Architectural Blueprint and Implementation Guide: Replicating the Netlify Platform with Replit

The Philosophy and High-Level Architecture of a Modern PaaS

Modern Platform-as-a-Service (PaaS) offerings like Netlify have fundamentally reshaped web development by abstracting away the complexities of infrastructure management. Their success is not rooted in a single groundbreaking technology but in a philosophical shift towards a more streamlined, automated, and developer-centric workflow. To replicate such a platform, one must first understand and internalize these core principles before architecting the technical components. The foundation of this approach rests on two pillars: Composable Architecture and a Git-native operational model, commonly known as GitOps.

Deconstructing the Magic: The Core Principles of Composable Architecture and GitOps

The perceived "magic" of platforms like Netlify lies in their ability to transform a simple git push into a globally deployed, highly performant, and secure web application. This is achieved through a deliberate architectural philosophy that prioritizes decoupling, automation, and developer experience.

Composable Architecture: This modern approach to application design stands in stark contrast to traditional monolithic systems where all components are tightly interwoven.[1] In a composable architecture, an application is built from a collection of independent, reusable, and loosely coupled services that communicate via APIs. This model has several defining traits that are critical to replicate [1]:

- **Reusability:** Components are designed as self-contained building blocks that can be reused across different applications, ensuring consistency and

accelerating development time.

- **Loose Coupling:** Each service or module operates as a cloud-native, self-contained program. This independence means that one component can be updated, scaled, or replaced without impacting the others, dramatically improving maintainability and resilience.
- **Future-Proofing:** In a monolithic system, an outdated component can render the entire application obsolete. The modular nature of composable architecture allows individual parts of the system to be updated incrementally, preventing the accumulation of technical debt and ensuring the platform remains current.

A prominent implementation of this philosophy is the **JAMstack** (JavaScript, APIs, and Markup) architecture, which Netlify pioneered.[2] The core principle of JAMstack is to pre-render the entire frontend of a web application into a set of static files (HTML, CSS, JavaScript) during a build process.[5] These static assets are then served from a Content Delivery Network (CDN), making them incredibly fast and secure.[7] All dynamic functionality—such as user authentication, data fetching, or e-commerce transactions—is handled by reusable APIs, which are accessed via JavaScript running in the browser. These APIs can connect to third-party services or custom-built serverless functions.[4] This fundamental decoupling of the frontend (the "presentation layer") from the backend (the "data and logic layer") is the source of the platform's key benefits: superior performance, enhanced security, and effortless scalability.[4]

GitOps: The second core principle is the adoption of Git as the central operational paradigm. In a GitOps workflow, the Git repository is the single source of truth for the application's state.[6] Every action on the platform, most notably a deployment, is triggered by a Git event, such as a

git push or the creation of a pull request.[8] This approach yields a transparent, version-controlled, and collaborative development process. Every change is auditable, every deployment is tied to a specific commit, and features like instant rollbacks become trivial to implement by simply redeploying a previous commit. This Git-native workflow eliminates the need for manual deployment steps, complex CI/CD configurations, and dedicated DevOps personnel, allowing developers to focus entirely on their code.[8]

The profound implication of this model is that the platform's primary value is not in providing raw infrastructure primitives like virtual machines or databases, but in the seamless *orchestration* of these primitives. It connects a developer's local Git repository to a global network of servers, CDNs, and serverless compute environments through a highly automated and opinionated pipeline. The engineering

challenge, therefore, is not to reinvent cloud infrastructure but to build a sophisticated orchestration engine that delivers a world-class developer experience (DX).

System Overview: A High-Level Diagram of the Cloned Platform's Components

To structure the replication effort, the platform can be conceptually divided into five core macro-services, each with a distinct responsibility in the application lifecycle. The following diagram illustrates the flow of data and control from the developer's initial code push to the end-user's browser request.

Codefragment

```
graph TD
    subgraph Developer Workflow
        A --> B{Git Provider <br> (GitHub, GitLab)};
    end

    B -- Webhook Event --> C;

    subgraph Platform Backend
        C --> D;
        D -- Build Artifacts --> E;
        D -- Function Packages --> F;
        G <--> C;
        G <--> D;
        G <--> E;
        G <--> F;
    end

    subgraph User Interaction
        H -- Request --> E;
        H -- API Call --> F;
    end
```

```
  style A fill:#f9f,stroke:#333,stroke-width:2px
  style B fill:#f9f,stroke:#333,stroke-width:2px
  style H fill:#ccf,stroke:#333,stroke-width:2px
```

1. **Ingestion Layer:** This is the platform's "front door." It receives external signals, primarily webhooks from Git providers, validates them, and initiates the deployment workflow.
2. **Build Layer:** This is the engine of the platform. It takes the instructions from the Ingestion Layer, fetches the user's code, and executes the build process in a secure, isolated environment to produce the static site assets and serverless function packages.
3. **Delivery Layer:** This layer is responsible for storing the build artifacts and serving them to users globally with high performance. It manages the static assets and ensures that deployments are atomic and can be rolled back instantly.
4. **Compute Layer:** This layer handles dynamic, server-side logic. It provisions and manages the runtime for serverless functions, exposing them as HTTP endpoints that can be called by the frontend application or other services.
5. **Control Plane:** This is the user-facing management interface for the entire platform. It consists of a web dashboard, a public REST API, and a command-line interface (CLI) that allow users to configure their sites, manage deployments, and monitor activity.

Technology Stack Rationale: Mapping Core Functionality to Cloud-Native Services

Building a platform of this scale from scratch is a monumental task. The most pragmatic approach is to leverage existing, battle-tested cloud-native services as the building blocks for our clone. This strategy allows the development effort to focus on the orchestration logic and developer experience, which are the true differentiators, rather than on re-implementing fundamental infrastructure. The following table maps the conceptual components of our platform to a proposed technology stack, primarily using services from Amazon Web Services (AWS) for a consistent ecosystem, but noting alternatives where applicable.

Platform Component	Netlify's	Our Clone's	Justification

	Implementation (Inferred)	Proposed Tech Stack	
Git Event Ingestion	Custom Webhook Service	Node.js/Express API on Replit	A lightweight, event-driven server is ideal for handling high-volume, short-lived webhook requests.[11]
Build Orchestration	Internal Queuing System	AWS SQS (Simple Queue Service)	A message queue decouples ingestion from execution, providing resilience, scalability, and workload management.[2]
Build Environment	Custom Container Runtime	Docker on ephemeral compute (e.g., AWS Fargate)	Containers provide secure, reproducible, and isolated environments for executing untrusted user code.[14]
Static Asset Storage	Proprietary Object Storage	AWS S3 or Google Cloud Storage	Highly durable, scalable, and cost-effective object storage is the industry standard for hosting static assets.[16]
Global CDN	Multi-Cloud Edge Network	Cloudflare or AWS CloudFront	A CDN is essential for global performance. The choice impacts cost, security features, and multi-cloud flexibility.[7]
Serverless Functions	Managed AWS Lambda	Direct AWS Lambda Integration	Directly leveraging a mature FaaS provider like AWS Lambda abstracts away the

			entire serverless runtime management.[4]
API Gateway	Custom API Gateway	AWS API Gateway	Provides a managed, scalable "front door" for serverless functions, handling routing, authorization, and throttling.[4]
Control Plane API	Internal REST API	Node.js/Express API with a Database (PostgreSQL)	A standard REST API architecture provides a flexible and well-understood foundation for the platform's management interfaces.[21]
Dashboard UI	React SPA	React/Vite SPA on Replit	A modern single-page application framework allows for a rich, interactive user experience for the management dashboard.[22]

Comparative Analysis of Open-Source Alternatives

Before finalizing the architecture, it is instructive to analyze existing open-source PaaS solutions to understand established patterns and identify opportunities for differentiation. Platforms like Coolify [23], CapRover [25], and Dokku [23] offer self-hosted alternatives to managed services like Netlify and Heroku.

- **Common Patterns:** These platforms almost universally rely on Docker as the core unit of deployment. They provide a user-friendly interface (web UI and/or CLI) on top of a user's own server(s) to manage the lifecycle of Docker containers. They typically handle reverse proxying (often with Nginx or Traefik), SSL certificate

management, and database provisioning as one-click services. This validates the architectural choice of using containers as the fundamental building block for application deployment.
- **Key Differences and Strategic Implications:** While these platforms provide excellent control and cost-effectiveness, their focus is often on general-purpose Docker hosting. A Netlify clone, by contrast, is more specialized. It is built around the JAMstack and GitOps philosophy, with a deep focus on the frontend developer workflow. Features like atomic deploys of static assets, integrated serverless functions as first-class citizens, and deploy previews for every pull request are what set the Netlify model apart.

This analysis reveals a clear segmentation in the PaaS market based on the level of abstraction and control. At one end, self-hosted solutions like Coolify offer maximum control over the infrastructure. At the other, managed platforms like Netlify offer maximum convenience by abstracting the infrastructure away entirely. A successful clone must consciously decide where it wants to sit on this spectrum. The architecture presented in this report aims to replicate the high-abstraction, high-convenience model of Netlify. However, this blueprint also highlights strategic forks in the road—for instance, choosing a more flexible but complex backend like Kubernetes over pure Lambda, or deeply integrating stateful database services as Northflank does [26]—that could allow a new platform to carve out a unique niche by offering a different balance of convenience and power. The technical decisions that follow are therefore framed not just as engineering choices, but as strategic moves within a competitive landscape.

The Ingestion Layer: Git Integration and Build Orchestration

The Ingestion Layer serves as the primary interface between the developer's workflow and the platform's internal machinery. Its sole purpose is to listen for Git events, validate their authenticity, and schedule a corresponding build job in a way that is both secure and scalable. This layer is composed of two main components: a Webhook Gateway to receive events and a Build Scheduler (queuing system) to manage the workload.

The Webhook Gateway: A Secure Service for Git Events

The entry point for every deployment is the Webhook Gateway. This is a lightweight web service, ideally built with a non-blocking, event-driven framework like Node.js with Express, designed for high availability and low-latency responses. It exposes a single, dedicated endpoint (e.g., /api/v1/hooks/git) that users will configure in their Git provider's settings (GitHub, GitLab, Bitbucket).[11] This configuration instructs the Git provider to send an HTTP POST request to this endpoint whenever a specified event, such as a

push to a branch or the creation of a pull_request, occurs.[28]

Security and Validation: The gateway's most critical responsibility is security. It must not blindly trust incoming requests. All major Git providers sign their webhook payloads with a pre-shared secret. For GitHub, this signature is sent in the x-hub-signature-256 HTTP header.[12] The gateway must perform the following validation steps for every request:

1. Check for the presence of the signature header. If it is missing, the request is immediately rejected with a 401 Unauthorized status.
2. Compute its own HMAC SHA256 signature of the raw request body using the secret key stored securely within the platform's configuration.
3. Compare its computed signature with the one provided in the header. If they do not match, the request is rejected. This cryptographic verification ensures that the request genuinely originated from the configured Git provider and has not been tampered with in transit.[12]

Decoupled Responsibility: The gateway's job ends after validation and job creation. It should parse the validated payload to extract essential information for the build job: the repository's clone URL, the specific commit SHA, the branch name, and identifiers for the user and site. It then formats this information into a JSON message and places it onto a message queue. Immediately after enqueuing the job, it must return a 202 Accepted status code to the Git provider. This rapid response is crucial, as Git providers have short timeouts and will mark a webhook as failed if a response is not received quickly.[27] The gateway should not wait for the build to start or complete; its role is strictly to ingest and delegate.

The Build Scheduler: A Queuing System for Resilience and Scale

A direct, synchronous link between the webhook gateway and the build system would be a fragile and unscalable architecture. A single user pushing 100 commits in rapid succession could easily overwhelm the system, causing timeouts and failed deployments. To solve this, a message queue, such as AWS Simple Queue Service (SQS), is an architectural necessity.

The queue acts as a durable, elastic buffer between the high-volume, bursty nature of webhook events and the resource-intensive, long-running nature of the build process. When the Webhook Gateway receives a valid event, it simply adds a message to the queue. A separate fleet of build workers then consumes messages from this queue at a controlled rate.

This decoupled architecture provides several key benefits:

- **Resilience:** If the build system is down or overloaded, messages remain safely in the queue until the system recovers. No deployments are lost.
- **Scalability:** The number of build workers can be scaled up or down independently based on the queue depth, allowing the platform to efficiently handle both quiet periods and intense bursts of activity. This is the foundation for managing concurrency and offering features like "concurrent builds".[29]
- **Cost Management:** The pay-per-use model of services like SQS and the ability to scale build workers based on demand are essential for controlling the operational costs of a PaaS, whose business model is often tied to resource consumption like "build minutes".[2]

The message on the queue is a self-contained "work order" for the build system. It is a JSON object containing all the information a build worker needs to execute the job without any further lookups:

JSON

```
{
  "jobId": "build-uuid-12345",
  "siteId": "site-uuid-abcdef",
  "userId": "user-uuid-67890",
```

```
  "repositoryUrl": "https://github.com/user/repo.git",
  "branchName": "main",
  "commitSha": "a1b2c3d4e5f6...",
  "buildContext": "production"
}
```

The Build Environment: Secure, Isolated, Container-Based Build Runners

The heart of the build system is the environment where user code is executed. This environment must be reproducible, isolated, and secure. Docker containers are the ideal technology for this purpose.[14] They provide a lightweight, ephemeral sandbox for each build, ensuring that dependencies and configurations from one build cannot interfere with another.[15]

The execution flow is managed by a fleet of "build workers." These can be implemented as an Auto Scaling Group of EC2 instances or, more ideally, using a serverless container platform like AWS Fargate, which eliminates the need to manage the underlying virtual machines. These workers run a simple loop:

1. Poll the SQS queue for a new build job message.
2. If a message is received, process it. If the queue is empty, wait.

Upon receiving a job message, the worker orchestrates the following steps:

1. **Container Launch:** It launches a fresh, ephemeral Docker container. The container is started from a pre-built base image that includes common runtimes and tools (e.g., Node.js, Python, Go, Git).
2. **Code Checkout:** Inside the container, it clones the user's repository using the repositoryUrl and checks out the specific commitSha from the job message.
3. **Build Execution:** It inspects the repository for configuration files (e.g., netlify.toml, package.json) to determine the build command and publish directory. It then executes the user-defined build command (e.g., npm install && npm run build).
4. **Log Streaming:** Throughout the build process, the standard output and standard error streams from the container are captured. These logs are streamed in real-time to a centralized logging service (like Amazon CloudWatch Logs) and simultaneously pushed to the platform's database, allowing the user to view the live build progress in the dashboard UI.

5. **Artifact Archiving:** Upon successful completion of the build command, the worker identifies the specified publish directory (e.g., build, dist). It creates a compressed archive (e.g., a .zip file) of this directory's contents.
6. **Artifact Upload:** The worker uploads this archive to a temporary location in the platform's object storage (AWS S3), tagged with the unique jobId.
7. **Status Update:** The worker updates the status of the build job in the platform's database (e.g., to SUCCESS or FAILED).
8. **Container Termination:** The Docker container is destroyed. This is a critical step to ensure that no state, artifacts, or secrets are left behind, guaranteeing a clean slate for the next build.

Managing User Code: Security and Execution

The build environment represents the platform's most significant security vulnerability. The system is designed to execute arbitrary code provided by users, so it must operate under a zero-trust, "hostile tenant" model from the very beginning. Security cannot be an afterthought; it must be a foundational architectural principle.

Several layers of mitigation are required to create a secure execution sandbox:

- **Principle of Least Privilege:** Containers must be run as a non-root user. The base Docker image should include a USER instruction to switch to a low-privilege user before any user code is executed. This dramatically reduces the potential impact of a container breakout, as an attacker would not have root privileges on the host machine.[31]
- **Network Isolation:** By default, build containers should have no network access. A strict firewall policy should be applied, allowing outbound connections only to a whitelist of essential, trusted domains, such as official package registries (registry.npmjs.org, pypi.org). All inbound connections must be blocked. This prevents malicious build scripts from probing the internal network or exfiltrating data to arbitrary servers.
- **Resource Constraints:** Each build container must be launched with strict resource limits on CPU, memory, and total execution time. This prevents denial-of-service attacks where a malicious or poorly configured build script consumes excessive resources, impacting the stability of the host and the performance of other tenants' builds.
- **Ephemeral Filesystems:** The container's filesystem is ephemeral. Any changes

made during the build are discarded when the container is terminated, preventing state from leaking between builds. The only persistent output is the explicitly uploaded artifact archive.
- **Minimal Worker Permissions:** The build worker instances themselves must operate with a minimal set of IAM permissions. They should only have the permissions necessary to read from the SQS queue, write to the logging service, and upload artifacts to a specific prefix in the S3 bucket. They should have no access to other parts of the platform's infrastructure or other customers' data.

By implementing these security measures, the build system is transformed from a simple CI runner into a robust, multi-tenant secure execution environment, capable of safely handling code from thousands of different users.

The Delivery Layer: Atomic Deployments and Global Distribution

Once the Build Layer has successfully produced a set of static assets, the Delivery Layer takes over. Its responsibilities are to store these assets durably, serve them to users across the globe with minimal latency, and, most importantly, manage updates in a way that guarantees the site is never in a broken or inconsistent state. The cornerstone of this layer is the concept of "atomic deployments."

Static Asset Storage Strategy: Leveraging Object Storage

The definitive source of truth for all deployed assets will be a cloud object storage service, such as Amazon S3 [16] or Google Cloud Storage.[17] These services provide virtually unlimited scalability, extremely high durability (often 99.999999999%), and cost-effective storage, making them the ideal foundation for this layer.

A crucial architectural decision is the directory structure used within the storage bucket. To enable atomic deployments and instant rollbacks, every deployment's assets must be stored in a unique and immutable location. A simple file synchronization to a single "live" directory is explicitly not atomic and can lead to inconsistent states where a user's browser loads a new HTML file but receives old,

incompatible CSS or JavaScript assets.[36]

Instead, a versioned directory structure is employed. Each successful build artifact is uploaded to a path prefixed by the site's unique identifier and the deploy's unique identifier. For example:
s3://<bucket-name>/sites/<site-id>/deploys/<deploy-id-123>/index.html
s3://<bucket-name>/sites/<site-id>/deploys/<deploy-id-123>/css/main.css
...
This ensures that the complete set of files for any given deploy is stored as a self-contained, immutable unit. The assets for a previous deploy, say deploy-id-122, remain untouched in their own directory, readily available for a rollback.

The Global Edge Network: Integrating a CDN

To achieve the low-latency performance expected of a modern web platform, serving assets directly from a single object storage bucket is insufficient. A Content Delivery Network (CDN) is required to cache copies of the assets at edge locations around the world, physically closer to the end-users.[7] The platform will integrate with a major CDN provider like AWS CloudFront [34] or Cloudflare.[38]

The CDN distribution is configured with the object storage bucket as its "origin." When a user requests a file, the request hits the nearest CDN edge location. If the file is in the cache, it is served immediately. If not, the CDN fetches it from the origin S3 bucket, serves it to the user, and caches it for future requests.

The choice between CloudFront and Cloudflare represents a significant architectural fork with long-term implications:

- **AWS CloudFront:** As an AWS-native service, it offers seamless integration with S3, IAM for access control, and other AWS services. A key advantage is that data transfer from S3 to CloudFront is free, as it remains within the AWS network backbone.[40] However, its configuration can be more complex, and its security features, like a Web Application Firewall (WAF), are often configured as separate, add-on services.[41]
- **Cloudflare:** Cloudflare operates as a reverse proxy at the DNS level and bundles an extensive suite of services, including a global CDN, DNS management, and robust security features like DDoS mitigation and a WAF, into a single, easier-to-manage package.[42] It is often more multi-cloud friendly and has

programs designed to reduce or eliminate data egress fees from cloud providers, which can lead to significant cost savings at scale.[18]

The prompts for infrastructure setup will need to diverge significantly based on this decision, as it affects everything from domain management and SSL provisioning to the cost model of the platform.

The Mechanics of Atomic Deploys and Instant Rollbacks

The "atomic deploy" is the feature that guarantees a site update is an all-or-nothing operation, preventing users from ever seeing a partially updated, broken site.[5] This is achieved not by overwriting files, but by atomically changing a pointer that directs the CDN to the new set of immutable assets.

The process unfolds as follows:

1. **Upload:** A new, successful build (e.g., deploy-id-123) has its assets uploaded to its unique, immutable directory in S3: .../deploys/deploy-id-123/. The currently live site is being served from a different directory, for example, .../deploys/deploy-id-122/.
2. **Atomic Swap:** The "live" version of the site is determined by the CDN's origin path configuration. To make the new deploy live, the platform's deployment service makes a single API call to the CDN provider (e.g., an UpdateDistribution call to CloudFront) to change the origin path from .../deploys/deploy-id-122 to .../deploys/deploy-id-123. This configuration change is an atomic operation at the CDN level. Within moments, all new requests to the CDN will be served from the new directory.
3. **Cache Invalidation:** Immediately following the atomic swap, the deployment service issues a cache invalidation command to the CDN (e.g., for the path /*). This purges the old files from the CDN's edge caches, forcing it to fetch the new versions from the new origin path on the next request.
4. **Instant Rollback:** A rollback is the exact same process in reverse. If deploy-id-123 is found to be faulty, an operator (or an automated system) can trigger a rollback. The deployment service simply makes another atomic API call to the CDN, changing the origin path back to .../deploys/deploy-id-122. Since the old assets were never deleted, they are immediately available to be served again.

This pointer-swapping mechanism is the core technical implementation of the atomic

deployment and instant rollback features that are central to the Netlify developer experience.

Implementing Deploy Previews

Deploy Previews, which provide a unique, shareable URL for every pull request, are a natural extension of the atomic deployment architecture.[8] Since every commit's build results in a unique, immutable set of assets stored at a stable path (

.../deploys/<deploy-id>/), creating a preview is a matter of routing.

When a build is triggered for a pull request, the platform does the following:

1. The build runs as usual, and its assets are uploaded to its unique S3 directory.
2. The platform generates a unique, stable subdomain for this specific deploy preview. The convention is often deploy-preview-<PR-number>--<site-name>.<platform-domain>.com.
3. The platform programmatically configures the DNS and/or CDN to route requests for this new subdomain to the corresponding S3 directory. This can be done by creating a new, temporary CDN distribution or, more efficiently, by using a single wildcard CDN distribution (*--<site-name>.<platform-domain>.com) and adding a routing rule or behavior that maps the specific preview subdomain to the correct S3 origin path.

This allows developers and stakeholders to review the exact changes from a pull request in a live, production-like environment before they are merged into the main branch, dramatically improving the quality and velocity of the development cycle.

The Compute Layer: Serverless and Edge Runtimes

While the Delivery Layer excels at serving static content, modern web applications require dynamic, server-side logic to handle tasks like API requests, form submissions, and user authentication. The Compute Layer provides this capability through a serverless execution model, abstracting away the complexities of managing servers

and allowing developers to deploy backend code as easily as frontend code.

Serverless Function Architecture: From Repository to Runtime

The platform's core promise is a seamless workflow. For serverless functions, this means a developer simply places a code file in a designated directory, and it automatically becomes a live API endpoint. This requires a sophisticated "meta-build" process that runs in parallel with the static site build.

1. **Detection:** During the build process, the build runner scans the user's repository for a conventionally named directory, such as netlify/functions or api/.[19] It identifies all valid function files within this directory (e.g., .js, .ts, .go files).
2. **Packaging:** For each detected function, the build runner initiates a separate packaging process. For a Node.js function, this involves:
    - Creating an isolated directory for the function.
    - Copying the function's source code and any associated package.json.
    - Running the appropriate package manager command (e.g., npm install --production) to install only the necessary runtime dependencies.
    - Creating a deployment package, which is typically a .zip archive containing the function's code and its node_modules folder.[47]
3. **Deployment:** After the static assets are uploaded, the platform's deployment service takes these function packages and programmatically deploys them to a Function-as-a-Service (FaaS) provider, such as AWS Lambda.[49] Using the AWS SDK, it will either create a new Lambda function or update the code of an existing one. Each function version is tied to the parent site deploy, ensuring that a rollback of the site also rolls back the associated functions.

The API Gateway: The Public-Facing Interface for Functions

By default, FaaS functions like AWS Lambda are not directly accessible from the public internet via a simple HTTP request. They are invoked via the cloud provider's API. To expose them as standard web endpoints, an API Gateway is required.[20]

The platform will programmatically manage an AWS API Gateway instance for each

site that uses functions.

- **Provisioning:** When the first function for a site is deployed, the platform creates a new API Gateway.
- **Routing:** For each serverless function, a corresponding route is created in the API Gateway. This route maps a user-friendly, conventional URL path (e.g., https://<site-name>.netlify.app/.netlify/functions/hello) to an invocation of the specific backend Lambda function.[46]
- **Integration:** The connection between the API Gateway route and the Lambda function is configured as a **Lambda Proxy Integration**.[48] This is a critical choice that simplifies the architecture immensely. In this mode, the API Gateway forwards the entire raw HTTP request—including headers, query parameters, path, and body—as a JSON event to the Lambda function. It then takes the entire response object returned by the Lambda (including status code, headers, and body) and transforms it back into an HTTP response for the client. This avoids the need for complex request and response mapping templates within the API Gateway itself.

Runtime Environment and Context Injection

A key feature that enhances the developer experience of Netlify Functions is the injection of a rich context object into the function handler.[53] This object provides valuable metadata that would otherwise be difficult or impossible for the function to obtain. Replicating this is paramount to replicating the DX.

The API Gateway layer is responsible for assembling this context. Before invoking the backend Lambda function, it can gather information such as:

- **Geolocation Data:** The client's IP address can be used to look up their approximate geographic location (city, country, timezone).[53]
- **Site Metadata:** Information about the site itself, such as its ID and name, can be injected.[53]
- **User Identity:** If the platform includes an authentication service, the API Gateway can validate a user's JSON Web Token (JWT) and inject their identity information into the context.

This metadata is then passed to the Lambda function as part of the invocation event payload, making it available to the developer's code. Similarly, environment variables

defined by the user in the platform's dashboard are securely passed to the Lambda function's runtime configuration, making them accessible within the code.[52]

Architecting for Edge Functions (Advanced Topic)

Edge Functions represent a different paradigm of serverless compute. Unlike standard serverless functions that run in a centralized region (e.g., us-east-1), Edge Functions execute on the CDN's edge nodes, physically close to the user.[19]

This architecture provides two main benefits:

- **Ultra-Low Latency:** Since the code runs at the edge, it can respond to requests in milliseconds, making it ideal for tasks like personalization, A/B testing, and header manipulation.
- **Request Interception:** Edge Functions can run *before* a request hits the cache or the origin server, allowing them to modify the incoming request or the outgoing response on the fly.

However, this performance comes with significant constraints. Edge runtimes are highly optimized and sandboxed (e.g., using V8 Isolates instead of full containers), which means they have much shorter execution limits (e.g., 50ms vs. 10+ seconds), smaller memory footprints, and no access to native Node.js APIs like the filesystem (fs).[19]

To support Edge Functions, the platform's architecture would need to integrate with a provider like Cloudflare Workers or AWS Lambda@Edge. The deployment pipeline would require a separate path for these functions, as they have different packaging requirements (often a single script file rather than a ZIP archive with dependencies) and are deployed to the CDN configuration rather than to a regional Lambda service.

The Control Plane: Dashboard, API, and CLI

The Control Plane is the collection of user-facing tools that allows developers to interact with and manage their resources on the platform. It is the tangible manifestation of the platform's developer experience. A well-architected control plane

consists of a central, authoritative REST API that serves as the "brain" of the platform, with the web dashboard and command-line interface (CLI) acting as its primary clients. This API-first approach is fundamental to building a scalable and maintainable system.

Designing the Developer Dashboard: UI/UX Principles for a PaaS

The web dashboard is the primary graphical interface for the platform. Its design should be guided by user-centered principles, focusing on clarity, simplicity, and goal-oriented navigation.[56] The dashboard is not merely a set of controls; it is a critical tool for user activation and retention. It must be designed around the developer's "jobs to be done."

For a GitOps-based PaaS, the primary user journey revolves around the deployment lifecycle. Therefore, the dashboard must prioritize and surface information related to this workflow. Key views and components should include:

- **Site Overview/Dashboard:** This is the landing page for a specific site. It should provide an "at-a-glance" summary of the most recent and relevant activity, such as the status of the latest production deploy, a list of active branch deploys and deploy previews, and summaries of other integrated features like form submissions.[22] This view directly supports the user's need to quickly assess the current state of their project.
- **Deploys List:** A detailed, historical log of all deploys for a site. Each entry should display the deploy status (e.g., Building, Published, Failed), the associated Git commit message and SHA, the author, and the time of the deploy. Crucially, this view must provide direct links to the detailed build logs and a prominent one-click "Publish deploy" button for any previous successful deploy, enabling the instant rollback feature.[58]
- **Live Logs Viewer:** A real-time, streaming view of the build logs from the build runner container. This is essential for debugging failed builds and understanding the build process. The interface should be clean, searchable, and provide clear timestamps.
- **Site Settings:** A well-organized section with forms for managing all site configurations. This includes build settings (build command, publish directory, base directory), custom domains, DNS management, environment variables (with scoping per deploy context), and controls for enabling and configuring add-on

services.[58]

The design should follow a clear information hierarchy, placing the most critical information (like the status of the current deploy) in the most prominent positions, following established F- and Z-pattern reading behaviors.[57] Drawing inspiration from the clean interfaces of open-source tools like CapRover [25] and Coolify [60] can provide a solid foundation, but the final design must be tailored to the specific Git-centric workflow of this platform.

Architecting the Public REST API: The Engine of the Platform

The REST API is the most critical component of the control plane. It is the single source of truth and the programmatic interface for every action performed on the platform. Both the web dashboard and the CLI will be built as clients consuming this API. This ensures consistency, reduces code duplication, and allows for third-party integrations and automation.

The API design should be resource-oriented, following standard RESTful principles. The core resources and endpoints can be modeled directly on Netlify's own public API, which provides a proven and well-documented structure.[21]

Resource	Endpoint	Method	Description	Key Data Model Fields
Site	/api/v1/sites	POST	Create a new site by linking a Git repository.	name, repo_url, repo_branch, build_command, publish_dir
	/api/v1/sites/{site_id}	GET	Retrieve the full configuration and status of a specific site.	id, name, url, admin_url, build_settings, custom_domain
	/api/v1/sites/{site_id}	PATCH	Update the settings for a site.	build_command, publish_dir, env_vars

Deploy	/api/v1/sites/{site_id}/deploys	GET	List all historical and active deploys for a site.	id, state, commit_sha, commit_message, branch, deploy_url, created_at
	/api/v1/deploys/{deploy_id}	GET	Get the detailed status and log of a specific deploy.	id, status, log_output, error_message
	/api/v1/deploys/{deploy_id}/restore	POST	Atomically set a previous deploy as the current live production version.	deploy_id
Environment Variable	/api/v1/sites/{site_id}/env	GET	List all environment variables for a site.	key, values (array of objects with value and context)
	/api/v1/sites/{site_id}/env	POST	Create or update an environment variable with contextual values.	key, value, context (e.g., production, deploy-preview, branch-deploy)
Custom Domain	/api/v1/sites/{site_id}/domains	POST	Add a custom domain to a site and begin the verification process.	domain_name
	/api/v1/sites/{site_id}/domains/{domain_name}	GET	Check the verification and SSL status of a custom domain.	domain_name, dns_status, ssl_status

This API will be built as a standard web application (e.g., using Node.js/Express) backed by a relational database (e.g., PostgreSQL) to store all state information about

users, sites, deploys, and configurations.

Building the Command-Line Interface (CLI)

The CLI is an essential tool for power users, scripting, and integration with external CI/CD systems. It provides a fast and efficient way to perform common platform operations directly from the terminal.[19]

The CLI will be a thin client that wraps the public REST API. Its primary responsibilities are:

- **Authentication:** Managing user authentication, typically via a long-lived API token that the user generates from the web dashboard. The CLI securely stores this token locally and includes it as an Authorization header in all API requests.
- **Command Mapping:** Mapping user-friendly commands (e.g., replit-clone deploy) to the corresponding API calls (e.g., POST /api/v1/sites/{site_id}/deploys).
- **Input/Output Handling:** Parsing command-line arguments and flags, and formatting the JSON responses from the API into a human-readable text output.
- **Local Development:** A key feature, often implemented as replit-clone dev, would be to run a local development server that emulates the production environment. This includes loading environment variables from the platform, proxying requests to serverless functions, and handling redirects, providing a high-fidelity local testing experience.[19]

By building the CLI on top of the public API, the platform ensures that all operations are consistent and authorized through the same central logic, whether they originate from the web UI, the command line, or a third-party script.

Essential Platform Services

Beyond the core deployment pipeline and control plane, a production-grade PaaS requires several essential supporting services to provide a complete and professional user experience. These services, particularly custom domain management and automated SSL provisioning, are complex, asynchronous systems that are critical for

any application intended for public use.

Custom Domain Management

The ability for users to map their own custom domains (e.g., www.example.com) to their deployed sites is a fundamental requirement. This process involves a coordinated effort between the user, the platform, and the global Domain Name System (DNS).

The workflow must be modeled as an asynchronous state machine, as DNS propagation can take anywhere from minutes to hours.

1. **Initiation (User Action):** The user enters their custom domain into the platform's dashboard. The control plane API receives this request, creates a new domain record in its database associated with the user's site, and sets its initial state to pending_verification.
2. **Instruction (Platform Response):** The dashboard displays the necessary DNS records the user must create with their domain registrar or DNS provider. Typically, this involves creating either an A record pointing to the CDN's IP address(es) or a CNAME record pointing to the platform's default site URL (e.g., <site-name>.replit-clone.app).[64] For apex domains (e.g., example.com), an A record is usually required, while for subdomains (e.g., www.example.com), a CNAME is preferred.
3. **Verification (Background Process):** The platform must run a background job that periodically performs a DNS query for the user's domain. This worker checks if the DNS records have been updated to match the required values.
4. **State Transition:** Once the DNS query confirms that the records are correctly configured, the background worker updates the domain's status in the database from pending_verification to verified. This transition signals that the domain is now correctly pointing to the platform's infrastructure and can proceed to the next step: SSL provisioning. If the records are incorrect after a certain period, the status can be moved to an error state, and the user is notified.

Automated SSL Provisioning with Let's Encrypt

In the modern web, HTTPS is non-negotiable. Providing free, automated SSL

certificates is a table-stakes feature for any hosting platform.[45] Let's Encrypt has become the industry standard for this, offering certificates through an automated protocol called ACME (Automated Certificate Management Environment).[65]

To integrate this, the platform must act as a sophisticated ACME client.

1. **Challenge Initiation:** Once a domain's status becomes verified, the platform's certificate management service initiates a certificate request to the Let's Encrypt API. To prove ownership of the domain, Let's Encrypt provides a "challenge." The most robust and automatable challenge type for a PaaS is the DNS-01 challenge.[66]
2. **DNS-01 Challenge Fulfillment:** For a DNS-01 challenge, Let's Encrypt provides a unique token. The platform must prove control over the domain by creating a specific TXT DNS record with this token as its value (e.g., _acme-challenge.www.example.com).
3. **Managed DNS Requirement:** This step reveals a critical architectural requirement. To programmatically create this TXT record, the platform must have API-level control over the user's DNS zone. While some DNS providers offer APIs, the most reliable way to ensure this capability is for the platform to provide its own managed DNS service. The user is then instructed to delegate their domain's nameservers (NS records) to the platform's nameservers. This gives the platform the authority to automatically manage all necessary DNS records for verification, SSL provisioning, and ongoing renewals.[62]
4. **Certificate Issuance and Installation:** After creating the TXT record, the platform notifies Let's Encrypt, which then verifies the record. Upon successful verification, Let's Encrypt issues the SSL certificate. The platform's certificate manager securely stores the certificate and its private key, and then makes an API call to the CDN provider (CloudFront or Cloudflare) to upload the certificate and associate it with the custom domain.
5. **Automated Renewal:** Let's Encrypt certificates have a short 90-day validity period to encourage automation.[66] The platform must have a scheduled background job that automatically renews certificates well before they expire (e.g., at the 60-day mark) by repeating the ACME challenge process. This ensures that HTTPS remains active without any manual intervention from the user.

Framework for Extensible Add-ons (Future Vision)

To move beyond a simple hosting platform and become a true application development ecosystem, the architecture should be designed for extensibility. Features like Netlify Forms [3] or Netlify Identity [3] are not part of the core deployment pipeline but are value-add services.

An extensible architecture would be based on microservices. Each add-on (e.g., a "Forms Service" or an "Identity Service") would be a separate, self-contained application. When a user enables an add-on for their site via the control plane, the platform would:

1. Provision the necessary resources for that service (e.g., a new database table for form submissions).
2. Use the internal API to configure the necessary integrations. For example, the Forms service might need to inject a small piece of JavaScript into the site's HTML during the build post-processing step or configure an API Gateway route to receive form submissions.

This microservices approach allows new features to be developed and deployed independently of the core platform, fostering innovation and allowing the platform to evolve over time.

The Replit Implementation Strategy: A Phased Prompting Guide

This section translates the architectural blueprint into a practical, step-by-step implementation plan. It is structured as a series of detailed, layered prompts designed to be fed into an AI coding assistant within the Replit environment. This approach builds the system incrementally, allowing for testing and verification at each stage. Each prompt set represents a distinct, functional component of the platform.

Phase 1: Building the Core CI/CD Pipeline

This phase focuses on creating the fundamental workflow: receiving a Git push, building the code in a container, and storing the result.

Prompt Set 1.1 (Webhook Gateway)

- **Prompt 1.1.1:** "Create a new Replit project using the Node.js template. Initialize an Express server that listens on the default port. Create a single POST endpoint at /api/v1/hooks/git. This endpoint should log the raw request body to the console and respond with a JSON object {"status": "received"} and a 200 status code."
- **Prompt 1.1.2:** "Modify the /api/v1/hooks/git endpoint. Use middleware to verify the x-hub-signature-256 header sent by GitHub. Read a secret key from a Replit environment variable named GITHUB_WEBHOOK_SECRET. Use the crypto module to create an HMAC SHA256 digest of the raw request body. If the calculated signature does not match the one in the header, respond immediately with a 401 Unauthorized status. If it matches, proceed to the main handler."
- **Prompt 1.1.3:** "In the main handler for the webhook, parse the JSON payload. Extract the repository clone URL (repository.clone_url), the branch name (from the ref field, e.g., refs/heads/main), and the latest commit SHA (after). Log these three values to the console."

Prompt Set 1.2 (Queuing)

- **Prompt 1.2.1:** "Install the AWS SDK for JavaScript v3 (@aws-sdk/client-sqs). Configure an SQS client using credentials (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, AWS_REGION) stored in Replit secrets."
- **Prompt 1.2.2:** "Modify the webhook handler. After successfully validating and parsing the Git payload, construct a JSON object containing repositoryUrl, branchName, and commitSha. Convert this object to a string and send it as a message to an AWS SQS queue. The queue URL should be stored in a Replit secret named SQS_QUEUE_URL. After successfully sending the message, respond to the original HTTP request with a 202 Accepted status."

Prompt Set 1.3 (Build Worker)

- **Prompt 1.3.1:** "Create a separate Replit project named 'build-worker' using the Node.js template. Install the AWS SDK (@aws-sdk/client-sqs). Write a script that continuously polls the SQS queue specified by the SQS_QUEUE_URL secret for new messages. When a message is received, log its body to the console and then delete the message from the queue."

Prompt Set 1.4 (Containerized Build)

- **Prompt 1.4.1:** "This prompt requires a local Docker environment or a cloud environment with Docker installed, as Replit's default environment may not support Docker daemon access. The logic would be implemented in the build

worker. Write a function in the build worker that takes repositoryUrl, branchName, and commitSha as input. This function should use Node.js's child_process.exec to perform the following shell commands:
    1. Create a temporary, unique directory (e.g., /tmp/build-xyz).
    2. git clone --branch <branchName> --single-branch <repositoryUrl>. inside the directory.
    3. git checkout <commitSha>.
    4. Log the output of each command."
- **Prompt 1.4.2:** "Extend the previous function. After cloning the repository, it should detect the project type. If a package.json file exists, run npm install followed by npm run build. If a requirements.txt file exists, run pip install -r requirements.txt. Capture and stream the standard output and standard error of these build commands to the console in real-time."

Prompt Set 1.5 (Artifact Upload)

- **Prompt 1.5.1:** "Install the AWS SDK (@aws-sdk/client-s3) and a library for creating zip archives, like archiver. In the build worker, after a successful build, identify the publish directory (assume build or dist for now). Use archiver to create a zip file of this directory's contents."
- **Prompt 1.5.2:** "Using the S3 client, upload the generated zip file to an S3 bucket specified by the S3_BUCKET_NAME secret. The object key in S3 should follow the pattern build-artifacts/<site-id>/<deploy-id>.zip. For now, use a hardcoded site ID and a randomly generated UUID for the deploy ID."

Phase 2: Integrating the Delivery Layer

This phase makes the built site publicly accessible via a CDN and implements the atomic deployment logic.

Prompt Set 2.1 (CDN Provisioning)

- **Prompt 2.1.1:** "Write a TypeScript script using the AWS CDK. Define a new CDK stack. Within the stack, create an S3 bucket for website hosting. Then, create a CloudFront distribution. Configure the distribution's default origin to point to the S3 bucket. The script should be runnable from the command line to deploy the infrastructure."

Prompt Set 2.2 (Atomic Deploy Logic)

- **Prompt 2.2.1:** "Create a new 'deploy-service' Replit project. This service will have an endpoint that is triggered after a build is successfully uploaded. In this service, write a function that takes a deployId as input. This function should:
  1. Unzip the artifact from s3://<bucket>/build-artifacts/.../<deploy-id>.zip into a new S3 prefix: s3://<bucket>/deploys/<deploy-id>/.
  2. Use the AWS SDK (@aws-sdk/client-cloudfront) to get the current configuration of a CloudFront distribution (ID stored in a secret).
  3. Modify the distribution's origin path to point to the new /deploys/<deploy-id> directory.
  4. Execute the UpdateDistribution command with the modified configuration."

Prompt Set 2.3 (Cache Invalidation)

- **Prompt 2.3.1:** "After the UpdateDistribution call in the deploy service succeeds, add another AWS SDK call to create a cache invalidation for the same distribution. The invalidation path should be /* to clear all cached files."

Phase 3: Developing the Serverless Compute Engine

This phase adds the ability to deploy and run serverless functions.

Prompt Set 3.1 (Function Detection & Packaging)

- **Prompt 3.1.1:** "In the build worker, add a step to check for the existence of a netlify/functions directory in the cloned repository. If it exists, iterate through each .js file in that directory."
- **Prompt 3.1.2:** "For each function file found, create a separate temporary directory. Copy the function file into it. If a root-level package.json exists, copy it as well. Run npm install --production inside this directory. Finally, create a unique zip archive for each function, containing the function file and its node_modules."

Prompt Set 3.2 (Lambda Deployment)

- **Prompt 3.2.1:** "In the deploy service, after the main site is deployed, process the function zip archives. For each function, use the AWS SDK (@aws-sdk/client-lambda) to call updateFunctionCode if the function already exists, or createFunction if it's new. The function name should be predictable, like

<site-id>-<function-name>. The function's execution role ARN should be pre-configured and stored as a secret."

Prompt Set 3.3 (API Gateway Provisioning)

- **Prompt 3.3.1:** "Using the AWS CDK, extend the infrastructure script. Create an AWS API Gateway (HTTP API type). For a given site, create a route like /.netlify/functions/{functionName}. Configure this route with a Lambda proxy integration to trigger the corresponding Lambda function (<site-id>-<function-name>)."

Phase 4: Creating the User-Facing Control Plane

This final phase builds the management interfaces for the platform.

Prompt Set 4.1 (Core API)

- **Prompt 4.1.1:** "Create a new Replit project for the 'core-api'. Use Node.js, Express, and Prisma with a PostgreSQL database. Define Prisma schemas for User, Site, and Deploy. The Site model should have fields for name, repositoryUrl, buildCommand, publishDirectory, and a relation to a User. The Deploy model should have fields for commitSha, branch, status, and a relation to a Site."
- **Prompt 4.1.2:** "Create authenticated REST API endpoints for CRUD operations on Sites. For example, POST /sites to create a new site, and GET /sites to list sites belonging to the authenticated user."
- **Prompt 4.1.3:** "Create an endpoint GET /sites/{siteId}/deploys that lists all deploys for a given site, sorted by creation date."

Prompt Set 4.2 (Dashboard UI)

- **Prompt 4.2.1:** "Create a new Replit project using the React (Vite) template. Create a login page and a dashboard page protected by authentication. Use a library like axios to communicate with the 'core-api'."
- **Prompt 4.2.2:** "On the dashboard page, fetch and display a list of the user's sites from the GET /sites endpoint. For each site, display its name and URL."
- **Prompt 4.2.3:** "Create a detail page for a single site. When a user clicks on a site from the dashboard, this page should fetch and display the deployment history from the GET /sites/{siteId}/deploys endpoint. Display the commit message, branch, and status for each deploy."

Prompt Set 4.3 (CLI)

- **Prompt 4.3.1:** "Create a new Node.js Replit project for the 'platform-cli'. Use a library like commander.js or yargs to define commands. Create a login command that prompts the user for an API token and saves it to a local configuration file."
- **Prompt 4.3.2:** "Create a sites:list command that reads the saved API token, makes an authenticated request to the GET /sites endpoint of the 'core-api', and prints the list of sites in a formatted table to the console."

Conclusion: From Clone to Competitor

This report has laid out a comprehensive architectural blueprint and implementation strategy for replicating the core functionality of the Netlify platform. The architecture is grounded in modern cloud-native principles, leveraging containers for secure builds, object storage and CDNs for high-performance delivery, and a serverless model for dynamic compute. The provided prompts offer a phased, actionable roadmap for building this complex system using an AI-assisted development environment like Replit.

It is crucial to recognize the significant engineering effort this undertaking represents. While the individual components—a webhook listener, a build runner, an API—are well-understood problems, the true complexity and intellectual property lie in their seamless and robust orchestration. Building a production-grade PaaS requires meticulous attention to security, scalability, resilience, and user experience at every layer of the stack.

Beyond the technical implementation, running such a platform at scale introduces significant operational challenges. Cost management is paramount; the platform's profitability will depend on efficiently managing resource consumption across build minutes, function execution time, data storage, and CDN bandwidth. Continuous security monitoring of the build environment is non-negotiable to protect the platform and its users from malicious actors. Finally, providing reliable customer support and comprehensive documentation is essential for user adoption and retention.

While the goal of this report is to create a clone, the true opportunity lies in innovation. The analysis of the PaaS landscape reveals a spectrum of solutions balancing control and convenience. By understanding the architectural trade-offs, a

new platform can differentiate itself. It could target a specific niche by offering deeper database integration, providing more flexible backend hosting options beyond simple serverless functions, or building a unique set of add-on services. This blueprint should be seen not as an end-point, but as a solid foundation upon which to build a differentiated and competitive platform in the dynamic world of web development.

Geciteerd werk

1. A Beginner's Guide to Composable Architecture | Netlify, geopend op juli 19, 2025, https://www.netlify.com/blog/beginners-guide-to-composable-architecture/
2. Vercel vs Netlify vs Heroku - Which is the best option? - Back4App Blog, geopend op juli 19, 2025, https://blog.back4app.com/vercel-vs-netlify-vs-heroku/
3. Vercel vs Netlify: Compare Jamstack Hosting and Deployment - Prismic, geopend op juli 19, 2025, https://prismic.io/blog/vercel-vs-netlify
4. Hosting Web Applications on Netlify Edge - Serverless Web Apps, geopend op juli 19, 2025, https://www.netlify.com/for/web-applications/
5. Exploring Netlify: Key Features and Advantages for Modern Web Development, geopend op juli 19, 2025, https://www.newweborder.co/blogs/netlify-features-overview
6. What is Netlify? The Ultimate Guide for Web Developers - Netguru, geopend op juli 19, 2025, https://www.netguru.com/blog/what-is-netlify
7. What is Netlify and How Does it Work? Everything in Detail ..., geopend op juli 19, 2025, https://yourdigilab.com/blog/what-is-netlify
8. Netlify: Scale & Ship Faster with a Composable Web Architecture, geopend op juli 19, 2025, https://www.netlify.com/
9. The Workflow - Netlify, geopend op juli 19, 2025, https://www.netlify.com/platform/core/workflow/
10. Netlify: A comprehensive overview - Ikius, geopend op juli 19, 2025, https://ikius.com/blog/what-is-netlify
11. WebHook Setup Examples, geopend op juli 19, 2025, https://docs.copado.com/articles/copado-ci-cd-publication/webhook-setup-example
12. GitHub Webhook CI/CD: Step-by-step guide - DEV Community, geopend op juli 19, 2025, https://dev.to/techlabma/github-webhook-cicd-step-by-step-guide-1j6g
13. Vercel vs Netlify: Choosing the right one in 2025 (and what comes next) | Blog - Northflank, geopend op juli 19, 2025, https://northflank.com/blog/vercel-vs-netlify-choosing-the-deployment-platform-in-2025
14. CI - Docker Docs, geopend op juli 19, 2025, https://docs.docker.com/build/ci/
15. How to Create a CI/CD Pipeline with Docker [Tutorial] - Spacelift, geopend op juli 19, 2025, https://spacelift.io/blog/docker-ci-cd
16. How to Host a Static Website on AWS S3 and CloudFront - freeCodeCamp, geopend op juli 19, 2025,

https://www.freecodecamp.org/news/host-a-static-website-on-aws-s3-and-cloudfront/
17. Host a static website | Cloud Storage, geopend op juli 19, 2025, https://cloud.google.com/storage/docs/hosting-static-website
18. Cloudflare vs. AWS CloudFront | CDN & DDoS, geopend op juli 19, 2025, https://www.cloudflare.com/cloudflare-vs-cloudfront/
19. Netlify Functions, geopend op juli 19, 2025, https://www.netlify.com/platform/core/functions/
20. Get started with API Gateway - Serverless - AWS Documentation, geopend op juli 19, 2025, https://docs.aws.amazon.com/serverless/latest/devguide/starter-apigw.html
21. Get started with the Netlify API | Netlify Docs, geopend op juli 19, 2025, https://docs.netlify.com/api/get-started/
22. Introducing Site Dashboards - Netlify, geopend op juli 19, 2025, https://www.netlify.com/blog/2017/08/22/introducing-site-dashboards/
23. 8 Netlify alternatives for 2025 - Hostinger, geopend op juli 19, 2025, https://www.hostinger.com/tutorials/netlify-alternatives
24. Deep Dive into Bettertori.fi, geopend op juli 19, 2025, https://blog.bettertori.fi/blog/02-in-depth/
25. CapRover · Scalable, Free and Self-hosted PaaS!, geopend op juli 19, 2025, https://caprover.com/
26. 7 Netlify alternatives in 2025: Where to go when your app grows up | Blog - Northflank, geopend op juli 19, 2025, https://northflank.com/blog/netlify-alternatives
27. Webhooks - GitLab Docs, geopend op juli 19, 2025, https://docs.gitlab.com/user/project/integrations/webhooks/
28. About webhooks - GitHub Docs, geopend op juli 19, 2025, https://docs.github.com/en/webhooks/about-webhooks
29. Pricing and Plans - Netlify, geopend op juli 19, 2025, https://www.netlify.com/pricing/
30. CI/CD With Docker: The Basics And A Quick Tutorial | - Octopus Deploy, geopend op juli 19, 2025, https://octopus.com/devops/ci-cd/ci-cd-with-docker/
31. Understanding the Docker USER Instruction, geopend op juli 19, 2025, https://www.docker.com/blog/understanding-the-docker-user-instruction/
32. Secure Your Docker Containers: Dockerfile User Command - CyberPanel, geopend op juli 19, 2025, https://cyberpanel.net/blog/docker-file-user-command
33. 3 Essential Steps to Securing Your Docker Container Deployments - JFrog, geopend op juli 19, 2025, https://jfrog.com/devops-tools/article/3-steps-to-securing-your-docker-container-deployments/
34. Get started with a secure static website - Amazon CloudFront, geopend op juli 19, 2025, https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/getting-started-secure-static-website-cloudformation-template.html
35. HOSTING A STATIC WEBSITE USING GOOGLE CLOUD STORAGE | Pawa IT

Solutions, geopend op juli 19, 2025, https://pawait.africa/hosting-a-static-website-using-google-cloud-storage/
36. Atomic Sync For Website Deployments to S3 - Aleksa Cukovic, geopend op juli 19, 2025, https://aleksac.me/til/atomic-sync-for-website-deployments-to-s3/
37. Use an Amazon CloudFront distribution to serve a static website, geopend op juli 19, 2025, https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/getting-started-cloudfront-overview.html
38. Static Site Hosting Using Google Cloud Storage and Cloudflare (with SSL!), geopend op juli 19, 2025, https://devopsdirective.com/posts/2020/10/gcs-cloudflare-hosting/
39. CloudFlare and Google Cloud for Hosting a Static Site | by Pablo A. Del Valle H. | Medium, geopend op juli 19, 2025, https://medium.com/@pablo.delvalle.cr/cloudflare-and-google-cloud-for-hosting-a-static-site-fd2e1a97aa9b
40. CloudFront vs CloudFlare: Choosing the right CDN - DEV Community, geopend op juli 19, 2025, https://dev.to/clickit_devops/cloudfront-vs-cloudflare-choosing-the-right-cdn-39jk
41. Amazon CloudFront vs. Cloudflare - Mission Cloud Services, geopend op juli 19, 2025, https://www.missioncloud.com/blog/amazon-cloudfront-vs-cloudflare
42. Hosting Web Apps: Cloudflare vs. AWS | by Kamol - Medium, geopend op juli 19, 2025, https://medium.com/by-devops-for-devops/hosting-web-apps-cloudflare-vs-aws-04c570b26dcc
43. Cloudflare Vs AWS Cloudfront: A Detailed Comparison - ValueCoders, geopend op juli 19, 2025, https://www.valuecoders.com/blog/cloud-services/cloudflare-vs-aws-cloudfront/
44. What is Netlify: A Review of Hosting and Deployment Features - Bejamas, geopend op juli 19, 2025, https://bejamas.com/hub/hosting/netlify
45. What is Netlify? Features, Pricing, Pros & Cons Explained - Talentelgia Technologies, geopend op juli 19, 2025, https://www.talentelgia.com/blog/what-is-netlify/
46. Super simple start to Netlify functions - Kent C. Dodds, geopend op juli 19, 2025, https://kentcdodds.com/blog/super-simple-start-to-netlify-functions
47. A Minimal GitHub Actions Workflow to Deploy Lambda Code Automatically, geopend op juli 19, 2025, https://aws.plainenglish.io/a-minimal-github-actions-workflow-to-deploy-lambda-code-automatically-9656cfad012e
48. Serverless with AWS Lambda and API Gateway | Guides - Terraform Registry, geopend op juli 19, 2025, https://registry.terraform.io/providers/hashicorp/aws/2.33.0/docs/guides/serverless-with-aws-lambda-and-api-gateway
49. How to set up an AWS Lambda and auto deployments with Github Actions - posts, geopend op juli 19, 2025,

https://blog.jakoblind.no/aws-lambda-github-actions/
50. Serverless Framework : Manage AWS Lambda functions from a Git repository - Medium, geopend op juli 19, 2025, https://medium.com/@verove.clement/serverless-framework-manage-aws-lambda-functions-from-a-git-repository-4373fcf1e51e
51. API Gateways for Serverless: Top 5 Solutions & Tips for Success - Lumigo, geopend op juli 19, 2025, https://lumigo.io/serverless-monitoring-guide/api-gateways-for-serverless-top-5-solutions-and-tips-for-success/
52. Get started with functions | Netlify Docs, geopend op juli 19, 2025, https://docs.netlify.com/functions/get-started/
53. Functions API reference | Netlify Docs, geopend op juli 19, 2025, https://docs.netlify.com/functions/api/
54. What is Vercel: A Definitive Guide - Fishtank, geopend op juli 19, 2025, https://www.getfishtank.com/insights/what-is-vercel
55. Can you use Vercel for backend? What works and when to use something else - Northflank, geopend op juli 19, 2025, https://northflank.com/blog/vercel-backend-limitations
56. How to create a value-based SaaS dashboard design your users will love | ProductLed, geopend op juli 19, 2025, https://productled.com/blog/how-to-create-a-value-based-saas-dashboard-design
57. Dashboard Design: best practices and examples - Justinmind, geopend op juli 19, 2025, https://www.justinmind.com/ui-design/dashboard-design-best-practices-ux
58. See 13 Netlify features for the best control of development workflow, geopend op juli 19, 2025, https://www.netlify.com/blog/2020/05/12/see-13-netlify-features-for-the-best-control-of-development-workflow/
59. CapRover | Server Dashboard, geopend op juli 19, 2025, https://captain.shared-api-staging.sqeds.com/
60. Dashboard | Coolify Docs, geopend op juli 19, 2025, https://coolify.io/docs/services/dashboard
61. Coolify, geopend op juli 19, 2025, https://coolify.io/
62. Netlify Documentation | Netlify Docs, geopend op juli 19, 2025, https://docs.netlify.com/
63. APIs for code agents | Netlify Docs, geopend op juli 19, 2025, https://docs.netlify.com/extend/building-code-agents/apis-for-code-agents/
64. Set up an existing custom domain name for your app - Azure App Service | Microsoft Learn, geopend op juli 19, 2025, https://learn.microsoft.com/en-us/azure/app-service/app-service-web-tutorial-custom-domain
65. Getting Started - Let's Encrypt, geopend op juli 19, 2025, https://letsencrypt.org/getting-started/
66. Let's Encrypt and DNSimple, geopend op juli 19, 2025,

https://support.dnsimple.com/articles/letsencrypt/