**Example RACI Matrix in SDD:**

| **Task/Deliverable** | **Responsible (R)** | **Accountable (A)** | **Consulted (C)** | **Informed (I)** |
| --- | --- | --- | --- | --- |
| Requirements Analysis | Business Analyst | Project Manager | Product Owner, Developers | Stakeholders |
| High-Level System Design | Solution Architect | Solution Architect | Development Team, QA Team | Stakeholders |
| Detailed Design Documentation | Developers | Lead Developer | Solution Architect, QA Team | Project Manager |
| Code Development | Developers | Lead Developer | QA Team | Project Manager, Client |
| Testing and QA | QA Team | QA Lead | Developers, Project Manager | Stakeholders |
| Deployment | DevOps Engineer | DevOps Lead | Developers, QA Team | Stakeholders, Client |

In the **Output Section** of an API or system, error codes are used to communicate the status or failure of an operation. These codes are part of the response sent back to the client, helping them understand what went wrong (if anything). Below is a list of **common error codes** (both HTTP status codes and custom application-specific error codes) that might appear in the output section of an API or system response:

**HTTP Status Codes (General Errors)**

1. **400 - Bad Request**
   * The server cannot process the request due to invalid syntax (e.g., missing or incorrect parameters).
2. **401 - Unauthorized**
   * The request lacks valid authentication credentials (e.g., missing or invalid API key/token).
3. **403 - Forbidden**
   * The server understands the request but refuses to authorize it (e.g., lack of permissions).
4. **404 - Not Found**
   * The requested resource could not be found on the server (e.g., an endpoint or resource does not exist).
5. **405 - Method Not Allowed**
   * The HTTP method used (e.g., GET, POST) is not supported by the resource.
6. **406 - Not Acceptable**
   * The server cannot generate a response that is acceptable based on the Accept headers provided by the client.
7. **408 - Request Timeout**
   * The server timed out waiting for the request (e.g., due to network issues or server delays).
8. **409 - Conflict**
   * The request could not be processed because of a conflict with the current state of the resource (e.g., trying to create a duplicate entry).
9. **413 - Payload Too Large**
   * The request is too large for the server to process (e.g., file uploads that exceed size limits).
10. **415 - Unsupported Media Type**
    * The media type of the request is not supported by the server (e.g., wrong Content-Type in the request).
11. **429 - Too Many Requests**
    * The user has sent too many requests in a given amount of time (rate limiting).
12. **500 - Internal Server Error**
    * A generic error message indicating that an unexpected condition was encountered on the server side.
13. **502 - Bad Gateway**
    * The server, acting as a gateway or proxy, received an invalid response from an upstream server.
14. **503 - Service Unavailable**
    * The server is temporarily unable to handle the request (e.g., due to maintenance or overload).
15. **504 - Gateway Timeout**
    * The server, acting as a gateway or proxy, timed out while waiting for an upstream server's response.

**Custom Application-Specific Error Codes**

These error codes might be defined by the application or API to provide more granular feedback on errors:

1. **1001 - Invalid Input**
   * The input data provided in the request is invalid or incomplete (e.g., a required field is missing).
2. **1002 - Invalid Authentication**
   * The provided authentication credentials are incorrect or expired (e.g., invalid API token).
3. **2001 - Resource Not Found**
   * A specific resource or record could not be found (e.g., the user or product does not exist in the database).
4. **3001 - Resource Conflict**
   * There is a conflict while creating or updating a resource (e.g., trying to create a record with a unique constraint violation).
5. **4001 - Insufficient Permissions**
   * The user does not have the required permissions to access the resource or perform the action.
6. **5001 - Database Error**
   * A failure occurred during interaction with the database (e.g., a query execution failure).
7. **6001 - External Service Failure**
   * A third-party service or external system failed to respond or returned an error (e.g., failure to call an external API).
8. **7001 - Data Validation Error**
   * The data provided does not pass validation rules (e.g., incorrect format or value out of bounds).
9. **8001 - Timeout Error**
   * The operation timed out (e.g., a database query or external service request took too long).
10. **9001 - Internal Application Error**
    * An unexpected internal application error occurred that prevented the request from being processed correctly.

In the Deployment Section under Technical Specifications, the goal is to outline the processes, tools, and strategies involved in deploying the application, service, or system to a production environment. It ensures that the deployment is carried out efficiently, securely, and with minimal disruption. Here are the key areas of knowledge that should be covered in this section:

1. Deployment Strategy

Continuous Integration (CI) / Continuous Deployment (CD):

Automation of code integration, testing, and deployment processes.

Tools: Jenkins, GitLab CI, CircleCI, Travis CI.

Deployment Models:

Blue/Green Deployment: Reduces downtime by having two identical environments (one active, one inactive) and switching between them.

Canary Deployment: Gradual rollout of new versions to a small subset of users before full deployment.

Rolling Deployment: Sequential replacement of old versions with new ones, ensuring high availability.

Phased Deployment: Deploying in stages to manage risk and issues progressively.

Zero-Downtime Deployment: Ensuring that the system remains operational during the entire deployment process.

2. Environment Configuration

Infrastructure as Code (IaC):

Using tools to manage infrastructure (servers, networks) through code (e.g., Terraform, CloudFormation).

Environment Variables:

Defining and managing environment-specific variables (e.g., API keys, database connection strings).

Environment Parity:

Ensuring consistency between development, staging, and production environments (e.g., using Docker or Kubernetes).

3. Dependency Management

Library/Package Dependencies:

Managing external libraries, packages, or modules needed by the system (e.g., npm for JavaScript, pip for Python).

Version Control:

Maintaining version control for dependencies to avoid conflicts and ensure compatibility (e.g., dependency lock files).

Containerization:

Using containers (e.g., Docker) to package the application and its dependencies for consistent deployment across environments.

Microservices:

Managing the deployment of multiple microservices and their interactions, including service discovery and inter-service communication.

4. Deployment Automation

Automated Deployment Tools:

Tools like Jenkins, GitLab CI/CD, Bamboo, and Ansible to automate deployment pipelines.

Scripted Deployments:

Writing scripts (e.g., Bash, PowerShell, Python) for manual or semi-automated deployment steps.

Rollbacks and Retries:

Automated rollback mechanisms to revert deployments in case of failure.

Post-Deployment Scripts:

Running automated post-deployment checks, data migrations, or updates after the deployment is completed.

5. Release Management

Versioning:

Semantic versioning (e.g., v1.2.3) to clearly communicate changes and updates in the system.

Release Notes:

Documenting new features, bug fixes, and known issues with each release.

Changelog:

Maintaining a changelog that tracks every change in each release.

Tagging and Branching:

Using Git tags/branches to mark release points in the codebase.

Deployment Window:

Defining deployment windows or maintenance windows for deployments, especially for critical systems.

6. Rollback and Recovery Strategy

Backup and Restore:

Backing up data and configurations before deployment to ensure recovery in case of failure.

Database Rollback:

Managing database migrations and ensuring rollback mechanisms for schema changes or data updates.

Deployment Rollback Plan:

Procedures to revert to a previous stable version in case of deployment failure, including both code and data rollbacks.

7. Testing and Validation

Pre-Deployment Testing:

Ensuring tests are executed before deployment (unit tests, integration tests, user acceptance tests, etc.).

Smoke Testing:

Running basic functionality tests immediately after deployment to verify the application is working.

Post-Deployment Testing:

Running tests after deployment to ensure that the application functions as expected in production.

Performance Testing:

Ensuring that the system performs well under expected load after deployment.

Canary Testing:

Deploying to a subset of users and monitoring their experience before fully rolling out the release.

8. Monitoring and Logging

Real-Time Monitoring:

Setting up application and infrastructure monitoring (e.g., New Relic, Prometheus, Grafana) to observe system performance post-deployment.

Error Logging:

Implementing error logging (e.g., Sentry, Loggly) to capture issues post-deployment and facilitate debugging.

Alerting:

Configuring alerts (e.g., PagerDuty, Opsgenie) for critical failures during or after deployment.

9. Security Considerations

Access Control:

Managing user roles and permissions during deployment (e.g., limiting access to sensitive deployment stages).

Encryption:

Ensuring sensitive data, both in transit and at rest, is encrypted during deployment.

Secrets Management:

Using tools like HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault to manage secrets and credentials securely.

Vulnerability Scanning:

Scanning for security vulnerabilities before deployment (e.g., dependency vulnerabilities, container security).

Compliance:

Ensuring that deployments meet regulatory compliance requirements (e.g., GDPR, SOC 2, HIPAA).

10. Deployment Logs and Documentation

Deployment Logs:

Maintaining detailed logs of each deployment, including timestamps, success/failure statuses, and reasons for failure.

Documentation:

Documenting the entire deployment process, configurations, tools, and rollback procedures for future reference.

11. Cloud and Infrastructure Management

Cloud Deployment:

Deploying applications on cloud platforms (e.g., AWS, Azure, Google Cloud) using specific tools (e.g., AWS Elastic Beanstalk, Google Kubernetes Engine).

Infrastructure Provisioning:

Using cloud provisioning tools to manage and scale infrastructure based on demand.

Auto-Scaling:

Configuring auto-scaling policies for cloud-based services to handle variable load.

12. Post-Deployment Support

Post-Deployment Monitoring:

Monitoring the system post-deployment to detect issues early.

User Feedback:

Gathering feedback from end users or stakeholders to validate the deployment's success.

Continuous Improvement:

Using metrics and feedback to improve future deployment processes.

For a **Support Document** related to the **Scalability Section** under **Technical Specifications**, the goal is to provide the support team with the necessary information to manage and troubleshoot scalability issues in production. It should outline how the system should be scaled, the monitoring required, and the steps for handling any performance or scalability issues. Here are the **key areas of knowledge** in the **Scalability** section for a support document:

**1. Horizontal and Vertical Scaling**

* **Horizontal Scaling**:
  + Instructions for adding or removing instances (servers, containers) to handle increased load.
  + Load balancing setup and troubleshooting (e.g., configuring, testing, and troubleshooting load balancer performance).
* **Vertical Scaling**:
  + Procedures for scaling individual instances by adding more CPU, RAM, or storage.
  + Identifying when vertical scaling is needed and how to increase resources without downtime.

**2. Auto-Scaling Configuration and Troubleshooting**

* **Auto-Scaling Setup**:
  + Guidelines on configuring auto-scaling policies based on metrics such as CPU usage, memory usage, or request count.
  + Information on adjusting auto-scaling triggers to align with system requirements.
* **Troubleshooting Auto-Scaling**:
  + Identifying common issues with auto-scaling, such as over-scaling or under-scaling.
  + Steps to debug and resolve failed auto-scaling events, including log analysis.

**3. Load Balancer Management**

* **Load Balancing Techniques**:
  + Understanding of how the load balancing works (e.g., round-robin, least connections) and configuration management.
* **Monitoring Load Balancers**:
  + How to monitor load balancer performance, identify bottlenecks, and ensure even distribution of traffic.
* **Failover and High Availability**:
  + Instructions for ensuring that load balancers are set up for high availability to prevent a single point of failure.

**4. Caching Strategies**

* **Cache Management**:
  + Overview of caching systems in place (e.g., Redis, Memcached) and procedures for scaling them.
* **Cache Expiration and Invalidation**:
  + Steps to manage cache expiry and invalidation during scaling or after an update.
* **Troubleshooting Caching Issues**:
  + Identifying and resolving cache-related issues that can affect scalability (e.g., cache misses, stale data).

**5. Database Scaling and Management**

* **Read/Write Splitting**:
  + Guidelines for managing read replicas and ensuring load is distributed between primary and replica databases.
* **Sharding and Partitioning**:
  + Instructions on managing database sharding and partitioning for scalability.
* **Database Performance Tuning**:
  + Key indicators to monitor for database performance and methods for scaling databases horizontally or vertically.
* **Replication and Failover**:
  + Setting up and troubleshooting database replication for high availability and fault tolerance.

**6. Monitoring and Alerts**

* **Performance Monitoring**:
  + List of performance metrics to monitor (e.g., CPU usage, memory, response time, request throughput) and tools for monitoring.
  + Recommendations for setting up alerts based on thresholds to proactively manage scalability.
* **Scaling Indicators**:
  + Information on scaling indicators and when to initiate scaling actions based on performance data.

**7. Service Discovery in Microservices**

* **Service Discovery Configuration**:
  + Steps to configure and maintain service discovery mechanisms (e.g., Consul, Eureka) to support scaling of microservices.
* **Troubleshooting Service Discovery**:
  + Identifying and resolving issues where services might not be properly discovered or are scaling inconsistently.

**8. Asynchronous Processing and Queues**

* **Queue Management**:
  + Overview of message queues (e.g., RabbitMQ, Kafka) used for asynchronous processing and scaling their capacity as needed.
* **Job Scheduling and Scaling**:
  + Information on managing background job processing systems (e.g., Celery) to scale the number of workers or queue lengths.

**9. Traffic Management and Rate Limiting**

* **Traffic Throttling**:
  + Instructions for rate limiting to prevent overload during high traffic periods.
* **Backpressure Management**:
  + Understanding backpressure techniques and how to scale components to handle high traffic loads.
* **API Rate Limiting**:
  + Steps to implement or adjust API rate limiting to manage incoming traffic effectively.

**10. Cloud and Infrastructure Scalability**

* **Cloud Auto-Scaling**:
  + Guidelines for managing cloud scaling features (e.g., AWS Auto Scaling, Azure Scale Sets) and ensuring smooth scaling operations.
* **Cost Management During Scaling**:
  + Tips for managing cloud costs during scaling and preventing over-provisioning or inefficient resource usage.
* **Virtual Machine and Container Scaling**:
  + Overview of VM or container-based scaling, including using orchestration tools like Kubernetes to manage scaling of pods.

**11. Backup and Data Recovery**

* **Scaling Backup Solutions**:
  + Ensuring that backup strategies are scalable and that backups are taken without impacting system performance during scaling.
* **Data Recovery Plans**:
  + Instructions for recovering from failures in scaled environments, ensuring data consistency and availability.

**12. High Availability and Fault Tolerance**

* **Redundancy Setup**:
  + Steps to ensure high availability through redundancy strategies (e.g., clustering, multi-region setups).
* **Failover Management**:
  + Configuration of failover systems for scalability, including cloud and database failover procedures.

**13. Stress Testing and Load Testing**

* **Testing for Scalability**:
  + Guidelines on conducting load testing and stress testing to validate the scalability of the system under increased demand.
* **Identifying Bottlenecks**:
  + How to run tests to identify scalability bottlenecks and areas that need optimization for better performance.

**14. Documentation and Support Resources**

* **Scaling Documentation**:
  + Maintain documentation on the scaling architecture and any custom scaling features or configurations.
* **Troubleshooting Guides**:
  + Providing support team with common issues and resolutions related to scaling, such as bottlenecks, lagging systems, and server overload.