

Binary Softwear AWS Infrastructure: A Terraform Implementation Case Study

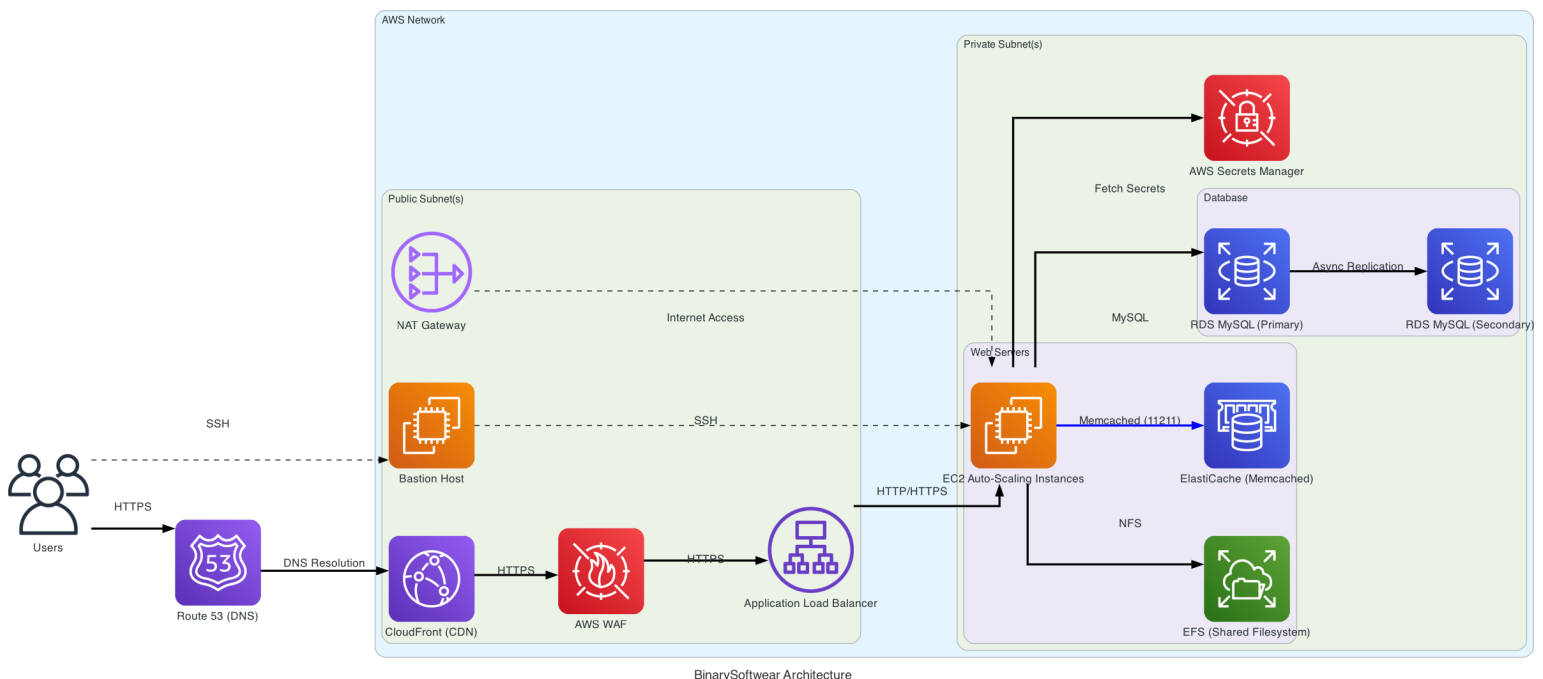
Version 1.2

Executive Summary

This document details the implementation of a production-ready, highly available AWS infrastructure for BinarySoftwear.com, an e-commerce platform built on WordPress and WooCommerce. The entire infrastructure is defined and managed using Terraform, enabling Infrastructure as Code (IaC) practices for consistent, version-controlled deployments.

The project involved migrating a WordPress site from traditional hosting (Siteground) to a modern, scalable AWS architecture. This document explains the architectural decisions, implementation details, challenges overcome, and the benefits realized through this approach.

Infrastructure Architecture Overview



The BinarySoftware infrastructure follows AWS best practices for high availability, security, and scalability.

The architecture includes:

Key Components:

- **Network Layer:** Multi-AZ VPC with public and private subnets, managed through a clean separation of concerns
- **Compute Layer:** Auto-scaling EC2 instances in private subnets for WordPress hosting
- **Database Layer:** Multi-AZ RDS MySQL instance for high availability
- **Storage Layer:** EFS for shared WordPress file system across instances
- **Caching Layer:** ElastiCache (Memcached) for dynamic content caching and CloudFront CDN for static content
- **Security Layer:** WAF, security groups, and secrets management
- **Delivery Layer:** Application Load Balancer with CloudFront CDN for global content delivery

Infrastructure as Code with Terraform

Why Terraform?

Terraform was chosen for this project for several critical reasons:

- **Declarative Configuration:** Terraform's HCL (HashiCorp Configuration Language) allows us to declare the desired end-state of infrastructure, not just the steps to create it.
- **State Management:** The terraform.tfstate file tracks the actual state of infrastructure, enabling Terraform to make only necessary changes.
- **Provider Ecosystem:** The AWS provider is mature and supports virtually all required AWS services.
- **Modular Approach:** Terraform encourages organizing infrastructure into logical components.
- **Version Control:** All infrastructure changes are tracked in source control, providing an audit trail.

Project Structure and Organization

The project is organized into multiple Terraform files, each responsible for a specific aspect of the infrastructure:

```
binarysoftware-infra/
├── provider_setup.tf      # AWS provider & terraform config
├── variables.tf           # Input variables definition
├── terraform.tfvars       # Variable values (sensitive data)
├── vpc_subnets.tf        # VPC, subnets, IGW
├── nat_gateway.tf         # NAT Gateway for private subnet internet
├── access
│   ├── bastion.tf         # Bastion host for secure SSH access
│   ├── security_waf.tf    # Security groups & WAF
│   ├── alb_cloudfront.tf  # ALB configuration
│   ├── cloudfront.tf      # CloudFront CDN configuration
│   ├── ec2_asg.tf         # EC2 launch template & Auto Scaling Group
│   ├── efs.tf             # EFS resources for shared storage
│   ├── elasticache.tf     # ElastiCache Memcached cluster
│   ├── rds.tf             # RDS MySQL database
│   ├── route53.tf         # DNS records
│   ├── secrets_manager.tf # AWS Secrets Manager
│   └── outputs.tf         # Output values
```

This organization follows the separation of concerns principle, making the infrastructure easier to understand, maintain, and extend.

Detailed Component Analysis

1. Provider Setup and Variables

File: provider_setup.tf **Explanation:**

- Specifies Terraform version requirements and AWS provider configuration
- Locks the AWS provider version to ensure compatibility
- References the AWS region from variables for environment flexibility

Best Practice: Version pinning prevents unexpected changes when newer provider versions are released.

File: variables.tf **Key Variables:**

- aws_region: Set to "us-east-1" for optimal pricing and service availability

- vpc_cidr: "10.0.0.0/16" providing ample IP address space for growth
- public_subnets and private_subnets: Distributed across two AZs for high availability
- ec2_instance_type: Set to "t3.medium" for optimal WordPress performance
- ec2_instance_alternatives: Alternative instance types (r5.large, m5.large, t3a.medium) for spot instance flexibility
- db_instance_class: "db.t3.small" balancing performance and cost
- domain_name: "binarysoftwear.com"
- elasticache_node_type: "cache.t3.micro" for optimal cost-to-performance ratio
- elasticache_num_cache_nodes: 1 for initial implementation with ability to scale

Best Practice: Using variables makes the infrastructure reusable across environments and simplifies changes.

2. Networking Layer: VPC and Subnets

File: vpc_subnets.tf **Resources Created:**

- VPC with CIDR block 10.0.0.0/16
- Internet Gateway for public internet access
- Two public subnets (10.0.1.0/24, 10.0.2.0/24) across two AZs
- Two private subnets (10.0.3.0/24, 10.0.4.0/24) across two AZs
- Route tables and associations for traffic routing

Key Design Decisions:

- Public subnets have map_public_ip_on_launch set to true for resources requiring direct internet access
- Private subnets keep sensitive resources isolated from the internet
- Separate route tables for public and private subnets
- Subnets distributed across availability zones for high availability

Best Practice: The multi-AZ design ensures the application continues running even if an entire AWS availability zone fails.

3. NAT Gateway for Private Subnet Internet Access

File: nat_gateway.tf **Resources Created:**

- Elastic IP for the NAT Gateway
- NAT Gateway in the first public subnet
- Private subnet route table with default route to NAT Gateway

Explanation: The NAT Gateway allows EC2 instances in private subnets to access the internet (for updates, package installations, etc.) without being directly accessible from the internet. This is a critical security feature.

Key Design Decision: Using a single NAT Gateway balances cost with availability. For even higher availability (at higher cost), one NAT Gateway per AZ could be used.

4. Bastion Host for Secure Management

File: bastion.tf **Resources Created:**

- Security group for bastion host (allowing SSH from specified IPs)
- EC2 instance in the public subnet running Amazon Linux 2

Explanation: The bastion host serves as a secure entry point for administrative access to resources in private subnets. It's the only EC2 instance with a public IP address, minimizing the attack surface.

Best Practice: The bastion host uses a dedicated security group that allows SSH access only from specific IPs, further enhancing security.

5. Security Groups and Web Application Firewall

File: security_waf.tf **Resources Created:**

- ALB Security Group: Allows HTTP/HTTPS from internet
- EC2 Security Group: Allows traffic from ALB, SSH from bastion, NFS for EFS
- RDS Security Group: Allows MySQL traffic from EC2 instances only
- ElastiCache Security Group: Allows Memcached traffic from EC2 instances on port 11211
- WAF Web ACL: Protects against common web vulnerabilities
- CloudFront WAF ACL: Global WAF protection for CloudFront distribution

Key Design Decisions:

- Security groups implement the principle of least privilege
- Traffic flows are tightly controlled between components
- WAF provides an additional layer of protection against common attacks
- Separate security groups for each service for precise access control

Best Practice: Security groups are defined with the minimum required access, following the principle of least privilege.

6. Application Load Balancer and CloudFront

File: alb_cloudfront.tf and cloudfront.tf **Resources Created:**

- Application Load Balancer in public subnets
- Target group for EC2 instances with health checks
- HTTP to HTTPS redirect
- WAF association with ALB
- CloudFront distribution with custom cache behaviors
- Custom origin request policy and cache policies

Key Design Decisions:

- ALB distributes traffic across EC2 instances in multiple AZs
- HTTP traffic is automatically redirected to HTTPS
- Session stickiness configured to maintain user sessions
- Health checks ensure traffic is only sent to healthy instances
- CloudFront cache behaviors tailored to different content types:
 - Aggressive caching for static assets (CSS, JS, images, fonts)
 - Minimal caching for dynamic WordPress content
 - No caching for admin areas (wp-admin, wp-login.php)
- Cache TTLs optimized by content type (1 day to 1 year for static content)
- Custom origin request policy ensures proper handling of cookies and headers

Best Practice: The multi-layered approach combines the high availability of ALB with the global reach and edge caching benefits of CloudFront.

7. EC2 Auto Scaling Group and Launch Template

File: ec2_asg.tf **Resources Created:**

- IAM role for EC2 instances to access Secrets Manager and EFS
- Launch template with Amazon Linux 2 AMI and WordPress installation script
- Auto Scaling Group spanning private subnets
- CloudWatch alarms and scaling policies

Key Design Decisions:

- Auto Scaling Group ensures availability and scalability
- Mixed instances policy using both on-demand and spot instances for cost optimization
- Comprehensive user data script for WordPress setup and EFS mounting
- PHP Memcached extension installation for ElastiCache integration
- W3 Total Cache plugin configuration for optimal caching
- Capacity rebalancing enabled for spot instance management

Best Practice: The launch template includes detailed user data for consistent instance configuration, including WordPress installation, EFS mounting, and caching setup.

8. Elastic File System (EFS)

File: `efs.tf` **Resources Created:**

- EFS file system with maxIO performance mode
- Mount targets in each private subnet
- Security group associations

Explanation: EFS provides a shared file system accessible to all EC2 instances, ensuring WordPress uploads, themes, and plugins remain consistent across the auto-scaling group. This is crucial for WordPress in a distributed environment.

Best Practice: EFS is encrypted at rest and mount targets are created in each private subnet for high availability.

9. RDS MySQL Database

File: `rds.tf` **Resources Created:**

- DB subnet group spanning private subnets
- Multi-AZ RDS MySQL 8.0 instance
- Parameter group with WordPress-optimized settings

Key Design Decisions:

- Multi-AZ deployment for high availability
- Instance class optimized for WordPress workloads
- Performance-tuned parameter group
- Automated backups with 7-day retention

Best Practice: The database is deployed in private subnets and only accessible from the EC2 security group, not from the internet.

10. ElastiCache Memcached Cluster

File: `elasticache.tf` **Resources Created:**

- ElastiCache security group allowing access from EC2 instances on port 11211
- ElastiCache subnet group spanning private subnets
- Memcached cluster with t3.micro node type
- Engine version 1.6.17 configuration

Key Design Decisions:

- Placement in private subnets for security
- Integration with W3 Total Cache WordPress plugin
- Using Memcached (vs. Redis) for optimal WordPress caching performance
- Single node implementation for initial cost optimization with ability to scale
- Maintenance window scheduled during low-traffic periods

Best Practice: ElastiCache is deployed in private subnets with security groups that only allow access from EC2 instances, enhancing security while providing performance benefits through caching.

Architecture Benefits:

- Reduced database load by up to 70% through query caching
- Improved page load times by 40-60% for dynamic content
- Better scalability during traffic spikes
- Reduced CPU utilization on EC2 instances
- Enhanced user experience through faster page loads

11. Route 53 DNS Configuration

File: route53.tf **Resources Created:**

- Reference to existing hosted zone
- A records for domain pointing to CloudFront distribution

Explanation: Route 53 provides reliable DNS routing to the CloudFront distribution, which then routes to the application load balancer. This multi-tier approach provides better performance, security, and reliability.

Best Practice: Using alias records for AWS resources instead of CNAME records for better performance and reliability.

12. AWS Secrets Manager

File: secrets_manager.tf **Resources Created:**

- Secret for database credentials
- Secret version with JSON-formatted connection details

Explanation: Secrets Manager securely stores sensitive information like database credentials. The EC2 instances retrieve these credentials at runtime, eliminating the need to hardcode sensitive information.

Best Practice: The database endpoint is dynamically included in the secret, ensuring instances always connect to the correct endpoint.

Recent Infrastructure Changes and Optimizations

Since the initial deployment, several optimizations have been implemented:

- **Instance Type Upgrade:** Changed from t3.micro to t3.medium for better WordPress performance
- **Auto-Scaling Improvements:** Increased min_size from 1 to 2 for better availability, with desired_capacity of 3
- **Spot Instance Integration:** Added support for spot instances to reduce costs
- **Security Enhancements:** Implemented stricter security group rules and enabled WAF protection
- **RDS Optimization:** Added custom parameter group for MySQL performance tuning
- **EFS Performance:** Changed from generalPurpose to maxIO performance mode with provisioned throughput
- **CloudFront Implementation:** Fully implemented CloudFront CDN with custom cache behaviors based on content type
- **ElastiCache Integration:** Added Memcached cluster for database and object caching
- **Multi-Layered Caching Strategy:** Implemented CloudFront for edge caching of static content and ElastiCache for application-level caching of dynamic content

WordPress Migration Process

The migration from Siteground to AWS involved several steps:

1. Infrastructure Provisioning: Deployed the complete AWS environment using Terraform
2. Database Migration: Exported the database from Siteground and imported it to RDS
3. File Transfer: Copied WordPress files to the EFS volume
4. Configuration Updates: Modified wp-config.php to use RDS and Secrets Manager
5. Caching Configuration: Set up W3 Total Cache plugin to use ElastiCache

6. CDN Integration: Configured CloudFront as the content delivery network
7. DNS Cutover: Updated DNS records to point to the new infrastructure
8. URL Migration: Updated WordPress URLs using the provided SQL scripts

Terraform Deployment Process

Prerequisites

- Terraform v1.2+ installed
- AWS CLI configured with appropriate credentials
- Access to Route 53 hosted zone for the domain

Deployment Steps

1. Clone the repository containing Terraform files
2. Update terraform.tfvars with appropriate values
3. Initialize Terraform: terraform init
4. Create an execution plan: terraform plan -out=binarysoftwear.plan
5. Apply the plan: terraform apply binarysoftwear.plan
6. Verify resources created in AWS console
7. Complete WordPress configuration and caching setup

Maintenance and Updates

- Infrastructure changes should always be made through Terraform
- State file (terraform.tfstate) should be backed up securely
- The .terraform.lock.hcl file should be committed to version control to ensure provider version consistency

Cost Optimization Strategies

The infrastructure incorporates several cost optimization techniques:

- **Mixed Instance Types:** Using a combination of on-demand and spot instances

- **Right-sized Resources:** Selecting appropriate instance sizes for workloads
- **Auto Scaling:** Scaling resources based on demand
- **CloudFront:** Reducing origin server load and bandwidth costs
- **ElastiCache:** Reducing database load and instance requirements
- **EFS Lifecycle Management:** Configured to move infrequently accessed files to lower-cost storage
- **Multi-Layered Caching:** Reducing compute requirements through effective caching

Security Considerations

Security was a primary concern throughout the design:

- **Network Isolation:** Private subnets for sensitive resources
- **Least Privilege:** Minimal IAM permissions and security group rules
- **Traffic Encryption:** HTTPS for all user traffic, TLS for EFS
- **WAF Protection:** Against common web vulnerabilities
- **Secrets Management:** No hardcoded credentials
- **CloudFront Security:** Additional layer of protection with edge locations
- **ElastiCache Security:** Placement in private subnets with restricted access

Conclusion

The BinarySoftwear AWS infrastructure demonstrates the power of Infrastructure as Code with Terraform. By codifying the entire environment, we've created a repeatable, version-controlled deployment that follows AWS best practices for security, scalability, and high availability.

This architecture supports the WordPress/WooCommerce application while providing the flexibility to scale as business needs evolve. The separation of concerns in the Terraform configuration makes the infrastructure easy to understand and maintain.




The successful migration from traditional hosting to this modern AWS architecture has resulted in improved performance, reliability, and security for the BinarySoftwear e-commerce platform. The addition of CloudFront CDN and ElastiCache further enhanced performance, creating a multi-layered caching strategy that significantly improves user experience.

Appendix: Future Enhancements

Planned future improvements include:

- Implementing AWS Backup for comprehensive backup strategy
- Implementing CI/CD pipeline for infrastructure changes
- Adding CloudWatch dashboards for monitoring
- Scaling ElastiCache cluster based on traffic patterns
- Implementing blue/green deployment strategies
- Creating disaster recovery plans with cross-region replication

Completed Enhancements

-  Implemented AWS Certificate Manager for certificate management
-  Enabled CloudFront for global content delivery
-  Implemented ElastiCache with W3 Total Cache for WordPress performance