a transferable tarball for offline installations.
2. Verify flow: The orchestrator parses the spec, resolves expected artifacts (including platform filters and entitlement lists), and reports ready / missing before install.
3. Deploy flow: Spec plus vault (and optional pre-validation) drive Ansible playbooks against inventories derived from the engagement.

4.3 Supported Products (Installer Scope)

Product / bundle
NSO
CNC
CDG
BPA
CNC + NSO

---

5. Specification and Intent Model

5.1 User Specifications

Specifications are YAML documents describing platforms (for example vCenter, OCP, VM, KVM), hosts, applications with versions, topology (for example NSO CFS/RFS layouts), entitlements (NEDs and add-on packages), and file paths for OVAs, qcow2 images, and application-tier tarballs.

5.2 Common Fragments

A specific application user_spec supplies defaults and path fallbacks for CNC/CDG. The orchestrator's parser treats the user spec as source of truth and uses common spec entries when user keys are absent.

5.3 Intent Generation

The intent generator supports guided collection of requirements via set of questionnaire, a rule engine (date driven logic), and schema-backed mapping to intent.yaml.

---

6. Implementation Deep Dive

6.1 Deployment Orchestrator (cx_deploy_orchestrator.py)

The orchestrator is the single entry for scripted or interactive generate-intent, verify-bundle,

and install coordination. Its design is explicitly hybrid:

- ARTIFACT_DEFS: Declares artifact types and naming patterns per application (NSO installer, NED signed packages, optional packages; CNC OVA/qcow2/tier tarballs; CDG images; BPA charts and air-gap image tarballs).
- APP_CONFIG: Maps CLI --app values (nso, crossworksuite, bpa) to specification folder names and default intent filenames.
- Parser / Handlers: Resolve CNC/CDG paths using user spec and common spec; custom handlers cover non-uniform naming (for example TSDN/DLM) and BPA version formatting for chart and tarball paths.

Readiness**:**AReportaggregates discovered artifacts; **is_ready**is true when no required files are missing after spec parsing. The module supports PyInstaller-frozen binaries via sys.frozen path resolution.

6.2 Prerequisite and Packaging Tooling (setup_cxinstaller_prereqs)

This component provides interactive menus and CLI modes for artifact packaging and host prerequisite installation: online, air-gap, or auto-detect; multi-application packaging including combined`CNC_NSO`. It populates the artifact tree expected by Ansible and by verify-bundle logic.

6.3 Ansible Automation Plane

- **Composition**: This illustrates the multi-track nature of the stack: it imports different Kubernetes bootstrap paths —reflecting that different products target different Kubernetes bootstrap paths.
- **Roles (representative families)**: preinstall (SELinux, firewall, SSH, registry helpers), k8s_install / rke2, deploy_nso, deploy_cnc, deploy_cdg, deploy_bpa, postinstall, uninstall and certificate renewal helpers. Ownership and testing are best managed at role boundaries; nested task folders implement sub-workflows (for example NSO L3 HA).

6.4 Validation Checks Framework (validation_checks/)

The framework provides hierarchical policy control (global → app → stage → individual check), auto-discovery of checks, enhanced reporting to structured logs, and optional JIRA integration. Operators run pre- or post-deployment phases against the same spec used for install, aligning automation with quality gates suitable for enterprise change discipline.

Indicative scale on a typical branch: on the order of thirty checks across BPA and NSO (counts to be confirmed with**make list-validation-checks**on your checkout).

6.5 Vault Secrets Manager (scripts/vault_secrets_manager.py)

Derives required vault variables from specs, prompts or accepts passwords under policy, and emits encrypted group_vars/all_secrets.yaml plus a vault password file for Ansible—reducing ad-hoc secret

embedding in playbooks.

---

7. Deployment Patterns and Runtimes

Pattern	Summary
Native (AlmaLinux / RHEL)	Set **PYTHONPATH** and **ANSIBLE_CONFIG**; run packaging, vault, validation, orchestrator, and playbooks per product guide
Docker-based installer	**scripts/setup_installer.sh** and **scripts/start_installer.sh** with host mounts for large artifacts;
Air-gap	Package on a connected machine; transfer bundle; extract on target; install with**--airgap**
macOS bundle creation	Use**python3 ./setup_cxinstaller_prereqs.py** on Mac to prepare bundles; target deployment remains Linux-oriented per project docs

---

8. Security, Validation, and Observability

8.1 Security Posture (Design Intent)

- **Secrets:**Prefer Ansible Vault encrypted group variables; use vault manager strict modes where appropriate.
- **Installer host:**Treat as a high-trust control plane; restrict access and monitor.
- **Artifacts:**Protect acquisition channels; organizational processes can augment verify-bundle with cryptographic verification.
- **Logging:**Application and Ansible logs underdeploy/logs/

8.2 Validation as an Operational Gate

Example pre-deployment invocation:

```
cd /opt/cx-installer
python3 validation_checks/run_validation_checks.py -t pre -s specification/your_spec.yaml
```

With optional flags:

- **-p <policy_file>**— use a custom validation policy (defaults to validation_checks/validation_policies/default.yaml)
- **-a <app>** — limit checks to a specific app (lowercase, such as cnc, nso, bpa, cdg)
- **--report-file <path>** — write a standalone JSON precheck report

8.3 Release Discipline

This is a model for inner sourcing where for more CISCO application installation same principle can be followed by forking the repository.

---

9. Benefits and Outcomes

Theme	Outcome
Time and toil	Fewer manual steps; failures detected at verify-bundle and validation phases rather than late in Ansible or product installers
Consistency	Shared schemas, roles, and artifact layout across engagements reduce “snowflake” differences
Disconnected operations	Documented bundle transfer supports regulated networks without runtime downloads
Governance	Structured validation reports and optional JIRA hooks support change records and follow-up
Extensibility	Clear extension points: ARTIFACT_DEFS, handlers, new roles/playbooks, intent schemas

Quantitative metrics (install duration, defect rates) are**organization-specific**; teams must baseline against legacy runbooks on comparable topologies.

---

10. Extensibility and Maintenance

10.1 Adding Artifact Types

1. Extend**ARTIFACT_DEFS**(and labels if needed) in**cx_deploy_orchestrator.py**.
2. Add a**custom handler**when naming cannot be captured by patterns alone.
3. Update packaging logic in**setup_cxinstaller_prereqs**when downloads are automated.

10.2 Adding Ansible Behavior

Prefer new tasks inside cohesive roles; introduce new roles when boundaries are clear. Wire playbooks via**import_playbook**or documented entry playbooks. Keep safe defaults in group_vars / vars.

10.3 Intent Ggenerator Evolution

Update YAML schemas underintent-generator/schema/ and chatbot inputs; ensure generated files match filenames expected byAPP_CONFIG.

10.4 Roadmap Considerations (Illustrative)

- Deeper SBOM or image signature verification integration.
- Expanded validation coverage for CNC/CDG scenarios.
- CI references for spec linting and Ansible syntax checks.

---

11. Conclusion

The CX Programmable Installer combines declarative specifications, a Python control plane for packaging and verification, and an Ansible automation plane for scalable, repeatable deployments of Crosswork-related products across diverse infrastructure and connectivity models. Its architecture intentionally separates intent from execution, applies data-driven artifact expectations where practical, and embeds validation and vault workflows suitable for enterprise delivery. For full operational annexes—playbook tables, troubleshooting matrices, connectivity matrices, and extended command references—see the companion internal white paper.

References and Documentation Map

Document	Path
Operator guide	README.md
Release playbook	RELEASE_GUIDE.md
Internal architecture annexes	docs/CX_INSTALLER_TECHNICAL_WHITE_PAPER_INTERNAL.md
Docker (online / air-gap / usage)	SETUP_ONLINE_DOCKER.md, SETUP_AIRGAPPED_DOCKER.md, USAGE_DOCKER.md
Validation framework	docs/validation_checks/README.md
Vault manager	docs/scripts/VAULT_SECRETS_MANAGER.md
Product guides	docs/nso.md, docs/bpa.md, docs/CNC_VCENTER_DEPLOYMENT_GUIDE.md, docs/CNC_OCP_DE
Intent generator	intent-generator/README.md
Chatbot / rule flow overview	docs/HowItWorks.md

Appendix A — Repository Layout (Summary)

```
cx-installer/
```

```
├── ansible_playbooks/     # ansible.cfg, files/, group_vars/, playbooks/, roles/, vars/
├── apps/                  # App-specific supporting content
├── deploy/                # Python deploy helpers, logging utilities
├── docs/                  # Technical documentation
├── intent-generator/      # Chatbot, rule engine, schemas, output/
├── scripts/               # Docker setup/start, vault_secrets_manager.py, ...
├── specification/         # User specs, samples, common fragments
├── validation_checks/     # Policies, runners, reports
├── cx_deploy_orchestrator.py
├── setup_cxinstaller_prereqs*
├── requirements.txt
└── README.md
```

Post-setup artifact focal points (typical):ansible_playbooks/files/artifacts/, files/bin/, files/charts/, files/images/.

Appendix B — Orchestrator CLI (Summary)

Script:cx_deploy_orchestrator.py

Argument	Description
--app / -a	nso | crossworksuite | bpa
--spec / -s	Path to YAML specification
--step	generate-intent | verify-bundle | install
--verify-only	Verify bundle; exit non-zero if not ready
--dry-run	Dry run where supported
--list-specs	List known specs

Environment (typical session):

```
export PYTHONPATH=$(pwd)
export ANSIBLE_CONFIG=$(pwd)/ansible_playbooks/ansible.cfg
```

Appendix C — Glossary

Term	Definition
Spec	YAML user specification: platforms, apps, topology, paths, entitlements
Intent	Normalized YAML from the intent generator or hand-authored equivalent
Bundle	Packaged installer tree (often tarball) for air-gap transfer
Orchestrator	**cx_deploy_orchestrator.py** — verify / intent / install coordination
Artifact verification	Filesystem checks that required binaries/images exist per spec

Term	Definition
Vault	Ansible Vault–encrypted variable file for secrets
NED	Network Element Driver package (NSO)
CFS / RFS	NSO cluster forwarder / redundant forwarder topology concepts
Air-gap	Environment without installer-time access to package download endpoints

---

Document Revision History

Version	Date	Notes
1.0	2026-03-27	Initial publication-ready technical white paper (Programmable Installer framing)