ChatGPT

Nerdwork+ AWS Infrastructure Setup Handbook

Welcome to the Nerdwork+ cloud infrastructure setup guide! This handbook will walk you through manually provisioning all the AWS components for the Nerdwork+ platform. We'll cover everything from networking and security to deploying the backend microservices and frontend application. **No prior DevOps experience is required** – we'll explain each step in beginner-friendly terms and provide context for why each component is needed. By the end, you will have a working cloud architecture aligned with Nerdwork's design and best practices [1] [2].

Overview of the Nerdwork+ Architecture

Nerdwork+ is a comic-viewing web platform with a microservices backend and a Next.js frontend. The architecture follows a cloud-native, **modular microservices design** for scalability and maintainability [3] [4]. Here's a high-level summary of the main components we will set up, and how they fit together (see the finalized Figma cloud diagram for a visual reference):

- **Virtual Private Cloud (VPC):** An isolated network environment in AWS where our services will run. This provides network segmentation, with *public subnets* for internet-facing resources and *private subnets* for internal services and databases. All compute resources (ECS tasks, Lambda functions, etc.) run inside this VPC for security [5].

- **Backend Microservices:** The platform is broken into multiple microservices (Auth, User, Comic, Wallet, Payment, Loyalty, Notification, Analytics, etc.), each encapsulating a specific domain [6] [7]. Services communicate through a unified API and events. Many services will be deployed as **AWS Lambda** functions or containerized on **AWS ECS** (Fargate). We use an **API Gateway** as a single entry point for all client API calls, routing requests to the correct service [8]. This simplifies client interactions and centralizes security (we can attach WAF and do auth checks at the gateway).

- **Data Storage:** We use both relational and NoSQL databases to meet different needs [9] [10]. An **Amazon RDS** PostgreSQL (or Aurora Serverless) database will store relational data (e.g. user profiles, comic metadata, and the NWT token ledger). **Amazon DynamoDB** tables will handle high-throughput or flexible-schema data (e.g. analytics events, caching lookups) in a cost-efficient way [11]. We also have **Amazon S3** buckets for file storage (comic pages, images, static web assets), fronted by **Amazon CloudFront** for global CDN caching [12] [13].

- **Security and Identity:** Security is paramount. We will configure **IAM roles** for each service to grant least-privilege access (no hardcoded AWS keys) [14]. We will set up **AWS WAF** (Web Application Firewall) to protect against common web exploits like SQL injection and XSS [15]. Sensitive secrets (e.g. database passwords, API keys) will be kept in **AWS Secrets Manager**. Network security is enforced with private subnets (no direct internet access for servers – they reach out via a NAT Gateway or VPC Endpoints) [16] and **security groups** (firewall rules).

- **Frontend Deployment:** The Next.js frontend will be built and deployed as a static website to an S3 bucket, served via CloudFront CDN for speed and TLS security. This “serverless” hosting means the frontend can scale globally with minimal cost [13]. The frontend calls the backend API (through API Gateway) over HTTPS.

- **Additional Services:** We will integrate **Amazon EventBridge** for decoupled, event-driven communication (for example, publishing an event when a user reads a comic, so the Loyalty service can award points). **Amazon SES** (Simple Email Service) will handle outgoing emails (e.g. verification emails, notifications) at low cost (first 62k emails per month are free) [17]. We’ll also mention options for analytics pipelines or workflow schedulers (e.g. using Apache Airflow via Cloud Composer or AWS Managed Airflow) if needed for advanced analytics down the road.

- **DevOps and CI/CD:** Finally, we’ll set up a continuous integration/continuous deployment (CI/CD) pipeline using **GitHub Actions**. This will automate building and releasing the application – for example, building Docker images for microservices, pushing to ECR, and deploying to ECS or updating Lambda code [18], as well as uploading frontend assets to S3. We’ll also cover using Infrastructure as Code and strategies to keep costs low and quality high (like automated security scans in the pipeline [18]).

Throughout this guide, we’ll highlight **cost-saving tips** (e.g. using on-demand serverless services, consolidating resources, and minimizing NAT Gateway usage) and **security best practices** appropriate for a startup-scale deployment (0–10k users) [19] [15]. Let’s get started!

1. Setting Up the Network (VPC, Subnets, and Networking)

First, we need to create the foundation: the **Virtual Private Cloud (VPC)** that will house our infrastructure. The VPC is like our own private network in AWS. We will create subnets (sections of the VPC) to group resources, an internet gateway for external connectivity, and a NAT gateway and VPC Endpoints for controlled outbound access.

1.1 Create a VPC:

- **Using the AWS Console:** Log in to the AWS Management Console and navigate to **VPC** service. Click **Create VPC**. For a simple setup, choose the **“VPC and more”** creation wizard which can create subnets and gateways in one go [20]. Give the VPC a name (e.g., `nerdwork-vpc`). Use a CIDR block like `10.0.0.0/16` (which provides ~65k private IPs) – this should be fine for our scale. Ensure **Enable DNS Hostnames** is checked so that instances/Lambdas in the VPC can resolve internal DNS names (this is usually enabled by default for new VPCs). You can omit IPv6 for now unless needed.

- **Using AWS CLI (optional):** If you prefer CLI, you can run:

```
aws ec2 create-vpc --cidr-block 10.0.0.0/16 --tag-specifications
'ResourceType=vpc,Tags=[{Key=Name,Value=nerdwork-vpc}]'
```

  Then enable DNS hostnames:

```
aws ec2 modify-vpc-attribute --vpc-id <vpc-id> --enable-dns-hostnames
"{\"Value\":true}"
```

(Replace `<vpc-id>` with the VPC ID returned by the create command.)

1.2 Create Subnets (Public and Private):

We'll use two types of subnets: - **Public subnets:** for resources that need direct internet access (or to host the NAT gateway or a load balancer). - **Private subnets:** for internal resources like application servers, ECS tasks, Lambdas connecting to a database, etc., that should not be directly reachable from the internet.

It's good practice to have subnets in multiple Availability Zones (AZs) for high availability. We'll create at least two public and two private subnets (one in each of two AZs). Each subnet will cover a smaller range within the VPC, e.g.: - `10.0.1.0/24` – Public subnet in AZ A (we'll use this for NAT Gateway, possibly an ALB) - `10.0.2.0/24` – Public subnet in AZ B - `10.0.3.0/24` – Private subnet in AZ A (for backend resources) - `10.0.4.0/24` – Private subnet in AZ B

Console Steps: In the VPC console, under **Subnets**, click **Create subnet**. Select the new VPC. Create four subnets with the details above (choose two distinct AZs from the dropdown for distribution). Tag them with names like `public-a`, `public-b`, `private-a`, `private-b` for clarity. Ensure the public subnets have **Auto-assign public IPv4** set to "Enable" (so resources launched there can get public IPs if needed, e.g. a NAT or bastion host).

1.3 Internet Gateway:

An **Internet Gateway (IGW)** allows traffic from the VPC to go to the internet. In the VPC console, go to **Internet Gateways** and **Create internet gateway**. Name it (e.g. `nerdwork-igw`). After creation, **attach** it to your VPC (you can do this from the IGW's Actions menu).

1.4 Route Tables and Routes:

Now we'll set up routing: - The **public subnets' route table** should send internet-bound traffic to the Internet Gateway. - The **private subnets' route table** should send internet-bound traffic to the NAT Gateway (we'll create NAT next). This way, instances/Lambdas in private subnets can reach the internet *outbound* (for example, to call external APIs or AWS services) but still have no inbound exposure.

When using the "VPC and more" wizard, AWS may have created default route tables for you. Check **Route Tables** in the console: - There should be a main route table for the VPC. Typically, by default, it will route `10.0.0.0/16` (the VPC range) locally and nothing else. - Create a **public route table**: In Route Tables, click **Create route table**, name it `nerdwork-public-rt`, associate it with the VPC. Under Routes, add a rule: Destination `0.0.0.0/0` (meaning "any IPv4 address") -> Target = your Internet Gateway (you should see the IGW ID in dropdown). Save. Then under **Subnet Associations**, associate this route table with **both public subnets** (select the public subnets created). - Create a **private route table** similarly (name `nerdwork-private-rt`). Add a route: Destination `0.0.0.0/0` -> Target = (will fill in once NAT is created). For now, just create the table and associate it with the **private subnets**. We will come back to add the NAT target.

1.5 NAT Gateway (Network Address Translation):

A NAT Gateway is needed to allow instances in private subnets to **initiate** outbound connections to the internet (for example, a Lambda in a private subnet calling an external API or downloading packages) while still being protected from inbound traffic. We will create **one** NAT Gateway to minimize cost, rather than one per AZ (NATs are ~$30+ per month each, plus data charges). Using a single NAT in one AZ means if that AZ goes down, resources in the other AZ lose internet access, but at early stage this cost saving might be acceptable. We'll also mitigate this by adding **VPC Endpoints** next to reduce reliance on NAT for AWS services.

Console Steps: Go to **NAT Gateways** in the VPC console -> **Create NAT Gateway**. Choose one of your public subnets (e.g. `public-a`) to place the NAT. Allocate a new Elastic IP for it (an Elastic IP is a static public IP address). Name the NAT Gateway (e.g. `nerdwork-nat`). Create. Wait a minute for status to become "available".

Once the NAT is up, edit the **private route table** `nerdwork-private-rt` and add the route: Destination `0.0.0.0/0` -> Target = your NAT Gateway ID. Now, any instance/Lambda in private subnets will send outbound internet traffic to the NAT, which will forward it out through the IGW using that Elastic IP.

Cost Tip: By default, AWS might encourage one NAT per AZ for high availability [21] [22]. However, at small scale, one NAT saves cost (one NAT ~$32/month vs two NATs ~$64, excluding data). We accept a bit of risk for cost efficiency. To further reduce NAT data usage, we'll set up VPC Endpoints next.

1.6 VPC Endpoints (Private Connectivity to AWS Services):

VPC Endpoints allow your VPC resources to connect privately to AWS services (like S3, DynamoDB, SNS, etc.) without going through the NAT/internet. This not only improves security (traffic stays on AWS network) but also **saves NAT data costs**. Gateway endpoints for S3 and DynamoDB are free of hourly charges [23] (and no data processing fees), so we definitely want those. Interface endpoints (for services like Secrets Manager, ECR, etc.) have a small hourly cost, so we'll add them only if needed frequently.

Let's add common endpoints: - **S3 Endpoint (Gateway):** In the VPC console, go to **Endpoints** -> **Create Endpoint**. Search for the service "com.amazonaws.<region>.s3" which should say "Type: Gateway". Select it. Choose your VPC. Under Route Table, select the **private route table** (`nerdwork-private-rt`) so that this endpoint will automatically add a route for S3. Create endpoint. This will add a route in the private RT for the S3 service prefix (plenty of AWS services, like S3 and DynamoDB, provide a *prefix list* for their IP ranges). After creation, verify the private route table now has a route to `pl-xxxx -> s3 endpoint`.

- **DynamoDB Endpoint (Gateway):** Still under Endpoints, create another. Service name "com.amazonaws.<region>.dynamodb" (Type: Gateway). Again attach to the VPC and the private route table. Create. Now the private subnets have direct access to DynamoDB without NAT.

Using these, when our Lambdas or ECS tasks interact with S3 or DynamoDB, that traffic will **not** go out via NAT (saving costs). *For example, VPC endpoints can be 80% cheaper than routing through a NAT Gateway for these services* [24] *.*

- **Other Endpoints (Optional):** Depending on our use-cases, we might add:
- **Secrets Manager endpoint** (com.amazonaws.<region>.secretsmanager, Type: Interface) – so that retrieving secrets doesn't use NAT. This costs a small hourly fee (~$7/month per AZ) plus data. For light usage it might be fine to use NAT instead, but for heavy secret access it's worth it.
- **ECR endpoints** for Docker image pulls (if using ECS extensively). AWS ECR has two endpoints: one for the API (to get auth tokens) and one for image registry (docker pull). If our ECS tasks run in private subnets, add these to avoid NAT during deployments.
- **SNS, SQS endpoints** if using those services and want to keep traffic internal.

Each interface endpoint is generally ~$7.20/month + data per AZ, so you can decide based on expected usage. Often, start with S3/Dynamo (free gateway endpoints) and add others if NAT costs become significant or security mandates it. Remember, **gateway endpoints for S3/DynamoDB incur no additional cost** [23] .

1.7 Summary & Verification:

At this stage, our network is set. We have a VPC with subnets, an IGW, a single NAT Gateway, and VPC endpoints. Let's recap: - **Public Subnets:** have route to IGW (0.0.0.0/0 -> IGW). These subnets are for things like load balancers or NAT. - **Private Subnets:** have route to NAT (0.0.0.0/0 -> NAT) and also routes to S3/Dynamo via endpoints. Resources here can reach out but aren't reachable from outside. - **Connectivity check:** If you launch an EC2 instance in a private subnet (with no public IP) and SSH into it via a bastion or Session Manager, you should be able to ping an external site or do `curl google.com` and get a response (traffic will go through NAT). Also try `aws s3 ls` via CLI on that instance to ensure S3 access (should go through endpoint if configured).

We now have a secure, cost-efficient network foundation to deploy the rest of Nerdwork+ services.

2. Identity and Access Management (IAM) Configuration

Next, we'll set up AWS Identity and Access Management (IAM) roles and policies. Proper IAM configuration ensures our services can access what they need and nothing more – following the principle of **least privilege**. It also eliminates the need to embed AWS credentials in code by leveraging roles.

2.1 IAM Roles for AWS Services:

We will create dedicated IAM roles for various components: - **EC2 Instance Role (if needed):** If we use any EC2 instances (for a bastion host, or if running something not on Lambda/ECS), we would create an instance profile role for them. This is optional in our mostly serverless design. - **Lambda Execution Role:** AWS Lambda functions require an IAM role that defines what AWS resources the function is allowed to interact with. We'll create a role `nerdwork-lambda-exec` that can, for example, read from S3, write to DynamoDB, send to EventBridge, etc., depending on function needs. We will attach policies accordingly. - **ECS Task Role and Task Execution Role:** For ECS Fargate tasks, we need two roles: the **task execution role** (grants ECS agent permissions to pull container images from ECR, write logs to CloudWatch, etc.) – we can use AWS's managed policy `AmazonECSTaskExecutionRolePolicy` for this – and the **task role** (this is the role the application inside the container assumes, to access other AWS services like S3, SQS, etc.). We'll create a `nerdwork-ecs-task-role` for the latter and attach custom policies as needed (e.g. allow reading a specific S3 bucket or access to Secrets Manager for DB credentials). Each microservice running on ECS can reuse the same task role if permissions are similar, or you can create distinct roles per service for stricter isolation.

- **CI/CD Deployment User or Role:** For GitHub Actions to deploy infrastructure, we have two approaches:
- Create an **IAM User** (e.g. `github-ci-user`) with Access Key and Secret. Attach a policy that allows it to deploy resources (e.g. push to ECR, update Lambda, etc.). We will limit permissions to only necessary services (ECR, ECS, Lambda, S3, CloudFront, etc.). Store these credentials as GitHub Action secrets.
- Use **OIDC federation** from GitHub to AWS: This is a modern approach where GitHub Actions can assume a role in AWS without static credentials, using OpenID Connect trust. This requires setting up an IAM Role with a trust policy for GitHub's OIDC provider and configuring the workflow with that role ARN. This avoids storing long-term keys. It's a bit advanced, so initially you might use the simpler IAM user method, but consider migrating to OIDC for better security.

Console Steps to Create Roles: - Navigate to **IAM** in AWS console. Go to **Roles** -> **Create Role**. - For a Lambda execution role: choose AWS service **Lambda** as the trusted entity. Attach policies like: - `AWSLambdaBasicExecutionRole` (managed policy for CloudWatch Logs access, so the Lambda can write logs). - Custom policy for specific needs: e.g., if a Lambda needs to access S3 bucket `nerdwork-content`, create an IAM policy granting `s3:GetObject`/`PutObject` on `arn:aws:s3:::nerdwork-content/*`. Or allow `dynamodb:PutItem` on a specific table if it writes to DynamoDB, etc. - If the Lambda needs to access Secrets Manager (e.g. to retrieve DB credentials), attach a policy granting `secretsmanager:GetSecretValue` on the specific secret ARN. - Name the role e.g. `nerdwork-lambda-basic-role`. Complete creation. Later, when creating each Lambda function, we'll assign this role (or a more specialized one) to it.

- For ECS: create a role for **ECS task execution**:
- Trusted entity: **ECS** (for Fargate tasks).
- Attach the managed policy **AmazonECSTaskExecutionRolePolicy** (this covers ECR pull, CloudWatch logs, XRay, etc.).
- Name it `ecsTaskExecutionRole` (AWS recommends this exact name for Fargate).
- Then create another role for **ECS task runtime** (the app's role):
    - Trusted entity: **ECS**.
    - Attach policies needed by your microservices. For example, if a container needs to read a DynamoDB table or publish to EventBridge, include those permissions.
    - Name it `nerdwork-ecs-task-role` (or separate per service, e.g., `nerdwork-wallet-svc-role` if you want fine-grained roles per microservice).

2.2 IAM Policies and Best Practices:

When creating custom policies, follow these tips: - **Least Privilege:** Only allow the specific actions and resources needed. For instance, instead of granting access to all S3 buckets, specify the exact bucket ARN [25]. Instead of all DynamoDB tables, specify the table ARN. - **Use IAM Roles for AWS services instead of**

long-term keys: As noted, our Lambda and ECS won't use static credentials at all – AWS automatically provides temporary credentials to the role at runtime. This is a security best practice [14]. - **Secure secrets access:** If a service needs database credentials, prefer storing those in Secrets Manager and give the service permission to fetch that secret, rather than storing the password in code or environment variables in plaintext. - **Multi-factor & admin users:** Ensure the root AWS account has MFA enabled, and create an Admin IAM user for daily use (don't use root for operations). For team collaboration, create individual IAM users or use AWS SSO with permissions if possible, rather than sharing accounts.

We'll revisit IAM when configuring specific services (for example, setting up an IAM authorizer for API Gateway or fine-tuning bucket policies). But at this point, the key roles should be in place.

3. Setting Up Data Storage (RDS, DynamoDB, S3, Secrets Manager)

With networking and IAM ready, we proceed to setting up databases and storage solutions.

3.1 Relational Database (Amazon RDS – Postgres/Aurora)

Nerdwork+ uses a relational database for core data that needs transactions and relationships – e.g. user accounts, comics metadata, and the NWT ledger (wallet balances and transactions). We will use **Amazon RDS** for this. Specifically, we can choose **PostgreSQL** or **MySQL**. The architecture suggests using Amazon Aurora (which is Amazon's clustered database engine) for better scalability and possibly Aurora Serverless v2 for cost savings [26]. We'll outline both options:

- **Option A: Aurora Serverless v2 (PostgreSQL)** – *Recommended for cost efficiency.* This allows the database to automatically scale down to a very small capacity (0.5 ACUs) when idle, saving money [27]. It can also scale up on demand as load increases, and it's managed by AWS. Aurora has the advantage of auto-pausing when completely idle (though v2 doesn't pause fully, it just scales to minimum).
- **Option B: Standard RDS PostgreSQL (or MySQL)** – you could start with a **db.t4g.micro** instance (which is low-cost, often under the free tier or around ~$10-15/month on-demand). This is a single instance (we can start with single AZ to save cost, and later move to multi-AZ for high availability when needed).

We will assume **PostgreSQL** dialect for examples (feel free to use MySQL if the team prefers).

Console Steps to create Aurora Serverless v2: 1. Go to **RDS** service in AWS console, click **Create database**. 2. Choose **Aurora PostgreSQL-Compatible**. For edition, pick a recent version. 3. For **Capacity type**, select **Serverless**. For Aurora Serverless v2, you'll specify a capacity range (e.g., `0.5 ACUs (min)` to maybe `4 ACUs (max)` – you can adjust max as you scale). 4. In **Settings**, give your cluster an identifier (e.g., `nerdwork-db-cluster`). Set master username (e.g., `admin`) and password – **IMPORTANT:** Use a strong password and consider storing it in AWS Secrets Manager right away (there's an option "Store credentials in Secrets Manager" – check that to have AWS create a secret for you). 5. For **Network & Security**, select our VPC (`nerdwork-vpc`). Choose **Private subnets** for deployment (the DB doesn't need to be in public subnets). It will ask for a subnet group – if none exists, create a new subnet group selecting our two private subnets. This ensures the DB cluster spans those AZs. 6. Ensure **Public access** is **No** (we do *not* want the DB to have a public IP). We will access it only from inside the VPC. 7. Choose or create a **security group** for the DB. You can make one (e.g., `sg-nerdwork-db`) that allows inbound traffic on port 5432 (PostgreSQL) **only**

from the VPC's internal range or specific app servers. For simplicity, you might allow inbound from the security group that your ECS tasks and Lambdas use. For example, if we create an SG called `sg-backend` for all application services, you can set the DB SG inbound rule: source = `sg-backend`, port 5432. This way, only your app services can talk to the database. This is more secure than opening by IP range. 8. For **Initial database name**, you can specify a name (e.g., `nerdworkdb`) so AWS creates a default database. 9. Leave other defaults (Aurora serverless v2 doesn't require instance size; it uses ACUs). For **Availability Zone**, serverless v2 is multi-AZ by design (no standby needed). 10. Click **Create database**. It will take a few minutes to provision.

If using standard RDS Postgres: - Choose "PostgreSQL" engine, "Dev/Test" template for free tier eligibility. - Pick db.t4g.micro (or db.t3.micro if not eligible for t4g). - Allocate storage (e.g., 20 GB). - Multi-AZ: choose No for now (to save cost; we can upgrade later). - Other steps are similar: set private subnets, no public access, security group, name, credentials.

Accessing the DB: After creation, we'll have an endpoint address (something like `nerdworkdb.cluster-XYZ.region.rds.amazonaws.com`). Only resources in the VPC (in allowed SGs) can reach it. In development, you might use an EC2 bastion or AWS Systems Manager Session to connect for debugging. In production, only the app services should connect.

Cost Consideration: Aurora Serverless v2 will cost based on ACU hours used. At minimum 0.5 ACU, that's roughly $0.06-$0.12 per hour (region dependent) when idle – a few dollars a month if rarely used, which is economical [27]. A db.t4g.micro under free tier might be effectively $0 if you have credits, but outside free tier runs ~$0.018/h (~$13/mo) and is fixed size. Serverless v2 can auto-scale beyond micro when needed, providing performance headroom without manual upgrades. Either is valid; Aurora gives more elasticity.

We will use this DB to implement the **NWT ledger** and other relational data. **Important:** Save the master credentials securely (if not stored in Secrets Manager by the wizard). We will likely rotate away from using the master user for the app and create a specific DB user for the application with least privileges on the needed schemas.

3.2 NoSQL Database (Amazon DynamoDB)

For certain microservices, we use **DynamoDB**, a fast key-value NoSQL store, ideal for high write volumes or flexible schema data [28]. Nerdwork+ might use DynamoDB for: - Logging analytics events (lots of writes, simple queries) [29] [30]. - Caching or quick lookups, e.g., mapping a crypto wallet address to a user ID in the Wallet service for quick verification [31]. - Storing sessions or temporary data (since DynamoDB can scale and has TTL expiration features).

DynamoDB is fully managed and serverless, so no need to worry about VPC placement. However, to use it from our VPC without NAT, we added the VPC Endpoint for Dynamo earlier.

Creating a DynamoDB Table: Go to **DynamoDB** in the AWS console -> **Tables** -> **Create table**. You'll specify: - Table name (e.g., `AnalyticsEvents` or `UserActions` for logging events). - Primary key: choose a partition key (and sort key if needed). For an events table, maybe `userId` as partition and `timestamp` as sort key, or a composite like `eventId` as key. For a wallet mapping table, maybe primary key `walletAddress` maps to a `userId` attribute. - Leave default settings but **select Capacity mode =**

On-Demand. This is crucial for cost: on-demand means you don't have to pre-provision read/write capacity; you pay per request, which at low traffic is cheaper and you won't accidentally over-provision. Dynamo on-demand can handle spikes seamlessly and is very cost-effective for up to millions of requests in the free tier [11]. - Enable encryption (AWS encrypts Dynamo tables by default with a key, so that's fine).

Repeat for any other tables needed by other microservices: - Perhaps a `LoyaltyPoints` table if tracking points or progress. - Or a `NotificationsQueue` table (though you might use SQS instead for a queue). For now, you can create tables as needed when implementing each service. Keep in mind each microservice should ideally have its own table(s) (to maintain data isolation per service) [11].

3.3 Object Storage (Amazon S3) for Files and Frontend

We have two main uses for **Amazon S3** in Nerdwork+: 1. **Static website content** – the Next.js frontend (built as a bundle of HTML, CSS, JS) can be stored here and served via CloudFront. 2. **User and app file storage** – e.g. comic images, thumbnails, user avatar uploads, etc., managed by the File Service. These will also reside in S3, typically in a separate bucket.

We will create two buckets to keep concerns separate: - `nerdwork-frontend` (for frontend web assets) - `nerdwork-content` (for user content like comics, images, etc.)

Console Steps: Go to **S3** service -> **Create bucket**. Name the first bucket (S3 bucket names must be globally unique, so consider a unique suffix like `nerdwork-frontend-12345`). Choose your region. For now, leave "Block all public access" **ON** (checked) – we'll use CloudFront to deliver content so the bucket itself doesn't need to be public [25]. We will manage access via a CloudFront Origin Identity. Enable bucket versioning if you want to keep versions of files (useful for safekeeping or rollback of site updates, though it can incur extra storage cost). Create bucket.

Repeat to create the `nerdwork-content` bucket. Keep it private as well (we will access it from the app via signed URLs or CloudFront). You might enable **Object Lock** on this bucket if you need immutable storage for compliance (probably not needed here). But do enable **Lifecycle rules** for cost savings: - For `nerdwork-content`: Set a lifecycle rule to transition old objects to cheaper storage. For example, you might move any comic image that hasn't been accessed in 90 days to S3 Standard-Infrequent Access, or to Glacier after a year, to reduce storage cost. If the content is user-uploaded and not often accessed after some time, this can save money. In the bucket's **Management** tab, create a Lifecycle Rule like "Glacier-archive-old-comics" that transitions objects older than 180 days to Glacier Deep Archive (very cheap storage) [32]. This is optional and can be adjusted based on usage patterns. - The `nerdwork-frontend` bucket likely doesn't need a lifecycle rule except maybe to clean up old deployment files if you're not overwriting (but usually you will just overwrite files on each deploy, or use invalidations).

Bucket Security: Both buckets have blocking public access. For the frontend bucket, we will allow CloudFront to read from it. We'll configure that when setting up CloudFront. For the content bucket, we will allow the backend to upload/read (via IAM roles). For instance, the File Service Lambda or ECS task will have an IAM policy to put objects into `arn:aws:s3:::nerdwork-content/*`. You might also create a bucket policy that only allows GetObject if the request comes from CloudFront (using the CloudFront Origin Access Identity). We will do that soon.

3.4 Secrets Manager for Sensitive Configuration

We touched on this in IAM, but now let’s actually store our sensitive data in **AWS Secrets Manager**. This service lets us securely keep things like database credentials, API keys, third-party secrets, etc., and easily retrieve them from our code when needed. It also can auto-rotate certain secrets.

Secrets to store: - **Database credentials:** The RDS Aurora/Postgres master username & password (if you didn’t already have the RDS wizard store it). If it’s already stored, note the secret ARN. - **Database connection string:** You could store a full connection string or just components (host, db name, etc.). Sometimes it’s convenient to create a JSON secret that contains: `{"host": "...", "port": 5432, "dbname": "nerdworkdb", "username": "...", "password": "..."}`. The RDS console can create such a secret for you if you enable that option. - **JWT signing key / OAuth secrets:** If the Auth service uses JWTs and you have a signing private key or a symmetric secret for token signing, store it here. - **3rd-party API keys:** e.g., Helio API key (for crypto payments), Pinata API key/secret (for IPFS), Magicblock API credentials for NFT minting, etc., as referenced in the architecture [33] [34]. - **SMTP creds (if using any) or SES credentials:** For SES, if you use SMTP interface you’d have SMTP username/password (which can be generated in SES console). But if you use AWS SDK for SES, you can use IAM role permission and no secret needed.

Console Steps: Go to **Secrets Manager** -> **Store a new secret**. Select secret type **“Other type of secrets”** (for arbitrary secret). For a DB credential, you can pick **“Credentials for RDS database”** and it will ask the details. If not using that, just choose other and input key/value pairs. For example, to store DB creds manually, add keys `username`, `password`. Give the secret a name like `/nerdwork/prod/dbCredentials`. Choose encryption key (default AWS-managed is fine). Finish storing. You’ll get an ARN for the secret.

Repeat for other secrets, e.g. `/nerdwork/prod/helioApiKey` etc., storing the values. Organize by naming convention (maybe prefix with env name like dev/prod).

Accessing Secrets from code: Both Lambda and ECS can call the Secrets Manager API (GetSecretValue) if their IAM role permits. A best practice is to **not** store secrets in plaintext env variables. Instead, your code can fetch them on startup or use an integration. For example, AWS Lambda has an option to automatically pull a secret and set it as an environment variable (using `aws:secretsmanager:secret:ARN:json-key` in the env value). Alternatively, use AWS SDK in your code to fetch the secret when needed. We’ll ensure our IAM roles include permission to `secretsmanager:GetSecretValue` on the specific secret ARNs needed.

Now that our storage layers (RDS, Dynamo, S3) and secrets are configured, let’s move on to deploying the actual application components: backend services and the frontend.

4. Deploying Backend Microservices (Lambda, ECS, API Gateway)

Nerdwork+ backend comprises multiple microservices, which we will deploy on AWS using a mix of **Lambda functions** and **ECS containers**. We also set up an **API Gateway** to route API requests to the appropriate service. In this section, we detail how to deploy these services step-by-step.

4.1 Microservices Deployment Strategy

From the architecture, most microservices can be **stateless** and run in containers or serverless functions [35] [36]. For the initial 0–10k users scale, we favor serverless to minimize cost for low-traffic periods [37] [38]. We will assume: - **Auth, User, Loyalty, Notification, Analytics services**: Could run as lightweight Node.js or Python functions in Lambda (especially if their workloads are intermittent or can execute within Lambda limits). - **Comic, File, Payment, Wallet services**: These might involve heavier processing (image handling, external API calls, etc.). We can deploy them as containers in **ECS Fargate** for more control over runtime and easier long-running tasks. ECS is a good fit if you need more than 15 minutes execution (Lambda limit) or need custom binaries, etc. That said, you could also implement parts of these with Lambda if they can be split into smaller tasks.

Our API Gateway will abstract whether a backend integration is Lambda or ECS – it can integrate with both. The API design might have routes like `/auth/*`, `/user/*`, `/comics/*`, `/wallet/*`, etc., each pointing to the respective service.

4.2 Setting up API Gateway

Let’s set up the API Gateway first, as a skeleton for our API:

Choose API type: We can use **HTTP API (v2)** or **REST API (v1)**. HTTP API is simpler, faster, and cheaper, but slightly less feature-rich. It’s fine for our use case (it supports Lambda integration and JWT authorizers, etc.). We’ll proceed with HTTP API.

Console Steps for API Gateway: 1. Go to **API Gateway** in AWS console. Click **Create API**. Choose **HTTP API**. 2. For API name, enter “NerdworkAPI”. 3. We can start by creating a quick route. For example, define a route GET `/health` that responds with a 200 OK (easy for testing). 4. Choose an integration – for now, we might not have a backend Lambda ready, so you can select “Mock integration” or any placeholder. We will later attach real Lambdas and URLs. 5. Create the API.

Once created, note the **Invoke URL** (e.g., `https://abc123.execute-api.region.amazonaws.com`). We may later add a custom domain for prettiness (like `api.nerdwork.com`), but not required for functionality.

Now we will add routes for our microservices and integrate them: - For example: - **Auth service:** routes like POST `/auth/signup`, POST `/auth/login`, GET `/auth/profile` etc. We might integrate these with a Lambda function called `AuthServiceFunction`. - **User service:** routes like GET `/users/{userId}`, PUT `/users/{userId}` etc., integrated to `UserServiceFunction` (Lambda or ECS). - **Comic service:** routes like GET `/comics/{id}`, POST `/comics` (if creators upload), etc. - And so on for others (Wallet, Payment, Loyalty, etc.).

You can add one route at a time via the console, but a more efficient way is to use an OpenAPI definition or AWS CLI. Given a beginner context, let’s do a couple manually: - In API Gateway console, under your HTTP API, go to **Routes** -> **Create**. For **Method** select e.g. ANY and **Resource** `/auth/{proxy+}`. This route will catch all requests under `/auth/...`. (Alternatively, define specific ones, but proxy approach is convenient to forward all to one integration). - For Integration, choose **Lambda** and then select the Auth Lambda function (once created). If the Lambda is not made yet, you could create an empty Lambda now or come back later to attach.

We'll hold on adding all routes until we actually create the Lambda functions and ECS services so we have the ARNs. But keep in mind the plan: unify under one API endpoint. This means the front-end only needs to call `https://your-api/` and the gateway fanouts to services. This approach also helps security by centralizing things like authentication, rate limiting, and WAF in one layer [39] [8].

Enabling JWT Auth (optional): If using JSON Web Tokens for user auth (as indicated in architecture), we can integrate that into API Gateway: - If using **Amazon Cognito** for user pools, you can easily set up a Cognito authorizer on routes, which will automatically validate JWTs from Cognito. - If using custom JWT (e.g., issued by our Auth service after login), we can use a Lambda Authorizer or API Gateway's JWT authorizer if we provide the public key or JWKS. A simple approach: have the Auth service issue JWTs signed with a known key. In API Gateway, create a **JWT Authorizer**, provide the issuer URL (if any) and JWKS (if using JWKS endpoint) or upload the public key. Then attach that authorizer to the protected routes (so that API Gateway will reject requests with invalid/missing tokens before hitting our service code). This adds a security layer at the edge [8].

For now, know that API Gateway can handle a lot of this heavy lifting.

WAF integration: Later, when we set up CloudFront, we'll likely route API traffic through CloudFront as well, which has WAF attached [40]. However, we can also attach WAF directly to the API Gateway. Typically, one would use a regional WAF on API Gateway. But since our CloudFront is going to cover the frontend, an elegant solution is to front API Gateway with CloudFront as well (using a custom domain or path pattern). We will discuss that soon in the CloudFront section.

4.3 Deploying Microservices as Lambda Functions

Let's start by deploying a couple of services on AWS Lambda, since it's beginner-friendly and quick to see results.

Example: Auth Service as a Lambda - Assume we have code (Node.js/Express or Python/Flask etc.) for the Auth service. If not, you can create a simple Lambda handler that returns e.g. "Hello from Auth". - In AWS console, go to **Lambda** -> **Create function**. Name it `AuthServiceFunction`. Runtime: choose the language (e.g. Node.js 18 or Python 3.9). - Execution role: choose **Use an existing role** and pick the `nerdwork-lambda-basic-role` (from IAM step). This ensures proper permissions (CloudWatch Logs etc., plus any custom policies we attached). - For now, Author from scratch is fine (we'll edit code online or zip upload). Create function. - In the function code section, you can write a basic handler that matches the API Gateway event structure. For example, in Node.js, something like:

```
exports.handler = async(event) => {
  console.log("Request: ", JSON.stringify(event));
  // simple response
  return {
    statusCode: 200,
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ message: "Auth service alive" })
  };
};
```

This is just a placeholder. Normally, you'd include logic for sign-up, login, etc., possibly routing by `event.path` if using a proxy integration. - Hit Deploy to save the code.

- Now integrate it with API Gateway: In API Gateway Routes, if you made the `/auth/{proxy+}` route, you can edit the Integration to point to this Lambda (choose "Lambda Function" and select `AuthServiceFunction`). Save. Now any call to e.g. GET `/auth/test` will invoke this Lambda.

- Test it: Copy the API invoke URL (like `https://abc123.execute-api.region.amazonaws.com`). Since we didn't set up stages, the URL might be direct or with `/v1` if default stage v1. Try hitting `https://.../auth/test` in a browser or using curl. You should see the JSON message from the Lambda. If you get permissions error, ensure API Gateway has permission to invoke Lambda – normally AWS auto-configures an `AWS::Lambda::Permission` allowing the API Gateway to call your function. If not, you might add it via CLI or the Lambda console (under Configuration > Permissions, add an "API Gateway" trigger which sets the permission).

- Repeat for other simple services: e.g., create a `UserServiceFunction`, `LoyaltyServiceFunction`, `NotificationServiceFunction`. These can be mostly stubs returning dummy data until real logic is added. Attach them to routes like `/users`, `/loyalty`, `/notify` etc. Make sure to assign the same IAM execution role if appropriate (or different roles with tailored permissions if, say, Notification function needs SES access – then attach a policy for SES SendEmail to its role).

Environment Variables: For each Lambda, set configuration like DB connection info or other service URLs via environment variables. Never hardcode secrets here; instead, you might store something like `DB_SECRET_ARN` env var and in code call Secrets Manager using that ARN to fetch credentials. Also set any needed config like an API key for a third party, or mode (dev/prod flags).

VPC Configuration for Lambdas: If a Lambda needs to access VPC-only resources (our RDS database, or ElastiCache, etc.), you **must configure the Lambda to belong to the VPC**. This is under the Lambda's Networking settings: choose the VPC, and then select at least two subnets (private subnets typically) and a security group for the Lambda. The security group for Lambdas can be the same as we use for ECS/app (e.g. `sg-backend`). If the Lambda needs DB access, ensure that SG is allowed by the DB's SG. *Note:* When a Lambda is in a VPC, it will not have internet access by default. That's why we set up NAT Gateway and endpoints – the Lambda in private subnet will use those. If the Lambda calls Secrets Manager or Dynamo, our endpoints cover that; if it needs to call an external API or something like Stripe/Helio, it will go out via NAT.

For example, our AuthService might not need VPC (if it's just handling Cognito or issuing tokens), but Wallet or Payment service might need to query RDS for balances, so those Lambdas should be in the VPC.

4.4 Deploying Microservices on ECS (Optional for heavier services)

For microservices that are better suited for a container (perhaps the Comic service which might integrate with image processing libraries, or if we had a need for persistent WebSocket connections, etc.), we use **Amazon ECS (Elastic Container Service)** with Fargate. Fargate means we don't manage EC2 servers; AWS runs our containers and charges per vCPU/Memory used per second.

Prerequisites: - We need a container image for the service. You or your dev team should have Dockerfiles for each service. If not, create one (for Node.js, base on `node:18-alpine` for example, copy code, `npm install`, `npm start`). Ensure it listens on a configurable port (say 8080). - We also need an **Amazon ECR (Elastic Container Registry)** repository to store the image. We can use one repo per service (e.g., `nerdwork-comic-service`).

Create an ECR repository: - Go to **ECR** in console -> **Create repository**. Name it `nerdwork/comic-service` (for example). Choose visibility private (it's the default). - Create repository. Note its URI (something like `123456789.dkr.ecr.region.amazonaws.com/nerdwork/comic-service`).

Build and push the container (via CLI): You can do this from your local dev machine or in a later CI step. But manually:

```
aws ecr get-login-password --region <your-region> | docker login --username AWS
--password-stdin 123456789.dkr.ecr.<region>.amazonaws.com
docker build -t nerdwork/comic-service:latest .
docker tag nerdwork/comic-service:latest
123456789.dkr.ecr.<region>.amazonaws.com/nerdwork/comic-service:latest
docker push 123456789.dkr.ecr.<region>.amazonaws.com/nerdwork/comic-
service:latest
```

Ensure your AWS CLI user has ECR push permissions (if using the GitHub deploy user, you'd do this in CI ideally).

Now **deploy to ECS:** - Go to **ECS** console. Create a new cluster: choose **Networking only (Fargate)** cluster template. Name it `nerdwork-cluster`. - Next, create a **Task Definition**. Choose Fargate, give it a name like `comic-service-task`. Set task memory and CPU (e.g., 0.5 vCPU and 1GB for a small service; adjust based on needs). - In Task Definition, add a **Container**: - Name: `comic-container`. - Image: use the ECR image URI you pushed (you can use latest tag or a specific version tag). - Memory & CPU reservation: can leave blank if using task-level, or specify if needed. - Port mappings: if the container listens on e.g. 8080, put container port 8080. We might not expose it publicly, but this is for linking with a load balancer. - Environment: add any env variables (e.g., DB connection strings, AWS region, etc.). For sensitive values, you can integrate with Secrets: under Environment, choose **Secret** -> select which secret from Secrets Manager to inject (ECS can inject secrets as env variables). - Log configuration: set up to use CloudWatch Logs – create a log group `/ecs/nerdwork-comic-service` and set region, etc., so logs from container go to CloudWatch.

- Save the task definition.

- Now, create a **Service** for this task (so it runs continuously):

- In ECS, Services -> Create Service. Launch type: Fargate. Cluster: `nerdwork-cluster`. Task definition: select the one we made, choose revision.
- Service name: e.g. `comic-service`.
- Number of tasks: start with 1 (one instance of the container). We can scale out later if needed (either manually or with auto-scaling).
- Deployments: choose rolling update (default).
- Networking: choose our VPC, and **for Subnets, select the two private subnets** (so that tasks run in private subnets).
- Security group: attach the `sg-backend` or create a new one for this service. The SG should allow inbound from wherever needs to call it. If we plan to use an Application Load Balancer for this service, the SG should allow the ALB SG to connect on the container port (e.g. ALB SG to target SG on port 8080).
- Load balancer integration: We have two options:
    1. **Use an Application Load Balancer (ALB):** This is standard if we want to expose the service. You'd create an ALB in the public subnets, have a Target Group for the ECS service (target type IP, since Fargate tasks have IPs), and the service will register tasks with that target group. The ALB can then have a listener (port 80/443) for this service's path. *However,* since we already have API Gateway, we might skip creating an ALB per service and instead let API Gateway call the service directly (via a VPC Link).
    2. **Use API Gateway's VPC Link:** API Gateway can integrate with a *private* HTTP endpoint if we set up a VPC Link (essentially an ENI in our VPC that routes to a NLB or a set of IPs). A common pattern is to put an **NLB (Network Load Balancer)** in front of ECS services (NLB because it's simpler and works with VPC Link) or use the new HTTP Private Integration which might call AWS Cloud Map. This is a bit advanced, but possible.
    3. **Simplest (for now):** We can actually register the service with Cloud Map and then call it directly if needed from other services, but for external client calls, let's use API Gateway + NLB.

Given this is a manual guide and ALB may be simpler conceptually: - Create an **Application Load Balancer** now (go to EC2 console -> Load Balancers -> Create Load Balancer -> Application). Name: `nerdwork-alb`. Scheme: Internet-facing (so it has a public endpoint). IP type: IPv4. Listeners: Start with HTTP :80 (we will use CloudFront + WAF for external, but ALB can be behind CloudFront or just for internal use via API GW). - AZs: select our public subnets (A and B). - Security Group for ALB: create one `sg-alb` that allows inbound on port 80 (and 443 later if needed) from anywhere (0.0.0.0/0) – or more locked down if we only want CloudFront to reach it, but CloudFront uses many edge IPs, so usually open to public and rely on WAF/ CloudFront token for security). - Now, for the ALB target. On ALB creation, it asks for a target group. Create a new target group: * Target type: IP (for Fargate tasks). * Name: `tg-comic-service`. * Protocol: HTTP, Port: 8080 (whatever container listens). * VPC: our VPC. * Health check path: maybe `/health` if your service has it, or `/` for now. - Finish ALB creation. - Now, go back to ECS service creation, in the Load balancer settings: * Load balancer type: Application * Choose the ALB just created. * It will ask to choose listener (select the HTTP:80 listener) and the target group (select `tg-comic-service`). * It will also ask to specify a name for the service's target group attachment (just accept default or name it similarly). - Create Service. ECS will launch the container and register with ALB target group. After a minute or two, you should see the target healthy in the target group.

To test: The ALB DNS (something like `nerdwork-alb-123.region.elb.amazonaws.com`) if you hit it in a browser on the correct path the service expects, you should get a response from the container. But since the ALB has no path routing configured explicitly, by default it sends all to the target group. So hitting ALB root goes to comic-service container root.

Integrate ECS service with API Gateway: We have two patterns: - **Via ALB domain:** Simplest is to skip API Gateway for this service and let CloudFront talk to ALB directly for `/comic/*` paths. But we want unified API and usage of API Gateway features. - **Via API Gateway HTTP integration:** API Gateway HTTP APIs can integrate with **private ALBs** if configured. However, our ALB is internet-facing. We can make it internal only and use VPC Link. Alternatively, API Gateway can integrate with any HTTP URL as a backend (but for a public ALB, that's going out and coming back in – not ideal). - **Better approach:** Make ALB internal (only has internal hostname) and set up an **API Gateway VPC Link**: - In API Gateway, create a **VPC Link** (under integrations -> VPC Links). Select the VPC and the subnets (pick the two private subnets ideally) and a security group that can reach the ALB. Actually, for VPC Link with ALB, AWS suggests using an NLB because VPC Link only works with NLB or Cloud Map integrations for HTTP APIs currently. For REST APIs, you could use ALB. - If using HTTP API with private integration: we might need to use an NLB. Perhaps it's easier: use an NLB for the ECS service (NLB can target IPs of tasks as well). - Given complexity, as an *interim solution*, you could allow the ECS service's ALB to be public and skip API Gateway for that microservice. But then your frontend has to call a different domain or CloudFront has to route differently.

To keep this simpler: we will assume either the ECS-based services are also accessed via API Gateway (which might need an NLB and VPC Link), or we handle them differently. Because of the advanced nature of VPC Link, we might not detail every step here. Instead, note: - **Alternate simpler design:** Deploy all microservice endpoints as Lambdas behind API Gateway (at least initially). The architecture did say "mix of Fargate and Lambda" for cost and flexibility [36], but it's not mandatory to use ECS from day one. If ECS complexity is high for a beginner, you can postpone ECS and use Lambdas for everything possible. Then later migrate heavy ones to ECS when needed (with minimal changes to API Gateway if the interface stays same).

If you do use ECS/ALB, ensure **the ALB is protected**: - The ALB can be put behind CloudFront + WAF by making CloudFront forward to it (CloudFront origin as ALB DNS). - Or at least attach WAF to ALB and restrict access (source IPs). - Possibly restrict the ALB security group to only allow CloudFront IP ranges (which you can get from AWS public IP list) – this is doable but CloudFront has many IPs.

For brevity, we'll proceed with the simpler assumption for now: using Lambda for most services and maybe one ECS as needed, with the understanding that integration via API Gateway is possible but requires additional networking setup (which could be an appendix on its own).

4.5 Backend Summary

At this point, our backend infrastructure includes: - Multiple Lambda functions for various services (all running in our VPC if they need DB access, each with appropriate IAM roles). - Possibly an ECS Fargate service for one or two microservices, fronted by an ALB (with security considerations noted). - API Gateway with routes for each service, mostly invoking Lambdas. This gives us a unified API layer with a single domain endpoint [39]. - The microservices themselves can communicate internally either by direct calls (one Lambda calling another via API Gateway endpoint) or by **EventBridge events** for async communication (discussed in Section 6). - All backend components can reach the RDS database and DynamoDB tables as needed (thanks to being in VPC or having correct IAM permissions).

Before moving on, a quick **cost note**: By using Lambda for many endpoints, we incur zero cost when endpoints are unused (beyond free tier, cost is per ms of execution) [41]. Using Fargate for a constantly running service does incur a baseline cost (for example, 0.25 vCPU + 0.5GB Fargate task might cost a few dollars a month). We try to minimize always-on services; if something can be event-driven or on-demand (like processing jobs, or even an API that is rarely used could be Lambda), we lean that way to save money [37]. The mix of Lambda and Fargate lets cost scale with usage and avoid paying for idle capacity [36] [38].

Now, let's set up the frontend and then cover CI/CD.

5. Deploying the Frontend (Next.js) on AWS

The Nerdwork+ frontend is a Next.js web application. We will deploy it as a static site behind Amazon CloudFront to ensure fast, global access with CDN caching. This approach works best if the Next.js app can be exported to static files (using static generation for pages). If the app requires server-side rendering for some pages, an alternative would be to use Next.js serverless mode on Lambda@Edge or an ECS service running Node.js. But to keep things straightforward (and cost-efficient), we will assume a static deployment (which is often suitable for content sites that fetch data via API calls, as our app likely does via the microservices API).

5.1 Build the Next.js App:

On your development machine or CI pipeline: - Ensure environment config is set so that Next's API calls point to the right backend. For example, if in development you used a local API, update the base URL to the production API Gateway URL or a custom domain (we will likely use CloudFront domain). Typically, you might have an environment variable `API_BASE_URL` that you use in the frontend code to call the backend. - Run `npm run build` (for production build) and optionally `npm run export` if using Next's export feature. If the app is fully static (no getServerSideProps), `next export` will generate an `out/` folder with HTML and assets. If using `next build` without export, you will get a `.next` folder – but those need a server to serve since Next might expect a Node server for dynamic routes. To avoid that, try to use static generation (getStaticProps) or client-side fetching from the APIs, which is likely the case here given the microservices approach. - The output should be a directory (let's assume `out/` or `.next/out`) containing `index.html`, other page HTML files, `assets/` folder with JS chunks, etc.

5.2 Upload Frontend to S3:

We will use the `nerdwork-frontend` S3 bucket for hosting these files. - Using AWS CLI: `aws s3 sync out/ s3://nerdwork-frontend-bucket/ --delete` (the `--delete` option removes old files that are not present in new build). - Or use the console: open the S3 bucket, click **Upload**, and upload the contents of the build directory (ensure the folder structure is preserved).

The files will now reside in S3. Since we kept the bucket private, access is restricted. We need CloudFront to serve them.

5.3 Set up CloudFront CDN:

CloudFront will give us a distribution with a URL (and we can attach our custom domain later). It will cache content globally and also allow us to use HTTPS easily and attach WAF.

Console steps: 1. Go to **CloudFront** in AWS console, **Create Distribution**. 2. **Origin:** Set the origin to our S3 bucket. In Origin Domain, select the `nerdwork-frontend` bucket from the dropdown. It might suggest an origin name automatically. - For an S3 bucket website hosting URL vs S3 bucket directly: * We want CloudFront to fetch from S3 via the bucket's REST API, not the public website endpoint, since our bucket is not public. So choose the bucket name (it will show something like `nerdwork-frontend.s3.amazonaws.com`). - **Origin Access:** We need to ensure CloudFront can access the private bucket. The modern way is to use an **Origin Access Control (OAC)**. Click "Add origin access control" (OAC) or if using older method, "Origin Access Identity (OAI)". * Using OAC: Create a new OAC, set it to allow CloudFront to sign requests (this basically means CloudFront will use a special header to access the bucket). * After creating OAC, we must go to the S3 bucket's permissions and attach a **Bucket Policy** granting access to CloudFront. The console might show a policy snippet or you can manually set:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "<CloudFront OAC origin access identity ARN>"
      },
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::nerdwork-frontend/*"
    }
  ]
}
```

If using OAI, the principal is the OAI user ID. If OAC, it's a service principal. The console can do this if you check "grant access to bucket" during setup. - **Origin path:** empty (unless your site is in a subfolder in the bucket). - **Origin ID:** auto filled or you can name it "FrontendS3".

1. **Default Behavior:** This controls caching and origin request for root path and all not otherwise specified paths.
2. Viewer Protocol Policy: select **Redirect HTTP to HTTPS** (we want HTTPS always).
3. Allowed HTTP Methods: since it's static, GET, HEAD are enough (you can allow OPTIONS too for CORS preflight if your frontend needs to call APIs on a different domain).
4. Cached HTTP Methods: GET, HEAD (others not needed).
5. Cache policy: use **CachingOptimized** or create a custom one. For static assets, default is fine. If we had query strings, etc., we might adjust. Leave it default for now.
6. Compress objects automatically: Yes (to gzip/brotli compress files for faster transfer).

7. (If you had dynamic site and needed cookies or auth at edge, more configs would be needed, but for pure static assets, this is okay.)

8. **Error pages / Default root object:** Set **Default root object** to `index.html`. This means if user hits the distribution URL with no path, CloudFront will serve index.html. Also, if this is a single-page app or uses client routing, you might want all 404 errors to serve index.html (so that unknown routes are handled by the client app). CloudFront allows customizing error responses (e.g. on 404, respond with /index.html). For Next.js with static export, it might generate a 404.html for unknown routes anyway, which you could use instead. For now, you can leave it or set up a rule: "If 404, return /index.html with 200".

9. **WAF and Security:** If you have created an AWS WAF web ACL (we'll discuss WAF rules later in Security section), you can attach it here. It's recommended to attach WAF to CloudFront to filter malicious traffic globally [42] [15]. We might do that after creating a WAF.

10. For now, leave Web ACL as none (we'll add later).
11. TLS Certificate: If you want to use a custom domain (like `app.nerdwork.com`), you need to have an ACM certificate for that domain in us-east-1 (because CloudFront only uses us-east-1 certs). Assuming we might not set up the domain now, you can use the default CloudFront domain (which will be something like d123.cloudfront.net). If you do have a domain ready, request or import an SSL certificate in ACM (in us-east-1), then select "Custom SSL Certificate" here.

12. Price Class: Use "Use only North America & Europe" or "Use All Edge Locations" depending on budget/performance. (Using all gives best performance worldwide but slightly higher cost; for a global audience including Africa, probably want all regions for best latency.)

13. Create the distribution. It will take ~10-15 minutes to deploy.

Once deployed, CloudFront will fetch from S3 and cache content. Test it by accessing the CloudFront URL. You should see the Nerdwork+ frontend load (assuming index.html and assets are set up correctly). If something is missing or you see 403 errors, double-check the bucket policy (it must allow CloudFront OAC/OAI to read objects). The architecture noted using CloudFront for the UI for low latency and cost-effective distribution [12] [13].

CORS for API calls: If your frontend (which is on CloudFront domain) needs to call the API (which might be on a different domain, e.g., the API Gateway URL or a custom api domain), you need to enable CORS on those API responses. The simplest method: in API Gateway settings for each route, enable CORS to allow the origin domain (or `*` in dev). Alternatively, since we might unify under the same domain via CloudFront, next step shows how to use CloudFront for API too.

Optional – Serve API via CloudFront: We can add a second origin in CloudFront pointing to our API Gateway, so that both static site and API calls go through the same CloudFront domain (and thus, we can use the same cookies or avoid CORS issues). For example: - Add an origin in CloudFront: type = custom, domain = your API Gateway domain (or a custom domain for it). If using the execute-api domain, be sure to include the stage if needed. - Then add a **Cache Behavior** with Path pattern like `/api/*` (or whatever base path you want) that points to this API origin. You'd set it to **Cache Disabled** (since API responses are dynamic, unless you want to cache some GETs). Also, set Allowed Methods to include POST, etc., and forward all query strings, headers as needed. If API requires Authorization header, ensure to whitelist it. - With that, requests to `https://<cloudfront>/api/xyz` will be forwarded to API Gateway. You might set up your frontend to use relative paths or the same domain for API.

This unified approach means WAF at CloudFront covers both site and API traffic [40]. The architecture indeed mentioned attaching WAF to CloudFront to filter bad requests globally (and CloudFront adds AWS Shield DDoS protection by default) [43] [40].

Custom Domain Setup (if applicable): If you own a domain (say nerdwork.com), you can create subdomains like `app.nerdwork.com` for the front-end and `api.nerdwork.com` for the API: - In Route 53 (if your DNS is there, or your DNS provider), create a CNAME or Alias record: e.g. `app.nerdwork.com -> CloudFront distribution domain`. For an alias in Route 53, it will list the distribution. - Ensure SSL certificate for `app.nerdwork.com` is on CloudFront (via ACM). - For API, you can set up a custom domain in API Gateway (`api.nerdwork.com`) and then either use that directly or also front it by CloudFront. If fronting by CloudFront path pattern, you might not need a separate domain for API, just use CloudFront's domain.

This is icing on the cake – not strictly needed to function, but it provides a nicer URL and WAF integration.

At this stage, the frontend is live and talking to the backend. We can now focus on tying up some loose ends: event-driven workflows, email service, CI/CD, security best practices, and the NWT ledger specifics.

6. Event-Driven Architecture and Additional AWS Services

Nerdwork+ benefits from an event-driven design for certain features, which helps decouple services. We'll set up **Amazon EventBridge** to handle events and **Amazon SES** for emails in this section. We'll also briefly mention analytics workflow options and the optional use of Cloud Composer/Airflow.

6.1 Amazon EventBridge (Event Bus for Microservice Integration)

EventBridge is a serverless event bus that allows services to publish events that other parts of the system can react to. For example: - When a user reads a comic, the Comic service could emit a `ComicRead` event. The Loyalty service (listening for this event) could then award loyalty points to the user. - When a new user signs up (`UserSignedUp` event), the Notification service could send a welcome email and the Analytics service could log this event. - A scheduled event (cron job) could trigger a cleanup Lambda or a summary report generator.

To set this up: - In AWS console, go to **EventBridge** -> **Event Buses**. There's a default event bus already (you can use it). Or create a dedicated bus (e.g., `nerdwork-bus`) for the application. - **Publishing events:** Our application code will use AWS SDK to put events on EventBridge. For example, in a Lambda after processing an action, you might do:

```
const eb = new AWS.EventBridge();
await eb.putEvents({
  Entries: [{
    EventBusName: 'default', // or 'nerdwork-bus'
    Source: 'nerdwork.comic',
    DetailType: 'ComicRead',
    Detail: JSON.stringify({ userId: '123', comicId: '456', timestamp:
Date.now() })
```

```
  }]
}).promise();
```

Permissions: the Lambda's IAM role must have `events:PutEvents` allowed on that bus.

- **Rules (Subscriptions):** To react to events, create rules. For instance:
- Rule name: `LoyaltyOnComicRead`. Event pattern: you can match `{"detail-type": ["ComicRead"]}` (and optionally source, or other detail fields). Target: choose a Lambda function, e.g., `LoyaltyServiceFunction`. Now, when EventBridge sees a `ComicRead` event, it will invoke Loyalty's Lambda (and we'd code that to award points).
- Another rule: `NotifyOnUserSignup`. Pattern: detail-type `UserSignedUp`. Target: `NotificationServiceFunction` (which sends welcome email via SES).

- You can also target other services, like SNS topics, SQS queues, Step Functions, etc., but Lambdas are straightforward for our microservices.

- **Scheduled events:** EventBridge can do cron schedules without needing a server. For example, to send a weekly summary email or do a DB cleanup nightly. You'd create a rule with a schedule expression (like `cron(0 0 * * ? *)` for midnight UTC daily). Target a Lambda that performs the task.

EventBridge thus acts as the central nervous system for Nerdwork+, enabling loosely coupled interactions. This is all pay-per-use (very cheap per million events), and much easier than polling or direct HTTP calls.

6.2 Amazon Simple Email Service (SES) for Emails

Nerdwork+ will send emails for things like verification, password resets, notifications (new content updates), etc. **Amazon SES** is a reliable and cheap way to send emails.

Verify Identities: SES requires verifying the sender identity (domain or email): - If you have a domain (e.g., nerdwork.com), it's best to verify the domain so you can send from any address on it (like no-reply@nerdwork.com). In SES console, go to **Verified Identities** -> **Verify new domain**. Enter your domain, and choose to generate DKIM records. AWS will give you a few DNS record entries (TXT for SPF, CNAME for DKIM) to add in your DNS provider. Add those, then click verify. Once AWS sees them (might take some minutes), the domain is verified. - If you don't have a domain, you can at least verify an email (like your personal or a test email) to send from, but that's just for sandbox testing.

Move out of Sandbox: New SES accounts are in "Sandbox" which limits sending to verified addresses only. To use SES in production (sending to arbitrary user emails), you must request AWS Support to move you to production. This involves creating a case explaining your use (transactional emails for your app, etc.), and ensuring you have no spamming intentions.

Sending Emails: You can send via: - **SES API/SDK** – where you provide parameters (From, To, Subject, Body). For example, using AWS SDK in a Lambda:

```
const ses = new AWS.SES();
await ses.sendEmail({
  Source: "Nerdwork+ <no-reply@nerdwork.com>",
  Destination: { ToAddresses: [userEmail] },
  Message: {
    Subject: { Data: "Welcome to Nerdwork+" },
    Body: { Text: { Data: "Hello and welcome..." } }
  }
}).promise();
```

Or use the `SendTemplatedEmail` if you create templates. - **SMTP Interface** – you can use SES like an SMTP server (it provides SMTP credentials). Likely not needed if using AWS SDK directly.

We will integrate SES with the Notification service. The Notification Lambda needs permission for `ses:SendEmail` or `ses:SendRawEmail`. You can attach AWS managed policy `AmazonSESFullAccess` (too open, better make a custom: allow SendEmail on identity `no-reply@nerdwork.com` or on your domain identity ARN).

Cost note: SES is inexpensive: first 62,000 emails per month are free when sent from an app hosted in EC2/ Lambda (after that, $0.0001 per email) [44]. Overage and attachment sizes can incur small fees. It's far cheaper than third-party email services at this scale. Just monitor bounce rates and complaints (SES can feed those into CloudWatch or SNS notifications).

6.3 Analytics Workflows and (Optional) Cloud Composer/Airflow

For up to 10k users, our analytics needs might be satisfied by the Analytics microservice (logging events, maybe aggregating some stats). We might not need a full data pipeline yet. However, as the data grows, we might want to run periodic analysis or machine learning jobs. This is where workflow tools like **Apache Airflow** come in handy.

Cloud Composer is Google Cloud's managed Airflow service (not directly applicable to AWS). On AWS, the analogous service is **Amazon Managed Workflows for Apache Airflow (MWAA)** or running Airflow on an EC2/ECS. We mention it as an option: - If Nerdwork+ decides to perform batch analytics (e.g., computing a weekly engagement report, or training a model for comic recommendations), an Airflow instance can orchestrate these tasks with dependencies. - For now, this is likely overkill. Simpler AWS-native solutions could be: - AWS **Glue** or AWS **Athena** for running queries on data in S3. - AWS **Step Functions** for orchestrating a few steps of a job (like extract data, process, load). - Even cron Lambdas via EventBridge if only a few jobs.

So, **we will not set up Cloud Composer in this AWS guide** (since it's GCP), but you should be aware: - If needed, you can set up MWAA from AWS console. It provisions an Airflow environment where you can deploy DAGs. This is a fairly heavy resource (cost hundreds per month), so not something to do unless you have a clear need. - Alternatively, you might run a small Airflow on an EC2 or ECS yourself for cheaper, but that adds ops overhead.

In summary, for analytics up to 10k users, use built-in services (Dynamo, Athena, Redshift if needed for BI, etc.) and only consider Airflow/MWAA when you have complex pipelines that justify it.

7. Cost Optimization Strategies

Building on the choices we've made, let's summarize **cost-saving strategies** implemented and additional tips to keep the AWS bill in check while maintaining performance:

- **Serverless First Approach:** We utilized Lambda for many microservices and on-demand DynamoDB tables, meaning we pay primarily per invocation or per request [41] [11]. At low usage, these costs are negligible (often within free tier limits for the first year). By not running always-on servers, we avoid paying for idle capacity.

- **Right-Sizing and Consolidation:** For our relational DB, we chose either Aurora Serverless v2 or a small RDS instance that scales with usage [27]. We can also run multiple schemas in one DB cluster if needed (e.g., user and content databases sharing one Aurora cluster) to avoid multiple underutilized DB instances [45]. This sacrifices some isolation but saves cost by not duplicating infrastructure for each service when load is low.

- **NAT Gateway minimization:** NAT Gateways are convenient but can become one of the highest costs in a small setup. We used a single NAT instead of one per AZ. This cuts the hourly cost in half. The risk is if that AZ goes down, cross-AZ traffic might be impacted, but we deemed it acceptable for early stage.

- We also aggressively used **VPC Endpoints** (S3, DynamoDB) to route internal AWS traffic without NAT [23]. S3 and Dynamo have no hourly cost, and they avoid NAT data charges. This can significantly reduce the data processing fees on NAT (which are ~$0.05/GB). *Using VPC endpoints for AWS services can save up to 80% of the cost compared to routing through NAT for those services [24].*

- Monitor NAT usage: AWS Cost Explorer or VPC Flow Logs can show how much data egress is going via NAT. If high, see if additional endpoints or architectural changes can cut it down.

- **S3 and CloudFront Free Tier:** Serving the frontend and images via S3+CloudFront is extremely cheap. CloudFront has a free tier of 50 GB data transfer and 2 million requests per month for a year, and even after, the cost per GB is low [13]. CloudFront will cache content globally, reducing load on S3. S3 itself is pennies per GB-month storage and free for ingress; egress to CloudFront is free. So essentially, static content delivery cost is minimal until you scale to very high bandwidth.

- **DynamoDB On-Demand & Free Capacity:** DynamoDB on-demand pricing charges per million requests and per GB storage. The first 25 GB storage and 25 RCUs/WCUs of throughput are free in many regions. At our scale, logging events or storing small items will cost almost nothing initially [11]. As usage grows, we can consider switching to provisioned capacity with auto-scaling to get bulk pricing if needed [46].

- **Aurora Serverless v2 & Automatic Scaling:** If using Aurora Serverless, it scales down to 0.5 ACU when idle, which might be ~$5-10/month depending on region (plus storage) – very cost-effective for an always-on database [27]. We also used a single DB cluster for multiple purposes which avoids multiple idle DBs. Make sure to turn on storage auto-pause (for v1) or at least monitor usage (v2 doesn't pause but will sit at min ACUs). If not using Aurora, a small RDS instance (t4g.micro) is cheap and can be stopped when not in use (for dev environments, you can stop RDS to save cost 7 days at a time).

- **Reserved Instances / Savings Plans (Later):** Once usage stabilizes, consider reserving 1-year or 3-year terms for constant services (like an RDS instance or if we end up with an EC2, or Lambda compute savings plan) to save ~30-50%. Early on, stick to on-demand to keep flexibility.

- **Cleanup Unused Resources:** Since this is a complex setup, ensure that in dev/test, you delete anything not needed. Unused EIPs, NAT Gateways, etc., still cost money. AWS Trusted Advisor and Cost Explorer can help identify idle resources.

- **Monitoring and Alerts for cost:** Use AWS **Cost Explorer** and maybe set up a **budget alert**. You can create an alert to email you if the monthly cost exceeds a threshold (say $500, to warn you before hitting the estimated $580 baseline [47]). Review the cost breakdown: e.g., if CloudFront egress grows, maybe it's a good sign (more users), but also maybe time to use more caching or something [48]. If Lambda costs grow due to duration, consider optimizing code or moving long tasks to ECS if cheaper at that point [49].

- **Development vs Production environment:** If you replicate this setup for dev/staging, don't forget to tear down or scale down resources when not needed. For instance, run non-prod RDS only during work hours or use smaller sizes, etc.

By following these strategies, Nerdwork+ can operate on a lean budget while still providing a robust experience. The architecture was explicitly designed with cost-awareness in mind [50] [51] – using managed services to reduce ops overhead, and ensuring costs scale with actual usage.

8. Continuous Integration and Deployment (CI/CD) with GitHub Actions

Manually setting up infrastructure is one side of the coin; the other is making sure code changes can be delivered to this infrastructure easily. We will use **GitHub Actions** for CI/CD, given the team uses GitHub. We'll outline a pipeline for both the frontend and backend.

8.1 Overview of the Deployment Pipeline

Our CI/CD goals: - **Automate builds and tests:** On each push or pull request, run unit tests, linting, etc., to ensure code quality. - **Build artifacts:** For backend, build Docker images or Lambda packages. For frontend, build the static files. - **Deploy to AWS:** After merging to main (or on a tagged release), automatically deploy the new version: - Push Docker images to ECR (for microservices). - Update ECS services to use the new image or update Lambda functions with new code. - Upload new frontend assets to S3 and invalidate CloudFront cache. - Apply any infrastructure changes (though we did manual setup, some parts like security group rules or new resources might need to be added via infra-as-code in future). - **Security scans:** Possibly incorporate dependency scanning or container vulnerability scans. ECR already scans images on push (if enabled) [18]. We can also run npm audit or other tools in pipeline.

We assume the code is in a GitHub repo (or multiple repos). Let's say: - One repo for frontend, - One repo for backend microservices (monorepo or multi-repo per service, adjust accordingly).

Using GitHub Actions, we'll set up workflows in YAML files in `.github/workflows/` directory of each repo.

8.2 AWS Credentials for GitHub Actions

As discussed in IAM section, choose either: - **Access Key method:** Create an IAM user (e.g. `github-actions-deployer`) with minimal permissions and store its AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY in GitHub as repository secrets. In the workflow, use those to configure AWS CLI. - **OIDC method:** Use GitHub's OIDC provider to assume a role. This requires setting up a role trust with GitHub's identity. It's more secure (no static keys). GitHub provides instructions for this. If going this route, you'd use the `aws-actions/configure-aws-credentials` action with `role-to-assume` set.

For simplicity, let's assume we used secrets.

In GitHub repo settings, add secrets: - AWS_ACCESS_KEY_ID - AWS_SECRET_ACCESS_KEY - AWS_REGION (for convenience)

Make sure the IAM user has permissions needed: - For backend: ECR push, ECS update service, Lambda update, etc. - For frontend: S3 put, CloudFront invalidate.

We might instead create separate users for frontend and backend deploy if we want to limit scope further.

8.3 Frontend Deployment Workflow

Filename: `.github/workflows/deploy-frontend.yml` (in frontend repo)

Trigger: on push to main (and maybe manual trigger). Pseudocode for the workflow steps:

```
name: Build and Deploy Frontend
on:
  push:
    branches: [ main ]

jobs:
  build-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 18

      - name: Install dependencies
        run: npm ci

      - name: Build Next.js app
        run: |
          npm run build
          npm run export  # if using next export for static
      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1  # or your region for S3, though bucket can be
elsewhere
      - name: Sync to S3
        run: aws s3 sync ./out s3://nerdwork-frontend-bucket --delete
      - name: CloudFront Invalidation
        run: aws cloudfront create-invalidation --distribution-id YOUR_DISTR_ID
--paths "/*"
```

Let's explain: We install and build the app, resulting in static files. We configure AWS creds (the `aws-actions` is a nice way, it can also assume roles). Then use AWS CLI to sync the `out` directory to the S3 bucket. Finally, create a CloudFront invalidation for all files (so that the new deployment is served fresh). Invalidation ensures that CloudFront doesn't serve cached old files; it costs a bit if over 1000 paths, but `/*` counts as one invalidation path. We could optimize by only listing changed files if needed.

One more step you might include: - **Cache npm dependencies:** Use actions/cache to cache `~/.npm` or `node_modules` to speed builds. - **Only deploy on specific events:** If you use PRs, you might not deploy on every push to main, maybe only on version bump or a tag. But let's assume main branch auto-deploy is fine.

8.4 Backend Deployment Workflow

If each microservice is in a separate repo, each will have its pipeline. If it's a monorepo, we need a single pipeline that can build and push all services (or just the ones that changed).

For simplicity, assume one repo per service or example for one service (repeat similarly for others):

`.github/workflows/deploy-backend.yml` (in say Auth service repo):

```
name: Build and Deploy Auth Service
on:
  push:
    branches: [ main ]
```

```
jobs:
  build-push:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Docker Buildx (for multi-arch builds if needed)
        uses: docker/setup-buildx-action@v2

      - name: Log in to ECR
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: |

aws ecr get-login-password --region ${{ secrets.AWS_REGION }} | docker login --
username AWS --password-stdin <account-id>.dkr.ecr.<region>.amazonaws.com

      - name: Build Docker image
        run: |
          docker build -t <account-id>.dkr.ecr.<region>.amazonaws.com/nerdwork-
auth-service:${{ github.sha }} .
          # also tag 'latest' for convenience
          docker tag <account-id>.dkr.ecr.<region>.amazonaws.com/nerdwork-auth-
service:${{ github.sha }} <account-id>.dkr.ecr.<region>.amazonaws.com/nerdwork-
auth-service:latest

      - name: Push Docker image
        run: |
          docker push <account-id>.dkr.ecr.<region>.amazonaws.com/nerdwork-auth-
service:${{ github.sha }}
          docker push <account-id>.dkr.ecr.<region>.amazonaws.com/nerdwork-auth-
service:latest

      - name: Update ECS Service
        run: |
          aws ecs update-service --cluster nerdwork-cluster --service auth-
service --force-new-deployment
```

This example logs into ECR, builds the image, pushes it with two tags (commit SHA and latest), then calls ECS to deploy. We assume an ECS service exists named `auth-service` (if we had one running container for Auth). The `--force-new-deployment` on ECS will tell it to pull the new image and restart tasks. If we use the `latest` tag in the task definition, this works but is a bit non-deterministic because `latest` might be cached. A better approach is to update the task definition with the new image URI including SHA tag: - We could use `aws ecs register-task-definition` with a new revision specifying image URI with the commit tag, then update-service to that revision. That's more involved but doable via the AWS CLI in pipeline.

If Auth was a Lambda function (not ECS), we would instead do:

```
      - name: Update Lambda Function
        run: |
          zip -r function.zip . -x "node_modules/*"
          aws lambda update-function-code --function-name AuthServiceFunction --
zip-file fileb://function.zip
```

(This assumes interpreted language lambda. If container image lambda, the process is same as Docker build & push then use `aws lambda update-function-code --function-name X --image-uri image:tag`.)

For microservices that are Lambdas, you could also consider using the Serverless Framework or AWS SAM for deployment, but since we stick to manual, CLI is fine.

The pipeline can also run tests: - Add steps for `npm test` or `pytest` etc. before building image. If tests fail, abort.

Security in CI: We should ensure secrets are protected (GitHub masks them in logs). Use least privilege keys (the IAM user should only be able to do the things above, not nuke resources). Additionally, consider adding a step to run a vulnerability scan: - e.g., use Trivy or Clare to scan the Docker image for known CVEs. Or use `pip audit`/`npm audit` for dependencies. Our architecture mentions that CI pipeline runs security scans on images and alerts on vulnerabilities [18] [25]. You can integrate this and have the pipeline fail (or warn) if critical vulnerabilities are present.

Multi-service coordination: If microservices are in separate repos, their pipelines run independently. If it's a monorepo, you might have one big workflow building all, or one that only builds what changed using path filters. That can be complex – separate repos might be easier for independent deploys.

Finally, ensure that the pipeline artifacts (like the function.zip or built static files) do not inadvertently publish secrets. For example, don't bake Secrets Manager values directly into images; fetch at runtime. Use dummy placeholders in config that get real values from env at runtime.

Our CI/CD, once configured, will enable a new team member to push changes and see them live without manually touching AWS – reducing error and speeding up development.

9. Security Best Practices and Considerations

Security is woven throughout the steps above. Here we'll emphasize key security best practices relevant to Nerdwork+ and how to implement them:

- **Least Privilege IAM:** Every AWS role or user should have only the permissions required [25]. We created specific roles for Lambda, ECS, etc. Avoid wildcards in policies where possible. For instance, if a Lambda only needs read access to one S3 bucket, scope the ARN to that bucket. This limits blast radius if a credential is compromised.

- **No Hard-Coded Secrets:** We use Secrets Manager for database passwords and API keys, and IAM roles for AWS access [14]. This means if someone scans our code repo, they won't find plaintext creds. Also rotate secrets periodically (Secrets Manager can automate rotation for RDS credentials if using Aurora with certain engines).

- **Network Segmentation:** We placed all internal services in private subnets (no public IPs) and only exposed what's necessary (API Gateway, CloudFront/ALB) to the internet. The database is not public, and only the app servers' security group can talk to it. This segmentation (and security group rules) acts as a defense-in-depth layer: even if an attacker somehow runs code on the webserver, they still need to get through to other layers if each layer is restricted.

- **Web Application Firewall (WAF):** We have AWS WAF in front of our application (attached to CloudFront/API Gateway). We should configure WAF with AWS's Managed Rule Groups for common threats (OWASP top 10) [15]. These rules will, for example, block SQL injection attempts, XSS, bots, etc., before they hit our app. We can add custom rules too, like rate-limiting (if >100 requests from an IP in a minute, block it for a while) [52], or block specific countries if needed. WAF also provides logging of malicious requests for further analysis [53]. It's essentially our shield at the edge.

- **DDOS Protection:** AWS CloudFront and API Gateway are both protected by AWS Shield Standard (free, auto-enabled) which guards against common DDoS attacks at the network and transport layer. WAF covers the application layer. This layered approach ensures resilience against denial of service attacks without us having to do much.

- **Secure API Endpoints (Auth):** Ensure that protected API routes require valid authentication (JWT or Cognito token). API Gateway can handle JWT validation (with Cognito or a public key) so that invalid tokens never hit our services [8]. Use **HTTPS** everywhere – which we do by using CloudFront and API Gateway (both only use HTTPS for communication). If we set up custom domains, get AWS Certificate Manager SSL certs (free) for them.

- **Input Validation and Encoding:** In the application code, continue to validate all inputs (even with WAF as first line). For example, on forms or API payloads, use library validators or checks to avoid processing unexpected data. Also, encode outputs to prevent XSS in the frontend. These are more development best practices but worth mentioning for completeness.

- **Secure Configurations:**

- S3 buckets: Keep them private (as we did) and use principles of least access. If any bucket must be public (none in our design except via CloudFront), enable **Block Public Access** at account level to avoid accidental public buckets. Use bucket policies to only allow access via CloudFront OAC or certain principals.
- RDS: Ensure encryption at rest is enabled (AWS does by default for Aurora). Also enable **RDS automatic backups** (retention 7+ days) for disaster recovery.
- CloudWatch Logs: Set retention periods for logs so they don't live forever (we might choose 30 days or 90 days depending on compliance needs). This balances forensic needs with cost and privacy.
- CloudTrail: Turn on AWS CloudTrail for the account. This logs all AWS API calls, which is vital for auditing. If something happens, CloudTrail can tell who did what. Store these logs in an S3 bucket (which you can encrypt and restrict).
- GuardDuty: As noted in architecture, enable **AWS GuardDuty** [54] . It's a threat detection service that will continuously monitor CloudTrail, VPC Flow Logs, DNS logs for signs of malicious activity or compromised instances. It's relatively low cost and high value for security oversight.
- AWS Config: Consider enabling AWS Config rules for security posture. For example, a rule to flag if any S3 bucket becomes public, or if security groups become too open. The architecture mentioned using AWS Config to ensure best practices like no public S3 buckets [25] .

- **Patch Management:** Keep Lambda runtime up to date (use latest Node/Python versions). For container images, apply updates regularly (the CI pipeline or a Dependabot can alert of base image vulns). ECR's image scan or other scans should be reviewed and fix high severity issues promptly [18] [25] .

- **Application Security (JWT & Permissions):** The Auth service should use robust password hashing (if not offloaded to Cognito). Implement measures like account lockout on many failed logins, and possibly MFA for admin accounts. The JWT tokens should be signed with a strong algorithm and ideally short-lived with refresh tokens if needed. Services should verify the JWT signature and claims on every request to protected endpoints, and enforce authorization (e.g., a user can only modify their own data, creators only their content, etc.). Use a standard library for JWT verification to avoid mistakes.

- **Network Security & Monitoring:** Use security groups as needed to isolate components (we did e.g. DB SG only allows app SG). Enable **VPC Flow Logs** on the VPC if you want to analyze traffic patterns or detect anomalous flows (store them in CloudWatch or S3). Although at this scale it might be a lot of data to sift through, it's useful for security audits.

- **Auditing and Incident Response:** Set up alerts for important events. For example, if GuardDuty finds something or if an IAM role credentials are used in an unusual way, get notified. AWS Security Hub can centralize findings from GuardDuty, Config, etc. Though a small team might just manually check or rely on emails. If using an external SIEM like Wazuh (mentioned in docs [55]), integrate CloudWatch Logs and other sources to it.

- **Penetration Testing and Drills:** Once things are running, it's valuable to do a pen-test or at least use vulnerability scanners (like OWASP ZAP or others) against the API and frontend to catch any misconfigurations or code issues. AWS allows pen-testing on your own infrastructure (there's a policy that you can test some services without prior approval, e.g. EC2, CloudFront, etc., but check AWS policy on penetration testing).

To sum up, our approach ensures multiple layers of security: **Perimeter (WAF, Shield)** [15] **-> Network (VPC isolation, SGs) -> Identity (IAM, Cognito/JWT) -> Application (validation, roles) -> Data (encryption, backup) -> Monitoring (CloudTrail, GuardDuty)** [42] [54]. By adhering to these best practices, we help protect Nerdwork+ and its users' data from threats while maintaining a smooth user experience.

10. Implementing the NWT Ledger with Double-Entry Accounting (RDS)

One unique aspect of Nerdwork+ is the **NWT (Nerdwork Token) in-app currency**. The platform needs to track user balances and token transactions in a reliable way. The team has decided to use a **double-entry accounting system** for this ledger [56]. This approach is borrowed from finance: every transaction is recorded with at least two entries – a debit in one account and a credit in another – so that the books always balance (total credits = total debits).

Why double-entry? It prevents tokens from being accidentally created or destroyed without record, and makes it easier to audit and find errors. It ensures consistency: the sum of all user balances plus the system's own balance should always net to zero (considering some baseline like initially 0 or considering all issued tokens as debits from a "mint" account and credits to users).

Schema Design in RDS (PostgreSQL):

We will implement this in our relational database (RDS). A possible schema: - **accounts** table – listing each account and its current balance. Each user has an account (or multiple accounts if needed, but likely one per user), and the platform itself might have one or more accounts (e.g., a "treasury" account for the system or separate accounts for revenue, fees, etc.). - Columns: `account_id` (PK), `user_id` (nullable if it's a user's account vs system account), `account_name` (like 'User 123 Wallet' or 'Platform Revenue Account'), `balance` (cached current balance for convenience), … maybe `type` (asset/liability for accounting, but not strictly needed if we treat user balance as liability and system as asset). - **transactions** table – each transaction (like a transfer, purchase, reward issuance) gets a transaction record for grouping. - Columns: `transaction_id` (PK), `timestamp`, `type` (e.g., 'TIP', 'PURCHASE', 'REWARD', etc. for business logic categorization), `reference` (maybe an external reference ID or notes like "ComicID 456 purchase"), etc. - **entries** table – each transaction has two or more entries, each entry represents either a debit or credit to an account. - Columns: `entry_id` (PK), `transaction_id` (FK to transactions), `account_id` (FK to accounts), `amount` (could be positive for credit and negative for debit, or have a separate column indicating debit/credit), `currency` (just in case, 'NWT'), `description` (like "debit from User A", "credit to User B"). - We ensure for each transaction, the sum of all its entries' amounts = 0. For a simple two-entry transaction, one will be positive, one negative of equal magnitude.

For example, say User A tips 50 NWT to User B: - Accounts: A (user A's wallet), B (user B's wallet). - Transaction T created (type = 'TIP'). - Entries: - Entry 1: account = A, amount = -50 (debit A's account by 50, reducing A's balance) - Entry 2: account = B, amount = +50 (credit B's account by 50, increasing B's balance) - After committing, the accounts table for A and B should update balances accordingly (A's balance = old balance - 50, B's = old + 50).

Similarly, for a purchase of content where user spends NWT: - Suppose user A buys an item for 100 NWT that goes to the platform (platform revenue account): - Transaction type 'PURCHASE'. - Entry 1: account = A (user), amount = -100. - Entry 2: account = PlatformRevenue, amount = +100. This way, user's balance down, platform's up (representing platform "owes" service equal to that value).

For rewarding NWT via Loyalty: - The platform might "mint" NWT into existence for rewards (if it's off-chain and fully controlled, essentially the supply can increase via rewards). Or maybe there was a pre-allocated pool. - Either way, treat it as coming from a system "Treasury" account: - If user gets 20 NWT reward: - Entry 1: account = PlatformTreasury, amount = -20 (system account down, or if system account starts at some large number, it decreases; or it could even go negative representing outstanding liability). - Entry 2: account = User, amount = +20.

The double-entry ensures if tokens are created, it's clear (the treasury account will show a negative change, meaning effectively the system issued value to user).

Implementing in Code (Wallet Service): - The Wallet service will provide an API like `POST /wallet/transfer` or internal functions for other services to call when a token transfer or issuance happens. - When such an API is called (say user A tipping B): - The service will start a **database transaction** (very important to maintain atomicity). - It will insert a new row in `transactions` (get an ID). - It will insert two rows in `entries` for the debit and credit. - It will update the `accounts` balances: e.g., `UPDATE accounts SET balance = balance - 50 WHERE account_id = A` and similarly `+50` for B. - Commit the transaction. If any part fails (e.g., insufficient balance), roll back the whole thing so nothing is partially applied. - Use database constraints or application logic to enforce balance cannot go negative (or decide if it can; maybe a user cannot go below 0 NWT, so enforce that by check before update or a CHECK constraint). - We can also enforce the double-entry integrity by a trigger or stored procedure: e.g., a trigger on entries insertion could ensure that for each transaction, sums zero out (this might be complex to do purely in constraints; easier is to trust the application or have a procedure handle it).

The internal ledger model being double-entry ensures **auditability**: at any point, you can sum all accounts balances and if the system is correct, the net should equal 0 (if we define one account as source of truth for issued tokens). Also you can produce a ledger report for each account to see all debits/credits (like a bank statement for users, and for the system accounts) [57] .

Additionally, if an error or anomaly happens (say somehow a transaction was recorded with only one entry due to a bug), the sums won't match and it will be evident that the ledger is out of balance – prompting investigation. This is much harder to detect in a single-entry system where you might just have "add/subtract" operations on balances.

Ensuring Consistency: - Use proper isolation in the database (e.g., serializable or repeatable read transactions when updating balance to avoid race conditions). Or we can decide to not store `balance` in accounts at all and always compute by summing entries. But computing on the fly is slow as transactions grow, so usually you store balances and keep them in sync via the entries. - If storing balance, a nice trick is to use Postgres **FOR UPDATE** locking on the account rows when updating, to serialize concurrent updates to the same account. - We could also implement the ledger as an append-only log (entries only) and derive balance from it daily, but real-time balance display is needed, so best to update it transactionally.

Double-checking with Accounting Principles: In accounting terms: - User accounts could be considered liability (the platform "owes" the tokens to the user in some sense, since user can spend them) – these would be credit-normal accounts. - Platform cash or treasury accounts are asset or equity – debit-normal accounts. - Double-entry rule: total assets = total liabilities + equity. In our simplified model, maybe treat all NWT issued as a liability to users and an equal asset in system, so they cancel out. - This is more conceptual; our implementation just needs to ensure every credit has a matching debit [58] .

By implementing the NWT ledger this way, we align with the design requirement of an internal ledger audit and fraud detection capability [59] . It will be possible to answer queries like “How did this user get X NWT?” by tracing transactions, or “Is there any NWT that isn’t accounted for?” by running a sum check. The system can even implement triggers to flag if any transaction doesn’t balance or if any account goes negative beyond allowed, etc.

Example Walk-through (User Purchase): - User A buys a comic chapter for 10 NWT. The platform might split that 10 NWT as revenue (or perhaps share with creator, but that’s another complexity). - Say platform takes 2 NWT as fee (goes to PlatformRevenue account) and 8 NWT goes to Creator’s account (User B). - We can record that as one transaction with three entries: - Debit User A: -10 - Credit User B (creator): +8 - Credit PlatformRevenue: +2 - All under transaction “PurchaseChapter #123”. Sum of entries = 0 (8+2-10=0). - This shows multi-entry transaction usage.

Our database can handle that easily since entries are just rows linked to a transaction.

We should document such logic in the codebase for maintainers and possibly implement some unit tests on the ledger calculation (simulate transactions and assert balances).

Auditing: We can periodically run an audit job (maybe a Lambda triggered daily) that: - Sums all entries per account and compares to stored balance (to catch any discrepancies if manual corrections happened). - Sums all balances to ensure net zero relative to a base account. - Checks for any orphaned entry without a pair (shouldn’t happen if all is atomic). - That can alert us if something’s off, which is part of “fraud detection” in the deliverables [59] .

By carefully designing and coding the Wallet service with these principles, Nerdwork+ ensures NWT token accounting is **robust, transparent, and error-resistant**. This double-entry ledger, much like in a bank or accounting system, is essential for maintaining trust in the in-app currency [57] and will support future features (like if ever allowing refunds, reconciliation, etc., we have the data to do it accurately).

Conclusion & Next Steps

Congratulations! You have manually set up the core cloud infrastructure for Nerdwork+ on AWS. We covered everything from networking to deployments, and integrated cost-conscious and security-minded practices at each step. By following this handbook, a new team member or engineer should be able to spin up the Nerdwork+ environment and understand the rationale behind each component.

Recap of what we’ve done: - Created a VPC with public/private subnets, one NAT Gateway (with VPC Endpoints to minimize its use) [23] [24] . - Configured IAM roles and policies for secure access (no leaked keys, least privilege) [14] . - Set up data stores: Aurora Postgres (Serverless) for relational needs, DynamoDB for NoSQL needs, S3 for static files – all optimized for cost and scale [26] [10] . - Deployed backend microservices on Lambda and ECS, fronted by API Gateway, implementing a microservices architecture with decoupling and stateless principles [2] [36]. - Deployed the Next.js frontend on S3+CloudFront for efficient global delivery [12] [13]. - Integrated supporting services: EventBridge for events, SES for emails, with thoughts on future analytics workflows. - Emphasized cost optimizations like serverless usage, right-sizing, and monitoring [50] [51]. - Established CI/CD pipelines using GitHub Actions for consistent and safe deployments [18]. - Outlined robust security measures (WAF, encryption, auth, auditing) to protect the platform and users [15] [54]. - Explained the design of the NWT ledger in the database using double-entry bookkeeping to ensure financial integrity of the token system [56] [57].

With this infrastructure in place, Nerdwork+ is well-prepared to launch and handle growth to the first 10,000 users and beyond. The architecture can scale by increasing Lambda concurrency or ECS tasks per service, adding read replicas or larger instances for the database if needed, and leveraging AWS's managed services to handle more load (e.g., moving analytics to Redshift or using Cognito for auth if user count grows, etc.).

Next Steps / Recommendations: - **Infrastructure as Code:** While we did everything manually here for learning purposes, consider using tools like AWS CloudFormation or Terraform to codify this setup. That makes it reproducible and version-controlled. AWS CDK is another great option to define infrastructure in code (TypeScript/Python). - **Monitoring & Observability:** Set up dashboards in CloudWatch (or use third-party like Datadog) to monitor key metrics: Lambda invocation errors/throttles, ECS CPU/memory, RDS CPU connections, etc. Also set up alarms (e.g., on high error rate or low DB storage). - **Testing the whole system:** Do an end-to-end test: register a user, simulate a token purchase, ensure the ledger entries are correct, ensure an email is received, etc., to validate the plumbing. - **Document and Train:** Onboard other team members by walking through this handbook. Keep documentation updated as architecture evolves (maybe new microservices or refactoring from Lambda to containers, etc.).

This guide can be your reference as you operate and expand Nerdwork+. Keep an eye on AWS announcements, as new services or features (like improved integration between API Gateway and ECS, or cheaper serverless offerings) could further simplify or reduce cost.

By adhering to the structured approach and best practices outlined, you'll ensure Nerdwork+ is not only running successfully in the cloud but is also secure, maintainable, and poised to scale when user demand grows.

Good luck with Nerdwork+, and happy cloud building!

Sources:

- AWS Architecture and Design references from Nerdwork+ documentation, confirming the use of AWS services like Lambda, ECS, RDS, S3, CloudFront, etc., and security/cost strategies [1] [13].
- AWS documentation for best practices and setup of networking, endpoints, and WAF [23] [15].
- Nerdwork+ technical overview for NWT token and ledger requirements, emphasizing a double-entry ledger model for accuracy and auditability [56].
- Modern best practices for ledger design in wallet applications, highlighting the importance of balanced transactions (debits = credits) for financial systems [57].

---

1 19 47 55 59 Nerdwork+_Product_Overview.docx.pdf
file://file-KzkdCYTc1idk6PQ4u7Poqg

2 3 5 6 8 9 10 11 12 13 14 15 16 17 18 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 48 49 50 51 52 53 54 Nerdwork- Platform Architecture (0–10k Users).pdf
file://file-W3sPJgoKzkvqmDMoRvsUEt

4 7 56 Nerdwork - Product Overview (NWT bias).pdf
file://file-WqtRCYyqRvqnTkqFDdR5Xw

20 21 22 23 Example: VPC with servers in private subnets and NAT - Amazon Virtual Private Cloud
https://docs.aws.amazon.com/vpc/latest/userguide/vpc-example-private-subnets-nat.html

24 Save by Using Anything Other Than a NAT Gateway - Vantage.sh
https://www.vantage.sh/blog/nat-gateway-vpc-endpoint-savings

57 58 Accounting for Developers, Part II | Modern Treasury Journal
https://www.moderntreasury.com/journal/accounting-for-developers-part-ii