Automating AWS infrastructure documentation generation and analysis

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Automating AWS infrastructure documentation generation and analysis

Table of Contents

Automating AWS infrastructure documentation generation and analysis

Amazon Web Services (contributors)

November 2025 (document history)

Cloud environments, particularly those leveraging multiple AWS accounts, AWS Regions, and AWS services, present significant challenges in maintaining accurate and up-to-date documentation. Manual methods fall short in capturing real-time changes, ensuring configuration accuracy, mapping inter-service dependencies, and assessing alignment with AWS best practices. This visibility gap adversely impacts security, cost control, compliance, and operational efficiency.

The *Infrastructure Documentation Generator* is an advanced automation tool designed to comprehensively scan, document, and analyze AWS Cloud environments with minimal human intervention. It tackles the persistent challenge of maintaining accurate, current infrastructure documentation while extracting actionable insights for optimization and compliance. As cloud architectures rapidly evolve, manual documentation becomes increasingly time-consuming, error-prone, and quickly outdated. This tool resolves these issues by performing thorough scans of AWS services, configurations, and relationships. Then, it can automatically generate rich documentation including dependency maps between resources and detailed configuration reports.

Going beyond simple resource listings, the Infrastructure Documentation Generator incorporates intelligent analysis that's aligned with the AWS Well-Architected Framework. This alignment helps organizations to understand their current environment. In addition, the tool also assesses how well the environment adheres to best practices across security, operational efficiency, cost optimization, performance, and reliability. The tool organizes discovered resources within their application context. As a result, teams can visualize service relationships, fostering improved collaboration among cloud infrastructure teams, solutions architects, developers, and security personnel. The reports generated by the tool also provide valuable insights for business stakeholders in areas such as cost management, compliance tracking, and strategic planning.

Intended audience

The Infrastructure Documentation Generator serves a wide range of stakeholders within cloud-focused organizations:

- **Cloud infrastructure teams** – Provides access to real-time, comprehensive infrastructure visibility enabling rapid troubleshooting, proactive optimization, and efficient management of AWS resources across multiple AWS accounts and AWS Regions.
- **Solutions architects** – Enhances architectural decision-making through detailed infrastructure visualization and dependency mapping, helping new designs align with existing systems while adhering to AWS best practices.
- **Development teams** – Offers clear insight into the production environment's configuration and dependencies. Enables developers to better understand how their applications interact with the underlying infrastructure and make informed deployment decisions.
- **Security teams** – Supports automated discovery of security vulnerabilities, misconfigurations, and compliance gaps, providing a streamlined approach to maintaining a robust security posture across the cloud environment.
- **Business stakeholders** – Provides access to high-level insights about infrastructure costs, resource utilization, and compliance status, supporting data-driven decisions for strategic planning and budget allocation.

Objectives

This tool is designed to deliver measurable business value, from daily operations to long-term strategic planning, and can help you achieve the following:

- **Operational excellence** – This tool dramatically transforms operational efficiency by automating the documentation process, reducing manual effort by up to 80%. Teams can now resolve infrastructure issues 60-70% faster with instant access to accurate configuration details and dependency mappings. This automation reduces the traditional documentation backlog and enables teams to focus on strategic initiatives rather than routine documentation tasks.
- **Cost optimization** – Through comprehensive resource tracking and utilization analysis, organizations can identify and reduce unnecessary cloud spending. The tool provides detailed insights into resource allocation, enabling precise capacity planning. By highlighting potential consolidation opportunities, companies typically achieve significant cost savings while maintaining optimal performance levels.
- **Risk management** – The tool substantially reduces organizational risk by providing monitoring and assessment of security configurations and compliance requirements. This proactive approach to risk management helps maintain a robust security posture and ensures regulatory compliance across the cloud environment.

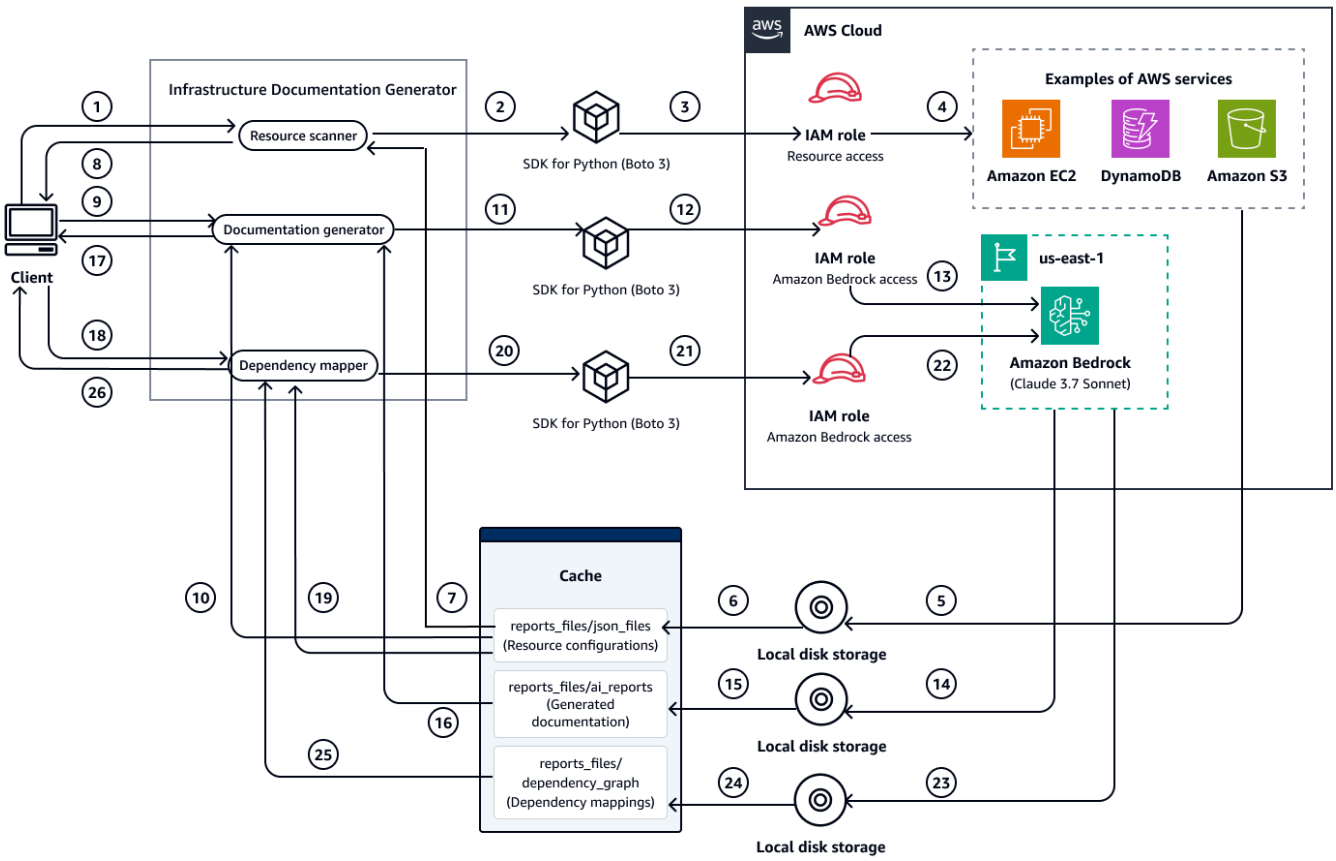
- **Strategic value** – By providing infrastructure insights, the tool enables faster and more informed decision-making at all levels of the organization. Leadership teams can better align IT operations with business objectives, while technical teams can more effectively plan and execute infrastructure changes.
- **Infrastructure governance** – The tool establishes a standardized approach to infrastructure documentation and management across all cloud environments. By maintaining consistent, up-to-date records of configurations and changes, organizations can better demonstrate compliance during audits and more effectively manage infrastructure modifications. This standardization helps all teams to work from the same accurate information, improving collaboration and reducing confusion.

Architecture of the Infrastructure Documentation Generator

The Infrastructure Documentation Generator is designed with a modular architecture. The system works in the following major phases:

- ***Resource scanning*** – The initial process of querying AWS services to discover and inventory resources, collecting metadata and policies needed for further documentation and dependency analysis. The subsequent phases rely on the data collected in this phase.
- ***Documentation generation*** – The system creates detailed Markdown-based reports that describe AWS resource configurations, best practices, and security posture.
- ***Dependency mapping*** – A process that analyzes resource-based policies and identifiers to build a graph of how AWS resources relate and interact with each other, enabling visualization of resource connections and access paths.

Each phase follows a structured sequence of steps, as shown in the following diagram.



The following list provides details about each step:

1. ***Resource scanning phase***

   Request resource information – The process begins when a client sends an initial request to the system to gather information about their AWS infrastructure.

2. **Request resource scan** – The resource scanner component activates and prepares to scan multiple AWS services. It formulates the appropriate API calls to collect comprehensive infrastructure data.

3. **Authenticate and access** – The system uses the AWS SDK for Python (Boto3) to interact with AWS services by making authenticated API requests using the provided credentials. This establishes a valid session for retrieving resource information.

4. **Authorize access to query resources** – AWS Identity and Access Management (IAM) role permissions are verified to confirm the scanner has the appropriate access rights to query the required AWS services. This serves as a crucial security check before retrieving resource data.

5. **Store resource data** – The system collects detailed configuration data from various AWS services, including instance details, network configurations, storage settings, and service relationships.
6. **Transfer resource data to cache storage** – The collected data is efficiently moved to local disk cache storage. This enables faster access and reduces repeated API calls to AWS services, improving performance.
7. **Respond with resource configuration** – The system compiles the collected data into a structured format suitable for client consumption and prepares it for transmission back to the client.
8. **Display resource configurations** – The formatted resource configuration data is presented to the client through the interface, showing detailed insights into their AWSinfrastructure.

Documentation generation phase

9. **Request documentation generation** – The client initiates the documentation process based on the scanned resource data, triggering the documentation generator component.
10. **Use resource configurations** – The system retrieves cached resource data from the scanning phase to serve as the foundation for documentation generation.
11. **Initiate AI documentation generation** – The system begins the AI-powered documentation process using Amazon Bedrock, preparing the necessary context and prompts for documentation creation.
12. **Authenticate for Amazon Bedrock API** – The system establishes a connection to Amazon Bedrock by making authenticated API requests with valid credentials.
13. **Authorize access for documentation** – IAM permissions are verified to make sure that the system is allowed to use Amazon Bedrock for generating documentation.
14. **Store generated AI documentation** – The AI-generated documentation is saved in persistent storage for future reference.
15. **Transfer documentation to cache** – The generated documentation is copied into cache storage for quicker access and performance.
16. **Return documentation response** – The system formats and sends the generated documentation back to the client.
17. **Display documentation** – The documentation is presented to the client through the interface in a clear and organized format.

Dependency mapping phase

18. **Request dependency analysis** –The client initiates the process to analyze and map relationships between AWS resources.
19. **Use resource configurations** – The system accesses cached resource data from the scanning phase to identify interdependencies among resources.
20. **Initiate relationship analysis** – The dependency mapper processes the resource data to identify and map connections between AWS services.
21. **Authenticate for Amazon Bedrock** – The system establishes a connection to Amazon Bedrock by making authenticated API requests with valid credentials.
22. **Authorize access for dependency analysis** – IAM permissions are verified to ensure the system can use Amazon Bedrock for dependency analysis tasks.
23. **Store dependency results** – The completed dependency analysis results are saved in persistent storage.
24. **Transfer mappings to cache** – Dependency mapping data is moved into cache storage for faster access and reuse.
25. **Return dependency response** – The system prepares and sends the dependency analysis results back to the client.
26. **Display dependency graph** – The final dependency relationships are visualized and shown to the client as an interactive graph, illustrating service connections and resource dependencies.

Core components of the Infrastructure Documentation Generator

The Infrastructure Documentation Generator is built in Python with FastAPI for deployment and a React-based user interface for visualization. Its architecture is composed of modular components, each responsible for a specific part of the infrastructure discovery and documentation process.

This section explains the following key components in detail:

- Resource scanning
- Documentation generation
- Dependency mapping
- Caching mechanism

Resource scanning

The *resource scanning* component is the foundation of the Infrastructure Documentation Generator. Its job is to discover and inventory AWS resources across AWS Regions in a consistent and structured way. This process is critical because all subsequent phases (documentation generation and dependency mapping) rely on the data collected here.

Key responsibilities of the resource scanning component

This section describes the key actions and responsibilities of the resource scanning component.

Enumerate AWS resources

The resource scanner begins by identifying all AWS resources across AWS services. For each service, it performs targeted API calls through AWS SDK for Python (Boto3) to retrieve detailed configurations. Many AWS services consist of multiple sub-resources, such as Amazon Elastic Compute Cloud (Amazon EC2) instances, volumes, and security groups. The scanner systematically captures them all to ensure complete visibility of the cloud environment.

The scanner uses a declarative configuration approach through the SERVICE_SCAN_FUNCTIONS dictionary, which maps each AWS service to its specific API calls and response structures.

Optimize parallel execution

To maximize efficiency when scanning large AWS environments, the system implements parallel processing across multiple AWS Regions and services simultaneously. Using a thread pool architecture, the scanner distributes API calls across concurrent workers, dramatically reducing total execution time.

The core of this parallel execution strategy is implemented in the scan_resources function, which orchestrates the entire scanning process using the Python `concurrent.futures` framework.

Extract resource-based policies

Certain AWS services expose resource-based policies that define how external entities can interact with them. Examples of such services are Amazon Simple Storage Service (Amazon S3), AWS Lambda, Amazon Simple Notification Service (Amazon SNS), and Amazon Simple Queue Service (Amazon SQS). The scanner retrieves these policies directly from the resources and attaches them to the discovered metadata. This provides valuable context for understanding not only what resources exist but also how they are secured and who can access them.

The add_lambda_resource_policies function retrieves and attaches resource-based policies to Lambda functions, enriching the resource data.

Normalize and filter fields

AWS API responses can be verbose, containing extensive metadata that isn't always useful for documentation. To address this, the scanner filters responses down to a consistent set of essential fields, such as `Name`, `Arn`, and `Id`. This normalization ensures the results are lightweight, uniform, and easier to consume for both human readers and downstream processes like dependency mapping or documentation generation.

The filter_resource_fields function implements the tool's field filtering strategy, defining essential fields in RESOURCE_ESSENTIAL_FIELDS for each resource type. The function applies these filters to create consistent, streamlined resource representations:

Return results in JSON format

All scanned data is returned in a structured JSON format. The output includes details of AWS services and Regions scanned, lists of discovered resources with metadata, aggregated resource counts, and any attached policies. This JSON representation serves as a standardized contract

between the scanning component and other parts of the system, supporting interoperability and easy integration with visualization, analysis, or reporting layers.

UI integration and delivery

The scanned results are displayed in the UI with expandable sections, grouped by service and Region. Users can drill down into specific AWS services and explore discovered resources interactively before moving to further phases.

Workflow of the resource scanning component

The resource scanning component uses the following workflow:

1. **Start scan** – The process begins when the client (through an API or UI) initiates a scan request. The client provides AWS credentials or specifies a target account and IAM role to assume. This input allows the system to securely access the required AWS environment.
2. **Create session** – The system establishes an SDK for Python (Boto3) session with the provided credentials. If scanning across multiple accounts, AWS Security Token Service (AWS STS) is used to assume the specified IAM role in the target AWS account. All subsequent API calls are executed within this authenticated and authorized context.
3. **Discover AWS Regions and AWS services** – The scanner enumerates all available Regions and builds a service catalog of API operations to be performed. Both Regional services (such as Amazon EC2 and AWS Lambda) and global services (such as IAM and Amazon Route 53) are included to ensure full coverage.
4. **Parallel execution** – Using a thread pool, the scanner issues API calls across Regions and services in parallel. This design significantly reduces scan time and enables scalability across large, multi-account AWS environments.
5. **Process responses and attach policies** – The raw API responses are normalized and filtered to include only essential fields (such as `Name`, `Arn`, and `Id`). During this stage, the scanner also retrieves resource-based policies from services that expose them (for example, Amazon S3 buckets, Lambda functions, Amazon SNS topics, and Amazon SQS queues). These policies are directly attached to the corresponding resources in the results, providing deeper insight into access configurations.
6. **Aggregate results** – Results from all Regions and services are combined into a single structured JSON document. This aggregated output includes metadata about Regions and services scanned, discovered resources, resource counts, and attached policies. The JSON format

makes the results easy to consume for downstream processes such as dependency mapping, documentation generation, or visualization in the UI.

Example JSON output

The following example shows typical JSON output when scanning Amazon S3 buckets and Lambda functions. This output includes the AWS account ID, scan timestamp, AWS Regions and AWS services scanned, and detailed resource information including names and ARNs. For each resource, it also indicates whether a resource-based policy is present (as shown for the S3 bucket) or absent (as shown for the Lambda function).

```
{
  "account_id": "123456789012",
  "scan_time": "2025-09-03 10:20:15",
  "regions_scanned": ["us-east-1", "us-west-2"],
  "services_scanned": ["s3", "lambda"],
  "resources": [
    {
      "service": "s3",
      "region": "us-east-1",
      "function": "list_buckets",
      "resources": [
        {
           "Name": "my-app-bucket",
           "Arn": "arn:aws:s3:::my-app-bucket",
           "resource_based_policy":
            {
              "Version": "2012-10-17",
              "Statement": [...]
            }
        }
       ],
      "resource_count": 1
    },
    {
      "service": "lambda",
      "region": "us-west-2",
      "function": "list_functions",
      "resources": [
        {
           "FunctionName": "process-data-fn",
```

```
            "Arn": "arn:aws:lambda:us-west-2:123456789012:function:process-data-fn",

            "resource_based_policy": null
          }
        ],
        "resource_count": 1
      }
    ]
}
```

User view of resource scanning

The scanned infrastructure is displayed in the UI as expandable sections organized by AWS service and Region. Users can drill down into each service to view discovered resources and attached resource-based policies.

Documentation generation

The *documentation generation* component converts scanned AWS infrastructure data into AI-powered analyses and structured documentation. It uses Claude by Anthropic in Amazon Bedrock to perform in-depth analysis, produce recommendations, and format results as Markdown.

For large datasets, the component applies an intelligent chunking strategy and synthesizes or combines chunk outputs to produce a final analysis artifact. *Intelligent chunking* is a technique to divide large infrastructure datasets into manageable subsets for analysis by generative AI services (for example, Claude on Amazon Bedrock). Chunking helps bypass token or size limits while preserving analytical accuracy. The final documentation is displayed directly in the UI. Users can also download the complete Markdown file for offline review.

Key responsibilities of the documentation generation component

This section describes the key actions and responsibilities of the documentation generation component.

Analyze infrastructure data and create detailed analysis outputs

The component ingests the JSON response from the resource scanning phase and determines the appropriate strategy. For smaller datasets, it runs a single-call analysis with Amazon Bedrock to generate a Markdown report. For larger datasets, it uses intelligent chunking, analyzing subsets individually and then combining results into a unified document.

The system uses the generate_infrastructure_report function, which intelligently determines whether the infrastructure data requires a single-call analysis or chunked processing based on its size.

Produce AI-powered documentation with actionable recommendations

The system generates Markdown documentation that includes findings, severity levels, and optimization or remediation recommendations. Users can choose from four analysis types—comprehensive, security, cost, or performance. Users can also provide a custom prompt to tailor the analysis to their specific needs (for example, compliance checks or compute security posture).

The create_single_analysis_prompt function highlights how the system tailors the user preference and creates the necessary prompt for generating a custom Markdown report.

Handle large-scale inputs using chunking and combination/synthesis

When infrastructure data exceeds token limits, the dataset is divided into intelligent chunks. Each chunk is processed with Amazon Bedrock. Then, it's either combined directly or synthesized through an additional Amazon Bedrock call to produce a cohesive final report. Partial results and metadata are preserved for traceability.

The InfrastructureChunker class is responsible for intelligently splitting large infrastructure data into manageable chunks based on AWS services. This approach helps to make sure that token limits for Amazon Bedrock processing are not exceeded.

Retry, backoff, and error handling

To ensure reliability, the component implements retries with exponential backoff when Amazon Bedrock throttling or API errors occur. Failed chunks are logged, and the final report contains an "Analysis Limitations" section if any subset could not be processed.

The call_claude_with_retry function implements robust retry logic with exponential backoff. If the API call encounters a rate-limiting issue (such as throttling), it retries up to a maximum number of attempts with progressively increasing delays. Non-rate-limit errors are handled separately, with immediate failure and logging.

UI integration and delivery

The generated Markdown report is rendered in the UI for interactive viewing. Users can also download the Markdown file directly for offline use or integration into documentation systems.

Workflow of the documentation generation component

The documentation generation component uses the following workflow:

1. **Request documentation generation** – The client requests documentation by providing the normalized scan JSON. The request may include a selected analysis type (comprehensive, security, cost, or performance) and an optional custom prompt to refine scope.
2. **Estimate data size and decide strategy** – The system estimates token usage to decide between a single comprehensive Amazon Bedrock call or splitting the dataset into multiple intelligent chunks for parallel analysis.
3. **Create prompts** – For single-call cases, the system builds a detailed expert prompt that requests findings, risks, and recommendations. For chunked cases, chunk-specific prompts are created, each embedding contextual instructions and the selected analysis type or custom prompt.
4. **Call Amazon Bedrock** – The prompts are sent to the Amazon Bedrock runtime. For chunked analysis, each chunk is processed sequentially with conservative backoff and retry logic to avoid hitting rate limits.
5. **Combine or synthesize chunk outputs** – If chunking was used, the component either combines chunk outputs into a single Markdown document or synthesizes them into a cohesive analysis with a final Amazon Bedrock call. If some chunks failed, their status is recorded and an "Analysis Limitations" note is added.
6. **Add metadata and return result** – The final output is packaged with metadata (for example, model used, chunking strategy, timings, and error details) and delivered back to the client. In the UI, the Markdown is displayed interactively, and the download option is provided for exporting the report.

User view of documentation generation

The AI-generated Markdown analysis report is shown directly in the UI with a structured, readable format. Users can preview findings, risks, and recommendations in the browser and download the full Markdown file for offline reference.

Dependency mapping

The *dependency mapping* component analyzes relationships between AWS resources and builds a comprehensive graph of how AWS services interact with each other. Unlike the scanning phase

(which only inventories resources) and the documentation phase (which explains configurations and best practices), dependency mapping focuses on interconnections. To infer how resources are linked, this phase relies heavily on resource-based policies (for example, Amazon S3 bucket policies, Lambda execution permissions, and Amazon SNS or Amazon SQS access controls).

While the system produces a structured Markdown document for consistency and auditability, the same data is also consumed by the UI to generate an interactive graph view. This makes it possible to explore resource connections visually, identify access paths, and spot potential misconfigurations.

Key responsibilities of the dependency mapping component

This section describes the key actions and responsibilities of the dependency mapping component.

Extract policies and identifiers

The process begins by examining the infrastructure data (JSON from the scanning phase) and isolating all resources that expose policies, such as Amazon S3 buckets, Lambda functions, Amazon SNS topics, Amazon SQS queues, or Amazon EC2 instances with IAM profiles. For each resource, the system extracts identifiers and any attached resource-based policies. Examples of identifiers are IDs, Amazon Resource Names (ARNs), and names. These identifiers and policies form the basis of relationship discovery.

The extract_resource_policies function handles the extraction of policies and identifiers, by processing the infrastructure data to extract resources with attached policies, such as Amazon S3 buckets and Amazon EC2 instances.

Construct resource ARNs

Where possible, full ARNs are extracted directly from the resource metadata. For resources that don't provide ARNs explicitly, the system reconstructs them by using service-specific patterns. This approach guarantees a uniform representation across all resources, which is essential for cross-service dependency analysis.

The extract_resource_arn function handles the process of constructing ARNs. It either directly extracts ARNs from resource metadata or reconstructs them by using service-specific patterns defined in `ARN_CONSTRUCTION_MAP`.

Dependency analysis powered by Amazon Bedrock

After policies and ARNs are extracted, the system generates structured prompts that are passed to Amazon Bedrock (using the Claude 3.7 Sonnet model). The prompts strictly enforce a structured Markdown output format where each resource and its connections are described in a predictable schema. This approach helps to produce output that machines can parse, and humans can read.

The generate_resource_mapping_from_infra_data function generates a service-by-service Markdown output after extracting and processing resource based policies using Amazon Bedrock.

Continuation and retry handling

The mapping process includes a continuation mechanism that does the following:

1. If Amazon Bedrock can't analyze all resources in a single call, it returns `CONTINUE_ANALYSIS`.
2. The system generates a continuation prompt with the remaining resources, waits briefly (to avoid rate limits), and retries.
3. This cycle repeats until all resources are analyzed or the marker is detected.

The generate_service_markdown_with_continuation function ensures that services are processed sequentially and that the continuation prompt logic is handled with retries in case of rate limits or incomplete analysis.

Aggregation and combination

Finally, all per-service dependency mappings are merged into a single, consolidated Markdown document.

This process is handled by the method combine_service_markdown_results. It ensures that all the dependency mappings are merged in a logical, readable format. The format starts with a header, followed by the total resources, successful services, and resource dependencies for each service. The final document presents both the successes and failures of the analysis in a structured way.

Graph conversion for UI

The Markdown output is parsed and transformed into a graph data model (nodes and edges) using React Flow as follows:

- *Nodes* represent resources such as Lambda functions, Amazon S3 buckets, and Amazon DynamoDB tables.

- *Edges* represent connections or access relationships such as Lambda to `S3: InvokeFunction`. This enables the frontend to render a network graph where dependencies can be visually explored.

Workflow of the dependency mapping component

The dependency mapping component uses the following workflow:

1. **Input: infrastructure data** – The dependency mapping process begins with the JSON output generated in the document generation phase. This file contains detailed information about AWS resources, their metadata, and any attached resource-based policies, serving as the foundation for all subsequent analysis.
2. **Policy extraction** – Before relationships can be mapped, the system runs a policy extractor that identifies resources with attached policies. It normalizes identifiers such as ARNs, cleans up service-specific variations, and outputs a refined dataset focused only on resources relevant for dependency mapping.
3. **Prompt construction** – After the clean dataset is prepared, the system constructs specialized prompts tailored for Amazon Bedrock (which is using Claude). These prompts embed extracted policy data and enforce strict Markdown formatting rules, so that the AI generates structured, machine-readable, and human-readable output instead of free-form text.
4. **Amazon Bedrock analysis** – Amazon Bedrock processes the prepared dataset and generates dependency mappings by analyzing policies and resource metadata. To handle large infrastructures, the system includes retry and continuation support, allowing analysis to continue seamlessly across multiple calls until all resources are covered.
5. **Aggregation and post-processing** – Outputs from individual service analyses are aggregated into a single unified document. During this stage, duplicates are removed and errors are flagged. Metadata, such as total resources analyzed, services covered, and processing time, is appended for completeness.
6. **Graph transformation** – The structured Markdown output from Amazon Bedrock is transformed into a graph data model. Each resource becomes a node and relationships derived from policies or metadata are represented as edges, enabling a clear visualization of dependencies and access flows.
7. **Visualization in UI** – Finally, the dependency graph is rendered in the UI using React Flow, where users can explore it interactively. Features like zoom, filtering, and search help engineers and

security teams to quickly identify relationships, spot misconfigurations, and understand the broader AWS resource landscape.

User view of dependency mapping

The dependency mapping results are displayed as an interactive graph in the UI. Users can explore resource relationships visually, making it easier to understand complex dependencies across AWS services. You can also select any specific resource card to focus on its specific dependencies.

Caching mechanism

The *caching mechanism* ensures that repeated infrastructure scans, analyses, and dependency mappings are efficiently served without redundant API calls or reprocessing. The caching mechanism is implemented across all components—resource scanning, document generation, and dependency mapping. This layer reduces response times, optimizes Amazon Bedrock usage, and improves scalability for large or frequent workloads.

Key responsibilities of the caching mechanism

This section describes the key actions and responsibilities of the caching mechanism.

Avoid redundant computation

Before performing a new scan, analysis, or mapping, the system checks the cache for an existing result that matches the input dataset and request parameters. If found, the cached response is immediately returned, significantly reducing processing overhead.

The compare_infrastructure_data function is responsible for checking if the infrastructure data has changed and compares it to cached data, thus preventing redundant computation.

Cache AI-powered reports and dependency graphs

Both documentation reports and dependency mappings generated by Amazon Bedrockare stored with associated metadata. Cached entries include analysis type, input file fingerprints, timestamps, and processing details, so that repeat requests can be served instantly.

The check_ai_report_cache and check_dependency_graph_cache functions handle the checking of cache for AI reports and dependency graphs, respectively. If cached data is available, they return the data, saving computation time.

Track and expose cache statistics

The system provides a `/cache/stats` endpoint to expose cache utilization metrics, such as total cache hits, misses, and file counts. A *cache hit* is an event where a requested resource or result is found in the cache, allowing instant retrieval without repeating the underlying computation or API call. A *cache miss* is an event where no matching cached resource is found, triggering a fresh scan or analysis to generate new results which are then cached.

This enables observability into how often caching is used and how much computation is being saved.

The get_cache_stats function provides valuable cache utilization metrics, including the number of cache hits, misses, and the total size of the cache.

Enable cache maintenance and cleanup

Through the `/cache/cleanup` endpoint, old or stale cache files can be removed based on configurable retention periods. This approach prevents uncontrolled cache growth and ensures storage remains efficient over time.

The cleanup_old_cache function helps to manage old and stale cache files. It cleans up outdated files, so that only recent data is retained in the cache, preventing it from growing too large.

Workflow of the caching mechanism

The caching mechanism uses the following workflow:

1. **Cache lookup** – When a request for documentation generation, infrastructure analysis, or resource mapping is received, the system first queries the cache with the normalized input data and request type.
2. **Cache hit** – If a valid cached entry is found, the system enriches it with request-specific metadata (processing time, file details, and `served-from-cache` flags) and immediately returns the cached result to the client.
3. **Cache miss** – If no valid cache entry is found, the system processes the request normally by calling AWS APIs or Amazon Bedrock, generates fresh results, and then saves them to the cache for future reuse.
4. **Cache monitoring and cleanup** – Administrators can retrieve real-time statistics or trigger cache cleanup to remove entries older than a defined threshold (default 30 days). This keeps the cache performant and prevents excessive storage usage.

Set up the Infrastructure Documentation Generator

This section describes how to build and run the Infrastructure Documentation Generator tool. It provides a systematic approach to setting up both the backend service (FastAPI) and the frontend application (React UI) so that you can scan, analyze, and visualize AWS infrastructure. The backend is responsible for discovering resources, generating documentation, and mapping dependencies. The frontend presents results in an interactive UI. Together, these components form a complete approach for understanding and optimizing AWS environments.

Set up the backend service

These instructions guide you through setting up the backend service, which is responsible for scanning AWS accounts and generating structured infrastructure data. The backend API layer is built with FastAPI, a modern Python web framework. FastAPI provides endpoints for infrastructure scanning, documentation generation, dependency mapping, and cache operations. The backend interacts with various AWS services and provides an API for managing and retrieving cloud infrastructure details.

Prerequisites

Before you begin, make sure you meet the following prerequisites:

- Python 3.9 or a later version installed on your machine.
- AWS credentials configured—this prerequisite is mandatory. The tool requires AWS credentials to interact with your AWS account. You must use the AWS Command Line Interface (AWS CLI) `aws configure` command to configure your credentials to allow the service to access AWS resources.
- Amazon Bedrock with access to Claude 3.7 Sonnet enabled.

Set up and run the backend environment

To set up the backend environment, use the following steps:

1. To clone the repository, run the following command:

```
git clone https://github.com/awslabs/InfraDocGen.git
```

2. To create and activate a virtual environment, run the following command:

```
python3 -m venv venv source venv/bin/activate
```

3. To install backend dependencies, run the following command:

```
pip install -r requirements.txt
```

4. To start the API server, run the following command:

```
uvicorn main:app --reload
```

Set up the frontend application

These instructions guide you through setting up the frontend application, which visualizes AWS resources, analysis reports, and dependency graphs in an interactive web UI. The frontend uses a React Flow library to render interactive, zoomable, and searchable graphs of AWS resource dependencies. The frontend application interacts with the backend service to display infrastructure data in a user-friendly and insightful way.

Prerequisites

Before you begin, make sure you meet the following prerequisites:

- Node.js v20+ or a later version installed.
- A package manager to install frontend dependencies, for example, npm or Yarn.

Set up and run the frontend application

To set up the frontend environment and run the application, use the following steps.

1. To move to the UI project, run the following command:

```
cd infra-ui
```

2. To create an environment file to connect to the backend service, run the following command:

```
echo "VITE_API_BASE_URL=http://127.0.0.1:8000" > .env
```

3. To install frontend dependencies, run the following command:

```
npm install
```

4. To start the frontend development server, run the following command:

```
npm run dev
```

Run the backend and the frontend together

After the backend service and the frontend application are set up successfully, you can run the full system to experience the complete functionality. To use the system effectively, start both the backend API server and the frontend UI in parallel, each in its own terminal window or tab.

To run the backend and the frontend together, use the following steps:

1. Start the backend API (`uvicorn main:app --reload`) in one terminal.
2. Start the frontend UI (`npm run dev`) in another terminal.
3. Open the UI at **http://localhost:3000"**. The UI connects automatically to the backend at **http://127.0.0.1:8000**.

From the UI, you can trigger scans, view service-wise expandable infrastructure data, explore dependency graphs, and download Markdown documentation.

Best practices for using the Infrastructure Documentation Generator

This section outlines best practices and key considerations for using the Infrastructure Documentation Generator responsibly and effectively. By following these practices, you can support secure handling of AWS credentials, generate accurate and meaningful infrastructure insights, and collaborate more effectively with your team. This approach helps you leverage the system for security, cost, and performance optimization. At the same time, you can respect organizational policies, protect sensitive data, and maximize the value of the generated documentation and visualizations.

Using credentials securely

Always provide temporary credentials or assume roles instead of by using long-lived AWS access keys. This minimizes security risks while scanning your environments. Use AWS Identity and Access Management (IAM) users and roles with read-only permissions only, so the tool never modifies resources. For more information, see Grant least privilege and Security best practices in the IAM documentation.

Interpreting reports

You can choose from the following types of analysis reports:

- **Comprehensive** for a full overview
- **Security** to identify risky configurations
- **Cost** for optimization opportunities
- **Performance** to check resource efficiency

You can also add custom prompts for focused checks, for example, "Focus on compute resources and their security posture".

Managing large environments

For big accounts with thousands of resources, the system may chunk data before analysis. Users should expect slightly longer processing times. In such cases, pay special attention to the "Analysis Limitations" section of the report if some chunks were skipped or failed.

Exploring dependencies

Use the graph visualization in the UI to explore resource relationships. Filters, search, and zoom capabilities help navigate complex environments. Look for clusters of dependencies that can indicate tightly coupled services or potential single points of failure.

Maintaining cache awareness

The system caches recent analyses to speed up repeated queries. If your infrastructure changes, always trigger a fresh scan instead of relying solely on cached results. This practice helps to ensure that your report reflects the latest state.

FAQ

This section provides answers to frequently asked questions about the Infrastructure Documentation Generator.

What is the Infrastructure Documentation Generator ?

It's an automated tool that scans AWS environments to create comprehensive documentation and analysis, including dependency maps, configuration reports, and AWS Well-Architected Framework assessments.

What problems can the tool help solve?

The Infrastructure Documentation Generator addresses several critical challenges:

- Outdated documentation that's maintained manually
- Lack of visibility into infrastructure relationships
- Time-consuming compliance reporting
- Difficulty in identifying optimization opportunities
- Gaps in understanding resource dependencies
- Challenge of maintaining Well-Architected Framework alignment
- Communication barriers between technical and business teams

How does the Infrastructure Documentation Generator work?

The Infrastructure Documentation Generator tool operates through the following systematic process:

1. Performs automated discovery of AWS resources and configurations
2. Maps relationships and dependencies between services
3. Generates visual architecture diagrams and documentation
4. Analyzes configurations against the Well-Architected Framework

5. Produces actionable recommendations

It also does the following:

- Maintains real-time updates as infrastructure changes
- Creates customized reports for different stakeholders

What are the key benefits of the Infrastructure Documentation Generator?

Key advantages of the tool include:

- Eliminates manual documentation effort
- Helps to ensure always up-to-date infrastructure documentation
- Provides actionable optimization insights
- Improves cross-team collaboration
- Accelerates compliance reporting
- Reduces risk of configuration errors
- Enables data-driven infrastructure decisions

What technical capabilities does the tool rely on?

The tool leverages the following:

- AWS APIs and SDKs, such as AWS SDK for Python (Boto3)
- Amazon Bedrock for intelligent report generation
- React frameworks for building dependency graphs

Are there any limitations to the tool?

Although comprehensive, the Infrastructure Documentation Generator:

- Might need customization for unique use cases

- Works primarily with AWS services (limited hybrid capabilities)

How can my organization measure success if we use this tool?

Success metrics include the following:

- Reduction in documentation time
- Improved documentation accuracy
- Number of optimizations implemented
- Cost savings identified
- Compliance improvements
- User adoption rates

Resources

AWS resources

- Amazon Bedrock Documentation
- AWS Cloud Operations Blog: Best Practices
- Code repository for the Infrastructure Documentation Generator
- AWS SDK for Python (Boto3) documentation
- AWS Well-Architected Framework

Other resources

- Fast API
- Python documentation
- React Flow Library

Contributors

The following individuals contributed to this guide.

Author:

- Vijit Vashishtha, Delivery Consultant, AWS Professional Services

Co-authors:

- Mansi Doshi, Delivery Consultant - AppMod, AWS Professional Services
- Ishita Gupta, Associate Delivery Consultant, AWS Professional Services
- Rohit Saha, Associate Delivery Consultant, AWS Professional Services

Reviewers:

- Junaid Baba, Senior Delivery Consultant, AWS Professional Services
- Abhimanyu Chhabra, Solutions Architect, AWS

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	November 20, 2025