7.0 - AI Guardrails: As-Built Report Generator

AI Development Reference Guide: As-Built Report Generator

Version: 1.0

---

1. Project Core Directives

This section establishes the foundational understanding of the project. It is the primary source of truth for the AI's objectives and must be reviewed at the beginning of every development session to align with the project's vision.

1.1. Primary Objective

The primary objective is to develop a Python-based command-line tool, the **VAST As-Built Report Generator**, that automates the creation of a comprehensive, professionally formatted "as-built" report following the deployment of a VAST Data cluster. This tool must be reliable, secure, and easy for VAST Professional Services (PS) engineers to use.

1.2. Core Vision

The vision is to transform a manual, time-consuming documentation process into a rapid, automated, and error-free workflow. The final deliverable should not only be a functional script but a polished product that enhances the value delivered to VAST's customers.

1.3. Key Deliverables

1. **Python CLI Application:** A command-line tool that connects to a VAST cluster, extracts data, and generates reports.
2. **PDF Report:** A professionally formatted, human-readable PDF document suitable for customer delivery.
3. **JSON Data File:** A machine-readable JSON file containing all the raw data extracted from the cluster, serving as a data source for the PDF and for any future automation needs.
4. **GitHub Repository:** A well-maintained Git repository containing the source code, documentation, and status logs.

2. Development Guardrails and Coding Standards

Adherence to these standards is non-negotiable. They are designed to produce clean, consistent, and maintainable code, which is critical for long-term project success and ease of handoff between development sessions.

2.1. Code Style and Formatting

- **PEP 8 Compliance:** All Python code must strictly follow the PEP 8 style guide. The maximum line length is set to 120 characters.
- **Naming Conventions:**
  - `snake_case` for variables, functions, and modules.
  - `PascalCase` for classes.
  - `UPPER_SNAKE_CASE` for constants.
- **Modularity:** The application must be structured into logical modules (e.g., `api_handler.py`, `report_builder.py`, `main.py`). This isolates concerns and improves code organization.

2.2. Exception Handling and Resilience

- **Specific Exceptions:** Always catch specific exceptions. Avoid generic `except Exception:`.
- **API Fault Tolerance:** All network-facing calls must include a retry mechanism with exponential backoff to handle transient failures.
- **Graceful Degradation:** The tool must not crash if non-critical information is unavailable. It should log a warning and clearly mark the data as "Unavailable" in the final report.

2.3. Security

- **Credential Management:** Credentials must **never** be hardcoded or stored in configuration files. Use environment variables or the `getpass` module for secure credential handling at runtime.
- **No Sensitive Data in Logs:** Ensure that passwords, API keys, or other sensitive information are never written to log files.

3. Task Management and Context Inheritance

This process is critical for maintaining momentum and ensuring seamless transitions between development sessions, especially when a session is interrupted by reaching a context limit.

3.1. The `STATUS.md` Log

A status tracking log named `STATUS.md` must be maintained at the root of the project directory. This file is the single source of truth for project progress.

- **Purpose:** To provide an immediate, at-a-glance understanding of the project's current state, the last completed action, and the immediate next step.
- **Update Frequency:** The `STATUS.md` file must be the **last file updated** at the end of a development session and the **first file read** at the beginning of a new one.

3.2. `STATUS.md` Format

The log must follow this exact format:

```
# VAST Report Generator - Development Status

**Last Updated:** YYYY-MM-DD HH:MM:SS UTC

---

### Current Project Phase

**Phase:** [Name of the current phase from the Project Plan, e.g.,
"Core Functionality & Data Collection"]
**Sprint:** [Sprint Number, e.g., 1]

---

### Last Completed Action

**Description:** [A clear, concise description of the last
successfully completed action. e.g., "Implemented the
`get_cluster_info` function in `api_handler.py` and verified it
returns the cluster name and version."]
**Commit SHA:** [The full SHA of the last commit pushed to GitHub.]

---

### Next Immediate Task

```

```
**Objective:** [A precise and actionable description of the very next task to be performed. e.g., "Implement the `get_cnode_details` function in `api_handler.py` to fetch CNode inventory."]
**Relevant Files:**
- `/path/to/file_to_be_modified.py`
- `/path/to/relevant_test_file.py`

---

### Blockers

[List any issues or blockers preventing progress. If none, state "None."]

```

3.3. Context Inheritance Workflow

When a new development session begins (or context is inherited), the workflow is as follows:

1. **Clone/Pull Repository:** Ensure the local environment is synchronized with the remote GitHub repository.
2. **Read `STATUS.md`:** Open and parse the `STATUS.md` file to understand the exact state of the project.
3. **Verify Last Commit:** Cross-reference the "Last Completed Action" and "Commit SHA" with the Git log to confirm the starting point.
4. **Execute Next Task:** Proceed with the "Next Immediate Task" as defined in the log.
5. **Update `STATUS.md`:** Upon completing the task and committing the changes, update the `STATUS.md` file with the new status before pushing.

4. GitHub Workflow and Version Control

A disciplined Git workflow is essential for maintaining a clean history and enabling effective collaboration and review. All code must be managed in the designated GitHub repository.

- **Repository of Record:** `git@github.com:rstamps01/ps-deploy-report.git`

4.1. Branching Strategy

- `main` Branch: This branch is protected and represents the stable, production-ready version of the tool. Direct commits to `main` are forbidden.
- `develop` Branch: This is the primary development branch. All feature branches will be merged into `develop`.
- **Feature Branches:** For every new piece of functionality (e.g., a new report section, a new module), a new feature branch must be created from `develop`. Branch names should be descriptive (e.g., `feature/add-pdf-generation`, `fix/api-auth-bug`).

4.2. Commit and Push Protocol

1. **Clone the Repository:** At the start of the very first session, clone the repository using SSH:
   ```
   git clone git@github.com:rstamps01/ps-deploy-report.git
   ```
2. **Pull Latest Changes:** At the beginning of every subsequent session, ensure you have the latest code:
   ```
   git pull origin develop
   ```
3. **Commit Atomically:** Make small, atomic commits. Each commit should represent a single logical change. Commit messages must be clear and follow the conventional commit format (e.g., `feat: Add function to fetch DNode details`, `docs: Update README with setup instructions`).

4. **Push Changes:** After committing your changes and updating the `STATUS.md` file, push your feature branch to the remote repository:

   ```
   git push origin <your-feature-branch-name>
   ```

5. **Create Pull Request:** Once a feature is complete, create a Pull Request (PR) on GitHub to merge your feature branch into the `develop` branch. The PR description should summarize the changes.

5. Project Structure and Module Organization

A well-organized project structure is essential for maintainability and clarity. The following structure must be adhered to throughout the development process.

5.1. Directory Structure

```
ps-deploy-report/
├── README.md                    # Project overview and setup instructions
├── STATUS.md                    # Development status tracking log
├── requirements.txt             # Python dependencies
├── config/
│   └── config.yaml             # Configuration file template
├── src/
│   ├── __init__.py
│   ├── main.py                 # Entry point for the CLI application
│   ├── api_handler.py          # VAST API interaction module
│   ├── data_extractor.py       # Data extraction and processing logic
│   ├── report_builder.py       # PDF and JSON report generation
│   └── utils.py                # Utility functions and helpers
├── tests/
│   ├── __init__.py
│   ├── test_api_handler.py     # Unit tests for API module
│   ├── test_data_extractor.py  # Unit tests for data extraction
│   └── test_report_builder.py  # Unit tests for report generation
├── templates/
│   └── report_template.html    # HTML template for PDF generation
├── logs/
│   └── .gitkeep               # Placeholder to ensure logs directory exists
└── output/
    └── .gitkeep               # Placeholder for generated reports
```

5.2. Module Responsibilities

- `main.py` : The entry point for the CLI application. Handles command-line argument parsing, configuration loading, and orchestrates the overall workflow.
- `api_handler.py` : Manages all interactions with the VAST REST API. Includes authentication, connection management, retry logic, and error handling.
- `data_extractor.py` : Contains functions to extract specific data sets from the API responses and organize them for report generation.
- `report_builder.py` : Responsible for generating both the JSON data file and the PDF report from the extracted data.
- `utils.py` : Contains utility functions such as logging setup, configuration file parsing, and common helper functions.

6. VAST API Integration Guidelines

This section provides specific guidance for working with the VAST Data REST API version 7, targeting VAST Cluster version 5.3.

6.1. Critical API Endpoints

The following endpoints are essential for gathering the required data for the as-built report:

Report Section	API Endpoint	Purpose
Executive Summary	`GET /api/clusters/`	Cluster name, GUID, version, state, license
Physical Inventory	`GET /api/cnodes/`, `GET /api/dnodes/`	CNode and DNode hardware details
Network Configuration	`GET /api/dns/`, `GET /api/ntps/`, `GET /api/vippools/`	DNS, NTP, VIP pool configurations
Logical Configuration	`GET /api/tenants/`, `GET /api/views/`, `GET /api/viewpolicies/`	Data access and security policies
Authentication	`GET /api/activedirectory/`, `GET /api/ldap/`, `GET /api/nis/`	External authentication providers
Data Protection	`GET /api/snapprograms/`, `GET`	Snapshot and replication policies

	`/api/protectionpolicies/`	

6.2. API Authentication

The VAST API uses session-based authentication. The authentication flow must be implemented as follows:

1. **Login:** Send a POST request to `/api/sessions/` with username and password.
2. **Session Management:** Store the session cookie returned from the login response.
3. **Request Headers:** Include the session cookie in all subsequent API requests.
4. **Session Expiry:** Handle 401 Unauthorized responses by re-authenticating and retrying the request.

6.3. Error Handling for API Calls

- **Network Errors:** Implement retry logic with exponential backoff for connection timeouts and network failures.
- **HTTP Status Codes:**
  - `200 OK` : Success, process the response data.
  - `401 Unauthorized` : Re-authenticate and retry the request.
  - `404 Not Found` : Log a warning and mark the data as "Not Available" in the report.
  - `500 Internal Server Error` : Log an error and attempt a retry after a delay.
- **Response Validation:** Always validate that the API response contains the expected data structure before attempting to access nested fields.

7. Logging and Debugging Standards

Comprehensive logging is essential for troubleshooting issues in production environments. The logging system must provide clear, actionable information for Professional Services engineers.

7.1. Log Configuration

- **Log File Location:** All logs must be written to the `logs/` directory within the project root.
- **Log File Naming:** Use the format `vast_report_YYYYMMDD_HHMMSS.log` for each execution.
- **Log Rotation:** Implement log rotation to prevent excessive disk usage. Keep the last 10 log files.

7.2. Log Levels and Usage

- `DEBUG` : Detailed information for diagnosing problems. Include API request/response details (excluding sensitive data).
- `INFO` : General information about the program's execution flow. Use for major milestones like "Starting data collection" or "PDF generation complete."
- `WARNING` : Indicates a potential issue that doesn't prevent the program from continuing. Use when optional data cannot be retrieved.
- `ERROR` : Serious problems that prevent a function or section from completing successfully.
- `CRITICAL` : Very serious errors that may cause the program to terminate.

7.3. Log Message Format

Each log entry must include:

```
YYYY-MM-DD HH:MM:SS,mmm - [LEVEL] - [MODULE] - MESSAGE
```

Example:

```
2024-01-15 14:30:25,123 - [INFO] - [api_handler] - Successfully authenticated to VAST cluster at 192.168.1.100
2024-01-15 14:30:26,456 - [WARNING] - [data_extractor] - Could not retrieve quota information for tenant 'finance', skipping section
2024-01-15 14:30:27,789 - [ERROR] - [api_handler] - Failed to connect to API endpoint '/api/views/' after 3 retries. Status Code: 500
```

7.4. Sensitive Data Protection

- **Never Log Credentials:** Passwords, API tokens, and session cookies must never appear in log files.
- **Sanitize URLs:** When logging API endpoints, ensure that any authentication parameters in the URL are redacted.
- **Data Masking:** If logging API responses for debugging, mask any potentially sensitive customer data.

8. Testing and Quality Assurance

A robust testing strategy ensures the reliability and correctness of the tool. Testing must be integrated into the development workflow from the beginning.

8.1. Unit Testing Requirements

- **Test Coverage:** Aim for at least 80% code coverage across all modules.
- **Test Framework:** Use Python's built-in `unittest` framework or `pytest` for all unit tests.
- **Mock External Dependencies:** Use the `unittest.mock` library to mock API calls and external dependencies during testing.
- **Test Data:** Create sample API response data for testing data extraction and report generation functions.

8.2. Integration Testing

- **Live API Testing:** Develop a separate test suite that can be run against a live VAST cluster in a lab environment.
- **End-to-End Testing:** Test the complete workflow from API authentication to final report generation.
- **Error Scenario Testing:** Test how the application handles various error conditions (network failures, authentication errors, malformed API responses).

8.3. Test Execution

- **Automated Testing:** All tests must be executable via a single command (e.g., `python -m pytest tests/`).
- **Continuous Integration:** Tests should be designed to run in a CI/CD pipeline, though the initial implementation may not include this.
- **Test Documentation:** Each test function must have a clear docstring explaining what it tests and why.

9. Session Handoff Protocol

This protocol ensures seamless transitions between development sessions and maintains project momentum when context limits are reached.

9.1. End-of-Session Checklist

Before ending a development session, the following steps must be completed:

1. **Commit All Changes:** Ensure all code changes are committed to Git with clear, descriptive commit messages.

2. **Push to Remote:** Push the current feature branch to the GitHub repository.
3. **Update `STATUS.md`:** Update the status log with the current state, last completed action, and next immediate task.
4. **Run Tests:** Execute any relevant unit tests to ensure the current code is functional.
5. **Document Blockers:** If any issues or blockers are encountered, document them clearly in the `STATUS.md` file.

9.2. Start-of-Session Checklist

When beginning a new development session, follow this protocol:

1. **Pull Latest Changes:** Ensure the local repository is synchronized with the remote repository.
2. **Read `STATUS.md`:** Carefully review the status log to understand the current project state.
3. **Verify Environment:** Confirm that the development environment (Python version, dependencies) is properly set up.
4. **Review Last Commit:** Examine the most recent commit to understand what was last implemented.
5. **Execute Next Task:** Proceed with the "Next Immediate Task" as defined in the status log.

9.3. Context Inheritance Best Practices

- **Granular Tasks:** Break down large features into small, manageable tasks that can be completed within a single session.
- **Clear Documentation:** Always document the reasoning behind implementation decisions in code comments or commit messages.
- **Incremental Progress:** Aim to complete at least one meaningful unit of work per session, even if it's small.
- **State Preservation:** Ensure that the project can be resumed at any point without loss of context or progress.

10. Troubleshooting and Common Issues

This section provides guidance for resolving common issues that may arise during development and deployment.

10.1. API Connection Issues

- **Problem:** Unable to connect to the VAST Management Service (VMS).
- **Diagnosis:** Check network connectivity, VMS IP address, and firewall settings.
- **Solution:** Verify the VMS is accessible via ping or telnet. Ensure the correct IP address and port (typically 443 for HTTPS) are being used.

10.2. Authentication Failures

- **Problem:** Receiving 401 Unauthorized responses from the API.
- **Diagnosis:** Invalid credentials or expired session.
- **Solution:** Verify username and password. Implement session refresh logic to handle expired sessions automatically.

10.3. Incomplete Data in Reports

- **Problem:** Some sections of the report show "Data Unavailable" or are missing entirely.
- **Diagnosis:** API endpoints may be returning unexpected data structures or the cluster may not have certain features configured.
- **Solution:** Review the API response structure and update the data extraction logic. Implement graceful handling for optional or missing data.

10.4. PDF Generation Errors

- **Problem:** PDF report generation fails or produces malformed output.
- **Diagnosis:** Issues with the PDF library, template formatting, or data encoding.
- **Solution:** Validate input data before passing it to the PDF generator. Ensure proper encoding of special characters and handle empty data gracefully.

10.5. Performance Issues

- **Problem:** Report generation takes an excessive amount of time.
- **Diagnosis:** Too many sequential API calls or inefficient data processing.
- **Solution:** Implement concurrent API calls where possible. Optimize data processing algorithms and consider caching frequently accessed data.

11. Final Deliverables and Success Criteria

This section defines the specific deliverables and criteria that must be met for the project to be considered complete and successful.

11.1. Primary Deliverables

1. **Functional CLI Application:** A Python command-line tool that can be executed with simple arguments to generate reports.
2. **Professional PDF Report:** A well-formatted, customer-ready PDF document containing all required sections.
3. **Structured JSON Data:** A machine-readable JSON file containing all extracted cluster data.
4. **Complete Documentation:** README file with installation, configuration, and usage instructions.
5. **Test Suite:** Comprehensive unit and integration tests with at least 80% code coverage.

11.2. Success Criteria

- **Functionality:** The tool successfully connects to a VAST cluster, extracts all required data, and generates both PDF and JSON reports without errors.
- **Reliability:** The tool handles common error scenarios gracefully and provides clear error messages when issues occur.
- **Performance:** Report generation completes within 5 minutes for a standard cluster configuration.
- **Usability:** A Professional Services engineer can install and run the tool with minimal setup and clear instructions.
- **Maintainability:** The code is well-organized, documented, and follows established coding standards.

11.3. Acceptance Testing

Before considering the project complete, the following acceptance tests must pass:

1. **End-to-End Test:** Successfully generate a complete report from a live VAST cluster.
2. **Error Handling Test:** Verify graceful handling of network failures, authentication errors, and missing data.
3. **Documentation Test:** A new user can follow the README instructions to install and run the tool successfully.
4. **Code Quality Test:** All code passes linting checks and meets the established coding standards.

Conclusion

This AI Development Reference Guide serves as the definitive resource for maintaining consistency, quality, and progress throughout the development of the VAST As-Built Report Generator. By adhering to these guidelines, the development process will produce a professional, reliable, and maintainable tool that meets the needs of VAST Professional Services and their customers.

The success of this project depends on strict adherence to these standards and the disciplined use of the status tracking and version control protocols outlined in this document.

---

11. Updated API Data Points (September 2025)

11.1. Additional API Capabilities Discovered

Based on recent API schema analysis, the following data points have been identified as available through the VAST API v7, improving the automated data collection coverage from ~70% to ~80%:

11.1.1. Rack Height Information

- **CBox Rack Heights**: Available via `Schema/CBox/index_in_rack`
- **DBox Rack Heights**: Available via `Schema/DBox/index_in_rack`
- **Implementation**: Update the hardware data collection module to include rack positioning information
- **Impact**: Eliminates manual data entry for physical rack positioning

11.1.2. Cluster PSNT (Product Serial Number Tracking)

- **Endpoint**: Available via `Schema/Cluster/psnt`
- **Implementation**: Include in cluster identification section of the report
- **Impact**: Provides automated support tracking identifier for customer records

11.2. Updated Data Collection Strategy

The discovery of these additional API capabilities requires updates to the following modules:

1. **API Handler Module** (`src/api_handler.py`):
   - Add `get_rack_positions()` method for CBox/DBox rack heights
   - Update `get_cluster_info()` method to include PSNT field
   - Update error handling for these new data points
2. **Data Processing Module** (`src/data_processor.py`):
   - Include rack height validation and formatting
   - Add PSNT to cluster summary section
   - Update data schema validation
3. **Report Builder Module** (`src/report_builder.py`):
   - Add rack height information to hardware sections
   - Include PSNT in cluster identification header
   - Update PDF formatting to accommodate new data points

11.3. Implementation Priority

These updates should be implemented as part of Task 1.2.1 (API Handler Module) to ensure the improved data collection capabilities are available from the initial implementation.

11.4. Testing Requirements

- Verify rack height data accuracy against physical installation

- Validate PSNT format and ensure it matches support system requirements
- Test graceful handling when these fields are not available in older API versions