AWS Architecture Design for LAMP

Design 1 assuming basic PHP web application

Assumptions

1. 2 tier system, web and database (if there is separate application-tier, need another set of EC2 instances and load balancer).
2. Stateless web-tier (if not, ELB needs to provide user session stickiness).
3. No large files uploaded by users (needs additional flow to S3).
4. No queries to database that can potentially take time to respond because of large dataset or concurrent requests (needs those database tables to be moved to DynamoDB).
5. No long running jobs for user requests (needs additional flow to backend-tier via SQS and also to SES if required to respond to users).

Control Flow Diagram

Diagram shows control flow in the distributed system. Flow to S3 and Glacier may be occasional, depending on archival strategy etc. Flow will temporarily change in case of disaster-recovery situation, for example if primary database is not available, web-tier and ElastiCache will connect to secondary. Number of instance in web-tier will also be changed by ELB based on load.
AWS Services Used

**EC2 Instances** to run web-tier. PIOPS enabled EBS for predictable IO performance and persistent drive. t2.small will be good to start with and can be changed later to more appropriate type after profiling or based on CloudWatch metrics.

**Auto Scaling** for high-availability by ensuring that minimum number of (1 to start with) healthy instances are always up. When high volumes occur, it can be used to scale up (and down) with volumes.

**Elastic Load Balancer** for multi-AZ load balancing of web-tier, fault-tolerance by ensuring traffic goes to healthy EC2 instances, high-availability by scaling up and down as load varies and SSL termination to offload encryption/decryption/ key-management from web-tier.

**CloudWatch** to provide instance health/performance metrics to Load Balancer and send notifications to ops team in case of potential problems.

**VPC** to isolate the resources from external network. Traffic to VPC should be allowed through gateways and bastion hosts only.

**RDS (MySQL/Aurora)** with PIOPS for hosted, predictable, scalable system. Multi-AZ deployment (physical, synchronous replication with primary and secondary instances in different AZs) for durability, high-availability and disaster-recovery. Read replicas can be added later, in case of high read volume. This is preferred over DynamoDB to support existing LAMP stack based app.

**S3** for durable persistence of static objects like deployment-archive and scripts, backups of database and logs, media files etc.

**ElastiCache**(redis) to cache data for better performance and to reduce database hits.

**CloudFront** for edge-caching of frequently used static files for faster delivery and reduced load.

**Glacier** for low-cost archival using S3 lifecycle rules to automate recurring archival of logs and other unused data.

**BeanStalk** or CloudFormation to manage resource lifecycle if developers handle the AWS resources and deployments. **OpsWorks** instead, if there are operations specialists. Both offer templates for PHP web application. BeanStalk deployments can also be managed using CloudFormation.

**Route 53** Geo DNS Service for multi-region scaling, when global traffic arrives.

**How this addresses the concerns listed in this assignment**

*scaling to meet the demand* – This architecture is scalable to high loads due to auto-scaling of web-tier and offloading of tasks to other services (e.g. CloudFront, ELB). For very large scale where database usage is high, database will need to be manually-partitioned or else migrated to a fast key-pair database that can auto-scale (DynamoDB or MongoDB).
lack of Disaster Recovery – DR is automated in this case because of multi-AZ web-tier and multi-AZ database, even if the one whole AZ goes down. Multi-AZ RDS provides same endpoints after failover, so that if web-tier is programmed to retry, it will connect without any change after the failover.

ability to configure their database for high performance – RDS is easily upgradable (to larger instances) and fully configurable as a MySQL database. Read-replicas are also managed by RDS if configured. ElastiCache helps a lot for read-heavy volumes.

user experience in browser very low latency – Cloud Front enable edge caching of static data (and even dynamic data if required) distribution where low latency is required. High PIOPS can reduce latency for EBS.

effective distribution of load – Handled by ELB in web-tier and PIOPS in RDS.

a self-healing infrastructure that recovers from failed service instances – ELB (with CloudWatch and AutoScaling) replaces unhealthy instances with new. Multi-AZ RDS does auto-healing for database.

security of data at rest and in transit – VPC isolates the resources from others’ network. RDS is secure at database level (MySQL provided security/authentication/authorization features) and network level (can be connected only by the clients given privilege explicitly using IAM). Data in transit can be transferred over SSL for another layer, if data is that sensitive.

securing access to the environment as the delivery team expands – Most restricted and individual IAM roles should be used for all users and MFA should be used for authentication when team grows. VPC should be used to isolate network and only bastion hosts should allow SSH.

an archival strategy for inactive objects greater than 6 months – S3 object lifecycle-rules to move logs and other data regularly to Glacier. Also a strategy to move old/unused database data to S3.

ability to easily manage and replicate multiple environments – Easy infra management can be achieved using BeanStalk PHP app or CloudFormation sample LAMP RDS template, customized for this application (change application-source, default types etc.) takes care of this. Replication can be done by extracting a CloudFront template from running infra and then using that to create replicas.

**Design 2 for 3 tier application**

**Change in Assumptions**

1. 3 tier system with web-tier and application-tier in separate stateless processes

**Change in Components**

Another Elastic Load Balancer for application-tier.

Another set of EC2 Instances to run application-tier processes.

CloudFormation to manage resource-lifecycle for developers, and not BeanStalk.
Design 3 for long running jobs

Change in Assumptions

1. Long running jobs to process data and respond to users. e.g. processing uploaded media or data that needs manual approval.

Change in Components

SQS to queue up the requests with reference to data to be processed. This provides loose-coupling and high level of fault-tolerance as SQS waits for acknowledgement before deleting request.


Another set of EC2 Instances to run backend -tier processes.

CloudFormation to manage resource lifecycle for developers, and not BeanStalk.