During meeting Team discussed **Migration Goals and Must-Haves: Azure to AWS Multi-Tenant Database**

Initial requirements:

1. **Tenant-Level Database Separation**   
   Each tenant’s data must reside in a dedicated database, ensuring strong isolation, security, and simplified compliance.
2. **Scalable Design for Variable Tenant Workloads**   
   The architecture should support tenants of varying sizes and usage patterns, allowing databases to scale independently based on demand.
3. **Centralized Compute with Dynamic Tenant Routing**   
   Shared compute resources (e.g., application servers, , containers) must be able to dynamically route requests to the appropriate tenant database based on request context.
4. **Automated Schema Management for All Tenants**   
   Efficient mechanisms are required to update, version, and migrate database schemas for both existing and newly onboarded tenants, with support for zero-downtime deployments.
5. **Robust Rollback and Recovery Mechanisms**   
   Ability to roll back schema changes or migrations in case of failures, ensuring data integrity and minimal service disruption.
6. **Automated Tenant Provisioning and Lifecycle Management**   
   Streamlined onboarding, offboarding, and resource management for tenant databases,
7. **Monitoring, Alerting, and Performance Management**   
   Integrated monitoring and alerting for tenant databases, enabling proactive management of health, performance, and capacity.
8. **Security and Compliance**   
   Implement strong access controls, encryption at rest and in transit, and auditing to meet regulatory and customer requirements.
9. **Cost Optimization and Resource Efficiency**   
   Design for efficient use of AWS resources, including database consolidation where appropriate, right sizing, and automated scaling to balance cost and performance.
10. **Documentation and Change Tracking**   
    Maintain clear documentation, version control, and auditing of schema changes and migrations for traceability and operational excellence.

Meeting notes from discussion

* **Tenant-Level Database Separation:** Dmytro, Mohammad, Sumit, Andrew, and Gopala Krishna Sriram discussed the migration of Unity databases to AWS, focusing on tenant-level database separation, AWS RDS instance limits, and the phased migration strategy, with Sumit and Sriram highlighting the need to document constraints and options for future scalability.
* **Tenant-Level Database Separation:** The team confirmed that the current design supports tenant-level database separation, and AWS migration should not impact this; however, Dmytro raised the AWS RDS instance hard limit of 100 databases, which could be a constraint given the potential for thousands of tenants.
* **AWS RDS Instance Limits:** Dmytro and Sriram explained that AWS RDS SQL Server has a hard limit of 100 databases per instance, which may not be sufficient for future scaling, and discussed Aurora PostgreSQL as an alternative for elastic scaling, though this would require additional migration steps.
* **Phased Migration Strategy:** Andrew and Sriram agreed to start with a lift-and-shift approach using RDS for SQL Server on AWS, with the possibility of migrating to Aurora PostgreSQL in a later phase, and Sumit requested documentation of the limits and constraints for each migration phase.
* **Application Impact of Database Choice:** Sumit asked whether switching to Postgres or Aurora would impact the application team, and Andrew clarified that while schema and stored procedure changes would be needed, application changes would be minimal due to the use of internal libraries for tenant mapping.
* **Security, Compliance, and Authentication Mechanisms:** Mohammad, Dmytro, Andrew, and Sriram reviewed security and compliance requirements for AWS migration, focusing on authentication mechanisms, backup and restore processes, and the need to consult the security team regarding SQL authentication.
* **Authentication Mechanisms:** The group discussed the lack of managed identity support for SQL Server on AWS, agreeing to use basic authentication initially and consider IAM authentication if migrating to Aurora or other supported databases; Sriram and Andrew noted the need to consult the security team about this approach.
* **Security Best Practices:** Mohammad emphasized that security best practices from Azure would need to be replicated on AWS, with logical data separation per tenant and similar network security controls, while Sriram confirmed that AWS provides native VPC, subnet, and routing table controls.
* **Backup and Restore Compliance:** Sriram and Mohammad confirmed that AWS RDS supports customizable backups and restore points, which are required for compliance, and these can be managed as part of the database-as-a-service model.
* **Scalability and Resource Management in AWS:** Mohammad, Andrew, Sriram, and Sumit discussed the limitations and strategies for scalable database design on AWS, including manual scaling, downtime considerations, and handling tenants of varying sizes.
* **Manual Scaling and Downtime:** The team noted that AWS RDS for SQL Server does not support auto-scaling, requiring manual vertical scaling with associated downtime, and discussed the use of RDS Proxy and multi-AZ configurations to minimize downtime at additional cost. RDS Proxy won’t be used it is not satisfied requirements fully.
* **Handling Variable Tenant Workloads:** Sumit highlighted the need to support tenants ranging from 100 to 150,000 devices, and requested documentation of strategies to handle this variety, including deployment options for on-premises or edge scenarios.
* **Resource Allocation and Monitoring:** Sriram explained that resource allocation must be pre-estimated and pre-allocated due to the lack of auto-scaling, and Cloudwatch alarms can be used for monitoring resource consumption, with auto-scaling available only for storage.
* **Automated Schema Management and Migration Processes:** Mohammad, Dmytro, Sumit, and Andrew reviewed the mechanisms for schema management and migration, confirming the use of DACPAC for SQL Server, discussing downtime, and outlining strategies for onboarding new tenants and handling schema upgrades.
* **Schema Migration Mechanisms:** The team agreed to continue using DACPAC-based in-place upgrades for schema management during the initial AWS migration, with custom versioning and upgrade scripts, and to consider alternatives like Liquibase/Flyway for future phases.
* **Downtime During Upgrades:** Mohammad clarified that current schema upgrades involve minimal downtime, not true zero downtime, and achieving zero downtime would require active-active or active-passive setups with failover.
* **Onboarding New Tenants:** For new tenants, the process involves cloning a template database with the latest schema, ensuring all new tenants start with the current version.
* **Rollback and Recovery Strategies:** Mohammad, Sumit, Andrew, and Sriram discussed rollback and recovery mechanisms for schema changes, agreeing to limit rollback to the immediate previous version using backups, and emphasizing the need to document these constraints.
* **Rollback Mechanism:** The team agreed that rollback would be supported only to the immediate previous version by restoring from backups taken before upgrades, as supporting rollback to older versions is impractical due to data changes.
* **Failure Handling During Upgrades:** Mohammad described the current behavior where if a schema upgrade fails for a tenant, only the affected tenant remains at the previous version, and the need to document whether all tenants should be rolled back in such cases.
* **Automated Tenant Provisioning and Lifecycle Management:** Mohammad, Viktor, and Sriram described the in-house tenant management tool used for onboarding and managing tenants, including database creation, Kafka topic setup, and service account provisioning, with minor modifications needed for AWS compatibility.
* **Tenant Management Tool Functionality:** The tenant management tool automates onboarding by creating required databases, Kafka topics, service accounts, and permissions, and is implemented in .NET Core for cross-platform compatibility.
* **Database User Management:** Viktor asked about user creation for application access, and Mohammad clarified that currently all applications use the same database user, managed via deployment scripts rather than the tenant management tool. DB User is created in DB deployment , not in Tenat Management tool
* **AWS Compatibility Adjustments:** Mohammad noted that only minor changes to SQL commands are needed in the tenant management tool to support AWS RDS, mainly due to differences in database creation syntax.
* **Monitoring, Alerting, and Performance Management:** Mohammad, Sriram, and Andrew discussed the use of Datadog and Cloudwatch for monitoring and alerting, covering integration options, custom metrics, and cost considerations for AWS migration.
* **Datadog Integration:** Datadog is used as the primary observability tool, with native integration for SQL Server metrics and the ability to extend with custom metrics; logs can be sent from Cloudwatch to Datadog if needed.
* **Alerting and Cost Management:** Sriram explained that alerting can be managed in Datadog or Cloudwatch, with cost implications for pushing large volumes of logs to Datadog, and suggested keeping alerts in Cloudwatch if cost is a concern.
* **Performance Dashboards:** Performance dashboards are maintained in Datadog, and the SQL Server query store feature is supported on both Azure and AWS for performance analysis.
* **Security, Auditing, and Access Control:** Mohammad, Dmytro, and Sriram reviewed security and auditing features available on AWS, confirming that SQL-level auditing and network security controls are supported, and discussed the use of IAM and service accounts.
* **Auditing Features:** SQL Server auditing is available by default on AWS RDS, and additional auditing via CloudTrail is not required for their use case.
* **Network and User Security:** AWS provides native support for VPC, subnets, and routing tables, and user/application security must be explicitly defined, similar to Azure.
* **Cost Optimization and Resource Efficiency:** Mohammad, Andrew, and Sriram discussed strategies for cost optimization on AWS, including right-sizing, manual scaling, and the limitations of auto-scaling for SQL Server.
* **Manual Scaling and Monitoring:** Cloudwatch alarms can be used to monitor resource consumption, and auto-scaling is available only for storage, with CPU and memory scaling requiring manual intervention and downtime.
* **Database Consolidation:** Multiple tenant databases can be consolidated into a single RDS instance, similar to the elastic pool model in Azure, but subject to AWS instance limits.
* **Documentation, Change Tracking, and Future Use Cases:** Mohammad, Andrew, and Sumit confirmed that documentation, version control, and auditing of schema changes are in place, and discussed future requirements for customer-managed encryption keys.
* **Documentation and Change Tracking:** The team maintains clear documentation and version control for schema changes and migrations, with no current use cases for change data capture or change tracking.
* **Customer-Managed Encryption Keys:** Sumit raised the future requirement for allowing customers to manage their own encryption keys, and Sriram confirmed that AWS RDS and Aurora support customer-managed keys, though implementation details need to be defined.

Follow-up tasks:

* **Database Migration Limits Documentation:** Document the limits and constraints for each migration phase option, including database limits and cost implications for different database choices.
* **Security Authentication Review:** Run the proposed use of SQL authentication for initial AWS migration past the security team for approval. (Andrew)